ТРУДЫ

ИНСТИТУТА СИСТЕМНОГО ПРОГРАММИРОВАНИЯ РАН

PROCEEDINGS OF THE INSTITUTE FOR SYSTEM PROGRAMMING OF THE RAS

ISSN Print 2079-8156
Том 31 Выпуск 3

ISSN Online 2220-6426
Volume 31 Issue 3

Институт системного
программирования
им. В.П. Иванникова РАН

Москва, 2019

ИСП РАН

ISSN 2079-8156 (Print), ISSN 2220-6426 (Online)

Труды Института системного программирования РАН
Proceedings of the Institute for System Programming of the RAS

Труды ИСП РАН – это издание с двойной анонимной системой рецензирования, публикующее научные статьи, относящиеся ко всем областях системного программирования, технологий программирования и вычислительной техники. Целью издания является формирование научно-информационной среды в этих областях путем публикации высококачественных статей в открытом доступе.

Издание предназначено для исследователей, студентов и аспирантов, а также практиков. Оно охватывает широкий спектр тем, включая, в частности, следующие:

- операционные системы;
- компиляторные технологии;
- базы данных и информационные системы;
- параллельные и распределенные системы;
- автоматизированная разработка программ;
- верификация, валидация и тестирование;
- статический и динамический анализ;
- защита и обеспечение безопасности ПО;
- компьютерные алгоритмы;
- искусственный интеллект.

Журнал издается по одному тому в год, шесть выпусков в каждом томе.

Поддерживается открытый доступ к содержанию издания, обеспечивая доступность результатов исследований для общественности и поддерживая глобальный обмен знаниями.

Труды ИСП РАН реферируются и/или индексируются в:

Proceedings of ISP RAS are a double-blind peer-reviewed journal publishing scientific articles in the areas of system programming, software engineering, and computer science. The journal's goal is to develop a respected network of knowledge in the mentioned above areas by publishing high quality articles on open access.

The journal is intended for researchers, students, and practitioners. It covers a wide variety of topics including (but not limited to):

- Operating Systems.
- Compiler Technology.
- Databases and Information Systems.
- Parallel and Distributed Systems.
- Software Engineering.
- Software Modeling and Design Tools.
- Verification, Validation, and Testing.
- Static and Dynamic Analysis.
- Software Safety and Security.
- Computer Algorithms.
- Artificial Intelligence.

The journal is published one volume per year, six issues in each volume.

Open access to the journal content allows to provide public access to the research results and to support global exchange of knowledge. **Proceedings of ISP RAS** is abstracted and/or indexed in:

УДК004.45

Адрес: 109004, г. Москва, ул. А. Солженицына, дом 25.
Телефон: +7(495) 912-44-25
E-mail: info-isp@ispras.ru
Сайт: http://www.ispras.ru/proceedings/

Address: 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
Tel: +7(495) 912-44-25
E-mail: info-isp@ispras.ru
Web: http://www.ispras.ru/en/proceedings

С о д е р ж а н и е

Table of Contents

DOI: 10.15514/ISPRAS-2019-31(3)-1

Tolerant parsing using modified LR(1) and LL(1) algorithms with embedded “Any” symbol

A.V. Goloveshkin, ORCID: 0000-0001-6947-0594 <alexeyvale@gmail.com>
Vorovich Institute for Mathematics, Mechanics and Computer Science,
Southern Federal University,
8a, Milchakova st., Rostov-on-Don, 344090, Russia

Abstract. Tolerant parsing is a form of syntax analysis aimed at capturing the structure of certain points of interest presented in a source code. While these points should be well-described in a tolerant grammar of the language, other parts of the program are allowed to be described coarse-grained, thereby parser remains tolerant to the possible variations of the irrelevant area. Island grammars are one of the basic tolerant parsing techniques. “Islands” term is used as the relevant code alias, the irrelevant code is called “water”. Efforts required to write water rules are supposed to be as small as possible. Previously, we extended island grammars theory and introduced a novel formal concept of a simplified grammar based on the idea of eliminating water description by replacing it with a special “Any” symbol. To work with this concept, a standard LL(1) parsing algorithm was modified and LanD parser generator was developed. In the paper, “Any”-based modification is described for LR(1) parsing algorithm. In comparison with LL(1) tolerant grammars, LR(1) tolerant grammars are easier to develop and explore due to solid island rules. Supplementary “Any” processing techniques are introduced to make this symbol easier to use while staying in the boundaries of the given simplified grammar definition. Specific error recovery algorithms are presented both for LL and LR tolerant parsing. They allow one to further minimize the number and complexity of water rules and make tolerant grammars extendible. In the experiments section, results of a large-scale LL and LR tolerant parsers testing on the basis of 9 open-source project repositories are presented.

Keywords: tolerant parsing; robust parsing; lightweight parsing; partial parsing; island grammars; simplified grammar; LanD parser generator

For citation: Goloveshkin A.V. Tolerant parsing using modified LR(1) and LL(1) algorithms with embedded “Any” symbol. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 3, 2019. pp. 7-28. DOI: 10.15514/ISPRAS-2019-31(3)-1

Толерантный синтаксический анализ с использованием модифицированных алгоритмов LL(1) и LR(1) со встроенной обработкой символа «Any»

А.В. Головешкин, ORCID: 0000-0001-6947-0594 <alexeyvale@gmail.com>
Институт математики, механики и компьютерных наук им. И.И. Воровича,
Южный федеральный университет
344090, Россия, г. Ростов-на-Дону, ул. Мильчакова, д. 8а

Аннотация. Толерантный синтаксический анализ используется для разбора структуры областей программы, представляющих интерес в контексте определённой задачи. В то время как эти области должны быть подробно описаны в толерантной грамматике языка, описание остальных частей программы может быть менее детальным, в результате парсер толерантен по отношению к возможным вариациям нерелевантных областей. Островные грамматики — один из основных способов реализации толерантного парсинга. Термином «остров» обозначаются релевантные области

кода, нерелевантный код обозначается термином «вода». Предполагается, что на написание водных правил грамматики должно тратиться как можно меньше усилий. Ранее автором настоящей работы была введена формальная концепция упрощённой грамматики, расширяющая теорию островных грамматик. Данная концепция основана на идее устранения описаний воды в грамматике путём замены их на специальный символ «Any». Для работы с упрощёнными грамматиками был модифицирован стандартный LL(1) алгоритм синтаксического анализа и разработан генератор толерантных парсеров LanD. В настоящей статье модификация, встраивающая обработку «Any», описывается для LR(1) алгоритма синтаксического анализа. В сравнении с толерантными LL(1) грамматиками, толерантные LR(1) грамматики являются более простыми для разработки и исследования ввиду того, что в них каждый остров может быть описан одним непрерывным правилом. Предложены дополнительные механизмы обработки символа «Any», приводящие ряд интуитивно корректных сценариев его использования в соответствие с формальным определением упрощённой грамматики. Для LL и LR толерантного синтаксического анализа описаны специфические механизмы восстановления от ошибок, позволяющие ещё больше сократить количество водных правил, понизить их сложность и сделать толерантную грамматику расширяемой. В разделе экспериментов представлены результаты крупномасштабного тестирования толерантных LL и LR парсеров на 9 репозиториях крупных проектов с открытым исходным кодом.

Ключевые слова: толерантный парсинг; устойчивый парсинг; легковесный парсинг; частичный парсинг; островная грамматика; упрощённая грамматика; генератор синтаксических анализаторов LanD

Для цитирования: Головешкин А.В. Толерантный синтаксический анализ с использованием модифицированных алгоритмов LL(1) и LR(1) со встроенной обработкой символа «Any». Труды ИСП РАН, том 31, вып. 3, 2019 г., стр. 7-28 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(3)-1

1. Introduction

Tolerant parsing is a syntax analysis technique differing from the detailed whole-language (so-called baseline) parsing. The latter is performed by a full-featured compiler of a certain programming language to ensure the program satisfies the grammar and to prepare an internal program representation for some further steps. Tolerant parsing performs deep structural analysis only on certain parts of the program, passing other parts with minimal effort. It is achieved by generating the corresponding parser from a tolerant grammar, where these parts of interest are described in details and some minimal description of the irrelevant area is provided. From developer's perspective, tolerant parsing allows her to focus on the structure of the points valuable in the context of a current task, without worrying about irrelevant code variations. Among tolerant parsing use cases, the following ones are the most frequently mentioned:

- **Baseline grammar inaccessibility**: Full version of the language grammar can be inaccessible due to proprietary issues or manual baseline parser writing [1]. Besides, physical accessibility does not assume accessibility in terms of grammar comprehension. Baseline grammar usage requires intensive exploration to detect rules describing constructs of interest. Tolerant grammar structure and mapping between its entities and language constructs are transparent to the developer, as she writes it according to her own knowledge of the task and the language.
- **Language embedding**: Some program artifacts assume the usage of multiple languages in one source file. In this case, a parser for the relevant language must be tolerant to all the snippets written in other languages [2].
- **Domain-specific idioms**: In a certain project, some local domain-specific patterns can be applied [1, 3]. They represent a high-level abstraction layer which is not presented in the language syntax and obviously is out of scope of the whole-language parser. Nevertheless, tolerant parsers can be strictly focused on these patterns, ignoring the underlying structure.

According to the *island grammars* tolerant parsing paradigm [1, 3], parts of the program that are well-described in the grammar are called *islands*, others are called *water*. Detailed grammar rules

describing islands are named *patterns*, water is presented with as few liberal productions as possible. However, sometimes it is required to describe some water parts in a fine-grained island-like style to avoid confusion with proper islands. Water parts mistaken for islands are called *false positives*, well-structured water productions are called *antipatterns*. Island grammar development is always a finite iterative process consisting of in-the-wild parser testing and subsequent patterns and antipatterns refinement. Besides, some situations, when program entity can be treated as an island and as a water at the same time, are typically solved with generalized parsing algorithms [4, 5].

The author of the current paper is interested in tolerant parsing because of the long-term goal to develop a multi-language tool for concern-based markup of software projects. Talking about a program as a set of functionalities, so-called *concerns*, we may notice that many of them are implemented with pieces of code which are spread across solid program elements, such as classes or methods [6, 7]. These concerns are called *vertical layers* [8] or *crosscutting concerns* [9]. To work with this kind of concerns, it is vital to create and manipulate some meta-information about their location, this information should be sustainable with respect to code changes, so it cannot rely on text line and text column numbers. Abstract syntax tree is considered to be a more appropriate structure for meta-information binding, so, there must be a set of parsers for different languages, these parsers must build abstract syntax trees in one unified format. These trees should capture only the structure of program entities we plan to bind to, therefore, tolerant parsing is an option. It also should be easy to support new languages by developing additional grammars and generating tolerant parsers. Previously, to meet the requirements for parsers and trees, we developed a tolerant parser generator called LanD. It uses a modified LL(1) parsing algorithm which is theoretically and experimentally proved to be correct [10].

The contributions of this paper are: 1) a modified LR(1) parsing algorithm with incorporated notion of a special `Any` token allowing parser to match implicitly defined token sequences; 2) supplementary `Any` processing techniques for modified LL(1) and LR(1) parsing algorithms, filling the gap between the simplified grammar formal definition and real tolerant parsing use cases; 3) specific `Any`-based LL and LR error recovery mechanisms aimed at elimination of water rules and correct handling of possible ambiguities without parsing algorithm generalization; complexity analysis is also carried out; 4) lightweight LL(1) and LR(1) grammars for a broad range of languages, namely, for C#, Java, PascalABC.NET programming languages, Yacc and Lex specification formats, XML and Markdown markup languages; 5) an experimental evidence of the applicability of the generated tolerant parsers for large-scale software projects analysis.

The remainder of the paper is organized as follows. In Section 2, main goals of the current research are listed. A brief overview of the previous author's research, along with closest analogues analysis, is provided in Section 3. In Section 4, a modification of the standard LR(1) parsing algorithm aimed at `Any` symbol processing is introduced, `Any` implementation improvements and issues addressed are discussed, novel `Any`-based error recovery algorithms are described. Section 5 includes a sufficient volume of experimental data obtained by applying generated tolerant parsers for C# and Java languages to a number of real-world software repositories. In Section 6, a brief summary of the contribution of the paper is provided along with future work outlining.

2. Problem statement

The first assumption of the current research is that the concept of `Any`, previously successfully embedded into a top-down parsing, can be embedded in a bottom-up parsing too, making tolerant grammars more expressive and easy-to-write. The second assumption is that ambiguities originated in islands and water similarity can be resolved not only by adding special antipatterns or by generalized algorithms usage, but also by a special recovery mechanism embedded in a deterministic parsing.

The key goals of the current research are:

1) to design an LR(1) parsing algorithm with built-in notion of a special `Any` grammar symbol that provides skipping the token sequences that are not explicitly described in the grammar;
2) to introduce into the LanD parser generator additional capabilities for correct `Any` processing in case `Any` usage does not fully satisfy simplified grammar formalization;
3) to design specific error recovery mechanisms for LL(1) and LR(1) tolerant parsing, aimed at handling ambiguities originating in water and island similarity;
4) to implement tolerant island grammars for a broad range of languages;
5) to evaluate parser's applicability through the analysis of large-scale software projects written in C# and Java languages.

3. Related work

3.1 «Any» implementation

The concept of `Any` symbol is implemented in several parser generators. Historically, the first tool with embedded capability to match tokens from sets which are not directly specified in a grammar is the Coco/R recursive-descent parsers generator. According to the documentation [11, p. 14], developer can use a special symbol `ANY`, which *denotes any token that is not an alternative to that* `ANY` *symbol in the current production*. A set of admissible tokens for the position of a particular `ANY` is precomputed to make the situation when parser has to make a choice between `ANY` and some explicitly specified token unambiguously solvable in favor of the explicit option. As shown in [10], these precomputed sets are both incomplete due to the lack of nonterminal outer context analysis and excessively restrictive due to a single restriction applied to all the elements of the sequence corresponding to the iteration of `ANY`. As a result, there are grammars for which parsers generated by Coco/R do not parse programs valid from the developer's point of view. For example, a parser generated by the grammar

```
A = a b c | {ANY} d.
```

is not capable to recognize the input string `bad$` (`$` denotes the end of the input, `{ANY}` denotes zero or more `ANY` tokens).

Similar `Any` implementation is built into a tool for lightweight LALR(1) parser development, called LightParse [12]. LightParse grammar is not directly used to generate a parser. Instead, it is transformed to the YACC-like format supported by the standard LALR(1) parser generator GPPG. In the transformed grammar, every entry of `Any` symbol is presented as a separate rule with single-element alternatives, by an alternative for each of the admissible terminal symbols. To ensure these rules are valid in terms of GPPG, LightParse imposes additional restrictions on `Any` usage. It only deepens drawbacks inherited from Coco/R.

The most recent `Any` token implementation is introduced by the author of the current paper for LanD parser generator [10] aimed at LL(1) tolerant parsers generation by island grammars. In terms of the island grammars paradigm, `Any` symbol allows one not to specify the particular content of the water area, writing `Any` instead. Unlike `ANY` symbol in Coco/R, our `Any` corresponds to a sequence of zero or more tokens, not a single token. In its implementation, all the known shortcomings are eliminated. The decision about the current token's admissibility at `Any` position is made dynamically at the parsing stage and restricts the set of admissible tokens no more than necessary to avoid ambiguities. LanD's `Any` implementation does not assume the grammar translation to the form suitable for the standard parsing algorithm. Instead, the standard LL(1) algorithm is modified to integrate the notion of `Any` and make it possible to define admissible tokens by the content of a parsing stack.

In the current paper, LanD parser generator is extended with the capability to generate LR(1) parsers with embedded `Any` support.

3.2 Formal definition of a simplified grammar

In [10], through the `Any` token, we formulate a formal concept of the *simplified grammar*. We denote by $\mathrm{lhs}(p)$ and $\mathrm{rhs}(p)$, respectively, the left and the right part of the production p. Notation $x \in \mathrm{rhs}(p)$ for $x \in N \cup T$ means that $\mathrm{rhs}(p) = \alpha_1 x \alpha_2$, where $\alpha_1 \in (N \cup T)^*, \alpha_2 \in (N \cup T)^*$. $\mathrm{SYMBOLS}(\gamma)$ is used for the set of terminal symbols needed to compose all the $\omega : \gamma \overset{*}{\Rightarrow} \omega, \gamma \in (N \cup T)^*, \omega \in T^*$.

Definition 1. Let $G = (N, T, P, S)$ be a context-free grammar, $Any \notin T$. The grammar *simplified* with respect to G is a grammar $G_s = (N_s, T_s, P_s, S_s)$ defined as follows:

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

Intuitively, P_s contains productions for the start symbol of G_s and productions for all the nonterminals which are reachable from the start symbol. The definition of the mapping f means that some of the strings generated by G contain substrings which can be replaced with `Any`, then we obtain strings generated by G_s. Symbol `Any` can be written instead of the parts denoted by γ_i in production's right-hand side, in case these parts satisfy the criterion 2c of the definition 1. Verification of this criterion is possible only when solving a direct problem: when the grammar G_s is created on the basis of some available G. In theory, G can correspond to the baseline language grammar, as well as be a more tolerant version of the baseline grammar, containing all the anti-patterns described explicitly. In practice, it is usually not available or does not exist, so direct problem is rarely considered. Writing an island grammar for a certain programming language is equivalent to solving an inverse problem. Developer writes an initial approximation in the form of a simplified grammar in which `Any` usage allows one to minimize the efforts to describe a possible water content. Then she performs an iterative refinement in accordance with parsing results, making the grammar more and more corresponding to the language generated by some baseline.

Compliance with the criterion 2c is crucial for correct `Any` processing. At the same time, it is hard to maintain while solving an inverse problem. In this paper, additional `Any` processing mechanisms are offered. They allow grammar developers to weaken the control over the consistency with the formalization.

3.3 LL(1) parsing algorithm modification

In fig. 1, modified algorithms from [10] are rewritten in the form more suitable for further discussion. The delta between the standard algorithms and the modified ones is highlighted with grey. As shown in fig. 1a, when no action can be performed with a current token, parser tries to interpret this token as the beginning of a sequence corresponding to `Any`. `FIRST'` set, a modified version of a standard `FIRST`, is computed for the parsing stack content to get all the tokens that are explicitly allowed in the current place. This non-static approach is inspired in some sense by full-LL(1) parsing [13, pp. 247–251]. Set construction routine is shown in fig. 1c. A modification is needed to handle the consecutive `Any` problem defined in [10], this problem is explained in detail

in Section 4.2.1 along with a more general solution. `M` denotes a parsing table, `Stack` denotes a symbol stack which stores not just the symbols that are expected to be matched, but nodes of the syntax tree being constructed.

There are grounds for an analogy between the LL(1) parsing modification given and well-known error recovery algorithms: `Any` symbol looks similar to the `error` token denoting place in the grammar where recovered parsing can be resumed, `FIRST'` set seems like the set of synchronization tokens. Moreover, speaking in terms of the formal definition, a tolerant parser is built by a simplified grammar G_s, and a program from $L(G)$ is actually needed to be parsed. In terms of G_s, this program is erroneous.

However, here also lies a fundamental difference between `Any` processing and error recovery. Recovery is performed for a program which is incorrect regarding to a baseline grammar G. While success is not guaranteed, the main goal is to resume parsing at any cost, including the loss of some significant results of the previous analysis and skipping a significant part of the input stream, possibly containing some points of interest. The goal of `Any` processing is to translate a presumably valid $L(G)$ program into the language $L(G_s)$ by replacing some token sequences with `Any`. The premise that the program under consideration is correct with respect to G, in conjunction with the observance of the criterion 2c, makes input tokens skipping totally predictable. One can be sure that the parts of the input stream replaced with `Any` belongs to the water and can be discarded without loss of the land. Furthermore, predictable and correct replacement with `Any` is possible for a program that is incorrect with respect to G, in case incorrectness is located in a water area.

4. Algorithms and modifications

4.1 LR(1) parsing

Though the modified LL(1) parsing algorithm described in Section 3.3 is enough to create reliable tolerant parsers, describing a real programming language with LL(1) grammar is a challenge even when this grammar is supposed to be lightweight and tolerant. Constructs of interest, such as class members, usually have a common beginning up to a certain point, so they cannot be presented as solid alternatives for a single nonterminal symbol in LL(1). Instead, we have to write rule sequences in the style of taking the common factor out of the brackets and making a separate rule for a tail:

```
entity = attribute* keyword* (class_tail | member_tail)
member_tail = type name (method_tail | property_tail)
method_tail = arguments Any (init? ';' | block)
```

As a result, the grammar structure is not transparent enough for a newcomer because the connection between existing island rules written in such a distributed manner and particular language constructs is non-obvious.

This LL(1) limitation can be overcome through switching to a more complex LR(1) parsing. A modification of the standard LR(1) algorithm is shown in fig. 2a, modified areas are highlighted with gray. Like in a standard case, two stacks exist to keep parser state. `SymbolsStack` keeps the current viable prefix [14, p. 256]. In fact, similar to LL(1) `Stack`, in our implementation, it keeps not just symbols but nodes for a tree to be build. `StatesStack` keeps the indices of the states parser passed through to obtain the current viable prefix. An element `ACTIONS[s, t]` of the `ACTIONS` table keeps the knowledge of what action should be performed by the parser if token `t` is met while `s` is the parser's current state. There are two basic types of action in LR algorithm: `Shift` and `Reduce`, they are shown in fig. 2c. `GOTO[s, X]` contains the index of a state to which parser must go from `s` state after reducing some part of a viable prefix to `X`.

```
Stack := [];
Stack.Push(new Node($));
Stack.Push(new Node(S));

X := Stack.Peek().Symbol;
t := Lexer.NextToken();
while (X ≠ $) do
  if (t = ERROR_TOKEN) then
    return false;
  end if;
  if (X = t) then
    if (t = Any) then
      t := SkipAny(true);
    else
      Stack.Pop();
      t := Lexer.NextToken();
    end if;
  elif (M[X,t] ≠ null) then
    if (t = Any) then
      if (Any ∈ FIRST'(Stack)) then
        t := SkipAny(true);
      else
        t := Error(null);
      end if;
    else
      Apply(M[X,t]);
    end if;
  elif (t = Any) then
    t := Error(null);
  else
    t := Any;
  end if;
  X := Stack.Peek().Symbol;
end while;

if (t = $) then
  Accept();
  return true;
else
  return false;
end if;
```

(a)

```
SkipAny(recoveryIsEnabled):
  t := Lexer.CurrentToken();
  idx := Lexer.CurrentTokenIndex();
  while (Stack.Peek().Symbol ∈ N_S) do
    Apply(M[Stack.Peek().Symbol, t]);
  end while;

  Stack.Pop();

  stopTokens := FIRST'(Stack);
  while (t ∉ stopTokens and t ≠ $) do
    t := Lexer.NextToken();
  end while;

  if (t ∉ stopTokens) then
    if (recoveryIsEnabled) then
      Lexer.MoveTo(idx);
      return Error(stopTokens);
    else
      return ERROR_TOKEN;
    end if;
  end if;

  return Lexer.CurrentToken();
```

(b)

```
FIRST'(α = Y1Y2...Yk):
  first := ∅;
  for (i from 1 to k) do
    if (Yi ∈ TS) then
      first ∪= {Yi};
      if (Yi ≠ Any) then break; end if;
    else
      first ∪= MemorizedFirst'[Yi]\{ε};
      if (ε ∉ MemorizedFirst'[Yi]) then break; end if;
    end if;
  end for;
  if (∀i ∈ [1..k]: ε ∈ MemorizedFirst'[Yi] or Yi = Any) then
    first ∪= {ε};
  end if;
  return first;
```

(c)

```
Apply( X → Y1Y2...Yk):
  parent := Stack.Pop();
  for (i from k to 1) do
    child := new Node(Yi);
    Stack.Push(child);
    parent.Children.AddFirst(child);
  end for;
```

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

(d)

Fig. 1. Modified LL algorithms: (a) LL(1) parsing algorithm, (b) "Any" processing algorithm, (c) FIRST set construction, (d) Auxiliary algorithms: alternative applying and FIRST set memoization

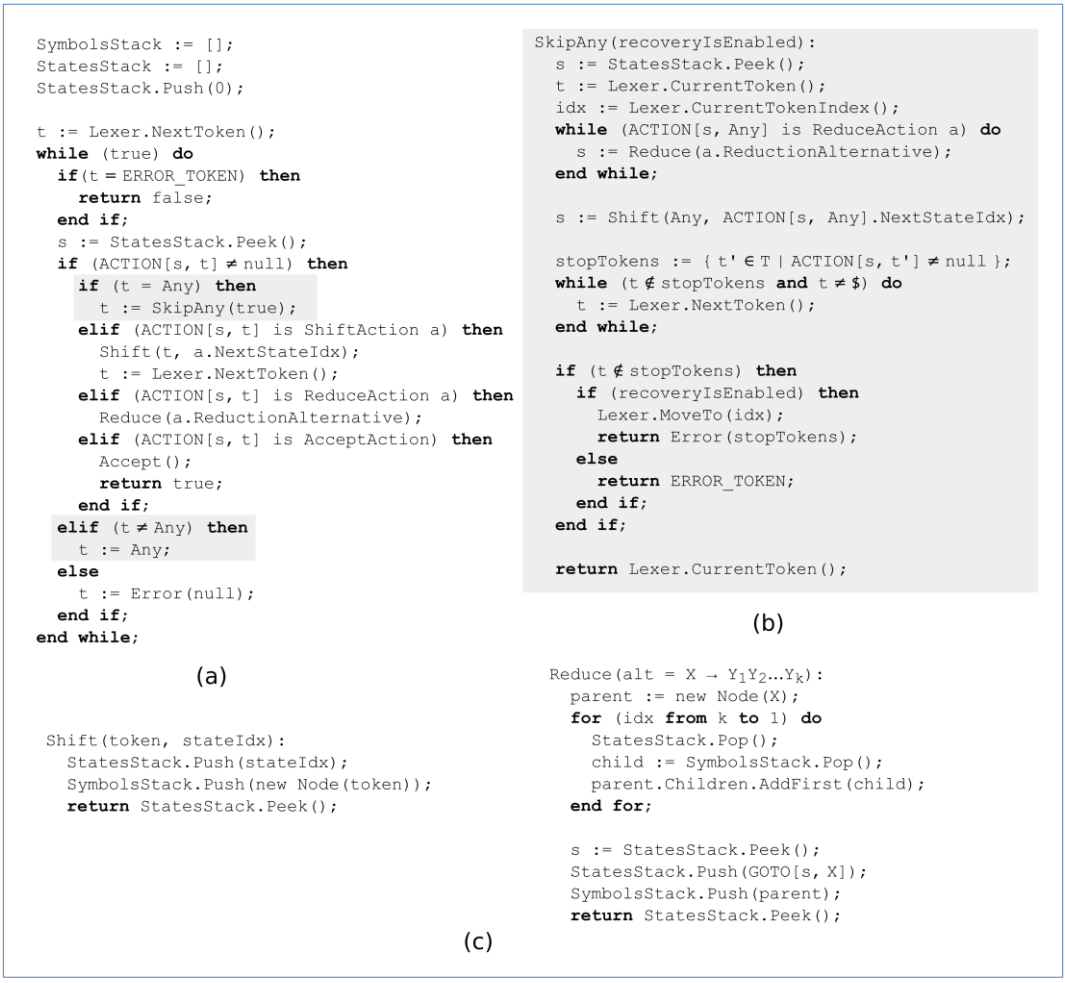

Fig. 2. Modified LR algorithms: (a) Modified LR(1) parsing algorithm, (b) "Any" processing algorithm, (c) Shift and reduce algorithms.

The essence of the parsing algorithm modification is similar to LL(1) case: tolerant parser is responsible not only for checking if the program can be derived from the start symbol, but also for translating it from a baseline language into a simplified one. In case an action for some actual combination of parser state and input token is undefined, parser tries to interpret the current token as the beginning of the subsequence of the program from $L(G)$ that corresponds to `Any` in the corresponding program from $L(G_s)$. In case there is an action available for `Any`, parser calls `SkipAny` routine (fig. 2b), where firstly all the possible `Reduce` actions are performed and secondly `Any` token is shifted. Note that we consider `ACTIONS` table to be cleared from Shift/Reduce conflicts in favor of `Shift` action. Also, there is no additional checking if `Shift` action exists, because this existence follows from the standard `ACTIONS` and `GOTO` construction algorithm. Having moved `Any` to a viable prefix, parser looks for the first token which is explicitly expected in $L(G_s)$ program and then continues parsing in the usual way.

In fig. 3, there is an LR(1) tolerant grammar for Java programming language written in the format supported by LanD parser generator. As it can be seen, island entities, such as enumerables, classes, methods, and fields, are clearly presented as solid rules. In comparison with a baseline

Java grammar, it is significantly shorter: the baseline grammar implementation for ANTLR parser generator[1] consists of 211 lines of lexer specification and 615 lines of parser description.

```
COMMENT : '//' ~[\n\r]* | '/*' .*? '*/'
STRING  : '"' ('\\"'|'\\\\'|.)*? '"'
CHAR    : '\'' ('\\\''|'\\\\'|.)*? '\''
MODIFIER : 'transient'|'strictfp'|'native'|'public'|'private'
    |'protected'|'static'|'final'|'synchronized'|'abstract'
    |'volatile'|'default'
ID       : [_$a-zA-Z][_$0-9a-zA-Z]*

CURVE_BRACKETED  : %left '{' %right '}'
ROUND_BRACKETED  : %left '(' %right ')'
SQUARE_BRACKETED : %left '[' %right ']'

file_content = entity*
entity = enum | class_interface | method
    | field_declaration | water_entity
enum  = common_beginning 'enum' name Any block ';'?
class_interface  = common_beginning ('class'|'interface')
    name Any '{' entity* '}' ';'?
method  = common_beginning type name arguments Any (';' | block)
field_declaration  = common_beginning type field (',' field)* ';'
field  = name ('['']')* init_value?
water_entity  = AnyInclude('@interface', 'import', 'package')
    (block | ';')+

common_beginning = (annotation|MODIFIER)*
init_value       = '=' init_part+
init_part        = Any | type_parameter

name = name_type
type = name_type
name_type_atom = type_parameter? ID type_parameter?
name_type      = name_type_atom ((('.'|'::') name_type_atom) | '['']')*
type_parameter = '<' (AnyAvoid(';') | type_parameter)* '>'

arguments  = '(' Any ')'
annotation = '@' name arguments?
block      = '{' Any '}'
```

Fig. 3. Java LR(1) tolerant grammar

4.2 “Any” processing improvements

4.2.1 Consecutive “Any” problem

In fig. 1c, `FIRST'` algorithm, which is the modified version of the standard `FIRST`, is presented. It is intended to solve the problem of consecutive `Any` described in [10]. The problem manifests itself when two or more `Any` tokens directly follow each other at the beginning of the sentence

[1] https://github.com/antlr/grammars-v4/tree/master/java

which can be derived from the stack. In this case, the subsequent `Any` hides some stop tokens from the previous one. Consider the following grammar G:

```
A = (a|b)+ B C; B = d | ; C = (e|f)? c
```

It can be simplified to the following G_s:

```
A = Any B C; B = d | ; C = Any c
```

The string `abc$` $\in L(G)$ is supposed to be successfully matched by the parser built for $L(G_s)$, because the following derivation may be performed:

$A \Longrightarrow Any\ B\ C \Longrightarrow Any\ C \Longrightarrow Any\ Any\ c.$

Having met the token `a`, the tolerant parsing algorithm starts the first `Any` processing. If the standard `FIRST` is used to find stop tokens, `FIRST(Stack)` set equals to `{d, Any}`, as a result, `SkipAny` skips all the input and returns an error. Taking into account that `Any` is allowed to match an empty sequence, `FIRST'` modification looks beyond the second `Any` and, in general, beyond all the subsequent `Any` symbols in searching some explicitly specified tokens which may follow a sequence corresponding to these `Any` tokens. Stop token set found with `FIRST'(Stack)` equals to `{d, c}`, thus the first `Any` captures `a` and `b` tokens and stops on `c`, the second one matches an empty sequence, and `abc$` string is admitted to be correct.

This approach is proved to be enough to build working parsers for real programming languages, such as C#, Java or PascalABC.NET. It can also be implemented for LR(1) through `ACTION` and `GOTO` static analysis. However, on closer inspection it becomes clear that algorithms modified in this way work correct only for a subclass of simplified grammars, satisfying an additional constraint:

Definition 2. Let $G_s = (N_s, T_s, P_s, S_s)$ be a grammar simplified with respect to a context-free grammar $G = (N, T, P, S)$. Enumerate as Any_1, Any_2, …, Any_n all the Any entries from the right-hand sides of productions from P_s, which appeared as a result of replacement of the corresponding γ_1, γ_2, …, γ_n subparts of the right-hand sides of productions from P in compliance with Definition 1. Derivation $S_s \overset{*}{\Rightarrow} \alpha_s Any_k Any_l \dots Any_t b \beta_s$, where k, l, …, $t \in [1..n]$, α_s, $\beta_s \in (N_s \cup T_s)^*$, $b \in T_s \backslash \{Any\}$, is not acceptable in G_s if $b \in \text{SYMBOLS}(\gamma_k \gamma_l \dots \gamma_t)$.

Informally speaking, the token which is a stop token for the last `Any` in a sequence is not allowed to appear in the area corresponding to one of the preceding `Any`, otherwise it will cause premature completion of `Any` processing. Let G has a different structure:

```
A = (a|b|c|d|e|f)+ B C; B = g | ; C = (h|i)+ a
```

It can be simplified to

```
A = Any B C; B = g | ; C = Any a
```

Herein, both replacements with `Any` are still satisfy the criterion 2c, but the restriction from Definition 2 is not satisfied, as `a` may follow the second `Any`, and at the same time it is a valid element of the area corresponding to the first one. As a result, while parsing `abba$`, the first `Any` is matched with an empty token sequence because `FIRST'([B, C])` equals to `{a, g}`, the second `Any` also cannot include `a`, so, valid input is not accepted.

In practice, the most common case of consecutive `Any` appearance does not break the restriction mentioned: in grammars we have developed, `Any` is often used as one of the possible variants for an element of a list, so, all the `Any` tokens in the derivation of such a list originate from a single `Any` entry in the grammar, therefore, derivation can be rewritten as $S_s \overset{*}{\Rightarrow} \alpha_s Any_k Any_k \dots Any_k b \beta_s$, and the corresponding condition $b \in \text{SYMBOLS}(\gamma_k)$ is false in accordance with Definition 1. To cover the general case, we introduce a mechanism for passing an additional information at `Any` processing stage. `Any` entry can be supplemented with two options: `Except` and `Include`. For each of them, a list of literals or token names can be passed as parameters. The concept of `AnyExcept` initially appeared in LightParse parser generator [15], but there it was intended to

compensate the lack of outer context analysis while constructing the set of admissible tokens. Our intention is different: symbols specified for `Except` option are supposed to compensate the lack of information in consecutive `Any` problem: they are supposed to be explicitly specified tokens that may follow the area corresponding to `Any` in $L(G)$. `Include` option allows one to approach this problem from a different angle, specifying tokens that shouldn't be interpreted as stop tokens despite their appearance in `FIRST'(Stack)`. So, for the grammar above we can use one of the following simplified analogues:

```
A = AnyExcept(g,h,i) B C; B = g | ; C = Any a
A = AnyInclude(a) B C; B = g | ; C = Any a
```

Having renamed `stopTokens` sets built in fig. 1b and 2b to `stopTokensBasic`, we transform stop token set construction for both LL and LR algorithms to

```
stopTokens := anyExceptSet.Count > 0
    ? anyExceptSet
    : stopTokensBasic.Except(anyIncludeSet);
```

where `anyExceptSet` and `anyIncludeSet` denote sets of tokens passed as option parameters for `Any` currently being matched. For error recovery purposes discussed in Section 4.3, `Any` also supports `Avoid` option. Its arguments are tokens the presence of which in the `Any`-corresponding area signals about program incorrectness or wrong alternative choice. To take `Avoid` into account, `while` loop condition transforms to

```
t ∉ stopTokens and t ∉ anyAvoidSet and t ≠ $.
```

In case token skipping is interrupted because current token equals to one of the `Avoid` arguments, this token passes to `Error` routine as a second argument.

Unlike in LL(1), there can be a situation in LR(1) when we do not know for sure what particular `Any` entry is being processed at the moment. This information is needed to access the corresponding options. To add support of `Any` options in LR(1), we introduce an additional type of LR(1) conflict called Any/Any conflict. It is reported when there is a state where multiple items have a dot before `Any`, and is needed to be resolved for successful parser generation.

4.2.2 Nesting level checking

While writing a tolerant grammar, developer usually has to make an additional effort to determine what bracketed areas may appear in the particular water, and if they can influence `Any` processing. Intuitively, such areas are perceived as a whole, and when `Any` is written instead of some better-grained water description, it may be missed that bracketed areas exist in that water in a real program. These areas may contain something that also appears right after that `Any` and therefore should be treated as a stop token. For example, being interested in fields of a C# class, we must capture `a`, `b`, `c`, and `d` in the fragment

```
int a = 0, b = 1;
DateTime c = new DateTime(2019, 5, 29),
    d = new DateTime(2019, 5, 31);
```

At the same time, we are not interested in initializers, so, the first intention is to describe field declaration with the rules

```
fields = type name init? (',' name init?)* ';'
init = '=' Any
```

Unfortunately, these rules work only for the first declaration. The set `{',', ';'}` is a stop token set for `Any`, and in the second declaration, comma separates not only fields but also arguments bordered with round brackets. Generally speaking, `Any` does not satisfy the formalization in this case. At the same time, simplicity is the crucial property of the tolerant grammar, and the way in which water is described above is more preferable than the following one:

```
init = '=' water
s_water = '[' (Any | s_water)+ ']'
r_water = '(' (Any | r_water)+ ')'
c_water = '{' (Any | c_water)+ '}'
water = (Any | c_water | r_water | s_water)+
```

To return the first version of `init` rule to the boundaries of the simplified grammar definition, we add to the parsing algorithms a capability to take into account nested bracketed structures. A pair of brackets is described like

```
ROUND_BRACKETED : %left '(' %right ')'
```

and nesting level is tracked by lexical analyzer. If several kinds of pairs are described, it is believed that any pair can be nested in any pair. When `Any` is processed, it is allowed to end only at the same depth at which it begins. To control this situation, `SkipAny` methods are modified uniformly both for LL and LR. Firstly, at the beginning of a skip process, an additional variable is initialized:

```
depth := Lexer.CurrentDepth();
```

Secondly, in-loop `Lexer.NextToken()` call is replaced with `Lexer.NextToken(depth)`. Passing the initial nesting level to a lexer, we force it to read the input stream until the depth of the next token equals the depth of the first token in the sequence corresponding to `Any`. Thus, `Any`-corresponding area is allowed to include stop tokens in nested structures because these nested structures are invisible to the parsing algorithm. Third modification is an additional checking to prevent moving through the upper nesting level. In fig. 4, there are two cases allowed by the first two modifications. Token `a` is the beginning of `Any` area, and `b` is a stop token. Obviously, the way `Any` symbol is matched on the right breaks the semantic integrity of a bracketed area. We consider such `Any` usage to be a bad practice, so, if lexer returns a token denoting the end of some pair and rise to the level above the initial, and this token is not a stop token, parser reports an error which means that grammar should be refined.

Fig. 4. Possible “Any” matching supported by nesting level tracking.

4.3 Error recovery

4.3.1 Algorithms

As noted in Section 1, in case water entities look similar to islands, developer has to refine patterns and to add some antipatterns to avoid false positives. For a deterministic parsing, the problem of water and island similarity may have unpleasant consequences not only when there is a full match between island pattern and water entity, but even if a water entity and an island have a number of common starting tokens. In this case, parser starts analyzing a water entity as an island, finds a mismatch and fails to proceed analysis. It is important to note that this parsing failure indicates not the incorrect $L(G)$ program but misinterpretation of the program in terms of G_s. Generalized parsing algorithms are able to process such a situation exploring both ways an entity can be interpreted in and rejecting the failed one. To get a similar benefit from our modified deterministic parsing while preserving mostly linear complexity, we add special `Any`-based error recovery routines in both LL(1) and LR(1) algorithms. These routines are shown in fig. 5.

```
Error(stopTokens):
  if (Lexer.CurrentTokenIndex() ∈ RecoveredIn) then
    return ERROR_TOKEN;
  end;
  RecoveredIn ∪= { Lexer.CurrentTokenIndex() };

  currentNode := Stack.Pop();
  do
    if (currentNode.Parent ≠ null) then
      maxChildIndex :=
        currentNode.Parent.Children.Count - 1;
      indexOfCurrent := currentNode.Parent
        .Children.IndexOf(currentNode);
      for (i from indexOfCurrent + 1
        to maxChildIndex) do
        Stack.Pop();
      end for;
    end if;
    currentNode := currentNode.Parent;
  while(currentNode ≠ null and (
    currentNode.Symbol ∉ RecoverySymbols or
    Any = GetDerivation(currentNode)[0] or
    IsUnsafeAny(stopTokens)
  ));

  if (currentNode ≠ null) then
    return SkipAny(false);
  else
    return ERROR_TOKEN;
  end if;
```

(a)

```
Error(stopTokens):
  if (Lexer.CurrentTokenIndex() ∈ RecoveredIn) then
    return ERROR_TOKEN;
  end;
  RecoveredIn ∪= { Lexer.CurrentTokenIndex() };

  lastMatched := ε;
  // possible derivation items
  PDI := {};
  basePDI := {};
  do
    if (SymbolsStack.Count > 0) then
      lastMatched := SymbolsStack.Pop();
    end if;
    StatesStack.Pop();
    if (StatesStack.Count > 0) then
      s := StatesStack.Peek();
      basePDI := { i = X → α•Yβ | i ∈ STATE[s],
        Y = lastMatched.Symbol,
        (PDI = {} ∨ ∃i' ∈ PDI : i' = X → αY•β) };
      PDI := basePDI;
      do
        PDI ∪= { i = X → α•Yβ | i ∈ STATE[s],
          ∃i' ∈ PDI : i' = Y → •β' };
        while (PDI changes);
    end if;
  while (StatesStack.Count > 0 and (
    |basePDI| = |PDI| or
    ∄i ∈ PDI \ basePDI : i = X → α•Yβ, Y ∈ RecoverySymbols or
    Any = GetDerivation(lastMatched)[0] or
    IsUnsafeAny(stopTokens)
  ));

  if (StatesStack.Count > 0) then
    return SkipAny(false);
  else
    return ERROR_TOKEN;
  end if;
```

(b)

Fig. 5. "Any"-based error recovery algorithms: (a) LL(1) algorithm, (b) LR(1) algorithm.

In the modified parsing algorithms, two types of error can occur. The first one happens when LL(1) parser cannot match the current token or apply some alternative and `Any` is not acceptable at the point, or when LR(1) parser has no shift or reduce action for the current token as well as for `Any`. The second type occurs when `Any` processing starts and no stop tokens are found till the end of the input or a token specified as `Avoid` argument is met. Recovery initiated for the first type does not influence the algorithm linearity as parsing is resumed at the token where the error occurred. Acting the same way for the second type is meaningless, especially when the end of the input is reached, because significant part of islands might be uncontrollably skipped. Instead, a limited backtracking is performed. In fig. 1b and 2b, `Lexer.MoveTo(idx)` call shifts a token stream pointer to the token that triggered `Any` processing, at this point recovery is tried to be carried out. In Section 4.3.2, the influence of this backtracking on parsing algorithm time complexity is analyzed. In both LL(1) and LR(1) error processing algorithms, `RecoveredIn` set stores all the indices of tokens at which recovery was once performed, so, it is guaranteed that from one recovery to another parsing process moves at least one token forward.

Like in standard recovery algorithms [16, pp. 283–285], a set of nonterminals at which recovery can be performed is defined. These nonterminals are called *recovery symbols*. Possible recovery symbols can be revealed through the static grammar analysis. Given the grammar $G_s = (N_s, T_s, P_s, S_s)$, we formally define $RecoverySymbols$ set as follows:

$$\left\{ n \in N_s \mid n \overset{*}{\Rightarrow} Any\ \alpha, \nexists n' \in N_s : \left(n \overset{*}{\Rightarrow} n' Any\ \alpha \wedge n' \overset{*}{\Rightarrow} \varepsilon \right) \right\}, \alpha \in (N_s \cup T_s)^*.$$

Recovery symbols are pre-computed at parser construction stage. Developer can disable recovery at all or specify particular nonterminals from this set which should be used for recovery, otherwise, all the elements of the set are taken into consideration when `Error` routine is called.

In the context of a deterministic tolerant parsing problem, recovery symbols have specific semantics. They represent decision points at which parser may choose the wrong alternative, try to match a water entity as an island, and provoke an error. Recovery itself means returning to a decision point through the grammar ancestors of the currently unmatched token or unparsed nonterminal and changing the interpretation of the part of the input that is already associated with a recovery symbol's subtree to water. More precisely, the part of the input from the first token mistaken for an island part to the first token at which the difference between an island pattern and an actual water entity manifests itself is supposed to be the beginning of the sequence corresponding to `Any` from which the water alternative starts. Backtracking to the token a wrong decision was made at is not needed in this interpretation. The end of an `Any`-corresponding sequence is looked for with a usual `SkipAny` call, then parsing continues in an ordinary way. In fig. 3, `entity` is one of the recovery symbols. It allows the parser to recognize classes, enumerables, methods, and fields as islands, while annotation definitions, constructors, initialization blocks, etc. are skipped as the water, sometimes with the involvement of recovery mechanisms.

LL(1) error recovery algorithm is presented in fig. 5a. We take advantage of the fact that at any stage of the top-down parsing a partially built syntax tree is available, and blank nodes for what is expected are on the stack. Knowing the tree node corresponding to the unparsed symbol, we may find a recovery symbol node by moving through its ancestors. The higher we go, the wider area will be reinterpreted. Simultaneously with walking up the syntax tree, right siblings of the `currentNode` should be removed from parsing stack as they are unparsed parts of the interpretation being rejected. The appropriate recovery symbol is considered to be found if it satisfies two additional conditions. Firstly, the water alternative should not be the alternative in favor of which the decision was originally made, otherwise no reinterpreting takes place as error actually occurred in the water. To check it, `GetDerivation` is called. It takes the built part of recovery symbol's subtree and returns a leaf sequence which is a partially revealed part of the $L(G_s)$ program, derived from this symbol. This sequence must not start with `Any`. Secondly, in case error took place at `Any` skipping, `IsUnsafeAny` prevents parsing resumption on `Any` from the recovery symbol alternative if new skipping leads to the same erroneous situation. The decision is made on the basis of old and new stop token sets comparison and `Avoid` options analysis.

For LR(1) algorithm, recovery is more complex and heuristic due to the nature of a bottom-up parsing. Unlike in LL case, we do not know for sure what are the exact entities that are currently being analyzed, so, we try to build a set of possible candidates basing on the information stored on the stacks. In fig. 5b, there is an LR(1) error recovery routine. On each iteration of `do-while` loop, one of the symbols already matched is popped along with the state parser went to after this successful matching, then `basePDI` set is constructed. It consists of the current state items having the dot before the last popped symbol. Productions of the items added to this set are possible participants of the erroneous area derivation. Basing on `basePDI`, `PDI` set is constructed in a way that looks like inverted `CLOSURE` [16, pp. 243–245] algorithm. Additional `PDI` items capture the higher-level grammar entities from which the area that is needed to be reinterpreted may be derived.

Recovery algorithms presented simplify the process of grammar extension and reduction. Recovery symbol alternatives become grammar building blocks: in case we are not interested in some Java island, its alternative can be excluded from `entity` rule, then program areas previously corresponding to that alternative are recognized as the water, possibly through recovery algorithm application. Inversely, to add a support for class constructors in the grammar in fig. 3, we have to write only one `constructor` rule and add this symbol in `entry` alternatives list, then

constructors stop being interpreted as the water, because the rule appears allowing the parser to analyze them from beginning to the end with no error occurred.

4.3.2 Complexity analysis

As noted, errors happening on `Any` processing require limited backtracking. The particular increase in running time of the algorithm depends on number and length of backtracked sequences. From the prohibition of multiple recovery at the same token, it follows that there can be only one backtracking to a particular position, so, the worst case is when the following situation repeats sequentially for each token except the first one: `Any` processing starts on the token, fails by reaching the end of the input and backtracks to that token, then recovery starts, the token matches successfully with the help of the water alternative, and the next token becomes the token under consideration. In this scenario, a number of times the token is examined equals to its sequential number counting from one. For the i_{th} token, $i-1$ examinations are occurred on `Any` skipping started at previous tokens and at the current one, and 1 examination is for some final match. As backtracking itself consists of a simple index reassignment, it does not increase this counter. It can be shown that this worst-case scenario takes place for inputs `ac$`, `aac$`, `aaac$`, etc. and a parser generated by the following LL(1) grammar:

```
S = a Any b | Any S |
```

```
…
CURVE_BRACKETED : %left '{' %right '}'
ROUND_BRACKETED : %left '(' %right ')'
SQUARE_BRACKETED : %left ('['|GENERAL_ATTRIBUTE_START) %right ']'
…
namespace = 'namespace' name '{' namespace_content '}'
entity = enum | class_struct_interface | method
   | field_decl | property | water_entity
enum = common 'enum' name Any '{' Any '}' ';'?
class_struct_interface =
   common ('class'|'struct'|'interface') name Any '{' entity* '}' ';'?
method = common type name arguments Any (init_expression? ';' | block)
field_decl = common type field (',' field)* ';'
field = name ('[' Any ']')? init_value?
property =
   common type name (block (init_value ';')? | init_expression ';')
water_entity =
   AnyInclude('delegate', 'operator', 'this') (block | ';')+

common = entity_attribute* modifier*
modifier = MODIFIER | 'extern'
init_expression  = '=>' Any
init_value = '=' init_part+
init_part = Any | type
arguments = '(' Any ')'
block = '{' Any '}'
…
```

Fig. 6. Fragment of the C# tolerant grammar.

The total number of token examinations equals to $\frac{1}{2}n^2+\frac{1}{2}n$, it means that our algorithms are $O(n^2)$ in the worst case. However, experiments show that the percentage of recoveries required

backtracking is insignificant in comparison with the total number of recoveries and tokens: for example, in all the Java projects from subsection 5.2 taken together, there are 27393 files splitting at 26255589 tokens, while total number of recoveries is 32683 for LL(1) and 31861 for LR(1), and only 20 recoveries for each type of parsing are performed after on-`Any` error.

5. Experiments

To test the algorithms described in Section 4, tolerant grammars for the following programming languages, markup languages and specification formats are developed: C#, Java, PascalABC.NET, XML, Markdown, YACC, and Lex. All the sources are available on GitHub[2]. For a large-scale testing, C# and Java are chosen as the languages complex enough and having a large number of well-known open-source repositories. For both languages, LL(1) and LR(1) tolerant parsers are generated with LanD parser generator on the basis of the corresponding tolerant grammars.

As tolerant parsers are created to capture particular islands, the purpose of the experiment is to evaluate precision and recall of this capturing. Stages of the experiment are the same for both languages. For each of the projects under consideration, tolerant parser is firstly applied to parse all the project files written in the corresponding language. By traversing syntax trees built, types and names of the islands are extracted in report files, per report for each island type. This extraction does not require some severe postprocessing: island type is actually a node type, and name is stored in one of this node's children. Secondly, the same files are parsed by a baseline parser. Roslyn is used as a baseline parser for C#, and Java parser is generated with ANTLR from the full grammar of the language[3]. Then information about program entities that are specified as islands for our tolerant parsers is extracted from trees built by these baseline parsers, so the second group of reports is obtained. At the third stage, two reports for the same type are compared in an automated way to eliminate the human factor. Matches are excluded, so only the information about entities found by one parser and not found by another one remains. It is then explored manually.

For each of the languages, there is a table whose rows correspond to projects parsed and columns correspond to island types. There is also an additional "Total files" column allowing to estimate the scale of the project. In a table cell, there is a number of islands of the corresponding type found by our tolerant parser for the corresponding project. We have obtained that these numbers are the same for LL(1) and LR(1) parser, so we do not need two separate tables for a single language. In case tolerant parser finds less island entities than the baseline one, the number of entities missed is specified in parentheses with a minus sign. In addition to the tables, a detailed analysis of mismatches is provided.

5.1 C# tolerant parsing

For C# programming language, five open-source projects from different domains are considered:

- **Roslyn** project includes C# and Visual Basic compiler sources and lots of test files capturing different complex and uncommon variants of a C# program;
- **PascalABC.NET** consists of the corresponding language compiler and IDE sources, it has a relatively long history reflected in the legacy code written by differently experienced contributors;
- **ASP.NET Core** refers to the web development domain: it is a cross-platform .NET-based web framework;
- **Entity Framework Core** is an object-relational mapper allowing one to work with a database using .NET objects;

[2] https://github.com/alexeyvale/SYRCoSE-2019

[3] https://github.com/antlr/grammars-v4/tree/master/java

- **Mono** is an open source third-party implementation of Microsoft's .NET Framework including C# compiler, Common Language Runtime virtual machine, lots of core libraries and, again, a great number of test files.

Parsing results are presented in Table 1, a fragment of the tolerant LR(1) C# grammar is presented in fig. 6. Note that classes, structures, and interfaces correspond to a single grammar entity, so their total number presented in a single "Classes" column of the table. In the discussion below, footnotes contain paths to files relative to the root directory of the corresponding project.

For Roslyn sources, there are 5 methods found by Roslyn and missed by LanD. Four of them are local[4] methods[5] (methods declared inside other methods), this feature recently appeared in C# 7.0. In case this kind of methods is important for a particular task, it is trivial to add their support in the grammar. One needs to modify the grammar above by adding `method` symbol as an alternative to `Any` inside the `block`. It is worth noting, that Roslyn project is the only project where the usage of this feature is revealed. The 5th lost method is from a test file where the text of the program is saved in Japanese Shift-JIS encoding[6]. The class name written in Japanese provokes an error which does not affect the detection of the class itself but stops parser from further class content analysis. We consider the usage of national alphabets for entity naming to be a rare case, but, if necessary, `ID` token can be adopted as needed.

Two properties from different files are not found by LanD, in both cases it is caused by missing expression for expression-bodied property preceding the uncaptured one. The expression depends on external conditional compilation symbols and is not substituted at all in case the isolated file is analyzed. In the following code, `IsWindows` is not recognized by LanD, because it is interpreted as a part of expression for `Configuration`:

```
public static ExecutionConfiguration Configuration =>
#if DEBUG
    ExecutionConfiguration.Debug;
#elif RELEASE
    ExecutionConfiguration.Release;
#else
    #error Unsupported Configuration
#endif
public static bool IsWindows =>
    Path.DirectorySeparatorChar == '\\';
```

This kind of inconsistency can be partially handled by using `AnyAvoid(MODIFIER)` instead of `Any` in `init_expression` grammar rule. For the example above, this handling leads to loss of the information about `Configuration` property, as it will be treated as water, but protect the following entities starting with the one that starts with the keyword.

For PascalABC.NET and ASP.NET Core, all the entities found by Roslyn are also found by LanD. For Entity Framework Core, the difference in number of fields and methods is caused by the situation[7] similar to the one presented in the code above, and the difference in number of properties is provoked by property types containing Greek letters[8]. The latter refers us again to the national alphabets problem.

[4] src/Compilers/CSharp/Test/Emit/Emit/EndToEndTests.cs

[5] src/Compilers/CSharp/Portable/FlowAnalysis/NullableWalker.cs

[6] src/Compilers/Test/Resources/Core/Encoding/sjis.cs

[7] test/EFCore.SqlServer.FunctionalTests/Query/SimpleQuerySqlServerTest.Where.cs

[8] test/EFCore.Tests/ModelBuilding/ModelBuilder.Other.cs

Voluminous results are obtained for Mono sources. Most losses are concentrated in files that are incorrect in terms of a full C# grammar: as an example, 26 files[9] contain unclosed conditional compilation directives and mismatch in the number and type of opening and closing brackets, half of the 122 missed classes belongs to a group of files[10] containing LINQ to SQL code written in accordance with Visual Basic syntax, there are also files with .cs extension written in a specific format, such as a skeleton file[11] for jay parser generator, where each line starts with a point. However, there is also a group of missed entities that illustrates a real LanD drawback. These entities are contained in `test-async`[12] and `test-partial`[13] groups of Mono test files.

Table 1. Number of entities found in C# projects.

Project	**Total files**	**Enums**	**Classes**	**Fields**	**Properties**	**Methods**
Roslyn	8759	482	23705	20265	23127 (-2)	116312 (-5)
PABC.NET	2802	359	5522	16739	12023	37027
ASP.NET Core	7356	333	12604	10214	16301	44163
EF Core	2997	101	7783	4687 (-1)	16941 (-2)	26421 (-135)
Mono	37224	4928 (-1)	60187 (-122)	166958 (-67)	99167 (-36)	309580 (-670)

At grammar design and refinement stage, we did not take into consideration, that there are some keywords in C# that appeared recently and were implemented as *contextual* keywords to protect legacy code. It means that they still can be names for classes, methods, etc. For example, the following code is valid in C# (method bodies are omitted):

```
namespace async
{
    partial class async
        { partial void partial(); }
    partial class partial
    {
        // async method named 'async'
        async Task<async> async() { ... }
        // method named 'async' returning an object of type 'async'
        async async(async async) { ... }
    }
}
```

Proper interpretation of a contextual keyword depends on a heavy context analysis going far beyond LL(1) or LR(1) parsing. In Roslyn sources, there is a special `ShouldAsyncBeTreatedAsModifier` method checking lots of specific conditions, each of which covers a particular `async` placement relative to non-contextual keywords, predefined types, and `partial` keyword. Besides, up to 2 additional tokens are required to make a correct decision.

[9] mcs/errors
[10] mcs/tools/sqlmetal/src/DbLinq/Test
[11] mcs/jay/skeleton.cs
[12] mcs/tests/test-async-*.cs
[13] mcs/tests/test-partual-*.cs

Fortunately, to meet contextual keywords used as identifiers seems to be almost improbable. In our experiments, such cases were revealed only in synthetically created testing files, not in a real production code. Moreover, using `async` or `partial` contextual keywords as public entity identifiers one breaks general C# naming conventions[14] which are usually used as a basis for particular code style rules being applied inside a developers team.

5.2 Java tolerant parsing

For Java programming language, the following projects are considered:

- **Java Development Kit** is a toolbox consisting of Java compiler, core libraries, and Java Runtime Environment;
- **Elasticsearch** is an engine for a full-text search;
- **Spring Framework** is a Java framework used to build applications for different subject domains;
- **RxJava** is a library for composing asynchronous and event-based programs.

Parsing results are presented in Table 2, and a tolerant LR(1) Java grammar is presented in fig. 3.

As it can be noted, there is the only difference between baseline and tolerant parsing results. `FIND_MASK`, `NEW_MASK`, and `RELEASE_MASK` fields are missed by the tolerant parser in the following code:

```
private final static int
    CREATE_MASK = 1<<CREATE,
    FIND_MASK = 1<<FIND,
    NEW_MASK = 1<<NEW,
    RELEASE_MASK = 1<<RELEASE,
    ALL_MASK = CREATE_MASK|FIND_MASK|NEW_MASK|RELEASE_MASK;
```

Unlike all the other types of brackets considered in Section 4.2.2, angle brackets cannot be defined as a pair in the LanD grammar because they may appear in the program in different meanings, some of which assume they can be used separately from each other. However, in case they bracket type parameters, it is important to match these parameters as a whole to prevent inner commas from being interpreted as field separators. It is hard to resolve this problem correctly staying in the tolerant parsing boundaries and, actually, in the boundaries of a context-free parsing and lexing too [17]. To make a correct decision, an analysis of the context angle bracket appears at is needed. Recovery algorithm combined with `Avoid`-based error triggering helps to handle inputs like

```
private static final long ADD_WORKER = 0x0001L << (TC_SHIFT + 15);
```

by interpreting all the angle brackets as opening for a `type_parameter` in fig. 3, triggering an error on `';'` token which is forbidden in type parameters, and reinterpreting the outermost type parameter as `Any` from `init_part` water alternative. However, this processing cannot prevent the loss of some middle fields from the group of fields defined simultaneously.

Table 2. Number of entities found in Java projects

Project	Total files	Enums	Classes	Fields	Methods
JDK	7704	151	10590	46176 (-3)	88709
Elastic	10972	387	14914	36830	94722
Spring	7063	100	12060	18402	61515
RxJava	1654	36	2728	6258	19931

[14] https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/capitalization-conventions

5.3 Summary

As experiments show, both C# and Java tolerant parsers using our modified LL(1) and LR(1) algorithms are viable and allow one to find almost all the islands that can be found with a baseline parser. Mismatches cannot be considered as a tolerant parsing disadvantage: the ones occurred in erroneous C# programs are not unexpected since our algorithms are designed to work with correct programs, while for the most part of the valid programs containing lost islands, possible grammar fix can be easily suggested due to grammar simplicity and extensibility. However, there is also a tiny group of valid programs for which it is impossible to catch the missing island without performing an additional context analysis. This problem is actually not a tolerant parsing problem but a context-free analysis problem in general.

6. Conclusion

In the present paper, several algorithms and algorithm modifications aimed at island-grammars-based deterministic tolerant parsing are proposed. LR(1) parsing algorithm modification is performed in accordance with the simplified grammar formal definition previously developed by the author of the paper. A special `Any` symbol is integrated into the algorithm to add a capability to match token sequences which are not explicitly described in the grammar. LR(1) tolerant grammars tend to be shorter and more comprehensible than their LL(1) analogues written for previously modified LL(1) algorithm. Additional restriction defining simplified grammars subclass for which LL(1) and LR(1) tolerant parsing algorithms are always able to correctly handle consecutive `Any` problem is revealed. `Any` processing mechanisms are introduced to expand correct consecutive `Any` processing to entire simplified grammars class. Nested bracketed structures tracking is implemented to give the grammar developer a possibility not to take into consideration the content of in-water bracketed areas while replacing water description with `Any`. Error recovery algorithms are proposed for LL(1) and LR(1) tolerant parsing. Unlike the standard error recovery, they are designed not to resume parsing for an incorrect program, but to find the area which was mistakenly interpreted as an island and reinterpret it as a water. Through the series of experiments with C# and Java parsers generated by tolerant grammars developed for LanD parser generator, modified LL(1) and LR(1) parsing algorithms are proved to be able to successfully analyze the source codes of industrial software products.

Though the current tolerant parsing implementation is enough to work on solution of the crosscutting concerns markup problem mentioned in Section 1, an improvement of parsing results for syntactically incorrect programs may broaden the markup tool application opportunities. We have an assumption that `Any`-based recovery responsibility area may be explicitly specified for a particular grammar, and outside of this area some other recovery algorithms aimed at parsing resumption for an incorrect program can be used. Thus, our tolerant parsers will be capable to capture constructs of interest in such a program, like baseline parser successfully does in Section 5.1, instead of totally failing or interpreting all of these constructs as a single water piece. Besides, as performance was not the key goal until the present, we were satisfied with the generally linear dependency between input length and running time of the algorithms. However, basing on the knowledge of LanD implementation details, we are sure that performance can be improved (not in terms of time complexity classes, but in terms of absolute values of the algorithm running time). So, the algorithms and structures optimization is the second possible direction for further work on tolerant parsing.

References / Список литературы

[1]. Moonen L. Generating robust parsers using island grammars. In Proc. of the Eighth Working Conference on Reverse Engineering (WCRE’01). IEEE Computer Society, 2001, pp. 13–22.

[2]. Afroozeh A., Bach J.-C., van den Brand M., Johnstone A., Manders M., Moreau P.-E., Scott E. Island grammar-based parsing using GLL and Tom. Software Language Engineering: 5th International Conference, Revised Selected Papers. Springer Berlin Heidelberg, 2013, pp. 224–243.
[3]. Moonen L. Lightweight impact analysis using island grammars. In Proc. of the 10th International Workshop on Program Comprehension (IWPC). IEEE Computer Society, 2002, pp. 219–228.
[4]. Scott E., Johnstone A. GLL parsing. Electronic Notes in Theoretical Computer Science, 2010, vol. 253, issue 7, pp. 177–189.
[5]. Tomita M. Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems. Norwell, MA, USA: Kluwer Academic Publishers, 1985, 201 p.
[6]. Goloveshkin A.V., Mikhalkovich S.S. LanD: a framework for layer-by-layer program development, In Proc. of the 25th conference "Modern information technologies: tendencies and perspectives of evolution", 2018, pp. 53–56 (in Russian) / Головешкин А.В., Михалкович С.С. LanD: инструментальный комплекс поддержки послойной разработки программ. Труды XXV всероссийской научной конференции «Современные информационные технологии: тенденции и перспективы развития». Издательство Южного федерального университета, 2018, стр. 53–56
[7]. Goloveshkin A.V. Searching and analysing crosscutting concerns in marked up programming language grammar. University News. North-Caucasian Region. Technical Sciences Series, 2017, issue 3, pp. 29–34 (in Russian). DOI: 10.17213/0321-2653-2017-3-29-34 / Головешкин А.В. Поиск и анализ сквозных функциональностей в размеченной грамматике языка программирования. Известия вузов. Северо-Кавказский регион. Технические науки, 2017, вып. 3, стр. 29–34. DOI: 10.17213/0321-2653-2017-3-29-34
[8]. Fuksman A. Technological Aspects of Program Design. Moscow: Statistika, 1979, 184 p. (in Russian) / Фуксман А.Л. Технологические аспекты создания программных систем. Москва: Статистика, 1979, 184 стр.
[9]. Conejero J., Hernández J., Jurado E., van den Berg K. Crosscutting, what is and what is not?: A formal definition based on a crosscutting pattern. Tech. Rep. 5/TR28/07, 2007, 30 p.
[10]. Goloveshkin A., Mikhalkovich S. Tolerant parsing with a special kind of "Any" symbol: the algorithm and practical application. Trudy ISP RAN/Proc. ISP RAS, 2018, vol. 30, issue 4, pp. 7–28. DOI: 10.15514/ISPRAS-2018-30(4)-1.
[11]. Mössenböck H. (2014) The compiler generator Coco/R. Available at: http://ssw.jku.at/Coco/Doc/UserManual.pdf, accessed 07.02.2019.
[12]. Malevannyy M. Lightweight parsing and its application in development environment. Informatization and communication, 2015, issue 3, pp. 89–94 (in Russian) / Малёванный М.С. Легковесный парсинг и его использование для функций среды разработки. Информатизация и связь, 2015, вып. 3, стр. 89–94.
[13]. Grune D., Jacobs C. J. Parsing Techniques: A Practical Guide (2nd Edition). New York, USA: Springer-Verlag New York, 2008, 662 p.
[14]. Aho A.V., Lam M.S., Sethi R., Ullman J.D. Compilers: Principles, Techniques, and Tools (2nd Edition). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006, 1000 p.
[15]. Malevannyy M., Mikhalkovich S. Aspect markup of a source code for quick navigating a project. In Proc. of the 11th Central and Eastern European Software Engineering Conference in Russia, ser. CEESECR '15. New York, NY, USA: ACM, 2015, pp. 4:1–4:9.
[16]. Aho A., Ullman J. Translations on a context free grammar. Information and Control, 1971, vol. 19, issue 5, pp. 439–475.
[17]. Van Wyk E.R., Schwerdfeger A.C. Context-aware scanning for parsing extensible languages. In Proc. of the 6th International Conference on Generative Programming and Component Engineering, New York, NY, USA: ACM, 2007, pp. 63–72.

Информация об авторе / Information about author

Алексей Валерьевич ГОЛОВЕШКИН в 2015 году получил степень магистра по направлению «Фундаментальная информатика и информационные технологии» в Южном федеральном университете, Ростов-на-Дону, Россия. В настоящее время проводит исследования на базе данного университета, готовит диссертацию на соискание учёной степени кандидата технических наук. К сфере его научных интересов относятся компиляторы, языки программирования, программная инженерия.

Alexey Valerievitch GOLOVESHKIN obtained the master’s degree in fundamental informatics and information technologies in 2015 at Southern Federal University, Rostov-on-Don, Russia. Currently he does research at Southern Federal University working on the PhD thesis. His current research interests include compilers, programming languages, and software engineering.

DOI: 10.15514/ISPRAS-2019-31(3)-2

Graphic DSL for Mobile Development

A. Gudiev, ORCID: 0000-0002-0674-8621 <arturgudiev93@gmail.com>
A. Grazhevskaya, ORCID: 0000-0002-5069-443X <sagrapro7@gmail.com>
Saint Petersburg State University,
7/9 University Embankment, 199034, Russia.

Abstract. Due to the increase in the number of platforms, languages and methods which are used in mobile development, the general technology elaboration problem is quite relevant nowadays. Graphic languages simplify software development, allowing to present program structure in terms of visual diagrams. Besides, graphic languages allow software engineers to avoid a lot of mistakes at the initial stages of design and development. Graphic domain-specific languages (DSL) facilitate application development by use of concrete domain abstractions. In this approach the mobile application structure will be presented in the form of various controllers connected among themselves through ports and corresponding to some complete fragments of logic. Controllers in turn consist of various states which allow to describe a data flow in the controller using various element connections. In each state the UI form which contains the graphic primitives and events connected with primitives can be described. Besides, code generator for UbiqMobile platform is implemented which will allow to generate UbiqMobile applications by the visual diagrams. At the end of the article demonstration examples on which the implemented DSL language was tested are given. The application allowing the user to get the trains schedule is provided in the first example. In the second application the user can log in to receive a check-in code.

Keywords: dsl; mobile development

For citation: Gudiev A.V., Grazhevskaya A.S. Graphic DSL for mobile development. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 3, 2019. pp. 29-34. DOI: 10.15514/ISPRAS-2019-31(3)-2

Графический DSL для разработки мобильных приложений

А.В. Гудиев, ORCID: 0000-0002-0674-8621 <arturgudiev93@gmail.com>
А.С. Гражевская, ORCID: 0000-0002-5069-443X <sagrapro7@gmail.com>
Санкт-Петербургский Государственный Университет,
199034, Россия, г. Санкт-Петербург, Университетская набережная 7-9

Аннотация. В связи с увеличением количества платформ, языков и методов, использующихся в разработке мобильных приложений, задача выработки общей технологии довольно актуальна. Графические языки упрощают разработку, позволяя представить структуру программного обеспечения в виде графических диаграмм. Кроме того, графические языки помогают избежать множества ошибок еще на начальных этапах проектирования и разработки. Графические предметно-ориентированные языки (DSL) облегчают разработку программ путем применения абстракций конкретной предметной области. В данной работе представлен архитектурный шаблон мобильного приложения и созданный на его основе графический DSL, позволяющий описывать основную структуру мобильного приложения в терминах котнроллеров, состояний и переходов между ними. При таком подходе структура мобильного приложения будет представлена в виде различных контроллеров, связанных между собой при помощи портов и соответствующих некоторым целостным фрагментам логики. Сами контроллеры в свою очередь состоят из различных состояний, которые позволяют описать поток данных в контроллере путем соединения при помощи элемента-связи. В каждом состоянии может быть описана экранная форма, в которой содержатся графические примитивы и связанные с ними события, срабатывающие при их изменении. Кроме того, для

разработанного DSL реализована автоматическую генерация кода для платформы UbiqMobile. В конце статьи приводятся демонстрационные примеры, на которых был апробирован DSL язык. В качестве первого примера приводится приложение, позволяющее пользователю посмотреть расписание электричек. Во втором приложении пользователь может войти в систему для того, чтобы получить check-in код.

Ключевые слова: предметно-ориентированные языки; мобильная разработка

Для цитирования: Гудиев А.В., Гражевская А.С. Графический DSL для разработки мобильных приложений. Труды ИСП РАН, том 31, вып. 3, 2019 г., стр. 29-34 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(3)-2

1. Introduction

A large number of platforms, languages, and methods are used in mobile application development. Existing mobile development tools significantly differ from each other, and the common technology implementation problem is still relevant.

There are various ways of the high-level description of mobile application – architectural patterns mvc, pac, microkernel, etc [1]. All these patterns were borrowed from other software areas, are quite actively applied during mobile application development, but not quite correspond to their nature. Mobile applications differ from desktop and web programs [2]. Mobile applications are commonly used for short sessions, more focused on specific objectives performance.

Use of a suitable architectural pattern allows to increase considerably application development efficiency, but a bigger result can be achieved by graphic languages usage. DSL is the programming language in terms of the concrete subject domain which is applied to the solution of concrete type tasks [3], [4], [5]. Graphic DSL languages help to represent applications using visual diagrams. The result code will be generated according to these diagrams.

The purpose of this article is to develop an architectural template for mobile applications and to create graphic DSL based on it. DSL should allow describing the main logical application structure in terms of states, controllers and transition conditions between them.

2. Tools

The Modeling SDK technology is used for the graphic DSL implementation [6]. Modeling SDK is the plugin for Visual Studio intended for visual domain-specific languages development. Visual DSL development happens in the following order. At first, the metamodel (the set of all syntactically correct diagrams) is developed and edited, the implemented classes are generated. Then a DSL package compilation and debugging take place in an experimental instance of Visual Studio.

For metamodel programming, the graphic editor of Modeling SDK is used, but also it is possible to redefine or add new methods to the generated partial classes of the C\# language. The T4 language is used for code generation [7]. The Dsl and DslPackage projects are automatically created in the new solution of Visual Studio. In the Dsl project, various metamodel artifacts of the created DSL are stored. DslPackage project contains the user interface settings.

3. Controllers and states model

An application state corresponds to some complete logic fragment. The result of state change is data transfer which is logically finished and clear to other states.

It is convenient to group states and transition conditions into controllers by their logical connectivity, data community, UI forms, transition frequency and data transfer between states. Grouping states into controllers gives an opportunity to define more strong transition logic, allowing transitions between states in the controller and forbidding them between conditions of unconnected controllers.

The main application cycle is run by the special mechanism starting and switching controllers of states. Each controller has an entry point and an opportunity to set input parameters when an

application switches to it. There can be several exits in a state. An application can return to the caused controller, switch to the next controller, etc. Execution logic is implemented in terms of the finite-state machine in the controller. Each controller has a set of the predefined states (in particular, initial and final states), and it is possible to add new states.

Mobile application implementation by means of controllers and states model allows to centralize its logical basis, the structure of the code becomes evident. When using controllers, the aim of the mobile application developer comes down to describing the necessary logical controllers, state and conditions of transition.

4. Graphic DSL Description

The model of controllers and states was tested on mobile applications of different classes and proved the efficiency. But the best results can be achieved, having taken this model as a basis of graphic DSL for mobile applications (see fig.1)

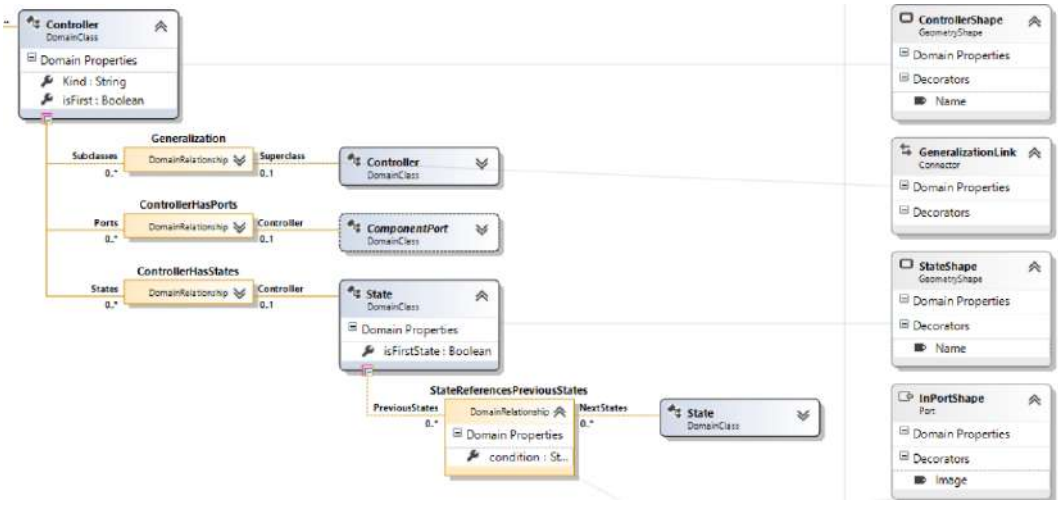

Fig. 1. Language implementation in Modelling SDK

Basic elements of the language are controllers and its states. States are placed on the controller, can connect among themselves and also to ports of the controller for the conditions description of an entrance and an exit from it. Each state opens in the separate diagram on which conditions of an entrance, an exit from a state and its internal logic are described. The logic of states includes a display of UI forms, processing of their events, services calls, conditions checking, etc. There is a display of a UI form for each state in the language. To connect the existing screen form with a state the ShowForm element is used.

5. Code generation

The language of T4 templates is used for code generation. The main components of the T4 language are directives, blocks of the text and control units. For a generation of the unchangeable code, text blocks are used, and dynamic parts are implemented by means of control units.

As a result of generation, the controllers' classes appear. Each controller has several states presented in the transfer type form. Process of work is implemented in terms of the finite-state machine. On links between states, the template of transitions are implemented. Controllers can also have ports. Ports are used for transitions between controllers.

The resulting code is applied to UbiqMobile platform [8]. UbiqMobile platform is aimed to cross-platform mobile development. The main features of the platform are that the business logic of all

applications is executed on the server. And mobile devices have only thick clients to represent the result of application work.

6. Samples

The purpose of the first sample is to display the train schedule for the user. The application consists of a single controller and two states. In start state, the user can choose departure and destination stations (see fig.2). After clicking on the button, the application will switch the current state from the first state to the second one (see fig.3).

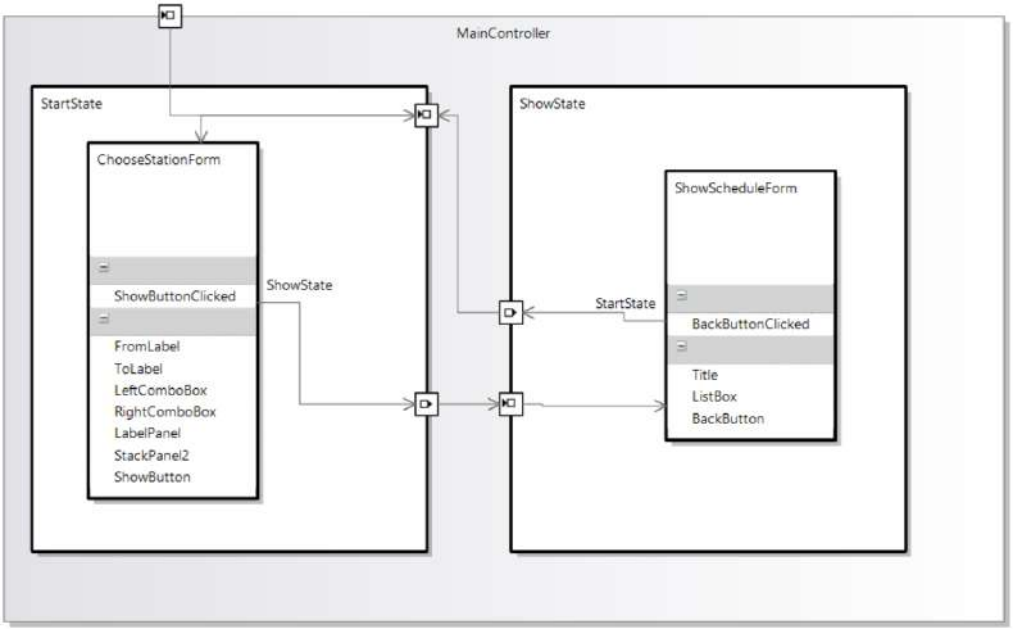

Fig. 2. Schedule application scheme

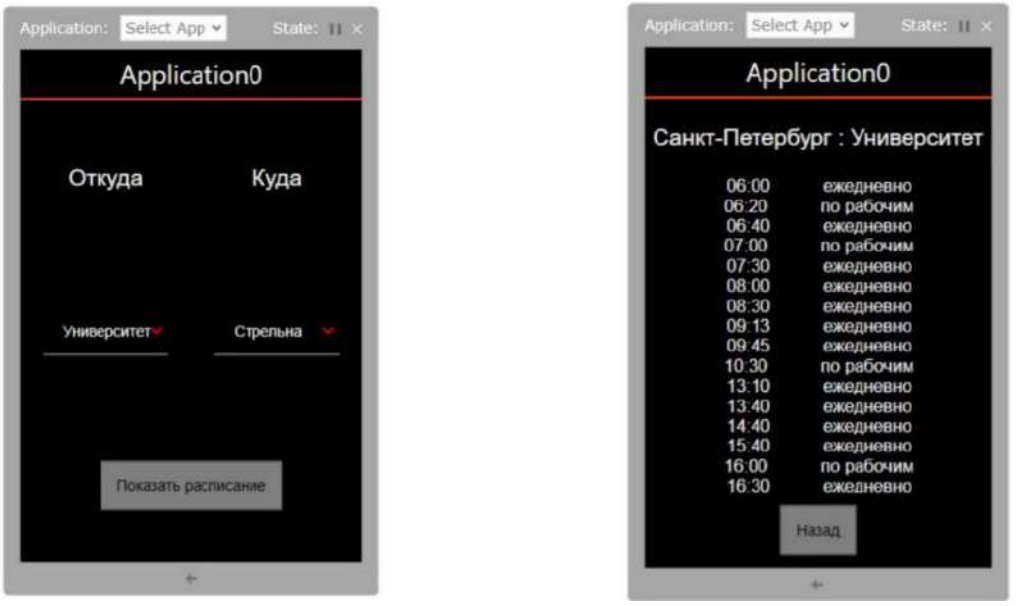

Fig. 3. Schedule application UI forms

The second sample allows the user to log in and receive the code which then can be used later (see fig. 4) There are two controllers in the application: LoginController and MainController. There is also a switching between controllers implemented by means of ports. In LoginController there is only one state. At MainController there are two states: a state with option selection and a state where a user can receive the necessary code. The UbiqMobile UI forms, corresponding to states of the application are given below (see fig. 5).

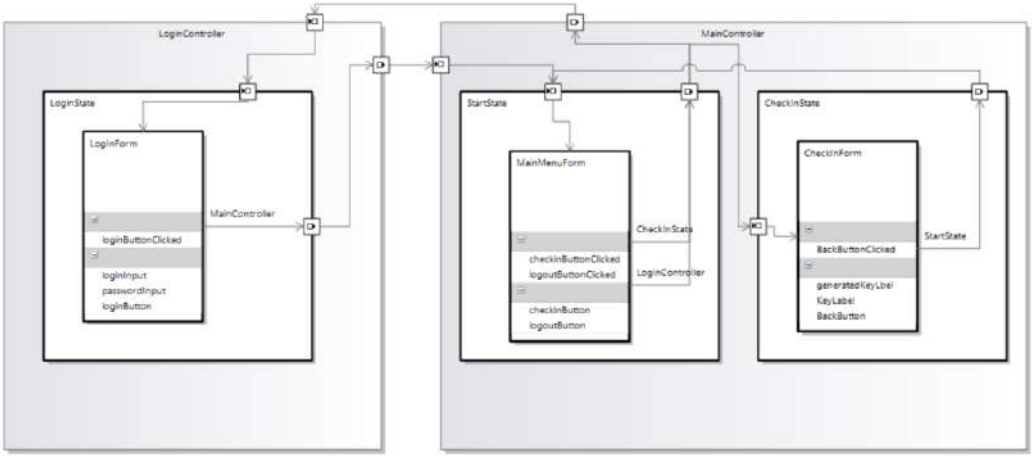

Fig. 4. Application with authorization scheme

Fig. 5. UbiqMobile screens

7. Conclusion

Within this work, the following results were achieved. The graphic DSL for mobile application development is implemented. The code generation for UbiqMobile platform feature is added. Demonstration samples are represented.

References / Список литературы

[1]. Plakalovic D and Simic D. Applying MVC and PAC patterns in mobile applications. arXiv preprint arXiv:1001.3489, 2010.

[2]. Flora Harleen K and Wang Xiaofeng and Chande Swati V. An investigation on the characteristics of mobile applications: A survey study. International Journal of Information Technology and Computer Science, vol. 6, issue 11, 2014, pp. 21-27.
[3]. Koznov D. Methodology and tools for domain-specific modeling. Thesis for the degree of Doctor of Technical Sciences, St. Petersburg State University, 2016 (in Russian) / Кознов Д.В. Методология и инструментарий предметно-ориентированного моделирования. Диссертация на соискание учёной степени доктора технических наук, СПбГУ, 2016.
[4]. Bryksin T.A. The platform for creation of specialized visual development environments of the software, PhD Thesis, St. Petersburg State University, 2016 (in Russian) / Брыксин Т.А. Платформа для создания специализированных визуальных сред разработки программного обеспечения. Диссертация на соискание учёной степени кандидата технических наук, СПбГУ, (2016).
[5]. Bryksin T.A. and Litvinov Yu.V. Environment of visual programming of QReal : Robots. In Proc. of the international conference on Information technologies in science and education, 2011, pp. 332-334 (in Russian) / Брыксин Т.А., Ю.В. Литвинов. Среда визуального программирования роботов QReal: Robots. Материалы международной конференции «Информационные технологии в образовании и науке», 2011, стр. 332-334.
[6]. Modeling SDK for Visual Studio – Domain-Specific Languages. Available at: https://docs.microsoft.com/ru-ru/visualstudio/modeling/modeling-sdk-for-visual-studio-domain-specific-languages, accessed 14.07.2019.
[7]. Code Generation and T4 Text Templates. Available at: https://docs.microsoft.com/ru-ru/visualstudio/modeling/code-generation-and-t4-text-templates?view=vs-2015, accessed 14.07.2019.
[8]. Onossovski V.V. and Terekhov A.N. Ubiq Mobile – a new universal platform for mobile online services. In Proc. of the 6th seminar of FRUCT Program, 2009, pp. 96-105.

Информация об авторах / Information about authors

Артур Владимирович ГУДИЕВ – магистр, выпускник кафедры системного программирования математико-механического факультета СПбГУ. Сфера научных интересов: мобильная разработка, предметно-ориентированное моделирование.

Artur Vladimirovich GUDIEV is a graduate of software engineering department of Mathematics and Mechanics faculty, St.Petersburg State University. Research interests: mobile development, domain-specific modelling.

Александра Сергеевна ГРАЖЕВСКАЯ – выпускница математико-механического факультета СПбГУ. Ее научные интересы включают предметно-ориентированные языки.

Alexandra Sergeevna Grazhevskaya is a graduate of the Mathematics and Mechanics faculty, St. Petersburg State University. Her research interests include domain-specific languages.

DOI: 10.15514/ISPRAS-2019-31(3)-3

Development of a software framework for real-time management of intelligent devices

T. Naumović, ORCID: 0000-0001-9849-7665 <tamara@elab.rs>
L. Baljak, ORCID: 0000-0003-3779-7335 <lukabaljak@elab.rs>
L. Živojinović, ORCID: 0000-0003-3536-3146 <lazar@elab.rs>
F. Filipović, ORCID: 0000-0001-7113-3802 <filipfilipovic@elab.rs>
Faculty of organizational sciences, University of Belgrade,
Jove Ilića 154, 11000 Belgrade, Serbia

Abstract. The subject of this paper is development of software framework for real-time management of intelligent devices. The framework enables intelligent management of IoT devices in cyber-physical systems using models based on recurrence relations and differential equations. The platform was developed using Python programming language, Django framework and wide corpus of modules and libraries that supports continuous simulation. The software framework incorporates application programming interface as well, for specification of system behaviour, transmission of input parameters and output results, sending control actions via web services to the IoT system.

Keywords: software framework; IoT; continuous simulation; python

For citation: Naumović T., Baljak L., Živojinović L., Filipović F. Development of a software framework for real-time management of intelligent devices. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 3, 2019. pp. 35-46. DOI: 10.15514/ISPRAS-2019-31(3)-3

Acknowledgments. Authors would like to thank prof. dr. Bozidar Radenkovic, prof. dr. Marijana Despotovic-Zrakic, prof. dr. Zorica Bogdanovic, prof. dr. Dusan Barac and prof. dr. Aleksandra Labus for guidance and mentoring throughout this research and software development.

Разработка программной среды для управления интеллектуальными устройствами в реальном времени

Т. Наумович, ORCID: 0000-0001-9849-7665 <tamara@elab.rs>
Л. Баляк, ORCID: 0000-0003-3779-7335 <lukabaljak@elab.rs>
Л. Живоинович, ORCID: 0000-0003-3536-3146 <lazar@elab.rs>
Ф. Филипович, ORCID: 0000-0001-7113-3802 <filipfilipovic@elab.rs>
Факультет организационных наук, Белградский университет,
11000, Сербия, г. Белград, Йове Илич 154

Аннотация. Предметом данной статьи является разработка программной среды для управления интеллектуальными устройствами в режиме реального времени. Платформа обеспечивает интеллектуальное управление устройствами Интернета вещей в кибер-физических системах с использованием моделей, основанных на рекуррентных соотношениях и дифференциальных уравнениях. Платформа была разработана с использованием языка программирования Python, инфраструктуры Django и широкого набора модулей и библиотек, поддерживающих непрерывное моделирование. Программная среда также включает интерфейс прикладного программирования для спецификации поведения системы, передачи входных параметров и выходных результатов, отправки управляющих действий через веб-сервисы для системы Интернета вещей.

Ключевые слова: программная среда; Интернет вещей; непрерывное моделирование; Python

Для цитирования: Наумович Т.**,** Баляк Л., Живоинович Л., Филипович Ф. Разработка программной среды для управления интеллектуальными устройствами в реальном времени**.** Труды ИСП РАН, том 31, вып. 3, 2019 г., стр. 35-46 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(3)-3

Благодарности. Авторы хотели бы поблагодарить проф., др. Божидара Раденковича, проф., др. Марияну Деспотович-Зракич, проф., др. Зорицу Богданович, проф., др. Душана Барака и проф., др. Александру Лабус за руководство и наставничество на протяжении всего исследования и разработки программного обеспечения.

1. Introduction

Cyber-physical systems (CPS for short) integrate devices, networks, interfaces, computer systems, and others with physical world. The fact that those elements are heterogeneous, hybrid, distributed and numerous, makes their analysis, design and implementation quite challenging and complex. In addition, CPSs are real time by their nature. Wide corpus of services, applications and interactions within CPS as well as huge growth of Internet of things further fuelled the need to change and improve existing approaches to managing those systems [1][2]. One of the most significant issues is to explore and model properties of CPS' elements, their connections and behaviour [3][4]. CPS immerged from the integration of devices with embedded systems, smart objects, people and physical environment typically connected via communication structure. Thus, it is no surprise that smart environments and systems are among the fields of CPS application.

Smart systems are integral part of CPS. The key technology for developing cyber physical systems is Internet of Things, IoT [5][6][7]. According to [8], cyber-physical systems, Internet of things and big data are related concepts of cooperative solutions, where people, autonomous devices and the environment interact with one another to achieve a certain goal. IoT technologies enable the connection of a large number of users, devices, services and applications to the Internet [9].Management of intelligent devices often needs to be done in real-time. Real-time Control System (RCS) is a reference model of architecture that defines the types of functions needed for intelligent real-time control [10]. RCS provides a comprehensive and basic methodology for design, engineering, integration and testing of control systems [11].

In RCS systems, the state of many variables changes continuously over time, so the management of these systems can be modelled using differential equations and recurrence relations. Hence, simulation enables investigation of behaviour of such dynamic systems by developing appropriate models and using these models in experiments designed to provide an insight into the future behaviour of the system under specific conditions [12][13][14]. Simulation of CPS is becoming extremely important for both academia and practice as results of simulations have huge potential to be applied in research, business and engineering. [15].

The main idea of the research is to develop a comprehensive platform that would enable modelling and simulation of different smart environments. To achieve this goal, it is vital to define a uniform formal model applicable to any smart environment whose mathematical representation can be mapped to its implementation as one-to-one correspondence. The software framework for real-time management of intelligent devices represents a cyber-physical system incorporating IoT devices as the physical component of the system and software framework accompanied with required network infrastructure as the cyber component. Having available information and input data from intelligent devices in real-time allows the simulation engine, as an integrated part of the solution, to calculate and create a plausible outcome. On the other hand, outcome created as the result of the simulation can be a trigger dispatching control actions towards the IoT system.

The formal model, implementation and example illustrating the applicability of the presented mathematical model will be explained further in the paper.

2. Formal Model

2.1 Continuous system simulation in IoT context

Continuous system simulation refers to experimenting with models whose states are changing continuously in time [11]. These types of simulation systems are often described by differential equitation. Time is independent variable in most cases. Continuous simulation can be used in different contexts and covers numerous types of real-world problems [16]. Considering time as an independent variable, digital computer has constraints solving differential equations, which is why it was necessary to develop a specific language to resolve this issue.

Different specialized languages for continuous simulation were developed, such as: CSMP (Continuous Simulation Modelling Programme) ESL (European Simulation Language) ACSL (Advanced Continuous Simulation Language) CSSL4 (Continuous System Simulation Language, Simulink, Matlab, Modelica and others that have been developed to simplify modelling, and to minimize problems related to programming continuous systems [16]. However, a majority of simulation tools have limitations related to low level of flexibility and adaptability, high costs, platform dependence, maintenance difficulties, etc. [16]

CSMP/FON platform for continuous system simulation was developed following these principles [17]:

- minimize required hardware resources and improve speed of execution;
- suitable and easy to use for educational purpose;
- simple and rich user interface;
- support for scientific research;
- saving costs.

The CSMP/FON is an open source solution and can be downloaded from the web site https://elab.fon.bg.ac.rs/softver/csmp. It has been used for many years in research and teaching within simulation related courses at University of Belgrade.

Software framework for real – time management of intelligent devices and IoT systems in general is a time dependent system, which requires a tool that can overcome any time – related performance issues. Ergo, using CSMP simulation logic in the software development process was a way of introducing control mechanism in the system.

2.2 Formal model of a continuous simulation system

Formal model of a continuous simulation system can be given as a tuple [18]:

$$M = (U, Y, S, \delta, \lambda, S0) \qquad (1)$$

with the following meanings:

- U – set of inputs;
- Y – set of outputs;
- S – set of state variables;
- δ – transfer function: $\delta: U \times S \to S$;
- λ – output function: $\lambda: U \times S \to Y$;
- $S0$ – set of initial states.

Function of a variable φ is a mapping of a non-empty set X, of variables x, signed as domain, in non-empty set Y, of variables y, signed as scope (or codomain, set of function values) [18]:

$$\phi: X \to Y;$$

a function of many variables is presented through mapping:

$$\phi: X \times X \times X \times ... \times X \to Y;$$

a block is presented as ordered set of three elements

$$b = (\phi, X, Y),$$

each $x \in X$ is input, while $y \in Y$ is output from the block.

The process of continuous simulation is based on solving differential equations and recurrence relations [17][18]. CSMP simulation language is block-oriented languages designed for solving systems of differential equations. Each block is specified by a set of inputs and parameters and a graphic symbol [12]. The graphic display of elements in the general form is presented in fig. 1

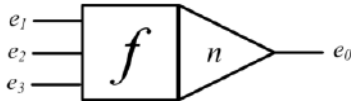

Fig 1. Graphic display of an element [17]

2.3 Formal model of a hybrid IoT system for real-time simulation

The current simulation model describes a system that allows solving differential equation systems in the given time with predefined variables and inputs [17][18]. The software framework for real-time management of intelligent devices requires a broaden model that will be suitable for use in real–time IoT systems [19].

Fig. 2 presents the concept of a hybrid IoT system for real-time simulation. This model enables having values measured in the environment in real-time included as variables of the simulation systems. In addition, the model enables managing the IoT system using variables obtained through the simulation.

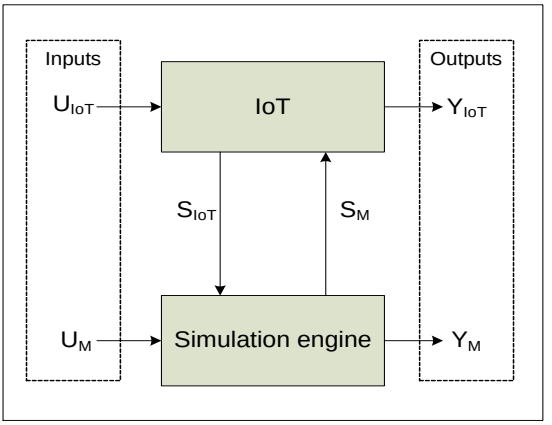

Fig 2. Hybrid IoT system for real-time simulation

For mathematical modelling of the hybrid IoT system for real-time simulation, the presented formal model needs to be extended with a set of state variables, inputs and outputs from the IoT system:

$$S = S_M \cup S_{IoT} \quad (2)$$
$$U = U_M \cup U_{IoT} \quad (3)$$
$$Y = Y_M \cup Y_{IoT} \quad (4)$$

In order to have the set of state variables S in the simulation model, it is necessary to get the values of state variables from the IoT system (S_{IoT}). This is done by developing and providing API of the IoT system. This API needs to implement the following functions:

$$\rho: S_{IoT}(t) \rightarrow S(t) \quad (5)$$
$$\omega: S(t) \rightarrow S_{IoT}(t) \quad (6)$$
$$\gamma: Y_{IoT}(t) \rightarrow Y(t) \quad (7)$$

The operation ρ is the operation of reading the values of variables from IoT system. These values then can be used in the simulation system for calculations. The operation ω enables writing the values of variables into the IoT system. These values are calculated in the simulation engine and then sent to the IoT system. These values can also be used for triggering specific actions of the IoT system. The operation γ enables reading the outputs of the IoT systems.

Having in mind that IoT systems are distributed, all these operations for interaction between the simulation and IoT systems need to be realized via web, using web services. Depending on the scenario, both PUSH and POP methods can be used.

After extending the formal model of continuous simulation system with the IoT elements, the process of continuous system simulation can be described with the finite automata equations [18]:

$$S(t) = I \times A_1 \cdot \{U(t), S(t)\} \quad (8)$$

$$Y(t) = A_2 \cdot \{U(t), S(t)\} \quad (9)$$

where A_1 and A_2 are matrix representation of algebraic functions, and I is the matrix representation of the integration operator (fig 3).

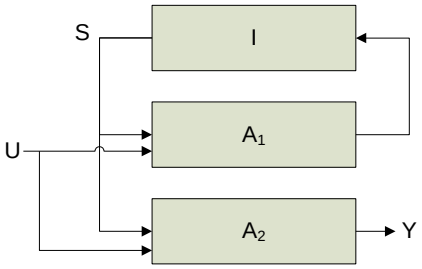

Fig 3. The structure of finite automata for simulation of continuous systems [18]

A more granular structure of continuous system simulation is presented in the fig. 4.

Fig. 4 depicts the decomposition of the operator A_1 to its elementary and primitive functions, represented as algebraic blocks. As explained later in section D, input of every block is an element that can come from either a set of inputs, a set of state variables or a set of the associated variables that represent inputs of the preceding algebraic blocks.

2.4 Orderliness

The essential feature of any non-trivial mathematical model of the continuous simulation is the feedback. The feedback occurs in the model as a result of a chain of cause-and-effect that generates a loop [20]. Considering the case of continuous simulation the model develops the feedback loop if it is impossible to mark all blocks from the set that satisfy a condition $i < j$, where block's b_i output is connected to block's b_i input [17]. The feedback loop imposes a compulsory requirement for computability of mathematical model called orderliness, defined as:

The set A of all countable algebraic blocks (blocks that correspond to algebraic functions) of M models is called orderly if all distinct objects $a_j \in A$ can be ordered (sorted) in such linear list where inputs of every distinct object a_j are elements of some of the following sets [18]:

1) U – set of inputs;
2) S – set of state variables;
3) Subset $C' \subset C$ defined as: $C' = \{c \in C \mid \forall i < j, \forall a_j = (\varphi_i, X_i, Y_i): c \in X_i\}$.

3. Mapping mathematical model to implementation

Mapping the mathematical method given in equations 1-9 is represented through series of UML sequence diagrams, where each method has its corresponding diagram.

The implementation of the software framework described through this research will be based on the concurrent computing and NoSQL concepts, such as threads and use of the MongoDB document-oriented database program.

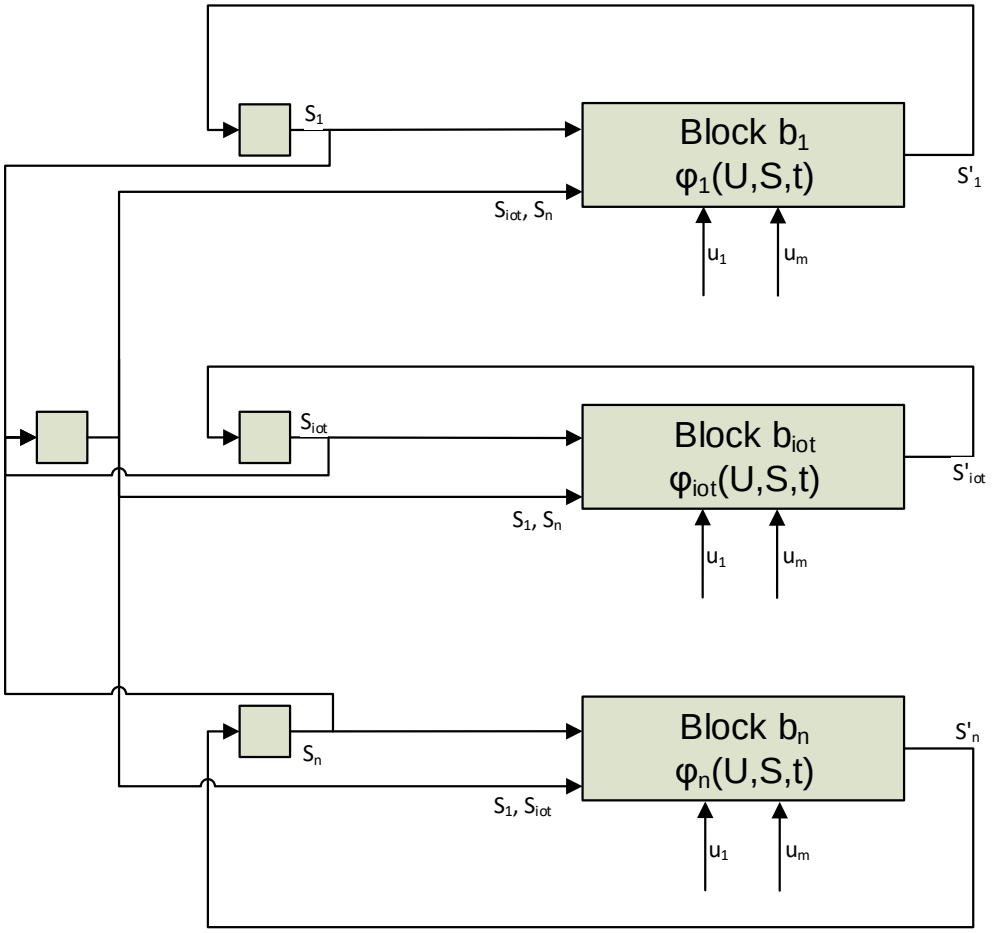

Fig 4. Block diagram of granular structure of continuous systems

3.1 Simulation engine

Fig. 5 illustrates the core simulation process depicted in equations (8) and (9). The diagram represents a typical continuous simulation process: begins with loading the simulation object from the MongoDB database in the engine, sorting the blocks, setting the primary conditions and starting the calculation process.

The calculation process itself is a looped process where series of computations are performed on every block in the simulation model: block type analysis, output generation through block function, output appending and call for next computation. The block type analysis determines if the current block is an IoT block, if it is engine provides a call to the IoT service, which performs specific operations based on the type of the call. Call types can be divided into two groups: a) reading and b) writing.

The call is a software representation of functions described through equations (5), (6) and (7). Depending on the call type, the simulation system will process data sent from the IoT system and

embed them as a part of the continuous simulation process, it will send a control action to the IoT system as a result of the continuous simulation process or it will read the output from the IoT system (fig 6).

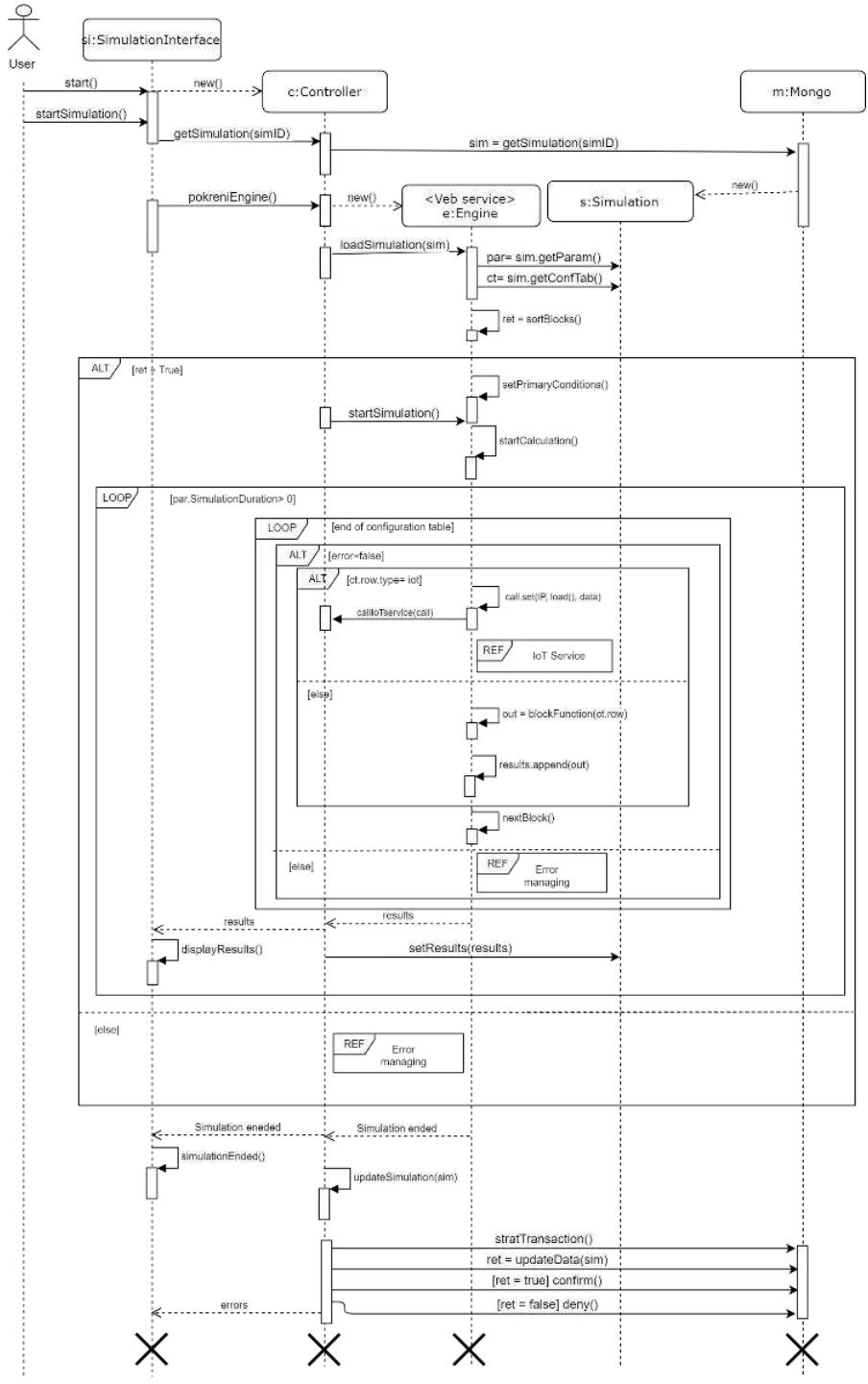

Fig 5. UML sequence diagram of the core simulation process

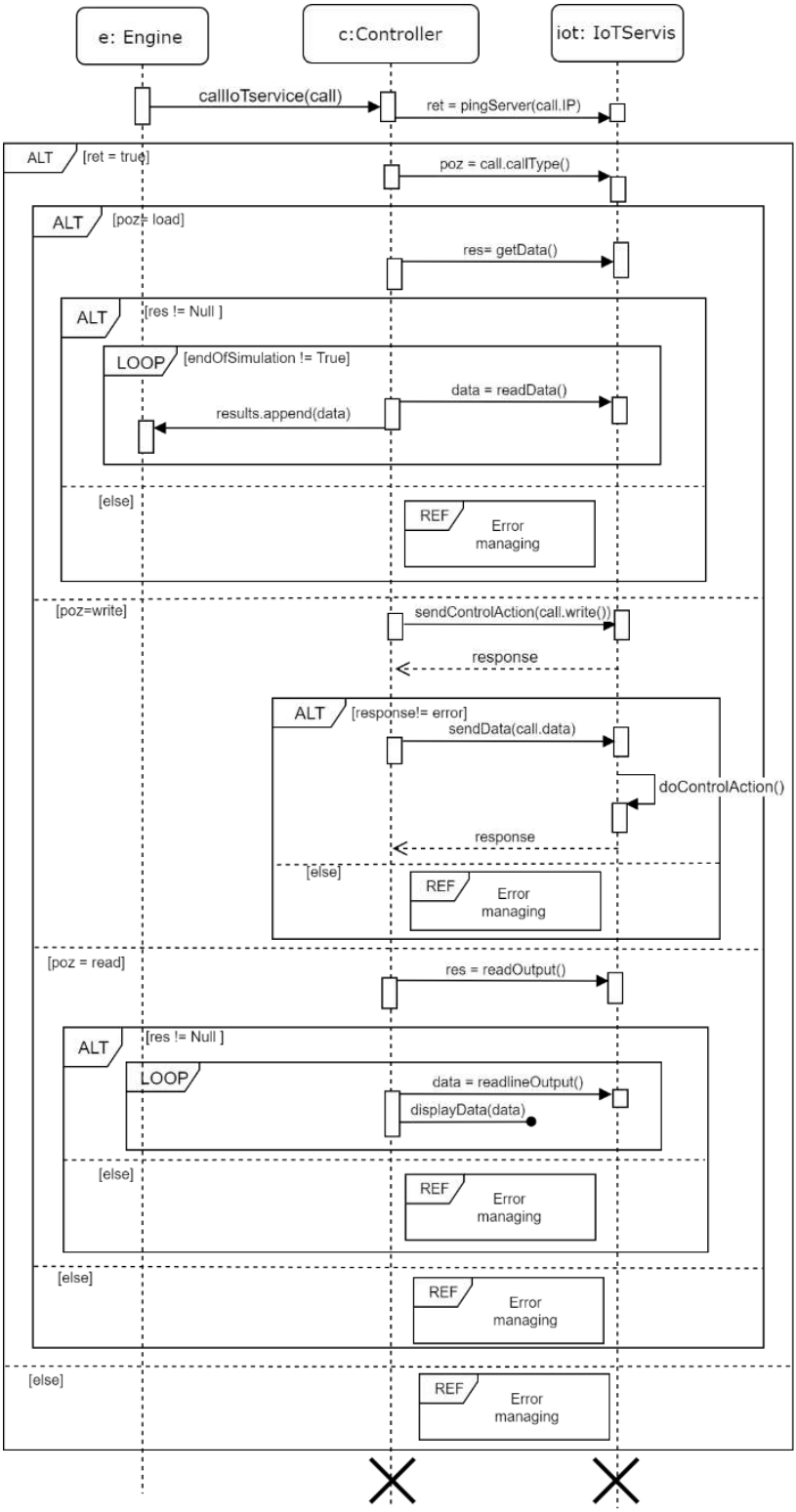

Fig 6. UML sequence diagram of processing calls to IoT system

3.2 Mapping the values of state variables and output from the IoT system

The call for executing operation ρ (5) – reading the values of variables from IoT system, is illustrated in the fig. 5 as a part of the calculation process, where simulation engine should consider IoT values as part of the calculation.

The control actions, ω (6) – writing the values of variables into IoT system, sent to the IoT system are, also, a type of call. By connecting to the IoT system, the engine can access its API, create a call to the function provided by the user, send data from the simulation engine and thus begin the given process on the IoT platform. Such call is illustrated in the fig. 7.

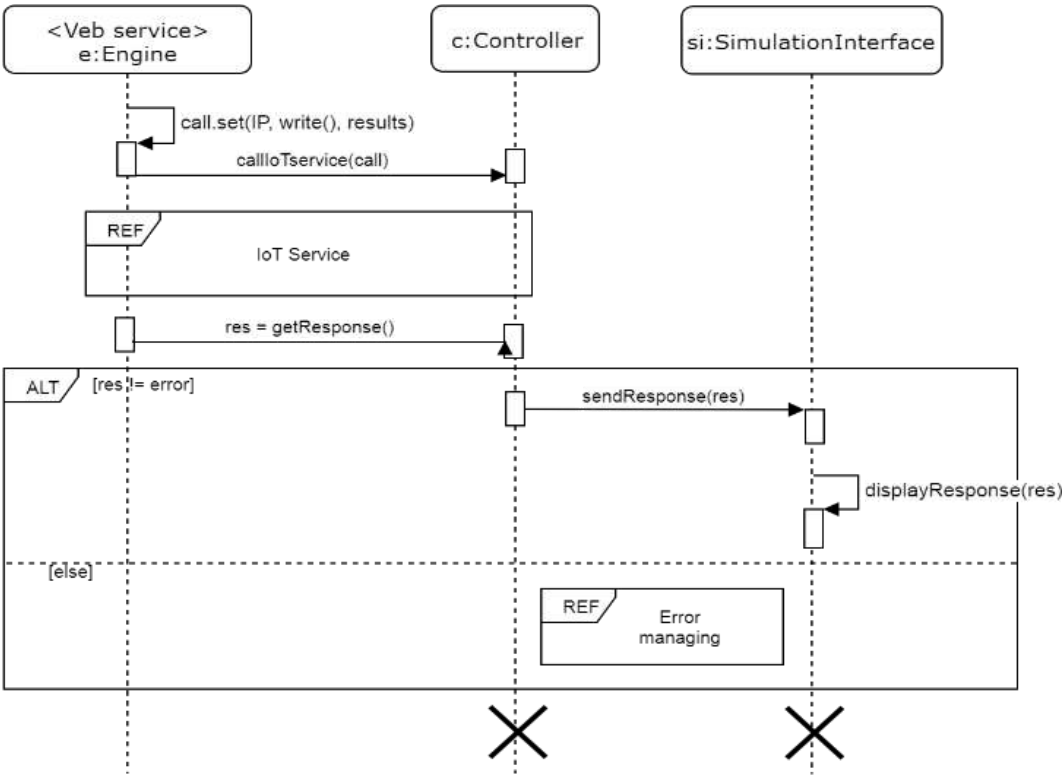

Fig 7. UML sequence diagram of the operation ω – writing the values of variables into IoT system

Through the connection made to IoT system our engine can retrieve IoT system outputs and display them though the platform interface, which is directly correlated with mathematical operation γ (7) – reading the outputs of the IoT systems (fig 8.).

3.3 Example: Smart watering system simulation

The example of smart watering system simulation is an illustration of the operation ω (6), where control actions and variables are being sent to IoT system.

For this example it is necessary to create control actions that will forward the data collected through the simulation of the environment and air humidity by the simulation engine, and signal the beginning of the IoT system actions.

Smart watering system is based on air humidity predictions, provided as input parameters given by the simulation engine. If the humidity is under the marginal value set in the IoT system, the watering process begins.

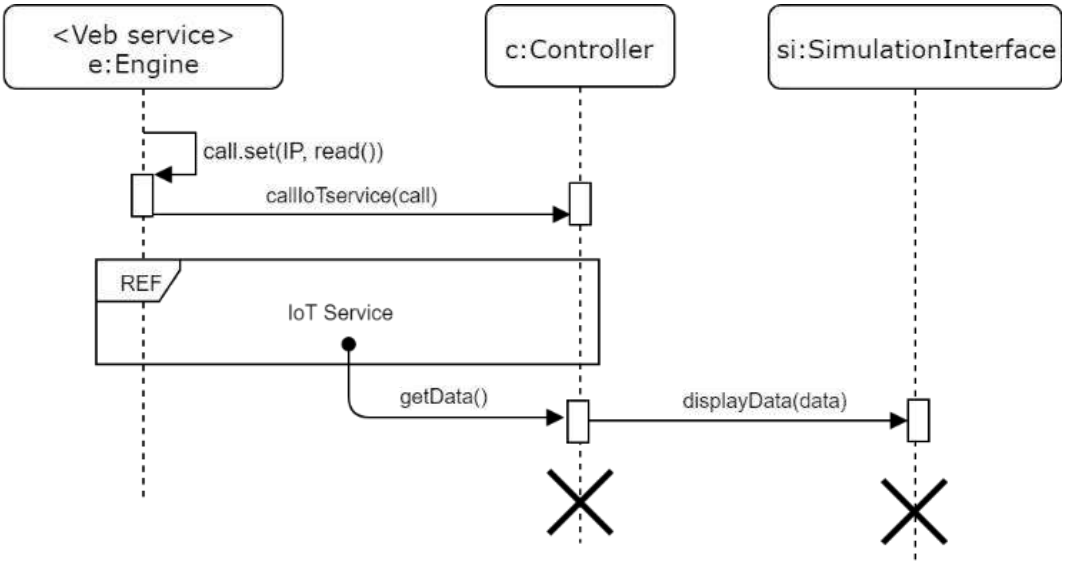

Fig 8. UML sequence diagram of the operation $\boldsymbol{\gamma}$ *– reading the outputs of the IoT systems*

4. Conclusion

Modelling hybrid IoT system for real – time simulation presents a focal point of this research. Thus, successfully mapping the values of state variables from the IoT system in the implementation process is essential.

The autonomous performance of the simulation program should be implemented using the concepts of concurrent computing – threads:

1) servicing requests for the simulation process control and error reporting,
2) servicing requests for configuration changes,
3) reading data and sending control actions to IoT system,
4) servicing requests for simulation results,
5) execution of the simulation process

Further research and work should be directed towards execution of the proposed implementation, integration of the platform in the students' educational process and evaluation and revision of the software performance. Upgrading the existing model with new modules should be considered.

References

[1] E. A. Lee, S. A. Seshia. Introduction to Embedded Systems. A Cyber-Physical Systems Approach, 2017, Second Edition.
[2] Y.Z. Lun, A. D'Innocenzo, F. Smarra, I. Malavolta, M. Benedetto, Maria. State of the Art of Cyber-Physical Systems Security: an Automatic Control perspective. Journal of Systems and Software, vol. 149, 2018, pp. 174-216.
[3] N. Canadas, J. Machado, F. Soares, C. Barros, L. Varela. Simulation of cyber physical systems behaviour using timed plant models. Mechatronics, vol. 54, 2018, pp.175-185.
[4] J. Liu, J. Lin. Design Optimization of Wireless HART Networks in Cyber-Physical Systems. Journal of Systems Architecture, vol. 97, 2019, pp. 168-184.
[5] K. Carruthers. Internet of Things and Beyond: Cyber-Physical Systems. 2016. Available at: https://iot.ieee.org/newsletter/may-2016/internet-of-things-and-beyond-cyber-physical-systems.html.
[6] L. Tan, N. Wang. Future Internet: The Internet of Things. In: Proceedings of 3rd International Conference on Advanced Computer Theory and Engineering, vol. 5, 2010, pp. 376- 380.
[7] M. Wu, T. J. Lu, F. Y. Ling, J. Sun, H. Y. Du. Research on the architecture of Internet of Things. In: Proceedings of 3rd International Conference on Advanced Computer Theory and Engineering, vol.5, 2010, pp. 484-487.

[8] S. F.Ochoaa, G. Di Fatta. Cyber-physical systems, internet of things and big data. Future Generation Computer Systems, vol. 75, 2017, pp. 82-84.
[9] B. Radenković, M. Despotović-Zrakić, Z. Bogdanović, D. Barać, A. Labus, Ž. Bojović. Internet inteligentnih uređaja [Internet of intelligent devices]. Beograd: Fakultet organizacionih nauka, 2017 (in Serbian).
[10] J. S. Albus. A Reference Model Architecture for Intelligent Systems Design. Springer, 1993. Available at: https://web.archive.org/web/20080916153507/http://www.isd.mel.nist.gov/documents/albus/Ref_Model_Arch345.pdf
[11] F. E. Cellier, E. Kofman. Continuous System Simulation. Springer-Verlag, First Edition, 2006.
[12] J. Banks. Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice. John Wiley & Sons, 1998.
[13] J. S. Keranen , T. D. Raty. Model-based testing of embedded systems in hardware in the loop environment. IET Software, vol. 6, no. 4, 2012, pp. 364-376.
[14] N. L. Celanovic, I. L. Celanovic, Z. R. Ivanovic. Cyber Physical Systems: A New Approach to Power Electronics Simulation, Control and Testing. Advances in Electrical and Computer Engineering, vol.12, no.1, 2012, pp.33-38.
[15] P. Garraghan, D. McKee, X. Ouyang, D. Webster, J. Xu. SEED: A Scalable Approach for Cyber-Physical System Simulation. IEEE Transactions on Services Computing, vol. 9, no. 2, 2016, pp. 199-212.
[16] M. Despotović-Zrakić, D. Barać, Z. Bogdanović, B. Jovanić, B. Radenković. Software Environment for Learning Continuous System Simulation. Acta Polytechnica Hungarica, vol. 11, no 2, 2014, pp. 187-202.
[17] B. Radenković. Program za simulaciju kontinualnih i diskretnih Sistema CSMP/MICRO [Program CSMP/MICRO for simulation of continuous and discrete systems]. Automatika, vol. 25, 1984, pp. 235-238 (in Serbian).
[18] B. Radenković, M. Stanojević, A. Marković. Chapter 6. Simulacija kontinualnih sistema. Racunarska Simulacija [Simulation of Continuous Systems. Computer Simulation], 2009, IV edition, pp. 89-110 (in Serbian).
[19] T.Naumović, B. Radenković, M. Despotović-Zrakić, D. Barać, A.Labus. A framework for real-time management of intelligent devices: an educational perspective. International Conference on New Horizons in Education, Proceedings Book, Volume 1, 2018, pp. 33-34
[20] Ford. Modeling the Environment: An Introduction To System Dynamics Modeling Of Environmental Systems, Second Edition, 2010.

Информация об авторах / Information about authors

Тамара НАУМОВИЧ получила степень магистра в 2018 году и в настоящее время готовит диссертацию на кафедре электронного бизнеса факультета организационных наук Белградского университета, Сербия. Ее область интересов включает Интернет вещей, физическое и непрерывное моделирование систем, а также разработку программного обеспечения.

Tamara NAUMOVIĆ received the M.S. degree in 2018 and is currently pursuing the Ph.D. degree at the Department of e-business Faculty of Organizational Sciences, University of Belgrade, Serbia. Her research and field of interest include Internet of Things, physics and continuous system simulation, and software development.

Лука БАЛЯК – студент факультета организационных наук. Его исследовательские интересы включает Интернет вещей, краудсенсинг и мобильную разработку.

Luka BALJAK is a graduate student at the Faculty of Organizational Sciences. His research interest include Internet of Things, crowdsensing and mobile development.

Лазар ЖИВОИНОВИЧ – аспирант кафедры электронного бизнеса факультета организационных наук Белградского университета, Сербия. Его исследовательские интересы включают в себя языки моделирования и симуляции, агентное моделирование и электронное обучение.

Lazar ŽIVOJINOVIĆ is a Ph.D. student at the Department of e-business Faculty of Organizational Sciences, University of Belgrade, Serbia. His research interests include simulation and simulation languages, agent-based simulation and e-learning.

Филип ФИЛИПОВИЧ – студент факультета организационных наук. Его исследовательские интересы включает 3D-моделирование, Интернет вещей и нейромаркетинг.

Filip FILIPOVIĆ is a graduate student at the Faculty of Organizational Sciences. His research interest include 3D modelling, Internet of Things and neuromarketing.

DOI: 10.15514/ISPRAS-2019-31(3)-4

An Exploration of Approaches to Instruction Pipeline Implementation for Cycle-Accurate Simulators of «Elbrus»

[1] *P.A. Poroshin, ORCID: 0000-0003-0319-5184 <poroshin_p@mcst.ru>*
[1,2] *A.N. Meshkov, ORCID: 0000-0002-8117-7398 <alex@mcst.ru>*
[1] *INEUM, 24, Vavilova st., Moscow, 119334, Russia*
[2] *MCST, 1, Nagatinskaya st., Moscow, 117105, Russia*

Abstract. Software simulation is of a big importance during development of processors as they provide access to hardware under development. Cycle-accurate simulators allow software engineers to design and optimize high-performance algorithms and programs with considerations of features and characteristics of processors being in development. This is especially important for architectures, whose performance is mainly achieved by advanced compiler optimizations. One of the core aspects of a cycle-accurate simulator is the way it simulates the pipeline of the target processor. A pipeline model has high impact on an overall structure of a simulator and its potential performance and accuracy. The main goal of this paper is to develop and analyze different approaches to pipeline simulation of "Elbrus" microprocessors, which let us reuse functionality of existing instruction set simulator and achieve good balance of performance and accuracy. We briefly describe features of "Elbrus" microprocessors and specifics of existing instruction set simulator, relevant for cycle-accurate simulation. We make several simple, but general and useful observations about various aspects of pipeline behavior in context of accurate and efficient cycle-accurate simulation of microprocessors. These observations are then used as a basis for justification, development and analysis of the several approaches to the pipeline simulation, described in this paper. We describe four different approaches, starting from simple and obvious one, which is then successively transformed into more advanced ones through several iterations. We analyze limitations of proposed approaches and outline further work.

Keywords: Simulation; Pipeline; Cycle-Accurate Simulator; Microprocessor; Elbrus

For citation: Poroshin P.A., Meshkov A.N. An Exploration of Approaches to Instruction Pipeline Implementation for Cycle-Accurate Simulators of «Elbrus» Microprocessors. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 3, 2019. pp. 47-58. DOI: 10.15514/ISPRAS-2019-31(3)-4

Исследование подходов к реализации конвейера инструкций в рамках потактово-точного симулятора микропроцессоров «Эльбрус»

[1] *П.А. Порошин <poroshin_p@mcst.ru>*
[1,2] *А.Н. Мешков <alex@mcst.ru>*
[1] *ПАО «ИНЭУМ им. И.С.Брука», 119334, Россия, г. Москва, ул. Вавилова, д. 24*
[2] *АО «МЦСТ», 117105, Россия, г. Москва, ул. Нагатинская, д. 1, стр.23*

Аннотация. Программное моделирование играют важную роль в цикле разработки процессоров, так как они предоставляют доступ к еще не существующему оборудованию. Потактово-точные симуляторы позволяют разработчикам программного обеспечения создавать и оптимизировать программы с учетом особенностей и характеристик разрабатываемых процессоров, что особенно

важно для архитектур, которые для достижения высокой производительности в основном опираются на агрессивные оптимизации компилятора. Одним из ключевых аспектов потактово-точного симулятора является способ моделирования конвейера симулируемого процессора. Программная модель конвейера оказывает большое влияние на общую структуру симулятора и на его производительность и точность. Основной целью данной статьи является разработка и анализ различных подходов к моделированию конвейера микропроцессоров “Эльбрус”, которые бы позволяли переиспользовать функционал существующего функционального симулятора без его существенных изменений, и которые бы достигали хорошего баланса производительности и точности. Мы коротко описываем особенности микропроцессоров “Эльбрус” и детали существующего функционального симулятора, важные для потактово-точного моделирования. Мы делаем несколько простых, но достаточно общих и полезных наблюдений о поведении конвейера с позиции точного и эффективного потактово-точного моделирования микропроцессоров. Данные наблюдения используются в качестве основы для обоснования, разработки и анализа нескольких подходов к моделированию конвейера, описанных в данной статье. Всего мы описываем четыре различных подхода, начиная с простого и достаточно очевидного, и заканчивая более сложными, полученными после нескольких итераций совершенствований и усложнений на основе ранее сделанных наблюдений. Для каждого подхода мы анализируем его преимущества, недостатки и фундаментальные ограничения.

Ключевые слова: программное моделирование; конвейер; потактово-точный симулятор; микропроцессор; Эльбрус

Для цитирования: Порошин П.А., Мешков А.Н. Исследование подходов к реализации конвейера инструкций в рамках потактово-точного симулятора микропроцессоров «Эльбрус». Труды ИСП РАН, том 31, вып. 3, 2019 г., стр. 47-58 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(3)-4

1. Introduction

Software based simulation of hardware is a very important tool for development of computing systems. This tool is especially important for software design, as simulators can be used in place of actual still in development (or unavailable for other reasons) hardware. Also simulators can provide wide range of debugging facilities and other information about inner workings of a system being simulated.

One of the widely used classes of simulators is simulators of microprocessors. Different tasks have different needs, so there are simulators with various characteristics. Ones may be oriented at simulation performance; others are aimed at accuracy and precision.

Instruction set simulator (ISS) is a simulator of microprocessor that mostly models a program visible architecture state without considerations of microarchitecture specifics and timings. And while for many tasks this is enough, there is a need for simulators with much greater degree of accuracy that can be used for performance evaluation.

Cycle-accurate simulators (CAS) are such simulators. They are important tools for code efficiency estimation during development of performance critical software and optimizing compilers. Ability to debug performance of code is especially crucial for microprocessor architectures, which achieve high performance not by invisible to programmer microarchitectural features, but mainly by static planning of instruction execution by smart compiler. The «Elbrus» family of microprocessor architectures is such type of architectures.

Modern microprocessors achieve their high performance and clock frequency through use of pipelining. Every cycle-accurate simulator must somehow simulate this pipelining logic to achieve accuracy of its timings. The way a pipeline is represented in a simulator influences various aspects of a simulator, how its components interact and its overall design and characteristics. There are different ways to represent a pipeline and to model it.

In this paper we describe several approaches that were considered as a basis for implementation of the pipeline model during development of the cycle-accurate simulator of microprocessors belonging to the «Elbrus» family of instruction set architectures.

The remainder of this paper has following structure. Section 2 gives brief overview of the «Elbrus» instruction set architecture and describes an existing instruction set simulator used as base for the cycle-accurate simulator implementation. Section 3 formulates desired properties and requirements for the pipeline model being developed. Section 4 describes in detail several considered approaches to pipeline model organization and explains its discovered advantages and drawbacks. Section 5 gives brief evaluation of described pipeline models. Section 6 is dedicated to other approaches to pipeline simulation that can be found in literature. Section 7 gives concluding remarks and briefly describes plans for further work.

2. Prerequisites

In this section we give some details of the architecture being simulated and of the available instruction set simulator that influence some design decisions around the pipeline model implementation.

2.1 «Elbrus» Family of Instruction Set Architectures

The «Elbrus» family of instruction set architectures is VLIW (Very Long Instruction Word) type of architectures [1]. Performance of this type of architectures is achieved by extracting ILP (Instruction Level Parallelism) through packing in one instruction several sub-operations, which are executed by hardware in parallel. «Elbrus» microprocessors are in-order and have no support of speculative execution (at least in the traditional sense).

In case of the «Elbrus», the packing format is not fixed and there are many ways several sub-operations can be packed in an instruction. Each of these sub-operations can belong to different kinds of operations: arithmetic and logical operations, control flow operations, predicate calculations, memory accesses and so on. And, while generally sub-operations observe only effects of previous instructions, there are several possible interactions of sub-operations within one instruction, for example, in case of a predicated execution.

Another important consideration is the way pipeline stalls work. Firstly, it is worth noting, that in case one sub-operation stalls (for example, because its arguments is not ready yet), the whole instruction stalls, which is a natural result for a VLIW architecture. Secondly, which is more specific for the «Elbrus» architectures, there are a mechanism of prolonged stalls. In simple terms, in some cases (determined by a stall cause and a current pipeline stage) an instruction is not immediately stopped, but effectively after several cycles its results are discarded (as invalid) and it is returned several stages back for its repeated execution in hopes that the original stall will not occur again. This process affects not only the instruction that is not ready for execution, but also several instructions immediately after it. There are two types of such stalls: a 2-cycle one and a 4-cycle one. Moreover, it is possible for several such stalls to interleave, and for such situation there is special pipeline control logic.

Later in this paper we will refer to the pipeline stages of the «Elbrus» microprocessors by following names: F, D, B, R, E0, E1, E2 etc.

2.2 Instruction Set Simulator

Our cycle-accurate simulator was not developed completely from the ground up. An existing instruction set simulator for the «Elbrus» architecture was used as a basis and a starting point for the development of its cycle-accurate version.

This instruction set simulator supports wide range of the various «Elbrus» microprocessors of different architecture iterations via compile time configuration. It also supports both a user mode simulation (with emulation of system calls) and a full system simulation (with MMU logic, peripheral devices etc.). All of this is implemented in a shared code base.

An important feature of the instruction set simulator to consider is how it executes individual (wide) instructions. Execution is divided in two separate steps, conventionally called «read phase» and «write phase». The «read phase» prepares some intermediate data and is mostly side-effect free. Then the «write phase» uses this intermediate data to complete instruction execution. This way of organization of instruction execution greatly simplifies support of precise exceptions and of some interactions of sub-operations.

3. Requirements to CAS and Its Pipeline Model

There are multiple valid ways to implement a cycle-accurate simulator and its pipeline model, and each design have its trade-offs. Therefore, it is important to define scope and requirements to the cycle-accurate simulator being developed, including its pipeline model implementation. We define following requirements.

- Support of a user mode simulation. At this stage of development it is planned that the cycle-accurate simulator will be used mainly as a tool for debugging performance problems during software and compiler development. For such purposes a user model simulation are used.
- Code reuse with the instruction set simulator. The existing instruction set simulator implements major parts of the «Elbrus» microprocessors, and it would be wasteful to reimplement this functionality separately.
- Configurability. It should be possible to configure the simulator to support the various «Elbrus» microprocessors (like the instruction set simulator) and to enable or disable its different components (for example, for the sake of performance).
- Flexibility. It should be reasonable easy to support new features of next iterations of the «Elbrus» microprocessors. And also, when need arises, it should be possible to adapt the pipeline model for a full system simulation mode.
- Reasonable performance. The cycle-accurate simulator should not be too slow compared to the instruction set simulator. We aim at no more than tenfold slowdown.
- Reasonable accuracy. Of course, exact timing accuracy is not achievable. However, the pipeline model design should not prevent possibility of further accuracy improvement and support of various microarchitectural aspects.

Some of these requirements are conflicting, and we do not expect to simultaneously meet all of them fully, but to achieve some balance between them.

4. Pipeline Simulation of «Elbrus» Microprocessors

In this section we explore several approaches to the pipeline simulation and describe theirs advantages and disadvantages.

4.1 Naïve «Direct Correspondence» Pipeline Model

The first approach that we tried to implement was based on the simple idea of direct and faithful representation of the real pipeline stages in the simulator. These stages would be responsible both for the timing related logic and for the purely algorithmic logic of the corresponding instruction.

We implemented this approach by transforming the «read» and «write» phases of the instruction set simulator into functions representing pipeline stages. During this transformation the «read» and «write» phases were split in parts and the missing pipeline related logic was added to them. To meet the requirement of code reuse, we made code of the new cycle-accurate simulator as base, and implemented the original instruction set simulator by «glueing» stages together into the «read» and «write» phases and removing the pipeline related logic, all of this at compile time and through configuration. Processing of such pipeline model is straightforward.

- Iterate through each pipeline stage.

- For each stage determine which instruction is at this stage and execute functions corresponding to all of the sub-operations of this instruction.
- If there are no stalls - advance instruction to the next stage. Otherwise not advance and propagate stall as necessary. In case of the prolonged stalls simulate related pipeline control logic.

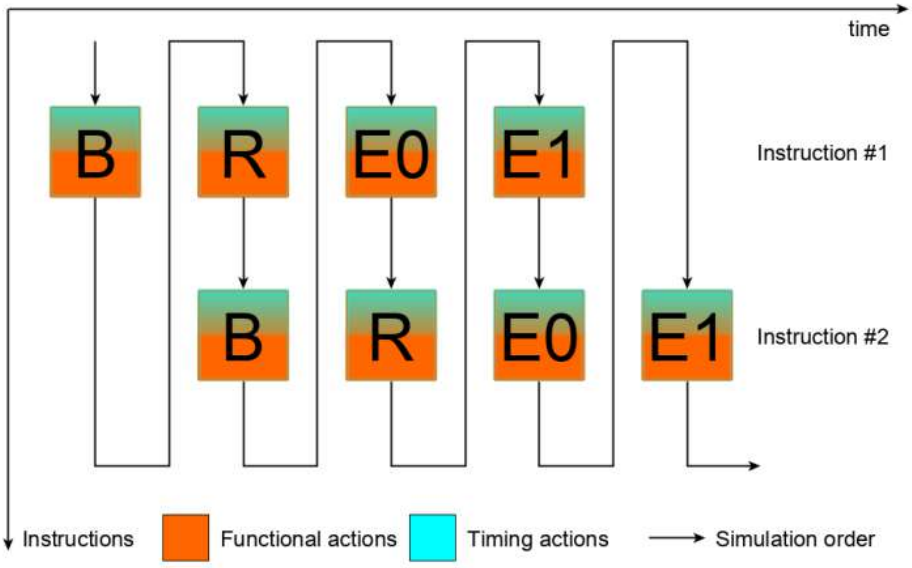

Fig. 1. Simplified illustration of pipeline stage processing in case naïve «direct correspondence» pipeline model

This pipeline model representation should facilitate direct and straightforward support of the various microarchitectural features, as this software model is close to the actual hardware. However, although this idea is conceptually simple, during its implementation we discovered its several major drawbacks.

- Splitting of phases of the instruction set simulator into stages and glueing them back together introduce a major disturbance to the original instruction set simulator functionality. There is no clear way to avoid that. Attempts to fully restore original phases introduce much ad hoc logic, which adds fragility to the whole system. This means there is no easy way to achieve code reuse with this approach.
- In the instruction set simulator there are many unobvious interactions between phases of different sub-operations. These interactions are not easily preserved during splitting of phases.
- While for the most of the operations there is a clear correspondence of phases to pipeline stages, there are exceptions, which add complexity to the glueing process.
- Keeping track of all pipeline stages adds considerable performance overhead, although for most operations only small subset of all pipeline stages are nontrivial (at least in the context of timings).
- Splitting phases into multiple pipeline stage related functions also inhibits compiler optimizations, which impact overall simulator performance.
- After this implementation attempt it became clear that for meeting our code reuse requirement we should minimize changes to the instruction set simulator.

4.2 Smart «Direct Correspondence» Pipeline Model

Next considered approach is a modification of previous one. Its improvements are based on the following key observations.

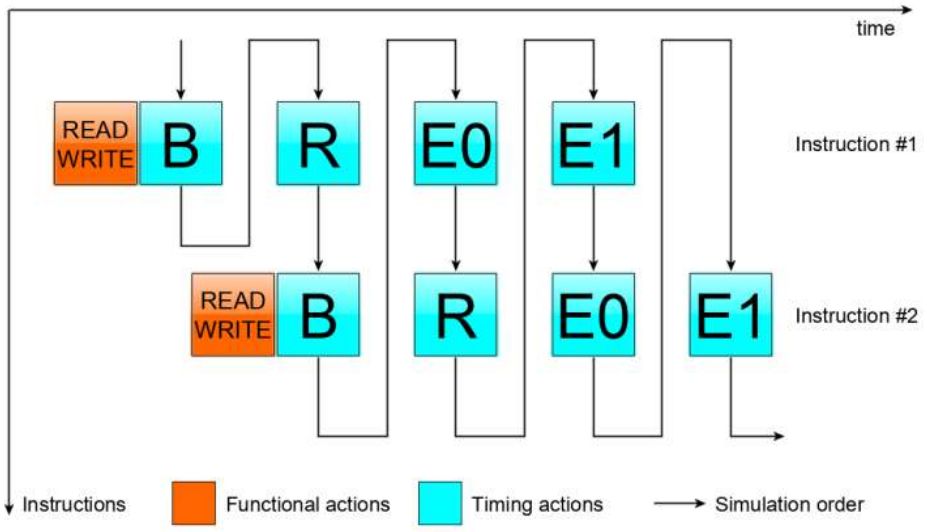

Fig. 2. Simplified illustration of pipeline stage processing in case smart "direct correspondence" pipeline model.

1) Algorithmic behavior of an operation (which is defined by an instruction set architecture and is considered by an instruction set simulator) can influence only an algorithmic behavior of operations of later (or in some cases current) instructions.
2) Algorithmic behavior of an operation determines its timing behavior.
3) Algorithmic behavior of one operation does not directly influence timing behavior of other operation.
4) Timing behavior has no direct influence on an algorithmic behavior (except in some limited number of special cases).
5) Timing behavior of one operation can influence timing behavior of other operation (but usually only in specific ways).
6) Simulator has more information about the execution process than hardware it simulates.
7) Not all details and inner workings of hardware contribute to its timing characteristics.

First six of these observations let us justify the separation of algorithmic and timing logic and moving of the algorithmic logic to the beginning of the instruction processing (right before its pipeline related processing). But we should uphold following conditions

- Algorithmic simulation of the instruction must occur before the algorithmic simulation of the next (in program order) instruction (based on the observation 1).
- Pipeline simulation of the instruction must occur after its algorithmic simulation (based on the observation 2).
- Pipeline simulation of different instructions must occur in order determined by the pipeline state (based on the observation 5).

All of these are satisfied by this approach.

Last observation let us simplify the timing logic by removing all microarchitectural details that are not directly necessary for correctly calculating timing information, as we are interested not in inner workings of hardware, but in timing details.

These transformations should not reduce accuracy of our simulator (except in some rare special cases, which are briefly considered later in this paper).

The algorithm to process such pipeline is very similar to the previous approach. The only difference is that in the beginning of the processing of the first pipeline stage of the instruction we do all algorithmic simulation of this instruction.

This approach let us use functionality of the instruction set simulator (for the algorithmic simulation of instructions) with minimal modifications, which remedy many major drawbacks of the previous approach. But we still have to address the performance concerns, as in this approach the simulator still keeps track of all pipeline stages, even if they are trivial, and the timing logic is still split into multiple independent functions.

4.3 «Fully Speculative» Pipeline Model

The next approach to the pipeline simulation is based on the assumption of stronger the observation 5:

5*) Timing behavior of operation of one instruction can influence timing behavior only of operations of the same or next instructions.

With this modified observation first five observations can be summarized as follows.

- Behavior (algorithmic and timing) of an operation of an instruction cannot depend on the behavior (algorithmic or timing) of operations of next instructions.

This assumption let us simulate all of the instruction's behavior in one go before even considering next instructions. It is just necessary to remember all effects (algorithmic and timing) of the instruction that can influence next instructions. And this is what we do in this approach.

The simulation of pipeline in this approach is as follows:

- Simulate algorithmic behavior of the instruction using the instruction set simulator functionality.
- «Speculatively» simulate timing behavior of the instruction by processing each of its nontrivial stages one by one from first to last, remembering in the process all information about produced effects and their moments in time for use by next instructions (at the same time using such information from previous instructions).
- Move to the next instruction.

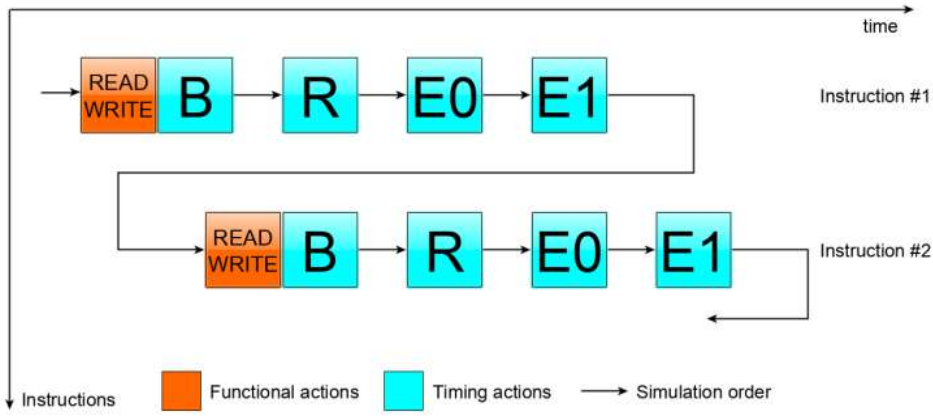

Fig. 3. Simplified illustration of pipeline stage processing in case "fully speculative" pipeline model

Such pipeline organization is expected to be more performant, as it has less overhead related to keeping track of the individual pipeline stages, better processes trivial stages, and in general has more optimization opportunities.

At the same time, with this approach it is necessary to transform the pipeline representation in the new form that supports «speculative» accumulating of effects. This was possible in our case, but may be difficult to achieve in others.

Also, such pipeline model is more complicated and unintuitive. For example, it has no reasonable notion of the current moment in (simulation) time. Time becomes in some sense distributed around the whole pipeline model.

Pipeline is not sole contributor to timing behavior, and it must interact with other components of microprocessor, such as L1 and L2 caches, IB (Instruction Buffer, the component responsible for the fetch of instructions) and others. It may be unfeasible to simulate these components in such «speculative» fashion, and the only reasonable way is the cycle-by-cycle type of simulation. And without a clear «current moment» concept, it is not obvious, when such cycle-by-cycle simulation must occur.

Let us consider L1 cache as a concrete example. Its cycle-by-cycle simulation must occur after all its inputs are available but before its results can influence simulation of the other components (including the pipeline). After careful study of possible interactions of the L1 cache model and the pipeline model we identified that such cycle-by-cycle simulation should occur right after the simulation of the stage R of the instruction. By similar reasoning the cycle-by-cycle simulation of the IB should be placed right after the simulating of the stage F of the instruction. Additional considerations must be made in case of stalls, but overall idea is the same.

Now let us consider interactions between the IB and the L1 cache. In principle, it is possible to the IB request of the future instruction to interfere with the L1 cache state observed by the current instruction. Therefore, it is possible to the timing behavior of the future instruction to influence the timing behavior of the current instruction, which is a violation of our earlier assumption. It means that in this approach we cannot accurately simulate some interactions between various microprocessor components.

Another example of violation of our assumption is the complex interactions during the interleaving of prolonged stalls, where stall of the next instruction can influence stall latency of the current instruction.

Overall, while this approach promises performance improvement, it sacrifices accuracy and flexibility and introduces additional complexity.

4.4 «Hybrid» Pipeline Model

The last approach to the pipeline simulation considered in this paper is a combination of second and third approaches. This pipeline model tries to retain accuracy of the smart «direct correspondence» model and to achieve some of the performance benefits of the «fully speculative» model. It is based on the two additional observations:

8) Pipeline behavior of an instruction interacts with pipeline behaviors of other instructions and other components at specific pipeline stages.
9) There are continuous sequences of stages that executed uninterrupted (without stalls and influence from other instructions and components).

For example, after the stage E2 there is no possibility of any stall and all further timing behavior of the instruction is predetermined. So it is possible to simulate such continuous uninterrupted sequences of stages speculatively in a manner similar to the «fully speculative» approach, but without the risk of decreasing timing accuracy. And after the instruction reached the pipeline stage E3, we can stop keeping track of it, as its timing behavior is completely simulated (partly normally and partly speculatively) at this point. This significantly decreases the pipeline simulation performance overhead and the overhead of dealing with trivial stages.

Processing of such pipeline model is very similar to the smart «direct correspondence» approach:

- Iterate through each pipeline stage.

- For each stage determine which instruction is at this stage.
- If it is a new instruction, then simulate its algorithmic behavior.
- If it is the first stage of an uninterrupted sequence, then speculatively simulate all stages of this sequence.
- If there are no stalls, then advance the instruction to the next stage. Otherwise not advance and propagate stall as necessary. In case of prolonged stalls simulate related pipeline control logic.

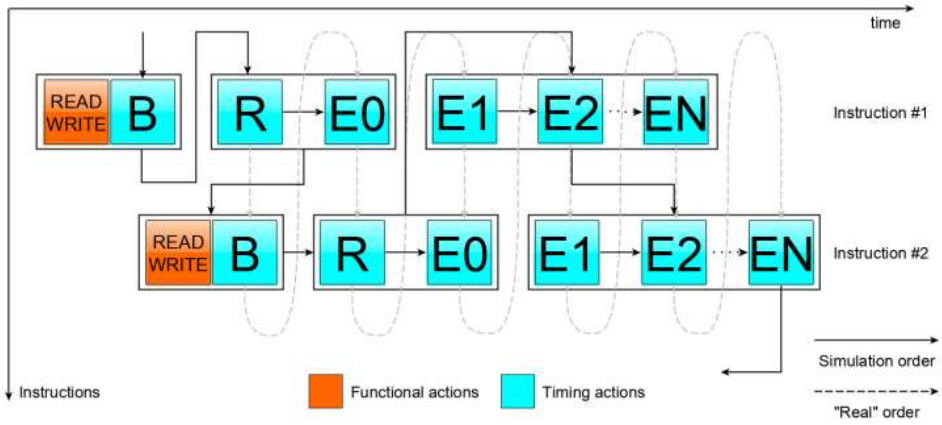

Fig. 4. Simplified illustration of pipeline stage processing in case "hybrid" pipeline model

Overall, this approach let us partially get performance gain of the «fully speculative» approach without its major drawbacks of sacrificing accuracy.

Unfortunately, all described approaches (except the naive one) do not cover the special case of the timing behavior influencing the algorithmic behavior. Example of such situation is operations that generate a predicate based on readiness of its arguments. Researching of ways to address this is part of our future work, and we hope it will be possible to implement a solution within the «hybrid» approach.

5. Evaluation

Although the cycle-accurate simulator is still in development and there is work to be done (for example, memory subsystems are not fully implemented yet and are greatly simplified), it is worth to do some preliminary evaluation of the pipeline model implementations described in this report.

Here we will consider only the «fully speculative» and the «hybrid» models, as the «direct correspondence» models were abandoned much earlier in the development and it is hard to make a fair comparison of them to the other models.

We compare the relative performance and the total number of the simulation cycles that were needed for the test completion. The instruction set simulator is used as a baseline. Individual test cases consist of the executing on the simulator part of one of the SPEC CPU95 benchmarks. Results presented in Table 1.

At this stage of the development we do not have a reasonable cycle count reference that we can use, because, for example, our simulators do not do proper simulation of various memory accesses. Nevertheless, we hope to get rough estimate of contribution of the more detailed simulation of the pipeline by the "hybrid" model to the total cycle count.

Results show that on average the «hybrid» model is slower than the «fully speculative» model by ~20%. At the same time, average difference in total cycle count is around 0.5% with one

significant outlier «146.wave5» with the cycle count difference of 6.1%. We expect that this is because less accurate simulation of the prolonged stalls in the «fully speculative» pipeline model.

It is possible to optimize both models and the performance difference after optimizations can change, but we expect that the «hybrid» model will always be slower. Despite this overall we consider the «hybrid» model as a better approach as it is more fully meets our requirements of accuracy and flexibility, and in a need of performance it should be possible to configure the «hybrid» model accordingly.

Table 1. Performance and total cycle count relative to instruction set simulator.

Test	«Hybrid» CAS		«Fully speculative» CAS	
	Relative Performance	***Relative cycle count***	***Relative Performance***	***Relative cycle count***
099.go	0,192	1,747	0,218	1,747
101.tomcatv	0,271	1,415	0,323	1,424
102.swim	0,329	1,983	0,407	1,981
103.su2cor	0,277	1,415	0,322	1,420
110.applu	0,218	1,999	0,266	2,001
124.m88ksim	0,243	1,151	0,299	1,151
126.gcc	0,291	1,376	0,341	1,378
129.compress	0,222	1,522	0,286	1,539
130.li	0,195	2,102	0,224	2,104
132.ijpeg	0,218	1,738	0,261	1,749
134.perl	0,252	1,541	0,301	1,553
141.apsi	0,248	2,364	0,286	2,379
146.wave5	0,328	1,734	0,398	1,840
147.vortex	0,250	1,600	0,308	1,601

6. Related Work

Cycle-accurate simulation of modern microprocessors is a very active area of research. But only small portion of this research is focused on simulating of general purpose VLIW microprocessors, let alone on the «Elbrus» architecture. And many of the available approaches do not quite translate to the «Elbrus» specifics.

Approaches of simulating a pipeline of VLIW microprocessors, similar to the «direct correspondence» approaches, are described in [2-4]. However, they do not address the issue of code reuse in the presence of an instruction set simulator.

All of the approaches described in this paper are execution-driven. Trace-driven simulation is one of the alternatives [5-9]. The basic idea of the trace-driven approach is a separation of the whole simulation process in two phases: generation of some data (trace), that represents an execution path, and using that data as an input for a cycle-accurate simulation of some microprocessor aspect. Trace can be generated by real hardware or other simulator (for example, an instruction set simulator). This approach gives benefits, similar to ones we aim to achieve by separation of algorithmic logic and timing logic introduced in our second approach, but makes extremely difficult to account for a possible dependence of an algorithmic behavior on a timing behavior (which we are planning to address in future work in our approach), as these interactions cannot be captured in trace during its generation before cycle-accurate simulation.

The pipeline representation, similar to our «fully speculative» approach, is used in [10]. Authors describe in details various aspects of the pipeline simulation (occupancy of stages, operand dependencies and control flow considerations), but do not discuss limits of this approach and complexities of interaction of such pipeline model with other components of microprocessor.

7. Conclusions and Future Work

Software based simulation of microprocessors is a very important tool. There are many possible ways to implement such simulators, each of them with its own set of advantages and disadvantages.

In this paper we explored several approaches to the pipeline simulation in the context of the cycle-accurate simulation of the «Elbrus» microprocessors. We made several simple, but general and powerful observations, which were used as the foundation for the design of the various pipeline models and for the analysis of their advantages and drawbacks. We described several of such approaches that were considered and at least partially implemented during development of our cycle-accurate simulator.

The cycle-accurate simulator described in this paper is still in active development. In the future work we are planning to address the issue of dependence of the algorithmic behavior of the instruction on the timing behavior and to explore additional ways to optimize performance of the simulation.

References / Список литературы

[1]. Kim A.K., Perekatov V.I., Ermakov S.G. Microprocessors and computing complexes of the Elbrus family. Piter, 2013, 272 p. (in Russian). / Ким А.К., Перекатов В.И., Ермаков С.Г. Микропроцессоры и вычислительные комплексы семейства «Эльбрус». Питер, 2013, 272 стр.

[2]. Cuppu Vinodh. Cycle accurate simulator for TMS320C62x, 8 way VLIW DSP processor. University of Maryland, College Park (1999).

[3]. Barbieri I. et al. Flexibility, Speed and Accuracy in VLIW Architectures Simulation and Modeling. In Proc. of the 2002 WSEAS International Conference on Electronics and Hardware Systems, 2002, pp. 1661-1665.

[4]. Barbieri I., Bariani M., Raggio M. A VLIW architecture simulator innovative approach for HW-SW co-design. In Proc. of the IEEE International Conference on Multimedia and Expo, 2000, vol. 3, pp. 1375-1378.

[5]. Uhlig R. A., Mudge T. N. Trace-driven memory simulation: A survey. ACM Computing Surveys, vol. 29, №. 2, 1997, pp. 128-170.

[6]. Joshua J. Y. et al. The future of simulation: A field of dreams. Computer, vol. 39, №. 11, 2006, pp. 22-29.

[7]. Agarwal A., Huffman M. Blocking: Exploiting spatial locality for trace compaction. ACM SIGMETRICS Performance Evaluation Review, vol. 18, №. 1, 1990. pp. 48-57.

[8]. Cho S. et al. TPTS: A novel framework for very fast manycore processor architecture simulation. In Proc. of the 2008 37th International Conference on Parallel Processing, 2008, pp. 446-453.

[9]. Lee H. et al. Two-phase trace-driven simulation (TPTS): a fast multicore processor architecture simulation approach. Software: Practice and Experience, vol. 40, №. 3, pp. 239-258.

[10]. Böhm I., Franke B., Topham N. Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator. In Proc. of the 2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, 2010, pp. 1-10.

Информация об авторах / Information about authors

Павел Александрович ПОРОШИН получил степень магистра в 2017 году в Московском физико-техническом институте. В настоящее время работает в ИНЭУМ им. И.С. Брука в качестве инженера-программиста. В область его научных интересов входят программное моделирование вычислительных систем и потактово-точное моделирование микропроцессоров.

Pavel Alexandrovitch POROSHIN received his MS degree from Moscow Institute of Physics and Technology in 2017. He is currently working as software engineer at INEUM. His research interests include software simulation of computing systems and cycle-accurate simulation of microprocessors.

Алексей Николаевич МЕШКОВ получил степень кандидата технических наук в ИНЭУМ им. И.С. Брука в 2013 году. В настоящее время является начальником отдела в АО «МЦСТ». Область научных интересов включает программное моделирование и верификацию компьютерных систем.

Alexey Nikolaevitch MESHKOV received his PhD degree at INEUM in 2013. He is currently working as a chief of department at MCST. His research interests include software modelling and verification of computer systems.

DOI: 10.15514/ISPRAS-2019-31(3)-5

Approach to test program development for multilevel verification

P.V. Frolov, ORCID: 0000-0002-9810-2210 <Pavel.V.Frolov@mcst.ru>
INEUM, 24, Vavilova st., Moscow, 119334, Russia
MCST, 1, Nagatinskaya st., Moscow, 117105, Russia

Abstract. Development of system-on-chips or network-on-chips requires verification of standalone units (peripherals and commutators) and a system as a whole. An approach to test development for verification of programmable standalone units is presented. The tests are written in C++ using a specific API to program the device-under-test (DUT) and the test environment. The API functions are implemented in the standard environment library; the specific implementation depends on the test environment structure: a standalone device, a device as a part of controllers block or a device as a part of the whole SoC. For system-level verification the test program is translated for execution on a general-purpose core of the verified SoC as well as the standard environment library. The testbench for unit-level verification consists of the environment library and the test linked to the testbench as a PLI-application, an adapter for the DUT-system bus interface and, possibly, a specific imitator of an external device. Different devices with one programming interface can be tested by the same test program even if they have different bus interfaces; different bus interfaces require different adapters to be implemented. The presented approach gives an opportunity to use the same test program both for standalone and for system-level verification (as an integration test). The implementation of the presented approach and its application to verification of microprocessors of the Elbrus family are described.

Keywords: hardware verification; simulation-based verification; test system; standalone verification; system-level verification

For citation: Frolov P.V. Approach to test program development for multilevel verification. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 3, 2019. pp. 59-66. DOI: 10.15514/ISPRAS-2019-31(3)-5

Разработка универсальных тестовых программ для автономной и системной логической верификации программируемых контроллеров

П.В. Фролов, ORCID: 0000-0002-9810-2210 <Pavel.V.Frolov@mcst.ru>
ПАО «ИНЭУМ им. И.С.Брука», 119334, Россия, г. Москва, ул. Вавилова, д. 24
АО «МЦСТ», 117105, Россия, г. Москва, ул. Нагатинская, д. 1, стр.23

Аннотация. При разработке систем-на-кристалле необходимо проводить верификацию как отдельных подмодулей (контроллеров периферийных интерфейсов и коммутаторов), так и системы в целом. В статье представлен подход к разработке тестов для верификации программируемых контроллеров. Тесты разрабатываются на языке программирования C++; программирование тестируемого устройства и тестового окружения осуществляется с помощью специального программного интерфейса. Функции этого программного интерфейса реализуются в стандартной библиотеке тестового окружения; реализация зависит от структуры тестового окружения: в качестве моделируемого устройства может выступать только тестируемый контроллер, контроллер в составе блока контроллеров, или контроллер в составе полной системы-на-кристалле. Для верификации системного уровня библиотека и тестовая программа компилируются для исполнения на одном из

вычислительных ядер системы-на-кристалле. При автономной верификации тестовая программа и библиотека окружения формируют программный модуль, взаимодействующий с симулятором RTL-описания с помощью стандартного интерфейса PLI; библиотечные функции взаимодействуют с моделируемым устройством через специальный адаптер системного интерфейса; кроме того, в тестовое окружение может быть включен имитатор внешнего устройства. При таком устройстве тестового окружения одна и та же тестовая программа может проверять устройства с одним программным интерфейсом, но разными системными интерфейсами; необходимо только реализовать соответствующие адаптеры. Представленный подход позволяет запускать тестовую программу как автономный тест, так и в качестве теста интеграции на верифицируемой системе-на-кристалле. В статье описаны реализация представленного подхода и его применение в маршруте верификации микропроцессоров семейства Эльбрус.

Ключевые слова: логическая верификация аппаратуры; верификация на основе моделирования; тестовая система; автономная верификация; системная верификация

Для цитирования: Фролов П.В. Разработка универсальных тестовых программ для автономной и системной логической верификации программируемых контроллеров. Труды ИСП РАН, том 31, вып. 3, 2019 г., стр. 59-66 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(3)-5

1. Introduction

Typical test scenarios for programmable standalone units (peripherals and commutators) are based on estimated work patterns of the designed chip operating. Such test scenarios are an indispensable part of a standalone verification testplan. They also must be included in a device integration test suite for system-level verification to check considered device interaction with other units.

This paper describes an approach to test development for verification of programmable standalone units which allows using the same test both for standalone and system-level verification. The presented approach also enables tests run in different *execution environments* (via an RTL simulator, an FPGA-based prototype or a manufactured chip).

The rest of paper is organized as follows. Section 2 reviews the existing techniques considering the same tests reuse for different execution environments. Section 3 introduces the structure of the framework for test development, implementing presented approach. Section 4 describes API provided by the framework for tests use. Sections 5 and 6 present test transformation for system-level and standalone verification respectively. In Section 7, results are presented and in Section 8, possible/planned future work is mentioned.

2. Related work

The main target of the presented approach is to reduce verification effort through the unit-level tests reuse for system-level simulation.

Review works on SoC verification suppose high level of the verification components reuse [1][2], but there is not much information about practical approaches for the test programs reuse. The problem of the stimulus reuse for different execution targets and environments is targeted by The Portable Test and Stimulus Standard (PSS) [3], but this standard provides only language for a test intent description [4].

Typical approach to unit-level verification is transaction-based verification, implemented, for example, in UVM (Universal Verification Methodoly) standard [5]. Such tests are written in SystemVerilog and commonly use constraint-random stimuli generation, implemented via external tools (RTL-simulator, for example). The reuse of such a test for system-level verification requires its additional adaptation. For example, the work [6] describes an approach which allows to get a system-level test based on the unit-level one for the separate IP-block (GPU) of the heterogeneous SoC. A trace of DUT interactions with the testbench is logged during unit-level simulation and then is compiled into assembly, ready for execution on the CPU at SoC level. The approach copes with register polling through the test driver library instumentation but DUT interrupts handling isn't described.

3. Test development framework

In the presented approach, a test is written in C++, so it can be translated to different host CPU architectures:

- to a PLI-application [7] (PLI is for Program Language Interface) interacting with a simulator modeling the RTL description of the standalone unit (or the block of controllers including this unit);
- for system-level execution on one of the general-purpose cores of the verified SoC.

The system-level test runs without an operating system and this restricts usage of standard C++/C library: no explicit usage of externally linked functions is allowed. Instead, the test development framework provides a common standard API for different test execution environments. The API is described in header files as a list of C++ function prototypes. For every supported test execution environment the framework provides a corresponding *environment library* implementing these functions.

Advantages of C++ as a test implementation language mainly address system-level test execution.

Firstly, C++ allows to transfer some calculations to the compilation stage via ***contexpr*** specifier (since C++11 [8] version of language standard). Secondly, C++-templates allow wrap of specific assembly instructions into ***inline*** functions to avoid function call overhead while preserving test portability. Besides, parts of device drivers or BIOS, commonly written in C, can be relatively simply ported for test use and vice-versa.

4. Environment library API

A typical programmable controller implements three kinds of interaction with a system: it provides access to the internal registers and memory for configuration (PIO, programmed input/output), can initiate DMA-transactions (Direct Memory Access) to the system memory and send interrupt messages. Thus the environment library API must provide means to perform, control and observe these interactions.

The API contains a description of typical operations:

- access to the registers and the internal memory of the device under test,
- system memory handling operations (allocation, pattern filling, data comparison),
- device interrupts handling,
- address translation for DMA-transactions programming,
- simulated time measuring and timeout setup,
- debug test output,
- other auxiliary procedures.

5. System-level verification

For system-level verification the test program is translated for execution on a general-purpose core of the verified SoC as well as the standard environment library. The framework also provides a bootstrap program for basic system initialization required for the test to run. The test and the library are linked into a single executable image (the system-level test). To run the test the execution environment places this image in the memory of the SoC (DRAM and/or NVRAM) and transfers control to the entry point of the environment library, which in turn calls the test function. After the test execution the environment library handles the exit code and provides diagnostic information (fig. 1).

The framework allows executing the same unit test with different system settings, providing comprehensive unit integration check. System settings programming is performed by the bootstrap part of the environment library; their values are described in additional files and are transmitted to

the system-level test either via compilation macro definitions or as object files with initialized C-structures during linkage.

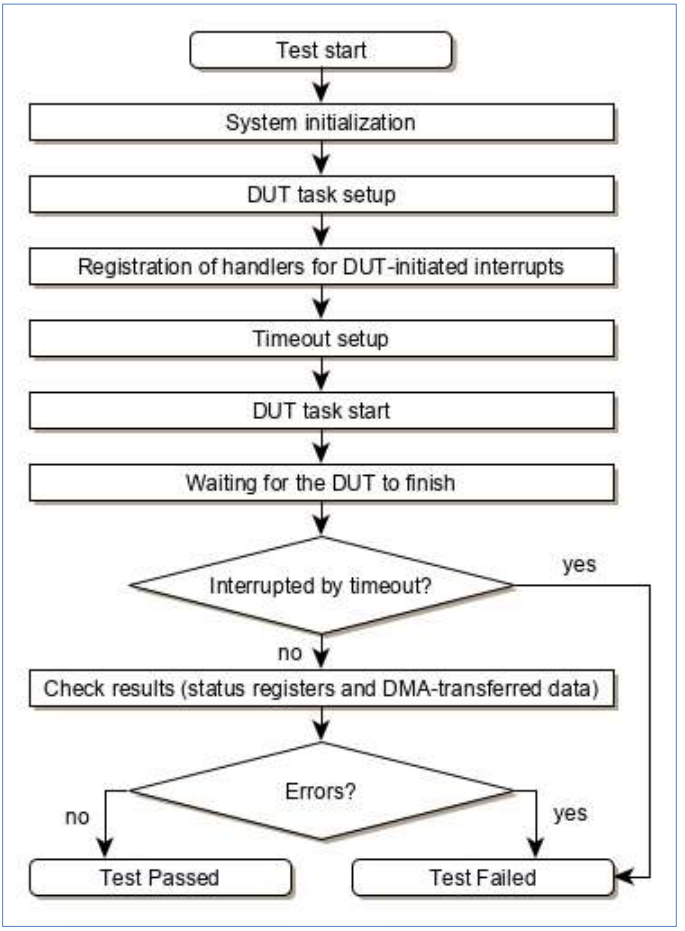

Fig. 1. The framework for system-level verification

Examples of system settings to vary range from separate bits in different control registers of the verified SoC to modes which require additional nontrivial setup. For example, DMA-transactions from the tested device can work directly with system physical addresses or can be additionally redirected via the IOMMU (Input/Output Memory Management Unit).

The environment library implements the API with functions executed in super-user mode. Read/write access to the device registers is implemented with load/store instructions with specific attributes (memory type specifiers). In microprocessors of the Elbrus family registers of external programmable devices are placed within PCI-address spaces: memory, I/O and PCI-configuration space. The test defines a target device address in a PCI-configuration space and allocates necessary address ranges in PCI I/O or memory spaces via appropriate API functions.

The environment library provides a simple heap manager without deallocation implementation. The test program allocates data arrays in the heap for use as RAM regions accessed from the tested device by DMA-transactions.

Virtual addresses for DMA-transactions are written to the device registers and/or to descriptor tables in RAM. In the simplest case the virtual address is equal to the physical address: so-called transparent translation, but DMA-transactions from the tested device can be redirected via the IOMMU, so the environment library provides functions for IOMMU configuration and in-test

address translation functions. The test uses that functions for getting virtual addresses from physical ones, which are returned from the heap allocation-function.

The environment library implements functions for the system interrupt controller configuration and test-defined interrupt handling. The test configures interrupts to be sent by the tested unit and registers callback functions handling those interrupts. During the test execution the environment library catches interrupts from the device and calls registered handlers.

Simulated time measuring is implemented via reading of the clock-counting register or programming local timer to send interrupts in defined time intervals.

The system-level test can be compiled for different execution environments: a functional model, a simulated RTL-description of the tested SoC, an FPGA-based emulator or a manufactured chip. The target execution environment determines the bootstrap procedure and the debug print support linked to the test.

The functional model allows fast execution with high observability (instruction execution trace, units programming trace), so it is used for the test and the environment library debug.

6. Unit-level verification

The structure of the unit-level testbench is presented on fig. 2. The testbench consists of the environment library and the test linked to the testbench as a PLI-application, an adapter for the DUT-system bus interface and, possibly, a specific imitator of an external device.

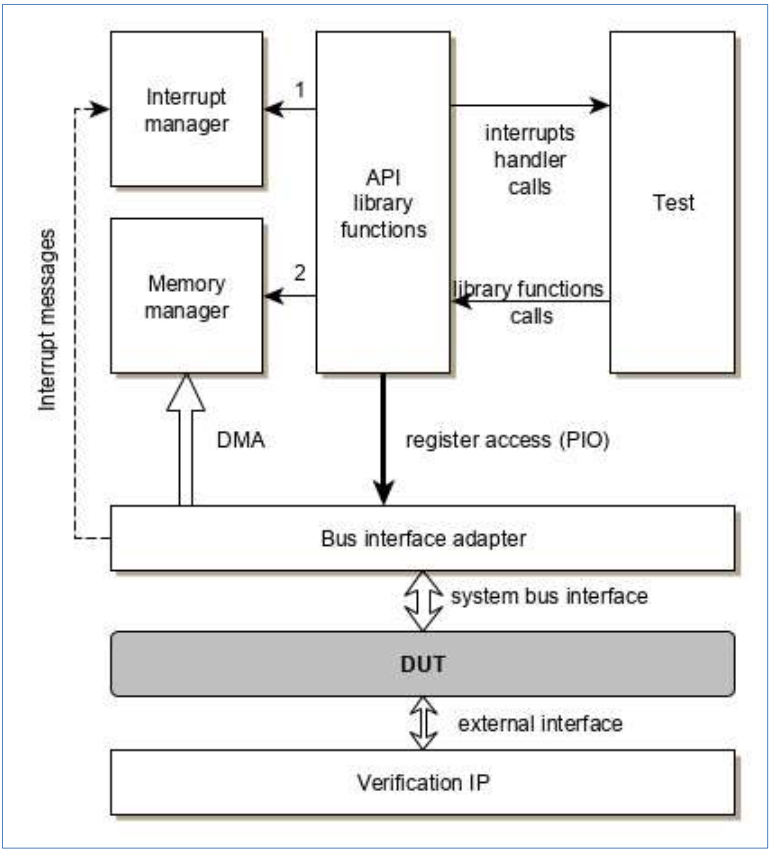

Fig. 2. The structure of the unit-level testbench

The interface of the device-under-test which connects it to the rest of the SoC requires an appropriate adapter for interaction with the testbench. It provides an interface-specific implementation for DUT registers access operations and redirects DUT-initiated transactions to the environment library.

There are two separate address spaces in the test: "internal" for direct access from the test and "external" for DMA-transactions. Memory manager returns to the test pointers with "internal" addresses for memory allocation requests and all library functions for on-core memory processing work with "internal" addresses. Addresses to be targeted by DMA-transactions are wrapped by translation functions that convert internal pointers to external ones and record this translation. DMA-requests are transferred by the adapter to the memory manager that checks DMA destination addresses against previously recorded translations. If there is an appropriate record of translation, the memory manager writes data from DMA-transactions or reads it for return to the adapter. Otherwise an error is detected.

Interrupt messages issued by the device are registered within the environment library; when the test calls library functions, pending interrupts are handled and a user-defined callback is executed.

Simulated time measuring is implemented by means of functions DPI-exported from the part of the library written in SystemVerilog.

Different devices with one programming interface can be tested by the same test program even if they have different bus interfaces; different bus interfaces require different adapters to be implemented. The tested controller can be connected to the adapter not directly, but through the root commutator of the block of controllers including the unit in consideration (fig. 3).

That variant of the DUT allows verification of interaction between system commutator and the tested controller (intermediate-level verification). Test scenarios with simultaneous work of several controllers can be implemented.

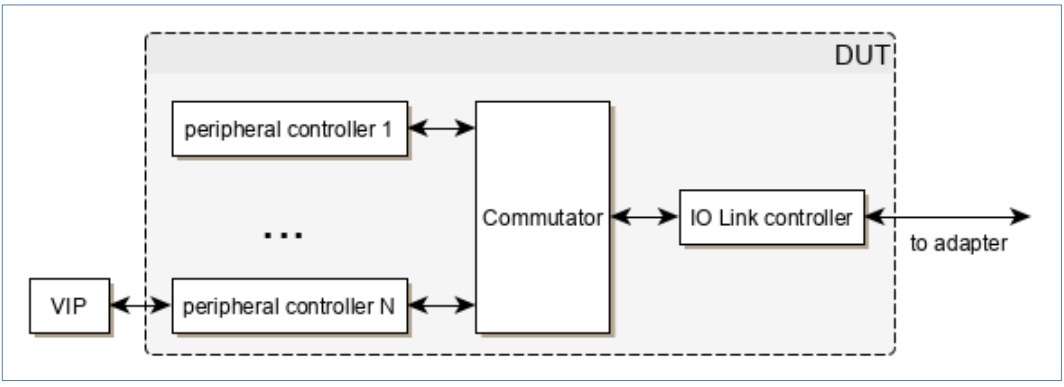

Fig. 2. Indirect connection through the root commutator

7. Results and use experience

The described approach to test development has been applied to verification of peripheral interfaces controllers of standalone southbridge ASICs developed in MCST [9], such as HD Audio, SATA, USB 2.0, PCI and PCI-e bridges, and multiple low-speed controllers. Now it is used for verification of embedded IOHubs being developed for a new generation of the Elbrus microprocessors. Standalone and embedded southbridges have different in-house interfaces to transfer packets based on PCI Express transaction layer packets [10], therefore different adapters have been implemented in order to reuse the same set of tests.

MCST designs computing systems based on CPU of Elbrus and SPARC instruction set architectures, thus the environment library for system-level tests is implemented for both

architectures and for different microprocessor models (starting from Elbrus-4C [13] for Elbrus-based microprocessors and R-1000 [14] for SPARC-based ones).

The typical test development and use flow consists of the following subsequent stages:

- system-level build for functional model execution and test logic debug;
- unit-level build for standalone unit verification;
- unit-level build for verification of the unit as a part of the southbridge;
- system-level build for test execution on full system-on-chip (RTL or FPGA-based prototype [11]).

The system-level environment library supports simultaneous execution of several tests for different controllers on multi-core systems. Tests are executed on different cores; shared resources are distributed between tests based on static planning [12].

8. Future work

The described approach for unit-level verification was implemented mainly for southbridge controllers of MCST projects. The future work is supposed to embrace adaptation of system-level tests for north-bridge integrated graphics cores to unit-level verification. It requires further development of internal system bus interface adapters for different target CPU models.

There is also an endeavor to use already developed system-level tests for verification of hardware I/O virtualization support in new microprocessors of Elbrus family. The test program is supposed to run as a simple guest OS while the environment library functions are executed in hypervisor mode. Different modes of virtual I/O support are to be implemented: emulation mode and direct device assignment.

References / Список литературы

[1]. Anil Deshpande. Verification of IP-Core Based SoC's. In Proc. of the 9th International Symposium on Quality Electronic Design, 2008, pp.433–436.

[2]. G. Mosensoson. Practical approaches to SoC verification. In Proc. of the DATE User Forum, 2002.

[3]. The Portable Test and Stimulus Standard. Available at: https://www.accellera.org/downloads/standards/portable-stimulus, accessed: 11-May-2019.

[4]. Bryon Moyer. Portable Stimulus Intent. Accellera's New Standard Goes to Early Adopters. EEJournal, July 31, 2017. Available at: https://www.eejournal.com/article/portable-stimulus-intent, accessed: 11-May-2019.

[5]. Standard Universal Verification Methodology. Available at: http://accellera.org/downloads/standards/uvm, accessed: 11-May-2019.

[6]. Narendra Kamat. IP Testing for Heterogeneous SOCs. In Proc. of the 14th International Workshop on Microprocessor Test and Verification, 2013, pp. 58–61.

[7]. IEEE Standard for SystemVerilog. IEEE Std 1800-2009.

[8]. ISO International Standard ISO/IEC 14882:2011(E) – Programming Language C++.

[9]. A.K. Kim, M.S. Mikhailov, V.M. Fel'dman. IO-subsystem for «MCST-4R» and «ELBRUS-S» SOCs based on peripheral interfaces controller IC. Issues of Radio Elactronics, series EVT, no. 3, 2012 (in Russian) / А.К.Ким, М.С.Михайлов, В.М.Фельдман. Подсистема ввода-вывода для систем на кристалле «МЦСТ-4R» и «Эльбрус-S» на основе микросхемы контроллера периферийных интерфейсов. Вопросы Радиоэлектроники, серия ЭВТ, вып. 3, 2012.

[10]. Petrochenkov M. V., Mushtakov R. E., Stotland I. A. Verification of 10 Gigabit Ethernet controllers. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 4, 2017, pp. 257-268. DOI: 10.15514/ISPRAS-2017-29(4)-17.

[11]. F. Budylin, I. Polishyk, M. Slesarev, S. Yurlin. The experience of prototyping MCST CJSC' microprocessors. Issues of Radio Elactronics, series EVT, 2012, no. 3 (in Russian) / Ф.К. Будылин, И.А. Полищук, М.В. Слесарев, С.В. Юрлин. Опыт прототипирования микропроцессоров компании ЗАО «МЦСТ». Вопросы радиоэлектроники, серия ЭВТ, 2012, вып. 3.

[12]. Frolov P.V. System-level test integration based on static resource allocation. Issues of Radio Elactronics, series EVT, 2018, no. 2, pp. 76–80 (in Russian) / П.В. Фролов. Система интеграции инженерных

тестов на основе статического распределения ресурсов. Вопросы радиоэлектроники, 2018, вып. 2, стр. 76–80.

[13]. Central processor unit «Elbrus-4C». [Online]. Available at: http://www.mcst.ru/elbrus-4c, accessed: 11-May-2019 / Центральный процессор «Эльбрус-4С».

[14]. Central processor unit «R1000». [Online]. Available at: http://www.mcst.ru/r1000, accessed: 11-May-2019 / Центральный процессор «R1000».

Информация об авторе / Information about author

Павел Викторович ФРОЛОВ – окончил Московский физико-технический институт в 2010 году. Начальник сектора системной верификации отдела моделирования и верификации АО «МЦСТ». Сфера научных интересов: логическая верификация аппаратуры, системы-на-кристалле, автоматизация верификации.

Pavel Viktorovich FROLOV graduated from Moscow Institute of Physics and Technology in 2010. Currently he is a head of the sector of system-level verification in the Department of Modeling and Verification in JSC MCST. Research interests: hardware verification, systems-on-chip, verification automation.

DOI: 10.15514/ISPRAS-2019-31(3)-6

Test environment for verification of multi-processor memory subsystem unit

D.A. Lebedev, ORCID: 0000-0002-9244-4949 <lebedev_d@mcst.ru>
M.V. Petrochenkov, ORCID: 0000-0001-7384-9732 <petroch_m@mcst.ru>
MCST, 1, Nagatinskaya st., Moscow, 117105, Russia

Abstract. State of the art microprocessor systems usually include complex hierarchy of a cache memory. Coherence protocols are used to maintain memory consistency. An implementation of memory subsystem in HDL (hardware description language) is complex and error-prone task. Ensuring the correct functioning of the memory subsystem is one of the cornerstones of a modern microprocessor systems development. Functional verification is used for this purpose. In this paper, we present some approaches for verification of memory subsystem units of multi-core microprocessors. We describe characteristics of memory subsystems that need to be taken into account in the process of verification. General structure of test environment for stand-alone verification of memory subsystem units is presented. Classification of checking model types and their advantages and disadvantages are described. The approach of construction of a standalone verification environment using Universal Verification Methodology (UVM) is presented in the paper. Restrictions that should be taken into account when verifying memory subsystem unit are listed. The generation stimulus algorithm stages are presented. Method of using "hints" from design under verification to eliminate nondeterminism is used in the implementation of checking module. We review several other techniques for checking the correctness of memory subsystem units, which can be useful at different stages of project development. A case study of applying the suggested approaches for verification of Home Memory Unit of microprocessors with Elbrus architecture is presented. Classification of detected and corrected errors in different submodules of verified device is provided. Further plan of the test system enhancement is presented.

Keywords: multicore microprocessors; cache memory; coherence protocols; test system; model-based verification; stand-alone verification.

For citation: Lebedev D.A., Petrochenkov M.V. Test environment for verification of multi-processor memory subsystem unit. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 3, 2019. pp. 67-76. DOI: 10.15514/ISPRAS-2019-31(3)-6

Тестовое окружение для верификации блока подсистемы памяти многопроцессорной системы

Д.А. Лебедев <lebedev_d@mcst.ru>
М.В. Петроченков <petroch_m@mcst.ru>
АО «МЦСТ», 117105, Россия, г. Москва, ул. Нагатинская, д. 1, стр.23

Аннотация. Современные микропроцессорные системы обычно включают сложную иерархию кэш-памяти. Протоколы когерентности используются для поддержания согласованности памяти. Реализация подсистемы памяти на языке описания аппаратуры является сложной и подверженной ошибкам задачей. Обеспечение корректного функционирования подсистемы памяти, является одной из важнейших задач в процессе разработки современных микропроцессорных систем. Для этого используется функциональная верификация. В данной работе представлены некоторые подходы к верификации блоков подсистем памяти многоядерных микропроцессоров. Описаны характеристики подсистем памяти, которые необходимо учитывать в процессе верификации. Представлена общая структура тестовой системы для автономной верификации блоков подсистемы памяти. Приведена

классификация типов проверяющих моделей, их преимущества и недостатки. В статье представлен подход к построению автономного окружения для верификации с использованием универсальной методологии верификации (UVM). Перечислены ограничения, которые следует учитывать при проверке блоков подсистемы памяти. Представлен алгоритм генерации входных стимулов. Для устранения неопределенности текущего состояния верифицируемого устройства в проверяющем модуле используется метод анализа «подсказок». Рассмотрен ряд других методов проверки корректности блоков подсистемы памяти, которые могут быть полезны на различных этапах разработки проекта. Представлен пример применения предложенных подходов к верификации блока HMU микропроцессоров с архитектурой Эльбрус. Приведена классификация обнаруженных и исправленных ошибок в различных подмодулях верифицируемого устройства. Представлен дальнейший план совершенствования тестовой системы.

Ключевые слова: многоядерные микропроцессоры; кэш память; протоколы когерентности; тестовая система; верификация на основе моделей; автономная верификация

Для цитирования: Лебедев Д.А., Петроченков М.В. Тестовое окружение для верификации блока подсистемы памяти многопроцессорной системы. Труды ИСП РАН, том 31, вып. 3, 2019 г., стр. 67-76 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(3)-6

1. Introduction

With the development of microprocessor technology and growth of the number of computational cores and CPUs in systems processor performance increases rapidly. Unfortunately, the speed of memory access is not growing as fast as the speed of the processor [1]. Thus, one of the biggest bottleneck elements become the memory subsystem. To level the difference in speed, designers of microprocessor systems implement a complex memory subsystem that includes cache hierarchy. State of the art microprocessor systems usually include 3-4 levels of cache memory. This approach is able to reduce the number of accesses to main memory, and, therefore, reduce memory access instructions average execution time.

In the multicore systems if multiple cores are simultaneously allowed to contain copies of a single memory location, the problem of maintaining memory consistency arises. A mechanism must exist to ensure that all copies remain consistent when the contents of that memory location are modified. Coherence protocols support such mechanism. Usually we have higher-level caches shared between cores and lower-level caches served by a single core. Complex systems that combine several multi-core processors may also have cache memory to speed up other processors' access to their memory.

A large number of processors and processor cores and complexity of system data exchange organization makes coherence protocol very complicated. An implementation of cache coherence protocol is a complex and error-prone task. Errors of this kind are critical and difficult to detect on system-level verification. Thus, a memory subsystem and implementation of coherence protocols in HDL (Hardware Description Languages) models must be thoroughly verified [2].

There are two main methods for verification of memory subsystem: a simulation-based verification and formal verification [3]. Formal verification is used to mathematically prove the correctness of a DUV (Device Under Verification) model with respect to its specification. It is widely known that main advantage of formal methods is their exhaustiveness. Many works are devoted to this method [4-6]. Disadvantages of these methods are complexity of development and high specification requirements. Simulation-based methods are not exhaustive, but they can be applied at earlier stages of development and they are much simpler.

Verification of a memory subsystem, as a part of whole microprocessor, can be provided by means of system verification [7]. However, it is essential to mention that some of the components of a memory subsystem are invisible from the point of view of a testing program and it is hard to recreate necessary conditions for verification with proper quality. To overcome these drawbacks, a stand-alone verification of the memory subsystem is usually used.

There are a number of methods to implement a standalone functional verification of a memory subsystem. One of them is C++TESK Testing ToolKit created in ISP RAS [8]. It is an open-source C++ based toolkit intended for automated functional testing of RTL (HDL) models of digital hardware (in Verilog and VHDL). The tool included a library of C++ classes and macros that define facilities for all parts of a verification environment. Some of disadvantages of this tool are high complexity of the application and needs documentation and checking reference model high accuracy.

Another tool name is Alone-env created in the MCST. The Alone-env provides a wrapper-class over Verilog description of the verified module. The Alone-env too has some disadvantages: the lack of collecting coverage means, high requirements for the checking reference model and the inability to reuse the test system.

Nowadays the most widespread verification methodology is Universal Verification Methodology (UVM). This is a standard verification methodology from the Accellera Systems [9]. UVM designed to enable creation of robust and reusable testbenches and their components. UVM is a class library helps to bring much automation to the SystemVerilog language. Disadvantages of UVM is learning curve is very high for new users and it takes a lot of code to create basic UVM testbench classes. Nevertheless, our team already have a number of test systems, basic classes and libraries written and debugged. Therefore, we choose UVM for developing the stand-alone verification environment of memory subsystem modules.

The rest of the paper is organized as follows. Section 2 reviews the existing techniques for standalone verification of the memory subsystem. Section 3 describes a case study suggests an approach to the problem of developing test system. Section 4 describes additional used approaches. Section 5 reveals results and Section 6 concludes the paper.

2. Standalone verification methods of memory subsystem

In a stand-alone verification we implement test system that allows to select a single part of the whole system and examine its behavior in the test environment that behaves in a way similar to the “real” system. Correct mechanisms of interaction with DUV are defined in its specification. One of the main advantages is that it is easier to explore edge and corner cases in the verified module.

When verifying a part of the memory subsystem with included cache, we need to take into account some features while developing the test system:

- it consists of cache lines that are fixed size blocks used to transfer data between two nodes of the system;
- logic to locate and transfer requested data;
- cache line also hold service information;
- may be several requesters which work with different cache lines
- if two or more requesters want to refer to the same cache line such request have to be serialized and completed in the same order as they received;
- controller support some of implementations of a coherence protocol;
- due to the limited amount of a memory, one of the data eviction algorithms is implemented.

Test environment (or testbench) for verifying the memory subsystem usually includes:

- generator of input stimulus;
- checker of collected reactions correctness;
- module collecting coverage information.

Generator of input stimuli is responsible not only for primary requests that perform operations with memory, it also collects reactions from verified device and generate answers from test environment - secondary requests. Generalized scheme of test environment shown on Fig. 1. Generation of stimuli can be simplified by using TLM [10] (Transaction Level Modeling) to

communicate with DUV. TLM allows focusing more on the functionality of the data transmission and less on its actual implementation.

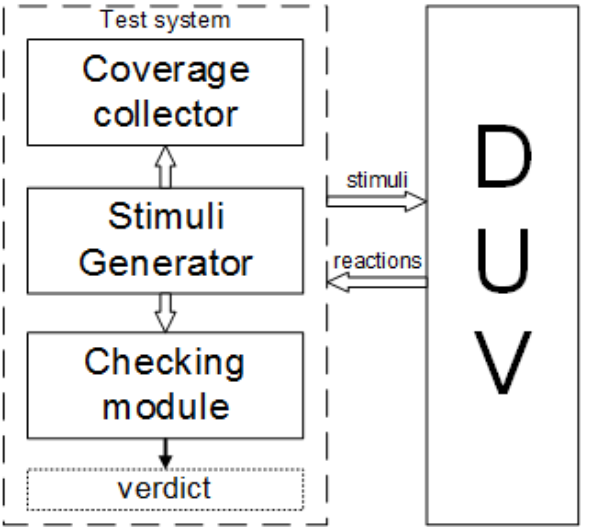

Fig 1. Generalized scheme of test environment

If the verified device has a complex structure and many states, the easiest way to check correctness of reactions is building the separate checking module. Checking module is based on the external to the test environment reference model usually written in high-level language (C, C++ or some specific languages for verification of hardware, such as SystemVerilog, SystemC or «e»). All requests and reactions from the verified device sent to the checking module where then made a conclusion about the correctness of the behavior.

The reference models could be divided into three types: cycle-accurate, discrete-event with time accounting and event models [11]. First two of them require a very accurate specification. It is hard to support design changes that happen very often on the first steps of the development. Furthermore, the similarity of the implementations of the model and the DUV can lead to duplication of errors. To check correctness of memory subsystem, it is reasonable to use event models because they require less time to develop and more flexible for changes of the design. Data interchange of the test system with reference model occurs instantly by calling appropriate functions. For some devices, there are several correct scenarios of the operation for the same input stimuli. We call those devices non-deterministic. There are two methods allowing using behavioral event models for verification of these devices [12].

The first method is dynamic refinement of transaction level model. A general approach is as follows. When a reference model gets a request and there are several possible ways to react to the request, the model creates additional instances and executes the requests in each of them. Then the models are waiting for the reactions from the device under verification. The reaction contents service information (such as a response type, a direction of sending, etc) which helps to exclude impossible states. Absence of suitable state for reaction signals about an error. The sign of a successful completion is comparison all the reactions of the DUV and removal of all unnecessary states. The complexity of this approach is that the number of possible states potentially grows exponentially with a number of stimuli. However, this method can be implemented efficiently for memory subsystem units because all requests to a single cache line are serialized and requests to different cache lines are independent.

Second method is identification of a single correct state using hints from the verified device or a "gray box" method. This method replaces usual “black box” method. When we cannot predict the

"real" sequence of interactions, we access inner interfaces of the verified device. Information from these interfaces have to be transferred to the test environment and helps to determine a single possible execution scenario and eliminate nondeterminism. This method imposes additional requirements to the device specification, but, as a result, it is quite simple to implement.

Coverage collection module extract information of functional code coverage. This information is used to identify unimplemented test cases and helps to improve stimuli generation by adding new test scenarios.

3. Using gray box approach for verification of home memory unit

Home Memory Unit (HMU) is a part of memory subsystem of 16-core "Elbrus" microprocessor responsible for the coherent and non-coherent access to the RAM from different requesters. HMU contains a global directory (MOSI protocol of coherence), which monitors the requests of other processors to its memory and a DMA directory which is a full copy of the DMA caches of all processors (supports the extended coherence protocol MOI). Total volume of the directory in HMU is 2.5 MB, size of entry of a cache line is 80 B, number of banks – 128, bank associativity – 16. Main functions of HMU include:

- serialization of all requests to RAM;
- reduction of coherence traffic and access time to RAM;
- support of interprocessor coherence.

Test system for stand-alone verification of the coherence protocol implementation and other functionality of HMU based on UVM. UVM helps to generate pseudo-random constrained input requests to cover possible states of the verified device.

We have to note some restrictions for generation primary, secondary stimuli and answers. The first of them is limited amount of space in input buffers. Due to process of verification, it is important not to lose some data. When generating random system settings, it is necessary to take into account that some setup combinations may be incorrect and lead to errors. There are several types of requesters in the test system. Each of them has special identifier and a set of possible operation codes. The specific implementation of the coherence protocol also imposes restrictions on the used operation codes. Sending an inappropriate operation code may result in undefined results. Address generation is also a non-trivial task. The address have to fit the interleaving conditions. In addition, each requester have to wait for the completion of previous request when working with same cache line.

Stimuli generation is divided into several stages:

1. randomization of device configuration registers. This allows to switch different ways for handling requests and determine request routing;
2. creating list of addresses for current configuration of device with respect to routing setup;
3. choosing random requester and cache line address;
4. checking cache line availability and presence of resources needed for request transfer;
5. choosing random operation code constrained by the current state of cache line;
6. sending primary request, collecting reactions from the device under verification, sending secondary requests;
7. collecting all of necessary reactions and completion of current request;
8. transferring transaction information to checking module.

To simplify handling of requests and reactions we create models of each used cache line. Model of cache line is an object that stores information about primary request, collected reactions, data and some auxiliary functions. For generation of correct requests we created an associative memory storing current states for each cache line. The choice of the next request type is made according to the limitations imposed by the coherence protocol and the current state of the cache line. For

example, one of these rules is there cannot be two requesters in a modified (M) state for a single cache line. Another feature, which was necessary to pay attention, is that one address tag corresponds to two-neighbor cache lines information. This mechanism allows increasing the ratio of the directory coverage (the ratio of the cache memory covered by the directory to the cache memory of the directory).

As noted before, there are two ways of building checking module. The choice of «gray box» method is determined by following sources of a nondeterminism inherent to HMU:

- HMU contains two input queues of primary requests what means exponential growth of possible device states size (2n+m, where n, m – number of requests inside input queues);
- cache eviction algorithm in the global directory.

HMU has two cache memories responsible for different functions of a memory subsystem: the global and DMA directories. The global directory has information only about data belonging to the own processor and used by other processors. Along with that there is no information about presence of this cache line in cache of own processor. Such information located in the L3 cache. DMA writes are also coherent requests. For a fast and correct handling of DMA writes sent by DMA controllers the special DMA cache directory is present inside HMU. This cache directory supports extended coherence protocol MOI with substates. Its main function is processor notification about cache lines captured by DMA controllers and storing their states.

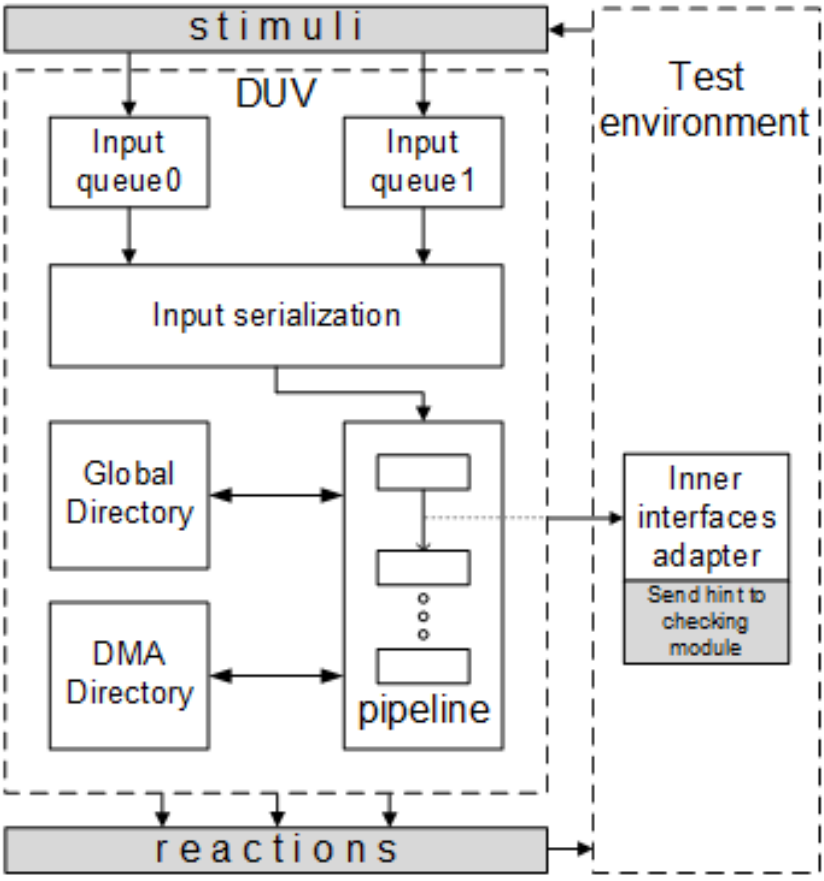

Fig 2. Simplified version of the test environment using the "gray box" method.

The device under verification connected with other parts of the system by means of network-on-chip and has two input channels for primary requests. All generated primary requests are sent to the DUV and the checking module simultaneously. The checking module (implemented in C++) receives requests and reactions from the verified device by means of DPI (Directed Programming Interface). Using of the DPI is necessary to match the types and classes of the test environment written in SystemVerilog hardware description language with the reference model interface functions. Inside checking module, all requests are received into two queues. Requests to the same cache line can get into the different queues. It is impossible to predict which of these requests will be handled first. Getting a sequence hint from the device under test eliminates the nondeterminism of the current state. Stable and well-described inner interfaces to the point where all request are serialized, made it possible to build simple checking module in the short time. In a similar way, an access to the eviction mechanism interface was obtained. In addition, the checking module and its behavior model may be modified if it will be needed in the future projects. Structure of the test environment with proposed “gray box” method shown in fig. 2.

*4. Additional verification methods*Management of transaction flow

To check if the verified device operates correctly, it is necessary to achieve multiple edge and corner cases. This involves filling all input and output primary and secondary requests queues, delaying some necessary types of answers and blocking of transactions from some modules [13]. HMU supports the credit-exchange mechanism, which indicates the devices ability to accept certain type of requests. We added the special configuration module that randomizes time delays of sending requests and credits. Management of delays setup allows to create different test scenarios with overflowing requests and responses buffers. This mechanism helps to detect livelocks and deadlocks. These types of system behavior are hard to implement during system testing.

4.2 Applying assertions

SystemVerilog Assertions (SVA) is an important subset of SystemVerilog [14]. The assertions are used to specify the behavior of the system. The assertions work as follows: we add some piece of verification code to the test system that monitors a design implementation for compliance with the specifications. In addition, the assertions can be used to flag that input stimuli do not conform to assumed requirements. In the beginning of the project, it may help to find more bugs and locate them faster.

In HMU verification process the assertions are used for checking for uncertain and unconnected states of signals. Usage of coherence protocols in the DUV involves certain restrictions on the stimuli generation and the state of the cache lines for different requesters. Thus, additional function of the assertions, which was used in the test system, is detection of the discrepancy between coherence protocol specification and generated requests types in the certain cache lines states. The disadvantage of this approach is the limitation of the properties of the verified device that can be checked by assertions.

4.3 On-the-fly ECC errors insertion

ECC bits are stored together with the state of the cache line. Special submodules of HMU encode the data written to the RAM and decode data read from RAM. Using ECC bits allows to detect single, double, parity errors and to correct single errors. This mechanism is a source of potential errors in the device. The special module with flexible configuration was developed to insert single and double errors. This module is managed by the test system. Frequency and type of error insertion can be regulated. Detecting and correcting ECC errors additionally loaded computing logic of verified device.

5. Results

The approaches described in this paper were applied for standalone verification of Home Memory Unit of 16-core and 2-core with 6 integrated graphic boosters microprocessors with “Elbrus” architecture.

There are some difference in operating with memory subsystem in the microprocessors. The 16-core microprocessor’s HMU has a global directory and DMA directory, sends coherent requests with accordance to the state in the global directory, and collects short coherent answers and coherent answers with data for write operations. For read operations, requester (DMA or L3 cache) collects all the answers.

The 2-core microprocessor does not have a global directory in HMU but includes DMA directory. HMU provides inter-core coherence. Coherence requests are sent broadcast to the cores and DMA. HMU also collects all the answers for write operations and for read operations only when requester is not DMA. Integrated graphic boosters are not snooped.

Due to the specificity of the test system construction, some part of MC controller (the MC adaptor) was also added to the verified system. Generator of responses from MC controller with randomized setups was also developed.

In the process of the standalone verification of the Home Memory Unit we found 28 errors that have not been found by other means of verification. All errors were corrected. The distribution of the number of bugs in different subsystems of the HMU are presented in Table 1. Code and functional coverage was carried out and 94% coverage was extracted. Total result indicates about effectiveness of the proposed methods of standalone verification.

Table 1. Types of found bugs and its quantity

Type of bugs	**Number of bugs**
Coherence protocol implementation	21
Configuration registers	2
Parity checker	1
Performance improvement	2
MC adaptor	3
Total:	28

6. Conclusion and directions for future work

Memory subsystem is one of the most important parts of microprocessors. Its parts that support coherence protocols are especially complicated and error-prone. Verification of these types of devices is time-consuming and labor-intensive work. The stand-alone verification designed to simplify this task. The approaches mentioned in this paper can be applied for stand-alone verification memory subsystem parts regardless of their implementation.

The proposed approaches have been applied in the verification of the Home Memory Unit as a part of multi-core microprocessor memory subsystem with “Elbrus” architecture developed by "MCST". Test environment and test scenarios made it possible to detect and correct a number of logical errors that were not detected by other verification methods.

In the future, it is planned to adapt the test environment for the forthcoming projects and possible changes in coherence protocols.

References / Список литературы

[1]. Hennessy J.L., Patterson D.A. Computer Architecture: A Quantitative Approach. Fifth Edition. Morgan Kaufmann, 2012. 857 p.

[2]. A. Kamkin, M. Petrochenkov. A Model-Based Approach to Design Test Oracles for Memory Subsystems of Multicore Microprocessors. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 3, 2015, pp. 149-160. DOI: 10.15514/ISPRAS-2015-27(3)-11.

[3]. W.K. Lam. Hardware Design Verification: Simulation and Formal Method-Based Approaches. Prentice Hall, 2005, 624 p.

[4]. Burenkov V.S. A Technique for Parameterized Verification of Cache Coherence Protocols. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 4, 2017, pp. 231-246. DOI: 10.15514/ISPRAS-2017-29(4)-15.

[5]. Ivanov Lubomir and Nunna R. Modeling and verification of cache coherence protocols. In Proc of the 2001 IEEE International Symposium on Circuits and Systems, vol. 5, 2001, pp. 129-132. DOI: 10.1109/ISCAS.2001.922002.

[6]. P.A. Abdulla, M.F. Atig, Z. Ganjeiy, A. Reziney, and Y. Zhu, Verification of cache coherence protocols wrt. trace filters. In Proc. of the 15th Conference on Formal Methods in Computer-Aided Design, pp. 9-16.

[7]. I.A. Stotland, V.N. Kutsevol, A.N. Meshkov. Problems of functional verification of Elbrus microprocessor L2-cache. Issues of radio electronics, ser. EVT, no. 1, 2015, pp. 76-84 (in Russian) / Стотланд И.А., Куцевол В.Н., Мешков А.Н. Проблемы функциональной верификации кэш-памяти второго уровня микропроцессоров с архитектурой «Эльбрус». Вопросы радиоэлектроники, сер. ЭВТ, 2015, no. 1, стр. 76-84.

[8]. C++TESK Testing ToolKit review. Available at: https://forge.ispras.ru/projects/cpptesk-toolkit, accessed 12.06.2019.

[9]. Standard Universal Verification Methodology. Available at: http://accellera.org/downloads/standards/uvm, accessed 12.06.2019.

[10]. Kamkin A., Chupilko M. A TLM-based approach to functional verification of hardware components at different abstraction levels. In Proc. of the 12th Latin-American Test Workshop (LATW), 2011, pp. 1-6.

[11]. Averill M. Law, W. David Kelton. Simulation Modelling and Analysis. McGraw-Hill Education, 3rd edition, 2000, 784 p.

[12]. Petrochenkov M., Stotland I., Mushtakov R. Approaches to Stand-alone Verification of Multicore Multiprocessor Cores. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 3, 2016, pp. 161-172. DOI: 10.15514/ISPRAS-2016-28(3)-10.

[13]. Lebedev D.A., Stotland I.A. Construction of validation modules based on reference functional models in a standalone verification of communication subsystem. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 3, 2018, pp. 183-194. DOI: 10.15514/ISPRAS-2018-30(3)-13.

[14]. 1800-2017 - IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language. Available at: https://standards.ieee.org/standard/1800-2017.html, accessed 22.06.2019.

Информация об авторах / Information about authors

Дмитрий Алексеевич ЛЕБЕДЕВ работает в АО МЦСТ, Москва, Россия. Он защитил диплом специалиста в области электроники в 2014 г. в НИЯУ МИФИ. Область его исследовательских интересов включает исследование методов верификации контроллеров связи, систем прерываний, устройств подсистемы памяти с поддержкой протоколов когерентности.

Dmitry Alexeevitch LEBEDEV works in AO MCST, Moscow, Russia. He earned a specialist in electronics diploma in 2014 at MEPhI. His area of research interests includes the verification methods study of communication system controllers, interrupts systems and memory subsystems devices with support of protocols of coherence.

Михаил Владимирович ПЕТРОЧЕНКОВ работает в АО МЦСТ, Москва, Россия. Он получил степень магистра в 2012 г. МФТИ. Область его научных интересов включает исследование методов верификации устройств подсистемы памяти и контроллеров связи.

Mikhail Vladimirovich PETROCHENKOV works in AO MCST, Moscow, Russia. He received his master's degree in 2012 from MIPT. Area of his scientific interests includes the verification methods study of memory subsystemsdevices and communication controllers.

DOI: 10.15514/ISPRAS-2019-31(3)-7

Standalone verification of IOMMU with virtualization supporting

[1]*A.A. Petrykin, ORCID: 0000-0001-5779-1980 <petrykin_a@mcst.ru>*
[1]*I.A. Stotland, ORCID: 0000-0002-4359-3059 <stotl_i@mcst.ru>*
[1,2]*A.N. Meshkov, ORCID: 0000-0002-8117-7398 <alex@mcst.ru>*
[1] *INEUM, 24, Vavilova st., Moscow, 119334, Russia*
[2] *MCST, 1, Nagatinskaya st., Moscow, 117105, Russia*

Abstract. This article presents an approach to standalone verification of I/O Memory Management Unit with virtualization supporting. We presented the base architecture of the test system. The main problems encountered during the verification of IOMMU with virtualization support are considered. One of the key problems was the formation of translation table pages. The number of translation tables depends on the mode of IOMMU operation and the type of translation. As a solution of this problem the approach to the dynamic generation of translation tables is proposed. The algorithm for formation of translation table pages in the generator is presented. The problem of validating the translation of a virtual address into a physical one using two-level translation tables is solved. The features of the reference model implementation are considered. Reference model and test system which have been used for IOMMU verification of microprocessor with the 6th generation «Elbrus» architecture are described. The main components of the test system and the methods of communication between test system and IOMMU model are presented. The results of IOMMU verification are considered.

Keywords: I/O Memory Management Unit; test system; reference model; Elbrus

For citation: Petrykin A.A., Stotland I.A., Meshkov A.N. Standalone verification of IOMMU with virtualization supporting. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 3, 2019. pp. 77-84. DOI: 10.15514/ISPRAS-2019-31(3)-7

Автономная верификация IOMMU с поддержкой виртуализации

[1] *А.А. Петрыкин, ORCID: 0000-0001-5779-1980 <petrykin_a@mcst.ru>*
[1] *И.А. Стотланд, ORCID: 0000-0002-4359-3059 <stotl_i@mcst.ru>*
[1,2] *А.Н. Мешков, ORCID: 0000-0002-8117-7398 <alex@mcst.ru>*
[1] *ПАО «ИНЭУМ им. И.С.Брука», 119334, Россия, г. Москва, ул. Вавилова, д. 24*
[2] *АО «МЦСТ», 117105, Россия, г. Москва, ул. Нагатинская, д. 1, стр.23*

Аннотация. В данной статье представлен подход к автономной верификации блока управления памятью ввода / вывода с поддержкой виртуализации. Мы представляем базовую архитектуру тестовой системы. Рассматриваются основные проблемы, возникающие при верификации IOMMU с поддержкой виртуализации. Одной из ключевых проблем стало формирование страниц таблицы трансляции. Количество таблиц трансляции зависит от режима работы IOMMU и типа трансляции. В качестве решения этой проблемы предложен подход к динамической генерации таблиц трансляции. Представлен алгоритм формирования страниц таблиц трансляции в генераторе. Решается проблема проверки трансляции виртуального адреса в физический с использованием двухуровневых таблиц трансляций. Рассмотрены особенности реализации эталонной модели. Описаны эталонная модель и тестовая система, которые использовались для верификации микропроцессора IOMMU с архитектурой 6-го поколения «Эльбрус». Представлены методы связи между тестовой системой и моделью IOMMU. Рассматриваются результаты проверки IOMMU.

Ключевые слова*:* блок управления памятью ввода/вывода; тестовая система; эталонная модель; «Эльбрус»

Для цитирования: Петрыкин А.А.**,** Стотланд И.А., Мешков А.Н. Автономная верификация IOMMU с поддержкой виртуализации. Труды ИСП РАН, том 31, вып. 3, 2019 г., стр. 77-84 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(3)-7

1. Introduction

An I/O Memory Management Unit (IOMMU) is a hardware device that translates virtual address received from the I/O subsystem requests to proper machine physical address. IOMMUs have long been used for prohibiting devices from DMA'ing into the wrong memory and for performance optimization. With the hardware support of operating system virtualization IOMMU is also used for extending the protection and isolation properties of VMs (Virtual Machines) for I/O operations, supporting isolation of interrupts from devices and external interrupt controllers and recording of DMA and interrupt errors to system software that may corrupt memory or impact VMs isolation [1][2]. Therefore, modern IOMMUs are quite complex devices that have many modes of operation and their verification is an important step in the development of the microprocessor system.

In the paper, we present a case study for functional verification of IOMMU with virtualization supporting of microprocessor with the 6th generation «Elbrus» architecture developed by MCST. The paper addresses the problem and methods of standalone verification of IOMMU with virtualization supporting.

The rest of the paper is organized as follows. Section 2 considers the problems arising from the verification of IOMMU. Section 3 suggests an approach to the problem of developing page table lines generator. Section 4 presents a common approach to the design a test system and describes its components. Section 5 reveals results. Section 6 concludes the paper.

2. IOMMU verification challenges

For hardware support of operating system virtualization, the translation of a virtual address into a physical one occurs according to a scheme that includes a two-level page structure. At the first level, the virtual guest OS address (GVA) is translated into the physical guest OS address (GPA) using its translation tables. At the second level, the resulting address is translated to the physical address (PA) of the hypervisor using the hypervisor translation tables. Information about the broadcast address for each device is stored in the device table. The table consists of Device Table Entry (DTE) elements. Each DTE contains information about the Domain ID (DID), host page table root pointer (HPTP) and guest page table root pointer(GPTP). In the IOMMU of microprocessor with the 6th generation «Elbrus» architecture, DID field has an extension and is called EDID. In addition to the domain number, it contains information about whether the device belongs to the guest or the hypervisor.

The pages number of translation tables depends on the mode of device operation and the size of the page themselves. Processors with the 6th generation «Elbrus» architecture support three page sizes: 4KB, 2MB and 1GB. For guest virtual address translation through 4KB pages, the number of memory hits can reach 25. Each memory hit takes a long time to process. Therefore, as part of IOMMU can be used a lot of different caches [1].

It follows from the above that the main goal of IOMMU verification is to check the correctness of all translation modes and check the following:

- translation on pages with various size;
- error handling;
- caches correctness;
- translations for the greatest possible number of addresses;
- absence of suspensions;

- absence of unknown logic value (X-state) on output signals.

There are several main problems encountered during the IOMMU verification of processors with the 6th generation «Elbrus» architecture:

- the large number of translation tables and different page size;
- large size of the entire address space;
- different virtual addresses can be translated into one physical.

To solve these problems, it was necessary to create the translation page tables. But their formation for the entire address space would require a large amount of computing resources. The paper [3] presents the approach based on constraint-random generation page table entries (PTEs), which we used for IOMMU verification as a part of northbridge of our previous microprocessor. However, the availability of hypervisor and guest translation caches as well as a large number of translation pages required for guest virtual address translation does not allow to use the approach described in [3]. The traditional approach is to generate a static table for a limited set of addresses which is used for verification Translation Lookaside Buffer (TLB) of MMU[4]-[6]. But using this approach, it is difficult to verify error handling and virtual address translation over various size pages. Therefore, for functional verification of the IOMMU of microprocessor with the 6th generation «Elbrus» architecture, it was decided to develop a dynamic generator of translation table pages, which generates rows of translation tables for any virtual address.

3. Generator of translation table pages

The formation of pages in the generator depends on the translation mode, which is specified directly in the tests and translation request. The algorithm of the generator work could be represented in the form of several consecutive steps:

1) receiving request for translation;
2) request translation and mode analysis;
3) DTE formation;
4) translation pages entry.

Translation modes are divided into single-level or native translations and two-level translations. In turn, native translations can take place in the same domain or split into different domains. Native translations in the same domain doesn't require DTE. For the rest of translation modes DTE formation is necessary. Since a unique translation identifier in Elbrus-12c processors is an EDID, each DTE element is stored in an associative array indexed by it. The formation of a DTE begins with the selection of a random EDID for a given device, after which the presence of a DTE for a given EDID in the array is checked and if it is missing, the remaining fields are formed.

Translation tables have a 4-level structure and consist of 512 elements. Each line of the table contains information about the access rights to it and whether it is the last in the translation structure. In addition, there is a field named PPN (Physical Page Number), that indicates the line from the next translation level, in case the line is on the last translation level, it contains the physical address. Hereinafter the pages of hypervisor and guest translations will be called HP (Host Page) and GP (Guest Page), respectively.

For native translation, the address of the first line is formed from the host page table index (hptp) contained in the DTE, and a part of the translated virtual address. Physical page number for each page level is calculated as follows:

$$ppn(n) = hptp + 512 * (5 - n) + VA(n), \qquad (1)$$

where n – host page level, $VA(n)$ – the part of translated virtual address on level n , $hptp$ – table root pointer.

The full list of pages for native translation is presented in Table 1.

Table. 1. List of pages for native translation

Page level	Page address	Physical Page Number
HP_L4	hptp, VA(4)	hptp + 512 + VA(4)
HP_L3	ppn_L4, VA(3)	hptp + 512 * 2 + VA(3)
HP_L2	ppn_L3, VA(2)	hptp + 512 * 3 + VA(2)
HP_L1	ppn_L2, VA(1)	hptp + 512 * 4 + VA(1)

When generating rows of a two-level table for guest virtual address translation, the generator first calculates the guest physical address (GPA) for first guest page (GP_L4) according to the formula:

$$GPA(4) = GPTP + VA(4) \qquad (2)$$

After that, the pages necessary for the translation of this GPA to HPA are written in the same way as the recording of the pages of translation VA to PA in the native mode. The list of pages for that translation may be seen at Table 2.

Table. 2. List of pages for guest physical addres translation into host physycal address

Page level	Page address	Physical Page Number
HP_L4	hptp, GPA(4)	hptp + 512 + GPA(4)
HP_L3	ppn_L4, GPA(3)	hptp + 512 * 2 + GPA(3)
HP_L2	ppn_L3, GPA(2)	hptp + 512 * 3 + GPA(2)
HP_L1	ppn_L2, GPA(1)	hptp + 512 * 4 + GPA(1)

Then the GP_L4 page itself is written and after that the guest physical address of the next page level (GPA_3) is calculated as the sum of HP_L1 page ppn and part of virtual address:

$$GPA_3 = ppn + VA(3) \qquad (3)$$

And so on to get the GPA of the original GVA. For the obtained GPA, the pages of the last hypervisor translation are formed, which give the desired HPA. In order to avoid writing identical lines at different translation levels, guest pages addresses and ppns are configured as followed:

$$addr(x) = \{ppn, GPA_{x(0)}\}, \qquad (4)$$

$$ppn(x) = hptp + 512 * (9 - x) + GPA_{x(1)}, \qquad (5)$$

where x – guest page level, $GPA\_x(0)$ – the part of the guest physical address that is not translated by hypervisor structures.

The list of guest pages for two-level translation is presented in the Table 3.

Table. 3. List of guest pages in two-level translation

Page level	Page address	Physical Page Number
GP_L4	ppn, GPA_4(0)	hptp + 512 * 5 + GPA_4(1)
GP_L3	ppn_L4, GPA_3(0)	hptp + 512 * 6 + GPA_3(1)
GP_L2	ppn_L3, GPA_2(0)	hptp + 512 * 7 + GPA_2(2)
GP_L1	ppn_L2, GPA_1(0)	hptp + 512 * 8 + GPA_1(1)

To verify the error handling on each level of table can be prescribed a row of the page table that causing an error. That table level can be set via test parameters or randomly. In the same way, translation page sizes can be set through the PS field in a line of translation table. To verify the error handling on each level of table can be prescribed a row of the page table that causing an error. That table level can be set via test parameters or randomly. In the same way, can be setted the translation page sizes through the PS field in a line of translation table.

4. Test System Structure

Test system was implemented with using of SystemVerilog language [7] and Universal Verification Methodology [8]. Use of this language allows for an easy interface with Verilog and SystemVerilog devices, and UVM describes a general test system structure and provides a library of basic verification components.

The test system includes a set of basic components, which are presented below.

4.1 Apb (Advanced Peripheral Bus) agent

Apb agent is used to entrance the set of configuration registers in IOMMU whose access interface is implemented according to the APB protocol.

4.2 Register model

A register model is an entity that encompasses and describes the hierarchical structure of class object for each register and its individual fields. Every register in the model corresponds to and actual hardware register in the design.

4.3 Translation agent

This is the agent, in which the translations are generated and then sent on the DUT (Design Under Test) query interface and generator of table pages. Translation generation is based on constrained randomization. To specify some test scenario, one must define specific constraints for transactions that will be issued. SystemVerilog offers a native support for constrained randomization constructs. Translation agent is also receives the results of the translation from the response interface.

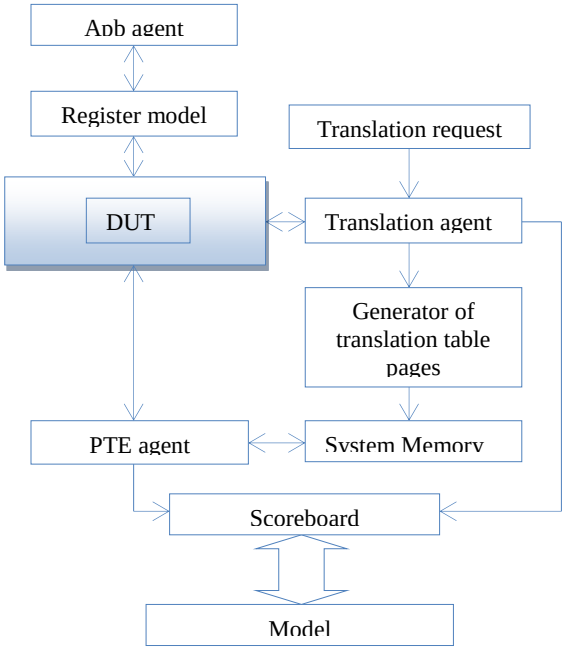

Fig. 1. Structure of a test system

4.4 Generator of translation table pages and system memory.

Each request received from the translation agent is processed by the generator as described in Section 3. All lines of translation pages are stored in system memory.

4.5 Page table entries (PTE) agent

PTE agent collects information about requested and received by the device PTEs from system memory. If for any reason the requested PTE is missing, the system memory will randomly generate it.

4.6 Model

The IOMMU reactions were tested using its reference event model. For reconciling the types and classes of the test system written in SystemVerilog with the C++ language in which the reference model is developed the DPI (Directed Programming Interface) was used. The reference model accepts input stimuli and generates output responses, which are then sent to scoreboard.

4.7 Scoreboard

Scoreboard receives transactions from translation and page table entries interfaces. After that, they are compared with the corresponding transactions received from the model. If a mismatch is detected, the module reports an error in the test system. Test system structure is presented in Fig.1.

5. Results

The approaches described above were applied to standalone verification of the IOMMU of microprocessor with the 6th generation «Elbrus» architecture. Due to standalone verification of the device, 58 errors in the RTL description that have not been found by other means of verification were found and corrected. Total result indicates about effectiveness of standalone verification of I/O memory management unit.

6. Conclusion

I/O memory management units are among the important parts of modern microprocessor systems have to be thoroughly tested. In this article, we presented a method of translation table pages forming. The main advantages of the described approach are:

- no need to create tables for the entire address space;
- the ability to dynamically set the page size;
- convenience of exception checking due to dynamic generation of page table row fields;
- ease of obtaining maximum coverage, due to the possibility of calls to any address.

The principles described in the paper do not depend mainly on the IOMMUs implementation and allow their full standalone verification.

The proposed approaches have been applied in the verification of IOMMU as a part of microprocessor with the 6th generation «Elbrus» architecture, developed by "MCST". The developed test system and tests made it possible to detect and correct a number of logical errors that were not detected by other test methods.

References

[1] Intel Virtualization Technology for Directed I/O Architecture Specification. Intel, 2018.
[2] AMD I/O Virtualization Technology (IOMMU) Specification. AMD, 2016.
[3] Lebedev D.A., Stotland I.A. Construction of validation modules based on reference functional models in a standalone verification of communication subsystem. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 3, 2018, pp. 183-194. DOI: 10.15514/ISPRAS-2016-28(3)-10.

[4] Alkassar E., Cohen E., Kovalev M., Paul W.J. (2012) Verification of TLB Virtualization Implemented in C. Lecture Notes in Computer Science, vol 7152, pp 209-224.
[5] Alkassar, E., Cohen, E., Hillebrand, M., Kovalev, M., Paul, W. Verifying shadow page table algorithms. In Formal Methods in Computer Aided Design (FMCAD), 2010. pp. 267-270.
[6] Kamkin A., Kotsynyak A. Specification-Based Test Program Generation for MIPS64 Memory Management Units. Trudy ISP RAN/Proc, ISP RAS vol. 28, issue 4, 2016. pp. 99-114. DOI: 10.15514/ISPRAS-2016-28(4)-6.
[7] IEEE Standard for SystemVerilog — Unified Hardware Design, Specification, and Verification Language. IEEE Std 1800-2012.
[8] 1800.2-2017 - IEEE Standard for Universal Verification Methodology Language Reference Manual.

Информация об авторах / Information about authors

Антон Алексеевич ПЕТРЫКИН получил степень магистра в 2019 г. в Российском университете дружбы народов. В настоящее время работает инженером-программистом в АО «МЦСТ». Научные интересы включают верификацию компьютерных систем и подсистем микропроцессоров.

Anton Alekseevich PETRYKIN received a M.Sc. degree in 2019 in the Peoples' Friendship University of Russia. He is currently working as a software engineer at AO «MCST». Research interests include verification of computer systems and microprocessor subsystems.

Ирина Аркадьевна СТОТЛАНД в настоящее время работает начальником сектора автономной верификации микропроцессоров в АО «МЦСТ». Она получила степень кандидата технических наук в 2012 году. Область ее научных интересов включает автономную верификацию подсистем микропроцессоров и систем на кристалле.

Irina Arkadievna STOTLAND is currently working as microprocessor verification team lead at MCST. She received a PhD degree in 2012. Her research interests include standalone verification of microprocessor and SoC subsystems.

Алексей Николаевич МЕШКОВ получил степень кандидата технических наук в ИНЭУМ им. И.С. Брука в 2013 года. В настоящее время является начальником отдела в АО «МЦСТ». Область научных интересов включает программное моделирование и верификацию компьютерных систем.

Alexey Nikolaevitch MESHKOV recieved a PhD degree at INEUM in 2013. He is currently working as a chief of department at MCST. His research interests include software modeling and verification of computer systems.

DOI: 10.15514/ISPRAS-2019-31(3)-8

Digital Modelling of Production Engineering for Metalworking Machine Shops

V.P. Kotlyarov, ORCID: 0000-0003-3973-5218 <vpk@spbstu.ru>
A.P. Maslakov, ORCID: 0000-0001-7383-3917 <alex.maslakov.ftk@gmail.com>
A.A. Tolstoles, ORCID: 0000-0003-2327-906X <gmlaletol@gmail.com>
Peter the Great St. Petersburg Polytechnic University,
29, Politekhnicheskaya st., Saint Petersburg, 195251, Russia

Abstract. This article presents a modular approach that reduces the labor costs for the technological preparation of small-scale metalworking production. Its idea is to formalize the technological processes, allowing generating them and their documentation from pre-prepared parameterized templates stored in the special database. Details to be processed are represented as the structures of their basic geometric components. For the template of machining operations for each component, symbolic parameters are fixed, defining the workpiece used, cutting tools options, machining modes, etc. The result of formalization is an automatically generated technological route in the form of an MSC diagram encoding it as a sequence of macro-operations for the machinery. This symbolic model is adapted to a specific instance of the detail being manufactured by replacing the symbolic variables with specific values set by the technologist. The MSC diagram is supplemented with the results of time and cost calculations of technological routes, which allows selection of the most efficient one. The correctness of the technological routes is ensured in the process of symbolic verification by checking the permissible ranges of parameters of the MSC diagram, as well as checking the correctness of order and compatibility of operations in the sequence. The results of the whole process obtained from the MSC diagram are the set of technological documentation of preproduction, which, in particular, includes a set of operating cards, and the fine-tuned schedule of production after its digital modeling with the real resources of the workshop taken into account. According to technologists, by applying the described automation, the time to prepare documentation for details of medium complexity is reduced from several weeks to 1-2 days.

Keywords: adaptive manufacturing; production engineering; small-scale metalworking manufacturing preparation; automation of the preparation of technological documentation.

For citation: Kotlyarov V.P., Maslakov A.P., Tolstoles A.A. Digital Modelling of Production Engineering for Metalworking Machine Shops. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 3, 2019. pp. 85-98. DOI: 10.15514/ISPRAS-2019-31(3)-8

Acknowledgements. The work was financially supported by the Ministry of Education and Science of the Russian Federation in the framework of the Federal Targeted Program for Research and Development in Priority Areas of Advancement of the Russian Scientific and Technological Complex for 2014-2020 (14.584.21.0022, ID RFMEFI58417X0022).

Цифровое моделирование технологии производства металлообрабатывающих механических цехов

В.П. Котляров, ORCID: 0000-0003-3973-5218 <vpk@spbstu.ru>
А.П. Маслаков, ORCID: 0000-0001-7383-3917 <alex.maslakov.ftk@gmail.com>
А.А. Толстолес, ORCID: 0000-0003-2327-906X <gmlaletol@gmail.com>
Санкт-Петербургский политехнический университет Петра Великого,
195251, Россия, г. Санкт-Петербург, ул. Политехническая, д. 29

Аннотация. В данной статье представлен модульный подход, позволяющий снизить трудозатраты на технологическую подготовку мелкосерийного металлообрабатывающего производства. Его идея состоит в том, чтобы формализовать технологические процессы, позволяя генерировать их и их документацию из предварительно подготовленных параметризованных шаблонов, хранящихся в специальной базе данных. Обрабатываемые детали представлены в виде структур их основных геометрических компонентов. Для шаблона операций обработки для каждого компонента фиксируются символические параметры, характеризующие используемую заготовку, параметры режущих инструментов, режимы обработки и т. д. Результатом формализации является автоматически генерируемый технологический маршрут в виде диаграммы MSC, кодирующей его как последовательность макроопераций для станков. Эта символическая модель адаптируется к конкретной изготавливаемой детали путем замены символических переменных определяемыми технологом значениями. Диаграмма MSC дополняется результатами расчётов времени и стоимости исполнения технологических маршрутов, что позволяет выбрать из них наиболее эффективный. Корректность технологических маршрутов обеспечивается в процессе символьной верификации путем проверки допустимых диапазонов параметров диаграммы MSC, а также проверки правильности порядка и совместимости операций в последовательности. Результатом всего процесса, полученного из диаграммы MSC, является набор технологической документации на подготовку производства, который, в частности, включает в себя набор операционных карт, а также отлаженный график производства после его цифрового моделирования с учётом реальных ресурсов мастерской. По оценкам технологов, после применения описанной автоматизации время на подготовку документации для деталей средней сложности сокращается с нескольких недель до 1-2 дней.

Ключевые слова: адаптивное производство; технология производства; подготовка мелкосерийного металлообрабатывающего производства; автоматизация подготовки технологической документации.

Для цитирования: Котляров В.П., Маслаков А.П., Толстолес А.А. Цифровое моделирование технологии производства металлообрабатывающих механических цехов. Труды ИСП РАН, том 31, вып. 3, 2019 г., стр. 85-98 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(3)-8

Благодарности. Работы были профинансированы Министерством образования и науки Российской Федерации в рамках Федеральной целевой программы «Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2014-2020 годы» (№14.584.21.0022, ID RFMEFI58417X0022).

1. Introduction

Comprehensive automation of technological processes based on information technologies provides:

- reduction of the time of pre-production;
- optimization of labor costs and funds for the manufacturing of products;
- operational implementation of changes in the process under the external conditions (replacement of technological equipment, material, cutting tools, etc.) with automatic recalculation of the process characteristics.

Technological preparation of production (TPP) includes the following activities:

- setting of technological problems;
- selection of the workpiece based on its parameters;

- development of technological processing routes;
- selection of technological equipment;
- formation of technological operations;
- development of a set of technological documentation.

Fields of automation of technological preparation of production include:

- development of technological documentation;
- development of control programs;
- development of technological processes.

The tasks of operational planning and automated production management are carried out by the manufacturing execution systems (MES) [1]. They occupy an intermediate place in the hierarchy of enterprise management systems between the level of information collection from equipment in workshops done by supervisory control and data acquisition (SCADA) systems [2] and the level of operations over a large amount of administrative, financial and accounting information done by enterprise resource planning (ERP) [1] systems. The key processes for MES are as follows [3].

1) Based on the external demand for production (which, in turn, is based on customer orders, sales plans, etc.), as well as previous production programs, taking into account all sorts of nuances and specifics of production at a particular enterprise, a detailed optimized production schedule of works and operations for machine tools, equipment, personnel is produced. In addition, automatic generation of all the documentation necessary for the work: production programs, outfits, limit maps, tables and equipment loading diagrams, etc. is also done.
2) In the course of the direct implementation of production programs, full dispatching of all operations and their results (both positive and negative - rejects, delays, etc.) is carried out.
3) If deviations from the planned programs are identified due to the external reasons, or when new demand (orders, etc.) appears, real-time re-planning is performed with all components corrected accordingly.

It should be noted that there exists an imbalance between production time and preparation time in single or small-scale productions in case of re-scheduling of the work of a production site, because it should be performed for the small batch of the details and not for the whole series of them.

Nowadays on the Russian market there are three most popular largest solutions, the products of many years of work of three scientific centers for developing systems of this class: PHOBOS system, YSB.Enterprise.Mes system and PolyPlan system. PHOBOS is traditionally used in large and medium-sized machine-building enterprises. YSB.Enterprise.Mes originated from the woodworking industry and focuses on the sector of medium and small enterprises. The PolyPlan system has a smaller set of MES functions, but is positioned as an operational scheduling system for automated and flexible manufacturing in engineering [3].

However, with all the attractiveness of such systems, due to the extensive set of functions provided and deep integration into the production processes in the enterprise at all stages, their practical implementation is a whole complex and expensive project in itself that not all enterprises, especially small-scale and individual productions, can afford. In addition to this, in order to work effectively with MES, high qualification of its operator is required. The automated workplace of the technologist given in this article is designed to solve a narrower class of problems - to simplify the TPP for small-scale machine-building production, it does not require interactions with other systems, and the results in the form of the required schedule of work distribution and a set of operating cards can be obtained in a couple of days of work of a technologist.

2. Formalization of the technological process

Let's look through the features of formalization based on an illustration of a specific example of work with the developed system of an automated workplace for a technologist.

The main input information for the technologist is the detail drawing. It can be done in any graphical design program, for example, in KOMPAS-3D [4]. The example of the drawing is demonstrated in the fig. 1.

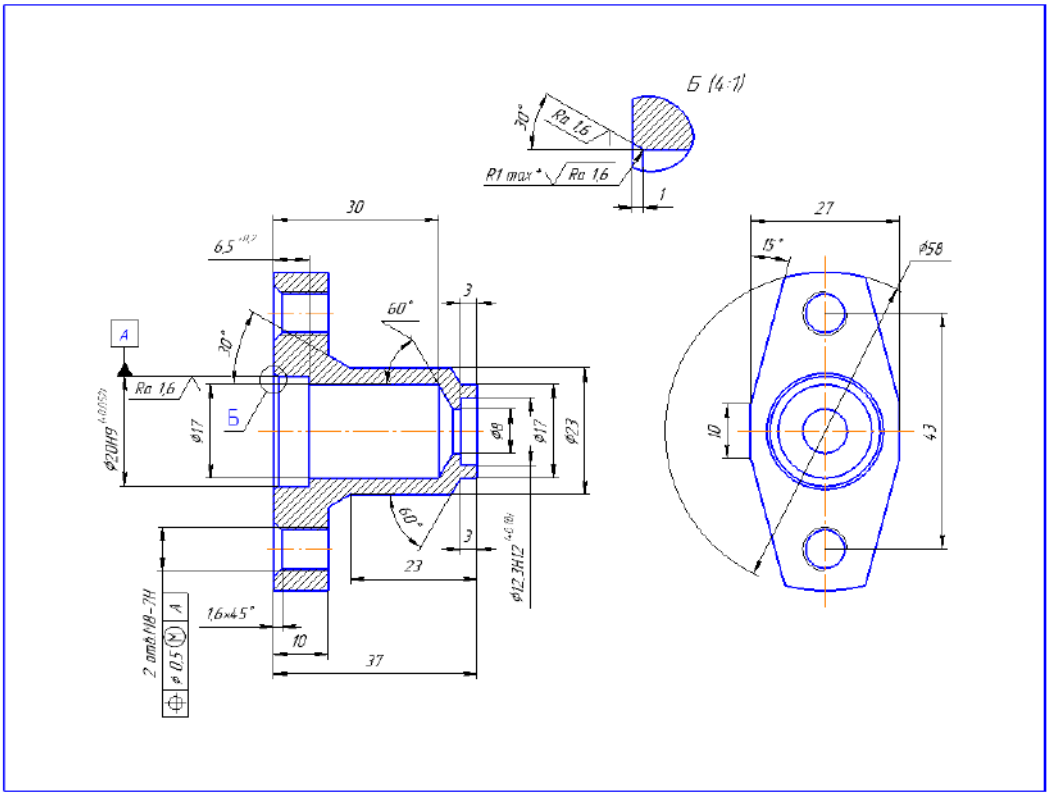

Fig. 1. Input detail drawing

To manufacture a part technologist chooses a workpiece for it. Several such workpieces may be selected; to determine the best fitting of them, all calculations of time and cost of production must be made for each selected one and compared to each other.

The next action of the technologist is the splitting of a given drawing into a set of sketches of elementary surfaces (ES), the parts of the detail. This step sets the way for the modular approach to the production technology [5]. Each of the surfaces is characterized by geometrical parameters and the number of stages required to process it. The processing stage is the smallest atomic operation, for example, turning, drilling or milling. The parameters of the processing stage are the types of machines on which it can be performed, the cutting tools to do so and the selected workpiece in the very beginning.

The form for setting information about the processing stages manually is shown in the fig.2.

The description of fields and tables of the form is as follows, from left to right and from up to down:

- the type of the elementary surface encoded by two digits;
- the unique number of the elementary surface used to distinguish between the surfaces of the same type;
- the geometrical parameters of the elementary surface, here the diameter and length are listed;
- the amount of the processing stages to be performed;

- the processing stages, divided into the following columns: the number of the stage, the name of it (here: turning), the codes of the applicable machinery for it, the codes of the applicable cutting tools for it (which are described in the table under this one), the amount of the allowance (here: determined by the chosen workpiece) and the name of the selected workpiece (here: the first one);
- the cutting modes, divided into the following columns: the code of the cutting tool, the type of it (here: cutter), the three technical characteristics of each cutting tool with maximum and minimum values and the minimum and maximum durability of the cutting tool.

Тип ЭП 1/1 Номер ЭП 1

Параметры ЭП

$D_{инд.}$	17
$l_{инд.}$	3

Количество ЭП

$Q_{эп}$	1

Этапы обработки

№ этапа	Метод обработки	Технологическое оборудование	Режущий инструмент	Величина припуска	Применяемость
1	Точение	Т01, Т02, Т03, Т04, Т05	Т(1-1), Т(1-2), Т(1-3)	Заг.	Заг.1

Режимы резания

Режущий инструмент	Тип РИ	Режимы резания						Стойкость РИ	
		V_{min}	V_{max}	S_{min}	S_{max}	t_{min}	t_{max}	min	max
Т(1-1)	резец	210	320	0,2	0,5	1	4	15	30
Т(1-2)	резец	230	310	0,1	0,45	1,5	4,5	20	35
Т(1-3)	резец	250	370	0,15	0,45	0,75	3,5	25	40

Fig. 2. Processing stage information form

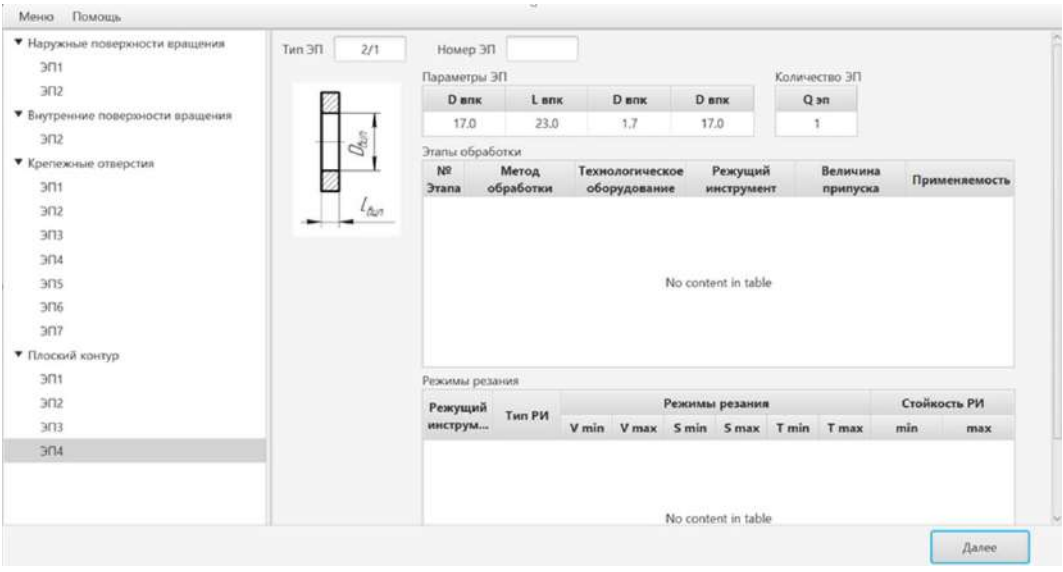

Fig. 3. The window for setting the information about an elementary surface

To translate it into a digital form, a developed solution is used, the set of user interface screens of which forms an automated workplace for the technologist (fig.3). The fields and tables on the right are essentially the same; the left side shows the sorted list of the already loaded surfaces by types: inner surfaces of revolution, outer surfaces of revolution, mounting holes and flat contour. There is also a place for the sketch of the surface in the middle.

Each cutting tool added by the technologist is characterized by its cutting modes. The parameters of cutting modes affect the running time and its cost. Usually, data for the cutting tool is taken from reference catalogs in *.pdf format [6]. The user interface allows the technologist to simplify entering data from catalogs through the use of hotkeys: after selecting data in the document and pressing the CTRL + SHIFT + C key combination, the data is copied directly into the table. This approach reduces the labor intensity of the manual data transfer and helps to avoid the human factor such as errors or typos.

To determine the order of processing of elementary surfaces, further formation of blocks of elementary surfaces from them takes place. Each block is characterized by its own positioning data on the machine. The window for creating blocks of surfaces is shown in fig.4. The left side of it shows the list of the blocks with the button "Create new block" at the very bottom of it, the rows on the right side consist of the surfaces corresponding to each block.

The next step of the formalization of technological process is the formation of groups of elementary surfaces inside blocks of elementary surfaces. Such group is a part of the block that can be processed in one operation without reinstalling the workpiece into the machine. Thus, the nesting hierarchy is created (fig.5).

The operation on a group of surfaces made up from initial operations on each surface is called a machining step, each one of them has its own physical meaning, for example, turning the outer surface of revolution, drilling through hole or boring the hole. All cutting tools for all elementary surfaces within a group must be the same. The window for creating groups is shown in the fig.6. The three tabs on the left are created for each block; they hold lists of groups of surfaces within the block. The right side shows the elementary surfaces of each selected group with their parameters.

In addition to the windows for filling in the information, the user interface has a menu containing “Help” section. There is a reference catalogs searching tool which works in conjunction with a system application for viewing files in *.pdf format and is capable of two types of searches:

- The window for keyword search in catalogs is shown in Fig.7. After entering keywords in the top field and selecting catalogs for search in the list, by pressing the leftmost button a search is performed on the selected documents. For each catalog, the following sequence of actions is carried out.
  1) One page of document is read from disk.
  2) Search for keywords is performed on this page.
  3) If at least one of the keywords is detected, the page is copied into the resulting PDF document.
  4) If the keywords are not found, proceed to the next page.

  As soon as all pages of all catalogs are processed, a resulting document with search results is written to disk and opened in the standard PDF document viewer in the system. The right button cancels the search, the bottom one allows technologist to add a catalog to the list.
- Search by image, in contrast to search by keywords, is possible only for catalogs formatted in advance. Its interface is shown in Fig.8. After selecting a PDF catalog from the drop-down list, if the necessary markup information exists for it, images, for example, of surfaces to be processed, are shown. By clicking on them a document containing information related only to the selected images is formed and, alike to the search by keywords, is opened in a standard PDF documents viewer. The button on the right allows technologist to add a catalog to the list.

The usage of these searching tools, especially in conjunction with copying data into tables with hotkeys, achieves a significant reduction in the complexity of data entry for elementary surfaces.

Fig. 4. The window for creating blocks of elementary surfaces

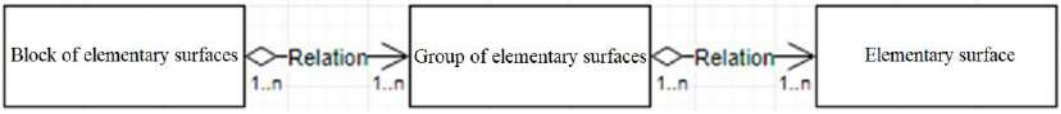

Fig. 5. The nesting hierarchy of elementary surfaces

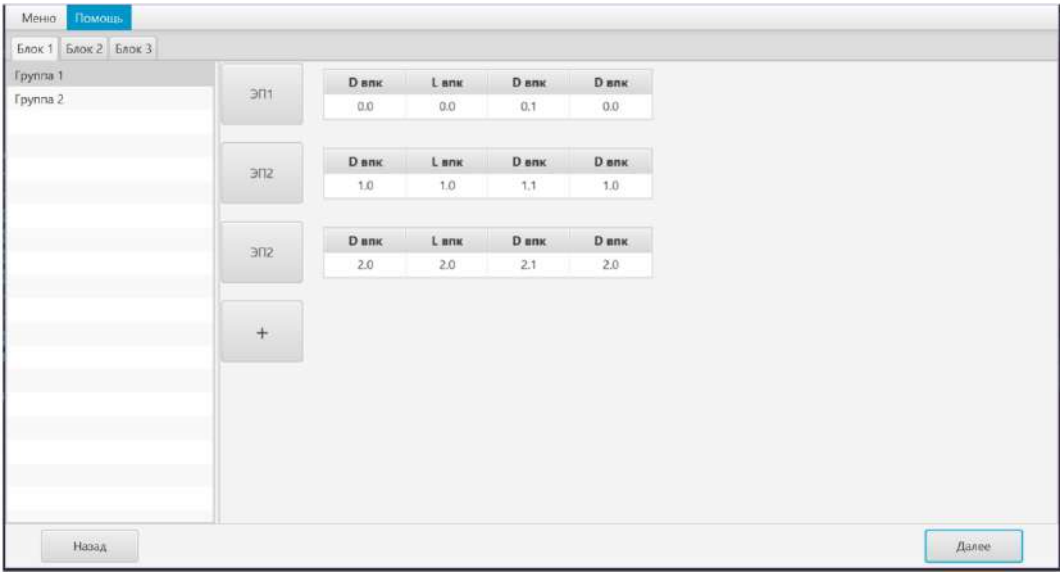

Fig.6. The window for creating groups of elementary surfaces

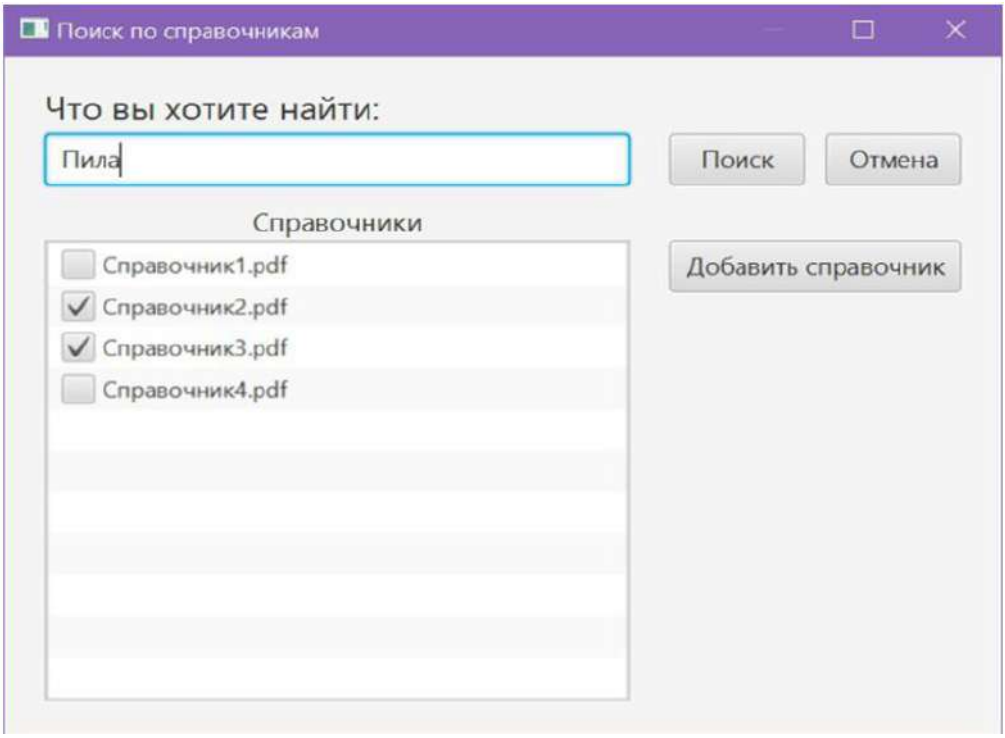

Fig. 7. Keywords search user interface

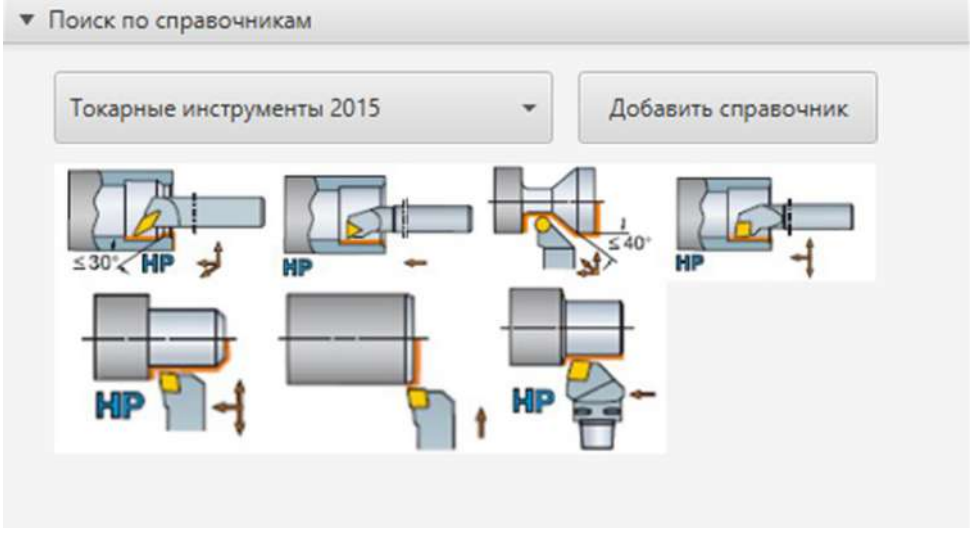

Fig. 8. Image search user interface

A special database based on PostgreSQL database management system [7] is used to store information entered by the technologist in the user interface [8].

The fig. 9 shows a fragment of its tables, where:

- «public.tb_methods» table stores information for calculating the time and cost of processing methods for elementary surfaces;
- «public.tb_app_machines» is dedicated to applicable machines for processing methods;

- «public.tb_processing_steps» stores the parameters of the processing methods;
- «public.tb_app_tools» holds information about the applicable cutting tools and cutting modes;
- the four lowest tables are used for linkage between other tables.

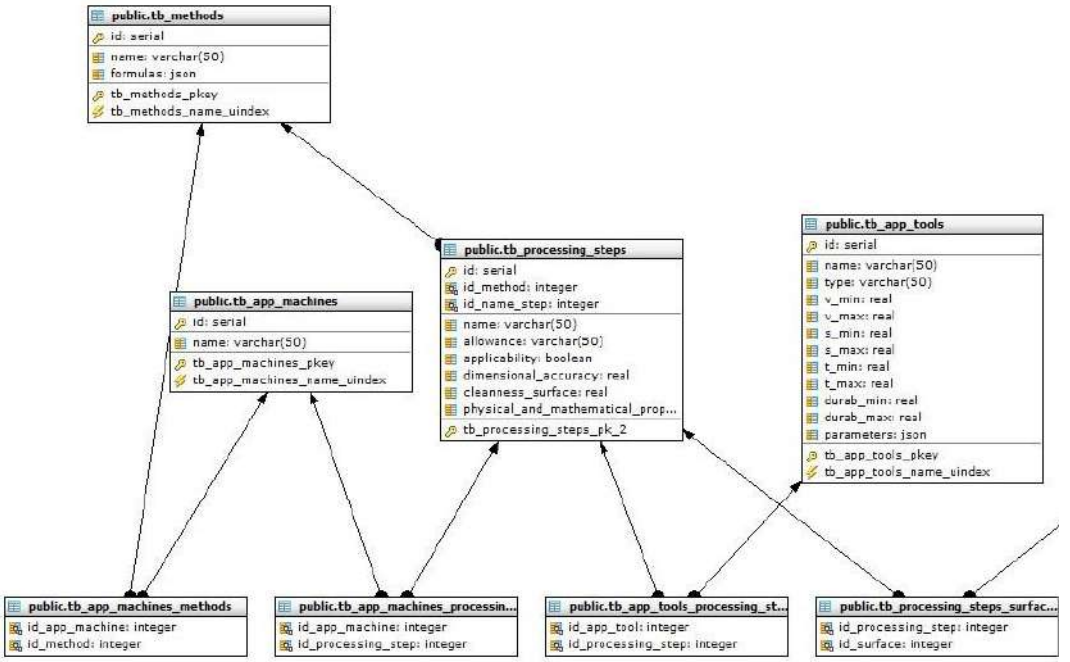

Fig. 9. Database fragment

To formulate the resulting technological route for the processing of the whole detail technologist must determine all the groups of elementary surfaces that can be processed together. There can be several routes constructed this way, the choice of the one is made based on which machinery and which tools are available and should be used. In the approach presented here we use the MSC language [9] for the encoding of the route. MSC is a standardized language for describing behaviors using message exchange diagrams between parallel-functioning objects (machines, robots). The main unit of the diagram is a line starting with a name of a processing stage of elementary surface followed by its parameters. To construct such line, only the index parameters of the stage are used, insofar as all other necessary data can be obtained from the database based on them. Such index parameters include:

- the number of the processing stage;
- the code of the type of the elementary surface, in two digits;
- the number of the elementary surface within the same type;
- the codes of the applicable machinery for the processing stage;
- the codes of the applicable tools for the processing stage;
- the code of the workpiece used;
- the number of the block of elementary surfaces which this elementary surface corresponds to;
- the index number of the elementary surface within the block;
- the number of the group of elementary surfaces within the block;
- the index number of the elementary surface within the group;

The resulting parameterized line takes the following form, for example:

Turning(stageNumber, surfaceType1, surfaceType2, surfaceNumber, [machine1, machine2, machine3, machine4, machine5], [cuttingTool_1_1, cuttingTool_1_2, cuttingTool_1_3], workpieceParams.code, blockParams.number, numberInBlock, groupNumber, numberInGroup);

The diagram comprises a set of these lines in order set by technologist earlier. The correctness of the technological routes is ensured in the process of symbolic verification, which checks the acceptable ranges of parameters of the diagram, as well as the correctness of order of the whole sequence [10]. The actual data is taken from the database and substituted instead of parameters.

3. The usage of the formalized technological process

The MSC diagram of the route is supplemented with the results of calculating the time and cost of each processing stage. The calculations use formulas stored in the database, they are partially shown in Table 1 and Table 2. The individual results for each processing stage of the route are summarized, which gives an estimate of the total time and cost of the technological route.

Table 1. Formulas for turning time calculations

Formulas	Parameters description
$T_m = \frac{L}{n \cdot s} \cdot i$	T_m - machining time L - estimated length of processing in mm n - workpiece rounds per minute s - cutter feed per round in mm i - the number of passes of the cutter
$L = l + l_1 + l_2$	l - the length of the workpiece in the feed direction, mm $l1$ - cutting-in length of the tool $l2$ - the length of the tool exit, mm
$n = \frac{1000 \cdot v}{\pi \cdot d}$	v - the speed of the cutting, mm per minute d - the diameter of the processed workpiece, mm
$i = \frac{h}{t}$	h - the amount of overmeasure in mm t - cutting depth in mm

Table 2. Formulas for drilling time calculations

Formulas	Parameters description
$T_m = \frac{L}{n \cdot s}$	T_m - machining time L - estimated length of processing in mm n - workpiece rounds per minute s - cutter feed per round in mm
$L = l + l_1 + l_2$	l - the length of the hole, mm $l1$ - cutting-in length of the tool $l2$ - the length of the tool exit, mm
$l_1 = \frac{d_t}{2} \cdot ctg(\phi)$	drilling in the solid material ϕ - the main angle in the plan, grad d_t - the diameter of the tool
$n = \frac{1000 \cdot v}{\pi \cdot d_t}$	v - the speed of the drilling, mm per minute d_t - the diameter of the tool, mm

By changing the route parameters and recalculating the measurements, technologist can choose the most effective one.

The selected technological route thus meets the criteria for the time and cost but yet does not take into account the conditions and resources of the workshop in which it will be implemented. For this, it is necessary to use simulation modeling of its performance on the equipment in the workshop. To use the developed simulation algorithm, technologist inputs three files describing following specifications.

- The composition of workshop resources (CNC machines, robots, maintenance personnel, etc.). The types of operations for each machine that it can perform are defined.
- The planned technological routes. The number of manufactured parts and the sequence of operations with the amounts of time of their execution are determined for each route.
- The priorities of the routes and resources used, as well as the initial state of the workshop equipment.

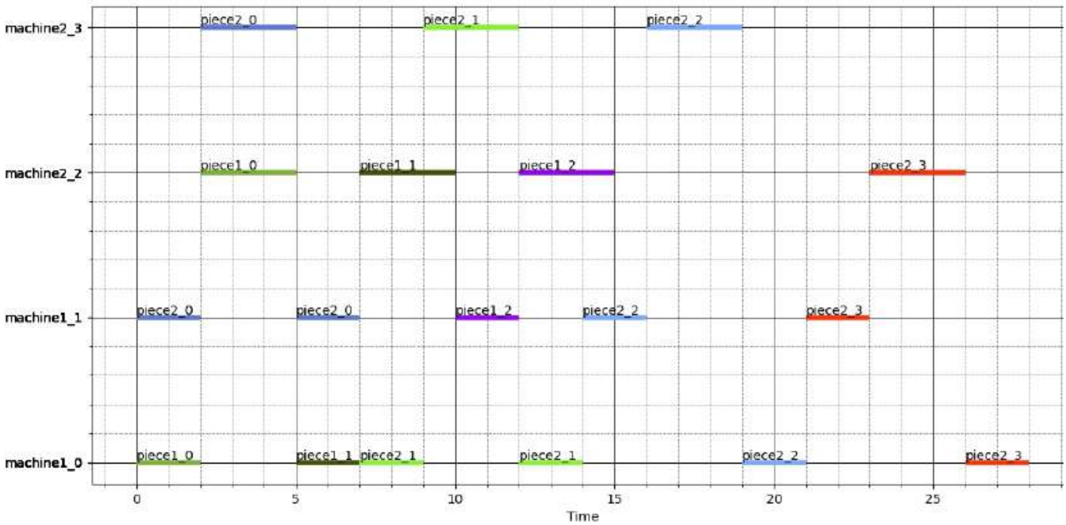

Fig. 10. Timing diagram fragment

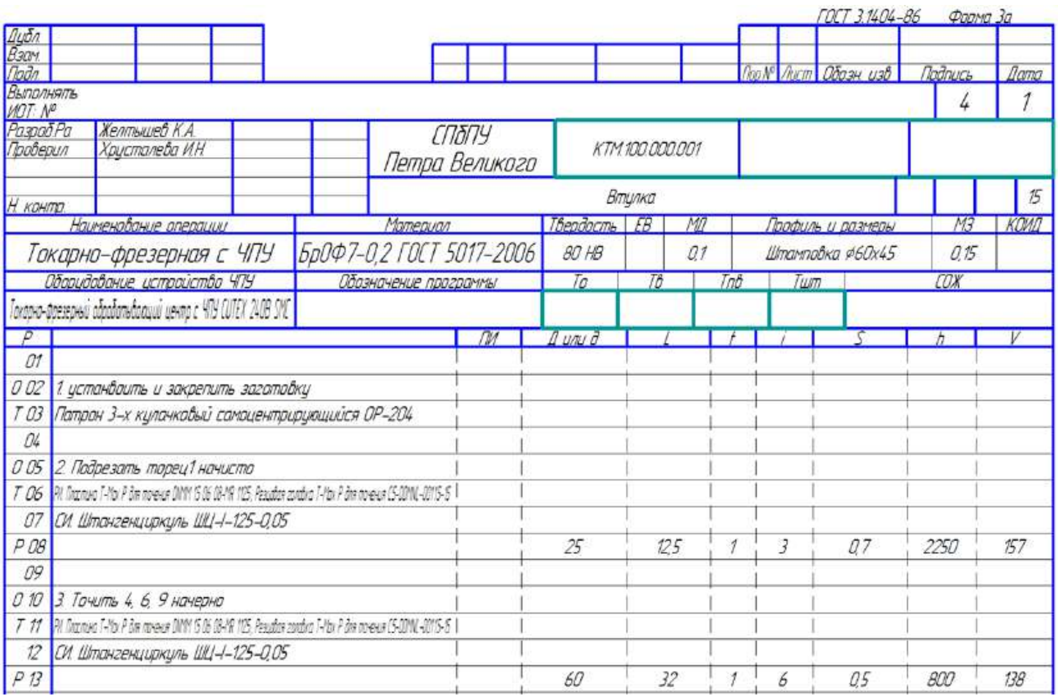

Fig. 11. Operating card

The result of the simulation is a timing diagram of the distribution of operations by each machine in the form of a Gantt chart, a fragment of it is shown in the fig.10.

Modeling allows technologist to take into account equipment downtime, the additional cost of transporting parts and machine changeover. As a result, the estimation of the time and cost of the technological route becomes more realistic. By changing the priorities of the technological routes technologist obtain several options for implementing the technological process of the workshop. By applying a hierarchy of criteria measuring the success of the work such as time, cost, equipment loading, material savings, etc. various problems of multicriteria optimization can be solved [11].

For each selected optimized version of the technological route, technological documentation of production preparation is automatically generated in the form of the operating card [12], its example is shown in fig.11 [13].

4. Conclusion

The paper considers the problem of technological preparation of single and small-scale production, which area is characterized by imbalance of work between preparation and implementation of production. The approach to its automation based on modular technology is proposed.

The important properties of automation system are demonstrated:

- the ability to adapt to specific production conditions such as different equipment, resources, orders and support staff;
- significant reduction of the complexity of creating a technological route for an order using a special automated workplace for the technologist;
- operational planning and scheduling of the technological process of the workshop;
- selection of the optimal characteristics of production processes during hierarchical multi-criteria optimization.

According to existing estimates the platform provides a multiple increase in productivity and a reduction in labor intensity and in amount of time of the preparation of technological documentation for engineering production.

References / Список литературы

[1]. Frolov E.B. Zagidullin R.R-b. MES as they are or the evolution of the production planning systems. Part II. 2007. Available at: http://www.fobos-mes.ru/stati/mes-sistemyi-kak-oni-est-ili-evolyutsiya-sistem-planirovaniya-proizvodstva.-chast-ii.html, accessed 14.07.2019 (in Russian) / Фролов Е.Б., Загидуллин Р.Р-б. MES-системы, как они есть или эволюция систем планирования производства. Часть II. 2007 г.

[2]. Davidyuk Y. SCADA systems at the top level of the advanced process control systems. Intelligent Enterprise, vol. 30, no. 13, 2001. Available at: https://www.iemag.ru/platforms/detail.php?ID=16479, accessed 14.07.2019 (in Russian) / Давидюк Ю. SCADA-системы на верхнем уровне АСУТП. Intelligent Enterprise, том 30, вып. 13, 2001 г.

[3]. Garaeva Y., Zagidullin R.R-b., Tsin S.K. Russian MES, or how to bring back optimism to production. CAD and graphics, vol. 11, 2005. Available at: https://sapr.ru/article/14614, accessed 14.07.2019 (in Russian) / Гараева Ю., Загидуллин Р.Р-б., Цин С.К. Российские MES-системы, или Как вернуть производству оптимизм. САПР и графика, том 11, 2005 г.

[4]. Statsenko D. Applications and work without stress. Tendency, vol. 1, no. 18, 2017. Available at: https://kompas.ru/source/articles/3.pdf, accessed 14.07.2019 (in Russian) / Стаценко Д. Приложения и работа без напряжения. Стремление, том 1, вып. 18, 2017 г.

[5]. Bazrov B.M. Modular technology in mechanical engineering. Moscow, Mechanical engineering, 2001, 366 pp. (in Russian) / Базров Б.М. Модульная технология в машиностроении. М., Машиностроение, 2001, 336 с.

[6]. SANDVIK COROMANT. Tools and equipment for turning on machines. 2015, 1253 pp. Available at: http://www.lab2u.ru/katalog-sandvik-coromant-2015-metallorezhushchii-tokarnyi-instrument-i-

instrumentalnaia-osnastka-dlia-tocheniia-obrabotki-kanavok-otrezki-rezbonarezaniia-reztcy-so-smennymi-rezhushchimi-plastinami-iz-tverdogo-splava-kompanii-sandvik-koromant-lab2u.html, accessed 14.07.2019 (in Russian) / SANDVIK COROMANT. Инструмент и оснастка для точения на станках. 2015, 1253 с.

[7]. The PostgreSQL Global Development Group. Postgres Pro Standard 11.4.1 Documentation, 2019. Available at: https://postgrespro.ru/docs/postgrespro/11/index, accessed 14.07.2019.

[8]. Cherepovskii D.K., Eizenakh D.S., Kotlyarov V.P. The database architecture for the creation of the technological routes for the small-scale production. In Proc. of the Modern technologies in the theory and practice of programming conference, St. Petersburg, 2019, 3 p. (in Russian) / Череповский Д.К., Эйзенах Д.С., Котляров В.П. Архитектура базы данных для создания технологических маршрутов мелкосерийного производства. Материалы конференции «Современные технологии в теории и практике программирования», Санкт-Петербург, 2019 г., 3 с.

[9]. Recommendation ITU_T Z.120. Message Sequence Chart (MSC), 2011, 146 pp. Available at: https://www.itu.int/rec/T-REC-Z.120-201102-I, accessed 14.07.2019.

[10]. Baranov S., Kotlyarov V., Letichevsky A., Drobintsev P. The Technology of Automation Verification and Testing in Industrial Projects. In Proc. of the St. Petersburg IEEE Chapter International Conference, St. Petersburg, Russia, May 18-21, 2005. pp. 81-86.

[11]. Voinov N., Chernorutsky I., Drobintsev P., Kotlyarov V. An approach to net-centric control automation of technological processes within industrial IoT systems. Advances in Manufacturing, vol. 5, no. 4, 2017, pp. 388-393.

[12]. Eizenakh D.S., Cherepovskii D.K., Kotlyarov V.P. The system for generation of the operating card of the technological process for a small-scaled mechanical engineering enterprise. In Proc. of the Modern technologies in the theory and practice of programming conference, St. Petersburg, 2019, 46 p. (in Russian) / Эйзенах Д.С., Череповский Д.К., Котляров В.П. Система генерации операционной карты технологического процесса для мелкосерийного машиностроительного производства. Материалы конференции «Современные технологии в теории и практике программирования», Санкт-Петербург, 2019 г., 46 с.

[13]. GOST 3.1404-86 Unified system for technological documentation (USTD). Forms and rules for paperwork on technological processes and machining operations. 1987. Available at: http://docs.cntd.ru/document/1200012135, accessed 14.07.2019 (in Russian) / ГОСТ 3.1404-86 Единая система технологической документации (ЕСТД). Формы и правила оформления документов на технологические процессы и операции обработки резанием. 1987 г.

Информация об авторах / Information about authors

Всеволод Павлович КОТЛЯРОВ получил степень кандидата технических наук в 1973 году в Ленинградском политехническом институте (ныне Санкт-Петербургский политехнический университет Петра Великого). Область научных интересов включает автоматизацию промышленных технологий разработки больших программных систем и сетей, программную инженерию требований и спецификаций, технологии интеграции верификации и тестирования.

Vsevolod Pavlovich KOTLYAROV received a Ph.D. degree at Leningrad Polytechnic Institute (now Peter the Great St. Petersburg Polytechnic University) in 1973. His research interests comprise automation of industrial technologies for the development of large software systems and networks, software engineering in area of requirements and specifications, technologies for integrations of verification and testing.

Алексей Павлович МАСЛАКОВ получил степень магистра в области информационных технологий в 2015 году в Санкт-Петербургском политехническом университете Петра Великого, Санкт-Петербург, Россия. Исследовательские интересы включают верификацию программного обеспечения, генерацию исполнимого кода, Интернет Вещей, цифровизацию производства.

Alexey Pavlovich MASLAKOV received the M.S. degree in information technology in 2015 at Peter the Great St. Petersburg Polytechnic University in St. Petersburg, Russia. His research

interests include software verification, generation of executable code, Internet of Things, digitalization of production.

Алексей Андреевич ТОЛСТОЛЕС получил степень магистра в области информационных технологий в Санкт-Петербургском политехническом университете Петра Великого в 2017 году. В настоящее время готовит диссертацию на соискание степени кандидата технических наук в том же университете в Высшей школе программной инженерии. Исследовательские интересы включают автоматизацию технологических процессов, промышленный Интернет Вещей и back-end разработку.

Alexey Andreevich TOLSTOLES received the M.S. degree in information technology at Peter the Great St. Petersburg Polytechnic University in 2017. He is currently pursuing the Ph.D. degree in the Higher school of software engineering of the same university. His research interests include automation of technological processes, industrial Internet of Things and back-end development.

DOI: 10.15514/ISPRAS-2019-31(3)-9

Reputation Systems in E-commerce: Comparative Analysis and Perspectives to Model Uncertainty Inherent in Them

M. M. Nosovskiy, ORCID: 0000-0003-4475-3787 <mmnosovskiy@edu.hse.ru>
K. Y. Degtiarev, ORCID: 0000-0001-5519-1033 <kdegtiarev@hse.ru>
National Research University Higher School of Economics (HSE),
Faculty of Computer Science, School of Software Engineering,
3, Kochnovsky Proezd, Moscow, 125319, Russian Federation

Abstract. E-commerce is a runaway activity growing at an unprecedented rate all over the world and drawing millions of people from different spots on the globe. At the same time, e-commerce affords ground for malicious behavior that becomes a subject of principal concern. One way to minimize this threat is to use reputation systems for trust management across users of the network. Most of existing reputation systems are feedback-based, and they work with feedback expressed in the form of numbers (i.e. from 0 to 5 as per integer scale). In general, notions of trust and reputation exemplify uncertain (imprecise) pieces of information (data) that are typical for the field of e-commerce. We suggest using fuzzy logic approach to take into account the inherent vagueness of user's feedback expressing the degree of satisfaction after completion of a regular transaction. Brief comparative analysis of well-known reputation systems, such as EigenTrust, HonestPeer, Absolute Trust, PowerTrust and PeerTrust systems is presented. Based on marked out criteria like convergence speed, robustness, the presence of hyper parameters, the most robust and scalable algorithm is chosen on the basis of carried out sets of computer experiments. The examples of chosen algorithm's (PeerTrust) fuzzy versions (both Type-1 and Interval Type-2 cases) are implemented and analysed.

Keywords: e-commerce; reputation system; peer-to-peer computing; trust management; uncertainty; fuzzy logic; linguistic variable; type-1 fuzzy set; type-2 fuzzy set

For citation: Nosovskiy M.M., Degtiarev K.Y. Reputation Systems in E-commerce: Comparative Analysis and Perspectives to Model Uncertainty Inherent in Them. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 3, 2019. pp. 99-122. DOI: 10.15514/ISPRAS-2019-31(3)-9

Репутационные системы в электронной коммерции: Сравнительный анализ и перспективы моделирования присущей им нечеткости

М.М. Носовский, ORCID: 0000-0003-4475-3787 <mmnosovskiy@edu.hse.ru>
К.Ю. Дегтярев, ORCID: 0000-0001-5519-1033 <kdegtiarev@hse.ru>
Национальный исследовательский университет "Высшая школа экономики",
факультет компьютерных наук, департамент программной инженерии,
125319, Россия, г. Москва, Кочновский пр-д, д. 3

Аннотация. В наши дни электронная коммерция (ЭК) показывает беспрецедентные темпы роста во всем мире, вовлекая в эту деятельность миллионы людей на всех континентах. В то же время, ЭК создает почву для злонамеренных действий, что требует особого внимания и контроля. Одним из способов минимизации таких угроз является использование репутационных систем для отслеживания степени доверия в среде пользователей сети. Большинство существующих репутационных систем

основаны на сборе отзывов относительно проведенных транзакций, и они, как правило, работают с представленными в виде чисел откликами клиентов (в частности, может использоваться привычная целочисленная шкала 0..5). В целом, понятия доверия и репутации являются примерами неопределенных (неточных) информационных данных, характерных для сферы электронной коммерции. Мы предлагаем использовать аппарат нечеткой логики для формального представления пользовательских отзывов, выражающих степень удовлетворенности результатом совершенных транзакций. В работе представлен краткий сравнительный анализ наиболее известных репутационных систем, таких как EigenTrust, HonestPeer, Absolute Trust, PowerTrust и PeerTrust. С учетом выделенных в результате анализа критериев (скорость сходимости, устойчивость (робастность), наличие гиперпараметров), проведенная серия компьютерных экспериментов позволила эмпирически выделить PeerTrust как наиболее устойчивый и масштабируемый алгоритм из числа рассмотренных. При наличии ограничений в отношении имеющихся данных, подготовлены реализации (Python 3.7) и проанализированы результаты, связанные с особенностями поведения нечетких версий алгоритма PeerTrust на основе нечетких множеств типа-1 (T1FS) и интервальных нечетких множеств второго типа (IT2FS).

Ключевые слова: электронная коммерция; репутационная система; пиринговые вычисления; управление доверием; нечеткость; нечеткая логика; лингвистическая переменная; нечеткое множество 1-го типа; нечеткое множество 2-го типа

Для цитирования: Носовский М.М., Дегтярев К.Ю. Репутационные системы в электронной коммерции: Сравнительный анализ и перспективы моделирования присущей им нечеткости. Труды ИСП РАН, том 31, вып. 3, 2019 г., стр. 99-122 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(3)-9

1. Introduction

E-commerce is a buying-selling runaway activity widening at an unprecedented rate all over the world and inveigling into fascination of various e-stores people of all ages. Ever-growing number of various websites and apps focusing on e-commerce domain makes it simple and alluring to find and to buy immediately almost anything whatever client's heart desires [1].

There is no doubt that e-commerce sales opportunities are rapidly progressing day by day. Owing to Internet, businesses bring their products and services to customers literally in eyewink. The e-commerce share of total retail sales in the United States amounted to 10% in 2018, in expectation of attainment of 12.4% by 2020 with further strengthening its ground [2]. With such perspectives in mind it is easy to realize why e-commerce entrepreneur position becomes so attractive. With an estimated 95% of purchases that will be made online by 2040 and expected growth of year to year sales standing at the level of 15%, the opportunity to find a niche for selling products online has massive indisputable potential [3]. During the last 5 years the amount of retail sales raised from $1.3 billion to $2.8 billion. The latter is expected to nearly double (up to $4.8 billion) by the end of 2021 [4].

One of the most growing types of e-commerce is *online marketplace* that can be defined as a website or app that facilitates shopping from many different sources [5]. Among well-known and successful examples of online marketplaces eBay, Amazon, Rakuten (worldwide) and Avito, Ozon (in Russia) can be mentioned. Online marketplace acts as a platform integrating buyers and sellers. Being a peer-to-peer (P2P) network, it allows buyers to purchase any goods or services offered by sellers through this online platform. Usually, peers (people or businesses) communicating through online marketplace remain in the status 'strangers' with respect to each other. They don't have at their disposal reliable information about alter peer, whether it is a buyer or a seller. Therefore, peers must manage the risk associated with transactions on condition that no prior experience and knowledge concerning mutual reputation of sides exists [6]. This problem can be addressed by means of developing a system on top of the network that should help peers to evaluate their past experience with other peers and to manage trust between them as well as reputation of each peer involved. This kind of systems is called *reputation systems*.

Various implementations of reputation systems exist starting with very simple to more complex ones designed mostly for P2P file-sharing networks [7-10]. Such systems have a positive impact

on peer's experience as they help to distinguish trustworthy peers from ill-intentioned and unreliable opponents. For example, in reputation system used by eBay, one of e-commerce leaders, buyers and sellers have a chance to rate each other with numeric scores +1, 0 or -1 after each carried out transaction. The overall reputation of a participant is calculated as a sum of scores earned over last six months [8]. At that all such systems rest upon notions of *trust* and *reputation*. Trust (or, local trust) represents personal experience (attitude) of a user regarding another user, while reputation constitutes an aggregate of these individual trust values on the scale of the whole community. Calculation of local trust and corresponding aggregates underlies implementations of all known reputation systems.

Despite the practical effectiveness of these systems, there is a substantial drawback inherent in them, viz. none of them can handle uncertainty "hidden" in online marketplace's data. The latter means data that relate to all transactions accomplished on the marketplace along with data collected from users after each transaction and metadata concerned with every user in the marketplace.

The primary concern of the paper is to provide the overview of best known reputation systems and to undertake their general comparative analysis on the basis of several key factors (criteria) – they are speed of convergence, complexity of calculations, use of hyper parameters expressing user's preferences, robustness and general system's suitability to handle imprecision and uncertainty of data. In the first place these factors are chosen to convey the requirements of key stakeholders who are owners and developers of a marketplace as well as its users. For the first group of stakeholders general system's effectiveness becomes important, and it is attributed above all to the efficiency of its functioning, computational resources needed to perform the work and ability for customization. Users are mostly interested in reliability of system's output and how well it suits each given user. The last factor mentioned above reflects how naturally specific implementation of the system can be extended to handle data uncertainty and imprecision, since the latter being an inherent part of virtually any system reveals itself in different forms. The recognition of such manifestation forms of uncertainty becomes a task of prime importance to represent appropriately (model) its distinctiveness. Consequently, fuzzy logic is getting one of pivotal theories that captures naturally the phenomenon of imprecision and uncertainty [11].

The rest of the paper is organized as follows: in section 2 notions of trust and reputation, difference between them, are considered. Uncertainty in the marketplace and verbal assessments that are inherent in reputation systems form the contents of section 3. Some basic terms and definitions relating to the field of fuzzy sets and logic are covered in the section 4. Section 5 is devoted to the brief comparison of five well-known reputation systems (EigenTrust, Absolute Trust, PeerTrust, et al.) and stressing their key differences as well as intrinsic similarities. Setup of computer-based experimental part of the work (parameters and their values used) constitutes the material of section 6, whereas the results of carried out experiments are discussed in the section 7. Thereafter, the transition from crisp to type-1 and interval type-2 fuzzy PeerTrust algorithm (analysis of such transition's outcome) is presented in finishing sections 8 and 9. Concluding remarks and observations are drawn in section 10.

2. Trust and reputation. What is the difference between these terms?

Trust and reputation are the main concepts underlying vast majority of reputation systems. In order to clearly recognize the purpose of reputation systems, we need to define what do trust and reputation in terms of online marketplace stand for. Diverse sources give different definitions of the term 'trust'. The basic definition presented in Oxford English Dictionary reads as follows: «Trust is a firm belief in the reliability, truth, or ability of someone or something» [12]. However, such definition cannot lay claim to completeness, since notions of trust and reputation as applied to peculiarities of Internet-based activities must be defined in a more context-specific way. Among other things, Alam & Paul define trust as «a belief, the trusting agent has in the trusted agent's willingness and capability to deliver the services that they are mutually agreed on in a given

context and in a given time slot» [13]. In addition, Wang & Vassileva associate term 'trust' with «a peer's belief in another peer's capabilities, honesty and reliability based on its own direct experiences» [14]. Starting from individual judgments and predictions, Gambetta state that «… trust is a subjective probability that relies on context and reputation, it describes how secure a situation is even though risk is associated with it» [15]. It can be noticed that trust is mainly linked to belief that peers (agents) mentally possess in malicious P2P environment. Thus, trust can be viewed as a soft system's factor that is difficult to express precisely and in complete form. It is tied to distinction of numerous generally inhomogeneous interactions between peers, organization of the network, in which humans play a pivotal role.

For reputation term situation seems resembling, i.e. there is also no conventional definition that most of sources agree on. According to [14], reputation is defined as «peer's belief in another peer's capabilities, honesty and reliability based on recommendations received from other peers». On the other hand, already cited above Alam & Paul propose to consider reputation as «aggregation of all recommendations provided by the third-party recommendation agents about the quality of the trusted agent» [13]. Abdul-Rahman & Hailes define reputation as «an expectation about an agent's behavior based on information about its past behavior» [16]. Kreps & Wilson link reputation to characteristic or attribute «ascribed to one person by another person (or community)» [17]. A complete (at least, voluminous) overview of definitions relating to trust and reputation can be found in [18]. In the present work, we use definitions for terms 'trust' and 'reputation' from [14] since both definitions agree with basic concepts of reputation system and interaction within P2P community.

Even though trust and reputation are very closely related concepts, and many sources simply use them virtually as synonyms, still there is a major difference to emphasize. While trust is subjective in nature, and it expresses local attitude of a peer regarding another agent on basis of his/her own past experience, reputation serves as a global and public perception of a given peer in the midst of other peers. With this point in mind, we may list those important characteristics of trust and reputation that must be taken into consideration when considering reputation systems.

- Context awareness (sensitivity) – trust or reputation of a peer is dependent on what the context of communication is. For instance, a peer can be really trustworthy in delivering books or office supplies, but unreliable in selling electronic accessories,
- Multi-faceted nature (diversity) – even in the same context, peer can evaluate the quality of communication with another peer on the strength of several aspects. In the case of online marketplaces delivery time, quality and price of goods (services) can be mentioned. While the context-sensitivity of trust underlines the fact that the trust in the same agent may vary with reference to different situations, the multi-faceted nature characteristic stands for manifoldness of trust. It definitely plays a substantive role in deciding whether an agent is trustworthy to interact with or not [14],
- Dynamism – apparently, levels of both trust and reputation increase or decrease in view of gaining experience (direct interaction). Such changes may alternate in due course depending on arising situation in the system, with a clear-cut declining tendency observed with time [14],
- Imprecision and uncertainty – it is not very habitual for humans to operate with estimates of trust and reputation in the form of numbers. Definitely, it is not difficult to perform relatively simple calculations even in passing, but explanations and interpretations are usually based on verbal forms (words, phrases and short sentences in natural language). The peer can be classified as *«very trustworthy»*, *«not too trustworthy»* or in some likewise manner. Thus, we express gradations (imprecise estimates) of the extent, to which the peer is reputed as trustworthy or not. The bounds of gradations (verbal granules) are inexact, but nevertheless linguistic forms are easily perceived and processed by specialists and ordinary people in talks, reasoning and decision-making process. We may conclude that trust is a highly subjective

category, and being apparently fuzzy it can be associated with verbal assessment values (granules). The vagueness of the trust is linked outright to uncertainty of reputation as well.

3. Uncertainty in marketplace data. Verbal assessments are very natural in reputation systems

The paper is focused on reputation systems as applied to e-commerce field (and specifically online marketplaces). Because of that it is essential to consider what kind of data concerning peers and their transactions are available, and what sort of data peer's feedback about fulfilled transaction contains. In online marketplaces there are two types of peers – they are sellers and buyers; every transaction implies participation of one seller and one buyer. It is important to distinguish these types of 'players', because they gain trust and reputation that differ by their gist. In the present work we consider three types of marketplace data:

- peer data, i.e. a set of general data pieces that relates to peer itself (personal data, registration date, etc.);
- transaction data – general data about transaction held between seller and buyer (delivery time, payment time, total sum and date of transaction, etc.);
- feedback data refer to data collected from both seller and buyer after completion of each transaction (goods quality, communication quality, shipping service reliability, etc.).

Most of the existing reputation systems work only with peer's feedback [7-10,19]. In general, feedback provides some subjective assessment of experience that a peer has with another peer in the course of transaction's realization. But in certain cases, such experience cannot be thoroughly expressed in terms of integers -1, 0 and +1 as it occurs in eBay system. Why do we think so? Firstly, regarding feedback as a number means neglecting diverse aspects of experience such as those mentioned above. Secondly, and this fact was also underlined earlier, it is more natural for humans to think of evaluating experience in terms of some ordinal scale stretching from «very bad» label to «very good» instead of using «good», «neutral» or «bad» plain marks as linguistic equivalents of -1, 0 and +1. In case of extended scale' use, its grades may overlap with each other, since each label or grade stands not for a single value, but for a range of values instead. For instance, there is no clear-cut border line between values (labels) «very bad» and «bad», but almost all people may differentiate these values mentally while impacting information chunks to others.

Similar situation comes to pass with reference to transaction data. For example, let us consider delivery time of basic electronic accessories from Moscow to Saint-Petersburg. We know that they are normally delivered within N days. Is it «quickly», «slowly» or «neither slowly, nor quickly» for a client? Maybe it is a little bit slowly, but not too much? What can be said about N+2, 2N days or even 5N days? At some point it becomes obvious that delivery time can be associated with label «slowly» or even «very slowly». But what do we mean by that some point? For different people it occurs at different moments, which are not fixed (crisp), and this is when imprecision and uncertainty (fuzziness) of these data reveal themselves.

4. Fuzzy logic theory. Some basic terms and definitions used in the study

Taking into account uncertainty inherent to notions of trust and marketplace data, we need to consider its formal representation for a possible use in trust management and reputation systems (models). The concept of uncertainty is many-sided and rich; furthermore, uncertainty 'accompanies' any interactions of humans with real world [11]. In this connection, reputation systems exemplify active communication of peers based on the exchange of information that is often a matter of human perceptions and interpretations to a various extent. Much depends here on cognition and verbal assessments expressed in the natural language. Such perceived units can be seen as granules with 'soft' bounds rather than exact quantities having unified meaning and interpretation by all parties involved into the process. The theory of fuzzy logic (FL) extends the ontology of

mathematical research in the context of formation of a composite methodology that leverages quality and quantity [20]. It provides ample means to model the "perceived meaning of words/phrases conveying the expert opinions (estimates) in a graded fashion" [21]. The following is a quick glance at main concepts and definitions concerned with fuzzy logic as used in the present study.

Definition 1. *Linguistic variables* (LV) – variables whose values are words, phrases or sentences (linguistic terms) expressed in natural or artificial language [22]. In short, we can state that LV constitute a form of information granulation serving as a base for further transition of those granules to computable counterparts [23]. For example, if we consider the case of delivery time from point A to point B, the label (term) "quickly" is one of possible linguistic values assigned to the variable *Delivery Time*. Its whole term-set can be represented as $\mathbf{T}_{(Delivery\ Time)}$ = «quickly» + «very quickly» + «slowly» + «more or less quickly» + … , where sign '+' denotes the aggregate of linguistic granules rather than arithmetic sum operation.

Linguistic variable *Delivery Time* is defined on the universal set U (realistic range of numeric values representing the delivery time in particular situation), i.e. each element $x \in U$ stands for the time (minutes, hours, etc.) that can be associated as a result of human's perception (judgment) with corresponding terms to various degrees.

Definition 2. Let U be a set of elements (objects) that are denoted generically as x (U={x}); *fuzzy set* $A \subseteq U$ is a set of ordered pairs $\{(x, \mu_A(x))\}$, where mapping $\mu_A : x \to [0,1]$ is a (type-1) membership function of a fuzzy set A. Value $\mu_A(x)$ is a degree (grade) of membership of x in the set A, and U is a problem's domain (universal set). Membership function (fuzzy set) represents possibility distribution of x-values over domain U, and it can be expressed as aggregation $\int_{x \in U} \frac{\mu_A(x)}{x}$ or union $\sum_{x \in U} \frac{\mu_A(x)}{x}$ of pairs $\{(x, \mu(x))\}$, $\mu(x) \in [0,1]$, in continuous and discrete cases, correspondingly.

Definition 3. Let A be a fuzzy set on U, then α-*cut* of A is a crisp (non-fuzzy) set A_α composed of all $x \in U$, whose grades of membership in A are greater or equal to α [22]. Formally, A_α can be expressed as $\{x \mid \mu_A(x) \geq \alpha\}$. A fuzzy set A may be decomposed into and restored from α-cut sets through the resolution identity [24, 25], i.e. $A = \int_0^1 \alpha A_\alpha$, or $A = \sum_\alpha \alpha A_\alpha$, $\alpha \in [0,1]$.

An integral part of any formal modeling approach is closely related to the use of functions. Along with pervasive processing of non-vague objects, fuzzy quantities in last three decades became widespread in algorithms covering enormous circle of application domains. The need to extend the possibility for functions to operate with arguments having the form of fuzzy sets has led to formulation of extension principle [22, 26, 27]. As its name speaks for itself, it is directed to spreading nonfuzzy mathematical concepts to fuzzy ones [28]. It is specifically what is required to handle aspects of uncertainty (fuzziness) with reference to existing reputation systems.

Definition 4. Assume f is a mapping from universal set U to set V, and A is a fuzzy set defined on U (for the sake of simplicity we may consider finite representation of such set, i.e. $A = \mu_A(x_1)/x_1 + \mu_A(x_2)/x_2 + ... + \mu_A(x_n)/x_n$). Relying on the *extension principle*, the image $f(A)$ of A under mapping f is obtained as follows:

$$f(A) = f\left(\mu_A(x_1)/x_1 + \mu_A(x_2)/x_2 + ... + \mu_A(x_n)/x_n\right) =$$
$$= \mu_A(x_1)/f(x_1) + \mu_A(x_2)/f(x_2) + ... + \mu_A(x_n)/f(x_n).$$

In other words, the image of A under f can be deduced from the knowledge of the images $f(x_1), f(x_2), ..., f(x_n)$.

Definition 5. The process of representing initial data (e.g. linguistic values) as membership functions is called *fuzzification*; most of applications require to perform at final stages the opposite translation from fuzzy functional forms to crisp values – this is achieved through *defuzzification procedures* [11, 21, 26, 29].

5. Brief comparison of existing reputation systems – their differences and intrinsic similarities

The number of publications devoted to trust management and reputation systems is pretty imposing, and it is growing from year to year [6, 8, 9, 10, 14, 19, 30]. In the paper, we wittingly touch upon (just brief overview augmented with performance considerations) the most significant systems that proved themselves as effective, robust and applicable to online marketplace reputation management. It is worth mentioning that only those systems that do not use basically fuzzy logic concepts are reviewed in the paper. For instance, systems that utilize fuzzy inference schemes or other fuzzy-logic related notions [20, 31, 32] constitute an interesting research topic, but on level with other relevant cases it is outside of the scope of the present paper.

Results of the conducted analysis of existing sources provided a basis for selection of those criteria that can be classified as crucial from the viewpoint of systems' comparison. They can be described concisely as follows:

- Speed of convergence – iterative algorithms form a core of nearly all reputation systems. Thus, one of important aspects of such algorithms is how fast they converge and produce a result. This feature is covered as a principal one in most of papers related to reputation systems [6,8,9,10],
- Robustness – it is the criterion concerned outright with the main purpose of every reputation system, namely, the prevention of malicious attacks. Therefore, it becomes essential to measure how well a given system is able to held out against malicious peers' activities. Such experiments are covered by S.D. Kamvar, M.T. Schlosser, H. Kurdi, N. Chiluka, N. Andrade, Y. Wang, L. Xiong, et al. in [6,8,9,10,14,19]; however, it is important to mention that papers referenced here cover different types of malicious behavior,
- Hyperparameters – their presence is an important point to consider in the process of system's deployment, since they show the extent, to which the system is customizable. But at the same time, factor of their presence is a 'double-edged sword', inasmuch as, on the one hand, tuning of hyperparameters may lead to better performance of a specific system. On the other hand, it enhances significantly the complexity of deploying the system,
- Handling data imprecision and uncertainty – most of hypothetical or artificial systems do their work in the presence of uncertainty. The latter is often linked to human factor being an integral part of a system in the context of verbally defined and/or interpreted data. The latter are elicited from active discussions with stakeholders and estimations commonly used as inputs in calculations provided for algorithms underlying system's work specifics. Those pieces of information are often 'soft' (imprecise) in their nature, and it opens manifest way for the use of fuzzy logic theory in models of reputation systems. Hence, the comparison of algorithms can be performed in the view of how well system (algorithm) adapts fuzzy logic extension.

5.1 EigenTrust Algorithm

EigenTrust system is originally proposed for a P2P file-sharing network by S.D. Kamvar, M. Schlosser and H. Garcia-Molina in the paper that became one of the most cited papers on reputation systems [8]. EigenTrust calculates a global trust value for each peer based on his/her past behavior by incorporating opinions of all peers in the system [19]. Opinions concerning a particular peer are represented as a local trust value. After each communication peer assesses his/her experience by

the value from restricted set comprised by integers -1, 0 or 1. The local trust value is an aggregation of all communication experience assessments. It was shown how to normalize local trust values in a way that leads to elegant probabilistic interpretation similar to the Random Surfer model and efficient algorithm to aggregate these values [8,33]. Pre-trusted peers that can be seen here as a hyperparameter (it must be chosen in advance for the whole system to operate) are used to guarantee convergence and breaking up malicious collectives. The choice of pre-trusted peers is important, and it can compromise the quality of system to a marked degree [8]. As also shown in [8], for a network with 1,000 peers the algorithm converges after completion of less than 10 iterations. Theoretical base for fast convergence of EigenTrust algorithm is discussed by T.H. Haveliwala and S.D. Kamvar in [34]. Robustness of the system is evaluated under several threat models, and the system shows good overall performance in all cases [8]. For both Individual malicious peers and Malicious collectives threat models, EigenTrust system outperforms non-trust-based systems showing five to eight times better results with fraction of inauthentic downloads (FID) less than 0.2 for every setting. For Malicious collectives with camouflage case (model 3), system shows less impressive results, but still Malicious Spies threat model (fourth model) tends to be the best strategy for malicious peers to attack trust-based network [8].

As already mentioned earlier, EigenTrust system uses aggregated local trust values that must be normalized beforehand to avoid system's 'demolition' due to assignment of very high and very low local trust values [8]. Normalized local trust value c_{ij} can be calculated as $c_{ij} = \max(s_{ij}, 0)/\sum_j \max(s_{ij}, 0)$, where s_{ij} is a local trust value. The shown way of normalization isn't free of drawbacks. For one thing, normalized values don't draw a distinction between a peer with whom a given peer *i* did not interact and a peer with respect to whom peer *i* has had a poor (negative) experience. Secondly, c_{ij} values are relative, and they cannot be interpreted easily in the absolute sense [8]. Thus, an attempt can be made to extend EigenTrust algorithm with fuzzy logic notions to obtain transparent and interpretable modification of the original computational scheme. Particularly, calculation of local trust value may be altered to accumulate different types of marketplace data, but further study that concerns the impact of fuzzification on probabilistic interpretation of EigenTrust algorithm is required.

5.2 HonestPeer Algorithm

HonestPeer as an enhanced version of EigenTrust algorithm is discussed by H. Kurdi [9]. The algorithm endeavors to address one of the major problems with EigenTrust system, viz. pre-trusted peers. HonestPeer minimizes the dependency on that pre-trusted set of peers by choosing one honest peer dynamically for every computation step of global trust value (GTV). This honest peer, i.e. the peer having the highest global trust value, plays a crucial role in further computations of GTV. The speed of HonestPeer's convergence is almost the same as for EigenTrust algorithm, despite the need to perform additional calculations. Following [9], two benchmarks are considered – they are EigenTrust algorithm and no algorithm. Performance of HonestPeer algorithm is estimated under different experimental settings embracing variable number of users and files as well as number of pre-trusted peers and with the examination of percentage of inauthentic file downloads by good peers and success rate of good peers (success rate of good peers equals the ratio of *#valid files received by good peers* to *#transactions attempted by good peers*).

HonestPeer algorithm surpasses EigenTrust in effectiveness and capability to 'help' good peers to download valid safe files. This fact can be attributed to the ability of HonestPeer to choose honest peers dynamically after each round, while in case of EigenTrust pre-trusted peers are chosen statically irrelative of their performance [9]. Since HonestPeer is basically an enhancement of EigenTrust algorithm, the use of fuzzy logic may be appropriate and explicable as a practical matter to address those forms of uncertainty that are typical for system under consideration.

5.3 PowerTrust Algorithm

The reputation system PowerTrust, which is based on power-law distribution of peer feedbacks discovered after examination of 10,000+ eBay users' transaction traces, is covered by R. Zhou and K. Hwang [10, 35]. In PowerTrust system a few power nodes are selected dynamically according to their reputation. These nodes can be dynamically replaced, if they become less active or demonstrate unacceptable behavior. Good reputation of power nodes is accumulated from the operation history of the system – functional modules of PowerTrust system as well as flow scheme that relates to collection of local trust scores and global reputation aggregation are visually demonstrated in [10]. Without going into particulars, it should be mentioned that raw data input for PowerTrust is treated as local trust scores, which are then aggregated to obtain global reputation score of each peer. The Regular Random Walk module supports the initial reputation aggregation, while Look-ahead Random Walk (LRW) module is used to update the reputation score periodically. LRW also works with Distributed Ranking Module to identify power nodes. The system leverages power nodes to update Global Reputation Scores (vector V) [10].

The experimental performance of PowerTrust in terms of reputation convergence overhead to measure aggregation speed, ranking discrepancy to measure the accuracy, and root-mean-square (RMS) aggregation error to quantify system's robustness to malicious peers shows that PowerTrust algorithm outperforms EigenTrust by more than factor 1.5 in case of convergence speed [10]. Under all settings PowerTrust exhibits its robustness against collusive peer groups of various sizes.

In much the same way as for both EigenTrust and HonestPeer, the point of fuzzy logic's application to PowerTrust algorithm is a local trust value. Several linguistic terms may be defined on [0,1] interval to be used consequently for computation of global reputation. It is of definite interest to research whether the use of fuzzy logic may affect properties of PowerTrust algorithm or not.

5.4 Absolute Trust Algorithm

The algorithm for aggregation of trust among peers in P2P networks (Absolute Trust algorithm) was presented by S.K. Awasthi and Y.N. Singh [30]. Most of reputation systems are built upon scenario when all peers evaluate other peers by way of assigning foregoing local trust values that are a subject for further aggregation aimed at obtaining peers global reputation scores. In general, three different types of evaluation scenarios (*one-to-many*, i.e. one person is evaluating many persons, *many-to-one* scheme, under which many persons are evaluating one person, and *one-to-one* case, which implies that one person is evaluating another person) can be identified. In an effort to strengthen feedback's reliability in many-to-one evaluation scheme, any evaluation provided must allow for the competence of evaluator (evaluating party in the system) in computations via proportional weight's factor. Global trust of *j*-th peer can be used by way of weight in aggregation of local trust scores in calculation of any given *i*-th peer's global trust. Thus, a set of peers communicating (providing services) to the *i*-th peer can be reduced to just one virtual representative. It results in obtaining one-to-one evaluation scheme, and the trust of a set will be dominated by peers having higher global trust [30].

The existence and uniqueness of global trust vector as an outcome of aggregation approach is proven in [30]. The closed-form peer's global trust expression lays a basis for direct comparison of global trust values calculated for any two peers in the system (network with N nodes). There is no theoretical explanation of fast algorithm's convergence, but experiments show that it converges fast (about 7 iterations for 100 peers in the network) [30].

Robustness of the algorithm is evaluated regarding behavior of EigenTrust and PowerTrust systems. Several network configurations are considered in [30] such as the ones under the presence of pure malicious peers, peers with unpredictable behavior as well as malicious collectives (groups of peers whose familiarity positively affects their own reputation values diminishing corresponding values of persons outside such groups). It was shown that for first two configurations the performance of Absolute Trust improves significantly as compared to

counterparts (by appr. 2% to 4% of authentic transactions that relate to exchanging files between peers, respectively). As concerns malicious collectives, performances of algorithms are almost identical, with marginal superiority of Absolute Trust over its aforesaid rivals.

The local trust metric in this algorithm can be defined in many ways, and it forms prospects to develop a fuzzy local trust metric. It is worth mentioning that aggregation procedure used in the algorithm can be practically retained. The customizable local trust metric allows to use fuzzy logic approach to extend the algorithm in relatively easy and natural way.

5.5 PeerTrust Algorithm

PeerTrust is another example of P2P reputation system designed specifically for e-commerce communities that are characterized by distinctive problems and risks [6]. L. Xiong and L. Liu identify five important factors that relate to evaluation of peer's trustworthiness as regards supplying other peers with corresponding services. These factors are feedback obtained by a peer, feedback scope (e.g. total number of transactions occurred between peers), credibility of feedback source, transaction context aimed at drawing distinction between extremely crucial and less important or uncritical transactions, and community context to address community-wide characteristics and vulnerabilities. Based on formalization of these parameters, the authors proposed a peer's j trust value (metric) $T(j)$ consisting of two parts [6]. The first one is a weighted multiplicative combination of amount of satisfaction peer obtained after realization of each transaction, adaptive transaction context for i-th transaction of a peer and credibility of the feedback received from peers. Community context factor constituting the second part of $T(j)$'s expression increases or decreases the impart of the first part to trust value owing to allowance of distinctive community's features. The proposed metric $T(\cdot)$ should be considered as a general form, in which corresponding parts can be 'tuned' in terms of parameters and factors used [6]. Every part of the metric can be implemented differently – alternatives of possible credibility measure metrics (trust value/TVM, personalized similarity/PSM) are presented by L. Xiong and L. Liu in their paper.

Speed of convergence and complexity of PeerTrust algorithm appreciably depend on metrics definitions and specific implementation strategies. In general, the performance of system under PSM metric is a bit worse than in case of TVM, but on the other hand, the former provides better results as the number of peers in the network is increasing. System's robustness is assessed on the grounds of effectiveness against malicious behavior of peers comparing to conventional algorithm, in which the average of the ratings is used to measure the trustworthiness of a peer without taking into account the credibility factor. The trust computation error as a root-mean-square error (RMSE) of the computed trust value of all peers and the actual likelihood of peers performing a satisfactory transaction are computed to evaluate the algorithm's performance. PeerTrust with PSM metric ensures striking results as calculated RMSE does not exceed the value of 0.05, and transaction success rate attains virtually unity.

It must be admitted that PeerTrust system is very flexible over the existing possibility to choose local trust metric. Therefore, it seems that the practical application of fuzzy logic approach to handle naturally nascent uncertainty (vagueness) of certain parameters and characteristics in the algorithm looks justified enough. The system also possesses a great potential to incorporate all types of marketplace data, especially through transaction and community context parts of the general metric $T(\cdot)$ that afford means of broad coverage of manifold system's peculiarities.

6. Experimental part – setup stage. General comments

For the experimental part of the study, we implemented a simulator (in Python 3.7), and the section describes the general simulation setup, including the community setting, peer behavior pattern, and trust computation.

We assume that hypothetical (simulated) community consists of N peers, for which two peer types are defined, namely, they are *honest* and *strategic*, or malicious, peers [36]. The first one embraces those commitment long-run players focused on cooperation, since the latter maximizes player's lifetime payoffs, if the player consistently sticks to action in long-range outlook. In contrast, opportunistic player who cheats whenever the occasion is beneficial for him is bound to a strategic type [6]. The percentage of malicious peers in the community we denote by K. It is reasonable that behavior pattern of good peers is to always cooperate and provide honest feedback after each transaction. However, a correct modeling of malicious peers behavior is a bit challenging task that may require certain simplifications. In particular, we may consider that malicious peers always cheat during transactions and give dishonest ratings to other peers, i.e. they rate negatively a peer who cooperates and provides good rating to a peer who cheats. In case of EigenTrust and HonestPeer algorithms there are also pre-trusted peers that play an important role from the standpoint of algorithm's consistency. Respective PRE_TRUSTED parameter stands for the percentage of pre-trusted peers that relate to good peers only. In general, *behavior pattern of peers* is a topic on a slippery ground, i.e. it can be placed among those aspects of models of reputation systems that require close scrutiny. Why? Potentially, the above cited pattern is definitely not unique, so in order to make models viable other feasible options must be addressed hereafter with great care.

We may also assume that community has CAT categories of services that are provided by peers. From amongst these categories each peer is interested only in a specific subset having the cardinality not less than S. Each category is associated with at least P percents of peers in the community. When a peer queries a service of a specific category, only peers associated with this category can respond to such query. At that, two transaction settings are simulated – they are random and trusted. Random (or, *simple*) setting means that a peer, which responds to the query, is selected randomly (uniform distribution is used) from a set of all peers that can provide queried category of service. In trusted setting the responder is also selected randomly from all peers that can respond to the query, but it is done with respect to their reputation, i.e. a peer with higher reputation has better chances to be chosen. If there are peers with zero reputation, then there is a 10% chance that the responding peer will be chosen uniformly from those peers. It efforts the opportunity for new peers to start building up their reputation.

Binary feedback system is used to evaluate peer after each completed transaction. It means that values 0 and 1 are practiced for PeerTrust and Absolute Trust algorithms, -1 and 1 are used in cases of both EigenTrust and HonestPeer approaches. Local trust and reputation computation steps as such depend on the algorithm in use. Some algorithms have their own hyperparameters that must be specified. Default values of parameters are listed in Table 1. Simulation session (cycle) consists of SIM_NUM transactions. Global reputation is updated after every UPDATE_NUM transactions. Experimental results are averaged by 5 cycles of simulation. Although we simulate online marketplace community – usually it is big enough, dozens to hundreds of thousands of peers – experiments are performed under the presence of modest number of peers. It may be considered as a perceptible limitation, however, the main aim of simulation is to obtain those *prior* results that lay down the ground for further analysis of weak/strong points of models considered here in terms of deeper understanding of their potential to incorporate formal representation of uncertainty (imprecision) factors into these models. In real-life environment it seems highly unlikely that the major part of marketplace peers is malicious as it was defined earlier. Therefore, we don't consider in simulation a malicious peers share exceeding 35%.

7. Experimental part – results and their comparison

We introduce a metric that shows the effectiveness of the reputation system as a rate of unsuccessful transactions (RUT). The unsuccess of transactions is bound up with the outcome of those transactions, in which responding peer happens to be malicious. It is obvious that the less value of the metric is the better. Besides, for the time being we *do not* consider PowerTrust algorithm in the empirical study, since it requires more close inspection and implementation cycle.

7.1 Effectiveness against malicious behavior

The objective of conducted experiments is to evaluate the robustness of the reputation systems against peers with malicious behavior. In the first experiment we alter the percentage of malicious peers in hypothetical community from 10% to 35% with other parameters keeping their default values (Table 1).

Table 1. Parameters and their values used in experiments

Affiliation with…	Parameter	Description	Default value
Community setting	N	number of peers	1000
	K	percentage of malicious peers	15
	CAT	number of categories	10
	S	minimal number of categories for each peer	3
	P	minimal percentage of peers associated with each category	5
Simulation setting	SIM_NUM	number of queries in a simulation	10000
	UPDATE_NUM	number of transactions in reputation update cycle	100
EigenTrust & HonestPeer	PRE_TRUSTED	percentage of pre-trusted peers	5
Absolute Trust	GOOD_W	weight of good transactions in local trust	10
	BAD_W	weight of bad transactions in local trust	1

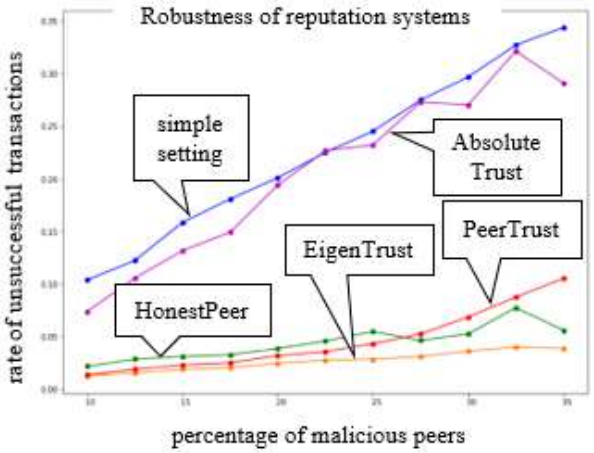

Fig. 1. The growth of rate of unsuccessful transactions depending on the increase of malicious peers' percentage (from 10% to 35%) for different algorithms

As is easy to see in fig. 1, the rate of unsuccessful transactions grows almost linearly with the increase of values (axis x) for simple setting; trusted settings show better results though. EigenTrust, HonestPeer and PeerTrust algorithms show extremely moderate growth of RUT with the increase of malicious peers' percentage. Absolute Trust algorithm demonstrates quite disappointing results characterized by negligible gain (within appr. 2.1% on average) as compared to simple (random) system's case.

7.2 Speed of convergence and scalability

In this set of experiments, we take aim at evaluating the general speed of algorithms convergence and their scalability with regard to the increase of number of peers (fig. 2). As will readily be observed, algorithms PeerTrust and Absolute Trust generally need not more than 2 iterations to converge, while EigenTrust and HonestPeer need to go through 4+ iterations. More than twofold difference on very small values practically equalizes rivals under the conditions of experiment. Thus, all algorithms seem to be quite scalable concerning the number of iterations needed to converge, since the latter does not grow substantially with the increase (from 1,000 to 3,500) of number of peers in the community.

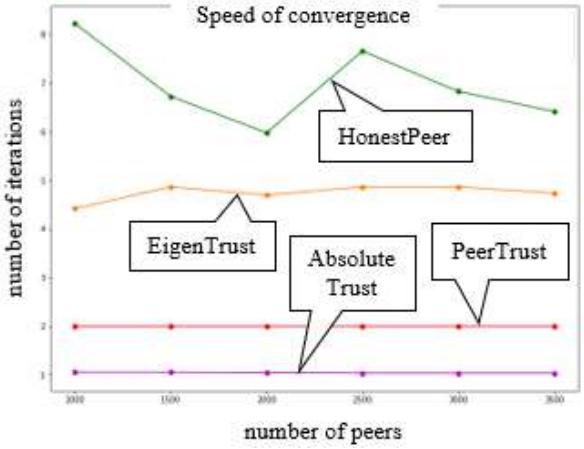

Fig. 2. The speed of convergence (number of iterations needed) of algorithms depending on the number of peers (in the range from 1,000 to 3,500)

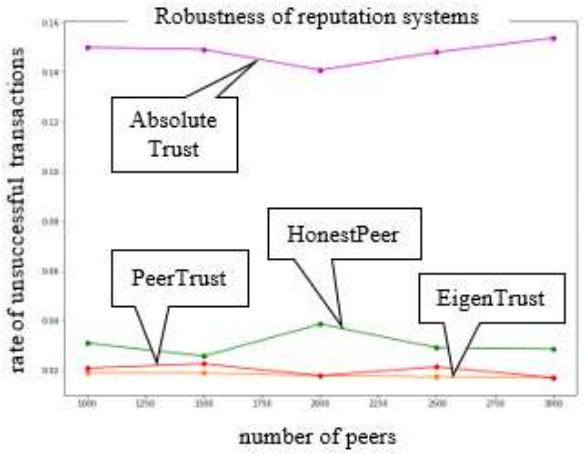

Fig. 3. The speed of convergence (number of iterations) of algorithms depending on the number of peers (in the range from 1,000 to 3,500)

We also evaluate how consistent corresponding systems are against the background of increasing number of peers under «freezing» of other parameters (fig. 3). It can be seen that situation remains almost indistinguishable be it small or bigger community – the rate of unsuccessful transactions mostly remains unchanged in the context of the same malicious peers' percentage.

7.3 Choice of the «best» (most feasible) system

According to the results of experiments summarized above as well as constraints and assumptions put forward, PeerTrust model appears the most robust and effective reputation system among alternatives. It is quite stable regarding the growth of percentage of malicious peers in the community and scalable enough to handle evenly larger number of peers. What is more, local trust metric in PeerTrust system is highly customizable, and this fact simplifies the possibility to extend it with fuzzy factors inhere in reputation systems. In a wide sense we can talk about marketplace data uncertainty that requires much attention in further development of the topic and elaboration of formal aspects of models. Thus, in this instance we opt for PeerTrust system with the object of its modification on the basis of Zadeh's extension principle.

8. Transition from crisp PeerTrust system to Fuzzy PeerTrust system. Is it worthy of notice?

In order to implement fuzzy reputation system, we need to understand above all what data will be represented by fuzzy sets (numbers). In non-fuzzy version of PeerTrust algorithm binary feedback system is used. We suggest utilizing a broader scale to express degrees of satisfaction concerning transaction. It naturally arises from peculiarities of human's perception of information (comments, judgments) – it is not a very convenient and alluring way for humans to think in terms of zeros and ones (or, any other numbers). For the human mind such terms as «bad», «normal» and other resembling options look more understandable and well-suited for interpretation and processing. Being guided by this observation, the new algorithm's feedback can be represented by five verbal degrees of satisfaction, namely, they are *«very bad»*, *«bad»*, *«normal»*, *«good»* and *«very good»*. More fine granulation does not look preferable here, because it may lead to certain confusion in view of human perception of satisfaction's shades – the 'magic' number 7±2 and the seminal paper (1956) by American psychologist George A. Miller on limits on our capacity for processing information straight away cross our mind.

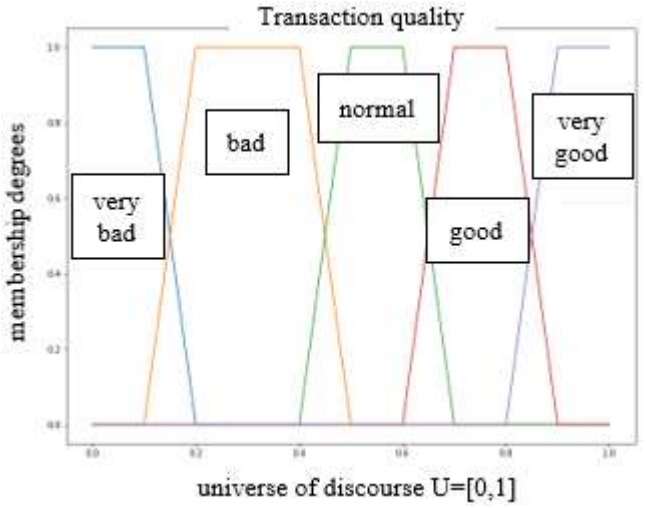

Fig. 4. Linguistic values (trapezoidal membership functions) of the variable 'Transaction quality' (universe of discourse U=[0,1])

Such verbal terms are treated as linguistic values of the variable *«degree of satisfaction»* or *«transaction quality»*; each value can be formally represented by trapezoidal membership function on universe of discourse U=[0, 1] as shown in fig. 4. The type (e.g. Gaussian, bell-shaped, etc.) and the location of fuzzy sets on U may vary noticeably depending on estimates provided by

expert group with reference to characteristic features and implicit shades of model under consideration [22]. The rest of the algorithm remains unchanged, and all specified operations are carried out with fuzzy numbers (intervals) instead of crisp values till the attainment of the defuzzification stage. Defuzzified reputation values are used to choose the responding peer exactly in the same way as described above. In the paper centroid method (COA) is used to obtain those values, but effectiveness and performance of the algorithm may depend distinctly on the chosen defuzzification approach [21, 26].

Here, special attention should be paid to the following: in the paper we consciously consider only one type of data falling under fuzzification, viz. the feedback regarding a buyer. Primarily it is connected with the amount of required modifications and scope of computational experiments to be covered by the text of the limited size. But we are aware that other foregoing types must be addressed thoroughly in the course of the ongoing empirical study.

In conditions of maintenance of community and simulation settings (see the details of conducted experiments described above), but under the imprecision (vagueness) taken into account in the feedback system, we compare the experimental components of Fuzzy PeerTrust with original PeerTrust and EigenTrust algorithms.

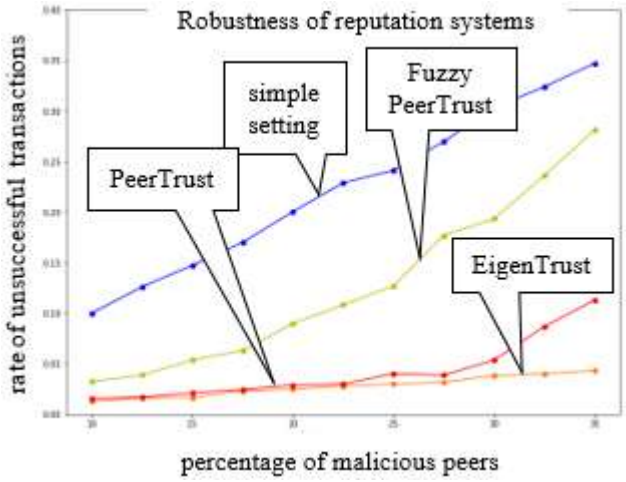

Fig. 5. The growth of rate of unsuccessful transactions depending on the increase of malicious peers' percentage (from 10% to 35%) for EigenTrust, PeerTrust and Fuzzy PeerTrust algorithms

8.1 Effectiveness against malicious behavior (Type-1 fuzzy case)

In the first place, we want to evaluate the robustness of fuzzy modification of PeerTrust system. Experiment settings are retained, the percentage of malicious peers is changing within the range from 10% to 35%. The results as shown in fig. 5 lead to the conclusion that Fuzzy PeerTrust algorithm is definitely more robust in comparison with Simple system. Under small percentage values (appr. interval [10%,18%]) of malicious peers, the performance's characteristic of Fuzzy PeerTrust is close enough to original PeerTrust and EigenTrust. However, it demonstrates worse results than crisp algorithms over the whole range of x-axis values concerned.

8.2 Speed of convergence and scalability

Another set of experiments was aimed at estimation of the speed of convergence of Fuzzy PeerTrust and its scalability in view of the community's growth. As expected, the speed of convergence remains the same as for original PeerTrust with two iterations on average to converge, and it differs essentially from corresponding characteristic (appr. 4.61 on average) of EigenTrust algorithm (fig. 6). In terms of robustness Fuzzy PeerTrust can also be pronounced

scalable, since it does not show significant decrease in quality with the growth of the number of peers in the community (fig. 7). We observe smooth fluctuations of RUT at the level of 0.064. It is worth mentioning that all properties of crisp algorithm remain intact in comparison with its fuzzy counterpart.

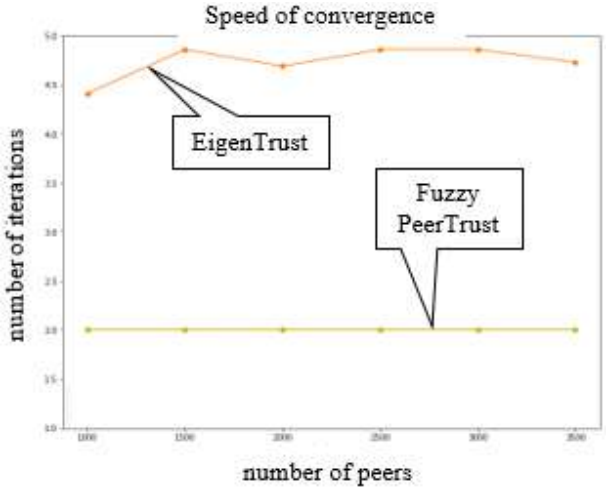

Fig. 6. The speed of convergence (number of iterations needed) of EigenTrust and Fuzzy PeerTrust algorithms depending on the number of peers (in the range from 1,000 to 3,500)

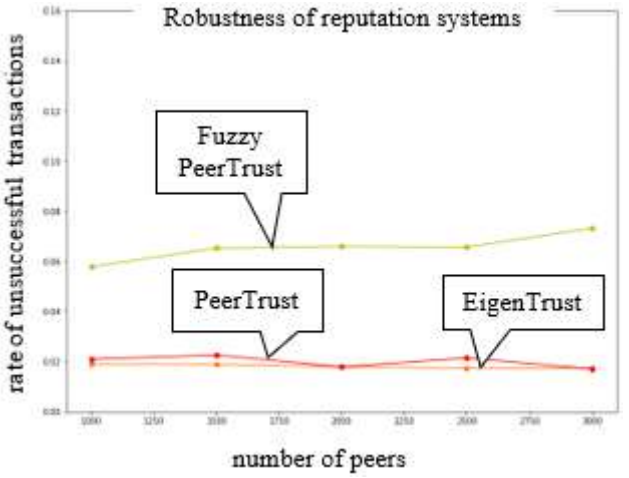

Fig. 7. The growth of rate of unsuccessful transactions depending on the number of peers (in the range from 1,000 to 3,500) for EigenTrust, PeerTrust and Fuzzy PeerTrust algorithms

Computations are initiated with trust vector $t^0 = \left(t_1^0, t_2^0, ..., t_{\mathrm{N}}^0\right)^{\mathrm{T}} = \left(\frac{1}{\mathrm{N}}, \frac{1}{\mathrm{N}}, ..., \frac{1}{\mathrm{N}}\right)^{\mathrm{T}}$ (default case), where N is the number of peers in the community, t_v^0 is a default trust value of a peer *v*, $v = \overline{1, \mathrm{N}}$ [6]. Reputation of a peer *v* in the form of fuzzy set (number) is denoted as $fuzzy\left(t_v^{i+1}\right)$; $fuzzy\left(S(v, \mathrm{j})\right)$ stands for a feedback (fuzzy number) of peer *v* concerning j-th transaction; $defuzz(\bullet)$ signifies the reduction of a fuzzy argument to crisp value (deffuzzification step). To

calculate the product of fuzzy number $fuzzy\left(S(v,\mathrm{j})\right)$ and crisp number $t_{\mathrm{j}}^{\mathrm{i}}\Big/\sum_{\mathrm{k}=1}^{\mathrm{I}(v)}t_{\mathrm{k}}^{\mathrm{i}}$ as well as the sum (1) of thus obtained fuzzy numbers, Zadeh's extension principle is used [28,29,37]. As a result, steps to be performed (Algorithm 1 / **case F1**) can be expressed as follows:

Result : $\mathrm{t}-$ vector of global trust values

$$t^0=\left(\frac{1}{\mathrm{N}},...,\frac{1}{\mathrm{N}}\right)^{\mathrm{T}},\ \mathrm{i}=0$$

repeat

for $\nu\leftarrow 1$ **to** N **do**

$$fuzzy\left(t_v^{\mathrm{i}+1}\right)=\sum_{\mathrm{j}=1}^{\mathrm{I}(v)}fuzzy\left(S(v,\mathrm{j})\right)\cdot\frac{t_{\mathrm{j}}^{\mathrm{i}}}{\sum_{\mathrm{k}=1}^{\mathrm{I}(v)}t_{\mathrm{k}}^{\mathrm{i}}} \qquad (1)$$

$$t_v^{\mathrm{i}+1}=defuzz\left(fuzzy\left(t_v^{\mathrm{i}+1}\right)\right) \qquad (2)$$

end

$$\sigma=\left\|t^{\mathrm{i}+1}-t^{\mathrm{i}}\right\|$$

$$\mathrm{i}=\mathrm{i}+1$$

until $\sigma<\varepsilon$;

As already mentioned, it is important to put emphasis on the choice of defuzzification method to use in (2). In general, the step of defuzzification relates to the conversion of a fuzzy quantity expressed in the form of membership function to a crisp number. In this case, we can talk about a diverse group of "fuzzy-to-crisp" data transformation methods, including, in particular, Center of Gravity (COG or centroid), Bisector of Area (BOA), Mean of Maximum (MOM), Smallest of Maximum (SOM) and Largest of Maximum (LOM) standard computational schemes as some of the most commonly used approaches. A rigorous and detailed discussion of defuzzification strategies can be found in [38, 39].

The results of the conducted experiments with Fuzzy PeerTrust under default values of parameters (Table 1) for different defuzzification methods shows that SOM scheme performs significantly better in the presence of smaller standard deviation as compared to other strategies. Intuitively SOM provides better results for the case in hand, because reputation system is punishing malicious peers more 'harshly', and it leads to better isolation of such peers from good peers. At the same time, changing defuzzification method in experimental settings does not affect scalability of the algorithm itself, since as the number of peers increases, the rate of unsuccessful transactions remains unchanged at insignificant fluctuations observed. Overall, we consciously avoid generalizations here, since the competitive advantage of SOM in the given algorithm should be confirmed empirically in the future.

At the same time, an important point of the algorithm shown above is that certain aforesaid attributes of trust and reputation like context-awareness (sensitivity), decrease (of the level) with time, their multifaceted nature (diversity) are not taken into account. We may regard this version of the algorithm as *basic* one (or, *F-basic* if we consider factor of fuzziness in its core); it paves a 'wide' way for algorithm's further revision, amendment and improvement.

9. Switching from using Type-1 fuzzy sets to Interval Type-2 fuzzy sets in reputation systems – the way to deal with uncertainty of expert's assessments

It can be noticed that the shift towards application of type-1 fuzzy sets in algorithms leaves us anyway within the scope of *crisp* real values of membership functions, which are associated with elements from a problem's domain (or, universal set) U. Despite active use of type-1 fuzzy sets in research works and industrial projects for almost forty years, existing publications specifically note that such sets exhibit very limited capabilities for modeling uncertainty, because of $\mu_A(x)$ crispness ($\forall x \in U$) mentioned above [40, 41]. In case of type-2 fuzzy sets, their membership functions are getting fuzzy, i.e. each specific $\mu_A(x)$ becomes associated with more than one value unlike their type-1 counterparts.

The latter allows for representation of vagueness inherent in natural language constructs (words, phrases) that express the assessments made by experts. Following explanations done by German philosopher F.G. Frege, the notion of vagueness relates to so-called «boundary line»; it can be expanded to the case of absence of clear truth conditions that is observed in most practical cases involving human judgments [42].

We represent linguistic values of the variable *«degree of satisfaction»* or *«transaction quality»* by interval type-2 trapezoidal membership functions defined on the universe of discourse U=[0, 1] as shown in Fig. 8. Types of membership functions as well as their location on the universal set U may vary noticeably depending on estimates provided by members of expert group [22]. It is important to note that type-2 fuzzy sets may appear due to natural slight differences in expert assessments and aggregation methods applied to them. As we can see, each of five functions shown in fig. 8 is bounded by two type-1 functions called upper (UMF) and lower (LMF) membership functions. Each function's 'thickened' boundary (footprint of uncertainty, FOU) is formed by *primary* interval-shaped memberships $\mu_A(x) \subset [0,1]$ ($\forall x \in U$) that can be seen as a collection of vertical slices of original type-2 function.

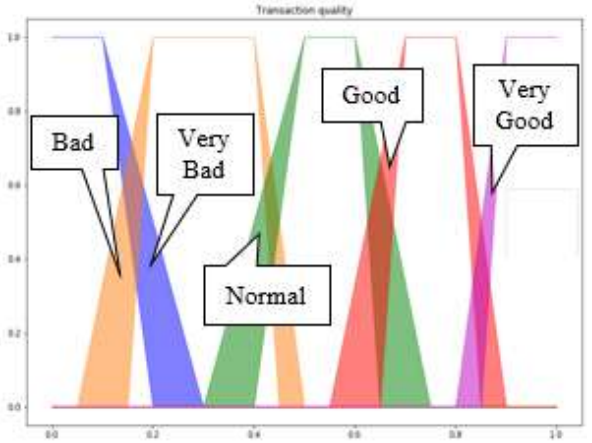

Fig. 8. Linguistic values of the variable 'transaction quality' (universe of discourse U=[0,1]) represented in the form of interval type-2 trapezoidal membership functions

Corresponding *secondary* function is connected to each interval $\mu_A(x)$, i.e. secondary membership functions are defined on the whole set of $\mu_A(x)$ for each type-2 function under consideration. The usual two-dimensional portrayal of type-2 MFs implies their 3D-view, which is determined by the presence of secondary grades. In the present study, the focus is restricted to interval type-2 fuzzy sets (IT2FS), for which all secondary grades are equal to one. Being a convenient uncertainty

modeling tool to capture representation of heterogeneous verbal responses formed within the group of domain experts, such functions are actively used when solving various practical problems due to their well-developed theoretical foundations and sound computational efficiency. If preceded by shown trapezoidal membership functions, the thicknesses of areas bounded by their pieces (LMF and UMF) convey degrees of uncertainty as a result of aggregation of converted to numeric form expert opinions concerning perception of values of the linguistic variable *«transaction quality»*.

Switch to using these functions in models leads to modification of the aforesaid *F-basic* algorithm, in which all operations are performed on interval type-2 fuzzy sets until defuzzification stage is reached. Steps to be performed can be expressed now (Algorithm 2 / **case F2**) as follows:

Result : $\mathrm{t}-$ vector of global trust values

$$t^0 = \left(\frac{1}{\mathrm{N}},...,\frac{1}{\mathrm{N}}\right)^{\mathrm{T}},\ \mathrm{i}=0$$

repeat

for $\nu \leftarrow 1$ **to** N **do**

$$fuzzy\left(t_v^{\mathrm{i}+1}\right) = \sum\nolimits_{\mathrm{j}=1}^{\mathrm{I}(v)} fuzzy\left(S(v,\mathrm{j})\right) \cdot \frac{t_{\mathrm{j}}^{\mathrm{i}}}{\sum_{\mathrm{k}=1}^{\mathrm{I}(v)} t_{\mathrm{k}}^{\mathrm{i}}} \qquad (1)$$

$$fuzzy\left(t_v^{\mathrm{i}+1}\right) = reduce\left(fuzzy\left(t_v^{\mathrm{i}+1}\right)\right) \qquad (1')$$

$$t_v^{\mathrm{i}+1} = defuzz\left(fuzzy\left(t_v^{\mathrm{i}+1}\right)\right) \qquad (2)$$

end

$\sigma = \left\| t^{\mathrm{i}+1} - t^{\mathrm{i}} \right\|$

$\mathrm{i} = \mathrm{i}+1$

until $\sigma < \varepsilon$;

It is proposed to implement modifications by means of «type-2 to type-1» type reduction (1′) to obtain the averaged trapezoidal membership function (resultant type-reduced set); afterwards, the latter is defuzzified. It is noteworthy that type reduction algorithms are the topical area of research, so extra experiments related to realization of defuzzification are an essential component of further extension of the work.

Results of comparing fuzzy Type-2 PeerTrust with other algorithms are shown in fig. 9 and 10. Just as expected, they're comparable to the performance of Fuzzy PeerTrust algorithm. Better results as compared to Simple case are quite predictable; there is a close enough resemblance to the original PeerTrust and EigenTrust systems, especially for percentage of malicious peers in the range from 10% to 20%. In average, fuzzy Type-2 PeerTrust shows slightly worse rates than crisp systems, although it is possible to find an intuitive explanation for that.

Consecutive application of type reduction and defuzzification procedures may lead to certain "displacement" of calculated values in comparison to original crisp models. It should not be considered as a shortcoming of the system; it is a fact that must be taken into consideration when using IT2FS. Potentially, it makes sense to use several type reduction and defuzzification procedures in every case in question. Calculations will obviously become more time-consuming

but will allow to take account of existing uncertainty factor in a more complete manner, leading to obtaining interval results rather than exact points. As shown in Fig. 10, the adoption of IT2FS in models does not affect the scalability of system in the context of experimental conditions.

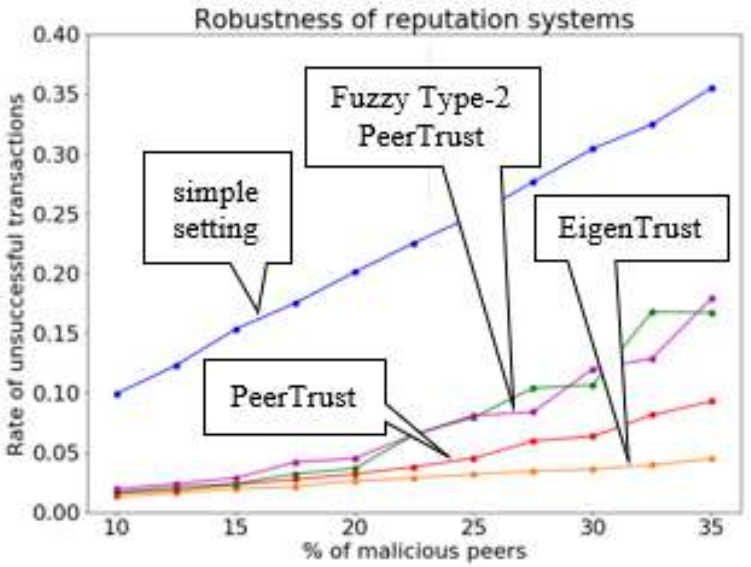

Fig. 9. The rate of unsuccessful transactions depending on the increase of malicious peers percentage (from 10% to 35%) for EigenTrust, PeerTrust and Fuzzy Type-2 PeerTrust algorithms

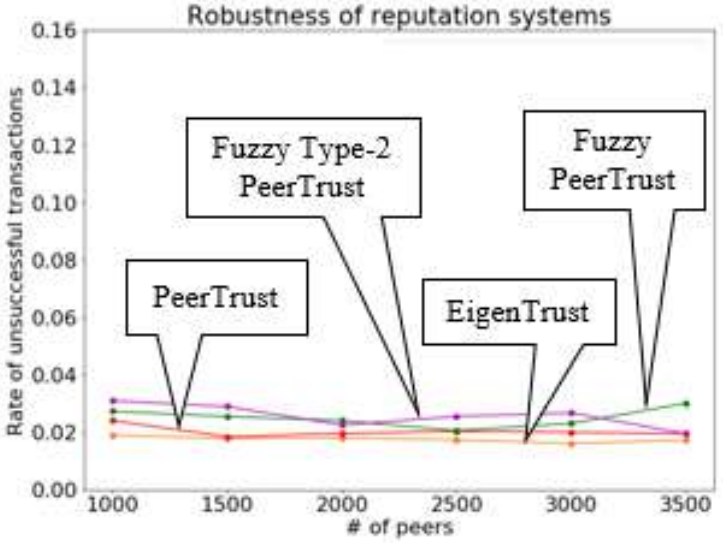

Fig. 10. The rate of unsuccessful transactions depending on the number of peers (in the range from 1,000 to 3,500) for EigenTrust, PeerTrust, Fuzzy PeerTrust and Fuzzy Type-2 PeerTrust algorithms

The results attained enable to speak decidedly about existing perspectives of fuzzy logic approach's application in reputation systems (corresponding algorithms), whether type-1 or interval type-2 fuzzy set is considered. Even despite somewhat higher computational costs compared to original crisp algorithms, greater transparency, better perceptibility by humans, interpretability and flexibility from a viewpoint of verbal expression and formalization of the

scores provide a basis for further studying of the topic. The present paper can be considered as a mere first step in this direction.

10. Conclusion

E-commerce is a fast-growing market that implies continual and utterly active communication between users being 'strangers' to each other on numerous occasions. Because of that it is essential to establish *reputation systems* to better handle available online information with the object to more accurately discern trustworthy and non-trustworthy players in systems creating grounds for peers to be more careful about their reputation. By the far-famed example of eBay reputation system, even relatively simple ones show themselves as very helpful from the viewpoint of malicious behavior's limitation and trustability increase. Online marketplaces that became immensely popular in the last 10-15 years as sites offering wide enough range of reasonably priced goods from various sources can be also considered as P2P networks. There is a vast range of reputation systems developed for P2P networks (mostly aimed at file-sharing) that can be adapted to e-commerce.

The main problem that is covered in the paper relates to the fact that none of these systems work with uncertainty (blurriness) of marketplace data and vagueness typical for notions of trust and reputation. Uncertainty in different forms of its manifestation is definitely inherent in reputation systems, and some of those forms can be addressed by fuzzy logic. This very inhesion, but not disconfirmed artificial desire, has served as an impellent factor to start this study.

Most likely, it can be argued that it is difficult to identify on the basis of several singled out key criteria unconditional leader among analyzed systems (algorithms EigenTrust, Absolute Trust, HonestPeer, PowerTrust and PeerTrust), since each of them has positive aspects as well as drawbacks. All algorithms, except PowerTrust, were implemented (Python 3.7) under the same conditions discussed in detail in the paper with the purpose of comparing fairly their relative performance. For reasons partially covered in the paper, Absolute Trust and PeerTrust systems were prudently regarded from the standpoint of their robustness and scalability as front-runners, i.e. candidates for reasoned fuzzification. Besides undertaking comparative analysis of those five significant and most popular reputation systems, the paper makes a mark for transition from crisp system (by the example of PeerTrust algorithm) to its fuzzy counterparts. The latter provided for an approach based on the use of type-1 (T1FS) and interval type-2 fuzzy sets (IT2FS).

Corresponding fuzzy models (we call them provisionally *F-basic* and modified *F-basic* algorithms – **cases F1** and **F2**, correspondingly, as they are denoted above) consider now only one characteristic of trust and reputation, namely, it is transaction quality or degree of peer's satisfaction. Other important attributes like context-awareness (sensitivity) or decrease of trust's level with time were not scrutinized yet. Nevertheless, initial experimental results attained in line with the fact of constant presence and active use of verbal assessments in reputation systems confirm the need to continue research in this field. Verbal forms are both habitual and convenient for human's perception despite of intrinsic vagueness and uncertainty. That is why, fuzzy logic approach, to the opinion of authors, has good prospects for both close examination and use in reputation systems.

At the same time, it should be mentioned that there are immediate tasks related to fuzzy models that require primary attention. The choice of shapes of membership functions and their fine tuning (location on the universe of discourse) on the basis of either existing data or expert assessments, a more detailed study of the potential of models using IT2FS as well as the use of different type reduction and defuzzification strategies are amongst the most topical ones.

References

[1]. The Next Scoop, 2018. E-Commerce is Growing at an Unprecedented Rate All over the Globe - The Next Scoop, web-resource: https://thenextscoop.com/e-commerce-is-growing-at-an-unprecedented-rate-all-over-the-globe/ (access date 17.02.19).

[2]. Statista, 2018. E-commerce Share of Total Retail Sales in United States from 2013 to 2021, web-resource: https://www.statista.com/statistics/379112/e-commerce-share-of-retail-sales-in-us/ (access date 26.02.19).

[3]. The Next Scoop, 2018. 2019 E-commerce Trends, Statistics and Metrics, web-resource: https://thenextscoop.com/ecommerce-trends-statistics-and-metrics-2019/ (access date 14.02.19).

[4]. Statista, 2018. Global Retail E-commerce Market Size 2014-2021, web-resource: https://www.statista.com/statistics/379046/worldwide-retail-e-commerce-sales/ (access date 08.02.19).

[5]. Forbes.com, 2017. What Are Online Marketplaces and What Is Their Future? web-resource: https://www.forbes.com/sites/richardkestenbaum/2017/04/26/what-are-online-marketplaces-and-what-is-their-future/#704431c13284 (access date 06.02.19).

[6]. Xiong L., Liu L. PeerTrust: Supporting Reputation-Based Trust for Peer-to-Peer Electronic Communities. IEEE Transactions on Knowledge and Data Engineering, vol. 16, no. 7, 2004, pp. 843-857.

[7]. eBay, 2019. web-resource: www.ebay.com (access date 15.03.2019).

[8]. Kamvar S., Schlosser M., Garcia-Molina H. The EigenTrust Algorithm for Reputation Management in P2P Networks. In Proc. of the 12th International Conference on World Wide Web, 2003, 640-651.

[9]. Kurdi H. HonestPeer: An Enhanced EigenTrust Algorithm for Reputation Management in P2P Systems. Journal of King Saud University - Computer and Information Sciences, vol. 27, no. 3, 2015, 315-322.

[10]. Zhou R., Hwang K. PowerTrust: A Robust and Scalable Reputation System for Trusted Peer-to-Peer Computing. Proc. IEEE Transactions on Parallel and Distributed Systems, vol. 18, no. 4, 2007, 460-473.

[11]. Çelikyılmaz A., Türkşen I.B. Modeling Uncertainty with Fuzzy Logic. With Recent Theory and Applications. Studies in Fuzziness and Soft Computing, vol. 240, 2009, 311 p.

[12]. English Oxford Living Dictionaries, web-resource: https://en.oxforddictionaries.com/ (access date 04.03.19).

[13]. Alam F., Paul A. A Computational Model for Trust and Reputation Relationship in Social Network. In Proc. of the 5th International Conference on Recent Trends in Information Technology, 2016, pp. 1-6.

[14]. Wang Y., Vassileva J. Bayesian Network Trust Model in Peer-to-Peer. Lecture Notes in Computer Science, vol. 2872, 2003, pp. 23-34.

[15]. Gambetta D. Can We Trust Trust? Chapter - Trust: Making and Breaking Co-operative Relations, Dept. of Sociology, University of Oxford, 1980, pp. 213–237.

[16]. Alfarez A.-R., Hailes S. Supporting Trust in Virtual Communities. In Proc. of the 33rd Annual Hawaii International Conference on System Sciences, 2000, pp. 1-9.

[17]. Kreps D.M., Wilson R., 1982. Reputation and Imperfect Information. Journal of Economic Theory, vol. 27, 253-279.

[18]. Hussain J.K., Hussain O.K., Chang E. An Overview of the Interpretations of Trust and Reputation. In Proc. of the IEEE Conference on Emerging Technologies and Factory Automation (EFTA-2007), 2007, pp. 826-830.

[19]. Chiluka N., Andrade N., Gkorou D., Pouwelse J., 2012. Personalizing EigenTrust in the Face of Communities and Centrality Attack. Proc. IEEE 26th Int. Conference on Advanced Information Networking and Applications, 503-510.

[20]. Zhang J. Trust Management Based on Fuzzy Sets Theory for P2P Networks. In Proc. of the WRI World Congress on Software Engineering, 2009, pp. 461-465.

[21]. Semenkovich S., Kolekonova O., Degtiarev K. A Modified Scrum Story Points Estimation Method Based on Fuzzy Logic Approach. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue. 5, 2017, pp. 19-38.

[22]. Zadeh L. The Concept of a Linguistic Variable and Its Application to Approximate Reasoning – I. Information Sciences, vol. 8, no. 3, 1975, pp. 199-249.

[23]. Zadeh L. A. Fuzzy Logic, Neural Networks and Soft Computing. Communications of the ACM, vol. 37, no. 3, 1994, pp. 77-84.

[24]. Zadeh L.A. Fuzzy Languages and Their Relation to Human and Machine Intelligence. In Proc. of the International Conference on Man and Computer, 1972, pp.130-165.

[25]. Zadeh L.A. Similarity Relations and Fuzzy Orderings. Information Sciences, vol. 3, no. 2, 1971, pp. 177-200.

[26]. Zimmermann H.-J. Fuzzy Set Theory – and Its Applications, 4th ed., Springer Science+Business Media, LLC, 2001.
[27]. Zadeh L.A. Fuzzy Sets. Information and Control, vol. 8, no. 3, 1965, pp. 338-353.
[28]. de Barros L.C., Bassanezi R.C., Lodwick W.A. The Extension Principle of Zadeh and Fuzzy Numbers. In A First Course in Fuzzy Logic, Fuzzy Dynamical Systems, and Biomathematics, Studies in Fuzziness and Soft Computing, vol. 347, 2017, pp 23-41.
[29]. Ross T.J. Fuzzy Logic with Engineering Applications, 3rd ed., John Wiley & Sons, 2010, 595 p.
[30]. Awasthi S.K., Singh Y.N. Absolute Trust: Algorithm for Aggregation of Trust in Peer-to-peer Networks, 2016, web-resource: http://arxiv.org/abs/1601.01419 (access date 17.03.2019).
[31]. Rao S., Wang Y., Tao X. The Comprehensive Trust Model in P2P Based on Improved EigenTrust Algorithm. In Proc. of the International Conference on Measuring Technology and Mechatronics Automation, 2010, pp. 822-825.
[32]. Song S., Hwang K., Zhou R., Kwok Y.-K. Trusted P2P Transactions with Fuzzy Reputation Aggregation. IEEE Internet Computing, vol. 9, no. 6, 2005, pp. 24-34.
[33]. Page L., Brin S., Motwani R., Winograd T. The PageRank Citation Ranking: Bringing Order to the Web, Technical Report, Stanford Digital Library Technologies Project, 1998.
[34]. Haveliwala T.H., Kamvar S.D. The Second Eigenvalue of the Google Matrix, Technical Report, Stanford University, 2003.
[35]. Faloutsos M., Faloutsos P., Faloutsos C. On Power-Law Relationship of the Internet Technology. In Proc. of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM-1999), 1999, pp. 251-262.
[36]. Dellarocas C. The Digitization of Word-of-Mouth: Promise and Challenges of Online Reputation Mechanism. Management Science (Special Issue on E-Business and Management Science), vol. 49, no. 10, 2003, pp. 1407-1424.
[37]. Klir G., Yuan B. Fuzzy Sets and Fuzzy Logic Theory and Applications, Prentice-Hall/Upper Saddle River, 1995, 592 p.
[38]. Tóth-Laufer E., Takács M. The Effect of Aggregation and Defuzzification Method Selection on the Risk Level Calculation. In Proc. of the EEE 10th Jubilee International Symposium on Applied Machine Intelligence and Informatics (SAMI), 2012, pp. 131-136.
[39]. Roychowdhury S., Pedrycz W. A Survey of Defuzzification Strategies. International Journal of Intelligent Systems, vol. 16, no. 6, 2001, pp. 679-695.
[40]. Mendel J.M. Uncertain Rule-Based Fuzzy Logic Systems: Introduction and New Directions, 1st ed. Prentice Hall, 2001, 674 p.
[41]. Mendel J.M., John R.I. Type-2 Fuzzy Sets Made Simple. IEEE Transactions on Fuzzy Systems, vol. 10, no. 2, 2002, pp, 117-127.
[42]. Dubois D. Fuzziness, Uncertainty and Vagueness: Toward a Less Blurry Picture (Is a Fuzzy Set Vague?), 2008, web-resource: https://www.logic.at/lomorevi/LoMoReVI/transvague.pdf (access date 14.06.2019).

Информация об авторах / Information about authors

Mikhail Mikchailovitch NOSOVSKIY is a student of the Bachelor's degree program «Software Engineering» at the National Research University Higher School of Economics (HSE), Moscow, Russia. His research interests include fuzzy modeling, data analysis and identification of fraud activity.

Михаил Михайлович НОСОВСКИЙ является студентом образовательной программы бакалавриата «Программная инженерия» в Национальном исследовательском университете «Высшая школа экономики» (НИУ ВШЭ), Москва, Россия. Его исследовательские интересы включают в себя нечеткое моделирование, анализ данных и выявление фродовой (мошеннической) активности.

Konstantin Yurievitch DEGTIAREV earned his M.S. degree in applied mathematics ('engineer-mathematician' qualification) from Moscow Institute of Electronic Engineering, Russia, and his Ph.D. degree in engineering sciences from Moscow Academy of Instrument Engineering and Informatics, Russia. He is currently an Associate Professor at the Software Engineering

Department of the Faculty of Computer Science at the National Research University Higher School of Economics (HSE) in Moscow. His research interests include fuzzy logic/soft computing in system analysis, verbal computing, perceptions and representations in system analysis, use of fuzzy time series in forecasting. He is a Member of the IEEE (Systems, Man and Cybernetics Society).

Константин Юрьевич ДЕГТЯРЕВ получил степень магистра (специалитет) прикладной математики в Московском институте электронного машиностроения и степень кандидата технических наук в Московской академии приборостроения и информатики, Россия. В настоящее время он является доцентом кафедры программной инженерии факультета компьютерных наук Национального исследовательского университета «Высшая школа экономики» (НИУ ВШЭ) в Москве. Его исследовательские интересы включают нечеткую логику/мягкие вычисления в системном анализе, вербальные вычисления, восприятие и представление в системном анализе, применение нечетких временных рядов в прогнозировании. Он является членом IEEE (общество 'Systems, Man and Cybernetics').

DOI: 10.15514/ISPRAS-2019-31(3)-10

Applying High-Level Function Loop Invariants for Machine Code Deductive Verification

P.A. Putro, ORCID: 0000-0001-9540-8321 <pavel.putro@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia
National Research University Higher School of Economics,
20, Myasnitskaya st., Moscow, 101000 Russia

Abstract. The existing tools of deductive verification allow us to successfully prove the correctness of functions written in high-level languages such as C or Java. However, this may not be enough for critical software because even fully verified code cannot guarantee the correct generation of machine code by the compiler. At the moment, developers of such systems have to accept the compiler correctness assumption, which, however, is extremely undesirable, but inevitable due to the lack of full-fledged systems of formal verification of machine code. It is also worth noting that the verification of machine code by a person directly is an extremely time-consuming task due to the high complexity and large amounts of machine code. One of the approaches to simplify the verification of machine code is automatic deductive verification reusing the formal specification of the high-level language function. The formal specification of the function consists of the specification of the pre- and postcondition, as well as loop invariants, which specify conditions that are satified at each iteration of the loop. When compiling a program into machine code, pre- and postconditions are preserved, which, however, cannot be said about loop invariants. This fact is one of the main problems of automatic verification of machine code with loops. Another important problem is that high-level function variables often have ‘projections' to both registers and memory at the machine code level, and the verification procedure requires that invariants be described for each variable, and therefore the missing invariants must be generated. This paper presents an approach to solving these problems, based on the analysis of the control flow graph, and intelligent search of the locations of variables.

Keywords: deductive verification; formal methods; machine code

For citation: Putro P.A. Applying High-Level Function Loop Invariants for Machine Code Deductive Verification. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 3, 2019. pp. 123-134. DOI: 10.15514/ISPRAS-2019-31(3)-10

Использование инвариантов функции высокого уровня для дедуктивной верификации машинного кода

П.А. Путро, ORCID: 0000-0001-9540-8321 <pavel.putro@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25
Национальный исследовательский университет “Высшая школа экономики”
101000, Россия, г. Москва, ул. Мясницкая, д. 20

Аннотация. Существующие на сегодняшний день инструменты дедуктивной верификации позволяют успешно доказывать корректность функций, написанных на высокоуровневых языках, таких как C или Java. Однако для критического ПО этого может быть недостаточно, поскольку даже полностью верифицированный код не может гарантировать корректной генерации машинного кода компилятором. На данный момент разработчикам таких систем приходится принимать

предположение о корректности компилятора, что, однако, является крайне нежелательным, но неизбежным поступком в силу отсутствия полноценных систем формальной верификации машинного кода. Стоит также отметить, что верификация машинного кода человеком напрямую является крайне трудоёмкой задачей из-за высокой сложности и больших объёмов машинного кода. Одним из подходов, позволяющих упростить верификацию машинного кода, является автоматическая дедуктивная верификация с переиспользованием формальной спецификации функции языка высокого уровня. Формальная спецификация функции состоит из спецификации пред- и постусловия, а также инвариантов циклов, позволяющих определить какие условия сохраняются на каждой итерации цикла. При компиляции программы в машинный код пред- и постусловия сохраняются, что, однако, нельзя сказать об инвариантах циклов. Этот факт является одной из основных проблем автоматической верификации машинного кода с циклами. Другой немаловажной проблемой является то, что локальные переменные функций высокого уровня могут иметь ‘позиции’ как на регистры, так и на память на уровне машинного кода. Если абстрагироваться от конкретного компилятора, то не существует строгих правил сопоставления локальных переменных их позициям, а процедура верификации инвариантов циклов, тем не менее требует того, чтобы локальным переменным были сопоставлены конкретные позиции. В данной работе приводится подход к решению этих проблем, а также рассматриваются альтернативные пути решения, предложенные в аналогичных исследованиях.

Ключевые слова: дедуктивная верификация; формальные методы; машинный код.

Для цитирования: Путро П.А. Использование инвариантов функции высокого уровня для дедуктивной верификации машинного кода. Труды ИСП РАН, том 31, вып. 3, 2019 г., стр. 123-134 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(3)-10

1. Introduction

In the 1960s, Floyd [1] and Hoare [2] put forward their theories that the full correctness of the program code can be proved mathematically. The proposed methods are called deductive verification, but could not immediately gain popularity due to the lack of automation, as well as low performance and the high cost of hardware computing resources. However, in recent decades, these methods are experiencing a rebirth due to the rapid development of methods for solving the SMT [3] problem, and growing performance of hardware devices. Technological leap in this area allowed to verify the application and system software by means of personal computers. In this paper, the author adheres to the use of methods of deductive verification, because unlike other methods of formal verification, such as for example model checking, deductive verification allows proving full correctness, but not only the absence of a certain class of errors.

Increasing the availability of formal verification methods has led to the fact that now formal verification is becoming a standard in the creation of systems designed to work with safety- and security-critical infrastructure. These systems are verified and tested carefully, but industrial verification tools work only at source code level when testing can't guarantee that are no errors in the program. Here we can notice a security hole when compilation of the correct code introduces errors that can't be detected by the testing system. Without anyways for solving this problem – developers have to make «The assumption of the correctness of the compiler». According to a study [4] conducted in 2016, the total number of bugs found in GCC+LLVM are more than 50000. This is one of the main reasons why this assumption may not be sufficient for critical systems. There are only two ways that can allow developers to abandon this assumption: the first is to create a fully formally verified compiler, and the second is to formally verify machine code. There are some tries in recent 15 years that aimed to verify machine code or to create fully formally verified compiler but there is still no generally accepted industrial solution in machine code verification.

In this paper we consider an approach of deductive verification of machine code obtained by compiling the source code, the correctness of which has been proved by methods of deductive verification. The approach proposes to use a number of techniques designed to reuse the function specification in a high-level language to prove the correctness of the compiled machine code.

The process of deductive verification of machine code has several serious differences from deductive verification of code in high-level languages. The first difference is that in machine-independent high-level programming languages, the set of basic operations amounts to several tens, and the size of instruction sets of modern processors amounts to hundreds and may even exceed a thousand different instructions. In addition, many of these instructions can have side effects, such as setting processor flags or storing the result in a predefined register. This variety of instructions does not allow to effectively generate state-change formulas during parsing of machine code and requires a definition of the processor model and its instruction set. The second difference is that machine code is always a sequence of instructions with operands and does not have the complex syntax that is present in modern programming languages. This feature allows you to automate the parsing of the machine code of different processors using only one tool. The third difference is that in machine code there is no explicit design to indicate the loops, such as operators «for» or «while» in languages C/C++. Instead, loops are organized by using a set of conditional and unconditional branches. However, the presence of such instructions does not mean that there are loops in the program because they are also used to organize any branching. This difference requires the construction and analysis of the control flow graph of the program. In this case, the control flow graph extractor should be able to distinguish branches from other processor instructions, be able to determine the target of the transition and the condition under which it will be done. If there is a control flow graph, the problem of finding loops in the program can be reduced to the problem of finding the components of strong connectivity in the oriented graph. The last significant difference is the absence of a direct connection between the names and positions of local variables in a high-level language program and their location in memory or in the registers of the program in machine code. It is necessary when we try to reuse specifications of the high-level function. A similar dependence exists when mapping function parameters to registers and stack and is determined by the target processor ABI. Using information about ABI allows you to automatically map the parameters and the result of the function to the appropriate positions in the machine code. However, the process of proving loop invariants involves the use of local variables. As a result, when trying to prove the invariants of a high-level function at the machine code level, there is a problem associated with the absence of the ability to directly associate high-level local variables with machine code. There are also other differences related to program function calls, system calls, and exception handling, but these are beyond the scope of this paper. As you can see the first two considered differences are common and observed in the processing of any machine code, while the second two appear only in the case of processing functions with loops. In this paper, the most attention is paid to the solution of the problems caused by the second two differences while the previous author's paper is devoted to the first two [5].

2. Related work

In [6], the HOL4 proof assistant [7] is used to verify machine programs in subsets of ARM, PowerPC, and x86 (IA-32). These ISAs were specified independently: the ARM and x86 models [8], [9] were written in HOL4 while the PowerPC model [10] was written in Coq [11] and then manually translated to HOL4. There are four levels of abstraction. Machine code (level 1) is automatically decompiled into the low-level function model in HOL4 (level 2). A user describes the high-level function model (level 3) as well as the functional specification (level 4). By proving the equivalence between the levels, the user ensures that the machine code complies with the functional specification. In our opinion, automation can be increased by using specialized ISA description languages and SMT solvers. For proving the correctness of the programs with loops, it uses loops to recursive functions translation technique. This technique is available only for interactive provers due to efficiency problems of automatic solvers while processing programs with loops.

An interesting approach aimed at verifying machine code against ACSL [12] specifications is presented in [13]. The general scheme is as follows: first, the ACSL annotations are rewritten as an inline assembly; second, the modified sources are compiled into assembly language; third, the

assembly code is translated into a Why program; finally, the Why environment generates verification conditions and proves them with external solvers. The approach looks similar to the proposed one; however, there some distinctions. E.g., there are separate primitives for storing/loading variables of a different type (32- and 64-bit integers, single and double floating-point numbers, etc.), which leads to certain limitations in dealing with pointers. It is also worth noting that verification at the assembly level does not allow us to abandon the compiler correctness assumption, as the assembly code is an intermediate form and needs additional translation. This approach relies on the compiler while processing programs with loops, as compiler places rewritten as inline assembly loop invariants into the beginning of the loop and automatically bind local variables to the registers or memory.

In [14], there have been demonstrated the possibility of reusing correctness proofs of high-level programs for the related machine code verification. The approach is illustrated on the example of a Java-like source language and a bytecode target language. The paper describes a scenario of using such a technology in the context of proof-carrying code (PCC) and shows (in a particular setting) that compilation preserves proof obligations, i.e. source code proofs (built either automatically or interactively) can be transformed to the machine code proofs. The problem we are solving is different (though some ideas may be useful); moreover, we would like to make our solution architecture and compiler independent. In the case of the processing of the programs with loops, if loop invariants are preserved by compilation, their proving will be a trivial process.

3. Using the control flow graph for VC generation

In deductive verification of programs in high-level languages, various syntactic constructions allow determining the presence of loops in the program, their contents, as well as the conditions of exit from the loop. However, when processing the machine code such structures do not exist, and to search for loops and other branch operations need to build a control flow graph (CFG). As part of the study for the processing of machine code used MicroTESK toolkit [15] (full justification for the use of MicroTESK for deductive verification of machine code is given in paper [5]). The use of this tool, in particular, allows to describe the processor model in the language of nML [16], and on the basis of this model to automatically analyze the binary code and build its behavior model in the logical language SMT-LIB [17]. In addition, CFG extractor has been added to this tool over the past year.

3.1 The format of the CFG

MicroTESK toolkit is able to determine whether the instruction described in nML is a branch instruction, determine the branch condition and the target address. In addition, MicroTESK has an advanced algorithm for calculating the target address of the transition, which allows it to calculate indirect targets, such as in a situation where the target address is preloaded into the register and the branch is carried out already on the register. Such capabilities in combination with the use of nML processor models allow you to automatically generate CFG for any processor modeled using nML. The generated CFG is saved in JSON format [18], and has the following format.

1) All basic blocks are placed in the list with the name «blocks».
2) Each basic block has an index in the «blocks» list and has the following format:
    a. The «range» list that includes the sequence number of the first and last instruction of the base block in the context of the entire function being analyzed. Used for extraction of the SMT-LIB representation of the block from the SMT-LIB representation of machine code.
    b. The «asm» list that contains instructions of a basic block in the assembler language of the target processor.
    c. The «vars_start» list, which contains the SMT-LIB versions of the main variables of the nML model of the processor such as registers and memory, but not temporary and auxiliary variables. Versions are specified for the entry point of the basic block.

d. The «vars_end» list contains values similar to the list of «vars_start», however, the version specified for the exit point of the basic block.
e. The field «condition» contains the branch condition. The MicroTESK nML internal representation syntax is used to write the condition «true» for unconditional branches and in the case when there is no branch in the block.
f. The field «condition_smt» same as «condition», however, is recorded using SMT-LIB.
g. The field «target_taken» containing the index of the basic block in the «blocks» list, which will be passed to the control in the case when «condition» is met, and «null» for blocks, in which is the function exit point.
h. Optional field «target_ntaken» containing the index of the basic block in the "blocks" list, which will be passed to the control in the case when «condition» is not met. Defined only for blocks with a conditional branch.

This structure of the graph contains all the necessary information for generating verification conditions. Below is an example of the extracted CFG for the function of calculating the sum of numbers from 0 to N (Table 1).

```
{
  "blocks": [
  {
    "range": [0, 8],
    "vars_start": ["MEM!1","XREG!1"],
    "vars_end": ["MEM!37","XREG!15"],
    "asm": [
      "addi sp, sp, -48",
      "sd s0, 40(sp)",
      "addi s0, sp, 48",
      "addi a5, a0, 0",
      "sw a5, -36(s0)",
      "sw zero, -20(s0)",
      "addi a5, zero, 1",
      "sw a5, -24(s0)",
      "jal zero, 0x10"
    ],
    "condition": "true",
    "target_taken": 1
  },
  {
    "range": [16, 20],
    "vars_start": ["MEM!53","XREG!27"],
    "vars_end": ["MEM!53","XREG!36"],
    "asm": [
      "lw a4, -24(s0)",
      "lw a5, -36(s0)",
      "addiw a4, a4, 0",
      "addiw a5, a5, 0",
      "bge a5, a4, -22"
    ],
    "condition": "i1 sge i64 a5, a4",
    "condition_smt":
"op_20_instruction.operation.action.block_0!1",
    "target_taken": 2,
    "target_ntaken": 3
  },
  {
    "range": [9, 15],
    "vars_start": ["MEM!37","XREG!15"],
    "vars_end": ["MEM!53","XREG!27"],
    "asm": [
```

```
            "lw a4, -20(s0)",
            "lw a5, -24(s0)",
            "addw a5, a4, a5",
            "sw a5, -20(s0)",
            "lw a5, -24(s0)",
            "addiw a5, a5, 1",
            "sw a5, -24(s0)"
          ],
          "condition": "true",
          "target_taken": 1
        },
        {
          "range": [21, 25],
          "vars_start": ["MEM!53","XREG!36"],
          "vars_end": ["MEM!53","XREG!45"],
          "asm": [
            "lw a5, -20(s0)",
            "addi a0, a5, 0",
            "ld s0, 40(sp)",
            "addi sp, sp, 48",
            "jalr zero, ra, 0"
          ],
          "condition": "true",
          "target_taken": null
        }
        ]
      }
```

Table 1. Example: ACSL-annotated C code, RISC-V assembler code and machine code of sum function

ACSL-annotated C code	Assembly code	Machine code

/*@ axiomatic Sum {
*@ logic integer sum(integer n);
 	
*@ axiom sum_init:
*@ \forall integer n;
*@ n <= 0 ==> sum(n) == 0;
 	
*@ axiom sum_step_dec:
*@ \forall integer n;
*@ n > 0 ==> sum(n) == sum(n-1) + n;
*@ }
*/
 	
/*@ requires 0 <= n <= 65535;
*@ ensures \result == sum(n);
*/
int sum(int n) {
  int s = 0;
  /*@ loop invariant 1 <= i <= n+1;
  *@ loop invariant s == sum(i-1);
  *@ loop variant n-i;
  */
  for(int i = 1; i <= n; i++) {
  s += i;
  }
  return s;
}
	addi sp, sp, -48
	sd s0, 40(sp)
	addi s0, sp, 48
	addi a5, a0, 0
	sw a5, -36(s0)
	sw zero, -20(s0)
	addi a5, zero, 1
	sw a5, -24(s0)
	jal zero, 0x10
	lw a4, -20(s0)
	lw a5, -24(s0)
	addw a5, a4, a5
	sw a5, -20(s0)
	lw a5, -24(s0)
	addiw a5, a5, 1
	sw a5, -24(s0)
	lw a4, -24(s0)
	lw a5, -36(s0)
	addiw a4, a4, 0
	addiw a5, a5, 0
	bge a5, a4, -22
	lw a5, -20(s0)
	addi a0, a5, 0
	ld s0, 40(sp)
	addi sp, sp, 48
	jalr zero, ra, 0
		fd01 0113
		0281 3423
		0301 0413
		0005 0793
		fcf4 2e23
		fe04 2623
		0010 0793
		fef4 2423
		0200 006f
		fec4 2703
		fe84 2783
		00f7 07bb
		fef4 2623
		fe84 2783
		0017 879b
		fef4 2423
		fe84 2703
		fdc4 2783
		0007 071b
		0007 879b
		fce7 dae3
		fec4 2783
		0007 8513
		0281 3403
		0301 0113
		0000 8067

3.2 Joining basic blocks for verification conditions generation

The basic blocks themselves are not suitable targets for generating verification conditions (VC), as they may not contain specific targets, but only state change formulas. There are several types of verification conditions in deductive verification. The first and foremost is the postcondition. Also as VC can be used various custom asserts or conditions for checking the security of the program execution, such as for example the absence of indexing out of range of the array. Also, as VC uses invariants of loops. In this case, each invariant can be further divided into checking the initialization of this invariant – that is, checking the condition of the invariant before the execution of the loop code, as well as checking the preservation of the invariant - the preservation of the compliance of the invariant for the next iteration of the loop, provided that all the invariants are compliances on the current one. Therefore, the basic blocks must be joined and marked so that one or more of these conditions can be matched to each of them. Accordingly, the algorithm for combining the base blocks can be defined as follows. In the first step, using the fields "target_taken" and "target_ntaken", the array of edges of the CFG is selected from the set of basic blocks. In the second step, to search for loops in the program, the graph uses an algorithm to search for strongly related components in a directed graph. The author's implementation uses Tarjan's algorithm [19] implemented by the ocaml-containers library [20]. To find nested loops, this step must be repeated recursively for all found base block sets, and the relationship between the first and last base block in the loop must be broken. In the third step, you need to depth-first search the graph for marking and joining blocks. The traversal must start from the zero base block – the program entry point. At the input, there is a set of basic blocks, as well as a set of chains of strongly related component - loops. The output is a set of joined and marked basic blocks suitable for VC generation.

1) If the block has two targets, they must be processed separately, and the results combined.
2) If the current block and its target is not in the loop– it is necessary to "join" these blocks and proceed to the processing of the joined block.
3) If the current block is not included in the loop, and its target is included in the loop, it must be marked as the loop entry point. Next, proceed to the processing of its target.
4) If the block and its target are in the same loop and do not make a loop, they must be joined and proceed to the processing of the joined block.
5) If the block and its targets are in the same loop and thus make a loop, they must be combined and the result is returned.
6) If the unit is part of the loop, and its target is included in a nested loop it is necessary to mark as a loop entry point and proceed to the processing its target.
7) If the block is included in the nested loop, and its target in the outer loop, then the block must be marked as the loop exit point and joined with the target and proceed to the processing of the joined block.
8) If the block in the loop but its target is no, then the block must be marked as the exit point of the loop and joined with the target and proceed to process the joined block.
9) If the block target is null, the result must be returned.

The procedure of joining blocks is the base for the graph traversal and is carried out according to the following rules.

1) The procedure allows you to create a new block based on two existing ones.
2) Joining is possible if the target («target_taken» or «target_ntaken») of the first block is the second block. In all other cases, the result of the join is not determined.
3) The targets of the joined block will be the targets of the second block.
4) Condition («condition_smt») of the joined block will be the condition of the second block.

5) If the second block is a loop exit point, the initial state «vars_start» of the combined block will be the initial state of the second block, otherwise the initial state of the first block.
6) If at least one of the joining blocks is marked as the loop exit point, the joined block must also be marked as the loop exit point.
7) If the second block is marked as the loop entry point, the result should be marked as the loop entry point.
8) If the second block «closes» the loop, i.e. its target is the first block, its SMT-LIB representation should be changed so that all elements of the final state «vars_end» of the second block should get new unique names. Any other conflicts between any variables in SMT-LIB representations of the joining blocks must be resolved in the same way.
9) SMT-LIB the representation of the joined block must be obtained by concatenating the SMT-LIB representations of the merged blocks. In this case, if the condition «condition» of the first block is not empty («true»), it must be added as SMT-LIB assert to the representation code of the joined block. Also, if the join follows the «target_ntaken» branch, the condition must be inverted. Also, if the blocks do not follow each other in the program, the final state of the first block also needs to be associated with the initial state of the second block at the level of the SMT-LIB representation.

As a result of following this algorithm, in most cases, you can create a set of code blocks on which you can directly prove various verification conditions. The algorithm allows processing machine code with loops, nested loops, sequential loops, as well as code generated by the presence of the brake and continue statements in the program, but is not able to cope with tasks when, for example, several entries to one loop and other non-trivial situations caused by the use of transition instructions are detected in the control flow graph. However, such situations cause difficulties already at the stage of verification of the source code, and the construction of an algorithm that allows you to automatically deductively verify any machine code is an unsolvable task.

If we apply the algorithm to the CFG function of the sum of the numbers presented above, we will be able to allocate three blocks to prove VC. The first block will have index 0, have loop entry status and be used to prove the correctness of the initialization of the loop invariants. The second block will be a join of blocks 1 and 2 and will be used to prove the preservation of loop invariants. The third block will be a join of blocks 1 and 3 and will be used to prove the postcondition provided the invariants are correct.

4. Automatic binding of high- and low-level local variables

In general, to describe loop invariants, high-level functions use local variable names that are not available when working with machine code. In general, information about binding local variables to specific positions on the stack or registers is not available. Of course, you can require the user to manually provide this data and even give examples where such requirements will have a positive impact on system performance. However, in most cases, manual mapping of local variables to their positions will be a bottleneck in the performance of the verification system, as well as reduce the degree of automation. From the above, we can conclude that the system should automatically determine the location of local variables, and the possibility of their manual input should be optional. To determine the positions of local variables, the author has developed an algorithm for efficient search of positions in the VC generation, which includes the following steps.

1) All potential positions of local variables are calculated. This can be done both by means of machine code analysis (similar to those used by modern disassemblers and debuggers) and with the help of an existing logical model of machine code and SMT-solver. The author proposes to use the second option because it is a more universal approach. In this approach, for each memory write instruction it is required to prove by solver that there are no positions on the stack that could change as a result of the operation. If solver managed to generate a

counterexample, the position found is a potential position for the local variable. This algorithm allows you to find positions for local variables, but it can be difficult if there is an array on the stack. In this case, if solver fails to determine which position of the array was recorded, the result may not be determined and the user will have to specify the position himself. Similarly, you can calculate the registers to which the local variable can be mapped.

2) Each local variable is assigned a potential positions set, which may depend on its size in bytes.
3) For each invariant from the list of invariants for the proof, the «cost» of proving its correct initialization should be calculated. Here, the cost is the number of all possible combinations of potential positions of local variables involved in the description of this invariant. Here it is necessary to take into account that each variable has its own unique memory location, therefore, combinations of potential positions should consist only of unique values. It is also worth noting that this step should be carried out only when proving the initialization of the invariants, since the initialization of the invariant is proved independently of the other loop invariants, and the proof of preserving the loop invariant must be proved given that all other invariants must be satisfied. Thus, when proving the correct initialization of the invariant, it is possible to reduce the number of local variables necessary for binding, and, as a consequence, the «cost» of the proof may differ for different invariants. If it is necessary to prove the loop invariant preservation the cost of all invariants should be considered equal to.
4) For the least cost invariant, it is necessary to try to prove correctness for each possible combinations of potential positions of local variables on which the invariant depends. The results («unsat»/«unknown»/«sat» verdicts) should be saved in a separate list.
5) If there was no one verdict is «unsat», it is necessary to mark the invariant unproven. If at least one verdict «unsat» has been obtained, then the invariant should be considered proved and the potential positions of local variables on which the proved invariant depends should be filtered, leaving only those positions that consist in combinations of potential positions for which the invariant has been proved (the «unsat» verdict).
6) Remove the proved invariant from the list of invariants for the proof and (if there are still invariants in the list) proceed to step 3.

Using this algorithm, it is possible in some cases to significantly reduce the number of generated targets relative to a complete search. As an example, let's take a function with 3 different local variables with values $s = -1, i = 0, n = 90$ at the loop entry point and three potential positions on the stack: $sp - 0x10, sp - 0x18, sp - 0x24$, respectively. For simplicity, the size of variables and positions will be considered equal to 4 bytes. Initially, the correspondence between the positions of local variables is not known. Based on these data, we try to prove the initialization of the following invariants: $0 <= i < 100, s == 2 * i - 1$ and $s < n + i$. According to the algorithm, we calculate the cost of proving the invariants, which will be equal to 3, 6 and 6, respectively. Next, try to prove the correctness of the first invariant with the cost of 3 and will find two potential positions ($sp - 0x18$ and $sp - 0x24$) for the variable i (we will also be able to prove the invariant for variable n). Second, we perform filtering to remove the position of the $sp - 0x10$ for the variable i. Recalculate the costs of the remaining invariants: the result of 4 and 4 (the cause of reducing the cost is reducing the number of potential positions for the variable i, on which the invariants depend). When proving the correctness of the initialization of the second invariant, only one set of potential positions for the variables s and i is $sp - 0x10$ and $sp - 0x18$, respectively, will be selected. We filter and proceed to the proof of the last invariant. It is possible to select only one target for it – the position for n will be selected by the elimination method. We prove the invariant, perform filtering and get the same correspondence between variables and their positions on the stack, which was set by the compiler.

5. Evaluation

At the time of writing, the approach proposed by the author was partially implemented in the system of deductive verification of machine code [5]. Using this approach, the machine code of the function of the sum of numbers from 0 to N (Table 1), as well as some other functions with loops, was successfully verified. For each generated VC, a verdict was obtained confirming its correctness. In addition, the positions of local variables on the function stack were strictly determined by the computer during the verification process. More complex testing is planned after the full implementation of the approach.

6. Conclusion

Verification of functions with loops is one of the main stumbling blocks in the verification of machine code. Various research groups have proposed various solutions that however impose serious limitations, such as the need to use interactive proof assistants or the introduction of dependency on the target compiler. However, due to the high complexity of the machine code structure, the use of interactive proof assistants can significantly slow down the verification process and require very experienced staff. Dependency on the target compiler also reduces the universality of the approach and requires its integration into the source code compilation process, which can cause some difficulties. In contrast to these works, the author proposes to use a compiler-independent approach based on the use of automatic SMT-solvers. To implement the approach, we propose to use two main algorithms, as well as a tool for CFG extraction. The first algorithm allows the basic blocks of the function CFG to be joined in such a way that they become suitable for VC proving. The second algorithm allows restoring the lost links between the local variables of the high-level function and their positions in memory or on the processor registers at the machine code level. Using this approach allows in most cases to generate such VC, which will be sufficient for deductive verification of the machine code of the function with loops.

The work is in progress. At the moment, the approach has been partially implemented in the system of deductive verification of machine code. Its full implementation and testing is the nearest direction for further work. Also among the possible areas for further research can be identified the study of problems arising in the proof of the correctness of functions containing calls to other functions or system calls. There is a separate issue with the security check of the machine code execution, i.e. the absence of exceptions at the processor level, incorrect memory readings or stack overflows. Also of great importance is the study of the applicability of the machine code deductive verification system for solving real industrial problems.

References

[1]. R.W. Floyd. Assigning Meanings to Programs. Mathematical Aspects of Computer Science, vol. 19, 1967. P. 19-32.

[2]. C.A.R. Hoare. An Axiomatic Basis for Computer Programming. Communications of the ACM, vol. 12, no. 10, 1969, pp. 576-585.

[3]. C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability Modulo Theories. In Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, 2009, pp. 825-885.

[4]. C. Sun, V. Le, Q. Zhang, Z. Su. Toward understanding compiler bugs in gcc and llvm. In Proc. of the 25th International Symposium on Software Testing and Analysis, 2016, pp. 294-305.

[5]. Putro P.A. Combining ACSL Specifications and Machine Code. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 4, 2018. pp. 95-106. DOI: 10.15514/ISPRAS-2018-30(4)-6

[6]. M.O. Myreen. Formal Verification of Machine-Code Programs. Ph.D. Thesis. University of Cambridge, 2009, 131 p.

[7]. K. Slind, M. Norrish. A Brief Overview of HOL4. Lecture Notes in Computer Science (LNCS), vol. 5170, 2008, pp. 28-32.

[8]. A. Fox. Formal Specification and Verification of ARM6. Lecture Notes in Computer Science (LNCS), vol. 2758, 2003, pp. 25-40

[9]. K. Crary, S. Sarkar. Foundational Certified Code in a Metalogical Framework. Technical Report CMU-CS-03-108. Carnegie Mellon University, 2003. 19 p.
[10]. X. Leroy. Formal Certification of a Compiler Back-End or: Programming a Compiler with a Proof Assistant. In Proc. of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 42-54.
[11]. Y. Bertot. A Short Presentation of Coq. Lecture Notes in Computer Science, vol. 5170, 2008, pp. 12-16.
[12]. P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, V. Prevosto. ACSL: ANSI/ISO C Specification Language. Version 1.13, 2018, 114 p.
[13]. T.M.T. Nguyen, C. Marché. Hardware-Dependent Proofs of Numerical Programs. Lecture Notes in Computer Science, vol. 7086, 2011, pp. 314-329.
[14]. G. Barthe, T. Rezk, A. Saabas. Proof Obligations Preserving Compilation. Lecture Notes in Computer Science, vol. 3866, 2005, pp. 112-126.
[15]. MicroTESK Framework – http://www.microtesk.org
[16]. M. Freericks. The nML Machine Description Formalism. Technical Report TR SM-IMP/DIST/08, TU Berlin CS Department, 1993, 47 p.
[17]. C. Barrett, P. Fontaine, C. Tinelli. The SMT-LIB Standard Version 2.6. Release 2017-07-18, 104 p.
[18]. JavaScript Object Notation – https://www.json.org/
[19]. R. E. Tarjan, Dep-first search and linear graph algorithms. SIAM Journal on Computing, vol. 1, no. 2, 1972, pp. 146-160.
[20]. ocaml-containers library – https://github.com/c-cube/ocaml-containers.

Информация об авторах / Information about authors

Павел ПУТРО получил степень бакалавра в области программной инженерии в Национальном исследовательском университете «Высшая школа экономики», Москва, Россия. В настоящее время он продолжает обучение в этом университете по магистерской программе «Системное программирование». Работает в институте системного программирования им. В.П. Иванникова РАН. Исследовательские интересы включают дедуктивную верификацию, логическое программирование и статический анализ машинного кода.

Pavel PUTRO received a bachelor's degree in software engineering from the National Research University Higher School of Economics, Moscow, Russia. Currently, he is continuing his studies at this University on the master's program «System programming". He works at the Ivannikov Institute for System Programming of the RAS. His research interests include deductive verification, logic programming, and machine code static analysis.

DOI: 10.15514/ISPRAS-2019-31(3)-11

Extracting Assertions for Conflicts in HDL Descriptions

[1,2,3,4]*A.S. Kamkin, ORCID: 0000-0001-6374-8575 <kamkin@ispras.ru>*
[1]*M.S. Lebedev, ORCID: 0000-0002-0207-7672 <lebedev@ispras.ru>*
[1]*S.A. Smolov, ORCID: 0000-0003-0173-3081 <smolov@ispras.ru>*
[1]*Ivannikov Institute for System Programming of the Russian Academy of Sciences,*
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia
[2]*Lomonosov Moscow State University,*
GSP-1, Leninskie Gory, Moscow, 119991, Russia
[3]*Moscow Institute for Physics and Technology,*
9, Institutskiy per., Dolgoprudny, Moscow Region, 141701, Russia
[4]*National Research University Higher School of Economics,*
20, Myasnitskaya st., Moscow, 101000, Russia

Abstract. Data access conflicts may arise in hardware designs. One of the ways of detecting such conflicts is static analysis of hardware descriptions in HDL. We propose a static analysis-based approach to data conflicts extraction from HDL descriptions. This approach has been implemented in the Retrascope tool. The following types of conflicts are considered: simultaneous reads and writes, simultaneous writes, reading of uninitialized data, no reads between two writes. Conflict assertions are formulated as conditions on variables. HDL descriptions are automatically translated into formal models suitable for the nuXmv model checker. The translation process consists of the following steps: 1) preliminary processing; 2) Control Flow Graph (CFG) building; 3) CFG transformation into a Guarded Actions Decision Diagram (GADD); 4) GADD translation into a nuXmv format. Conflict assertions are automatically built using static analysis of the GADD model and passed to the nuXmv model checker. Bounded model checking is used to check whether these assertions are satisfiable. If true, counterexamples are generated and then translated to HDL testbenches by the Retrascope tool. The proposed approach was applied to several open source HDL benchmarks like Texas-97, Verilog2SMV, VCEGAR and mips16 modules. Potential conflicts have been detected for all of these benchmarks. Future work includes propagation of conflict assertions to the interface level (thus getting assertions on modules' communication protocols) and generation of built-in HDL checkers.

Keywords: hardware design; hardware description language; functional verification; static analysis; test generation; data access conflict; control flow graph; guarded action; guarded actions decision diagram; model checking.

For citation: Kamkin A.S., Lebedev M.S., Smolov S.A. Extracting Assertions for Conflicts in HDL Descriptions. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 3, 2019. pp. 135-144. DOI: 10.15514/ISPRAS-2019-31(3)-11

Поиск конфликтов доступа к данным в HDL-описаниях

[1,2,3,4] *А.С. Камкин, ORCID: 0000-0001-6374-8575 <kamkin@ispras.ru>*
[1] *М.С. Лебедев, ORCID: 0000-0002-0207-7672 <lebedev@ispras.ru>*
[1] *С.А. Смолов, ORCID: 0000-0003-0173-3081 <smolov@ispras.ru>*
[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*
[2] *Московский государственный университет имени М.В. Ломоносова, 119991, Россия, г. Москва, Ленинские горы, д. 1*
[3] *Московский физико-технический институт, 141701, Россия, Московская обл., г. Долгопрудный, Институтский пер., д.9*
[4] *Национальный исследовательский университет «Высшая школа экономики», 101000, Россия, г. Москва, ул. Мясницкая, д. 20*

Аннотация. При проектировании модулей цифровой аппаратуры могут возникать конфликты доступа к данным. Одним из способов их выявления на ранних стадиях проектирования является статический анализ описаний цифровой аппаратуры (или HDL-описаний). В данной статье описывается метод поиска конфликтов доступа к данным в HDL-описаниях. Метод реализован в инструменте Retrascope и ориентирован на конфликты следующих типов: одновременные чтение и запись; одновременная запись; обращение к неинициализированным данным; отсутствие чтения между двумя актами записи. Конфликты задаются в виде условий (assertion) на внутренние переменные. Входное HDL-описание автоматически транслируется в формальную модель на языке, являющемся входным для инструмента проверки моделей nuXmv. Трансляция включает следующие этапы: 1) предварительная обработка; 2) построение графа потока управления; 3) трансформация графа потока управления в решающую диаграмму охраняемых действий (GADD-модель); 4) трансляция GADD-модели в формат инструмента nuXmv. Условия возникновения конфликтов строятся автоматически на основе статического анализа GADD-модели и передаются инструменту проверки моделей nuXmv. Найденные контрпримеры (последовательности значений входных сигналов, приводящие к достижению конфликта) автоматически транслируются инструментом Retrascope в тесты, которые могут быть исполнены на симуляторе. Предложенный метод поиска конфликтов был применен к ряду открытых тестовых наборов и модулей – Texas-97, Verilog2SMV, VCEGAR, mips16. Были выявлены потенциальные конфликты для всех указанных категорий. В качестве направлений дальнейших исследований рассматриваются вынос условий конфликтов на уровень входных сигналов (и получение, таким образом, сведений о протоколах взаимодействия между модулями), а также генерация встроенных проверок в коде HDL-описаний.

Ключевые слова: разработка аппаратуры; язык описания аппаратуры; функциональная верификация; статический анализ; генерация тестов; конфликт доступа к данным; граф потока управления; охраняемое действие; решающая диаграмма охраняемых действий; проверка модели.

Для цитирования: Камкин А.С., Лебедев М.С., Смолов С.А. Поиск конфликтов доступа к данным в HDL-описаниях. Труды ИСП РАН, том 31, вып. 3, 2019 г., стр. 135-144 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(3)-11

1. Introduction

Modern hardware designs contain multiple modules and processes operating on the common set of internal variables. In this case *conflicts*, i.e. illegal accesses from different processes to the same data, may appear. Requirements on how to operate with modules and avoid conflicts in a communication protocol can be described both in *formal* (machine-readable) and *informal* (human-readable) ways.

In this paper, a formal verification based approach to conflict extraction is proposed. The idea is to analyze an HDL description aimed at finding *data access conflicts* [1]. Both the conflicts and the target description are then automatically translated into the input format of a *model checking* tool. The tool generates counterexamples for the feasible conflicts.

2. Related work

In [1] several categories of data conflicts are described: *read after write* (RAW), *write after read* (WAR) and *write after write* (WAW). The HOL verification system [2] was used to check a RISC processor's pipeline. The formal specification of pipeline was implemented manually that is hard to be done for modern processors because of their complexity.

In [3], a GoldMine methodology is presented for automatic generation of hardware assertions. The method uses a combination of data mining and static analysis techniques. First, the HDL design is simulated to generate data about the design's dynamic behavior. Then, the generated data are mined for "candidate assertions" that are likely to be *invariants*. The data mining technique used is a decision-tree-based supervised learning algorithm. The candidate assertions are then passed through the Cadence Incisive Formal Verifier [4] tool to filter out the spurious candidates. The disadvantages of GoldMine are: 1) usage of commercial tool; 2) invariants' incompleteness because of random simulation usage at an early stage.

3. Assertion extraction method

We propose a new approach to data access conflicts extraction in HDL descriptions. Our goal is to detect conflicts and provide proofs that they may happen. The method is aimed at conflicts of the following types:

- *read-write* (RW): on the same clock tick one process writes the variable and the other process reads it;
- *write-write* (WW): on the same clock tick at least two processes write the same variable;
- *write-read-write* (WRW): we assume that a variable should be read between two writes;
- *undefined* (UNDEF): variable is read before it was written.

```
if (state_bank3[inst_addr] > 1) // C1
    prediction[3] <= 1;     // B1
else
    prediction[3] = 0;      // B2
if (state_bank2[inst_addr] > 1) // C2
    prediction[2] <= 1;     // B3
else
    prediction[2] <= 0;     // B4
if (state_bank1[inst_addr] > 1) // C3
    prediction[1] <= 1;     // B5
else
    prediction[1] <= 0;     // B6
```

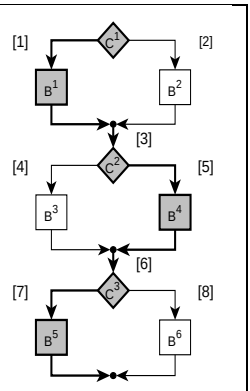

Fig. 1. Control Flow Graph Example

The method consists of the following steps: 1) *Control Flow Graph* (CFG) extraction; 2) transformation to *Guarded Actions Decision Diagram* (GADD); 3) process invariants and conflict assertions extraction; 4) invariants and assertions translation into an input format of a model checking tool; 5) counterexample generation. All method steps are made automatically. The CFG representation is built for every process of the HDL model using an abstract syntax tree traversal compiler-like approach [5]. From the structural view, CFG is a directed graph. Nodes of the graph contain HDL operators; edges of the graph mean control flows. On the left side of fig. 1 the fragment of Verilog code is shown; the related CFG is shown on the right side. Branch operators are shown as diamond nodes and called as C^i. Basic block operators are shown as rectangles and called as B^i. Graph edges contain the values that the branch conditions should be equal to for

edges to be passed. CFG is supposed to be acyclic: HDL loops with constant numbers of iteration are unrolled into sequences of operators.

The next step is the transformation of the CFG to a GADD that is a labeled DAG of guarded actions. A pair $\{\gamma, \delta\}$, where γ is a guard and δ is an action, is called a *guarded action* (GA) [6]. The main idea of the CFG-GADD transformation method is in extraction of branch-free sub-paths from the CFG. Every such sub-path (GA) contains a condition (*guard*) and a sequence of assignment operators (*action*). For action to be executed the guard should be satisfied. Actions are represented in the *static single assignment* (*SSA*) form [7]. To connect subsequent GAs into a complete CFG path an auxiliary *phase* variable is used.

To illustrate this step of the approach, let us take the previous example (see Fig. 1). The CFG model contains the following execution path: $C^1 \to B^1 \to C^2 \to B^4 \to C^3 \to B^5$. Path nodes are grey-colored in the fig. 1; path edges are highlighted too. The following GAs can be extracted from the path: $\{C^1, B^1\}$, $\{C^2, B^4\}$, $\{C^3, B^5\}$. Every GA corresponds to a unique value of the phase variable. The phase variable changes its value upon moving from one GA to another. On Fig. 1 related values of the phase variable are shown in brackets (the initial phase value is 0). Fig. 2 shows the example of GADD model from the previous example:

The main advantage of GADD model is path number reduction in comparison to CFG. In worst case (when CFG is a sequence of branch operators) the GADD has $O(n)$ paths, where n is the number of branches, but the CFG has $O(2^n)$.

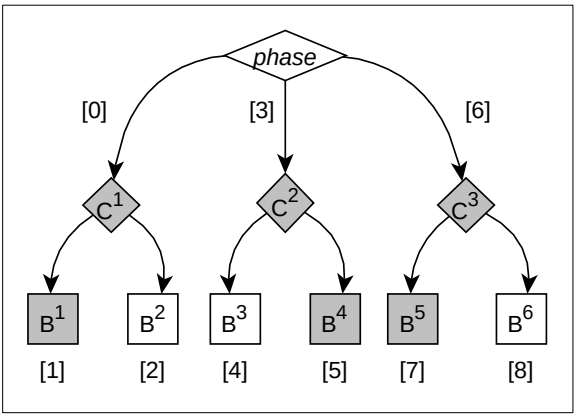

Fig. 2. GADD example

Then the GADD is transformed into the *invariants* of the processes, which represent the cycle-accurate behavior of the processes. The invariant is a logical formula and is a kind of a SSA representation of the whole process. Every GA of the GADD contains a unique phase variable value assignment. These unique values can be used as SSA version values of the variables. The phase variable is removed from the resulting formula because it does not affect the process behavior.

Each variable that is *defined* in a GA is labeled by the corresponding phase value. Each variable that is *used* in the GA is labeled by the set of phase values of the preceding GAs. For guards intermediate variables are introduced. To determine the values of the used variables, a backward search of the GADD is used: it is obvious that the variable value was defined in one of the preceding GAs or did not change from the previous cycle. After that, the process invariant formula is built as a conjunction of equality expressions representing each GA's guards and actions.

Let us see how a process invariant is built using a small example. Fig. 3 shows a part of the GADD and represents three guarded actions.

The guard conditions are: $z == a$, b and c respectively, and the actions contain definitions of variables x, $y$and unique definitions of $phase$. A set of the preceding phase values is $\{i, j\}$; z is a

variable; a, b and $c$are constants; f, g and h are functions defining the values of x and y; $V$is a set of process variables.

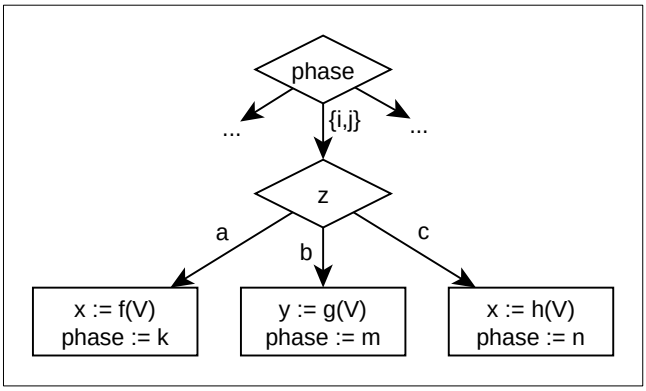

Fig. 3. Original part of the GADD

On the first step we label the variables by the corresponding phase values. The result of that is shown on fig. 4.

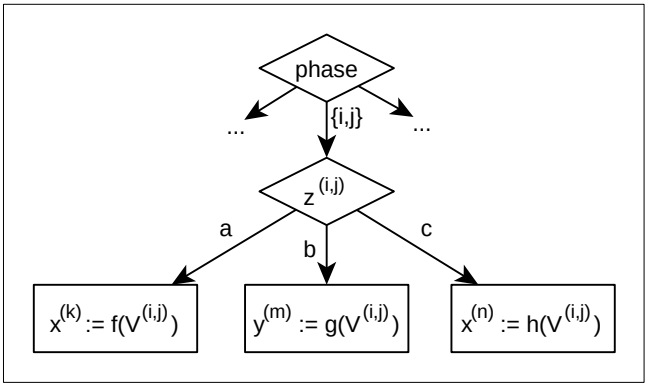

Fig. 4. GADD part with labeled variables

The used variables are now labeled by the preceding phase values $\{i, j\}$ and the defined variables are labeled by the corresponding phase values k, m, n. Phase definitions are removed.

Then we introduce and define a variable for each guard. The guard variable definition consists of a guard expression and a link to the preceding guards. This helps us restore the path from the beginning of the process to the corresponding guarded action. For example:

$$guard^{(k)} := (z^{(i,j)} == a)\ \&\ (guard^{(i)} \mid guard^{(j)})$$

When all the variables in all the GAs are labeled by phases, the remaining unknown used variables' values can be determined. Let us determine the value of $z^{(i,j)}$. So we traverse the GADD backward using the preceding phase values, starting from i and j (fig. 5). When a definition is found on some path (denoted def on fig. 5), the traversal of this path completes and the definition value is collected. If the beginning of the process is reached, the variable preserves its value from the previous cycle.

In the example on fig. 5 the variable z is defined on phases s and t or may not change its value. So the value of $z^{(i,j)}$ can be determined as follows:

$$z^{(i,j)} := guard^{(s)}\ ?\ z^{(s)} : guard^{(t)}\ ?\ z^{(t)} : z$$

On the final step the invariant formula is built. As it was mentioned, it is a conjunction of equality expressions for every labeled variable of the process including the guard variables:

$$x^{(k)} == f(V^{(i,j)}) \,\&\, y^{(m)} == g(V^{(i,j)}) \,\&\, x^{(n)} == h(V^{(i,j)})$$
$$\&\, guard^{(k)} == ((z^{(i,j)} == a)\ \&\ (guard^{(i)} \mid guard^{(j)}))$$
$$\&\ guard^{(m)} == ((z^{(i,j)} == b)\ \&\ (guard^{(i)} \mid guard^{(j)}))$$
$$\&\, guard^{(n)} == ((z^{(i,j)} == c)\ \&\ (guard^{(i)} \mid guard^{(j)}))$$
$$\&\, z^{(i,j)} == (guard^{(s)}\ ?\ z^{(s)} : guard^{(t)}\ ?\ z^{(t)} : z)\ \&\ \ldots$$

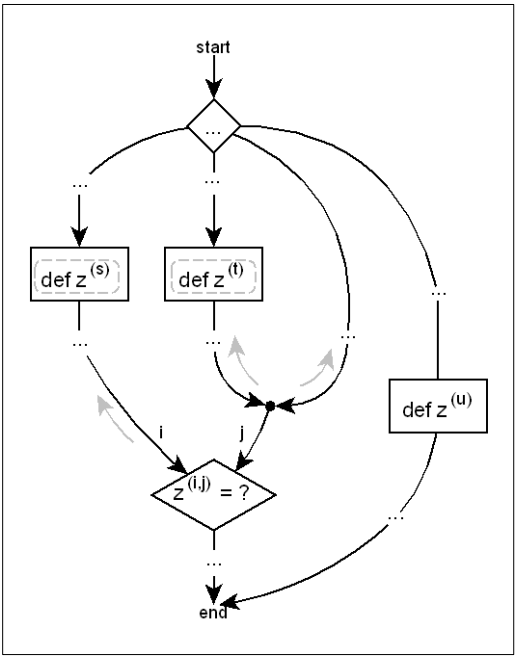

Fig. 5. Version value search (CFG view)

After the process invariant is built, the definition and usage conditions can be collected. They are collected only for internal and output variables of the HDL model, because input variables can be only used.

If a variable is *defined* (*used*) in the action of a GA, its definition (usage) condition equals the guard variable that corresponds to this GA. If a variable is used in the guard condition of a GA, its usage condition equals the disjunction of the corresponding guard variables of the preceding GAs. The variable definition (usage) condition of the whole process is the disjunction of the variable definition (usage) conditions of the GAs.

In our example, the definition conditions for variables $x$and y are:

$$def(x) = guard^{(k)} \mid guard^{(n)}$$
$$def(y) = guard^{(m)}$$

The usage condition for variable *z* is:

$$use(z) = guard^{(i)} \mid guard^{(j)}$$

Then the conditions are transformed into the assertions of conflict types described above. The assertions are represented as the *Linear-time Temporal Logic* (LTL) [8] formulas and state that the abovementioned conflicts never happen.

If, for example, a variable v is defined and used both in processes $p1$ and $p2$, the corresponding RW conditions are:

$$! \boldsymbol{F}\ (def_{p1}(v)\ \&use_{p2}(v))$$
$$! \boldsymbol{F}\ (def_{p2}(v)\ \&use_{p1}(v))$$

The corresponding WW condition:

$$! \boldsymbol{F}\ (def_{p1}(v)\ \&def_{p2}(v))$$

The corresponding WRW condition:

$$\boldsymbol{F}\ ((def_{p1}(v) \mid def_{p2}(v)) \ \&\ (\boldsymbol{F}\ (use_{p2}(v) \mid use_{p1}(v))\ \boldsymbol{U}\ (def_{p1}(v) \mid def_{p2}(v))))$$

The corresponding UNDEF condition:

$$\boldsymbol{G}\ (!\ (use_{p2}(v) \mid use_{p1}(v))\ \boldsymbol{U}\ (def_{p1}(v) \mid def_{p2}(v)))$$

Invariants and assertions are then translated into the SMV model. Their translation is rather straightforward. It is only important to define the variable value in the next state of the model using the keyword *next*. This value equals the last version of the variable before the end of the process. For example, if the final phase values of a process are k, l, m, then the next state value of a variable *v* is defined as:

$$next(v) := v^{(k,l,m)}$$

The SMV model is checked by the nuXmv[9] tool using bounded model checking. Ifan assertion is violated, a counterexample is generated and a potential conflict is found. The counterexamples may be later translated into test scenarios for the original HDL description.

4. Case study

The method was implemented in the Retrascope [10] tool. It was applied to arange of Verilog designs from the Texas-97 [11], VCEGAR [12] and Verilog2SMV VIS [13] benchmarks and the 16-bit MIPS processor [14].Table 1 contains the results of the method's application: benchmark descriptions and generated assertions amount. Here *N* means total amounts of top-level modules.Most of the assertions denote only suspicious situations, so the results should be analyzed by a verification engineer to filter out the real data conflicts.

Table 1. Benchmark descriptions and potential conflicts.

Bench	N	LOC	Assertions			
			RW	WW	WRW	UNDEF
Texas'97	17/58	69539	408	26	211	211
VCEGAR	20/34	15144	315	25	167	167
Verilog2SMV	12/20	4494	78	0	62	62
mips16	5/12	1007	10	0	9	9

Example of a RW situation, which is not a conflict (*mips16/ID_stage.v*):

```
  module ID_stage
  …
  wire  [2:0] ir_dest_with_bubble;
  wire  [2:0] write_back_dest;

  assign ir_dest_with_bubble = ( instruction_decode_en )
?
       ir_dest : 0;
assign write_back_dest = ir_dest_with_bubble;
```

Signal *ir_dest_with_bubble* is defined in one process and is used in the other process at the same time.

Example of a WW situation, which seems to be a real conflict (*Texas97/MPEG/prefixcode.v*):

```
module start_code_prefix(start, done…);
…
reg monitor;
…
always @(posedge read_signal) begin
 monitor=start;
…
end
always if( start==0) begin
…
 monitor=0;
end
```

Variable *monitor* is defined simultaneously, if *read_signal* rises and at the same time *start* equals 0.

Example of an UNDEF situation, which is also not a conflict (*mips16/ID_stage.v*):

```
module ID_stage


…
reg   [15:0]  instruction_reg;
…
always@(posedge clk or posedge rst) begin
 if (rst) begin
    instruction_reg<= 0;
 end
 else begin
    if (instruction_decode_en) begin
       instruction_reg <= instruction;
    end
 end
end
assign ir_op_code = instruction_reg[15:12];
```

Register *instruction_reg* is undefined from the start of simulation until the *clk* or *rst* rising edge.

5. Conclusion and future work

In this paper, the approach to data access conflicts extraction from HDL descriptions has been proposed. We extract assertions from the source code and automatically translate them into the input format of the model checker. The tool generates counterexamples that are proofs of conflicts' reachability. We have implemented the approach in the Retrascope toolkit and applied it to several open source HDL benchmarks.

One direction for future research is to propagate assertions from internal variables' to interface variables. Such assertions can be used to improve protocols of unknown third-party modules or even to reconstruct protocols. Another direction is the generation of *checkers*, i.e. HDL wrappers for target modules that check their behavior through simulation.

References

[1]. S.Tahar, R. Kumar. Formal Verification of Pipeline Conflicts in RISC Processors. In Proc. of the European Design Automation Conference (EURO-DAC), 1994, pp. 285-289.

[2]. M. Gordon,T. Melham. Introduction to HOL: A Theorem Proving Environment for Higher Order Logic, Cambridge University Press, 1993, 492 p.

[3]. S. Hertz, D. Sheridan, S. Vasudevan. Mining Hardware Assertions with Guidance From Static Analysis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 32, No. 6, 2013, pp. 952-965.
[4]. Cadence Incisive Formal Verifier. https://community.cadence.com//CSSharedFiles/forums/storage/22/10078/IncisiveFV_ds.pdf
[5]. A.V. Aho, R. Sethi, J.D. Ullman. Compilers: principles, techniques, and tools. Addison-Wesley, 1986, 796 p.
[6]. J. Brandt, M. Gemunde, K. Schneider, S. Shukla, J.-P. Talpin. Integrating system descriptions by clocked guarded actions. In Proc. of Forum on Specification and Design Languages (FDL), 2011, pp. 1-8.
[7]. R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems, vol. 13, issue 4, 1991, pp. 451-490.
[8]. A. Pnueli. The temporal logic of programs. In Proc. of the 18th Annual Symposium on Foundations of Computer Science, 1977, pp. 46-57.
[9]. nuXmv model checker. https://nuxmv.fbk.eu
[10]. Retrascope toolkit. https://forge.ispras.ru/projects/retrascope
[11]. Texas-97 Verification Benchmarks. https://ptolemy.berkeley.edu/projects/embedded/research/vis/texas-97
[12]. VCEGAR benchmark collection. https://www.cprover.org/hardware
[13]. Verilog2SMV tool. https://es-static.fbk.eu/tools/verilog2smv
[14]. Educational 16-bit MIPS Processor. https://opencores.org/projects/mips_16

Information about authors / Информация об авторах

Александр Сергеевич КАМКИН – ведущий научный сотрудник отдела технологий программирования Института системного программирования им. В.П. Иванникова Российской академии наук (ИСП РАН). Также он читает лекции в Московском государственном университете им. М.В. Ломоносова (МГУ), Московском физико-техническом институте (МФТИ) и Высшей школе экономики (НИУ ВШЭ). Кандидат физико-математических наук (2009). Область научных интересов: архитектура микропроцессоров, проектирование цифровой аппаратуры, верификация и тестирование цифровой аппаратуры, генерация тестовых программ для микропроцессоров, верификация подсистем управления памятью микропроцессоров, статический и динамический анализ HDL-описаний.

Alexander Sergeevich KAMKIN is a leading researcher at the Software Engineering Department of Ivannikov Institute for System programming of the Russian Academy of Sciences (ISP RAS). He is also a lecturer at Moscow State University (MSU), Moscow Institute of Physics and Technology (MIPT) and Higher School of Economics (HSE). His research interests include hardware design, functional verification, test program generation, and formal methods. He has a MS in computer science (MSU, 2003) and a PhD in computer science (ISP RAS, 2009). He is an expert of RAS and one of the organizers of Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE).

Михаил Сергеевич ЛЕБЕДЕВ – инженер 2й категории отдела технологий программирования ИСП РАН. Область научных интересов: архитектура микропроцессоров, проектирование цифровой аппаратуры, верификация и тестирование цифровой аппаратуры, статический и динамический анализ HDL-описаний.

Mikhail Sergeevich LEBEDEV is an engineer at the Software Engineering Department of ISP RAS. He has a MS in hardware engineering (MEPhI, 2011). His research interests include hardware design, functional verification, formal methods and static analysis.

Sergey Aleksandrovich SMOLOV is a junior researcher at the Software Engineering Department of ISP RAS. He has a MS in computer science (MIPT, 2010). His research interests include functional verification, formal methods and static analysis.

Сергей Александрович СМОЛОВ – младший научный сотрудник отдела технологий программирования ИСП РАН. Область научных интересов: функциональная верификация, формальные методы, статический анализ.

DOI: 10.15514/ISPRAS-2019-31(3)-12

Constructive heuristics for Capacitated Vehicle Routing Problem: a comparative study

S.M. Avdoshin, ORCID: 0000-0001-8473-8077 <savdoshin@hse.ru>
E.N. Beresneva, ORCID: 0000-0001-6710-2843 <eberesneva@hse.ru>
Department of Software Engineering,
National Research University Higher School of Economics,
20, Myasnitskaya st., Moscow, 101000 Russia

Abstract. Vehicle Routing Problem (VRP) is concerned with the optimal design of routes to be used by a fleet of vehicles to serve a set of customers. In this study we analyze constructive heuristics for a subcase of VRP, where the vehicles have a limited capacity – Capacitated Vehicle Routing Problem (CVRP). The problem is NP-hard, therefore heuristic algorithms which provide near-optimal polynomial-time solutions are still actual. The aim of this work is to make a comparison of constructive heuristics as there were not found any such classification. Finally, the leader by a criterion of quality is admitted being a Clarke and Wright Savings heuristic; however, this algorithm cannot find the solution for all used instances. This fact and other ones are discussed in the paper. Our future goal is to make an experimental comparison of the most common and state-of-the-art metaheuristics using well suited constructive heuristic to build a suboptimal solution.

Keywords: capacitated vehicle routing problem; classical heuristics; constructive heuristics

For citation: Avdoshin S.M., Beresneva E.N. Constructive heuristics for Capacitated Vehicle Routing Problem: a comparative study. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 3, 2019. pp. 145-156. DOI: 10.15514/ISPRAS-2019-31(3)-12

Эвристические методы конструирования маршрута для решения задачи маршрутизации с ограничением по грузоподъемности

С.М. Авдошин, ORCID: 0000-0001-8473-8077 <savdoshin@hse.ru>
Е.Н. Береснева, ORCID: 0000-0001-6710-2843 <eberesneva@hse.ru>
Департамент программной инженерии,
Национальный исследовательский университет "Высшая школа экономики", 101000,
Россия, г. Москва, ул. Мясницкая, д. 20

Аннотация. Задача маршрутизации – одна из широко известных задач комбинаторной оптимизации. Она состоит в отыскании оптимального множества маршрутов для транспортных средств с целью однократного обслуживания определенного множества клиентов. В данной работе исследуется подвид задачи маршрутизации – задача маршрутизации с ограничением по грузоподъемности, в которой каждое транспортное средство имеет свою грузоподъемность. Задача является NP-трудной, поэтому вместо точных алгоритмов решения исследуются только эвристические алгоритмы, позволяющие получить приближенные решения за полиномиальное время. Задача работы – провести экспериментальное исследование точности решения различных конструктивных эвристик, так как в других источниках не было найдено подобных сравнений. В большинстве случаев, лидером можно признать эвристику «Clarke and Wright Savings», однако существуют отдельные наборы данных, описанные в тексте, на которых лучше работают другие алгоритмы. Также в статье рассмотрены и другие интересные факты. В целом работа проделана с целью дальнейшего использования полученных знаний в экспериментальном исследовании наиболее известных и современных

метаэвристических алгоритмов решения задачи маршрутизации с ограничением по грузоподъемности, для которых будут получены предварительные решения на основе выявленных лучших эвристических методов конструирования маршрута.

Ключевые слова: задача маршрутизации с ограничением по грузоподъемности; эвристические методы конструирования маршрута

Для цитирования: Авдошин С.М., Береснева Е.Н. Эвристические методы конструирования маршрута для решения задачи маршрутизации с ограничением по грузоподъемности. Труды ИСП РАН, том 31, вып. 3, 2019 г., стр. 145-156 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(3)-12

1. Introduction

The Vehicle Routing Problem (VRP) is one of the most widely known questions in a class of combinatorial optimization problems. VRP is directly related to Logistics transportation problem and it is meant to be a generalization of the Travelling Salesman Problem (TSP). In contrast to TSP, VRP produces solutions containing some (usually, more than one) looped cycles, which are started and finished at the same point called «depot». The objective is to minimize the cost (time or distance) for all tours. For the identical type of input data, VRP has higher solving complexity than TSP. Both problems belong to the class of NP-hard tasks.

This work is aimed at analysis of VRP subcase, which is called Capacitated Vehicle Routing Problem (Capacitated VRP, CVRP), where the vehicles have a limited capacity. It means that there is a physical restriction on transportation more than determined amount of weight for each machine. Capacitated vehicle routing problems form the core of logistics planning and are hence of great practical and theoretical interest.

There are three types of algorithms that are used to solve any subcase of CVRP.

- *Exact algorithms.* These algorithms find an optimal solution but take a great time for solving large instances. Such methods include Branch-and-Bound, Branch-and-Cut, cutting plane, column generation, cut and solve, Branch-and-Cut-and-Price, Branch-and-Price, and dynamic programming techniques. It was shown in (Toth & Vigo, Branch-and-Bound algorithms for the capacitated VRP, 2002) that Branch-and-Bound algorithm was able to solve random CVRP instances with up to 300 customers and four vehicles within 1000 CPU seconds in 2002. However, according to the same source some real-world CVRP instances with up to 47 vertices only were successfully solved within 1000 CPU seconds. Current situation does not differ a lot. State-of-the-art exact methods can provide optimal solution for some SCVRP instances with up to 100 nodes, but it takes 30-40 minutes at average (Braekers, Ramaekers, & Nieuwenhuyse, 2016). Due to these restrictions, researchers all over the world concentrate on heuristic methods.
- *Classical heuristics.* These algorithms build an approximate solution iteratively, but they do not include further improvement stage. Different scientific works reveal that, in comparison to exact methods, classical heuristics work much faster. For example, an instance of 100-150 nodes can be solved up to a few (1-2) seconds (Braekers, Ramaekers, & Nieuwenhuyse, 2016). Heuristics are divided into two groups that include constructive heuristics and improvement heuristics.
- *Metaheuristics*. Such type of algorithms is also called a framework for building heuristics. According to (Golden, Raghavan, & Wasil, The vehicle routing problem: latest advances and new challenges, 2008), metaheuristics either explore the solution space by moving at each iteration from a solution to another solution in its neighbourhood (metaheuristics based on local search) or evolve a population of solutions which may be combined together in the hope of generating better ones (metaheuristics based on population, natural inspired).

Actuality of research and development of heuristics algorithms for solving VRP is on its top, because such approximate algorithms can produce near-optimal solutions in a polynomial time. It

is especially important in real-world tasks when there are more than one hundred clients in a delivery net. Among the best-known algorithms for CVRP there are metaheuristic proposed by Pisinger and Ropke (Pisinger & Ropke, 2007), Nagata and Braysy (Nagata & Braysy, 2009), and Vidal et al (Vidal, Crainic, Gendreau, Lahrichi, & Rei, 2012).

There are a lot of articles related to CVRP heuristics, but no works were found which compare solution quality, or gap, of classical heuristics using the same data bases. Solution quality is calculated as the percentage of difference in the obtained value of the solution with the optimal (or best-known) solution for the problem.

It is important to analyze classical heuristics since constructive heuristics are usually used in order to provide an initial (suboptimal) solution to improvement methods and to metaheuristics that allow to iteratively get near optimal solutions. So, we will discuss only algorithms from the first group.

The paper is structured as follows. In the second part a mathematical formulation of CVRP is given. In the third section, some notes on a classification of most popular constructive heuristics are provided, including description of chosen algorithms. The fourth part consists of design of experiments and their results. And, finally, in the fifth part conclusions and future goals are given.

2. Classical CVRP mathematical model

In the paper we will use CVRP abbreviation having in mind the mathematical formulation that was described in a previous work of authors (Beresneva & Avdoshin, 2018).

Let a complete weighted oriented graph $G =< V, V \times V >$ is given. Let $I = \{0, 1, \dots, N\}$, where $N + 1 = |V|$. Graph vertices are indexed as $ind = V \to I$, $(\forall v \in V)(\forall w \in V)\ v \neq w \Longrightarrow ind(v) \neq ind(w)$. Thus, $V = \{v_0, v_1, \dots, v_N\}$ is the set of vertices, here $i = ind(v_i), i \in I$. Let v_0 be the depot, where vehicles are located, and v_i be the destination points of a delivery, $i \neq 0$.

The distance between two vertices v_i and v_j is calculated using a distance function $c(v_i, v_j)$. Here a real-valued function $c(\cdot,\cdot)$ on $V \times V$ satisfies $\forall i, j, g \in I$ (Reed & Simon, 1972):

- $c(v_i, v_j) \geq 0$ (non-negativity axiom);
- $c(v_i, v_j) = 0$ if and only if $v_i = v_j$ (identity axiom);
- $c(v_i, v_j) = c(v_j, v_i)$ (symmetry axiom);
- $c(v_i, v_g) \leq c(v_i, v_j) + c(v_j, v_g)$ (triangle inequality axiom).

Each destination point v_i, $\forall i \in I$, is associated with a known nonnegative demand, d_i, to be delivered, and the depot has a fictitious demand $d_0 = 0$. The total demand of the set $V' \subseteq V$ is calculated as $d(V') = \sum_{i'=1}^{|V'|} d_{i'}$.

Let K be a number of available vehicles at the depot v_0. Each vehicle has the same capacity – C. Let us assume that every vehicle may perform at most one route and $K \geq K_{min}$, where K_{min} is a minimal number of vehicles needed to serve all the customers due to restriction on C. Clearly, next condition must be fulfilled – $(\forall v_i \in V)\ d_i \leq C, \forall i \in I$, which prohibits goods transportation that exceed maximum vehicle capacity.

Let introduce $V^0 = \{v_0\}$, where $v_0 \in V$. Let us divide V in $K + 1$ sets: $S = \{V^0, V^1, \dots, V^K\}$, each subset, except for V^0, represent a set of customers to be served for one vehicle. $S^{all} = \{S\}$ is a set of all possible partitions of V. Let $J = \{0, 1, \dots, K\}$ be a set that keeps indexes. Then $(\forall j \in J)\ |V^j| \geq 1$. There should be no duplicates in any of subsets from S: $(\forall g \in J)(\forall j \in J)(g \neq j \Rightarrow V^g \cap V^j = \emptyset)$. Also, all subsets from S must form set V. Thus, $V = \bigcup_{j=0}^{K} V^j$. In this notation, $V^{0k} = V^0 \cup V^k, \forall k \in J \backslash \{0\}$. It is obvious that $d(V^{ok}) \leq C$.

Let introduce $M^k = \{1, N^1 \dots, N^k\}$, $N^k = |V^k|$, $\sum_{k=1}^{K} N^k = N$. Then let $M^{0k} = \{0\} \cup M^k$. Let $I^k = \bigcup_{k=1}^{K} \{i \mid i = ind(v), \forall v \in V^k\}$ be a set of vertex indices from V^k. Then $I^{0k} = \{0\} \cup I^k, \forall k \in J \backslash \{0\}$.

Let $H^k = \{p^k : M^{0k} \to I^{0k} \mid p^k(0) = 0 \;\&\; (\forall x \in M^{0k})(\forall y \in M^{0k})\; x \neq y \implies p^k(x) \neq p^k(y)\}$ be a set of codes of all possible permutations $h^k = \left(v_{p^k(0)}, v_{p^k(1)}, \dots, v_{p^k(N^k)}\right)$ of V^{0k}. These permutations represent all possible Hamiltonian cycles of graph $G^{0k} < V^{(0k)}, V^{(0k)} \times V^{(0k)} >$, $\forall k \in J \backslash \{0\}$.

Weight of $h^k \in H^k$ can be found according to the formula 1:

$$f(h^k) = c\left(v_{p^k(0)}, v_{p^k(N^k)}\right) + \sum_{q=0}^{N^k-1} c\left(v_{p^k(q)}, v_{p^k(q+1)}\right) \qquad (1)$$

Let S' be a set of $\{V^{01}, V^{02}, \dots, V^{0K}\}$. In this notation the weight of S' is calculated as $F(S') = \sum_{k=\overline{1..K}} f(h^k)$, $\forall k \in J \backslash \{0\}$.

Overall, the formulation of CVRP is to find:

$$S^0 : F(S^0) = \min_{S \in S^{all}} F(S) \qquad (2)$$

3. Constructive heuristics

In this section the most popular constructive heuristics are described.

3.1 Sequential Insertion algorithm (SI)

Sequential Insertion algorithm (Laporte & Demet, Classical heuristics for the Capacitated VRP, 2002) constructs routes subsequently, one after another.

In the first step, a new tour $tour_k, k \leq K$, is initialized with a random unrouted node $v_i, i \neq 0$, and the depot v_0. Thus, a tour (v_0, v_i, v_0) is obtained.

In the second step, another unrouted vertex $v_j, j \neq 0$, is chosen, such that its incorporation in the current tour gives the least increase in a tour length and demand of a potential node v_j does not exceed vehicle capacity. So, the next two formulae must be hold:

- $\operatorname{argmin}_{\substack{v_a \in tour_k, \\ v_{a+1} \in tour_k, \\ v_j \notin tour_k}} c(v_a, v_j) + c(v_j, v_{a+1}) - c(v_a, v_{a+1})$;
- $D_{tour_k} + d_j \leq C$, where D_{tour_k} is a total demand of current $tour_k$.

If all conditions hold then this unrouted vertex $v_j, j \neq 0$ is inserted in a tour $tour_k$ between v_a and v_{a+1}.

The second step is repeated until no more unrouted vertex $v_j, j \neq 0$, can be feasibly inserted. In this case a new tour $tour_k, k \leq K$, is initialized, and the procedure starts from the first step.

3.2 Improved Parallel Insertion algorithm

Parallel Insertion Improved algorithm (Laporte, Nobert, & Desrochers, Optimal routing under capacity and distance restrictions, 1985) builds routes simultaneously. This method is a modification of Sequential Insertion algorithm.

In the first step, the minimum number K_{min} of feasible routes is defined as $K_{min} = \sum_{i \in |V|} d_i / C$. All these routes $tour_k \in Tours$ are initialized with K_{min} different closest to v_0 unrouted nodes $v_i, i \neq 0$. Thus, K_{min} tours (v_0, v_i, v_0) are obtained.

In the second step, a random unrouted node $v_j, j \neq 0$, is inserted in some route $tour_k$ at its best insertion position. The next two conditions must be hold – incorporation of v_j in this tour gives the

least increase in a tour length among all other tours and demand of a potential node v_j does not exceed vehicle capacity. So:

- $\operatorname{argmin}_{\substack{v_a \in tour_k, \\ v_{a+1} \in tour_k, \\ v_j \notin tour_k}} c(v_a, v_j) + c(v_j, v_{a+1}) - c(v_a, v_{a+1})$;
- $D_{tour_k} + d_j \leq C$, where D_{tour_k} is a total demand of current $tour_k$.

If all conditions hold then this unrouted vertex $v_j, j \neq 0$ is inserted in a tour $tour_k$ between v_a and v_{a+1}.

The second step is repeated until no more unrouted vertex $v_j, j \neq 0$, can be feasibly inserted in some route $tour_k$. In this case a new tour $tour_k, k \leq K$, is initialized as (v_0, v_j, v_0) and adds to set of all tours $Tours$, and the procedure continues.

3.3 Nearest Neighbor heuristic (NN)

Nearest Neighbor heuristic constructs routes subsequently, one after another, in a greedy way.

In the first step, an unrouted node $v_i, i \neq 0$, which is closest to the depot v_0, is chosen. A new open tour $tour_k, k \leq K$, is initialized with v_i and v_0. Thus, a tour (v_0, v_i) is obtained.

In the second step, another unrouted vertex $v_j, j \neq 0$, is chosen, which is the nearest to the last added vertex and a demand of a potential node v_j does not exceed vehicle capacity. So, the next two formulae must be hold:

- $\operatorname{argmin}_{v_i \in tour_k,\ v_j \notin tour_k} c(v_i, v_j)$;
- $D_{tour_k} + d_j \leq C$, where D_{tour_k} is a total demand of current $tour_k$.

If all conditions hold then this unrouted vertex v_j is added in the end of $tour_k$ after v_i, and since that time it turns to be the last added vertex.

The second step is repeated until no more unrouted vertex $v_j, j \neq 0$, can be feasibly inserted. In this case a new tour $tour_k, k \leq K$, is initialized, and the procedure starts from the first step.

3.4 Clarke and Wright Savings heuristic (CWS)

In the first step, all vertices $v_i \in V, i \neq 0$, must form $|V - 1|$ routes. Thus, $|V - 1|$ tours (v_0, v_i, v_0) are obtained.

In the second step, $\forall v_i \in V, \forall v_j \in V, i \neq 0, j \neq 0, i \neq j$, saving $s(v_i, v_j)$ is calculated as $s(v_i, v_j) = c(v_i, v_0) + c(v_0, v_j) - c(v_i, v_j)$. All savings are put in a list of $\bar{S}$, $\bar{S}$ must be sorted in a non-increasing order.

In the third step, the first unused saving in a list is taken. Then, existence of two routes $tour_x$ and $tour_y, x \neq y$, having the next conditions, is checked:

- there is an edge (v_i, v_0) in route x and edge (v_0, v_j) in tour $tour_y$;
- $D_{tour_x} + D_{tour_y} \leq C$.

If there are such routes then $tour_x$ and $tour_y$ are combined by removing edges (v_i, v_0), (v_0, v_j) and introducing edge (v_i, v_j). After that, despite of ability or absence these routes, the current saving is skipped and the next one in the list is checked.

The last step works until K tours are left.

3.5 Variant of Clarke and Wright Savings heuristic (CWS_2)

Classical variant of Clarke and Wright Savings algorithm forms good tours in the first part of its work mostly. However, it was noticed that it tends to produce less competitive tours towards the

end because of periphery nodes addition. Thus, Yellow (Yellow, 1970) and Gaskell (Gaskell, 1967) suggested improved form of savings calculation. It is $s(v_i, v_j) = c(v_i, v_0) + c(v_0, v_j) - \lambda c(v_i, v_j)$. Here λ is a parameter which responds for measuring the distance between the vertices to be joint. In one report (Golden, Magnanti, & Nguyen, Implementing vehicle routing algorithms, 1977) it was mentioned that the best value of λ is 0.4.

3.6 Subgroup of Cluster-First-Route-Second heuristics

Subgroup of Cluster-First-Route-Second heuristics belongs to two-phase methods, which are based on the decomposition of the CVRP solution process into two separate stages – clustering and routing. In the clustering stage, a partition of the customers into routes is made, and in the routing stage, the sequence of the customers on each subset is obtained.

In Cluster-First-Route-Second methods, nodes are first partitioned into different subsets called clusters and then routes are determined by sequencing the customers within each subset.

3.6.1 Sweep

This Cluster-First-Route-Second method can be applied only for planar instances (Laporte & Demet, Classical heuristics for the Capacitated VRP, 2002).

Clustering stage

Let us define $v_i \in V$ as $v_i = (x_i; y_i)$, where x_i and y_i are the Cartesian coordinates of point v_i.

In the first step, new normalized vertices $v_i' = (x_i'; y_i') = (x_i - x_0; y_i - y_0)$ are introduced, where the depot $v_0{}'$ has new Cartesian coordinates (0; 0), $\forall i \in |V|$.

Let $\overline{v_i'} = (\theta_i, r_i)$ be a vertex with polar coordinate of $v_i{}'$, where $r_i = x_i'^2 + y_i'^2$ and θ_i is calculated using formula 3:

$$\theta_i = \begin{cases} arctg\left(\frac{y_i}{x_i}\right), x_i > 0, y_i \geq 0 \\ arctg\left(\frac{y_i}{x_i}\right) + 2\pi, x_i > 0, y_i < 0 \\ arctg\left(\frac{y_i}{x_i}\right) + \pi, x_i < 0 \\ \frac{\pi}{2}, x_i = 0, y_i > 0 \\ \frac{3\pi}{2}, x_i = 0, y_i < 0 \end{cases} \quad (3)$$

In the second step, a list $\bar{V}$ of all $\overline{v_i'} = (\theta_i, r_i), \forall i \in |V|, i \neq 0$, is calculated and is sorted in increasing order by parameter θ_i.

In the next step, a new cluster is initialized with $\{v_0\}$ and maximum number of first L vertices from $\bar{V}$, such that $\sum_{i=0}^{L-1} d_i \leq C$. Parameter L is not a constant, it can be other for different clusters depending on weights of demands and total capacity. Then these used vertices are removed from $\bar{V}$, and the procedure is repeated until $\bar{V} = \emptyset$.

Routing stage

At this stage for each cluster TSP Cheapest Insertion heuristic is applied which forms a cycle.

3.6.2 Fisher and Jaikumar algorithm

In contrast to Sweep algorithm, this Cluster-First-Route-Second method can be applied not only for planar instances. Instead of using a geometric method to form the clusters, it solves a Generalized Assignment Problem (GAP).

Clustering stage

In the first step $\forall k = \overline{1..K}$ a vertex $v_{seed(k)} \in V \backslash \{v_o\}$ is chosen. These K vertices form K clusters. In the second step the cost $cost_{v_i}^k$ of allocating each node $v_i \in V, i \neq 0$, to each cluster k is calculated as $cost_{v_i}^k = c(v_o, v_i) + c\left(v_i, v_{seed(k)}\right) - c\left(v_{seed(k)}, v_0\right)$.

In the third step the algorithm solves GAP with $cost_{v_i}^k$, d_i and C, which determines a minimum cost assignment of items to a given set of bins of capacity C. The GAP can be solved using either exact or heuristic techniques.

Routing stage

The final routes are determined by solving a TSP on each defined cluster.

According to this work (Sultana, Akhand, & Rahman, 2017), this algorithm gives way to the algorithms described above and provides solutions with more solution quality. That is why it will not be considered in later comparison study as it was already done.

3.7 Subgroup of Route-First-Cluster-Second heuristics

Subgroup of Route-First-Cluster-Second heuristics also belongs to two-phase methods. However, in contrast to Cluster-First-Route-Second methods, these constructive heuristics at first solve TSP for all nodes and only then break built cycle to K routes. Unfortunately, many studies showed than these heuristics are applicable only if there is no constraint on the number of vehicles. In addition, they are not competitive with other constructive heuristics in general (Laporte & Demet, Classical heuristics for the Capacitated VRP, 2002).

4. Experiments and results

All algorithms are implemented as sequential algorithms in C++. The computational testing of the solution methods for CVRP has been carried out by considering eight sets of test instances from the next well-known database (Xavier). Total number of instances in sets A, B, E, F, G, M, P, X is 211. All instances inside one set have its own characteristics and a way of generation: cluster-based / uniform / geometric distribution of clients, real-world / imitative cases etc. The integer Euclidean metric is used for all instances. The naming scheme and data format for each instance is described here (Heidelberg University). Shortly, the first letter in names shows the name of used set, the figure after letter ‘n’ shows the number of nodes and the figure which stands after letter ‘k’ presents the number of vehicles.

Experiment starts with the choice of a constructive heuristic H from the set {SI, FI, NN, CWS, CWS_2, Sweep}. After that one dataset D is selected from the list of all benchmark datasets. Then an instance file F from the chosen dataset D is taken as input for the algorithm H and the heuristic is executed (only 1 time because all these algorithms do not use random generations, so all obtained solutions are the same). After that we report solution quality $\varepsilon(\mathrm{H}, \mathrm{F})$ found for the algorithm H on the test F. Solution quality ε (or percent above best-known, or gap) is calculated using formula 4 (Toth & Vigo, An overview of vehicle routing problems, 2002):

$$\frac{F(S^0) - F_{opt}(S)}{F_{opt}(S)} \cdot 100\%, \qquad (4)$$

where $F(S^0)$ is a length of obtained solution and $F_{opt}(S)$ is a length of optimal solution or best-known one. And finally, among all $\varepsilon(\mathrm{H}, \mathrm{F})$ from one dataset sample mean $\bar{X}_\varepsilon(\mathrm{H}, \mathrm{D}) = \frac{1}{|D|}\sum_{F=1}^{|D|} \varepsilon(\mathrm{H}, \mathrm{F})$ is calculated which shows average gap for the algorithm H on the dataset D, where $|D|$ is a number of input files in dataset D.

The plan of experiments on constructive heuristics described in Fig. 1.

```
Input: constructive heuristics, datasets
       1:     foreach constructive heuristic H
       2:     foreach dataset D from datasets
```

```
3:      foreach instance file F from D
4:      solution = run H on F
5:      calculate ε(H, F)
6:      calculate X̄_ε(H, D) // average gap on dataset
```

Fig.1. Plan of experiment on constructive heuristics

It should be mentioned that each algorithm is subsequently launched on all 211 instances from 8 datasets, so no input file is missed.

A criterion of running time was not considered because all instances were solved in a time which does not exceed 1 second. It is thought to be insignificant in comparison with time-consuming metaheuristic work.

Fig. 2, 3 and 4 represent the results of experiments conducted over algorithms using sets B, P and G of widely different types. The horizontal axis represents the name of instance data. The vertical axis shows the solution quality.

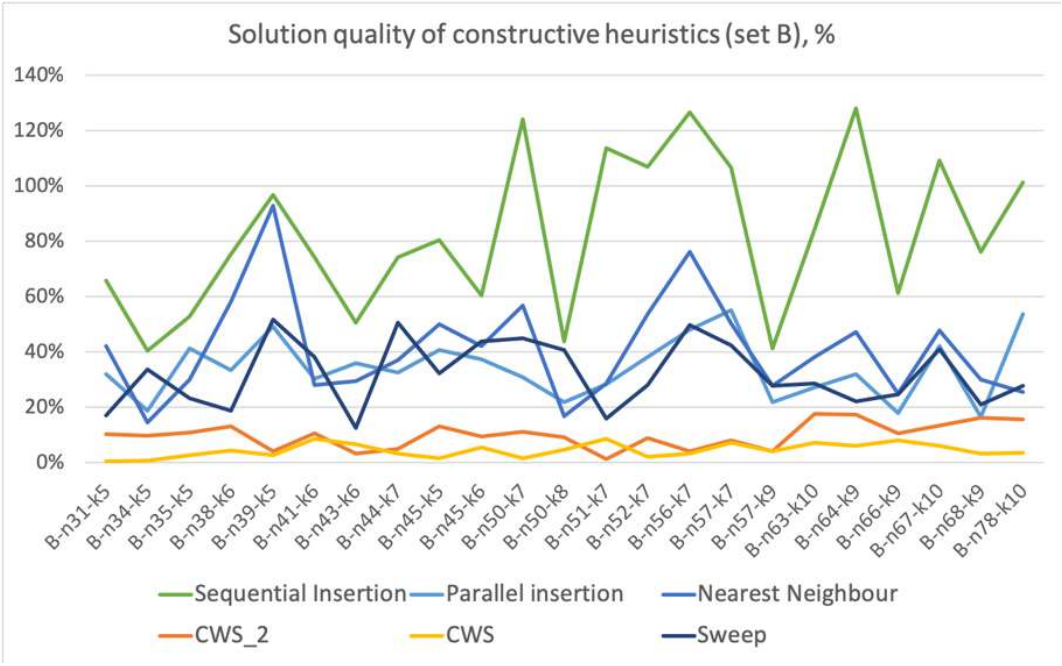

Fig. 2. Solution quality of constructive heuristics, set B

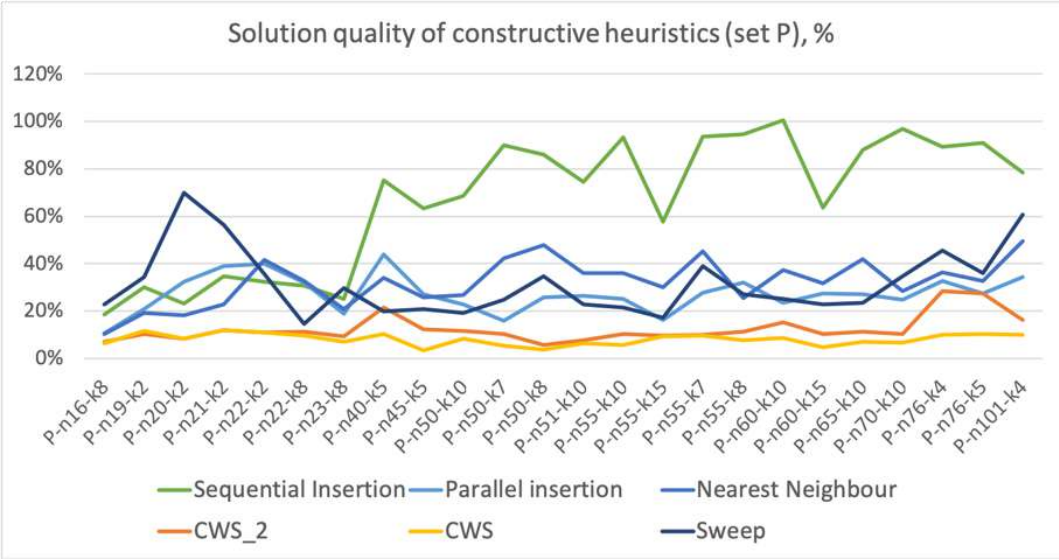

Fig. 3. Solution quality of constructive heuristics, set P

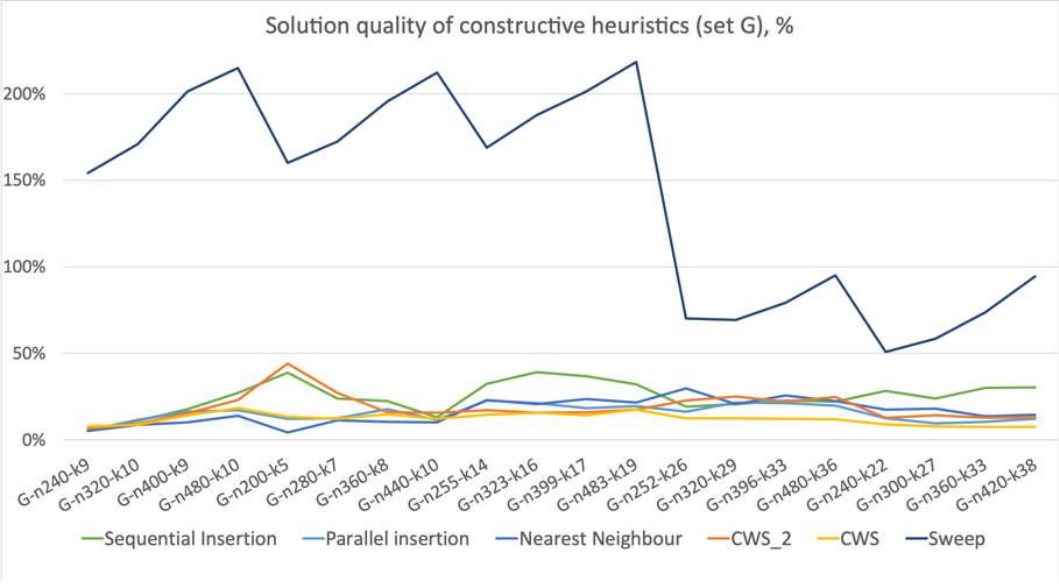

Fig. 4. Solution quality of constructive heuristics, set G.

Average gaps $\bar{X}_{\varepsilon}(\mathrm{H}, \mathrm{D})$ of each algorithm on different data sets are presented in Table 1 and fig. 5. These general figures can show an approximate overall effectiveness of algorithms. On the basis of Table 1, all fig. 2, 3, 4, 5 and other results which cannot be shown here because of their large volume, it can be easily seen that CWS algorithm (its column is made bold in the table) is a leader for all input files, except some instances from dataset G. Its average gap varies from 3,4% till 11,0%. The closest competitor is its variant CWS_2, which has average solution quality in a range [9,8%; 20,6%]. CWS_2 algorithm is able to construct the best solutions only for some instances in set B. In all other cases this algorithm nearly always takes second place and goes behind classical CWS.

Table 1. Average gaps of all heuristics for every set, %.

Average gap $\bar{X}_\varepsilon(H, D)$ in the dataset		Constructive heuristic					
		SI	PI	NN	CWS	CWS_2	Sweep
Set (its size)	A (26)	68,7%	33,2%	39,7%	**5,0%**	12,7%	40,2%
	B (23)	82,3%	34,0%	41,2%	**4,3%**	9,8%	31,7%
	E (11)	70,4%	30,0%	41,5%	**6,4%**	17,5%	36,4%
	F (3)	42,5%	48,5%	74,6%	**4,4%**	20,6%	71,9%
	G (20)	24,8%	15,6%	16,3%	**11,0%**	18,4%	142,4%
	M (4)	83,0%	35,5%	44,6%	**3,4%**	12,0%	89,2%
	P (24)	66,0%	25,6%	32,2%	**6,9%**	11,3%	31,4%
	X (100)	99,7%	23,3%	27,4%	**5,9%**	11,9%	82,9%

Table 2. Percentage of unsolved instances for every set, %.

Percentage of unsolved instances in the set		Constructive heuristic					
		SI	PI	NN	CWS	CWS_2	Sweep
Set (its size)	A (26)	0%	0%	0%	0%	0%	**69,0%**
	B (23)	0%	0%	0%	0%	0%	**50,0%**
	E (11)	0%	0%	0%	0%	0%	**43,5%**
	F (3)	0%	0%	0%	0%	0%	**63,6%**
	G (20)	0%	0%	0%	0%	0%	**66,7%**
	M (4)	0%	0%	0%	0%	0%	**90,0%**
	P (24)	0%	0%	0%	0%	0%	**50,0%**
	X (100)	0%	0%	0%	0%	0%	**58,3%**

There is only one algorithm that have a problem with finding an answer to the given problems – it is Sweep. This heuristic is not able to construct a set of routes without exceeding the number of vehicles for some input files. All the others coped with the task – they are NN, SI, PI, CWS and CWS_2. Table 2 shows the percentage and the number of unsolved instances for all sets. In average, Sweep algorithm cannot solve the instance without over limit in more than 50% cases. It can be explained by the fact that the next vertex to be added is chosen by criteria of distance (polar angle, for real) but not the capacity.

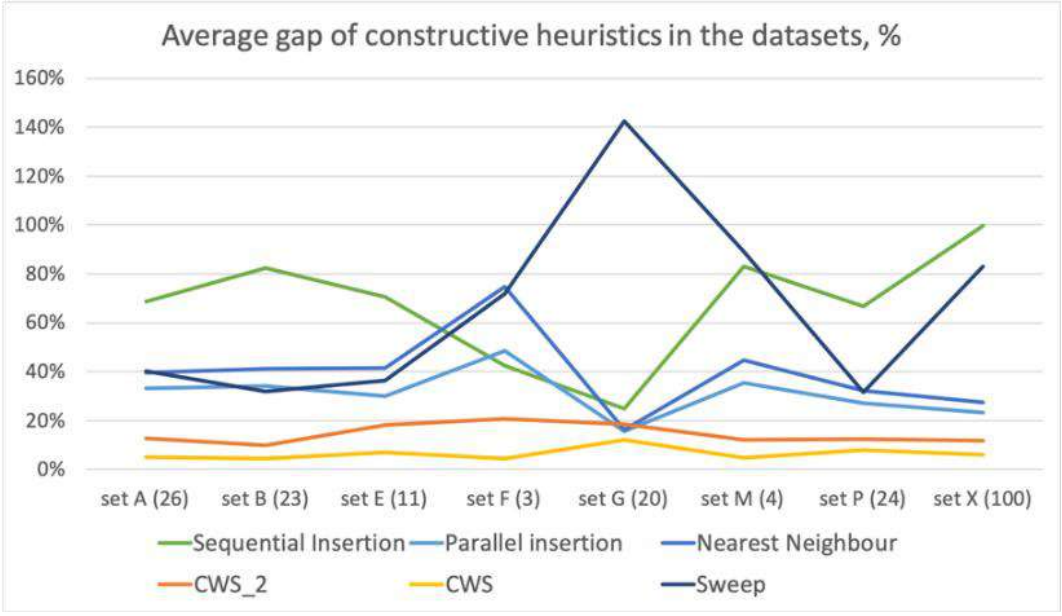

Fig. 5. Average gap of constructive heuristics in the datasets

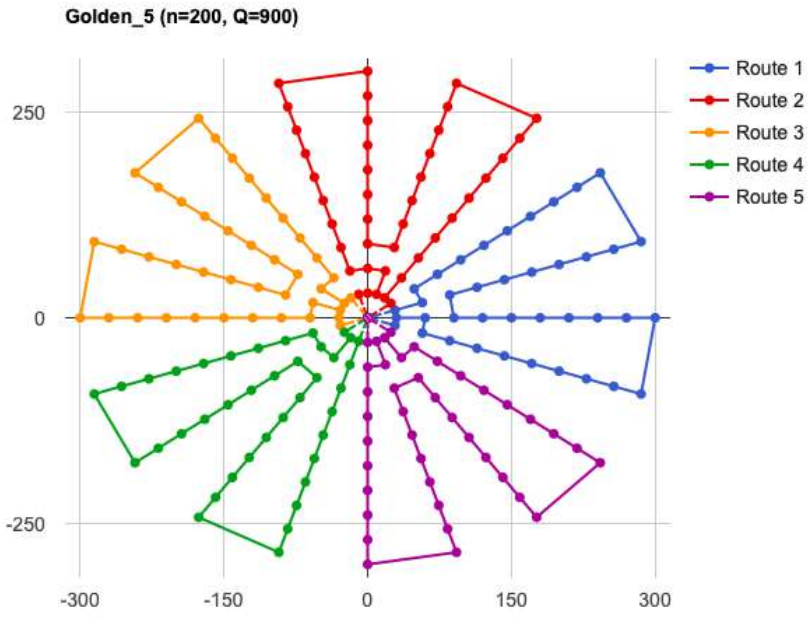

Fig. 6. Solution for instance G-n200-k5

It was mentioned earlier that CWS is not a leader for some instances from dataset G. There are 8 instances when NN finds the best solutions but not CWS (fig. 4). This interesting change of the leader is connected with the type of customers' distribution – these instances have a form of rays going from the center. If we look at fig. 6, where a solution for the instance is presented, we can see that the idea of nearest neighbor works here the best way.

5. Conclusions

Overall, the next recommendation should be given to the problem which has described variant of mathematical model of CVRP. In general, for all types of clients' distribution the best algorithm to be applied is Clarke and Wright Savings, however, in case of having input data in form of concentric rays (like in fig. 6) it is better to use Nearest Neighbor algorithm. Also, a few instances were solved best of all by Clarke and Wright Savings 2 algorithm, so it is important to have this algorithm in mind, however the difference between it and CWS is not very significant (no more than 1%). One more conclusion is that it is unreasonable to use Sweep heuristic as it is not able to construct a set of routes without exceeding the number of vehicles for more than 50% of input files. Finally, for our research it means that for all instances, except those 8 from set G, CWS heuristic will be used as initial algorithm for metaheuristic, otherwise – we will apply NN.

References

[1] P. Toth and D. Vigo, "Branch-and-Bound algorithms for the capacitated VRP," in The Vehicle Routing Problem, Philadelphia, SIAM, 2002, pp. 29-51.

[2] K. Braekers, K. Ramaekers, and I. Nieuwenhuyse. The vehicle routing problem: State of the art classification and review. Computers & Industrial Engineering, vol. 99, 2016, pp. 300-313.

[3] B. Golden, S. Raghavan and E. Wasil. The vehicle routing problem: latest advances and new challenges. New York: Springer, 2008.

[4] P. Pisinger and S. Ropke. A general heuristic for vehicle routing problems. Computers & Operations Research, vol. 34, no. 8, 2007, pp. 2403-2435.

[5] Y. Nagata and O. Braysy. Edge assembly-based memetic algorithm for the capacitated vehicle routing problem. Networks, vol. 54, no. 4, 2009, pp. 205-215.
[6] T. Vidal, T. Crainic, M. Gendreau, N. Lahrichi, and W. Rei. A hybrid genetic algorithm for multi-depot and periodic vehicle routing problems. Operations Research, vol. 60, no. 3, 2012, pp. 611-624.
[7] E. Beresneva and S. Avdoshin. Analysis of mathematical formulations of Capacitated Vehicle Routing Problem and methods for their solution. Trudy ISP RAN/Proc. ISP RAS, vol. 30, no. 3, 2018, pp. 233-250. DOI: 10.15514/ISPRAS-2018-30(3)-17.
[8] M. Reed and B. Simon. Methods of modern mathematical physics. London: Academic Press, 1972.
[9] G. Laporte and F. Demet. Classical heuristics for the Capacitated VRP. In The Vehicle Routing Problem, SIAM, 2002, pp. 109-128.
[10] G. Laporte, Y. Nobert, and M. Desrochers. Optimal routing under capacity and distance restrictions. Operations Research, vol. 33, no. 5, 1985, pp. 1050–1073.
[11] P. Yellow. A computational modification to the savings method of vehicle scheduling, Operational Research Quarterly, no. 21, 1970, pp. 281-283.
[12] T. Gaskell. Bases for vehicle fleet scheduling. Operational Research Quarterly, no. 18, 1967, pp. 281-295.
[13] B. Golden, T. Magnanti, and H. Nguyen. Implementing vehicle routing algorithms. Networks, no. 7, 1977, pp. 113-148.
[14] M. L. Fisher and R. Jaikumar. A generalized assignment heuristic for vehicle routing. Networks, vol. 11, no. 3, 1981, pp. 109-124.
[15] T. Sultana, M. Akhand and M. Rahman. A variant Fisher and Jaikuamr algorithm to solve capacitated vehicle routing problem. In Proc. of the 8th International Conference on Information Technology (ICIT), 2017, pp. 710-716.
[16] I. Xavier. CVRPLIB. [Online]. Available: http://vrp.atd-lab.inf.puc-rio.br/index.php/en/. [Accessed 09 07 2019].
[17] Heidelberg University. TSPLIB. [Online]. Available: https://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/. [Accessed 09 07 2019].
[18] P. Toth and D. Vigo. An overview of vehicle routing problems. In The Vehicle Routing Problem, SIAM, 2002.

Информация об авторах / Information about authors

Екатерина Николаевна БЕРЕСНЕВА – с 2017 года преподаватель департамента программной инженерии НИУ ВШЭ, с 2019 года – аспирант НИУ ВШЭ. Профессиональные интересы – дискретная математика, задача маршрутизации транспорта, задача коммивояжера.

Ekaterina BERESNEVA – lecturer at School of Software Engineering, Faculty of Computer Science, National Research University Higher School of Economics since 2017. Her research interests include discrete mathematics, the vehicle routing problem and the travelling salesman problem.

Сергей Михайлович АВДОШИН – профессор, руководитель департамента программной инженерии факультета компьютерных наук НИУ ВШЭ с 2005 года. Сфера научных интересов: разработка и анализ компьютерных алгоритмов, имитация и моделирование, параллельные и распределенные процессы, теневой интернет, технология блокчейн.

Sergey AVDOSHIN – Professor, Head of School of Software Engineering in National Research University Higher School of Economics since 2005. Research interests are design and analysis of computer algorithms, simulation and modeling, parallel and distributed processing, deep Web, blockchain technology.

DOI: 10.15514/ISPRAS-2019-31(3)-13

Overview of the Languages for Safe Smart Contract Programming

[1] *A.V. Tyurin, ORCID: 0000-0003-4820-3678 <a.tyurin@2016.spbu.ru>*
[1] *I.V. Tyulyandin, ORCID: 0000-0002-8429-8726 <i.tyulyandin@2015.spbu.ru>*
[1] *V.S. Maltsev, ORCID: 0000-0002-4948-3248 <v.maltsev@2016.spbu.ru>*
[1] *I.A. Kirilenko, ORCID: 0000-0003-4957-1974 <y.kirilenko@spbu.ru>*
[2] *D.A. Berezun, ORCID: 0000-0001-6306-275X <danya.berezun@gmail.com>*
[1] *Saint Petersburg State University, Mathematics and Mechanics Faculty*
7, University Embankment, Saint Petersburg, 199034, Russia
[2] *Higher School of Economics National Research University, Department of Computer Science*
16 Soyuza Pechatnikov Street, Saint Petersburg, 190121, Russia

Abstract. Blockchain technologies are gradually being found an application in many areas, especially in FinTech. As a result, a lot of blockchain platforms have emerged with the support of smart contracts that are intended to automate party interactions. However, it has been shown that they are prone to attacks and errors which lead to money loss. To date, there has been a wide range of approaches for making smart contracts safer that included analysis tools, reasoning models, and safer and more rigorous programming languages. In this paper, we provide an overview of smart contract programming languages design principles, related vulnerabilities, and future research areas. The provided overview is meant to outline the to date state of languages and to become a possible basis for future proceedings, and show approaches, used by the community, to reach safe and usable language for smart contracts. We have split all found vulnerabilities by source of their arising. Various languages' characteristics such as abstraction level, paradigm, Turing completeness and main features are summarized in the table. Additional information about languages is provided, e.g. model of execution and tools for static analysis.

Keywords: blockchain; smart contracts safety; programming languages

For citation: Tyurin A.V., Tyuluandin I.V., Maltsev V.S., Kirilenko I.A., Berezun D.A. Overview of the Languages for Safe Smart Contract Programming. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 3, 2019. pp. 157-176. DOI: 10.15514/ISPRAS-2019-31(3)-13

Обзор языков для безопасного программирования смарт-контрактов

[1] *А.В. Тюрин, ORCID: 0000-0003-4820-3678 <a.tyurin@2016.spbu.ru>*
[1] *И.В. Тюляндин, ORCID: 0000-0002-8429-8726 <i.tyulyandin@2015.spbu.ru>*
[1] *В.С. Мальцев, ORCID: 0000-0002-4948-3248 <v.maltsev@2016.spbu.ru,>*
[1] *Я.А. Кириленко, ORCID: 0000-0003-4957-1974 <y.kirilenko@spbu.ru,>*
[2] *Д.А. Березун, ORCID: 0000-0001-6306-275X <danya.berezun@gmail.com>*
[1] *Санкт-Петербургский государственный Университет, математико-механический факультет*
199034, Россия, г. Санкт-Петербург, Университетская Набережная, д. 7
[2] *Национальный исследовательский университет «Высшая школа экономики», Департамент информатики*
190121, Россия, г. Санкт-Петербург, ул. Союза Печатников, д.16

Аннотация. Технология распределенного реестра блокчейн становится все более популярной и находит применение в различных областях, в том числе и в финансовых технологиях. Многие блокчейн-платформы поддерживают функциональность смарт-контрактов, которые предназначены для автоматизации заключения договоров. Известны примеры, где ошибки или небрежности в коде смарт-контракта приводят к потере активов, например, из-за атаки злоумышленника или непонимания разработчиком особенностей блокчейн-платформы. На сегодняшний день существует множество различных подходов, которые позволяют сделать смарт-контракты безопаснее. Среди них инструменты анализа кода, модели вычислений и семантики языков программирования смарт-контрактов. В этой работе мы приводим обзор языков программирования смарт-контрактов, принципов их построения, а также потенциальные ошибки в программе смарт-контракта. Основная цель этого обзора — рассмотреть текущее на момент написания статьи состояние языков смарт-контрактов и возможные направления для будущих исследований, а также показать подходы, используемые сообществом для создания безопасного и удобного (с точки зрения абстракции) языка. Характеристики множества языков, такие как: уровень абстракции, парадигма, Тьюринг-полнота, проект, где язык используется, инструменты для анализа кода, система ограничения и главные особенности — были рассмотрены и сведены в таблицу. Предоставлена дополнительная информация о языках, например, о модели выполнения. Также мы кратко описали и разделили все найденные нами уязвимости по источникам их возникновения.

Ключевые слова: блокчейн; безопасность смарт-контрактов; языки программирования

Для цитирования: Тюрин А.В., Тюляндин И.В., Мальцев В.С., Кириленко Я.А., Березун Д.А. Обзор языков для безопасного программирования смарт-контрактов. Труды ИСП РАН, том 31, вып. 3, 2019 г., стр. 157-176 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(3)-13

1. Introduction

Initially, blockchains were designed for cryptocurrency management based on transactions. Further such systems involved *smart contracts* usage to enhance transactions, making them more sophisticated. This enabled to move part of an application logic into the blockchain, thus allowing to provide customizable redeeming conditions [1], develop crowdfunding systems [2], and other applications based on blockchain technology [3]. Fundamentally smart contracts are programmable objects beyond blockchain, intended to represent *automatable*[1] and *enforceable*[2] agreements [4].

Since smart contracts are essentially programs that are executed within the blockchain and written in some programming language, bugs and errors are possible. Erroneous transaction behavior can lead to financial damage. For example, a not-reentrancy of a function has caused $40 million loss [5]. Moreover, due to the immutable nature of the blockchain, it is often impossible to fix a

[1] «Automatable» rather than «automated» since parts of an agreement may require some human input.
[2] Enforceable either by law or by tamper-proof computer code.

contract with a bad[3] behavior that is already on the chain, i.e. contracts are irrevocably committed. One possible approach to detect such unwanted behaviors and minimize the number of vulnerabilities is to provide a way to formalize smart contracts properties and vulnerabilities. It will help to specify vulnerabilities sources and facilitate reasoning about smart contracts.

In [6] provided by IOHK research[4], an ontology that provides a set of basic conceptual primitives is specified. It can be used to construct desired propositions about smart contracts. It is not intended to be the only true ontology, rather the useful one. According to the ontology, blockchain based smart contracts can be considered as computations over blockchain state, that include the changing over time state itself as well as a transition function. And we will further refer to *modality* properties as to relationships between states, possibility or necessity properties that should be maintained throughout transitions.

These concepts allow thinking about smart contract behavior abstractedly over details. For example, consider a Deadline-*dependent Transfer*, a smart contract controlling property transfer between recipients[5]. Only *Recipient 1* may transfer the *Item* during some time interval prior to the deadline, while only *Recipient 2* may transfer the *Item* once the deadline has been passed. A *modality* property can be formulated in the following way. There always should be a blockchain state where at least one system participant who controls the *Item*, being transferred, exists, i.e. the absence of dead states in the blockchain.

Unfulfillment of those properties in blockchain based smart contracts may lead to money loss and malicious attacks. For example, a vulnerable sequence of smart contract library calls in PARITY wallet led to $150 million freezes on wallets [7].

State inconsistency and weaknesses may be caused by a number of different reasons such as blockchain-specific behavior, execution environment bugs, a model of underlying programming language that is not amenable to proof constructions, non-intuitive representation of programs in languages with good models, unintuitive semantics of underlying programming language for people who lack programming experience etc [8]. Also, some modality properties may never be proved because of possible non-termination of a program, which basically depends on a certain programming language.

Thereby, to make smart contracts secure it is desirable to be able to specify the intended behavior and properties that they should fulfill. These properties fulfillment can be provided with machine-checkable proofs and facilitated with more intuitive programming languages accompanied by tools for static analysis and formal verification to reduce the number of errors.

To date, various approaches, languages, and tools have been proposed: extensive type systems and various programming paradigms [9], programming languages that have easily checked termination conditions [10, 11], high-level languages that encourages safer programming via abstractions [12, 13], and intermediate and low-level languages that ease formal verification and compilers development [14–16].

Smart contract programming languages design is influenced by domain ontology, encountered vulnerabilities and ease of reasoning about modality properties. So, in this paper, we concentrate on the incorporation of known approaches used in design and development of smart contracts programming languages, proceed through vulnerabilities and domain-specific concepts that have been considered during design process, provide a classification of current efforts, and emphasize topics for future research.

The paper is organized as follows. Section 2 provides a short evaluation of similar works. Section 3 gives a brief summary of blockchain architecture principles relevant to languages and tools

[3] Here, by a «bad» contract behavior, we mean any behavior that is unexpected or undesirable by the contract owner, caused by any reason.

[4] IOHK company is one of the main customers of research in peer-to-peer networks. See https://iohk.io/about/ for details.

[5] Between users or other smart contracts.

design. Section 4 describes known vulnerabilities of smart contracts, classifying them for future analysis. Section 5 provides a survey of smart contract programming languages and their design ideas and principles, according to known vulnerabilities and blockchain architectures. In section 6 we discuss possible research gaps and future work. Section 7 concludes the paper.

2. Related works

Surely this paper is not the only one surveying smart contract programming languages, and we are aware of a couple of similar works. So, [17] provides an overview of smart contracts programming languages, security properties, and verification methods along with some classification of them. However, despite a good coverage, the proposed survey is rather superficial in a sense that it describes languages through the specification of their features, not going deep into design foundations that have provided the features. Another work [18] gives an overview of some distributed ledger systems, smart contracts languages, and technologies that might facilitate safety and performance, or make new applications possible. The paper is not aimed entirely at languages, hence it leaves the description without design foundations and any classification according to whether desirable properties or design principles. [15] also contains some overview of existing languages and their features, but the survey is performed from the perspective of comparison between them and the language proposed in the paper.

In contrast, this work is intended to enhance language coverage, provide foundations and intuition for reasoning, classification of languages, properties, and design fundamentals along with vulnerabilities that have influenced them.

3. Background

Since smart contracts are computations on a blockchain, underlying blockchain protocol basically sets the path for language and tools design. In this section, we review a few protocol details that influence further development of languages. Substantially there are two widespread blockchain architectures on top of which smart contracts are built to date — *UTxO-based* and *account-based* blockchains that allow stateless and stateful smart contracts respectively.

3.1 UTxO

Unspent transaction output, UTxO, model was introduced with the emergence of BITCOIN blockchain. A typical BITCOIN transaction contains a list of inputs that specify the funds that the transaction issuer can transfer and a list of outputs, that represent the way these funds are intended to be transferred. Each output can be used as an input for another transaction. For example, an issuer can set the amount of currency for each output or specify conditions, under which a possible receiver of funds can spend them, also they can specify themselves as the receivers to get so-called change. A set of UTxO consists of all transactions outputs that have not been yet used as inputs.

Redeeming conditions for transaction outputs in BITCOIN are defined with programs written in BITCOIN SCRIPT [10]. These programs describe properties that must be satisfied for the redeemer to be able to use these transaction outputs as their transaction inputs in order to spend the credits. The spender should provide input values to each locking script of referenced outputs of the previous transaction such that all scripts evaluate to value true, e.g. they may provide their wallet address and transaction signature to verify the authority.

Such scripts are stored within transactions and are being maintained only during a transaction, thus they have no state. Further scripts have limited access to blockchain data and essentially they are pure stateless functions of transaction data, i.e. of input parameters. Despite limitations, scripts along with transaction signatures can express complex redeeming conditions such as *multi-signature* payments, deposit providing, escrow, and dispute mediation, access to external data using oracles, time-locks, payment channels, cross-chain atomic trades etc [19]. Throughout the paper, we regard these scripts as stateless smart contracts.

3.2 Account-based blockchains

Account-based blockchains maintain an explicit state throughout transactions. A *state* is a mapping between account addresses and balances. Within these blockchain systems, each transaction is a mapping between the states. Basically, these systems are transaction-based state machines.

ETHEREUM is an example of such a system [20]. In ETHEREUM smart contracts are similar to users' accounts in a sense that they have their own address and a balance. Smart contracts are stored inside the blockchain and essentially these contracts are lists of functions that can be invoked through users' transactions or other contracts messaging. These functions are defined with bytecode of the corresponding execution environment called *Ethereum Virtual Machine*, EVM. Since any smart contract has a balance, it is a stateful function of a data transaction (or a message) and blockchain state, in which the transaction takes place, so they can write to blockchain state or read from it. Contracts state typically involves a stored amount of currency. However, in general, it can have arbitrary persistent storage that is maintained throughout the transitions of the blockchain.

3.3 Preventing the Denial-of-service attacks (DOS)

Despite the underlying blockchain model, smart contracts are computations that are replicated over blockchain via consensus protocol. To prevent DOS-attacks the number of computations for every program representing a smart contract should be restricted beforehand. Restriction mechanism depends on the underlying programming languages properties. One of the main properties in the context of smart contracts is halting, i.e. whether every program that has been written in it terminates or not. BITCOIN SCRIPT program always terminates since language is not Turing-complete and it does not have loops, or recursion, or any other mechanism that provides infinite computations. However, the size of a program also affects the performance of the system behind it. Thus BITCOIN SCRIPT programs are limited by the stack size and number of computationally heavy instructions, i.e. transactions that contain a script that does not satisfy restrictions are rejected.

For programs written in languages that do not guarantee program termination, e.g. EVM bytecode, program execution is limited via a gas system. *Gas* is basically an amount of cryptocurrency specified for contract execution. Fixed units of gas are charged to a miner for every instruction being executed. If the specified amount of gas is expired, execution of the contract stops. Furthermore, EVM contracts also have a limited stack size.

4. Smart contract weaknesses

In this section, programming language-level vulnerabilities that may cause unfulfillment of modality properties and possible mistakes are classified. It is worth to notice, that the most common property arising in distributed systems is that results of computations should be deterministic. While many smart contract programming languages have been designed with determinism in mind, sometimes general purpose programming languages are used for development [21]. A detailed overview of potential risks of non-determinism and causes can be found in [22].

We consider SOLIDITY language for stateful contracts since it is the most popular smart contract programming languages and generally it was one of the first languages that revealed such weaknesses, unfortunately on its own instance. Despite originally being known as unsafe, the language is evolving and to date its compiler is able to warn about code that might misbehave. However, Solidity has provided the foundation for the design of other languages. The most famous errors that have caused contracts failure are DAO [5] and PARITY [23].

SOLIDITY vulnerabilities are classified in the following subsections based on what level they occur on and the reasons that cause them. Code examples of the weaknesses could be found in [20, 24–28]. Possible attacks are discussed in [29]. Also, it is worth to mention, that SOLIDITY is a

Turing complete language, meaning that in general fulfillment of particular modality properties cannot be proved, even despite guaranteed termination due to gas limit.

4.1 Block content manipulation

Block of transactions in blockchains is formed by one of the participants who have the ability to influence block content. Thus, careless blocks handling may cause a number of errors.

Front-Running (Transaction-ordering dependence): It is important to be careful of transactions order. For example, Alice has deployed contract with possibility to sell a product and set a price for it. Bob wants to buy the product, and Alice wants to set a higher price. Let's assume, they want to do it at the same time. If Bob's request is the first, Alice loses money. In another case, Bob's transaction can be rejected, or Bob will spend more money than he expected.

Weak sources of randomness: Random values should be deterministic for all nodes in the network due to consensus considerations. One way to get randomness is to use pseudorandom values. Variables of contract, even the private ones, meta-variables of a block, or a hash of a previous and next block cannot be used as a source of entropy. In some blockchains (including ETHEREUM) it is possible to have influence over these variables during the validation process. A pseudo-random value in smart contract code can be predicted by a malefactor. Precalculation can be done via code analysis.

4.2 Contract interaction

A smart contract should be able to interact with other contracts. The following vulnerabilities appear due to the fact that smart contracts cannot rely on each other's behavior.

Unchecked return values for low-level calls: There are three functions to send ether [30] from account to account in ETHEREUM: send() and call() that return false if an error occurs but the transaction execution continues, and transfer( ) that rolls back the transaction in case of error. Low-level functions callcode() and delegatecall() behave in the same way as functions send() and call() . Thus handling of false value of corresponding functions is needed to avoid undesirable behavior of contract. According to Luu et al. [31], 27.9% of smart contracts in ETHEREUM blockchain do not check returned values.

Reentrancy: An external contract can call back functions of a caller contract before the first invocation has finished. It can lead to undesirable recursive function interactions and allow the callee contract to take over the control flow. The example of this vulnerability is a famous DAO smart contract [5].

Callstack bound: A failure may occur when an external call is made, but the program stack has reached its limit. Stack overflow is possible in smart contract languages. In EVM call stack is limited to 1024 stack frames. If the exception is not properly handled by a contract, the malefactor can use it to attack.

4.3 Resource limits

If the smart contract language is Turing-complete, there is a need in metering[6] mechanism to prevent infinite execution. ETHEREUM charges a fee, named gas. Amount of gas is proportional to the number of executed commands by EVM. Every transaction is bounded with the maximum amount of gas as well as blocks.

Infinite loops: Mistakes and misprints in operators usage may keep contracts syntactically correct but strongly affect their logic. For example, writing =+ instead of += in a loop terminating condition may lead to unexpected program behavior and even to an infinite loop. Moreover, in this case, excessive gas consumption may occur. It also includes situations when the number of memory addresses being used is significantly increased, e.g. when the number of elements in a map grows, it becomes too expensive to iterate over it.

[6] Metering is a way to limit and charge the execution of a smart contract.

4.4 Arithmetics

In SOLIDITY arithmetics is available on unsigned integers only and the language does not provide any arithmetic operations check for correctness. This class of mistakes mostly refers to common programmer errors. In the case of smart contracts, they may lead to a huge loss of assets. Thus, it is common to consider them as vulnerabilities in order to attract programmers attention.

Overflow and underflow: These vulnerabilities arise because numbers can have a fixed size. In case of ETHEREUM, maximum value for $uint(uint256)$ is 2^{256-1} and minimum – 0. A programmer has to manually checks overflow and underflow.

Floating points and precision: SOLIDITY does not have fixed and floating point types. Instead, a programmer has to emulate them via integers. All integer divisions are rounded down. Careless handling of such operations may cause unexpected program behavior.

4.5 Storage access

The following vulnerabilities are caused by negligent memory usage and access.

Uninitialized storage pointer: Local structures, arrays, and maps link to storage zero address by default. Using these objects without initialization will lead to overwriting whatever is in zero address.

Write to an arbitrary storage location: A smart contract can store some data and wrong variable assignment can break it. SOLIDITY has reference types. Mistake with references can lead to internal state corruption. If an array index is out of range, the exception will be thrown, and the smart contract will be reverted.

4.6 Internal control flow

This class of vulnerabilities is caused by a complex control flow graph structure and an ability to manipulate it.

Using inherited functions and variables: It is possible to use inheritance in smart-contracts languages with the object-oriented paradigm. SOLIDITY allows multiple inheritance. If several super-classes have a method or variable with the same name, their behavior in sub-class depends on the inheritance order. It could shadow previously defined values or functions and lead to undesirable results.

Using built-in functions: Programmers should be aware of using built-in functions and their behavior. E.g., someone would like to use assertions to check program invariant. SOLIDITY `assert()` function is intended for this purpose. In case of failure, this method throws an exception and does not return the remaining gas. Thus, to check for changing values, such as input data, it is recommended to use `require()` statement which in the same case does transaction rollback and returns remaining gas.

Using deprecated functions: It is not clear what new compiler versions do with deprecated functions. Therefore, it is not recommended to use these objects.

Locked assets: Contracts should provide a way to manage assets. Suppose in the example the contract has a method to take assets but does not have code to give them back. Due to smart contract code immutability in blockchain history, it is impossible to upgrade or fix this contract. It will cause property loss.

4.7 Authorization

Authorization is a major part of a person identification mechanism, designed to verify the permission for actions. Incorrect or insufficient authorization can lead to the following vulnerabilities.

Incorrect initialization: When the smart contract was deployed to a blockchain, it should be initialized. Often initialization contains sensitive operations such as a setting contract's owner. An

error in this action may violate the logic of the smart contract. In SOLIDITY, the *constructor* is a special function, which is called once to set the contract's state. In new SOLIDITY versions, constructors are denoted by a special keyword that made the definitions more obvious. But in earlier versions (less than 0.4.22) constructor is just a function with the same name as the class has. Thus, a typo in constructors' name makes it a usual function, which can be called by anybody since default modifier for a function is *public*.

Function default visibility: Incorrect access modifiers usage or a lack of them can lead to undesirable behavior. For example, calling the function that changes the contract owner with public access modifier allows everyone to become its owner. Default modifier for SOLIDITY is public. Thus, it is strongly recommended to explicitly define visibility for all functions and variables.

5. Smart contract languages

In this section, smart contract languages are considered with respect to their main features, paradigms, and common properties such as Turing-completeness, metering mechanism, reasoning, type system, code analyzers, etc. To reduce the number of subsections we have classified languages with respect to their level of usage.

Low-level: These languages are designed for direct execution by the underlying execution environment. Most concepts and principles of formal semantics, computational model, metering, logic for reasoning about programs, and typing are often introduced on that level. Furthermore, to date, smart contracts are mostly stored on the blockchain in low-level bytecode, which imposes suitability considerations. Examples of such languages are BITCOIN-SCRIPT [10], EVM [32], MICHELSON [33].

High-level: Languages with the idea of making the writing of contracts easier for developers via readability and safer high-level syntactic constructs enhanced by a type system that provides machine services abstractions. Safety aspect appears here and refers to the languages ability to guarantee the integrity of these abstractions and abstractions introduced by the programmer using definitional facilities of the language. In a safe language, such abstractions can be used abstractly while in an unsafe language they cannot: in order to completely understand how a program may (mis-) behave, it is necessary to keep in mind all sorts of low-level details such as the layout of data structures in memory and the order in which they will be allocated by the compiler [34]. The semantics of both levels should be considered here[7]. Examples of such languages are SOLIDITY [35], FLINT [12], and LIQUIDITY [36].

Intermediate-level: Languages that present a compromise between a high-level source and low-level target languages. As a general rule, they are designed in order to simplify program verification or static analysis, relying on the computation model, type system, reasoning, semantics, etc. Furthermore, they allow making unification of compilation, i.e. providing a language that can be compiled for different platforms. SCILLA [15] is an example of such a language.

It is also useful to emphasize some desirable language properties that affect language design.

- Reasoning – language behavior model should allow to specify modality properties and facilitate proving of their (un-) fulfillment. Underlying calculus model and type system are aimed at this.
- Safety – language abstractions should hold integrity property. Rigorous semantics promotes this.
- Expressivity – basically language should be expressive to fit a possible various range of use cases.
- Readability – language representation of a contract behavior should be intuitive, i.e. be easy to inspect and write with.

[7] Fundamentally safeness spreads to other levels since low-level language is an abstraction of its implementation, e.g. a virtual machine.

Table 1. Smart Contract Languages

Language	Level	Current state	Project	Paradigm / influence	Analyzers	Metering	Turing-completeness	Main features
Bamboo	high-level	alpha (experimental)	Ethereum	functional	EVM bytecode analyzers[1]	gas system	yes	program behaves as a state automata
Bitcoin Script	low-level	under development	Bitcoin	stack-based, reverse-polish	no[2]	script size	no	Forth-like syntax, any program always terminates
Chaincode	high-level	stable	Hyperledger Fabric	general purpose languages	no[2]	timeout	yes	GO, NODE.JS and JAVA extensions for smart contracts
EOSIO	high-level	stable	EOS.IO	object-oriented, statically typed	no[2]	bound system[3]	yes	C++11 library
EVM bytecode	low-level	stable	Ethereum	stack-based	EVM bytecode analyzers[1]	gas system	yes	well researched
Flint	high-level	alpha	Ethereum	contract-oriented, type safe	EVM bytecode analyzers[1]	gas system	yes	Swift-like syntax, safety
IELE	low-level	prototype	Ethereum	register-based	tools generated by K [37]	gas system	yes	generated from formal specification, LLVM IR-like syntax, safety
Ivy	high-level	prototype (experimental)	Bitcoin	imperative	no[2]	gas system	no	can be compiled to Bitcoin Script
Liquidity	high-level	under development	Tezos	fully-typed, functional	under development	gas system	yes	OClaml-like syntax, compiled to Michelson according to formal semantics, safety
LLL	intermediate-level	under development	Ethereum	stack-based	EVM bytecode analyzers[1]	gas system	yes	Lisp-like syntax, a wrapping over EVM bytecode
Logikon	high-level	experimental	Ethereum	logical-functional	EVM bytecode analyzers[1]	gas system	yes	translated to Yul
Michelson	low-level	under development	Tezos	stack-based, strongly typed	Typecheck system	gas system	yes	programs can be verified with Coq
Plutus (PlutusCore)	high-level (low-level)	under develop	Cardano	functional	no[2]	gas system	yes	Haskell-like syntax, formal specification
Rholang	intermediate-level	under development	RChain	functional	no[2]	rule reduction system[4]	yes	concurrent, Scala-like syntax, based on rho-calculus
Scilla	intermediate-level	under development	Zilliqa	functional	Scilla-checker	gas system	no[5]	embedded in Coq, formal specification
Simplicity	low-level	under development	Bitcoin	functional, combinator-based, typed	Bit Machine	Bit Machine cell usage	no	formal denotational and operational semantics
Solidity	high-level	stable	Ethereum	statically typed, object-oriented	EVM bytecode analyzers,[1] SmartCheck [38], ZEUS [39], Solidity* [40]	gas system	yes	JavaScript-like syntax, popularity
SolidityX	high-level	beta	Ethereum	secure-oriented	EVM bytecode analyzers[1]	gas system	yes	compiled to Solidity
Vyper	high-level	beta	Ethereum	imperative	EVM bytecode analyzers[1]	gas system	no	Python-like syntax, safety
Yul	intermediate-level	under development	Ethereum	object-oriented	EVM bytecode analyzers[1]	gas system	yes	intermediate language for future Solidity

[1] Programs can be translated to EVM bytecode, and then OYENTE [31], SECURIFY [41], EVM* [40], KEVM [42], MYTHRIL [43], VANDAL [44], RATTLE [45], MANTICORE [46] can be applied.
[2] To our knowledge there is no analyzer, which works with that smart contract language.
[3] Based on the amount of EOS tokens, more tokens — more computation power.
[4] Applying one reduction rule of rho-calculus [47] costs some value, paid by user.
[5] Any in-contract computation within a transition terminates, however non well-founded recursion in SCILLA can be implemented with contracts calling themselves or via explicit continuations, i.e. blockchain level interaction. Loops constructs are planned to be implemented via well-founded recursive functions.

Every smart contract language has domain specific instructions or/and types, e.g. cryptographic primitives, assets types, messaging instructions. So we will not emphasize this aspect much.

Notable features and models of several languages with respect to desirable properties are discussed below while a summary of a more expanded set of languages is presented in the table on the Table 1.

5.1 Low-level languages

1) **BITCOIN SCRIPT** is an untyped[8] stack-based low-level language for stateless smart contracts development in BITCOIN and handles transaction verification process. It is intentionally non-Turing-complete with the restricted instruction set where some opcodes are removed e.g. multiplication, division, strings operations, bitwise logic, due to possible overflow vulnerabilities and implementation bugs. Everything is allocated on the stack of limited size words while a program has access to some transaction fields e.g. a hash of transaction data, time field. Thus every program is a pure function of transaction data, i.e. transactions are *self-contained*.

To our knowledge, BITCOIN SCRIPT has no formal semantics, which makes metering ad-hoc and does not enforce formal verification. Furthermore, its stack-based nature and bytecode make smart contracts less auditable since only bytecode is stored inside transactions. Metering is performed via expensive operators counting and script size evaluation. However script's input is arbitrary, hence BITCOIN SCRIPT allows the specification of redemption properties like signature checking, pay-to-public-key-hash, pay-to-script-hash, multisignature checking, and arbitrary data storage inside transactions [1, 10, 48].

2) **SIMPLICITY**: is designed for extending BITCOIN SCRIPT capabilities. It is intended to enhance expressiveness, while enabling static analysis that allows to efficiently bound the number of computations, maintaining BITCOIN SCRIPT design of self-contained transactions, and providing formal semantic to facilitate reasoning about programs. It is anticipated to be used as a compilation target for high-level languages and deployed to *sidechains* [49]. SIMPLICITY is a typed non-Turing-complete combinator-based language with terms based on *Gentzen's sequent calculus*. Every SIMPLICITY type is finite: it contains finitely many values. Hence SIMPLICITY does not support recursive types and can express only finitary functions.

The core of SIMPLICITY consists of nine combinators for term construction with the corresponding denotational semantics. The language is formalized in COQ as well as the correctness of some functions built up from combinators, e.g. *half-adder* or *SHA-256 function*. Generally, the completeness, i.e. the notion that any function between SIMPLICITY types can be expressed with combinators, is verified in COQ.

Further, the operational semantics of SIMPLICITY is defined within the abstract machine called BIT MACHINE, intended to ease bounding of the number of computations, i.e. metering. It is designed to crash at anything that resembles undefined behavior. BIT MACHINE is an abstract imperative machine which state consists of two non-empty stacks of data frames formed by an array of cells. The machine has a set of instructions that manipulate the two stacks and their data frames, and corresponding operational semantics is defined by translating a SIMPLICITY expression into a sequence of BIT MACHINE instructions. It allows computational resources measuring with respect to cells and frames, e.g. the number of executed instructions, copied cells, maximum cells in both stacks at the given point, the number of frames in both stacks. Operational semantics correctness and its correspondence to the denotational semantics are verified in COQ. Furthermore, the set of core combinators can be extended for implementing a signature checking that requires transaction data, thus SIMPLICITY programs can be built to implement the pay-to-script hash scheme [50].

Summarizing, SIMPLICITY stateless nature and rather simple functional semantics without recursion and unbounded loops facilitate equational reasoning, avoiding complex logic. It provides means for formal verification of programs as well as static analysis more capable to effectively

[8] More precisely stack operates with byte vectors, which can be interpreted depending on the opcode.

bound the number of computational resources. To date SIMPLICITY has a HASKELL implementation under development [51].

3) **EVM**: is a bytecode language for *Ethereum Virtual Machine*. It is designed to support and execute arbitrary computations over ETHEREUM account-based blockchain, i.e. programs with loops and recursion. EVM is a *stack-based*, Turing-complete machine of 256-bit words with the memory model of word addressed byte array. The machine also has a persisted *storage* which is maintained between transactions and is a part of the blockchain state. It is a word-addressable word array. Program code is separated from data. Access to and modification of data in different types of memory is charged differently from storage — the most expensive to stack and memory being equally charged. The formal execution model and the environment is specified in ETHEREUM Yellow paper [32].

There are efforts on specifying formal semantics for EVM in OYENTE [31], F* [52], KEVM [42], and LEM [53] that focus on formal verification tools and detecting and avoiding insecure features of EVM, e.g. *delegatecall*, *overflows*, *undefined call/return*. Also, the poor human-readability of bytecode is a flaw. ETHEREUM includes many implementations of EVM, e.g. in JAVA SCRIPT, C++, PYTHON, and a promising WEBASSEMBLY implementation [54].

4) **IELE**: is a language defined within *K-framework*[9] [14]. It was designed to overcome EVM drawbacks with an idea of correctness by construction and formal verification in mind. It is intended to be secure and human-readable and to serve as a compilation target for high-level languages, thus unifying compilers construction. IELE is a register-based untyped[10] language: instructions operate on and store their output in an infinite number of virtual registers and have access to a persistent *storage* — the unbounded sparse array of arbitrary-precision signed integers. The language implementation is generated from its formal specification defined in K-framework, which provides generation of verification tools, debugger, interpreter, model checker, etc. IELE has functions and defines a call/return convention where a called function expects a specific number of parameters and returns a specific number of values or corresponding error status[11]. Furthermore, IELE avoids some insecure EVM features, e.g. by introducing *delegatecall* functionality and maintains arbitrary-precision arithmetic. Its operational semantics specifies contracts internal state, blockchain state, and transition rules, i.e. contract's code, intra-contract call stack, remaining gas, and the state of the local memory and virtual registers, storage content, balances, etc. Thus IELE makes formal verification less tedious, enhances human-readability, eliminates undefined, and implementation-defined behaviors, i.e. it is considered to be safe[12].

Gas costs for computation time are based on instructions asymptotic and the gas cost for memory is based on peak memory consumption. Gas model is designed to allow arbitrarily large valued instructions and to avoid artificial limits on the size of data or call stacks while preserving the existing goals of the EVM gas model. However, while arithmetics may cause overflows in EVM, in IELE it may cause out-of-gas exception, starting from some input size. Gas formulas are also specified in K.

5) **MICHELSON** [33, 55]: is a typed stack-based language designed to be on-chain code for stateful smart contracts in TEZOS. It is intended to be a more readable compilation target and more amenable for formal verification.

A MICHELSON program supports high-level types (e.g. *map*, *list*, *set*, etc.) and receives an input stack with parameters and storage being pushed on. It evaluates to a result stack with an output value and new storage or can fail. The language does not support closures in the sense that every

[9] Framework used to produce implementation derived from formal specifications, based on logic rules.

[10] Arbitrary-precision signed integer is the main datatype.

[11] For reference, in EVM caller sends an arbitrary byte stream containing the call arguments values since functions are represented as a set of JUMP labels.

[12] IELE is stated to be the first real-world language that is designed and implemented using formal semantics, with a zero gap between the formal specification and the implementation.

function has an empty environment. Messaging with other contracts is performed through passing a storage and not maintains the stack between calls. The types are predefined[13] and monomorphic, further types of input, output, and storage of a contract are fixed and it is statically ensured that resulting storage type is preserved. MICHELSON has a built-in type for cryptocurrency and operations defined for this type are mandatory checked for *underflow/overflow*. Typing is done via types propagation. Due to its computation model, MICHELSON has a straightforward semantics, based on rewriting rules defined on stack and syntax. Also, it defines what is considered as well-typed stacks and the resulting outputs. MICHELSON is currently implemented in OCAML via GADT with an interpreter defined corresponding to the semantics while leaving the type checking to OCAML. It is anticipated to replace current implementation with a one verified with either COQ or F* [56].

6) **PLUTUS CORE:** is a typed language designed for use as a transaction validation language in *UTxO*-based blockchain systems. Fundamentally it is eagerly-reduced higher-order polymorphic λ-calculus extended with iso-recursive types, higher kinds, and a library of basic types and functions, hence it has a straightforward operational semantics. The language is meant to be a compilation target since it is difficult to write and read but it is intended to be formally verifiable in proof assistants.

PLUTUS CORE program is a closed term, and its execution is performed by (possibly non-terminating) reduction of welltyped terms. All types can be normalized and normalization process always terminates. Further, operations on types allow to deal with sized types, i.e. sized integers or bytestrings that allows them to be tracked in the type system to facilitate charging for the appropriate amount of gas and detecting overflows at the type level. The language has a specified abstract machine intended to be amenable for a verification reference implementation. Moreover, PLUTUS CORE has its formal specification defined in K [57, 58].

Transaction validation is performed similarly to BITCOIN SCRIPT. Validation is successful if the PLUTUS CORE program reduces to a non-error value within an allotted number of steps. But it is more extended in a sense that a program has a read-only access to world state passed through a *monad* [59, 60].

PLUTUS CORE is an on-chain language for CARDANO blockchain and is embedded into HASKELL. Furthermore, the blockchain system itself is implemented in HASKELL as well as off-chain computations, e.g. wallets, it allows type checking on the level of the interaction between off-chain applications and on-chain code.

5.2 High-level languages

1) **SOLIDITY**: is a very rich and expressive high-level object-oriented Turing-complete language [35] for writing smart contracts for EVM with a syntax similar to JAVASCRIPT and C++. It has static types, inheritance, libraries, complex user-defined types supporting, and other features. As a consequence, that causes its prevalence as well as a large number of potential vulnerabilities (see section IV).

2) **SOLIDITYX**: is a high-level language [61] which compiles to SOLIDITY. SOLIDITYX is a *secure-oriented* language, which means that it has a defense from some vulnerabilities by default, for example, all access modifiers are *private* by default. However, SOLIDITYX is in beta development now and it is not recommended to be used in production.

3) **VYPER** (aka VIPER): is a high-level language for implementing smart contracts for the EVM [13]. It is PYTHON3 derived programming language. VYPER is an alternative to SOLIDITY that is aimed at code security, clarity, and unambiguity, for example, it excludes constructions that can lead to misleading code. To achieve this VYPER does not support modifiers, class inheritance, inline assembly, function overloading, operator overloading, recursive calling, infinite-length

[13] A programmer cannot define their own types.

loops, binary fixed point. The language also leverages overflow checking, array bounding, and limited state modification.

4) **FLINT**: is a high-level statically-typed contract-oriented language aimed to write robust smart contracts on EVM [12]. FLINT provides a mechanism to specify actors that can interact with a contract, immutability by default, assets types, and safer semantics with overflows causing revert of a transaction and explicit states.

5) **BAMBOO**: is a high-level language compiling to the EVM [62]. Its compiler is implemented in OCAML thus BAMBOO is well amenable to formal verification. BAMBOO creates clarify state transitions and avoids reentrancy problems by default. However, it does not support loops and assignments into storage variables, except array elements, which improves the ability of contracts to be verified but complicates their development.

6) **LOGIKON**: is a high-level logical-functional language compiled to YUL [63]. LOGIKON program represents a set of logical constraints statically and formally verified.

7) **IVY**: is a language [64], designed to simplify programming of stateless smart contracts for BITCOIN. Compare to BITCOIN SCRIPT, in IVY program it is possible to use named variables, named clauses, domain-specific types, syntax sugar for function calls.

8) **LIQUIDITY**: is a functional, statically and strongly typed language, compiled down to MICHELSON. It has OCAML syntax and keeps safety guaranteed by MICHELSON, while providing high-level constraints. LIQUIDITY has a formal specification of the compilation semantics [65] and supports decompilation back from MICHELSON, based on the graph produced by symbolic execution that is eventually transformed into LIQUIDITY *AST*. This feature greatly enhances readability, since stack-based MICHELSON code is rather hard to inspect manually.

9) **CHAINCODE**: is a smart contract program, written for HYPERLEDGER FABRIC [21] blockchain. CHAINCODE can be developed with GO, NODE.JS or JAVA. The code should implement a special interface to interact with the blockchain network. Unlike ETHEREUM smart contracts, CHAINCODE does not have account address or associated assets, but the smart contract can have a mapping of the real assets to the internal state. CHAINCODE has the similar conception to database stored procedures. When a transaction is created, CHAINCODE is called to perform operations according to the transaction data. Possible operations are: read, update or delete data, stored in the ledger. Also, it is possible to invoke or read the state of another CHAINCODE, if the caller has enough permissions.

5.3 Intermediate-level languages

1) **YUL** (JULIA or IULIA): is an intermediate language [66]. It can be compiled to a number of backends: EVM 1.0, EVM 1.5 and eWASM. It is planning to use YUL as an intermediate language in the future versions of the SOLIDITY compiler. YUL can be used for "inline assembly" inside SOLIDITY.

2) **RHOLANG**: is a functional, concurrent, based on *rho-calculus* [47] language [67], used in project RCHAIN. A smart contract in terms of RCHAIN is a process, which has persistent state, its own code, and associated address. Execution of code is done by applying the reduction rule of rho-calculus. RHOLANG has behavioral types [68], reflection, reactive API, asynchronicity. Synchronization primitives for parallel execution of transactions are *messages* and *channels*. Messages are the way to communicate smart contracts with each other, sending values through channels. A user has to pay a cost in special tokens, named *Phlogiston*, to the node in the system for computational resources. These tokens will be used for executing smart contract's code. Rate-limiting mechanism looks like the *gas system* in ETHEREUM. Unlike EVM, where gas metering is done on the VM level, manipulations on *Phlogistons* are injected in smart contract's source code by RHOLANG compiler.

3) **SCILLA**: is intended to be an intermediate level language as a translation target for high-level languages to facilitate program analysis and *verification* before compiling to executable bytecode.

SCILLA is a typed language built on stateful smart contracts, i.e. contracts that have a state represented with a storage and that can communicate either with other contracts via messages or with the off-chain world by raising events or with blockchain explicitly reading blockchain data. The language design is aimed to facilitate formal reasoning providing clear and principled semantics.

Specifically, its semantics is based on communicating automata that separate contract specific computations called *transitions* and blockchain-wide interactions, i.e. messaging with other contracts, thus making transitions atomic. Atomicity is achieved through allowing only tail-calling communications which eliminate reentrancy problems. However non-tail calls are needed for some computations e.g. passing and saving some value back from the callee, it is implemented with explicit continuations mechanism. Nevertheless, possible nonterminating execution can be caused by non-well-founded recursion, which is going to be handled with gas usage. Further, SCILLA specifies pure, i.e. that change the state and impure transitions and those reading blockchain data, e.g. block number with OCAML based syntax.

SCILLA has been shallow-embedded in COQ, specifying such properties as contract terminology, contract state, and transitions along with blockchain states, which allows properties verification in isolation. So its design implies leveraging of formal reasoning to prove different modality properties, e.g. safety[14], liveness[15] or termination for well-founded recursive functions. It is anticipated to enhance support for automating the proofs of safety/temporal properties.

4) **LLL**: is a Lisp-like language [69] for EVM. Main purpose of LLL is to provide a little bit higher level of abstraction upon EVM bytecode, i.e. programmer has more high-level constructions to work with the stack. Also language has more functionality over the base set of EVM opcodes, such as multiary operators (they can be applied to one or more arguments, the result of following code (+ 1 2 3 4 5) is 15), including files, control structures, and macro definitions. LLL has an analog of variables, it makes automatic memory management for saving values.

6. Discussion

We briefly described notable approaches for specification of smart contracts intended behavior and analysis of behavioral properties. However, this survey is nevertheless incomplete. The area of blockchain and smart contracts is under active research. The community tries to apply different approaches and ways in the area of smart contract languages and their execution environments development. Some of them are Turing-completeness, paradigm (e.g. imperative, object-oriented, functional), level of abstraction, a way to limit code execution (metering systems such as ETHEREUM gas, time bounds, number of instructions) and a formal theory on which a language is based.

In the rest of the section, we discuss contributions that have not been classified in previous sections, propose aspects that may worth future researching and related work, and summarize possible pros and cons of provided aspects.

Recall that most smart contracts in blockchains are irreversible, i.e. they are hard to fix once they are deployed. One approach to mitigate this is a design pattern provided in [70, 71] that leverages using *delegatecalls*. It suggests deploying contracts with another *dispatcher* contract. The increased number of messages makes analysis and reasoning more complicated since dispatcher contracts should be robust and safe then. Another approach is platforms that allow *upgradable* contracts [72].

Arguable concept is the representation in which contracts are deployed to a blockchain. Most of the systems included in our survey store on-chain code in some low-level form. Such form hardens

[14] These are invariants that hold through the lifetime of a contract, exposing that nothing should go wrong.
[15] Basically, it states that something should eventually happen.

auditability, while also may serve as a uniform compilation target. That facilitates the development with different languages. There are platforms where contracts are stored as programs written in high-level safe languages [72]. Another possible approach for this is decompilation from low-level byte code to more high level code like in MICHELSON and LIQUIDITY case. However, to our knowledge, only this couple of languages have formalized semantics of compilation, while none of the known works provides the correctness of interpretation and interpretation after compilation at all, i.e. the correctness of the compiler or the commutativity of the implied diagram.

One more problem is a metering system for smart contracts, such as ETHEREUM gas and its analogs. Gas estimation is in general undecidable. It could be useful to find mechanisms to predict gas consumption. Improper estimation may lead to vulnerabilities (e.g. DoS-attacks), or to fails during code execution (e.g. ETHEREUM out-of-gas exception). Gas consumption depends on many factors such as memory usage and blockchain state. Various adaptive methods like type system are already surveyed PLUTUS [58], rigorous semantics with asymptotic analysis as in IELE [14], or dynamic adjustment as adaptive gas cost mechanism in [73] may be promising, as well as methods based on symbolic paths exploration and resource analysis [74, 75]. For example, PLUTUS design of unbounded integers allows metering statically due to its type system, while unbounded integers in IELE allows only dynamic gas evaluation. One may apply techniques like RAML [76]. Gas reducing optimization are also worth considering[16].

Since smart contracts use cases are yet to be researched, it is undesirable to restrict either statefulness of contracts or Turing-completeness of languages they are written in. The compromise between an ability to run arbitrary computations on the blockchain and amenability to reasoning defines future research topics. For instance, in [9] dependent types of IDRIS language are leveraged for writing provable smart contracts, that are compiled down to run on ETHEREUM. Languages based on models, which better describe an interaction between contracts based on message passing may become future research objectives, e.g. languages based on process calculus [77]. Extensive type systems in such systems also worth researching, e.g. behavioral type systems or linear epistemic ones [78]. Type annotating while writing a contract with such languages is often non-trivial as well as robust and safe contracts development in general. There are researches aimed at domains formalizing, e.g. *finances* and at the design of simpler languages that are embedded in some safe language for only domain purposes [79]. Such domain specific languages tend to be visual to ease the development process for non-experts in programming. Approaches aimed at actor's behavior are as well interesting. There is a DSCP contracting protocol for trading proposed in [80]. The protocol was verified using game theory and statistical models, such as Markov decision processes.

There is still another point about properties to consider. It is modality properties formulating, an i.e. specification of such a property a smart contract should satisfy. If the property of unfulfillment can be proved, it would prevent some exploit, e.g. already mentioned DAO. Some such properties can be seen in [81]. It proposes BITML – Bitcoin Modelling Language that leverages process calculus to describe interactions between participants and generate BITCOIN transactions according to symbolic semantics. In [82] EVM is formalized in LEM for modeling smart contracts behavior with some properties defined.

To outline the discussion, it is worth to notice that many researches avoid the infrastructure around the language, i.e. development environments, testing and deployment tools, extensive *API* libraries. However, these are essential components of successful development and a field for a plenty of practical studies, since to date only ETHEREUM has a rather complete infrastructure.

[16] Due to safety considerations, such optimization should be proven to be semantically equivalent. However, we are not aware of any related results.

7. Conclusion

As smart contracts platforms are intended to reasonably automate the economy, smart contracts should be safe and robust. In this paper, we have presented an overview the state of art of smart contract programming languages. We have classified weaknesses and vulnerabilities smart contracts are prone to. Languages calculus models, semantics, and type systems have been surveyed as well as other properties according to reasoning, safety, expressiveness, and readability. In the end, we have summarized related work and possible future research topics.

References

[1] Bitcoin contract. URL: https://en.bitcoin.it/wiki/Contract (Date: 2019-01-30).
[2] Solidity-example-crowdfunding. URL: https://github.com/zupzup/solidity-example-crowdfunding (Date: 2019-01-30).
[3] D. Macrinici, C. Cartofeanu, and S. Gao. Smart contract applications within blockchain technology: A systematic mapping study. Telematics and Informatics, vol. 35, no. 8, 2018, pp. 2337–2354.
[4] C. D. Clack, V. A. Bakshi, and L. Braine. Smart contract templates: foundations, design landscape and research directions. CoRR, vol. abs/1608.00771, 2016.
[5] A 50 million hack just showed that the dao was all too human. URL: https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/ (Date: 2019-01-30).
[6] D. McAdams. An ontology for smart contracts. URL: https://cryptochainuni.com/wp-content/uploads/Darryl-McAdams-An-Ontology-for-Smart-Contracts.pdf (Date: 2019-02-07).
[7] N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on ethereum smart contracts sok. In Proc. of the 6th International Conference on Principles of Security and Trust, 2017, pp. 164–186.
[8] G. Destefanis, A. Bracciali, R. Hierons, M. Marchesi, M. Ortu, and R. Tonelli. Smart contracts vulnerabilities: A call for blockchain software engineering. ResearchGate, 2018.
[9] Safer smart contracts through type-driven development. URL: https://publications.lib.chalmers.se/records/fulltext/234939/234939.pdf (Date: 2019-01-30).
[10] Bitcoin script. URL: https://en.bitcoin.it/wiki/Script (Date: 2019-01-30)
[11] R. O'Connor. Simplicity: A new language for blockchains. CoRR, vol. abs/1711.03028, 2017.
[12] Flint. URL: https://github.com/flintlang/flint (Date: 2019-01-30).
[13] Vyper. URL: https://github.com/ethereum/vyper (Date: 2019-01-29).
[14] T. Kasampalis, D. Guth, B. Moore, T. Serbanuta, V. Serbanuta, D. Filaretti, G. Rosu, and R. Johnson. Iele: An intermediate-level blockchain language designed and implemented using formal semantics. University of Illinois, Tech. Rep., http://hdl.handle.net/2142/100320, July 2018.
[15] I. Sergey, A. Kumar, and A. Hobor. Scilla: a smart contract intermediate-level language. CoRR, vol. abs/1801.00687, 2018.
[16] Plutus core specification. URL: https://github.com/input-output-hk/plutus/tree/master/plutus-core-spec (Date: 2019-01-30).
[17] D. Harz and W. J. Knottenbelt. Towards safer smart contracts: A survey of languages and verification methods. CoRR, vol. abs/1809.09805, 2018.
[18] P. L. Seijas, S. Thompson, and D. McAdams. Scripting smart contracts for distributed ledger technology. Cryptology ePrint Archive, Report 2016/1156, 2016, https://eprint.iacr.org/2016/1156.
[19] Contract. URL: https://en.bitcoin.it/wiki/Contract (Date: 2019-01-30).
[20] Ethereum contract security techniques and tips. URL: https://github.com/ethereum/wiki/wiki/Safety (Date: 2019-01-29).
[21] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolic, S. W. Cocco, and J. Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In Proc. of the Thirteenth EuroSys Conference, 2018, pp. 30:1–30:15.
[22] K. Yamashita, Y. Nomura, E. Zhou, B. Pi, and S. Jun. Potential risks of hyperledger fabric smart contracts. In Proc. of the 2019 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE), 2019, pp. 1–10.
[23] 300m in cryptocurrency accidentally lost forever due to bug. URL: https://www.theguardian.com/technology/2017/nov/08/cryptocurrency-300m-dollars-stolen-bug-ether (Date: 2019-01-30).

[24] Smart contract weakness classification and test cases. URL: https://smartcontractsecurity.github.io/SWC-registry/ (Date: 2019-01-29).
[25] Decentralized application security project. URL: https://dasp.co (Date: 2019-01-29).
[26] Security considerations. URL: https://solidity.readthedocs.io/en/latest/security-considerations.html (Date: 2019-01-29).
[27] Vulnerabilities description. URL: https://github.com/trailofbits/slither/wiki/Vulnerabilities-Description (Date: 2019-01-30).
[28] Smart contract weakness classification and test cases. URL: https://smartcontractsecurity.github.io/SWC-registry/ (Date: 2019-01-22).
[29] N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on ethereum smart contracts sok. In Proc. of the 6th International Conference on Principles of Security and Trust, 2017, pp. 164–186.
[30] Ether — the crypto-fuel for the ethereum network. URL: https://www.ethereum.org/ether (Date: 2019-01-30).
[31] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In Proc. of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016, pp. 254–269.
[32] D. G. Wood. Ethereum: A secure decentralised generalised transaction ledger. URL: https://ethereum.github.io/yellowpaper/paper.pdf (Date: 2019-01-31).
[33] Michelson language. URL: https://www.michelson-lang.com/ (Date: 2019-01-31).
[34] B. C. Pierce. Types and Programming Languages, 1st ed. The MIT Press, 2002.
[35] Solidity. URL: https://github.com/ethereum/solidity (Date: 2019-01-29).
[36] Liquidity. URL: https://github.com/OCamlPro/liquidity (Date: 2019-01-29).
[37] Grigore Roșu and T. F. Șerbănuță. An overview of the k semantic framework. The Journal of Logic and Algebraic Programming, vol. 79, no. 6, 2010, pp. 397–434.
[38] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In Proc. of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain, 2018, pp. 9–16.
[39] S. Kalra, S. Goel, M. Dhawan, and S. Sharma. Zeus: Analyzing safety of smart contracts. In Proc. of the Network and Distributed Systems Security (NDSS) Symposium, 2018.
[40] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin. Formal verification of smart contracts: Short paper. In Proc. of the 2016 ACM Workshop on Programming Languages and Analysis for Security, 2016, pp. 91–96.
[41] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev. Securify: Practical security analysis of smart contracts. In Proc. of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 2018, pp. 67–82.
[42] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, B. Moore, Y. Zhang, D. Park, A. Stefanescu, and G. Rosu. Kevm: A complete semantics of the ethereum virtual machine. In Proc. of the 2018 IEEE 31st Computer Security Foundations Symposium, 2018, pp. 204–217.
[43] Mythril. URL: https://github.com/ConsenSys/mythril-classic (Date: 2019-01-29).
[44] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz. Vandal: A scalable security analysis framework for smart contracts. CoRR, vol. abs/1809.03981, 2018.
[45] Rattle. URL: https://github.com/trailofbits/rattle (Date: 2019-01-30).
[46] Manticore. URL: https://github.com/trailofbits/manticore (Date: 2019-01-30).
[47] L. G. Meredith and M. Radestock. A reflective higher-order calculus. Electronic Notes in Theoretical Computer Science, vol. 141, 2005, pp. 49–67.
[48] Bitcoin weaknesses. URL: https://en.bitcoin.it/wiki/Weaknesses (Date: 2019-01-30).
[49] A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timón, and P. Wuille, Enabling blockchain innovations with pegged sidechains. 2014. https://blockstream.com/sidechains.pdf (Date: 2019-01-30).
[50] Mediawiki. URL: https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki (Date: 2019-02-5).
[51] Simplicity. URL: https://github.com/ElementsProject/simplicity (Date: 2019-02-5).
[52] Grishchenko I., Maffei M., Schneidewind C. A semantic framework for the security analysis of ethereum smart contracts – technical report (2018). URL: https://secpriv.tuwien.ac.at/tools/ethsemantics. (Date: 2019-01-30).
[53] Formalization of ethereum virtual machine in lem. URL: https://github.com/pirapira/eth-isabelle (Date: 2019-01-30).
[54] Ewasm: design overview and specification. URL: https://github.com/ewasm/design (Date: 2019-01-30).

[55] Michelson: the language of smart contracts in tezos. URL: http://www.liquidity-lang.org/doc/reference/michelson.html (Date: 2019-01-30).
[56] Why michelson? URL: https://www.michelson-lang.com/why-michelson.html (Date: 2019-02-5).
[57] Plutus core semantics. URL: https://github.com/kframework/plutus-core-semantics (Date: 2019-01-30).
[58] Plutus implementation and tools. URL: https://github.com/input-output-hk/plutus (Date: 2019-01-30).
[59] The extended utxo model. URL: https://github.com/input-output-hk/plutus/tree/master/docs/extended-utxo (Date: 2019-02-5).
[60] Is it smart to use smart contracts? URL: https://plutusfest.io/presentations/Philip-Wadler/Wadler30.pdf (Date: 2019-02-5).
[61] Solidityx. URL: https://solidityx.org/ (Date: 2019-01-30).
[62] Bamboo. URL: https://github.com/pirapira/bamboo (Date: 2019-01-30).
[63] Logikon. URL: https://github.com/logikon-lang/logikon (Date: 2019-01-31).
[64] Ivy: Bitcoin smart contracts. URL: https://github.com/ivy-lang/ivy-bitcoin (Date: 2019-01-30).
[65] Çagdas Bozman, M. Iguernlala, M. Laporte, F. L. Fessant, and A. Mebsout. Liquidity: Ocaml pour la blockchain. Journées Francophones des Langages Applicatifs 2018, 2018.
[66] Yul. URL: https://solidity.readthedocs.io/en/latest/yul.html (Date: 2019-01-30).
[67] Rchain and rholang. URL: https://www.rchain.coop/platform (Date: 2019-01-30).
[68] D. Ancona, V. Bono, and M. Bravetti. Behavioral Types in Programming Languages. Hanover, MA, USA: Now Publishers Inc., 2016.
[69] G. Wood. LLL. URL: https://lll-docs.readthedocs.io/en/latest/index.html (Date: 2019-01-30).
[70] Upgradable contract with solidity. URL: https://gist.github.com/Arachnid/4ca9da48d51e23e5cfe0f0e14dd6318f (Date: 2019-01-30).
[71] Proxy libraries in solidity. URL: https://blog.zeppelin.solutions/proxy-libraries-in-solidity-79fbe4b970fd (Date: 2019-01-30).
[72] The pact smart-contract language. URL: http://kadena.io/docs/Kadena-PactWhitepaper.pdf (Date: 2019-01-30).
[73] T. Chen, X. Li, Y. Wang, J. Chen, Z. Li, X. Luo, M. H. Au, and X. Zhang. An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks. CoRR, vol. abs/1712.06438, 2017.
[74] E. Albert, P. Gordillo, A. Rubio, and I. Sergey. GASTAP: A gas analyzer for smart contracts. CoRR, vol. abs/1811.10403, 2018.
[75] M. Marescotti, M. Blicha, A. E. J. Hyvärinen, S. Asadi, and N. Sharygina. Computing exact worst-case gas consumption for smart contracts. In Proc. of the International Symposium on Leveraging Applications of Formal Methods, 2018.
[76] J. Hoffmann, A. Das, and S. Weng. Towards automatic resource bound analysis for ocaml. CoRR, vol. abs/1611.00692, 2016.
[77] J. Baeten. A brief history of process algebra. Theoretical Computer Science, vol. 335, no. 2, 2005, pp. 131–146.
[78] H. Deyoung and F. Pfenning. Reasoning about the consequences of authorization policies in a linear epistemic logic. In Proc. of the Workshop on Foundations of Computer Security, 2009.
[79] S. Thompson and P. L. Seijas. Marlowe: Financial contracts on blockchain. Lecture Notes in Computer Science, vol. 11247, 2018, pp. 356–375.
[80] G. Bigi, A. Bracciali, G. Meacci, and E. Tuosto. Validation of decentralized smart contracts through game theory and formal methods. Lecture Notes in Computer Science, vol. 9465, 2015, pp. 142–161.
[81] M. Bartoletti and R. Zunino. Bitml: A calculus for bitcoin smart contracts. In Proc. of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 2018, pp. 83–100.
[82] Y. Hirai. Defining the ethereum virtual machine for interactive theorem provers. Lecture Notes in Computer Science, vol. 10323, 2017, pp. 520–535.

Информация об авторах / Information about authors

Алексей Валерьевич ТЮРИН учится на четвёртом курсе на кафедре системного программирования СПбГУ. В его научные интересы входит исследование и создание инструментов для разработки смарт-контрактов и теория формальных языков.

Alexey Valerievitch TYURIN is a fourth-year student at the Department of Software Engineering at St. Petersburg State University. His scientific interests include research and creation of tools for the development of smart contracts and the theory of formal languages.

Иван Владимирович ТЮЛЯНДИН бакалавр кафедры системного программирования СПбГУ 2019 года выпуска. Областями научных интересов являются блокчейны, а также языки смарт-контрактов и их свойства.

Ivan Vladimirovitch TYULYANDIN is a Bachelor of the Department of Software Engineering at St. Petersburg State University (2019). Areas of scientific interest are blockchains, as well as languages of smart contracts and their properties.

Владимир МАЛЬЦЕВ перешел на четвёртый курс на кафедре системного программирования СПбГУ. Исследовательские интересы Владимира: анализ состояния блокчейн-сети и разработка приложений поверх блокчейна.

Vladimir MALTSEV moved to the fourth course at the Department of Software Engineering at St. Petersburg State University. Vladimir's research interests: analyzing the state of the blockchain network and developing applications on top of the blockchain.

Яков Александрович КИРИЛЕНКО закончил кафедру системного программирования СПбГУ, преподает на кафедре с 2006 года, совмещая с исследованиями и работой в коммерческих проектах по темам научных интересов: статический анализ кода, реинжиниринг программных комплексов, технология программирования киберфизических систем.

Jacob Alexandrovitch KIRILENKO graduated from the Department of Software Engineering at St. Petersburg State University. He has been teaching at the Department since 2006, combining research and work in commercial projects on scientific interests: code static analysis, software reengineering, cyber-physical programming technology.

Даниил Андреевич БЕРЕЗУН является кандидатом физико-математических наук. Он защитил кандидатскую диссертацию на тему «Трассирующая нормализация» в марте 2018 года. В настоящее время Даниил является руководителем исследовательской группы метавычислений и распределённых технологий в составе лаборатории языковых инструментов JetBrains Research, а также доцентом в Высшей школе экономики.

Daniil Andreevitch BEREZUN is a PhD in computer science. He defended his PhD thesis on traversal-based normalization in March 2018. Daniil graduated from the St. Petersburg State University, Mathematics and Mechanics Faculty, Department of Computer Science in 2014. Nowadays, he is the head of Metacomputations and Distributed Technologies research group of Programming Languages and Tools Lab in JetBrains Research. He also works as a lecturer at the Department of Computer Science at the Higher School of Economics.

DOI: 10.15514/ISPRAS-2019-31(3)-14

Vulnerabilities Detection via Static Taint Analysis

[1] *N.V. Shimchik, ORCID: 0000-0001-9887-8863 <shimnik@ispras.ru>*
[1,2] *V.N. Ignatyev, ORCID: 0000-0003-3192-1390 <valery.ignatyev@ispras.ru>*
[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia*
[2] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia*

Abstract. Due to huge amounts of code in modern software products, there is always a variety of subtle errors or flaws in programs, which are hard to discover during everyday use or through conventional testing. A lot of such errors could be used as a potential attack vector if they could be exploited by a remote user via manipulation of program input. This paper presents the approach for automatic detection of security vulnerabilities using interprocedural static taint analysis. The goal of this study is to develop the infrastructure for taint analysis applicable for detection of vulnerabilities in C and C++ programs and extensible with separate detectors. This tool is based on the Interprocedural Finite Distributive Subset (IFDS) algorithm and is able to perform interprocedural, context-sensitive, path-insensitive analysis of programs represented in LLVM form. According to our research it is not possible to achieve good results using pure taint analysis, so together with several enhancements of existing techniques we propose to supplement it with additional static symbolic execution based analysis stage, which has path-sensitivity and considers memory region sizes for filtering results found by the first stage. The evaluation of results was made on Juliet Test Suite and open-source projects with publicly known vulnerabilities from CVE database.

Keywords: static code analysis; taint analysis; vulnerabilities

For citation: Shimchik N.V., Ignatyev V.N. Vulnerabilities Detection via Static Taint Analysis. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 3, 2019. pp. 177-190. DOI: 10.15514/ISPRAS-2019-31(3)-14

Поиск уязвимостей при помощи статического анализа помеченных данных

[1] *Н.В. Шимчик, ORCID: 0000-0001-9887-8863 <shimnik@ispras.ru>*
[1,2] *В.Н. Игнатьев, ORCID: 0000-0003-3192-1390 <valery.ignatyev@ispras.ru>*
[1] *Институт системного программирования имени В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*
[2] *Московский государственный университет имени М.В. Ломоносова, 119991, Россия, Москва, Ленинские горы, д. 1*

Аннотация. В связи с большими объёмами кода в современных программных продуктах, в программах всегда существует целый набор малозаметных ошибок или дефектов, которые сложно обнаружить при повседневном использовании или в ходе обычного тестирования. Многие такие ошибки могут быть использованы в качестве потенциального вектора атаки, если они могут быть эксплуатированы удалённым пользователем посредством манипуляций над входными данными программы. В данной работе представлен подход к автоматическому обнаружению уязвимостей безопасности с использованием межпроцедурного статического анализа помеченных данных. Цель данного исследования – разработка инфраструктуры анализа помеченных данных, применимой для обнаружения уязвимостей в программах на языках C и C++ и расширяемой при помощи отдельных детекторов. Этот инструмент основывается на алгоритме решения задачи Межпроцедурных,

Конечных, Дистрибутивных Подмножеств (IFDS) при помощи её сведения к специальной задаче о достижимости на графе и способен выполнять межпроцедурный, чувствительный к контексту, нечувствительный к путям анализ программ, представленных в виде LLVM-биткода. Анализа помеченных данных недостаточно для получения хороших результатов, поэтому улучшения существующих методов, мы предлагаем дополнить его ещё одним этапом анализа, который основан на статическом символьном выполнении. Для фильтрации результатов первого этапа выполняется анализ, чувствительный к путям и учитывающий размеры регионов памяти. Оценка результатов была проведена на Juliet Test Suite и проектах с открытым исходным кодом, имеющих подходящие публично известные уязвимости из базы данных CVE.

Ключевые слова: статический анализ кода; анализ помеченных данных; уязвимости

Для цитирования: Шимчик Н.В., Игнатьев В.Н. Поиск уязвимостей при помощи статического анализа помеченных данных. Труды ИСП РАН, том 31, вып. 3, 2019 г., стр. 177-190 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(3)-14

1. Introduction

In the paper, we consider a specific subset of all possible software vulnerabilities – ones which are caused by utilizing unchecked user-provided data in critical functions or code instructions. This class includes but is not limited to vulnerabilities allowing such important attacks as SQL Injection, Buffer Overflow and XSS attacks.

One group of methods used to represent and discover such vulnerabilities is called taint analysis. In general, taint analysis starts from so-called taint sources – pre-defined functions, which provide special «tainted» data. For example, we may say that the result of a *read()* call will contain untrusted data and thus call it a taint source. Besides that, any value dependent on tainted data is declared to be tainted itself.

There are also so-called taint sinks – special functions or instructions which should never accept tainted data as arguments. Continuing our example, it is not safe to use values, obtained from *read()* call, as a buffer index, since this may lead to a memory corruption and a variety of other problems, thus we may call any pointer dereference instruction a taint sink.

Taint analysis is expected to report such potentially dangerous data flows for a manual or automated verification.

Taint analysis may be performed both as a part of dynamic and static analysis and each approach has its own advantages and drawbacks.

Dynamic analysis is performed during program execution and thus has low false positive rate, but it requires a lot of test runs and it could be close to impossible to explore all possible execution paths in a non-trivial program due to path explosion problem – this is important since some vulnerabilities could actually be hidden on complex execution paths, which are hard to discover using dynamic analysis or testing.

Static analysis on the contrary doesn't execute the analyzed program but processes model instead. Depending on a specific algorithm, this could enable an analyzer to explore almost all possible execution paths, which is a significant advantage in terms of security, but is also likely to increase number of false positives due to inconsistencies between program and its model.

In this work the term "taint analysis" will be used to refer to "static taint analysis" and we define taint sources as all functions providing untrusted data and taint sinks as all instruction parameters or function call arguments which may cause undesired program behavior if one allow an attacker to pick an arbitrary value for it.

We propose some extensions to the interprocedural context-sensitive taint analysis algorithm originally defined in [1]. Implementation of the algorithm is based on the LLVM compiler infrastructure and uses LLVM bitcode as an intermediate representation.

The paper is organized as follows. In Section 2 we briefly discuss the general idea of IFDS algorithm and its application to the taint analysis problem. Section 3 describes several approaches

developed to make memory model used by taint analysis more precise and our improvements of indirect calls resolution. Section 4 summarizes our attempts to decrease false positive ratio by performing additional verification step for all reported vulnerabilities. Section 5 reports experimental results. In the last section, we summarize the results of the work and present directions of future research.

2. Related Work

This section describes the characteristics of the styles used in this document.

2.1 IFDS Framework

Reps et al. [2] introduced an efficient, context-sensitive and flow-sensitive dataflow analysis framework which is able to solve a large class of interprocedural dataflow problems.

This class of problems is called IFDS (Interprocedural, Finite, Distributive, Subset) problems and consists of all dataflow problems in which the set of dataflow facts is a finite set and dataflow functions distribute over the meet operator (either ∪ or ∩). The algorithm solves an IFDS problem in a polynomial time by transforming it into a problem of reachability along interprocedurally realizable paths. The complexity of the algorithm is shown to be $O(ED^3)$ in general case and $O(ED)$ for locally separable problems, where E is the number of edges in the interprocedural control flow graph and D is the number of dataflow facts. According to the algorithm, the program should be represented as a directed super graph $G = (N, E)$, which contains a union of all functions' control flow graphs (CFG) with some special nodes and edges described below.

There is a single entry and exit node for every function in a program.

Every call statement is represented with two adjacent nodes: a call-site node and a return-site node. For every such statement there is an intraprocedural edge from call-site to the corresponding return-site node, an interprocedural edge from the call-site to the corresponding called function's entry node and an interprocedural edge to the return-site coming from the corresponding called function's exit node.

The general idea of the algorithm is to construct a directed exploded super graph $G_e = (N_e, E_e)$ with $N_e = N \times D$ set as nodes (where N is the set of super graph nodes and D is the set of dataflow facts), in which any node (s, d) is reachable from a special start node iff the dataflow fact d holds at node s.

Later several extensions to the IFDS algorithm were proposed by Naeem et al. [3], such as constructing nodes of a super graph on demand (which is important when dealing with large D sets) and exploiting existing subsumption relationships between elements of D set to perform more efficient analysis. It has been reported that these extensions are often necessary when applying the IFDS algorithm to non-separable problems, such as alias set analysis.

2.2 IFDS based taint analysis

Flowdroid [4] is one of the most well-known implementations of the IFDS framework for data leaks detection in Android applications. It demonstrates a possibility to perform taint analysis in terms of IFDS framework and also explains how to combine on-demand backward alias analysis with a regular forward taint analysis. In Flowdroid, dataflow set D is defined as the set of access paths, plus a special «true» fact $[\emptyset]$. An access path consists of a base value (such as a local variable or parameter) with potentially empty ordered list of fields and could be written e.g. like $x.f.g$, where x is the name of the base object, f is the name of the dereferenced field of x object and g is the name of the dereferenced field of $x.f$ object.

Dataflow fact x holds at node s iff an object, which is accessible through this access path at s, may contain tainted data here. x being tainted implies the fact that all object, accessible through this object (such as already mentioned $x.f.g$), are also considered tainted.

3. Taint Analysis Stage

Our experience with the development of vulnerabilities detection tool based on taint analysis shows that resulting warnings contain a lot of false positives. Particular results evaluation makes it possible to discover 2 main roots of the problem: path-insensitivity and inaccurate sizes of tainted objects. It's unclear how to resolve both issues without complex memory model, which would allow to build path and object size conditions. We deal with it by using external symbolic execution engine, forcing it to execute the exploded graph subset corresponding to any specific warning.

Initial warning set is generated by the tool based on [1] and is greatly inspired by Flowdroid design. It uses LLVM as intermediate representation. Due to a low-level nature of LLVM bitcode, we use another definition of access path: an access path consists of a base value (which is an actual LLVM value) and an ordered list of dereference offsets. This definition is suitable for referencing both structure fields (since in LLVM bitcode it is possible to calculate a fixed offset for any structure field) and memory locations, accessible with a help of simple pointer arithmetics. In this paper we will use *[pointer, offset]* to denote value, accessible through dereference of pointer value plus offset bytes, *[integer]* to denote value of the integer, and *[Ø]* to denote a special «true» fact. It is possible to specify a list of offsets to define a sequence of consecutive dereferences.

Let's consider an example on fig.1, written in C language.

```
1   extern void *a;
2   extern void *b;
3   void source(int *pointer) {
4       scanf("%d", pointer);
5   }
6   void sink(int size) {
7       memcpy(a, b, size);
8   }
9   void foo(int *t) {
10      source(t);
11      sink(*t);
12  }
```

Fig. 1. Example of a program with interprocedural taint flow

Omitting insignificant details, it is possible to say that

- *scanf* function call on line 4 is a taint source, since it changes the value pointed to by the *pointer* parameter to an arbitrary value chosen by the user;
- *memcpy* function call on line 7 copies *size* bytes from the object pointed to by *b* variable to the object pointed to by *a* variable. We may call it a taint sink, since it is dangerous to specify *size* values greater than actual size of objects pointed to by *a* or *b*;
- *source* function's entry-to-exit subgraph can be summarized with the path edge *(source-Entry,[Ø]) → (source-Exit,[pointer,0])*, i.e. value of *[pointer,0]* becomes unconditionally tainted;
- *sink* function's subgraph can be summarized with the path edge *(sink-Entry,[size]) → (Sink,[size])*, i.e. tainted data would reach a taint sink if the value of *[size]* is known to be tainted at the entry of the sink function;
- *foo* function's subgraph can be summarized as *(foo-Entry,[Ø]) → (source-callsite,[Ø]) → (source-retsite,[t,0]) → (sink-callsite,[t,0]) → (Sink,[size])*, i.e. tainted data unconditionally reaches a taint sink.

Unlike in known implementations and Flowdroid we don't store the whole exploded super graph, because it requires too much memory for regular industrial project with millions of lines of code

even if this graph is constructed on-demand. To solve this issue another analysis mode was developed, which doesn't require exploded super graph edges to be constructed. The existing IFDS analysis engine was supplemented with function summaries (similar to [3]) and explicit taint traces, which makes it possible to show user where the tainted data originate from and how did it get to the taint sink without the need to store the graph itself.

As we noticed during analysis of selected open source projects, taint sources are usually located far from each other in a program and thus their taint flow subgraphs are rarely intersecting. To decrease memory consumption, we have added an ability to run a separate analysis for every taint source. Therefore, exploded graph nodes and summaries can be cleared, but the total analysis time could increase since parts of the program graph can be potentially analyzed more than once.

The second significant difference with other implementations is that each request for a set of aliases for any specific tainted value is handled in a separate local environment with its own IFDS solver and exploded super graph. The graph is cleared after the call and only the resulting aliases set is preserved so that it could be reused both in main taint analysis and when calculating other aliases sets.

Remaining improvements are discussed in following subsections 3.1 and 3.2 with more details.

3.1 Unresolved function calls

When examining a call site, we assume that it is trivially easy to determine the called function by the generated bitcode. Unfortunately, C and C++ programs contain calls, whose targets couldn't be determined until runtime. There are three main sources of such unresolved calls, described below:

1) virtual functions;
2) external function;
3) indirect calls.

C++ virtual function is a member function declared within a base class and overridden by the method in the derived class. When performing a virtual call, the called function is determined by the actual type of the object and may vary in runtime. Such calls were already handled in the previous implementation of the algorithm [1] by adding interprocedural edges to all possible overrides.

Another case of a call with unknown called function is an external call. There are two main kinds of external calls: library functions and system calls. In both cases the analyzed program doesn't contain called functions' definitions and thus we cannot add any "call-site to entry" and "exit to return-site" edges to the call and has to rely on "call-site to return-site" intraprocedural edge only.

By default, we assume that an external function can change values of all its arguments, unless it contradicts with language semantics, so the corresponding facts are not propagated further. We also assume that external function doesn't change value of any global variable and leave it tainted. Usually such assumptions lead to an undertainting and thus we've developed several ways to deal with this issue.

1) Since there is a limited number of system functions and most of them are well-documented, it is possible to create summaries (models) for most frequently used ones manually. Our tool also provides the list of external functions encountered during analysis, which makes it possible for user to prepare summaries for them. It's also possible to specify custom sources, sinks and propagators using similar files in JSON format – those summaries are applied at "call-site to return-site" edges.
2) For an open-source library it is possible to compile it into LLVM bitcode and then link it together with the analyzed program's representation using llvm-link program, which is a part of LLVM infrastructure.
3) If some specific parameter of an external function has type with const qualifier, which specifies that its value should remain unchanged after invocation, we derive a rule for

propagating taint through the corresponding argument in every call of this function. Unfortunately, a lot of type-related information, including constancy, may be lost during compilation, so we had to add several modifications to the Clang compiler in order to store this information in the LLVM metadata.

Lastly, there is another type of calls where the memory address of the called function is calculated at runtime – such calls are named indirect calls. It is possible to say that a virtual call is just a special case of such indirect call. C developers sometimes use indirect calls to simulate virtual calls functionality available in C++ and thus it could be important to support such calls. For example, such technique is used in security-critical OpenSSL library.

We've evaluated several ways to handle indirect calls, described below. Firstly, we make the following assumptions for an indirect call:

a) the indirect call is never used for invocation of any external function, thus the analyzed program contains definitions for all possible candidate functions;
b) the set of all possible candidates is completely determined by the program itself, which means that for all possible target function there is a path in the program where the address of the given target is taken and transferred to the indirect call instruction.

If (A) is considered to be false, such indirect call is actually an external call and should be handled as appropriate, otherwise we can examine following approaches.

1) The naive approach is to take every function definition with compatible parameter types and consider as a candidate. The problem of this approach is that all functions with 0 parameters are indistinguishable and most functions with 1-2 parameters are divided into several large clusters. If we also assume (B) to be true, this approach could be slightly improved by excluding functions, whose address was never explicitly taken in the program.
2) Another approach relies on the assumption that both functions and variables in a program are usually named according to their semantics. In this case it should be possible to compare similarity between call instruction and different call candidates to choose the most likely called function. Unfortunately, the function and variable names are virtually never plain equal and while it should be possible to write an automated heuristic, its results would be unreliable due to lack of formal specifications regarding naming. Such a heuristic would need to put «encrypt string», «EncryptString», «EncryptUTF», «encstr» names into the same similarity cluster, but differentiate between «encrypt» and «decrypt» function names. Common abbreviations, such as «Context – ctx» and «Source – src» may also pose a problem. Right now, we are using a semi-automated solution, where the naive approach is used to generate a .txt file with «expression name» → «{called function names}» mapping, which can be filtered by a user for further analysis runs.
3) Assuming both (A) and (B) are true, it should be possible to implement an interprocedural backward-dataflow analysis to find possible candidates for an arbitrary indirect call. We suppose that one of the problems of such analysis is that its results are used to add missing interprocedural edges to the super graph, but at the same time they are dependent on the super graph structure, thus it should be performed as an iterative process. We don't have a working implementation of this approach by now.

3.2 Memory model

As it was already mentioned in Section 3, we use an access path-based memory model with an access path defined as a combination of base value and an ordered list of dereference offsets.

In the current implementation, offsets are represented with either constant integers or a special λ-offset, which is used to denote an unknown offset. This λ-offset has a special behavior: given any pointer *ptr* and a constant offset *a*, access path *[ptr,λ]* is considered to be subsuming *[ptr,a]*, i.e.

the statement `b = ptr[a]` would propagate taint from either one of these access paths to *[b]*, but the statement `ptr[a] = 0` would remove taint from the second one only.

Usage of arbitrary integer offsets instead of object fields in access paths, what is required to achieve complete support of all C++ features, leads to an extremely large *D* set, which has a great impact on worst-case complexity of the algorithm. Thus, it is necessary either to make sure that transfer functions would work with a limited subset of *D* for any given program, or to limit path length in the exploded super graph.

While this special offset enables us to give a simple and efficient representation for complex expressions like `s->packet+len+left`, where `s->packet` is a pointer and `len` and `left` are non-constant offsets, it inevitably leads to an overtaint problem, because λ-offset doesn't restrict the set of possible values and any two λ-offsets are considered to be equal. While it is possible to extend this model to achieve more precise analysis, it's required to keep reasonable size of the *D* set. We've evaluated following approaches.

1) A relatively simple extension of the model is to introduce "unknown non-negative" offset λ+, which is included into "unknown" λ-offset. Let us assume that an instruction writes tainted data into a single element with unknown index of an array field of an object. As a result the whole object becomes tainted, because resulting access path will be equal to [this,offsetof(field) + λ] = [this,λ]. If the used index is nonnegative, because it corresponds to an unsigned variable, it's possible to achieve better precision, tainting only consequent part of the object with the help of λ+. This approach has allowed us to slightly decrease number of false positives on LibTIFF library, but it hasn't proved to make any difference in other cases, since buffers are usually accessed via pointers and this situation seems to be rather an exception.
2) The access path is the core of used memory model. It's possible to use interval domain to get better granularity and precision. The model requires to define several predicates, such as that one access path corresponds to the memory region included into region of another access path. Since access path construction and predicate calculation for interval domain is significantly slower and the total number of created access paths (the size of the D set) grows significantly too, the total analysis time becomes unacceptable.
3) We also tried to implement another extension of memory model which uses symbolic expressions instead of integer offsets in access path. But this approach requires to gather constraints for these offsets using LLVM-instructions which are usually ignored by the IFDS engine, because values of these variables are not tainted. Therefore, it's better to build a dedicated symbolic execution engine and integrate it with existing IFDS engine or to build taint analysis immediately on symbolic execution [5]. Because of this we decided to use an existing symbolic execution tool as a second analysis stage.
4) We also considered using a region-based memory model, similar to the one proposed in [6], since that would allow us to merge all aliases in a single data fact, instead of propagating them independently. We don't have a working implementation of this approach.

4. Analysis Results Refinement Stage

One of the likely reasons of false positives is the lack of path-sensitivity in the IFDS algorithm. Let's consider an example based on a typical buffer overflow test from the Juliet Test Suite for C/C++ on fig.2.

There is a single taint source *fscanf(data)* and a single taint sink *buffer[data]*. Due to path-insensitivity of the algorithm, it reports a dangerous data flow between those instructions, but in reality there is no realizable path from source to sink, because that would require variable *globalFive* to be equal to 5 and not to be equal to 5 at the same time.

There are different ways to solve this issue. We propose performing a two-stage analysis: the first step is done by a relatively fast and simple analyzer, which is able to detect most errors in the

program but also produces a high amount of false positives and a second one is performed by a slower but more precise path-sensitive analyzer, whose task is to confirm or reject reports from the first stage.

Similar two-staged approach, consisting of static and guided dynamic analysis was proposed for example in [7] [8]. Preliminary static analysis helps to avoid path explosion problem in dynamic analysis, since it is necessary to check only those execution paths, which were already discovered by the first stage.

```
1   extern int globalFive;
2   int data = -1;
3   if(globalFive == 5) {
4       fscanf(stdin, "%d", &data);
5   }
6   if(globalFive != 5) {
7       int buffer[10] = { 0 };
8       if (data >= 0) {
9           buffer[data] = 1;
10      }
11  }
```

Fig. 2. Example for path insensitivity problem

We propose an analogical combination of two static analyses: IFDS-based analysis and symbolic execution. Unlike building taint analysis using static symbolic execution without IFDS framework, as we have already done for C# in [5], or as it is done for C and C++ in Svace [9], two-staged approach allows to deal with following issues. Every general-purpose static analyzer is required to balance between analysis precision and performance. We can enable very detailed and precise analysis because it's necessary to handle only minimal amount of possible dangerous paths, found by the previous stage. Hence states explosion problem together with conditions simplification for analyzer with states merging are solved.

We decided to use an existing symbolic execution engine for a second stage and the main requirement was that it should work with program representation in LLVM bitcode format to ease exchanging data between stages.

As an experiment, we consider a simpler form of report confirmation – a path confirmation. In this case the only task of the second analyzer is to confirm that the source-sink path is realizable, and it doesn't need to know anything about the vulnerability itself. Hereafter we plan to build more precise condition for each type of detected error. For example, considering usage of tainted data as an array index, we can ensure proper sanitizing by building condition to check if the index is out of bounds.

4.1 KLEE

KLEE [10] is a well-known open-source symbolic execution engine which is actively developed since 2008 and has more than one hundred related publications [11].

It analyses programs in LLVM format and is able to mix both concrete and symbolic execution. To enable symbolic execution, the program should be explicitly annotated with special functions, which are used to mark symbolic values to be created, conditions to be checked etc. This should be done either manually, or by linking the program with a special implementation of system libraries, such as *uClibc* [12].

We have tested a simple way to transfer information about taint sources and sinks in program to KLEE by instrumenting bitcode file with necessary special functions calls.

The path confirmation problem was modeled as follows.

Every warning trace contains an ordered list of instructions (tracepoints) in a program along the path from source to sink, which are important to demonstrate to user the taint flow. For every tracepoint except the last one corresponding to the sink, a special global variable *tracepoint_i_j* is created, where *i* is the index number of the current report trace and *j* is an index number of the tracepoint.

At the first tracepoint of every trace *i* we insert an LLVM instruction, corresponding to the assignment

```
tracepoint_i_1 = 1;
```

At every other tracepoint j of the trace i we insert LLVM instructions, corresponding to the code

```
if (tracepoint_i_(j-1))
    tracepoint_i_j = 1;
```

At the sink we insert an equivalent of the code

```
if (tracepoint_i_(j-1))
    klee_report_error(...);
```

In these terms, the error would be reported iff KLEE has found a realizable path which visits all tracepoints of the trace in a proper order.

Unfortunately, while the concept seemed to be working for simple test cases (and even there were some difficulties with external functions), the general idea has proved to be not so easy to properly implement.

In particular, we encountered following issues.

- KLEE doesn't support memory regions with symbolic size. It means that it's necessary to explicitly specify size for every input string and memory buffer, which is inappropriate for us. This problem is addressed in [13].
- KLEE as is can't start analysis from an arbitrary point of the program. It is acceptable for tests generation, but it is not very suitable for our purpose, since we are interested in simulating of a relatively small subpath from source to sink. In addition, many libraries don't have an entry point at all. The problem is addressed in [14].
- By design KLEE uses program traversing strategy which is aimed to increase code coverage. However, we were not satisfied with the existing "covering-new" heuristic and would need to implement another one for directed symbolic execution similar to [15].
- By default, KLEE works on self-contained isolated programs that don't use any external code (e.g. C library functions), but in practice most programs use external functions calls. To solve this issue, it is possible to link the program with the library or model representation or to automatically generate stub definitions for unknown functions.

After successful experiments on artificial tests, we've tried KLEE to automatically confirm our reports on the relatively small library LibTIFF, but we haven't managed to find a way to reach even the taint source.

As a conclusion, we've decided that it would be too hard to adapt this tool to our problem and it is better to use a static analysis approach for now.

4.2 Svace

Svace [16] is a static analysis tool for bug detection developed at the Institute for Systems Programming, Russian Academy of Sciences. It supports analyzing program written in various programming languages, including C, C++, C# and Java.

Unlike the previous tool, it performs purely static analysis, which means that it doesn't necessarily require a full model of the analyzed program and is suitable to analyze libraries without executable files.

We started with creating a symbolic execution based checker for Svace, which analyses the modified LLVM bitcode file to confirm the existence of a realizable source-sink path for traces of warnings reported by the first stage.

For the proof of concept, report traces are represented in a following manner.

1) Temporarily we don't use any tracepoints, other than the first corresponding to the source and the last one corresponding to the sink.
2) All unique sources appearing in any reported source-sink pair are sorted and enumerated. If tainted data from the source haven't reached any sink, such source is ignored.
3) For every source, a global variable source i tainted is created, where i is the source's index number.
4) A special function call taint_variable(source_i_tainted) is inserted after every source statement in the program to tell the analyzer that the variable is tainted.
5) A special function call check_tainted(source_i_tainted) is inserted before every sink statement appearing in a source-sink pair, where i is the corresponding source's index number.

For every function containing either *taint_variable* or *check_tainted* call, a summary is created. Summary contains intraprocedural reachability condition of the corresponding source or sink. Similar summaries are created for every caller function and contain conjunction of call reachability condition and condition from the callee translated into the caller context. Error condition is equal to the conjunction of the current path condition, the source reachability condition and the sink reachability condition. The resulting condition is passed to a solver for every function call, containing sink. If the resulting condition is UNSAT checker classifies corresponding report as false positive.

Hereafter we plan to check the reachability condition of the whole path or set of paths, instead of source to sink subpath. For example, the entry point can be considered as the entry of nearest function containing source to sink path. We also want to filter out sanitized taints by checking corresponding security conditions.

5. Testing Results

First of all, we performed empirical evaluation of some of the memory usage enhancements mentioned in Section 3. We have launched analysis 4 times on *libssl* library from OpenSSL version 1.0.1f with following configurations:

1) baseline configuration, with most enhancements disabled;
2) current default configuration;
3) default configuration without separate sources analysis;
4) default configuration with full exploded graph instead of currently used taint traces.

This library contains 3 taint sources and was chosen for the demonstration because it contains the well-known «Heartbleed» (CVE-2014-0160) vulnerability, which was successfully found by the analyzer. Also, we have to mention the fact that baseline configuration exceeds 20 Gb RAM usage limit on a full *openssl* executable – it contains 162 taint sources and has a huge taint flow graph mostly because of extensive use of cryptographic library *libcrypto*. Thus, mentioned enhancements seem to be necessary for the analysis of programs with vast taint flow graphs (Table 1).

Table 1. Evaluation of memory consumption

Run	# Reports	# Iterations	Time	Memory
1	75	3 614 thous.	45 s	938 MB
2	75	3 619 thous.	55 s	318 MB
3	75	3 614 thous.	53 s	367 MB

4	75	3 619 thous.	45 s	580 MB

Another launch without «*const*» qualifier handling in external function calls mentioned in Subsection 3.1 showed decrease in amount of reported warnings from 75 to 67, amount of covered functions from 141 to 136 and decrease in amount of algorithm iterations performed from 3.61 millions to 3.02 millions.

Regarding two-stage analysis concept, we performed an evaluation on the set of artificial tests, which was created during development of the first stage analyzer. While these tests were not designed to test path confirmation, that allowed us to find some obvious implementation flaws and compare analysis time of both stages.

The first stage analyzer has been launched for every test from the set with up to 4 tests being analyzed simultaneously.

On the next run it was supplemented by the second stage analyzer, which has been launched for 176 tests from the set in which first stage analyzer produced at least a single report to be confirmed (Table 2).

Table. 2. Evaluation of analysis time on artificial tests.

Stage	**# Tests**	**# Passed**	**Time**
First	272	269	1m 32s
Both	176	168	18m 36s

Out of 8 tests, incorrectly filtered out by the second stage analysis:

- were caused by the lack of indirect and virtual calls support in the second stage analyzer
- were caused by incorrect interpretation of traces produced by a backward analysis checker
- 2 has failed because of merged or duplicated reports, which seems to be an implementation issue

The substantial slowdown of the second stage analyzer is most likely caused by the fact that Svace is a general-purpose tool and performs a lot of actions which are not necessary for the path confirmation checker. Also, it requires more time to bootstrap and initialize analysis, which makes difference because every test file was analyzed in a separate instance.

We've also checked two-stage approach on Juliet 1.3 test suite for C/C++ [17] with and without work-in-progress Svace checker. The first stage was launched on a program which consists of all unix tests from directories, corresponding to CWE121, CWE122, CWE124, CWE126 and CWE127. There were 2688 taint sources in the analyzed program. Many tests from the set are ignored by the analyzer because they don't contain any taint sources and use a hardcoded invalid index instead.

Then we tried to confirm the results from the first stage with two versions of second stage analyzer: with path confirmation described in subsection 4.2 and another one, which also tries to filter out sanitized taint paths (marked with *, see Table 3).

It should be noted that the first stage analyzer is not yet able to utilize more than a single thread, while the second stage analyzer was allowed to use up to 4 threads during analysis, hence the difference in analysis time and memory consumption.

Table. 3. Evaluation of two-stage analysis on the Juliet Test Suite 1.3 for C/C++.

Stage	**# Reports**	**True positive rate**	**Time**	**Peak Memory**
First	2424	41%	16m 13s	1.9 GB
Second	2424	41%	13m 10s	8.9 GB
Second*	984	100%	14m 9s	9.8 GB

We don't have a working solution for security conditions checking right now, but for the purpose of proof of concept, we've implemented a simple addition to the current path confirmation checker, which should check that all LLVM values corresponding to the access paths from the reported trace are able to have arbitrary high or negative values within the current data type – otherwise such a taint path is filtered out. This is not a proper implementation of security conditions checking, but is enough to demonstrate filtering out most false positives on this particular test suite.

As it can be seen from the evaluation data, path confirmation checker alone is not enough to improve analysis results on the selected test suite, because it doesn't contain tests with unrealizable paths between source and sink. However, if supported with a security conditions checker, this approach should be able to significantly decrease number of false positives among reported warnings.

It is also interesting, that while Svace has its own set of buffer overflow and tainted data checkers which cover much greater set of test cases in the test suite, 152 out of our 984 reports seem not to be reported by Svace's own checkers.

We've also tested two-stage analysis on the *libssl* library, which contains "Heartbleed" vulnerability.

First stage analyzer was able to find the taint path from the *BIO_read* call in function *ssl3_read_n* to the *memcpy* call in *dtls1_process_heartbeat*, but also produced more than 70 other reports, which are most likely false positives.

After increasing default limits on procedure analysis time and max annotation size, the second stage analyzer was able to reduce total number of reports to 62 (48), while keeping the true positive report (Table 4).

Table. 4. Evaluation of two-stage analysis on the libssl library from OpenSSL 1.0.1f .

Stage	**# Reports**	**"Heartbleed" found?**	**Time**	**Peak Memory**
First	73	+	1m 51s	0.4 GB
Second (default limits)	1	-	2m 27s	3.3 GB
Second	62	+	5m 29s	6.6 GB
Second*	48	+	5m 42s	6.9 GB

In case of LibTIFF library and CVE-2018-15209 vulnerability, we could not confirm the true positive taint path with the second stage analyzer, because that would require handling of indirect calls which is not yet supported by the checker.

Therefore, our testing shows that the concept seems to be promising, but still requires further refinement.

6. Conclusion

Performing taint analysis for vulnerability detection via pure IFDS approach has several limitations in comparison to existing buffer overflow checkers, such as [9]: it doesn't make any assumptions about buffer size and is unable to detect several cases even from Juliet Test Suite, because there are no taint sources. For example, if a constant array index is used to access memory outside of array bounds. Moreover, considering error detection problem, pure static taint analysis generates too many alarms to be able to find few vulnerabilities uncaught in an industrial project. The majority of false alarms are introduced by path-insensitivity and overtainting due to inconsistencies between a program and its memory model. Our experience shows that addons to simple and efficient memory model used by IFDS lead to unreasonable analysis slowdown and offer just a minimal results improvement. Therefore, a postprocessing of results is required.

Proposed approach with two-staged analysis looks promising but requires a lot of enhancements to achieve industrial level quality and there are no guaranties that it is even possible.

We are going to continue our research by developing other types of report confirmation in Svace infrastructure, since despite all listed limitations, the tool has a potential to discover serious vulnerabilities, such as «Heartbleed» in OpenSSL and CVE-2018-15209 in LibTIFF.

References

[1] Koshelev V.K., Izbyshev A.O., Dudina I.A. Interprocedural taint analysis for LLVM-bitcode. Programming and Computer Software, 2015, vol. 41, issue 4, pp. 237-245. DOI: 10.1134/S0361768815040027.

[2] Reps T., Horwitz S., Sagiv M. Precise interprocedural dataflow analysis via graph reachability. In Proc. of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1995, pp. 49–61.

[3] Naeem N.A., Lhoták O., Rodriguez J. Practical extensions to the IFDS algorithm. In Proc. of the international conference on Compiler Construction, 2010, pp. 124–144.

[4] Arzt S., Rasthofer S., Fritz C., Bartel A., Klein J., Traon Y.L., Octeau D., McDaniel P. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In Proc. of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2014, pp. 259–269.

[5] Belyaev M.V., Shimchik N.V., Ignatyev V.N., Belevantsev A.A. Comparative analysis of two approaches to static taint analysis. Programming and Computer Software, 2018, vol.44, issue 6, pp. 459-466. DOI: 10.1134/S036176881806004X.

[6] Xu Z., Kremenek T., Zhang J. A memory model for static analysis of C programs. In Proc. of the International Symposium On Leveraging Applications of Formal Methods, Verification, and Validation. 2010, pp. 535–548.

[7] Gerasimov A.Yu., Kruglov L.V., Ermakov M.K., Vartanov S.P. An approach of reachability determination for static analysis defects with help of dynamic symbolic execution. Programming and Computer Software, 2018, vol. 44, issue 6, pp 267-275. DOI: 10.1134/S0361768818060051.

[8] Gerasimov A.Yu. Directed dynamic symbolic execution for static analysis warnings confirmation. Programming and Computer Software, 2018, vol. 44, issue 5, pp. 316–323. DOI: 10.1134/S036176881805002X.

[9] Dudina I.A., Belevantsev A.A. Using static symbolic execution to detect buffer overflows. Programming and Computer Software, 2017, vol. 43, issue 5, pp. 277–288. DOI: 10.1134/S0361768817050024.

[10] Cadar C., Dunbar D., Engler D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In Proc. of the Proceedings of the 8th USENIX conference on Operating systems design and implementation, 2008, pp. 209–224.

[11] Publications•KLEE. [Online]. Available at: http://klee.github.io/publications/, accessed 20.03.2019.

[12] GitHub - klee/uclibc: KLEE's version of uClibc. [Online]. Available at: https://github.com/klee/klee-uclibc, accessed 02.04.2019.

[13] Šimáček M. Symbolic-size memory allocation support for Klee. Master's thesis, Masaryk University, Faculty of Informatics, Brno, 2018. [Online]. Available at: https://is.muni.cz/th/mdedh/, accessed 21.03.2019.

[14] Ramos D.A., Engler D. Under-constrained symbolic execution: Correctness checking for real code. In Proc. of the Proceedings of USENIX Security Symposium, 2015, pp. 49–64.

[15] Marinescu P.D., Cadar C. KATCH: High-coverage testing of software patches. In Proc. of the 2013 9th Joint Meeting on Foundations of Software Engineering, 2013, pp. 235–245.

[16] Ivannikov V.P., Belevantsev A.A., Borodin A.E., Ignatiev V.N., Zhurikhin D.M., Avetisyan A.I. Static analyzer Svace for finding defects in a source program code. Programming and Computer Software, 2014, vol. 40, issue 5, pp. 265–275. DOI: 10.1134/S0361768814050041.

[17] Software assurance reference dataset. [Online]. Available at: https://samate.nist.gov/SARD/testsuite.php, accessed: 20.03.2019

Информация об авторах / Information about authors

Никита Владимирович ШИМЧИК – аспирант Института системного программирования им. В.П. Иванникова РАН. Его научные интересы включают статический анализ программного обеспечения.

Nikita Vladimirovich SHIMCHIK is a postgraduate student of Ivannikov Institute for system programming RAS. His research interests include static analysis of programs.

Валерий Николаевич ИГНАТЬЕВ, кандидат физико-математических наук, старший научный сотрудник Института системного программирования им. В.П. Иванникова РАН, старший преподаватель кафедры системного программирования факультета вычислительной математики и кибернетики МГУ им. М.В. Ломоносова. Научные интересы включают методы поиска ошибок в исходном коде ПО на основе статического анализа.

Valery Nikolayevich IGNATYEV, PhD in computer sciences, senior researcher at Ivannikov Institute for system programming RAS and senior lecturer at system programming division of CMC faculty of Lomonosov Moscow State University. He is interested in techniques of errors and vulnerabilities detection in program source code using static analysis.

DOI: 10.15514/ISPRAS-2019-31(3)-15

C# parser for extracting cryptographic protocols structure from source code

I.A. Pisarev, ORCID: 0000-0002-2055-1841 <ilua.pisar@gmail.com>
L.K. Babenko, ORCID: 0000-0003-2353-7911 <lkbabenko@sfedu.ru>
Southern Federal University, Department of Information Security,
Taganrog, Rostov region, 347928, Russia

Abstract. Cryptographic protocols are the core of any secure system. With the help of them, data is transmitted securely and protected from third parties' negative impact. As a rule, a cryptographic protocol is developed, analyzed using the means of formal verification and, if it is safe, gets its implementation in the programming language on which the system is developed. However, in the practical implementation of a cryptographic protocol, errors may occur due to the human factor, the assumptions that are necessary for the possibility of implementing the protocol, which entail undermining its security. Thus, it turns out that the protocol itself was initially considered to be safe, but its implementation is in fact not safe. In addition, formal verification uses rather abstract concepts and does not allow to fully analyze the protocol. This paper presents an algorithm for analyzing the source code of the C# programming language to extract the structure of cryptographic protocols. The features of the implementation of protocols in practice are described. The algorithm is based on the searching of important code sections that contain cryptographic protocol-specific constructions and finding of a variable chain transformations from the state of sending or receiving messages to their initial initialization, taking into account possible cryptographic transformations, to compose a tree, from which a simplified structure of a cryptographic protocol will be extracted. The algorithm is implemented in the C# programming language using the Roslyn parser. As an example, a cryptographic protocol is presented that contains the basic operations and functions, namely, asymmetric and symmetric encryption, hashing, signature, random number generation, data concatenation. The analyzer work is shown using this protocol as an example. The future work is described.

Keywords: cryptographic protocols; C#; parser; verification; tree; analysis; source code

For citation: Pisarev I.A., Babenko L.K. C# parser for extracting cryptographic protocols structure from source code. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 3, 2019. pp. 191-202. DOI: 10.15514/ISPRAS-2019-31(3)-15

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

C# парсер для извлечения структуры криптографических протоколов из исходного кода

И.А. Писарев, ORCID: 0000-0002-2055-1841 <ilua.pisar@gmail.com>
Л.К. Бабенко, ORCID: 0000-0003-2353-7911 <lkbabenko@sfedu.ru>
Южный федеральный университет, Кафедра информационной безопасности,
Таганрог, Ростовская область, 347928, Россия

Аннотация. Криптографические протоколы являются ядром любой защищенной системы. С их помощью передаются данные, которые нуждаются в защите от третьих лиц. Как правило, криптографический протокол разрабатывается, анализируется с использованием средств формальной верификации и, если он безопасен, реализуется на языке программирования, на котором

разрабатывается система. Однако при практической реализации криптографического протокола могут возникать ошибки из-за человеческого фактора, предположений, которые необходимы для возможности реализации протокола, что влечет за собой подрыв его безопасности. Таким образом, оказывается, что сам протокол изначально считался безопасным, но его реализация на самом деле небезопасна. Кроме того, формальная верификация использует довольно абстрактные понятия и не позволяет полностью проанализировать протокол. В данной статье представлен алгоритм анализа исходного кода языка программирования C# для извлечения структуры криптографических протоколов. Описаны особенности реализации протоколов на практике. Алгоритм основан на определении ключевых областей кода, содержащих специфические для криптографических протоколов конструкции, и определении цепочки преобразований переменных из состояния отправки или получения сообщений до их начальной инициализации с учетом возможных криптографических преобразований для составления дерева, из которого будет извлечена упрощенная структура криптографического протокола. Алгоритм реализован на языке программирования C# с использованием синтаксического анализатора Roslyn. В качестве примера представлен криптографический протокол, который содержит основные операции и функции, а именно: асимметричное и симметричное шифрование, хеширование, подпись, генерация случайных чисел, конкатенация данных. Работа анализатора показана с использованием этого протокола в качестве примера. Описана будущая работа.

Ключевые слова: криптографические протоколы; C#; парсер; верификация; дерево; анализ; исходный код.

Для цитирования: Писарев И.А., Бабенко Л.К. C# парсер для извлечения структуры криптографических протоколов из исходного кода. Труды ИСП РАН, том 31, вып. 3, 2019 г., стр. 191-202 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(3)-15

Благодарность. Работа выполнена при поддержке Министерства образования и науки Российской Федерации, грант № 2.6264.2017/8.9.

1. Introduction

The problem of verifying the security of cryptographic protocols is relevant nowadays despite the existence of a large number of already verified protocols. The need to use self-written protocols that use lightweight cryptography for IoT, mobile robots, as well as the imperfection of formal verification of protocols is a new challenge for verification methods, in particular, the possibility of verifying the security of cryptographic protocols implementation. Nearly all protocols are changed and supplemented during implementation, and for their initial analysis, for example, by means of formal verification this is not taken into account. Also there can be programming mistakes and logic flaws on source code. So we need verify cryptographic protocols on their last developing iteration - on implementation level for more attack finding which can help make any system more secure. Due to this fact this work is actual nowadays. The primary task in this matter is to extract the structure of the protocol from the source code. At the moment there are works in which the problem of extracting an abstract model from the source code of programming languages C [1-3], Java [4-6], F# [7-12] is being considered. Most of them require a special programming style for the possibility of use these algorithms or the use of additional annotations in the source code. The paper proposes to analyze the source code of the C# programming language. There are no other works, in which code analysis would be carried out, not involving the use of annotations or a special programming style.

2. Cryptographic protocols

Cryptographic protocols are a set of cryptographic algorithms and functions, with a correct combination of which is obtained a secure process of transferring messages between the parties. Protocol security is defined as complying with security requirements, the main of which are mutual authentication of the parties, protection against time attacks such as replay attacks, privacy and integrity of the transmitted data. Below is an example of a test protocol that does not have a

special meaning, but contains all the basic cryptographic algorithms and functions: asymmetric and symmetric encryption, hashing, signature, random number generation.

1. $A \to B: E_{pkB}(A, Na)$
2. $B \to A: E_{pkA}(Na, Nb, B)$
3. $A \to B: E_{pkB}(Nb, k)$
4. $B \to A: E_k(M1, E_{pkA}(M2)), Hash(M1)$
5. $A \to B: E_k(M1, M2, M3), Sign_{skA}(M1, M2, M3)$
6. $B \to A: E_k(M3)$

At the beginning of this protocol, messages 1-3 use the Needham–Schroeder public key protocol (NSPK) [13] for mutual authentication of the parties. In message 3, in addition to the random number Nb, the key k is also transmitted for further communication between the parties using a symmetric cipher. In message 4, $M2$ data is transmitted, asymmetrically encrypted on partys' A public key, and some $M1$ data. All this is encrypted symmetrically using the key k, after which the data hash $M1$ is applied. In message 5, side A applies its $M3$ data to the previously sent data $M1$ and $M2$, encrypts all this symmetrically on key k, applies a signature and sends this message to side B. In message 6, B sends A $M3$ data encrypted symmetrically on key k.

3. Features of the cryptographic protocols implementation

There are a number of problems with the implementation of cryptographic protocols. One of the problems is the dynamic size of messages. In the programming language, the transfer of messages between the parties is implemented using sockets. In this case, the party that receives the message must know in advance the size of the buffer to receive. For example, in the protocol described in the previous paragraph, in the first three messages random numbers and identifiers of the parties with a fixed length are used. In this case, everything is simple and at the reception of the message by the party, it will expect a previously calculated static message length. However, messages 4-6 use data $M1, M2, M3$, which may have different lengths. For example, in message 4, $M1$ data can be a video file, the length of which can vary from 1 MB to several GB. And the question is how to tell the receiving party the size of the receiving buffer. There are various options for how this can be done, for example, to add information about its length to the beginning of a message, to put a mark at the end of the message. Let us consider in more detail the option with the addition of information about the length of the message. This option involves the use of additional data before the main message, which will contain the size of the future message. An example of a message with additional size information is shown in fig. 1.

Fig. 1. Additional information about the size of the message

The receiving party in this case receives a fixed array of bytes, which contains the size of the message, after which the second portion takes the rest of the message knowing in advance its length.

```
A send: Buffer size, Message
B receive(4 bytes): Buffer size
B receive(Buffer size): Message
```

Since *Message* is usually encrypted and, in the context of a protocol, its transmission is protected, the question arises of how to protect information in *Buffer size*. All security requirements are important for us, except secrecy. To ensure them, you can, for example, use the signature of this area with timestamps. Thus, the transmission, for example, message 4, will have the following form when implementing the protocol:

$B \to A: Buffer\ size, T, Sign_{skB}(Buffer\ size, T),\ E_k(M1, E_{pkA}(M2)), Hash(M1)$

Another way is to get data into a fixed-length buffer until the buffer becomes empty. In this case, problems can also arise as shown in fig. 2.

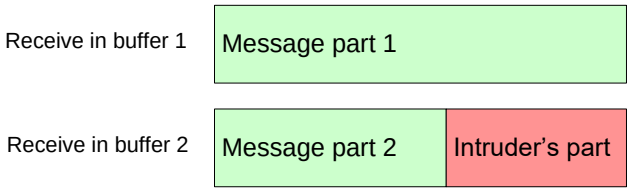

Fig. 2. Intruders' attack on the addition of real data

The result is that the message will be received longer than necessary and in some implementations, in which further processing of the message by the receiving party is tied to the use of the message length, some data may be imperceptibly corrupted when decrypting and dividing the data into the message elements (random numbers, keys, etc.). In order to avoid this, various methods of controlling the length of a message are also used.

4. Source code analysis algorithm

As an example for describing the operation of the algorithm, the previously considered protocol was taken and implemented in the C# programming language in the form of a client server application.

1. $A \rightarrow B: E_{pkB}(A, Na)$
2. $B \rightarrow A: E_{pkA}(Na, Nb, B)$
3. $A \rightarrow B: E_{pkB}(Nb, k)$
4. $B \rightarrow A: E_k(M1, E_{pkA}(M2)), Hash(M1)$
5. $A \rightarrow B: E_k(M1, M2, M3), Sign_{skA}(M1, M2, M3)$
6. $B \rightarrow A: E_k(M3)$

The analysis algorithm uses the C# Roslyn source code parser [14]. With it you can get the tree structure of the source code, and you can use filters. We need these filters:

1) InvocationExpressionSyntax – call expressions;
2) VariableDeclarationSyntax – declaration of variables;
3) AssignmentExpressionSyntax – an assignment expression;
4) IfStatementSyntax – statement with a condition statement.

Using filters, you can get the desired expression, after which you can view the tree structure of this expression. For example, using «*AssignmentExpressionSyntax*» we can find the expression «$M1enc1 = RSA.Encrypt\ (M1, true)$». The derived linear tree structure of the expression is shown in fig. 3.

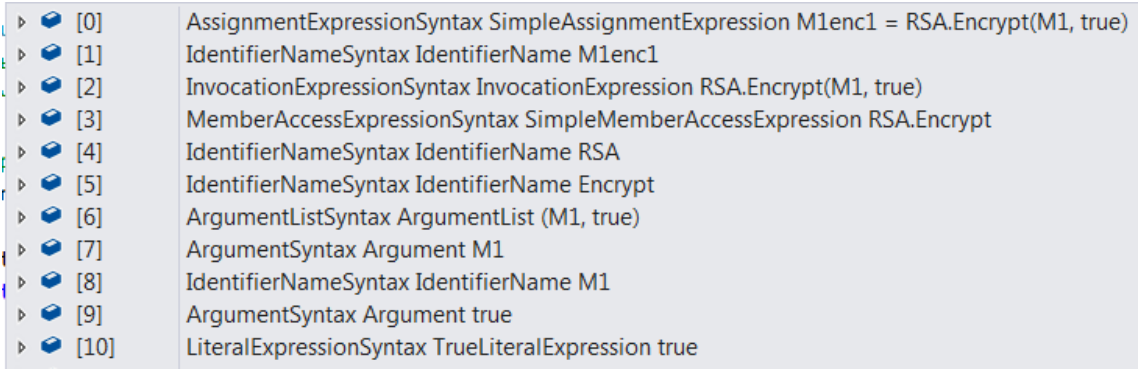

Fig. 3. Tree structure of expression in a linear form

The main purpose of using this parser is to find the transition from one variable to another. In this case, we are interested in the transition $M1enc1 \rightarrow M1$. This is achieved by searching for data such as «*IdentifierName*» together with the use of a black list of expressions. For example, it uses the call of the «*Encrypt*» method, as well as the previously declared object of the asymmetric encryption class «*RSA*», which are present in the black list, and $M1enc1$ and $M1$ that we need can be obtained from here, where the first element will be the variable to which the value will be assigned, and the rest of those that are lower and not included in the black list will be the new value assigned.

The algorithm is based on the definition of important code sections containing constructs specific to cryptographic protocols. Ultimately, the task is to find a chains of variables transformation from the state of sending or receiving messages (socket send/receive) to their initial initialization (static initialization, load from file, etc.), while taking into account possible cryptographic transformations (hash, encryption, etc.). In the course of building a chain, a tree is constructed, the nodes of which are variables with additional information about them, including data type definitions for the final leaves of the tree and cryptographic algorithms in the tree nodes. The tree structure allows you to describe all the chains of data transformations, since the data in the message is combined in various ways, the chains can be strongly branched and joined. Below is a fragment of the source code for the implementation of a part of the cryptographic protocol (messages 1-3) from participant A.

```
1       ...
2      Socket socA =
3      new Socket(ipAddress.AddressFamily,
4      SocketType.Stream, ProtocolType.Tcp);
5
6      socA.Connect(remoteEP);
7
8      RNGCryptoServiceProvider rng = new
9      RNGCryptoServiceProvider();
10
11     byte[] A = new byte[] { 132, 114 };
12     byte[] B = new byte[] { 15, 245 };
13
14     byte[] Na = new byte[64];
15     rng.GetBytes(Na);
16
17     byte[] M1 = new byte[2 + 64];
18
19     Array.Copy(A, 0, M1, 0, A.Length);
20     Array.Copy(Na, 0, M1, 2, Na.Length);
21
22     //1
23     byte[] M1enc;
24     using (RSACryptoServiceProvider RSA =
25     new RSACryptoServiceProvider())
26     {
27        RSA.ImportParameters(
28     rsaPB.ExportParameters(false));
29        M1enc = RSA.Encrypt(M1, true);
30     }
31
32     socA.Send(M1enc);
33
34     //2
```

```
35     byte[] MGet2Encr = new byte[256];
36     socA.Receive(MGet2Encr);
37
38     byte[] MGet2;
39     using (RSACryptoServiceProvider RSA = new
40     RSACryptoServiceProvider())
41     {
42        RSA.ImportParameters(
43     rsaSA.ExportParameters(true));
44        MGet2 = RSA.Decrypt(MGet2Encr, true);
45     }
46
47     byte[] BFromServer = new byte[2];
48     byte[] NaGet = new byte[64];
49     Array.Copy(MGet2, 0, BFromServer, 0, 2);
50     Array.Copy(MGet2, 0, NaGet, 0, 64);
51
52     if (!NaGet.SequenceEqual(Na) &&
53     !B.SequenceEqual(BFromServer))
54     {
55        socA.Shutdown(SocketShutdown.Both);
56        socA.Close();
57        return;
58     }
59
60     byte[] Nb = new byte[64];
61     Array.Copy(MGet2, 64, Nb, 0, 64);
62
63     byte[] k = new byte[32 + 16];
64     rng.GetBytes(k);
65
66     byte[] M3 = new byte[0];
67     M3 = Nb.Concat(k).ToArray();
68
69     //3
70     byte[] M3enc;
71     using (RSACryptoServiceProvider RSA = new
72     RSACryptoServiceProvider())
73     {
74        RSA.ImportParameters(rsaPB.ExportParameters(false));
75        M3enc = RSA.Encrypt(M3, true);
76     }
77     socA.Send(M3enc);
78     ...
79
```

First you need to define the declaration and initialization:

- objects of class *Socket*.
- class objects of the standard library cryptographic algorithms, such as the *RSACryptoServiceProvider* asymmetric encryption algorithm, the *RNGCryptoServiceProvider* random number generator, etc.

The variables of the class object *Socket: [socA]*, classes of cryptographic algorithms are defined: *[rng, RSA]*.

To find variable of the Socket class object, the sending and receiving messages is searched. In this case, there are 3 such constructions. At this stage, you can construct an interaction scheme of the following form:

1. $A \rightarrow B: M1$
2. $B \rightarrow A: M2$
3. $A \rightarrow B: M3$

To determine the structure of the message, it is necessary to build a tree, the nodes of which contain variables with additional information. Consider an example for determining the content of the first message. The order of the algorithm is as follows,

1. The expression of the first message *socA.Send (M1enc)* is taken as the root of the tree. It is necessary to understand the contents of the variable *M1enc*.
2. First you need to find the declaration of the variable *M1enc* using the filter *VariableDeclarationSyntax*. However, in our case, the variable is declared, but not initialized (line 23). In this case, the filter *AssignmentExpressionSyntax* is used and you can find in line 29 the assignment of the value to our variable. *M1enc* is added as a child node with the «*var*» tag, which means it is just a variable.
3. The simplest case of assignment is when the value of one variable is assigned to another. In this case, the situation is more difficult. The variable *M1enc* is assigned the value of the result of the work of the *Encrypt* method for an object of the asymmetric encryption class *RSACryptoServiceProvider*, which takes two parameters as input: what to encrypt and flag whether to use optimal asymmetric encryption with addition (OAEP padding). At the current stage, we remember that the content of the variable *M1* was asymmetrically encrypted and assigned to the variable for sending message 1. In the tree structure, this is displayed as adding a child node *M1* with the note «*AsymENC*», which means that the value of the variable *M1* is encrypted using an asymmetric cipher.
4. Similar to paragraph 2, we are looking for the initialization of the variable *M1*. Using the first filter, you can find out that the variable is a one-dimensional array (line 17). Using the second filter, you must find the assignment of values to our array. These are lines 19 and 20. Two children *Na* and *A* with the mark «*var*» are added to node *M1*.
5. For variable *A*, the final value can be found using the first *VariableDeclarationSyntax* filter (line 11). This is where static initialization occurs in the source code. It is enough for a person to simply understand that this is the initial value, but for the automated determination of this fact it is necessary to understand that this is not a variable. One way to solve this problem is to re-search the right side of the expression, and since more in the design code of the assignment is not detected, this value is final. In the tree structure for node *A*, the initialization leaf is added «*new byte [] {132, 114};*» marked «*DATA*», which means the presence of some semantic data in the variable *A*.
6. For the *Na* variable, the search is carried out further. Using filters, we look for the declaration of the array and its initialization. The declaration occurs in line 14, and initialization occurs in line 15 by calling some method of the *rng* variable, which in turn is an object of the *RNGCryptoServiceProvider* class of random numbers, thus, the value of this variable is defined as a random number. The last leaf «*rng.GetBytes (NaPrev);*» is added to the tree structure marked «*RANDOM*», which means generating a random number.
7. Further search initialization for current leaves gives nothing, therefore the structure of the tree is considered final. The output tree view is shown in fig. 4 in the «Full tree» area and it corresponds to the following chain: *Send (M1enc) -> M1enc = E (M1) -> M1 = {A, Na} -> A = new byte [] {132, 114}, Na = rand ()*. You can also see short tree structure and result message from it.

5. Return data problem

At the moment there is a problem in determining the returned data. For example, in message 1, a random number *Na* is sent, and then in the second message it is sent back. By default, there are

currently two data concepts: *DATA* and *RANDOM*. All that is not a random number – is considered semantic data, for example: keys, identifiers, transferred files, etc. And at this stage, all values are considered different. For example, for the following protocol:

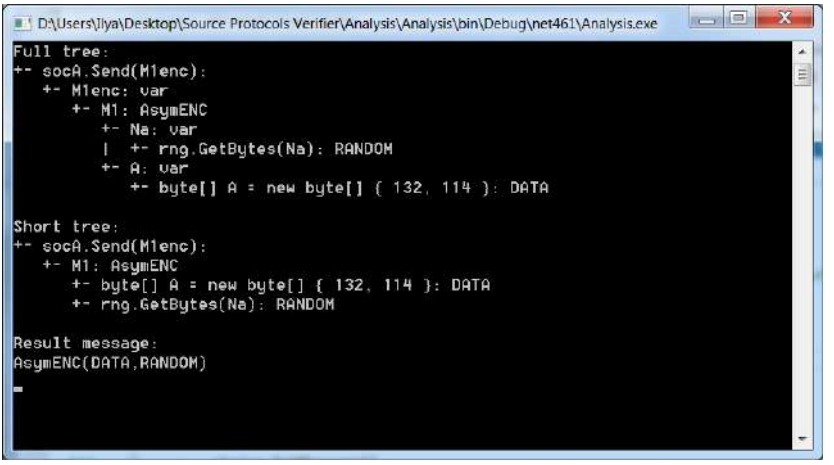

Fig. 4. Output for composing the structure of a single message

1. $A \rightarrow B: Ek(Na, A)$
2. $B \rightarrow A: Ek(Nb, B)$

The result of the work will be as follows:

1. $A \rightarrow B: SymENC(RANDOM, DATA)$
2. $B \rightarrow A: SymENC(RANDOM, DATA)$

And in our context, the default *DATA* in the first message is different from the one in the second message. If the protocol takes the following form:

1. $A \rightarrow B: Ek(Na, A)$
2. $B \rightarrow A: Ek(Nb, Na)$

There is a problem. *Na* just comes back, and on the receiving side we need to understand that this is the same data. For example, when processing message 2 (lines 34-58), we can trace the separated parts. In line 50, the value of the random number *Na* is obtained, after which it is checked for coincidence with what was sent in line 52. Most often in the context of cryptographic protocols, returned values are used for mutual authentication. There can be 2 types: the return of the same number or the return of a function from this number. In both cases, the return value is checked for a match with the one sent earlier. In our case, this is line 53. However, another value is checked here – identifier B. In this case, one of the solutions to this problem would be to find the situation when the variable was sent, and then a value is checked for a match with this variable. In this case, you can assume that this is the case of the return value. However, there may be a number of problems, in particular, just the occurrence of an error in writing code, or simply the absence of such a check of the return value. At the moment, the abstract notion of the type of the *RETURN* variable is used. This means that a variable of this type was returned in the current message.

6. Protocol output structure

Using the algorithm presented in the preceding paragraphs, the complete output structure of the protocol is constructed according to the messages. It is obtained both in short form for formal verification, and in full form for dynamic verification. The full view contains the last variable, before serving in the cryptographic function, the names of the last variables and their initial

initialization, for example, static in the code or loading data from a file. Dynamic analysis will be considered in further work and therefore the contents of the full protocol can be changed.

Short view:

1. $A \rightarrow B: AsymENC(DATA, RANDOM)$
2. $B \rightarrow A: AsymENC(RETURN, RANDOM, DATA)$
3. $A \rightarrow B: AsymENC(RETURN, RANDOM)$
4. $B \rightarrow A: SymENC(DATA, AsymENC(DATA)), HASH(DATA)$
5. $A \rightarrow B: SymENC(RETURN, RETURN, DATA), Sign(RETURN, RETURN, DATA)$
6. $B \rightarrow A: SymENC(RETURN)$

Full view:

1) $A \rightarrow B: AsymENC(DATA, RANDOM)$

```
M1 | byte[] A = new byte[] { 132, 114 } | rng.GetBytes(Na)
```

2) $B \rightarrow A: AsymENC(RETURN, RANDOM, DATA)$

```
M2 | socB.Receive(MGet1) | rng.GetBytes(Nb) |
byte[] B = new byte[] { 15, 245 }
```

3) $A \rightarrow B: AsymENC(RETURN, RANDOM)$

```
M3 | socA.Receive(MGet2Encr) | rng.GetBytes(k)
```

4) $B \rightarrow A: SymENC(DATA, AsymENC(DATA)), HASH(DATA)$

```
ForEncM4 | byte[] M1forSend = File.ReadAllBytes("Mess1.txt") | M2forSend
| byte[] M2forSend = File.ReadAllBytes("Mess2.txt") | M1forSend |
byte[] M1forSend = File.ReadAllBytes("Mess1.txt")
```

5) $A \rightarrow B: SymENC(RETURN, RETURN, DATA), Sign(RETURN, RETURN, DATA)$

```
ConcatMess5 | socA.Receive(MGet4) | socA.Receive(MGet4) |
byte[] M3forSend = File.ReadAllBytes("Mess3.txt") | ConcatMess5 |
socA.Receive(MGet4) | socA.Receive(MGet4) |
byte[] M3forSend = File.ReadAllBytes("Mess3.txt")
```

6) $B \rightarrow A: SymENC(RETURN)$

```
M3From5 | socB.Receive(MGet5)
```

7. Experiments

For testing parser on real project we take our previous project - e-voting system based on blinded intermediaries [15], which implemented on C# language. It consists 3 main components: Voter application, Authentication server, Voting server. The protocol in main voting stage is:

1. $AS \rightarrow V: E_{vas}(N_{as})$
2. $VS \rightarrow V: E_{vvs}(N_b, N_{vs})$
3. $VS \rightarrow AS: E_{asvs}(N_{asvs})$
4. $V \rightarrow AS: E_{vas}(N_{as}, userData, E_{vvs}(N_{vs}, N_v, filledBallot))$
5. $AS \rightarrow VS: E_{asvs}(N_{asvs}, E_{vvs}(N_{vs}, N_v, filledBallot)))$
6. $VS \rightarrow AS: E_{asvs}(N_b, N_{asvs}, "good")$
7. $VS \rightarrow V: E_{vvs}(N_v, N_{vs}, checkID)$

Before the protocol session keys *vas*, *vvs*, *asvs* were generated with ECDHE (the Diffie-Hellman protocol on elliptical curves using ephemeral keys and signing the secret parts) protocol. So at the beginning of the main voting protocol session keys are created. It is necessary to say that *Nb* is a number of blinding, a non-random random number, which is regenerated each time. It is introduced in order to add some data before the semantic random number for making full search more complicated (in particular, it is necessary to select two encryption keys for message 7 in order to find *userData*). Randomly generated random numbers are sent to authenticate the parties as shown in (1)-(3). The message (4) uses the principle of blind intermediaries. The voter encrypts

his vote *filledBallot* on the session key with *VS*, applies his personal data to the ciphertext, and encrypts it on the session key with *AS*. *AS* hashes the sent personal data, searches for the hash in the database and, and, if detected, redirects the message to the *VS* component. *VS* memorizes the vote, generates a *checkID* through which the user can check his vote after the end of the election, and sends it to the user.

Code organization of cryptographic protocols in this project is simple. Message sending or receiving located in methods' block, so there is no difficult code structure. Our parser was launched for this project and we cad this result:

1. $A \rightarrow B: SymENC(RANDOM)$
2. $C \rightarrow B: SymENC(RANDOM, RANDOM)$
3. $C \rightarrow A: SymENC(RANDOM)$
4. $B \rightarrow A: SymENC(RETURN, DATA, SymENC(RETURN, RANDOM, DATA))$
5. $A \rightarrow C: SymENC(RETURN, RETURN)$
6. $C \rightarrow A: SymENC(RANDOM, RETURN, DATA)$
7. $C \rightarrow B: SymENC(RETURN, RETURN, DATA)$

As we can see from output cryptographic protocol structure was extracted correctly. It is necessary to say that in message 4 A gets «*SymENC(RETURN,RANDOM,DATA)*», but in message 5 it sends this like «*RETURN*». So side A doesn't know key for decryption and for it this is some data that was sent to it and it sends this data to another side so there is 1 element «*RETURN*» instead of 3.

7. Future work

Future work primarily includes a segmentation of DATA semantic data into classes:

1) party identifiers;
2) keys;
3) timestamps;
4) authentication Codes;
5) data received from the user.

It is also an important point to determine the ownership of a key by any of the parties in the case of asymmetric encryption, and to the list of parties in the case of symmetric encryption. Support for protocols involving more than two parties will also be needed. In addition, a complete solution to the problem of accurately determining the returned data is necessary to make it possible to build a complete structure of a cryptographic protocol and its further analysis using formal verification tools. After obtaining the structure of the cryptographic protocol, it is necessary to develop an algorithm for automated translation into the specification language of the most well-known protocol verification tools, such as Avispa [16], Scyther [17], ProVerif [18], and others. It is also necessary to improve the parser. At the moment, the structure can only be retrieved from areas of code where all functions for sending and receiving messages are combined into one block, for example, into the body of a function or class method. In the future, it is planned to improve the parser to work with complex code structures.

8. Conclusion

An algorithm was presented for analyzing the source code of the C# programming language for extracting the structure of cryptographic protocols, based on identifying important code sections that contain cryptographic protocol-specific constructions and determining the chain of variable transformations from the sending or receiving status to their initial initialization, taking into account possible cryptographic transformations to compose a tree, from which it is possible to get simplified structure of a cryptographic protocol. An example of a protocol containing all cryptographic functions is given. The output structure of the cryptographic protocol is shown.

Successful practical testing on real e-voting system based on blinded intermediaries is done. For the further possibility of the application of formal verification of protocols and dynamic analysis, it is necessary to make an additional classification of semantic data, determine whether the keys belong to any party or parties, and also solve the problem with the returned values.

References

[1] Chaki S., Datta A. ASPIER: An automated framework for verifying security protocol implementations. In Proc. of the 22nd IEEE Computer Security Foundations Symposium, 2009, pp. 172-185.

[2] Goubault-Larrecq J., Parrennes F. Cryptographic protocol analysis on real C code. Lecture Notes in Computer Science, vol. 3385, 2005, pp. 363-379.

[3] Goubault-Larrecq J., Parrennes F. Cryptographic protocol analysis on real C code. Technical report, Laboratoire Spécification et Vérification, Report LSV-09-18, 2009.

[4] Jürjens J. Using interface specifications for verifying crypto-protocol implementations. In Proc. of the Workshop on foundations of interface technologies (FIT). 2008.

[5] Jürjens J. Automated security verification for crypto protocol implementations: Verifying the jessie project. Electronic Notes in Theoretical Computer Science, vol. 250, № 1, 2009, pp. 123-136.

[6] O'Shea N. Using Elyjah to analyse Java implementations of cryptographic protocols. In Proc. of the Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (FCS-ARSPA-WITS-2008). – 2008.

[7] Backes M., Maffei M., Unruh D. Computationally sound verification of source code. In Proc. of the 17th ACM conference on Computer and communications security, 2010, pp. 387-398.

[8] Bhargavan K. et al. Cryptographically verified implementations for TLS. In Proc. of the 15th ACM conference on Computer and communications security, 2008, pp. 459-468.

[9] Bhargavan K., Fournet C., Gordon A. D. Verified reference implementations of WS-Security protocols. Lecture Notes in Computer Science, vol. 4184, 2006, pp. 77-106.

[10] Bhargavan K. et al. Verified interoperable implementations of security protocols. ACM Transactions on Programming Languages and Systems, vol. 31, №. 1, 2008.

[11] Bhargavan K. et al. Verified implementations of the information card federated identity-management protocol. In Proc. of the 2008 ACM symposium on Information, computer and communications security, 2008, pp. 123-135.

[12] Bhargavan K. et al. Cryptographically verified implementations for TLS. In Proc. of the 15th ACM conference on Computer and communications security, 2008, pp. 459-468.

[13] Needham R. M., Schroeder M. D. Using encryption for authentication in large networks of computers. Communications of the ACM, vol. 21, №. 12, 1978. pp. 993-999.

[14] Capek P., Kral E., Senkerik R. Towards an empirical analysis of. NET framework and C# language features' adoption. In Proc. of the 2015 International Conference on Computational Science and Computational Intelligence (CSCI), 2015, pp. 865-866.

[15] Babenko L., Pisarev I. Distributed E-Voting System Based On Blind Intermediaries Using Homomorphic Encryption. In Proc. of the 11th International Conference on Security of Information and Networks, 2018.

[16] Viganò L. Automated security protocol analysis with the AVISPA tool. Electronic Notes in Theoretical Computer Science, vol. 155, № 12, 2006, pp. 61-86.

[17] Cremers C. J. F. The scyther tool: Verification, falsification, and analysis of security protocols. In Proc. of the International Conference on Computer Aided Verification, 2008, pp. 414-418.

[18] Küsters R., Truderung T. Using ProVerif to analyze protocols with Diffie-Hellman exponentiation. In Proc. of the 22nd IEEE Computer Security Foundations Symposium, 2009, pp. 157-171.

Информация об авторах / Information about authors

Илья Александрович ПИСАРЕВ в настоящее время является аспирантом кафедры безопасности информационных технологий Южного федерального университета. Область научных интересов включает верификацию безопасности криптографических протоколов, проверки на моделях, анализ исходных кодов программ.

Ilya Aleksandrovich PISAREV is a graduate student at the Department of Information Technology Security at the Southern Federal University. The area of scientific interests includes verification of the security of cryptographic protocols, model checks, and analysis of program source codes.

Людмила Климентьевна БАБЕНКО является профессором кафедры безопасности информационных технологий Южного федерального университета. Область научных интересов включает криптографические методы и средства обеспечения информационной безопасности, технология параллельно-векторных вычислений, оценка стойкости криптографических методов защиты информации.

Liudmila Klimentevna BABENKO is currently a professor at the Department of Information Technology Security at the Southern Federal University. The area of scientific interests includes cryptographic methods and means of ensuring information security, technology of parallel-vector computing, evaluation of the strength of cryptographic methods of information protection.

DOI: 10.15514/ISPRAS-2019-31(3)-16

SQLite RDBMS Extension for Data Indexing Using B-tree Modifications

A.M. Rigin, ORCID: 0000-0003-4081-9144 <amrigin@edu.hse.ru>
S.A. Shershakov, ORCID: 0000-0001-8173-5970 <sshershakov@hse.ru>
National Research University – Higher School of Economics,
20, Myasnitskaya st., Moscow, 101000, Russia

Abstract. Multiway trees are one of the most popular solutions for the big data indexing. The most commonly used kind of the multiway trees is the B-tree. There exist different modifications of the B-trees, including B^+-trees, B^*-trees and B^{*+}-trees considered in this work. However, these modifications are not supported by the popular open-source relational DBMS SQLite. This work is based on the previous research on the performance of multiway trees in the problem of structured data indexing, with the previously developed multiway trees C++ library usage. In this research the B^{*+}-tree was developed as the data structure which combines the main B^+-tree and B^*-tree features together. Also, in the research the empirical computational complexities of different operations on the B-tree and its modifications were measured as well as the memory usage. The purpose of the current work is the development of the SQLite RDBMS extension which allows to use B-tree modifications (B^+-tree, B^*-tree and B^{*+}-tree) as index structures in the SQLite RDBMS. The modifications of the base data structure were developed as a C++ library. The library is connected to the SQLite using the C-C++ cross-language API which is developed in the current work. The SQLite extension implements the novel algorithm for selecting the index structure (one of B-tree's modifications) for some table of a database. The provided SQLite extension is adopted by the SQLite EventLog component of the LDOPA process mining library. In addition, the experiment on the counting the empirical computational complexities of operations on the trees of different types is conducted using the developed in this work SQLite extension.

Keywords: B-tree; data indexing; SQLite; DBMS; RDBMS; multiway tree

For citation: Rigin A.M., Shershakov S.A. SQLite RDBMS Extension for Data Indexing Using B-tree Modifications. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 3, 2019. pp. 203-216. DOI: 10.15514/ISPRAS-2019-31(3)-16

Acknowledgements. This work is supported by RFBR according to the Research project No. 18-37-00438 «mol_a» and the Basic Research Program of the National Research University – Higher School of Economics.

Компонент-расширение РСУБД SQLite для индексирования данных модификациями B-деревьев

А.М. Ригин, ORCID: 0000-0003-4081-9144 <amrigin@edu.hse.ru>
С.А. Шершаков, ORCID: 0000-0001-8173-5970 <sshershakov@hse.ru>
Национальный исследовательский университет «Высшая школа экономики»,
101000, Россия, г. Москва, ул. Мясницкая, д. 20

Аннотация. Сильно ветвящиеся деревья являются одним из наиболее популярных решений для индексирования больших объёмов данных. Наиболее распространённой разновидностью сильно ветвящихся деревьев являются B-деревья. Существуют различные модификации B-деревьев, в том числе, рассматриваемые в настоящей работе B^+-деревья, B^*-деревья и B^{*+}-деревья, однако данные

модификации не поддерживаются по умолчанию в популярной реляционной СУБД с открытым исходным кодом SQLite. Данная работа выполняется на основе проведённого ранее исследования эффективности сильно ветвящихся деревьев в задаче индексирования структурированных данных, с использованием разработанной в рамках него C++-библиотеки структур данных – сильно ветвящихся деревьев. В этом исследовании было разработано B^{*+}-дерево как структура данных, совмещающая в себе основные свойства B^{+}-дерева и B^{*}-дерева. Также в исследовании были измерены эмпирические вычислительные сложности различных операций над B-деревом и его модификациями и объём потребляемой данными операциями оперативной памяти. Целью настоящей работы является разработка расширения для реляционной СУБД SQLite, позволяющего использовать модификации B-дерева (B^{+}-дерево, B^{*}-дерево и B^{*+}-дерево) в качестве индексирующих структур данных в РСУБД SQLite. Модификации базовой структуры данных были разработаны в виде C++-библиотеки. Данная библиотека подключается к SQLite, используя разработанный для неё в рамках настоящей работы API на языке C. Расширение для SQLite также реализует новый алгоритм выбора индексирующей структуры данных (одной из модификаций B-дерева) для заданной таблицы в базе данных. Предложенное расширение используется компонентом SQLite EventLog библиотеки LDOPA алгоритмов и структур данных для process mining. Кроме того, проведён эксперимент по сравнению эмпирической вычислительной сложности операций на деревьях разных типов в разработанном расширении для SQLite.

Ключевые слова: B-дерево; индексирование данных; SQLite; СУБД; РСУБД; сильно ветвящееся дерево

Для цитирования: Ригин А.М., Шершаков С.А. Компонент-расширение РСУБД SQLite для индексирования данных модификациями B-деревьев. Труды ИСП РАН, том 31, вып. 3, 2019 г., стр. 203-216 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(3)-16

Благодарности. Работа выполнена при поддержке РФФИ (проект № 18-37-00438) и Программы фундаментальных исследований Национального исследовательского университета – Высшей школы экономики.

1. Introduction

Last decades, the amount of data volume is growing substantially, which exposes the well-known problem of big data [1]. Many companies and laboratories need to collect, store and process big data. There exist many algorithmic and software solutions to cope with these problems. One of these solutions is using indices which are usually represented by data structures such as hash tables and trees.

Using indices creates a new problem – when data are stored on slow carriers, it is more efficient to load data batches from a storage instead of splitting to individual elements. Multiway trees solve this problem. One type of them is a B-tree which was initially described by Bayer and McCreight in 1972 [2]. The B-tree also has several modifications. In this paper, the following B-tree modifications are considered: B^{+}-tree [3], B^{*}-tree [4] and B^{*+}-tree (the latter is developed by the author of this paper data structure, which combines the main B^{+}-tree and B^{*}-tree features) [5].

This paper extends the research made in the framework of the term project [5].

One of the popular open-source relational database management systems (RDBMS) is SQLite [6]. It is used in mobile phones, computers and many other devices. However, this RDBMS does not support using B^{+}-tree or B^{*}-tree as data index structures by default.

The main goals of the work are the following:

- to add B-tree modifications such as B^{+}-tree, B^{*}-tree and B^{*+}-tree to SQLite;
- to develop and implement an algorithm that would allow selecting the appropriate indexing data structure (B-tree, B^{+}-tree, B^{*}-tree or B^{*+}-tree) when a user manipulates a table.

The work includes linking of B-tree modifications from a C++ library (developed by the author of this work previously) to SQLite using a C-C++ cross-language API and developing an algorithm for selecting an indexing data structure.

The rest of the paper is organized as follows. Firstly, B-tree, B^+-tree, B^*-tree and B^{*+}-tree are shortly described. After this, the SQLite, its indexing algorithms and extensions are presented. Then, the B-tree modifications C++ library and connecting it to the SQLite RDBMS is described. After this, our previous researches conducted using this library are presented. These researches have proved the main theoretical B-tree modifications complexity hypotheses and they show the abilities of this library. Then, the indexing approach, the methods for outputting the index representation and information and the development of algorithm of selecting the index structure for table are discussed, after which the experiment conducted using the developed in this work SQLite extension is described. After this, the main points of the paper are summarized in conclusion and used references are presented.

2. B-tree and its modifications

2.1 B-tree

The B-tree is a multiway tree. It means that each node may contain more than one data key. Furthermore, each node except of the leaf nodes contains more than one pointer to the children nodes. If some node contains k keys than it contains exactly $k + 1$ pointers to the children nodes [2].

The B-tree depends on its important parameter which is called B-tree order. The B-tree order is such a number t that:

- for each non-root node, the following is true: $t - 1 \leq k \leq 2t - 1$, where k is the number of keys in the node [2];
- for root node in the non-empty tree the following is true: $1 \leq k \leq 2t - 1$, where k is the number of keys in the node [2];
- for root node in the empty tree the following is true: $k = 0$, where k is the number of keys in the node [2].

B-tree operations complexities are the following (t is the tree order, n is the tree total keys count):

- for the searching operation: time complexity is $O(tlog_t n)$, memory usage is $O(t)$ and disk operations count is $O(log_t n)$ [2];
- for the nodes split operation (the part of the insertion operation): time complexity is $O(t)$, memory usage is $O(t)$ and disk operations count is $O(1)$ [2];
- for the insertion operation (includes the nodes split operation): time complexity is $O(tlog_t n)$, memory usage is $O(tlog_t n)$ for simple recursion and $O(t)$ for tail recursion and disk operations count is $O(log_t n)$ [2];
- for the deletion operation: time complexity is $O(tlog_t n)$, memory usage is $O(tlog_t n)$ for simple recursion and $O(t)$ for tail recursion and disk operations count is $O(log_t n)$ [2].

B-tree is usually used as the data index [2].

The example of B-tree is shown on the fig. 1.

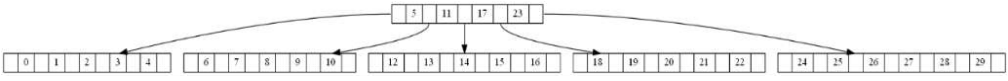

Fig. 1. The B-tree example, tree order t = 6

2.2 B-tree modifications

B^+-tree is the B-tree modification in which only leaf nodes contain real keys (real data), other nodes contain router keys for searching real keys. Leaf nodes in B^+-tree contain $t \leq k \leq 2t$ keys, where t is the tree order, the rules for other nodes are the same as in B-tree [3]. Keys

deletion in B^+-tree is expected to be faster than in B-tree since it is always performed on the leaf nodes.

B^*-tree is the B-tree modification in which each node (except of the root node) is filled at least by 2/3 not 1/2 [4]. Keys insertion in B^*-tree is expected to be faster than in B-tree.

B^{*+}-tree is the B-tree modification developed by the author of this paper which combines the main B^+-tree and B^*-tree features together. In this data structure only leaf nodes contain real keys (real data) as in B^+-tree and each node (except of the root node) is filled at least by 2/3 as in B^*-tree.

3. Implementation and tools

3.1 SQLite and its extensions

The SQLite is the popular open-source C-language library which implements the SQLite relational database management system (RDBMS) [6]. The SQLite default index algorithms are hash-table and B-tree. The SQLite does not implement B^+-tree and B^*-tree based indices.

Nevertheless, SQLite supports loading its extensions at run-time, which can add new functionality to the SQLite. For example, it can be a new index structure implementation. One of such extensions is the R-tree. The R-tree is a B-tree modification which allows to index geodata. It is loaded by the SQLite as the extension and delivered together with the SQLite RDBMS default build.

3.2 B-tree modifications C++ library

The B-tree modifications C++ library was developed by the author of this paper previously. It contains B-tree, B^+-tree, B^*-tree and B^{*+}-tree implementations written in C++ [5].

In the current work this library is connected to the SQLite as the run-time loadable extension. For this goal the C-C++ cross-language API is implemented. It is possible to do using the *extern "C" { ... }* C++ statement. The other tasks are to implement base SQLite extension's methods and to use Makefiles to make this extension run-time loadable correctly. The extension provides module for creating virtual tables (tables which encapsulate callbacks instead of simple reading from database and writing to database) based on this module.

3.3 Research conducted using the library

The B-tree modifications C++ library was previously used for conducting a research on the performance of multiway trees in the problem of structured data indexing by the author of this paper [5].

The CSV files with random content were generated for the indexing, with sizes of 25000, 50000, 75000, 100000 rows. The value of the first cell of each row was considered as a key («name») of the row and was saved in the tree together with the bytes offset of the row in the indexed CSV file. The charts of different dependencies were built using the Python 2.

The chart with the indexing time dependence on the tree order for a file where the «names» (keys) of the rows are uniformly distributed, with the size of 25000 rows is shown on the fig. 2.

According to this chart, B^*-tree and B^{*+}-tree have a better time performance on the keys insertion than B-tree and B^+-tree, as expected. These results are confirmed by the experiments with other parameters (for example, on the larger files with different keys).

However, the better time performance of B^*-tree and B^{*+}-tree on the keys insertion has a cost of a larger memory usage as shown on the fig. 3.

The monotonous dependence of the keys searching on the tree order is not detected as shown on the fig. 5.

The B^*-tree and B^{*+}-tree require more memory during the keys searching than the B-tree and B^+-tree as shown on the fig. 6.

In addition, the B^{+}-tree and B^{*+}-tree have a better time performance on the keys removing than B-tree and B^{*}-tree as expected and shown on the Fig. 7. This chart also proves that the B^{*+}-tree has the best time performance on the keys removing among all the considered in this paper multiway trees and that the dependence of keys removing time on the tree size is logarithmic.

Therefore, the main theoretical hypotheses were confirmed [5].

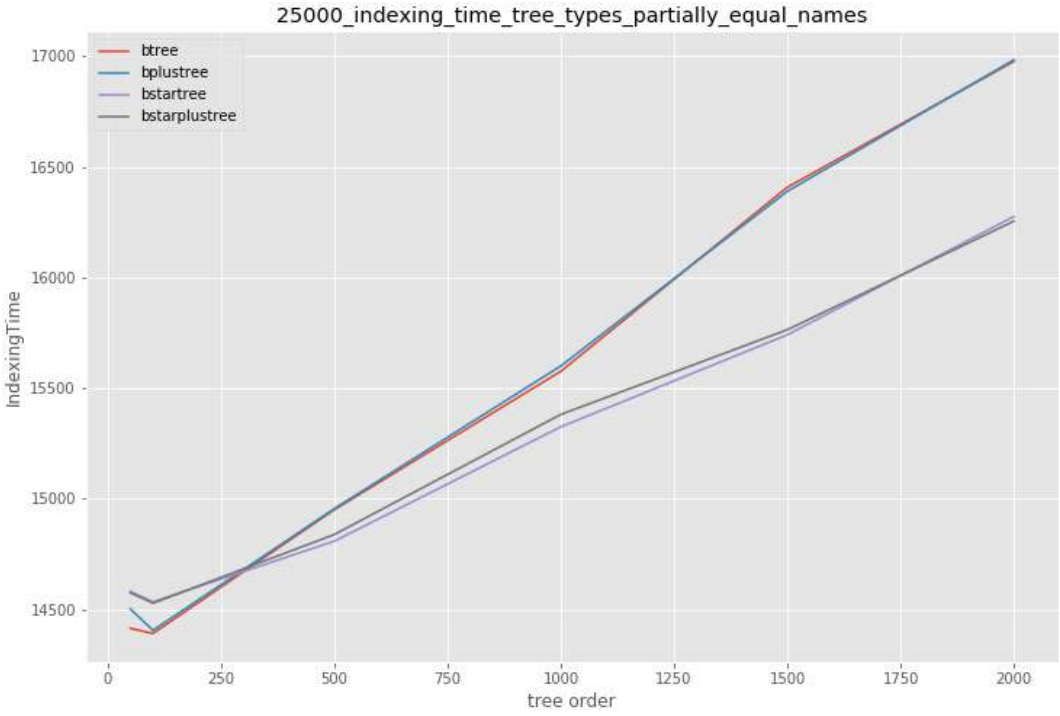

Fig. 2. The chart with the indexing time dependence on the tree order for a file where the «names» (keys) of the rows are uniformly distributed, with the size of 25000 rows

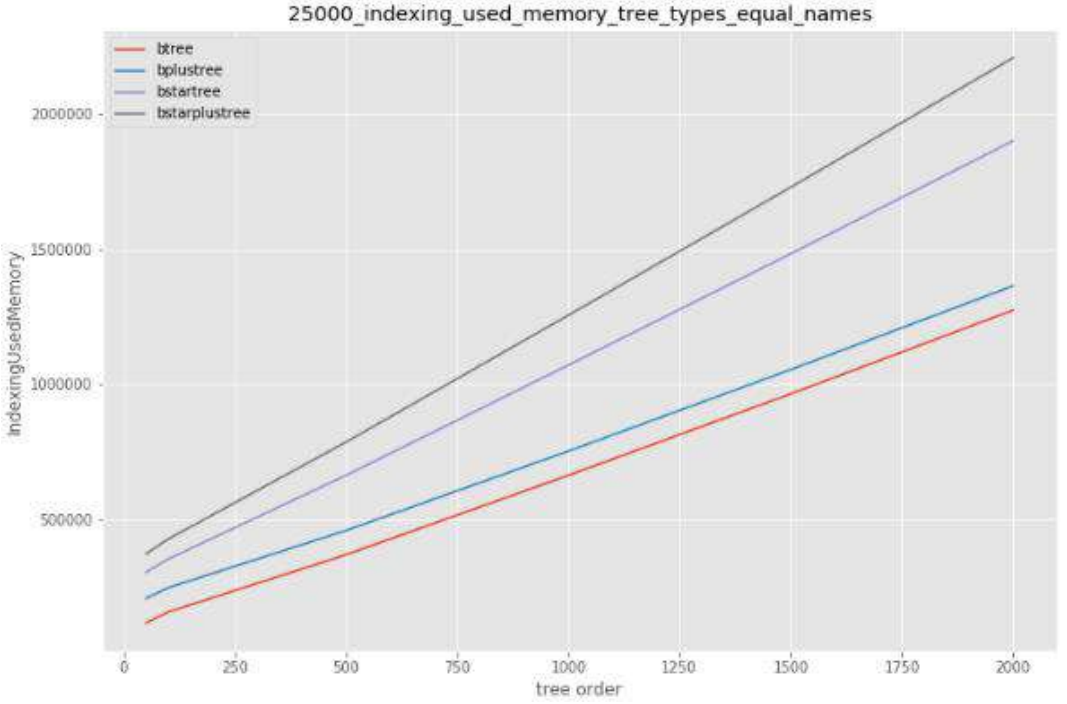

Fig. 3. The chart with the indexing memory usage dependence on the tree order for a file where all the «names» (keys) of the rows are equal, with the size of 25000 rows

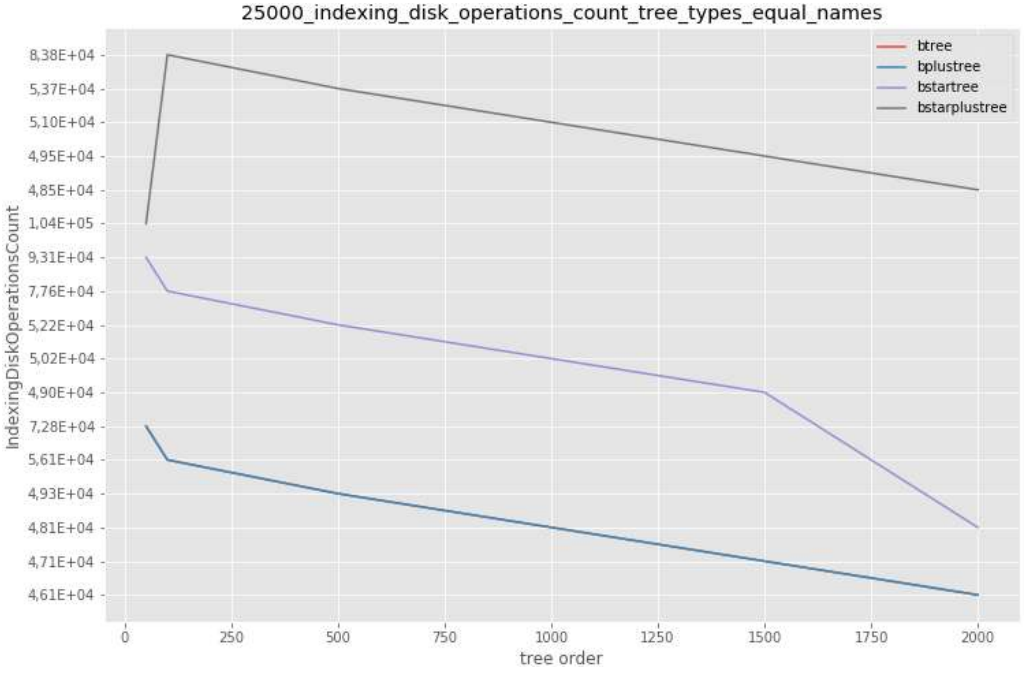

Fig. 4. The chart with the indexing disk operations count dependence on the tree order for a file where all the «names» (keys) of the rows are equal, with the size of 25000 rows

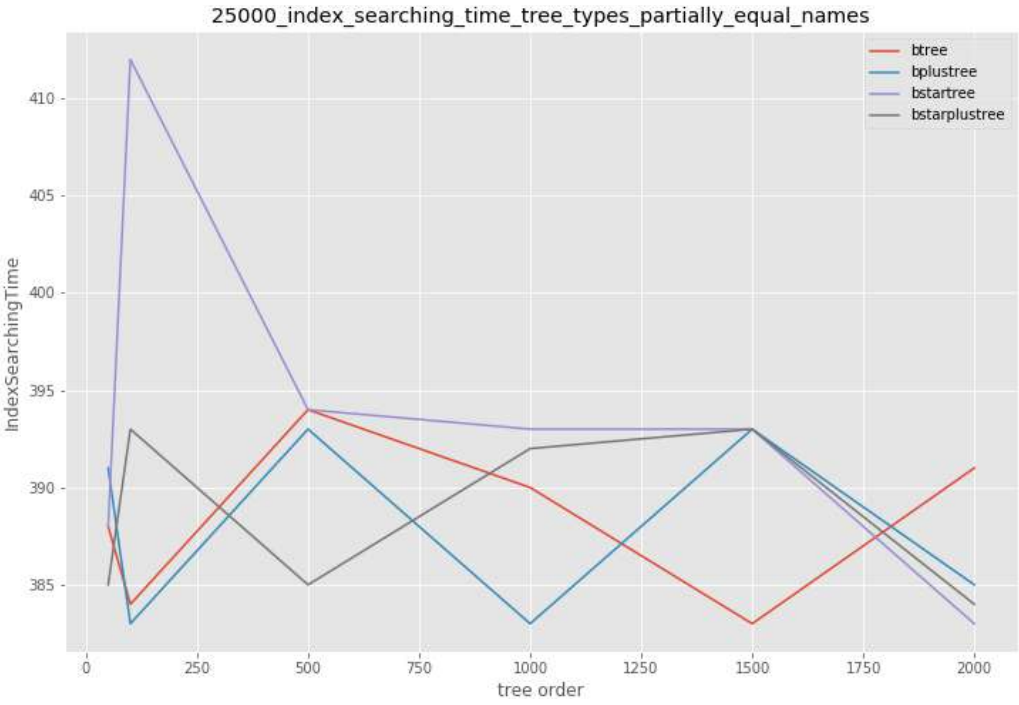

Fig. 5. The chart with the index searching time dependence on the tree order for a file where the «names» (keys) of the rows are uniformly distributed, with the size of 25000 rows

Also, indexing using B^*-tree or B^{*+}-tree requires more disk operations than indexing using B-tree or B^+-tree as shown on the fig. 4.

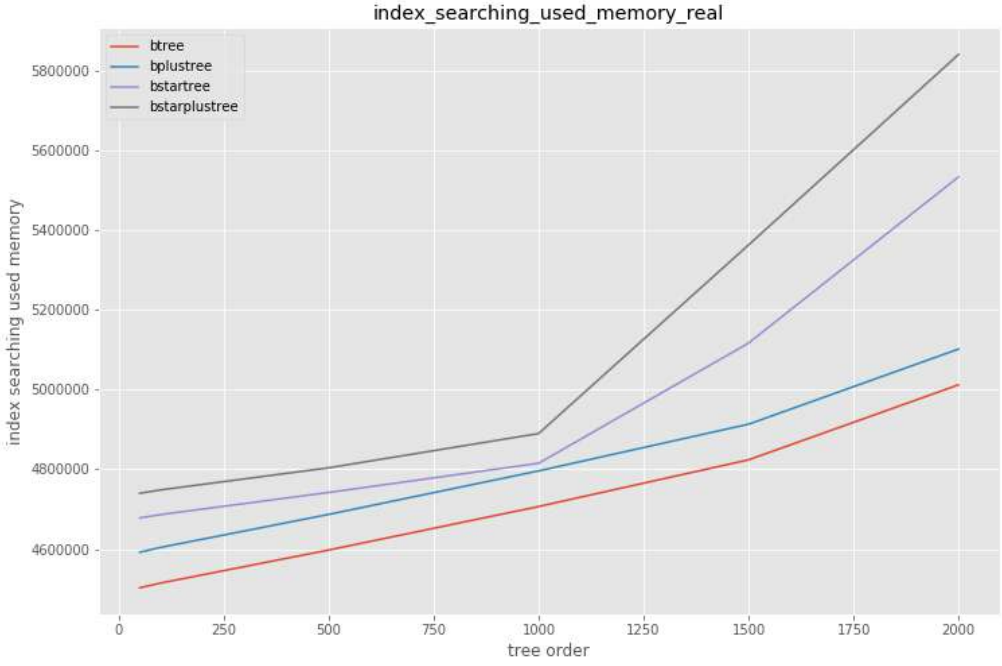

Fig. 6. The chart with the index searching memory usage dependence on the tree order for a file with real (not randomly generated) data

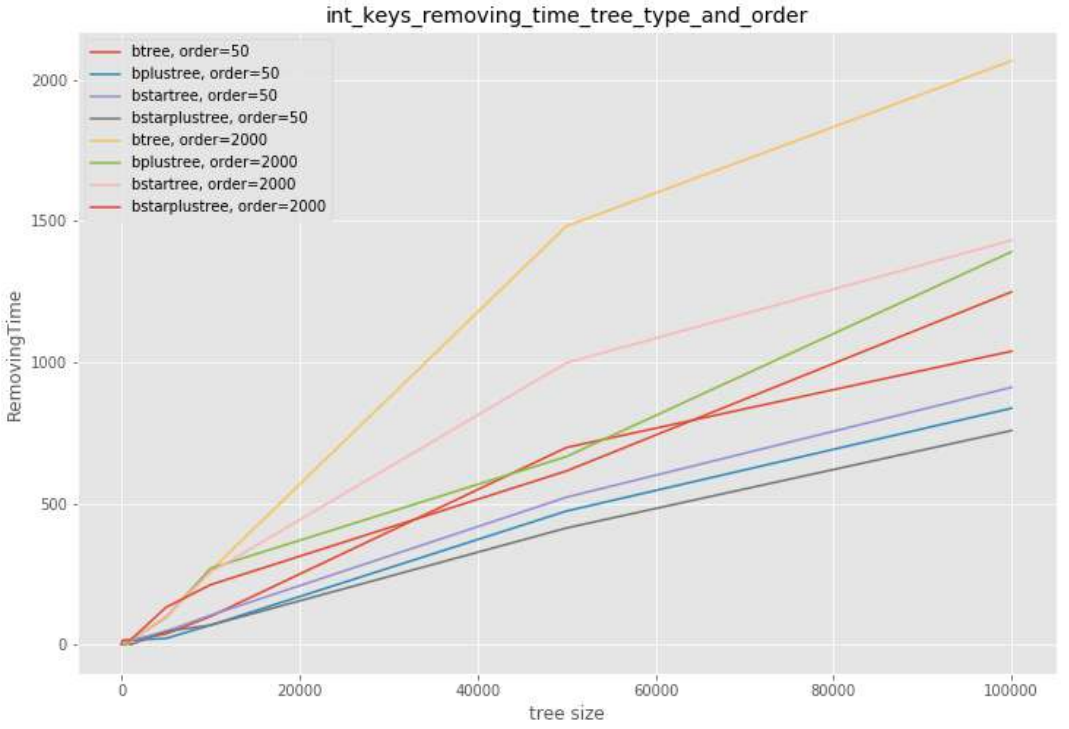

Fig. 7. The chart with the keys removing time dependence on the tree size

4. Working with indices while manipulating DB data

4.1 Table creation, data search and updating

In the current work B-tree modifications based indices are built over the existing SQLite table implementation which is represented in the storage as pages of a B-tree by default.

The table creation and main data operations (inserting, searching, deleting and updating) use the methods presented in the Table. 1.

Table. 1. Main extension methods

Method	Purpose
btreesModsCreate(sqlite3*, void*, int, const char* const*, sqlite3_vtab**, char**)	Creates a new table.
btreesModsUpdate(sqlite3_vtab*, int, sqlite3_value**, sqlite_int64*)	Inserts, deletes or updates a value of a row in the table.
btreesModsFilter(sqlite3_vtab_cursor*, int, const char*, int, sqlite3_value**)	Searches for a row in the table.

The extension with the B-tree modifications based indices provides module for creating virtual tables. User should create a virtual table using the module called *btrees_mods* in order to use one of the B-tree modifications as index for the table. When a user creates such virtual table, the *btreesModsCreate()* method of the extension is called and the matching real table is created in the database. Also, one of B-tree's modifications is created using the algorithm of selecting the index's structure (see the section 5) and the information about the created table and index's structure (including the name of the file with the B-tree or its modification and the attributes of the primary key of the table) is stored in a special table.

When a user inserts a row into a table, the *btreesModsUpdate()* method of the extension is called and a corresponding record for the index structure is created. The record consists of the primary key value of this row and the row id. This record is saved as a data key into the index structure (B-tree or one of its modifications).

When a user searches for a row in a table, the *btreesModsFilter()* method of the extension is called and the value of the primary key of the row being searched is compared with the keys of the index structure. During the key searching only the primary key value part of the tree's keys is compared with the value of the primary key of the row being searched. If the necessary tree's key is found, the row id is extracted from the key and a row found in the table by the row id is considered as a result of the searching.

When a user deletes a row from a table, the *btreesModsUpdate()* method of the extension is called, the primary key of the deleted row is found in the index structure using the same approach as in the search case. The found key is deleted from the index structure.

When a user updates the value of the primary key of a row in a table, the *btreesModsUpdate()* method of the extension is called. The old value of the primary key is deleted from the index structure and the new value is inserted to the index structure.

4.2 Index structure's graphical representation and main information outputting

Also, the several methods are available to output the index structure's graphical representation and main information. They are presented in the Table. 2.

Table. 2. Index structure's information and graphical representation outputting extension methods

Method	Purpose
btreesModsVisualize(sqlite3_context*, int, sqlite3_value**)	Outputs the graphical representation of the table's index structure (tree) into the GraphViz DOT file. It is called after the SQL query such as *SELECT btreesModsVisualize("btt", "btt.dot");*, where *btt* is the table name, *btt.dot* is the outputting GraphViz DOT file name.
btreesModsGetTreeOrder(sqlite3_context*, int, sqlite3_value**)	Outputs the order of the tree used as the table's index structure. It is called after the SQL query such as *SELECT btreesModsGetTreeOrder("btt");*, where *btt* is the table name.
btreesModsGetTreeType(sqlite3_context*, int, sqlite3_value**)	Outputs the type of the tree (1 – B-tree, 2 – B^+-tree, 3 – B^*-tree, 4 – B^{*+}-tree) used as the table's index structure. It is called after the SQL query such as *SELECT btreesModsGetTreeType("btt");*, where *btt* is the table name.

```
SQLite version 3.26.0 2018-12-01 12:34:55
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> .load ./btrees_mods
sqlite> CREATE VIRTUAL TABLE btt USING btrees_mods(id INTEGER PRIMARY KEY, a INTEGER, b TEXT);
sqlite> INSERT INTO btt VALUES (4, 2, "ABC123");
sqlite> INSERT INTO btt VALUES (7, 3, "def");
sqlite> SELECT * FROM btt WHERE id = 4;
4|2|ABC123
sqlite> SELECT * FROM btt WHERE id = 7;
7|3|def
sqlite> SELECT * FROM btt WHERE id = 4 OR id = 7;
4|2|ABC123
7|3|def
sqlite> .tables
btrees_mods_idxinfo  btt                  btt_real
sqlite> SELECT * FROM btt_real;
4|2|ABC123
7|3|def
sqlite> SELECT * FROM btrees_mods_idxinfo;
btt|1|0|id|INTEGER|4|tree_18291557263097.btree
sqlite> DROP TABLE btt;
sqlite> .tables
btrees_mods_idxinfo
sqlite> SELECT * FROM btrees_mods_idxinfo;
sqlite> .exit
```

Fig. 8. SQLite extension's usage example

4.3 SQLite extension's usage

The developed in this work SQLite extension's usage example is presented on the screenshot (fig. 8).

The provided SQLite extension is adopted by the SQLite EventLog component of the Library for Dynamic Operational Process Analysis (LDOPA) [7].

5. Algorithm of selecting the index structure

In this work an algorithm for selecting the index structure for a table is developed and implemented in the following way.

The algorithm considers B-tree's modifications (B^+-tree, B^*-tree and B^{*+}-tree) for using as an index structure.

The algorithm is executed at the start of each operation on the table (search, insertion, deletion or update of the table's row) which uses the *btrees_mods* module. The algorithm consists of the following steps.

1) If the current total number of the operations on a tree is equal to 0, or more than 10000, or not a multiple of 1000, then the algorithm stops, otherwise it goes to step 2.
2) If the current number of the modifying operations (key insertions, key deletions) on the tree is less than 10 % of the current total number of the operations on the tree, then the algorithm stops, otherwise it goes to step 3.
3) If the current number of the key insertion operations is more than $p = 73.97$ % of the total number of the modifying operations on the tree, then the algorithm selects the B^*-tree as the index structure and goes to step 5, otherwise it goes to step 4.
4) The algorithm selects the B^{*+}-tree as the index structure and goes to step 5.
5) If the new index structure has been selected at the steps 3 – 4, then the algorithm rebuilds the existing index structure replacing it by the new selected

index structure and copies all the data stored in the previous index structure to the new index structure.

The tree order of the B-trees and their modifications used in the SQLite extension developed in this work equals 750. For selecting this tree order the average times (for all the four tree types – B-tree, B^+-tree, B^*-tree and B^{*+}-tree) of performing 1000 modifying operations (insertions and deletions) on the tree were measured, for each of the tree orders from 100 to 1000 inclusive with the step of 50 (100, 150, 200, …, 1000). The least average time was achieved for the tree order of 750 and it was equal to 9.55 ms (for 1000 modifying operations on the tree).

The $p = 73.97\ \%$ constant was selected in the following way. The splines for the plots of the average time of performing 1000 modifying operations (insertions and deletions) on the tree depending on the percentage of the insertions among all the modifying operations were drawn for all the four tree types (B-tree, B^+-tree, B^*-tree and B^{*+}-tree) using the Python 2 language. The abscissa of the intersection point of the splines for B^*-tree and B^{*+}-tree was equal to $p = 73.97\ \%$. This intersection point is shown on the fig. 9.

The B^+-tree is used as the default index structure in the developed SQLite extension since its operations have the least memory usage according to the previously conducted experiments (see the section 3.3).

6. Experiment conducted using the developed SQLite extension

The experiment on the counting the empirical computational complexities of operations on the trees of different types is conducted using the developed in this work SQLite extension. The operations' times were counted using the SQLiteStudio GUI manager [8]. The results are presented in the Table 3.

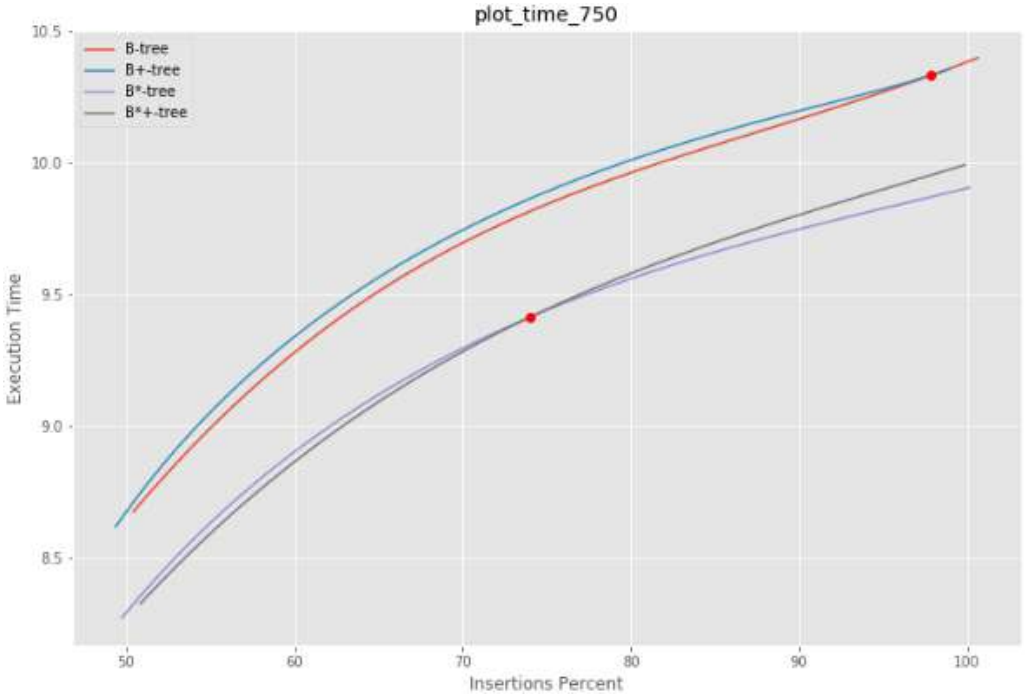

Fig. 9. The splines for the plots of the average time of performing 1000 modifying operations on the tree depending on the percentage of the insertions among all the modifying operations.

Table. 3. Experiment results

Operation on the table	Total execution time (ms)	Mean execution time per row (ms)
Table creation	20	-
First 500 rows insertion	10301	20.6
Next 500 rows insertion	10322	20.6
1001st row insertion (including the B^+-tree into the B^*-tree rebuilding)	40	40
Next 499 rows insertion	9386	18.8
Last 500 rows insertion	9032	18.1
First 500 rows deletion	11558	23.1
Next 500 rows deletion	10708	21.4
1001st row insertion (including the B^*-tree into the B^{*+}-tree rebuilding)	62	62
Next 499 rows deletion	9418	18.9
Last 500 rows deletion	8863	17.7
1000 rows insertion	18890	18.9
Next 5000 rows insertion (including the B^{*+}-tree into the B^*-tree rebuilding)	92395	18.5

According to the data in the Table. 3, the key insertion into the B^*-tree was faster than into the B^+-tree during the experiment. The key deletion from the B^{*+}-tree was faster than from the B^*-tree during the experiment. Also, the key insertion into the B^*-tree was slightly faster than into the B^{*+}-tree during the experiment.

The search in a table took about 1 ms on all the B-tree modifications considered in this work.

7. Conclusion

The big data problem currently affects the world. There are many mathematical and software solutions for collecting, storing and processing big data including the data indexing. Many of the index data structures are tree-based ones such as B-tree and its modifications. B-tree is used as an index structure in many DBMSs including the popular open-source RDBMS SQLite. However, the SQLite does not support its modifications which may be more appropriate for some tasks than the original B-tree. In the current work this problem is elaborated.

Firstly, the B-tree modifications C++ library is connected to the SQLite as the extension using C-C++ cross-language API. After this, the algorithm of the index structure selection is developed and implemented and the experiment is conducted using the developed in this work SQLite extension.

The developed B^{*+}-tree has smaller running time for keys insertion and deletion than B-tree, however it has greater memory usage, which is confirmed by the experiments conducted using the B-tree modifications C++ library.

This work tests new data indexing approaches using the SQLite as an example. The results of the work can be used by researchers and professors in this field and their students. The developed SQLite B-tree modifications extension can be used by all the developers who use this DBMS.

References / Список литературы

[1]. Manyika J., Chui M., Brown B., Bughin J., Dobbs R., Roxburgh C., Hung Byers A. Big data: The next frontier for innovation, competition, and productivity. McKinsey Global Institute, May 2011. Available at: https://www.mckinsey.com/~/media/McKinsey/Business%20Functions/McKinsey%20Digital/Our%20Insights/Big%20data%20The%20next%20frontier%20for%20innovation/MGI_big_data_exec_summary.ashx, accessed Jan. 20, 2019.

[2]. Bayer R., McCreight E. Organization and maintenance of large ordered indexes. Acta Informatica, vol. 1, no. 3, 1972, pp. 173 – 189.

[3]. Pollari-Malmi K. B^+-trees. Available at: https://www.cs.helsinki.fi/u/mluukkai/tirak2010/B-tree.pdf, accessed Dec. 24, 2018.

[4]. B^*-tree. NIST Dictionary of Algorithms and Data Structures. Available at: https://xlinux.nist.gov/dads/HTML/bstartree.html, accessed Dec. 24, 2018.

[5]. Rigin A.M. On the Performance of Multiway Trees in the Problem of Structured Data Indexing. Coursework, Dept. Soft. Eng., HSE, Moscow, Russia, 2018 (in Russian) / Ригин В.М. Исследование эффективности сильно ветвящихся деревьев в задаче индексирования структурированных данных. Курсовая работа, Департамент программной инженерии, ФКН, ВШЭ, Москва, 2018.

[6]. SQLite Home Page. Available at: https://www.sqlite.org/, accessed Jan. 20, 2019.

[7]. Library for Dynamic Operational Process Analysis (LDOPA). xiart.ru Projects. Available at: https://prj.xiart.ru/projects/ldopa, accessed Jul. 1, 2019.

[8]. SQLiteStudio. Available at: https://sqlitestudio.pl/, accessed Jan. 26, 2019.

Информация об авторах / Information about authors

Антон Михайлович РИГИН получил степень бакалавра в области программной инженерии в 2019 г. в Национальном исследовательском университете «Высшая школа экономики» (Москва, Россия). Его исследовательские интересы включают программную инженерию, алгоритмы и структуры данных и их применение в задачах хранения и индексирования данных в СУБД, включая использование B-деревьев и их модификаций для решения этих задач.

Anton Mikhailovitch RIGIN received his bachelor's degree in software engineering from National Research University – Higher School of Economics (Moscow, Russia) in 2019. His research interests include software engineering, algorithms and data structures and their usage in the problems of data indexing and storage in DBMSs, which involves the usage of the B-trees and their modifications in these problems solving.

Сергей Андреевич ШЕРШАКОВ получил степень магистра в области программной инженерии в Национальном исследовательском университете «Высшая школа экономики» (Москва) в 2012 году. В настоящий момент он является научным сотрудником научно-учебной лаборатории процессно-ориентированных информационных систем факультета компьютерных наук Высшей школы экономики. В число научных интересов входят извлечение и анализ процессов (process mining), верификация программного обеспечения, архитектуры информационных систем и преподавание программной инженерии.

Sergey Anreevitch SHERSHAKOV received the MS degree in software engineering from National Research University — Higher School of Economics (Moscow, Russia) in 2012. He is currently a researcher at PAIS Lab of the Faculty of Computer Science at Higher School of Economics. His research interests include process mining, software verification, information systems architectures and teaching software engineering.

DOI: 10.15514/ISPRAS-2019-31(3)-17

«Life» in Tensor: Implementing Cellular Automata on Graphics Adapters

N.A. Shalyapina, ORCID: 0000-0001-8742-4903 <nat.shalyapina@gmail.com>
M.L. Gromov, ORCID: 0000-0002-2990-8245 <maxim.leo.gromov@gmail.com>
¨National Research Tomsk State University,
36 Lenin Avenue, Tomsk, 634050, Russia

Abstract. This paper presents an approach to the description of cellular automata using tensors. This approach allows to attract various frameworks for organizing scientific calculations on high-performance graphics adapter processors, that is, to automatically build parallel software implementations of cellular automata. In our work, we use the TensorFlow framework to organize computations on NVIDIA graphics adapters. As an example cellular automaton we used Conway's Game of Life. The effect of the described approach to the cellular automata implementation is estimated experimentally.

Keywords: Cellular Automata; Conway's Game of Life; Tensor

For citation: Shalyapina N.A., Gromov M.L. «Life» in Tensor: Implementing Cellular Automata on Graphics Adapters. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 3, 2019. pp. 217-228. DOI: 10.15514/ISPRAS-2019-31(3)-17

«ЖИЗНЬ» в тензорах: реализация клеточных автоматов на видеокартах

Н.А. Шаляпина, ORCID: 0000-0001-8742-4903 <nat.shalyapina@gmail.com>
М.Л. Громов, ORCID: 0000-0002-2990-8245 <maxim.leo.gromov@gmail.com>
Национальный исследовательский Томский государственный университет,
634050, Россия, г. Томск, пр. Ленина, д. 36

Аннотация. В данной статье представлен подход к описанию клеточных автоматов с использованием тензоров. Такой подход позволяет привлекать различные фреймворки для организации расчетов на высокопроизводительных графических видеокартах, т.е. для автоматического построения параллельных программных реализаций клеточных автоматов. В нашей работы мы используем фреймворк TensorFlow для организации вычислений на графических видеокартах NVIDIA. В качестве примера клеточного автомата мы рассмотрели игру «Жизнь». Эффект от описанного подхода к программной реализации клеточных автоматов оценён экспериментально.

Ключевые слова: клеточный автомат; игра «Жизнь»; тензор

Для цитирования: Шаляпина Н.А., Громов М.Л. «ЖИЗНЬ» в тензорах: моделирование клеточных автоматов с помощью видеокарт. Труды ИСП РАН, том 31, вып. 3, 2019 г., стр. 217-228 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(3)-17

1. Introduction

The use of automata in description of a dynamic systems' behavior has been known for a long time. The key point of this approach to the description of systems is a representation of the object under study in the form of a discrete automatic device – automaton (State Machine or Transition

System). Under the influence of input sequences (or external factors) an automaton changes its state and produces reactions. There are many types of such automata: the Moore and Mealy machines [1], the cellular automaton [2], and others. The knowledge of the features of the object under study can provide enough information to select the appropriate type of automaton for the object's behavior description. In some cases, it is convenient to use an infinite model. But finite models are mostly common. In the latter case, the sets of states, input actions (or states of the environment), and output reactions are finite.

Our work deals with *cellular automata* (CA). The theory of cellular automata began to take shape quite a long time ago. The work of John von Neumann [3] might be considered as the first work of the cellular automata theory. Today, a large number of studies devoted to cellular automata are known [4, 5]. Note that a major part of these works is devoted to the simulating of spatially distributed systems in physics, chemistry, biology, etc. [6]. The goal of the simulation is to find the states of the cells of a CA after a predetermined number of CA cycles. The resulting set of states in some way characterizes the state of the process or object under study (fluid flow rate at individual points, concentration of substances, etc.). Thus, the task of simulating a certain process or object by a cellular automaton can be divided into two subtasks. First, the researcher must select the parameters of the automaton (the dimension of the grid of cells, the shape of the cells, the type of neighborhood, etc.). And secondly, programmatically implement the behavior of the selected cellular automaton. Our work is focused on the second task – the software implementation of the cellular automaton.

In itself, the concept of a cellular automaton is quite simple and the idea of software implementation is obvious. However, the number of required calculations and the structure of these calculations suggest the use of modern supercomputers with a large number of cores and supporting large-block parallelism. In this case, the cell field of the automaton is divided into separate blocks. Processing of blocks is done in parallel and independently from each other. At the end of each processing cycle, the task of combining the processing results of each block arises. This problem was solved in [7] in the original way. The experimental study in [7] of the efficiency of parallelization was carried out on clusters with 32 and 768 processors. Despite the high effectiveness of this approach, it has some issues. First, this approach assumes that a researcher has an access to a cluster. Supercomputers are quite expensive and usually are the property of some collective access center [8]. Of course, after waiting a certain time in the queue, access to the cluster is possible. But another difficulty arises: a special skill is needed to write parallel programs in order to organize parallel sections of the program correctly. And this leads to the fact that it takes a certain number of experiments with the program to debug it before use. The latter means multiple times of waiting in a queue for a cluster, which, of course, delays the moment of launching actual (not debugging) experiments with cellular automata.

We offer an alternative approach for software implementation of cellular automata, which is based on the use of modern graphics adapters. Modern graphics adapters are also well-organized supercomputers, consisting of several specialized computational cores and allowing execution of operations in parallel. Compared to clusters, graphics adapters are available for a wide range of users and we believe that their capabilities are enough to implement cellular automata. In addition, there are special source development kits or frameworks (for example, ThensorFlow [9]) that can exploit multi-core graphics adapters and help a researcher quickly and efficiently create a software product, without being distracted by thinking about parallelizing data flows and control flows. In this paper, we demonstrate an approach to implementation of cellular automata on graphics adapters based on TensorFlow.

In order to use this tool, we propose to describe the set of states of an automaton cells' by the main data structure used in this framework, namely, the *tensor*. Then we describe the process of evolution of the automaton in terms of tensor operations. A well-known cellular automaton, the Conway's Game of Life, is used as a working example.

The paper is structured as follows. Section 3 presents the basic concepts and definitions concerning the theory of cellular automata. Section 3 provides a description of the game Conway's Game of Life, its features and rules of operation. Section 4 is devoted to a detailed presentation of the proposed approach for software implementation of cellular automata on graphics adapters. The results of computer experiments with the implementation of the Conway's Game of Life and comparison with the results of a classical sequential implementation are presented in section 5.

2. Preliminaries

The Moore machine (finite, deterministic, fully defined) is a 6-tuple $A = \langle S, \hat{s}, I, O, \varphi, \psi \rangle$, where S is the finite nonempty set of states of the machine with a distinguished initial state $\hat{s} \in S$, I is the finite set of input stimuli (input signals), O is a finite set of output reactions (output signals), $\varphi: S \times I \to S$ is a fully defined transition function, $\psi: S \to O$ is a fully defined function of output reactions. If at some moment of time the Moore machine $\langle S, \hat{s}, I, O, \varphi, \psi \rangle$ is at the certain state $s \in S$ and the input signal $i \in I$ arrives, then the machine changes its state to the state $s' = \varphi(s, i)$, and the signal $o = \psi(s')$ appears at its output. The machine starts its operation from the initial state $\hat{s}$ with the output signal $\psi(\hat{s})$. It is important to note that originally Moore defined the machine so that the output signal of the machine is determined not by the final state of the transition, but by the initial one (i.e. in the definition above instead of $o = \psi(s')$ should be $o = \psi(s)$). However, for our purposes it is more convenient to use the definition we have specified.

Let $\mathbb{Z}$ be the set of integers. Consider the set of all possible integers pairs $(i, j) \in \mathbb{Z} \times \mathbb{Z}$. With each pair (i, j) we associate some finite set of pairs of integers $N_{i,j} \subseteq \mathbb{Z} \times \mathbb{Z}$, called *the neighborhood of the pair* (i, j). Pairs of $N_{i,j}$ will be called *neighbors* of the pair (i, j). The sets $N_{i,j}$ must be such that the following rule holds: if the pair (p, q) is the neighbor of the pair (i, j), then the pair $(p + k, q + l)$ is the neighbor of the pair $(i + k, j + l)$, where k and l are some integers. Note that the cardinalities of all neighborhoods coincide and the sets will have the same structure. For convenience, we assume that all neighbors from $N_{i,j}$ are enumerated with integers from 1 to $|N_{i,j}|$, where $|N_{i,j}|$ is the cardinality of the set $N_{i,j}$. Then we can talk about the *first*, *second*, etc. *neighbor* of some pair (i, j). If the pair (p, q) is the n-th neighbor of the pair (i, j), then the pair $(p + k, q + l)$ is the n-th neighbor of the pair $(i + k, j + l)$.

Consider the set of Moore machines of the form $A_{i,j} = \langle S, \hat{s}_{i,j}, S^{|N_{i,j}|}, S, \varphi, \psi \rangle$ such that $\psi(s) = s$. Here i and j are some integers, B^n is the n-th Cartesian power of the set B. The machines corresponding to the neighbors of the pair (i, j) are called *neighbors* of the machine $A_{i,j}$. Neighboring machines will be numbered as well as the corresponding neighboring pairs (that is, the first neighbor, the second, etc.). We specifically note that (i) for each machine $A_{i,j}$ the set of states is the same, i.e. S; (ii) for each machine $A_{i,j}$, the set of output signals coincides with the set of states, that is, also S; (iii) as an output signal, the machine gives its current state; (iv) all machines have the same transition function and the same function of output reaction; (v) as an input signal, machines take tuples of states (of their neighbors), the number of elements in the tuple coincides with the number of neighbors, that is, equals to $|N_{i,j}|$; (vi) machines differ only in their initial states. Let at a given time moment the current state of the first neighbor of the machine $A_{i,j}$ is equal to s_1, the state of the second neighbor is s_2, ..., the state of the n-th neighbor is s_n, where $n = |N_{i,j}|$. Then the tuple $(s_1, s_2, ..., s_n)$ is the input signal of the machine $A_{i,j}$ at this very moment. All machines accept input signals, change their states and provide output signals simultaneously and synchronously. That is, some global clock signal is assumed.

The resulting set $\{A_{i,j} | (i, j) \in \mathbb{Z} \times \mathbb{Z}\}$ of the Moore machines is called a *two-dimensional synchronous cellular automaton* (or simply *cellular automaton* – CA). Each individual Moore machine of this set will be called a *cell*. The set of states of all cells the CA at a given time moment will be called *the global state of the cellular automaton at this time moment*.

The transition rules of cells from one state to another (the function φ), the type of neighborhood of the cells (the sets $N_{i,j}$), the number of different possible cell states (the set S) define the whole variety of synchronous two-dimensional cellular automata.

For clarity, one can draw cellular automata on the plane. For this, the plane is covered with figures. Coverage can be arbitrary, but of course, it is more convenient to do it in a regular way. Classic covers are equal squares, equal triangles and equal hexagons. The choice of one or another method of covering the plane is dictated by the original problem a CA is used for and the selected set of neighbors. Next, the cover figures are assigned to the cells of the cellular automaton in a regular manner. For example, let the plane be covered with equal squares, so that each vertex of each square is also the vertex of the other three squares of the coverage (fig. 1a). Choose the square of this coverage randomly and associate it with the cell $A_{0,0}$. Let the cell $A_{i,j}$ be associated with a certain square. Then we associate the cell $A_{i+1,j}$ with the square on the right, the cell $A_{i-1,j}$ with the square on the left, the cell $A_{i,j+1}$ with the square above, and the cell $A_{i,j-1}$ with the square below (fig. 1b). Cell states will be represented by the color of the corresponding square (fig. 1c)

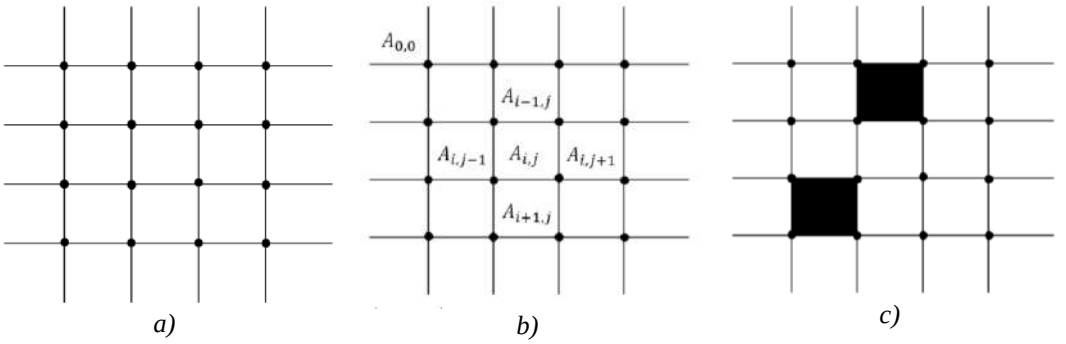

Fig. 1. A CA represented on a plane covered by equal squares: a) the coverage of the plane; b) association of the cells with the squares; c) colour representation of cells' states (for the case |S|=2, «white» – state 0, «black» – state 1)

The resulting square based representation of a CA on a plane is classical one. In our work we consider only this representation.

For the square based representation of a CA, the neighborhoods shown in fig. 2 are the most common.

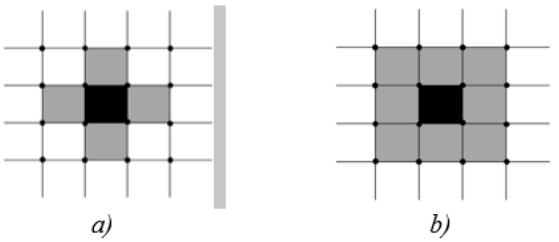

Fig. 2. The neighborhood (grey cells) of a cell (the black one) by a) von Neumann, b) Moore Geometric figures

If a given cellular automaton models a process (for example, heat transfer), then the various global initial states $\{\hat{s}_{i,j} | (i,j) \in \mathbb{Z} \times \mathbb{Z}\}$ of the cellular automaton correspond to different initial conditions of the process. According to the definition of cellular automata introduced by us, the set of cells in it is infinite. However, from the point of view of practice, especially in the case of an implementation of a cellular automaton, a set of cells have to be made finite. In this case, some of the cells lack some neighbors. Therefore, for them the set of neighbors and the transition function

are modified. Such modifications determine the boundary conditions of the process being modeled.

3. Conway's Game of Life

In the 70s of the 20th century, the English mathematician John Conway proposed a cellular automaton called the Conway's Game of Life [10].

The cells of this automaton are interpreted as biological cells. The state «0» corresponds to the «dead» cell, and the state «1» – «alive». The game uses the Moore's neighborhood (Fig. 2b), i.e. each cell has 8 neighbors. The rules for the transition of cells from one state to another are as follows:

- if a cell is «dead» and has three «alive» neighbors then it becomes «alive»;
- if a cell is «alive» and has two or three «alive» neighbors then it remains «alive»;
- if a cell is «alive» and has less than two or more than three «alive» neighbors then it becomes «dead».

For the convenience of perception, the behavior of each cell of the cellular automaton Conway's Game of Life can be illustrated using the transition graph (fig. 3).

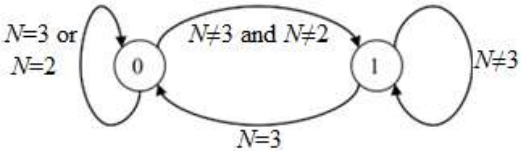

Fig. 3. Cell Transition Graph of the Conway's Game of Life, where N is the number of «alive» neighbors
Geometric figures

Despite the simplicity of the functioning of the automaton, it is an object for numerous studies, since the variation of the initial configuration leads to the appearance of various images of its dynamics with interesting properties. One of the most interesting among them are moving groups of cells – gliders. Gliders not only oscillate with a certain periodicity, but also move through the space (plane). Thus, as a result of experiments, it was established that on the basis of gliders logical elements AND, OR, NOT can be built. Therefore any other Boolean function can be implemented. It was also proved that using the cellular automata Conway's Game of Life it is possible to emulate the operation of a Turing machine.

4. Features of Conway's Game of Life Parallel Implementation

According to our definition, a set of states of a cell is finite. It is obvious that, in this case, without loss of generality, we can assume that the set of states is the set of integers from 0 to $|S|$ - 1, where $|S|$ – is the cardinality of the set of states. Therefore, the global state of the cellular automaton can be represented as a matrix $\boldsymbol{A}$. The element $\boldsymbol{A}_{i,j}$ of this matrix is equal to the current state of the cell $\boldsymbol{A}_{i,j}$. We call the matrix $\boldsymbol{A}$ *the matrix of the global state* of the cellular automaton. If there are no restrictions on the number of cells, then matrix $\boldsymbol{A}$ will be infinite. As have already been mentioned, the number of cells has to be limited from a practical point of view, that is, it is necessary to somehow choose the finite subset of cells. After that, only selected cells are considered. In this case, the ability to describe the global state of the cellular automaton by the matrix is determined by which cells are selected. We assume that the following set of cells is selected: $\{A_{i,j} | (1 \leq i \leq m) \wedge (1 \leq j \leq n)\}$, where m and n – two fixed natural numbers. In this case, the global state matrix is obtained naturally.

Since we use the TensorFlow framework for implementation of a CA, we should work with concepts defined in it. The main data structure in TensorFlow is a multidimensional matrix which in terms of this framework is called a tensor. However, in many cases, such a matrix may not

correspond to any tensor. The tensor in the n-dimensional space must have n^{p+q} components and is represented as $(p + q)$-dimensional matrix, where (p, q) is the rank of the tensor. And, for example, a 2 by 3 matrix does not follow these restricions. But the convenience of data manipulation provided by the framework justifies some deviations from strictly defining the tensor. Therefore, in the case when we are talking about the software implementation of a cellular automaton using TensorFlow, we will consider the notion of the global state matrix of a CA and the notion of *the global state tensor of a CA* as equivalent.

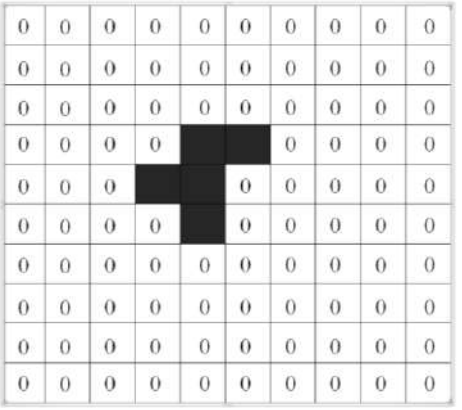

Fig. 4. Some initial global state of the finite state machine for the Conway's Game of Life

Thus, the evolution of the global state of a cellular automaton can be represented (within TensorFlow) as a transformation of the components of the global state tensor. Such a transformation will be called *the evolution of the tensor*.

Thus, the logic of the transition of the cellular automaton from a given global state to the next global state will be described using operations on tensors. In particular, for the software implementation of Conway's Game of Life in our work such operations are the convolution of tensors and the "restriction" of the components value. Let us consider a small example.

Let some initial global state of the cellular automaton (fig. 4) be given.

Black cells are a «alive» cell (state 1), zero means that the cell is «dead» (state 0). The corresponding tensor of the global state has the form (1):

$$\boldsymbol{T} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (1)$$

The next state of a cell of the cellular automaton of the Conway's Game of Life depends on the number of living neighbors of this cell. We suggest using convolution to count the number of living neighbors of a cell. Since set of neighbors in the Conway's Game of Life are specified by the Moore neighborhood, the convolution kernel will have the form (2):

$$\boldsymbol{S} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0{,}5 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad (2)$$

Note the special role of the element $\boldsymbol{S}_{22} = 0{,}5$. This element corresponds to the cell for which the number of living neighbors is calculated. Let the number of living neighbors of a certain dead cell

be calculated. Then it will turn out to be integer because component $\boldsymbol{S}_{22}$ will be multiplied by the state of the dead cell (and it is equal to 0), and in the sum the number $\boldsymbol{S}_{22}$ will not participate. It will turn out to be half-integer in the case when the number of living neighbors of a living cell is calculated. This is important when the cell has two living neighbors. Then the dead cell must remain dead, and the living cell must live. That is, if after the convolution the counted number of living neighbors turns out to be 2 (the cell is dead, it has 2 living neighbors), then in its place should be 0 in the tensor of the global state of the cellular automaton in the next cycle. If, after convolution, the counted number of living neighbors is 2.5 (the cell is alive, and it has 2 neighbors), then in its place should be 1 in the tensor of the global state of the cellular automaton in the next cycle.

Constructing a convolution with the kernel $\boldsymbol{S}$ of the tensor $\boldsymbol{T}$, we obtain the new tensor $\boldsymbol{C}$, where at the intersection of the i-th row and j-th column there is an element corresponding to the number of living neighbors for the cell $\boldsymbol{A}_{i,j}$. Note that we obtain a tensor $(m-2) \times (n-2)$ when constructing a convolution with a kernel of size 3×3 of an arbitrary tensor of the size $m \times n$. In order to save the initial dimensions of the global state tensor of a cellular automaton, we will set the elements in the first and last row and in the first and last column of the global automaton tensor to 0. We will append these zero rows and columns to the result after the convolution is completed. Appended elements in the formula (3) are highlighted in gray. The mentioned fact suggests that some of the subsequent computations are superfluous (namely computations on the appended elements). The amount of extra computations for the global state tensor with dimensions $m \times n$ will be $(2m-2)+(2n-2)$. Then, the part of extra computations in the amount of useful computations is $\frac{(2m-2)+(2n-2)}{(m-1)(n-1)} = O(\frac{1}{m}+\frac{1}{n})$.

$$C=\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 & 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 3 & 3,5 & 2,5 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 3,5 & 4,5 & 4 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 3 & 2,5 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (3)$$

Taking into account the agreement on the half-integer value of the number of living neighbors, the integer part of the value of the tensor component $\boldsymbol{C}$ determines the number of living neighbors of the cell, and the presence of the fractional part means that the cell was alive in the previous step.

According to the rules of the Conway's Game of Life it is necessary to transform the tensor $\boldsymbol{C}$ in order to determine the global state of the cellular automaton in the next step. Components with values in the range [2.5, 3.5] should take the value 1 (cells are alive). The remaining components should become 0 (cells are dead). Among the classical operations on tensors there is no operation that would allow to express the required transformation. However, the framework used in our work was created primarily for the problems of the theory of artificial intelligence, namely, for implementation of neural networks. The data flow there is the flow of tensors (a tensor as an input, a tensor as an output). Computational elements, that change data, are layers of the neural network.

So, for example, in our case for the convolution we use a two-dimensional convolution layer with the kernel $\boldsymbol{S}$ (formula (2)). Any tool for neural network implementation ought to have the special type of layers – activation layers (layer of non-linear transformations). These layers calculate activation functions (some non-linear functions) of each element of the input tensor and put the result into the output tensor. TensorFlow offers a standard set of non-linear activation functions. In addition, it is possible to create custom activation functions. We built our own activation function based on a function from a standard set of functions, called a *Rectified Linear Unit (ReLU)*. The function *ReLU* is defined as follows (formula (4)). Its graph is shown in fig. 5:

$$ReLU = max(0, x) \quad (4)$$

Taking into account the required transformation of the components of the tensor $\boldsymbol{C}$ described above, we suggested the function presented in (5):

$$\delta = ReLU\big(4(x - 2{,}125)\big) - ReLU\big(4(x - 2{,}125)\big) - ReLU\big(4(x - 2{,}125)\big) + ReLU\big(4(x - 2{,}125)\big) \quad (5)$$

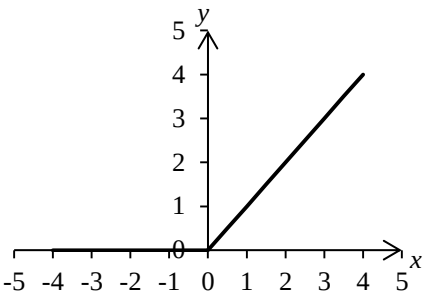

Fig. 5. Diagram of ReLU function

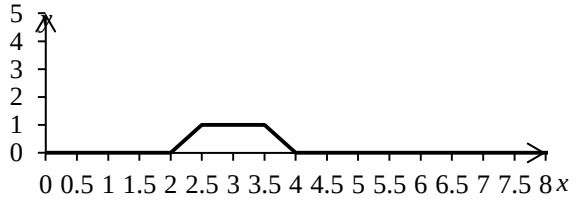

Fig. 6. Diagram of the transition function of the Conway's Game of Life

As a result of applying the function δ to each component of the tensor $\boldsymbol{C}$, the tensor of the global state of the cellular automaton will take the following form (formula (6)).

$$\boldsymbol{T}' = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (6)$$

Thus, the software implementation of the Conway's Game of Life using TensorFlow is a two-layer neural network. The first layer is convolutional, with the kernel from formula (2). The second layer is the activation layer with the activation function from formula (5).

5. Experimental results

We have implemented the described approach for the cellular automaton of the Conway's Game of Life in Python. Since there was no one in our group familiar with TensorFlow, but we have some experience in Keras [11], the implementation was built using Keras as a kind of wrapper over TensorFlow. Keras is a high level interface to various low-level artificial intelligence libraries, including TensorFlow.

The resulting program was launched on a graphics adapter with CUDA support. For comparison with the classical implementation of the cellular automaton of the Conway's Game of Life on a uniprocessor system, we used the implementation of [12].

R-pentamino located in the middle of the field (fig. 4) was used as the initial global state of the cellular automaton of the Conway's Game of Life in the experiments.

We took a square game field (the matrix of the global state of the cellular automaton) with dimensions $m \times m$, where m varied from 10 to 350 with the step 10. For each m, we calculated 1000 subsequent global states of the cellular automaton. The execution time was measured. The calculations were repeated 10 times. Time was averaged. All experiments were conducted on a computer with the following characteristics: Intel Core i5-3470@3.2 GHz CPU, 8 GB RAM, Windows 7-x64 OS, GeForce GTX 650 Ti graphics adapter (1024 MB RAM, 928 MHz base frequency, 768 CUDA cores).

Dependency diagrams of the program execution time on the «length» of the square field side m of the game are shown in fig. 7 and 8. We also built regressions. The regression curves are shown in fig. 7 and 8 as well. A second-degree polynomial was chosen as the regression hypothesis.

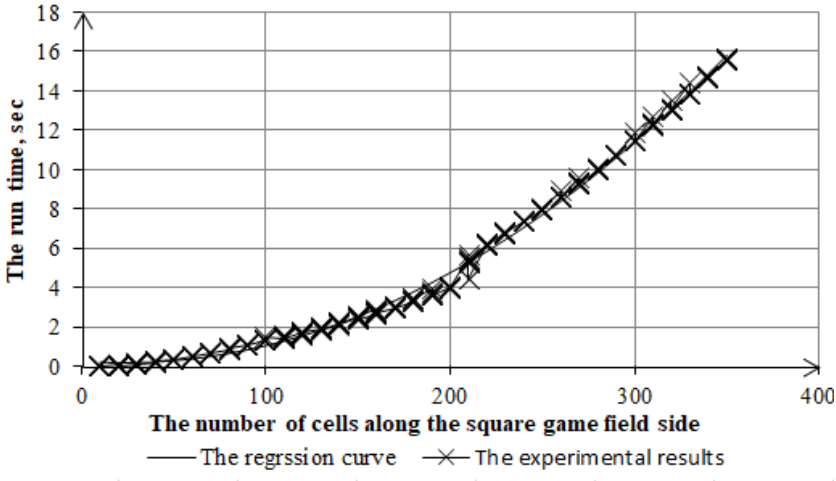

Fig. 7. Results of experiments with a single-threaded implementation

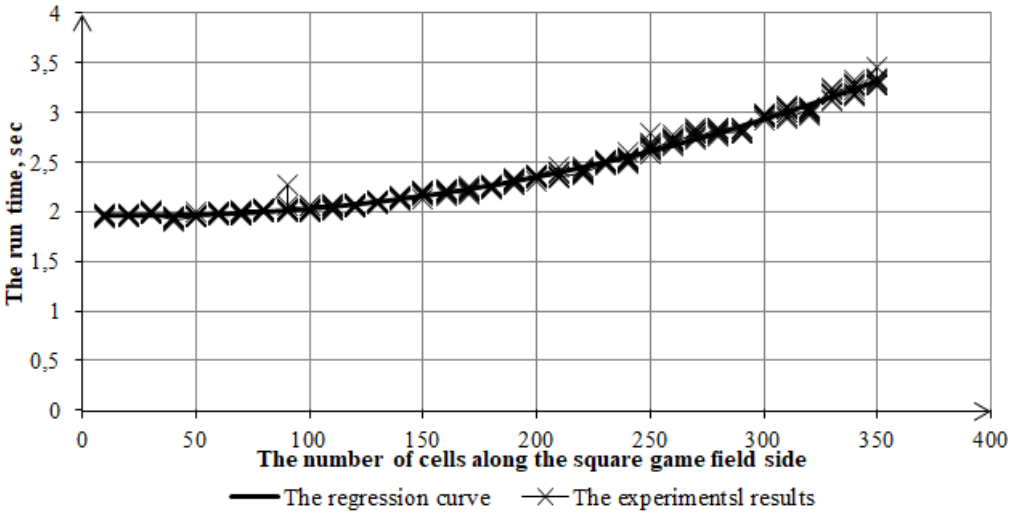

Fig. 8. Results of experiments with CUDA (Keras+TensorFlow) implementation

It can be noted that for small values of m, the execution time of a single-threaded program is smaller than the execution time of the multiprocessor (the graphics adapter) implementation

proposed by us. However, as m grows, the situation changes and the proposed multiprocessor implementation begins to outperform the classical single-threaded implementation. We associate this with the overhead of transferring data from the computer's general RAM to the graphics adapter's RAM and returning the result from the graphics adapter's memory to the computer's memory. When the dimensions of the game field of the Conway's Game of Life are small, the time of actual calculations of the global states of the cellular automaton is much less than the time of transmission of information. As the field size grows, the computation time of the cellular automaton state becomes significant and the multiprocessor implementation on the graphics adapter begins to outrun the single-threaded speed.

Obviously, the dependence of the execution time of programs on the "length" m of the square field side of the Conway's Game of Life must be parabolic. With the growth of m, the number of cells grows as m^2, each cell needs to be processed once per cycle. Therefore, the number of operations must be of the order of m^2. According to the obtained results we constructed regression polynomials of the second degree. Regression curves are in good agreement with experimental data (Fig. 7, 8). It may seem that for a multithreaded implementation the dependency should be different. However, we note that when the number of cells becomes much more than the number of cores in a multi-core system (in our case, the graphics adapter had 768 cores), then processing will be performed block by block: first comes one block of 768 cells, then another, etc. Thus, m^2/K operations will be done, where K is the number of cores, that is, also of the order of m^2.

6. Conclusions

In this paper, a tensor approach to the software implementation of cellular automata is described and programmatically implemented. The approach is focused on launching programs on multi-core graphics adapters. The program is implemented in Python using TensorFlow and Keras as an interface to TensorFlow. TensorFlow allows automatically generate and run multi-threaded programs on multi-core graphics adapters.

The effectiveness of using the developed approach was shown during a series of computer experiments. For the experiments the Conway's Game of Life was chosen. If the number of cells in the automaton is less or equal to the number of cores, then the maximum acceleration can be observed. If the number of cells exceeds the number of cores, then the parallel sections of the program are executed sequentially. This means that with a very large size of the playing field the type of dependence will be parabolic when using a graphics adapter. The latter is confirmed by regression analysis.

References / Список литературы

[1]. Harris D., Harris S. Digital Design and Computer Architecture. Morgan Kaufmann, 2012, 721 p.
[2]. Toffolli T., Margolus N. Cellular Automata Machines. MIT Press, 1987, 279 p.
[3]. von Neumann J. Theory of Self-Reproducing Automata. University of Illinois Press, 1966, 403 p.
[4]. Bandman O. Simulation Spatial Dynamics by Probabilistic Cellular Automata. Lecture Notes in Computer Science, vol. 2493, 2002, pp. 10–19
[5]. Malinetski G.G., Stepantsov M.E. Simulation of diffusion processes by means of cellular automata with Margolus neighborhood. Computational Mathematics and Mathematical Physics, 1998, vol. 38, no. 6, pp. 973-975.
[6]. Weimar J.R. Cellular Automata for Reaction-Diffusion Systems. Parallel Computing, vol. 23, no. 11, 1999, pp. 1699–1715.
[7]. Medvedev Yu.G. Development and Research of a Three-Dimensional Cellular Automaton Model of a Viscous Fluid Flow. PhD thesis, Novosibirsk, 2005, 108 p (in Russian). / Медведев Ю.Г. Разработка и исследование трехмерной клеточно-автоматной модели потока вязкой жидкости. Диссертация на соискание ученой степени кандидата технических наук, Новосибирск, 2005 г., 108 стр.
[8]. Computing Cluster «SKIF Cyberia». Available at: https://cyberia.tsu.ru, accessed 12.05.2019 (in Russian) / Вычислительный кластер СКИФ Cyberia.
[9]. TensorFlow. Available at: https://www.tensorflow.org, accessed 12.05.2019.

[10]. Gardner M. The Fantastic Combinations of John Conway's New Solitaire Game "Life". Scientific American, vol. 223, no 4, 1970, pp. 120–123.
[11]. Keras: The Python Deep Learning library. Available at: https://keras.io, accessed 12.05.2019.
[12]. Implementation of the Game "Life" using C++. Available at: https://code-live.ru/post/cpp-life-game, accessed 12.05.2019.

Информация об авторах / Information about authors

Наталия Андреевна ШАЛЯПИНА получила степень магистра радиофизики в 2018 г. В Национальном исследовательском Томском государственном университете, Томск, Россия. В настоящее время она готовит диссертацию на соискание степени кандидата физико-математических наук по направлению Информатика и вычислительная техника. Область интересов – клеточные автоматы, моделирование.

Natalia Andreevna SHALYAPINA received the M.S. degrees in radiophysics from National Research Tomsk State University, Tomsk, Russia. She is currently pursuing the Ph.D. degree in the field of Information and Computer Engineering. Research interests – cellular automata, simulating.

Максим Леонидович ГРОМОВ окончил радиофизический факультет Томского государственного университета и в 2004 году получил степень магистра радиофизики. В 2009 году защитил кандидатскую диссертацию. С 2009 года занимает должность доцента Томского государственного университета. Научные интересы связаны с дискретными моделями различных систем, обработки информации.

Maxim Leonidovitch GROMOV graduated from Radiphysics faculty of Tomsk State University and got master degree of Radiophysics in 2004. In 2009 he defended the PhD thesis in computer science. Since 2009 he holds the position of Associate Professor of Tomsk State University. Scientific interests are connected with dicrete models of different systems, information processing.

DOI: 10.15514/ISPRAS-2019-31(3)-18

Modeling Nonlinear Stabilization System on Clusters with Intel Xeon Phi Coprocessors

D.V. Melnichuk, ORCID: 0000-0002-6689-8904 <melnichukdv@sgu.ru>
Saratov State University,
83 Astrakhanskaya Street, Saratov, 410012, Russia

Abstract. Currently, cluster systems are widely used, the nodes of which use processors with a large number of cores. Effective software implementation on such computing systems requires that the corresponding mathematical models have a significant parallelism resource. For the problems of modeling of hybrid dynamical systems (HDS) a significant resource of parallelism is typical, since in this class of mathematical models the (theoretically infinite-dimensional) phase space of control objects with space-distributed parameters is isolated. The purpose of this work is to study the effectiveness of the software implementation on parallel computing systems of the class of modeling problems of the influence of typical nonlinearities and nonstationarity on the output vector function of the HDS. As an example, a nonlinear stabilization system for a mobile control object (the rocket taking into account the elastic deformations of its body) is considered.

Keywords: hybrid dynamical systems; processors with scalable architecture

For citation: Melnichuk D.V. Modeling of Angular Stabilization System on Processors with Scalable Architecture. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 3, 2019. pp. 229-240. DOI: 10.15514/ISPRAS-2019-31(3)-18

Моделирование нелинейной системы стабилизации на кластерах с сопроцессорами Intel Xeon Phi

Д.В. Мельничук, ORCID: 0000-0002-6689-8904 <melnichukdv@sgu.ru>
Саратовский государственный университет имени Н.Г. Чернышевского,
410012, Россия, г. Саратов, ул. Астраханская, 83

Аннотация. В настоящее время широкое распространение получают кластерные системы, в узлах которых используются процессоры с большим числом ядер. Эффективная программная реализация на подобных вычислительных системах требует, чтобы соответствующие математические модели обладали значительным ресурсом параллелизма. Для задач моделирования комбинированных динамических систем (КДС) типичен значительный ресурс параллелизма, поскольку в данном классе математических моделей (теоретически бесконечномерное) фазовое пространство объектов управления с распределенными по пространству параметрами является изолированным. Целью работы является исследование эффективности программной реализации на параллельных вычислительных системах класса задач моделирования влияния типовых нелинейностей и нестационарности на выходные вектор-функции КДС. В качестве примера рассмотрена нелинейная система стабилизации подвижного объекта управления (ракеты с учетом упругих деформаций ее корпуса).

Ключевые слова: комбинированные динамические системы; процессоры с масштабируемыми архитектурами

Для цитирования: Мельничук Д.В. Моделирование системы угловой стабилизации на процессорах с масштабируемыми архитектурами. Труды ИСП РАН, том 31, вып. 3, 2019 г., стр. 229-240 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(3)-18

1. *Introduction*

Currently, cluster systems are widely used, in the nodes of which one or several processors with a large number of cores are used. Examples include computing systems with new Intel Xeon processors with scalable architecture or computing systems with Intel Xeon Phi coprocessors that are used as virtual cluster nodes. Parallel computational architectures of this class are effective only when solving problems with a significant parallelism resource. In this case, classes of mathematical models that are effectively implemented on Intel Xeon Phi, will be effectively implemented on modern scalable Intel Xeon architectures.

Hybrid dynamical systems (HDS) [1, 2] are mathematical models of a number of technical systems containing control objects with lumped parameters and connected to them across the boundaries of control objects with distributed parameters (see fig. 1). HDS is characterized by input and output vector functions. The nonlinear system of angular stabilization of the movable control object with deformable body is the example [3, 4]. HDS are systems of ordinary differential equations (ODE) and partial differential equations (PDE) connected by means of boundary conditions (BC) and constraint's conditions (CC) under appropriate initial conditions (IC). For the problems of HDS modeling a significant resource of parallelism is typical, since the (theoretically infinite-dimensional) phase space of control objects with distributed parameters is isolated.

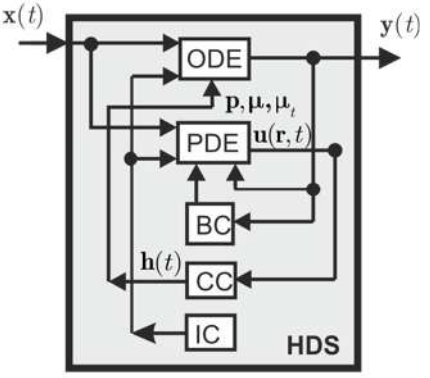

Fig. 1. HDS Structural scheme

2. Related work

MPI technology is standard for cluster systems with distributed memory, and both optimization of MPI itself [5] and parallel libraries [6] and algorithms based on it are relevant. Examples are problems of mathematical physics [7, 8], graph theory [9], sparse matrix factorization [10]. Optimization of parallel algorithms for cluster systems with many-core processors was considered in [11, 12]. If adaptive algorithms of numerical modeling are used or nodes of the cluster have different capacities, dynamic balancing of computational load is required [13]. For dynamic balancing of computational load on cluster systems, the MPI-MAP parallelization pattern was previously implemented [3]. The main theorems on the stability of linearized HDS are formulated and proved in [1, 2]. In work [4] adaptive algorithms of HDS parametric synthesis are offered. Modeling of systems of angular stabilization of movable control objects with deformable body was considered in [14, 3, 4].

3. Problem formulation

The purpose of this work is to study the effectiveness of the software implementation on parallel computing systems of the class of modeling problems of the influence of typical nonlinearities and nonstationarity on the output vector function of the HDS. We consider a similar [4] system of angular stabilization of the movable control object (the rocket taking into account the deformations of its body), but providing stabilization both with respect to the vertical direction and with respect to the longitudinal axis, as well as a smooth change in the time of the thrust force of the rocket engine.

4. Parallel algorithms for modeling of hybrid dynamical systems

HDS with piecewise continuous input vector function $\mathbf{x}(t)$, $\mathbf{x}\colon \mathrm{R} \to \mathrm{R}^{N_x}$ and continuous output vector function $\mathbf{y}(t)$, $\mathbf{y}\colon \mathrm{R} \to \mathrm{R}^{N_y}$ correspond to equations

$$\begin{gathered}\dot{\mathbf{y}} = \mathbf{f}(\mathbf{x}, \mathbf{y}, \mathbf{h}, \mathbf{p}, \boldsymbol{\mu}, \boldsymbol{\mu}_t t); \dot{\mathbf{u}} = \mathbf{F}(\mathbf{u}, \mathbf{x}, \mathbf{y}, \dot{\mathbf{y}}, \boldsymbol{\mu}, \boldsymbol{\mu}_t t), \mathbf{r} \in \Omega \\ \mathbf{G}(\mathbf{u}, \mathbf{y}, \boldsymbol{\mu})|_S = 0, S = \partial\Omega; \mathbf{h} = \int_S \mathbf{H}(\mathbf{u}, \boldsymbol{\mu}) dS \\ \mathbf{y}(0) = \mathbf{y}_0, \mathbf{u}(\mathbf{r}, 0) = \mathbf{u}_0(\mathbf{r})\end{gathered} \quad (1)$$

Here $\mathbf{r} \in \mathrm{R}^{N_r}$ – are independent spatial coordinates of individual points of the object with distributed parameters, $\Omega \subset \mathrm{R}^{N_r}$ – area occupied by an object with distributed parameters, $\mathbf{f}\colon \mathrm{R}^{N_x} \times \mathrm{R}^{N_y} \times \mathrm{R}^{N_h} \times \mathrm{R}^{N_p} \times \mathrm{R}^{N_\mu} \times \mathrm{R}^{N_t} \to \mathrm{R}^{N_y}$, $\mathbf{h}\colon \mathrm{R} \to \mathrm{R}^{N_h}$, $\mathbf{u}(\mathbf{r}, t)$, $\mathbf{u}\colon \mathrm{R}^{N_r} \times \mathrm{R} \to \mathrm{R}^{N_u}$ – distributed output vector function, operators $\mathbf{F}\colon (\mathrm{R}^{N_r} \times \mathrm{R} \to \mathrm{R}^{N_u}) \times (\mathrm{R} \to \mathrm{R}^{N_x}) \times (\mathrm{R} \to \mathrm{R}^{N_y}) \times (\mathrm{R} \to \mathrm{R}^{N_y}) \times \mathrm{R}^{N_\mu} \times \mathrm{R}^{N_t} \to (\mathrm{R}^{N_r} \times \mathrm{R} \to \mathrm{R}^{N_u})$, $\mathbf{G}\colon (\mathrm{R}^{N_r} \times \mathrm{R} \to \mathrm{R}^{N_u}) \times (\mathrm{R} \to \mathrm{R}^{N_y}) \times \mathrm{R}^{N_\mu} \to (\mathrm{R}^{N_r} \times \mathrm{R} \to \mathrm{R}^{N_g})$, $\mathbf{H}\colon (\mathrm{R}^{N_r} \times \mathrm{R} \to \mathrm{R}^{N_u}) \times \mathrm{R}^{N_\mu} \to (\mathrm{R} \to \mathrm{R}^{N_h})$ correspond to partial differential equations, boundary conditions, and coupling conditions; $\mathbf{p} \in \mathrm{R}^{N_p}$ – feedback parameters; $\boldsymbol{\mu} \in \mathrm{R}^{N_\mu}$ – the parameters of model nonlinearities; $\boldsymbol{\mu}_t \in \mathrm{R}^{N_t}$ – parameters characterizing the unsteadiness of the system from the point of view of the automatic control theory; the point at the top indicates the time t differentiation. When $\boldsymbol{\mu} = \boldsymbol{\mu}_t = 0$ HDS (1) becomes linear stationary. After parametric synthesis, numerical simulation of the effect of typical nonlinearities and unsteadiness on the output vector function of a nonlinear HDS (1) is performed. In this case, the input vector function $\mathbf{x}(t)$ and the initial conditions $\mathbf{y}_0$, $\mathbf{u}_0(\mathbf{r})$ are fixed, and the components of the vectors $\boldsymbol{\mu}$ and $\boldsymbol{\mu}_t$ change with a fixed step within a parallelepiped. The element-by-element transformation of sequence $(\boldsymbol{\mu}_j, \boldsymbol{\mu}_{t_j})$, $j = 1,2,3,\ldots$ into a sequence of values characterizing the maximum and standard deviations of function $\mathbf{y}(t; \boldsymbol{\mu}_j, \boldsymbol{\mu}_{t_j})$ from $\mathbf{y}(t; 0,0)$ is parallelized

$$\begin{gathered}(\boldsymbol{\mu}_j, \boldsymbol{\mu}_{t_j}) \to (v_{1_j}, v_{2_j})^T, j = 1,2,3,\ldots; v_1 = \max_{0 \le t \le t_{\max}} |\mathbf{y}(t; \boldsymbol{\mu}, \boldsymbol{\mu}_t) - \mathbf{y}(t; 0,0)| \\ v_2 = \left[t_{\max}^{-1} \int_0^{t_{\max}} |\mathbf{y}(t; \boldsymbol{\mu}, \boldsymbol{\mu}_t) - \mathbf{y}(t; 0,0)|^2 dt\right]^{1/2}, t_{\max} >> 1\end{gathered} \quad (2)$$

The transformation (2) can be adapted to the "two-layer" MPI-OpenMP scheme, where a separate MPI-MAP executing process performs the transformation of

$$\{(\boldsymbol{\mu}_j, \boldsymbol{\mu}_{t_j}), j = \overline{1,m}\} \to \{(v_{1_j}, v_{2_j})^T, j = \overline{1,m}\} \quad (3)$$

by parallelizing calculation of the values on the right side (3) based on OpenMP.

Numerical integration of the initial boundary value problem (1) is implemented by the Galerkin's projection method [4] and subsequent application of the BDF method to the resulting Cauchy problem for the system of ordinary differential equations.

5. Model of stabilization system

The object moves with respect to a fixed coordinate system $O_0 x_0 y_0 z_0$ (see fig. 2) under the action of force **P**, attraction to the Earth and external disturbing horizontal force $\mathbf{F}_e = (0, F_{e_{y0}}, F_{e_{z0}})^T$.

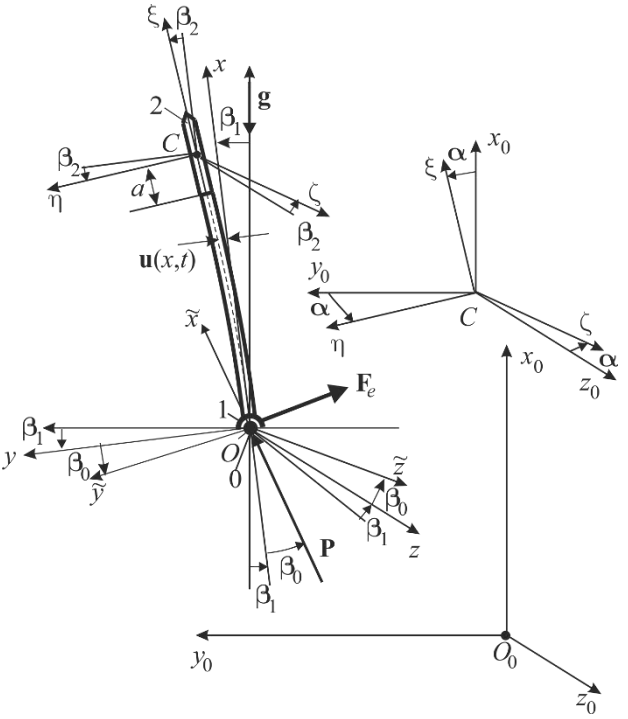

Fig. 2. Structural scheme

The coordinate system $Oxyz$ is connected to the body 1, and $\mathbf{r}_1 = (x_1, y_1, z_1)^T$ and $\boldsymbol{\beta}_1 = (\beta_{1,1}, \beta_{1,2}, \beta_{1,3})^T$ characterize its linear and angular displacements relative to $O_0x_0y_0z_0$. Linear $\mathbf{r}_2 = (x_2, y_2, z_2)^T$ and angular $\boldsymbol{\beta}_2 = (\beta_{2,1}, \beta_{2,2}, \beta_{2,3})^T$ displacement of body 2 with respect to $Oxyz$ is caused by the elastic displacement $\mathbf{u} = (u_x, u_y, u_z)^T = \mathbf{u}(x,t)$ of the centerline of the hull. The rotation angle $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \alpha_3)^T$ of the body 2 relative to $O_0x_0y_0z_0$ measures the gyrostabilizer, and the control moments of the forces $M_j^{(c)}$, $j = 1,2,3$ are formed. Under the action of $M_2^{(c)}$ and $M_3^{(c)}$ body 0 rotates at angles $\boldsymbol{\beta}_0 = (0, \beta_{0,2}, \beta_{0,3})^T$ relative to $Oxyz$. The moment $M_1^{(c)}$ acts on the body 1 and compensates for the rotation of the movable object relative to the longitudinal axis. Let $\boldsymbol{\omega}_0 = (\omega_{0_x}, \omega_{0_y}, \omega_{0_z})^T$, $\boldsymbol{\Omega}_1 = (\Omega_{1_x}, \Omega_{1_y}, \Omega_{1_z})^T$, $\boldsymbol{\Omega}_2 = (\Omega_{2_\xi}, \Omega_{2_\eta}, \Omega_{2_\zeta})^T$ be the relative and absolute angular velocities of bodies 0, 1, 2; $\mathbf{Q} = (Q_1, Q_2, Q_3)^T$, $\mathbf{M} = (M_1, M_2, M_3)^T$ be the internal forces and moments acting in the cross sections of the body. Here $\mathbf{x}(t) = (F_{e_{y0}}(t), F_{e_{z0}}(t))^T$ and $\mathbf{y}(t) = (\beta_{1,3}(t), \beta_{2,3}(t), \beta_{1,2}(t), \beta_{2,2}(t), \beta_{1,1}(t), \beta_{2,1}(t))^T$ are input and output vector functions, $\mathbf{p} = (p_1, p_2, \dots, p_{12})^T$ are feedback parameters. The model equations of the nonlinear stabilization system are given in Appendix A. The set of parameters $\boldsymbol{\mu} = (\mu_1, \mu_2, \mu_3)^T$ characterizes typical nonlinearities, and the parameter $\boldsymbol{\mu}_t = \{\mu_4\}$, $\mu_4 \geq 0$ characterizes a smooth change in the characteristic overload according to the law

$$a_x(t) = a_x^{(\min)} + (a_x^{(\max)} - a_x^{(\min)})e^{-\mu_4 t},\ t \geq 0,\ a_x^{(\min)} < a_x \leq a_x^{(\max)} \qquad (4)$$

At $\boldsymbol{\mu} = 0$, the model equations are linearized and decomposed into three independent subsets corresponding to the motion in the $O_0x_0y_0$ and $O_0x_0z_0$ planes (by virtue of symmetry, they pass into each other), as well as to the rotation relative to the longitudinal axis. In this case, $p_{5+j} = p_j$, $j = \overline{1,5}$, correspond to the stabilization system in the vertical direction, and p_{11}, p_{12} correspond to the stabilization system with respect to the longitudinal axis.

6. Numerical simulation results

In the numerical simulation of the output vector functions of the nonlinear angular stabilization system, the components of the input vector function were given as $F_{e_{y0}}(t) = 1(t)$, $F_{e_{z0}}(t) = 1(t) - 1(t-1)$, where $1(t)$ is the unit jump function of Heaviside. For stabilization system with a set of parameters

$$J_0 = 0.02, m_1 = 0.3, J_1 = 0.07, m_2 = 0.2, J_2 = 0.05, a = 0.166667, a_x^{(\min)} = 0.2, a_x^{(\max)} = 2,$$
$$\gamma = 0.01, J_{1k} = 0.1, J_{2k} = 0.05, J_k = 2, \mu_1 = 0.08, \mu_2 = 0.15, \mu_3 = 0.055, \mu_4 = 0.05 \quad (5)$$

The feedback parameters of the stabilization system in the direction of vertical $p_1 = p_6 = 6.347$, $p_2 = p_7 = 13.12$, $p_3 = p_8 = 17.59$, $p_4 = p_9 = 14.03$, $p_5 = p_{10} = 5.951$ were chosen on the basis of an adaptive algorithm of parametric synthesis of the family of linearized models of HDS [4]. Since the stabilization of the object with respect to the longitudinal axis is intended to compensate for the slow accumulation of errors due to nonlinear effects, the feedback parameters $p_{11} = 0.04$, $p_{12} = 1$ are selected in the central part of the stability region.

Fig. 3 presents the results of numerical simulation of the components $\beta_{1,2}$ and $\beta_{1,3}$ of the output vector functions of the original nonlinear unsteady HDS (shown as a solid line) and its linear stationary analog at $\mu_1 = \mu_2 = \mu_3 = \mu_4 = 0$ (shown as a dashed line). The significant difference of the results is explained by the fact that the dimensionless overload a_x decreases smoothly, with the decrease of a_x in the considered range of overload changes in the linear stationary system, the attenuation of transients decreases, and the characteristic value of the output vector function increases. Nevertheless, parametric synthesis by the linearized model allows to stabilize the original nonlinear system in the vertical direction in the entire range of overloads. As follows from the results presented in Fig. 4, the selected values of the feedback parameters p_{11} and p_{12} allow to stabilize the movable control object with respect to the longitudinal axis, i.e. to compensate for the slow accumulation of errors due to nonlinear effects.

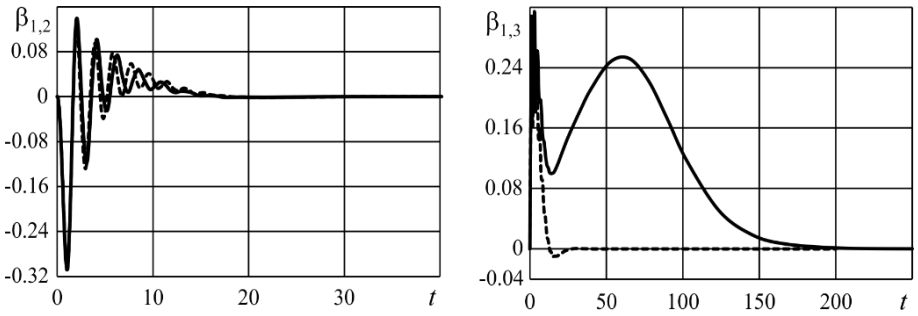

Fig. 3. Stabilization in the vertical direction

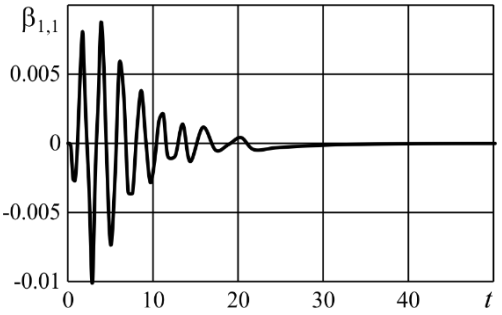

Fig. 4. Stabilization with respect to the longitudinal axis

Fig. 5 shows the dependences of the parameters $\mu_3 \in [0{,}0.055]$ and $\mu_4 \in [0{,}0.05]$ at fixed μ_1 and μ_2 maximum v_1 and standard v_2 deviations (see eq. (2)) of the output vector function of the nonlinear HDS on the output vector function of the linearized HDS for $t_{\max} = 250$. As follows from the data presented in Fig. 5, when changing the overload according to (9) the greatest influence on the output vector function of the nonlinear HDS has parameter μ_4, characterizing the unsteadiness of the system.

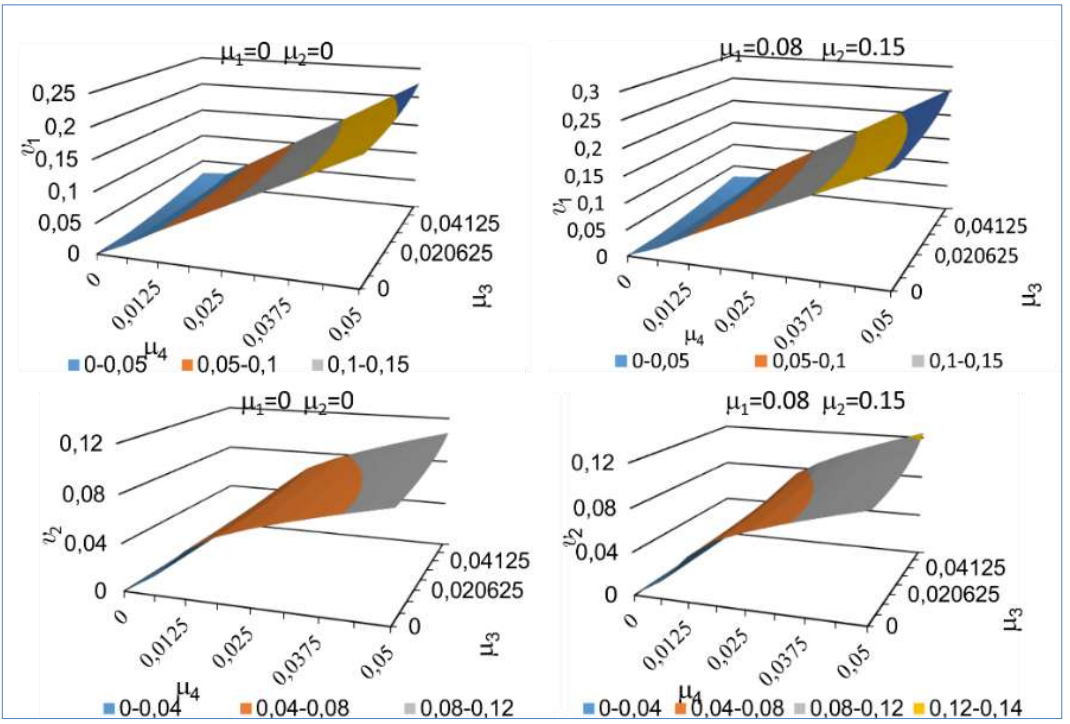

Fig. 5. Maximum and standard deviations

Similar data characterizing the efficiency of stabilization with respect to the vertical, longitudinal axis, as well as the influence of the parameters of nonlinearity and unsteadiness on the output functions of the stabilization system with parameters

$J_0 = 0.00003,\ m_1 = 0.0667,\ J_1 = 0.00009728,\ m_2 = 0.333,\ J_2 = 0.00345,\ a = 0.166667,$
$a_x = 1,\ \gamma = 0.01,\ p_1 = p_6 = 4.098,\ p_2 = p_7 = 9.553,\ p_3 = p_8 = 7.687,\ p_4 = p_9 = 7.714,$
$p_5 = p_{10} = 3.269, J_{1k} = 0.002, J_{2k} = 0.005, J_2 = 2, \mu_1 = 0.08, \mu_2 = 0.2, \mu_3 = 0.04, \mu_4 = 0.05,$
$p_{11} = 0.05, p_{12} = 1$ (6)

are shown in fig. 6-8. Similarly to the previously discussed non-linear stabilization system allows to compensate for the unwanted errors throughout the range of overload (see fig. 5, 6). The greatest influence on the output vector function of the nonlinear HDS has the parameter μ_4, which characterizes the unsteadiness of the system (see fig. 8).

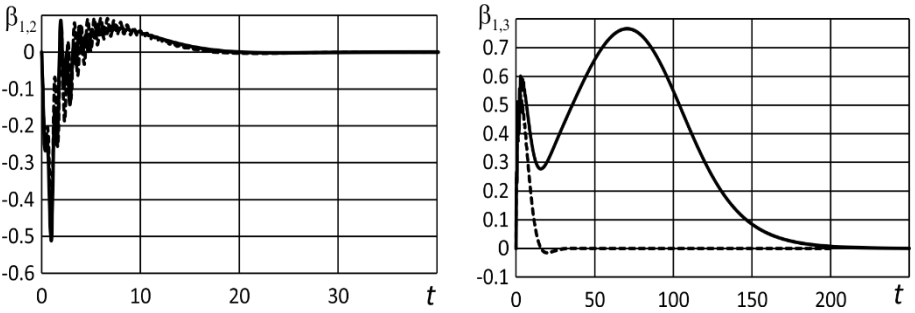

Fig. 6. Stabilization in the vertical direction

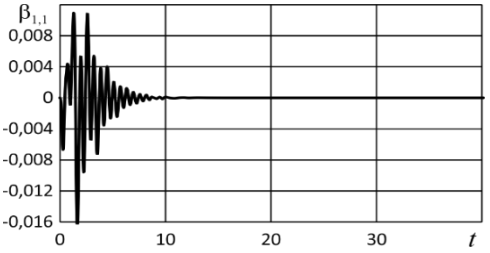

Fig. 7. Stabilization with respect to the longitudinal axis

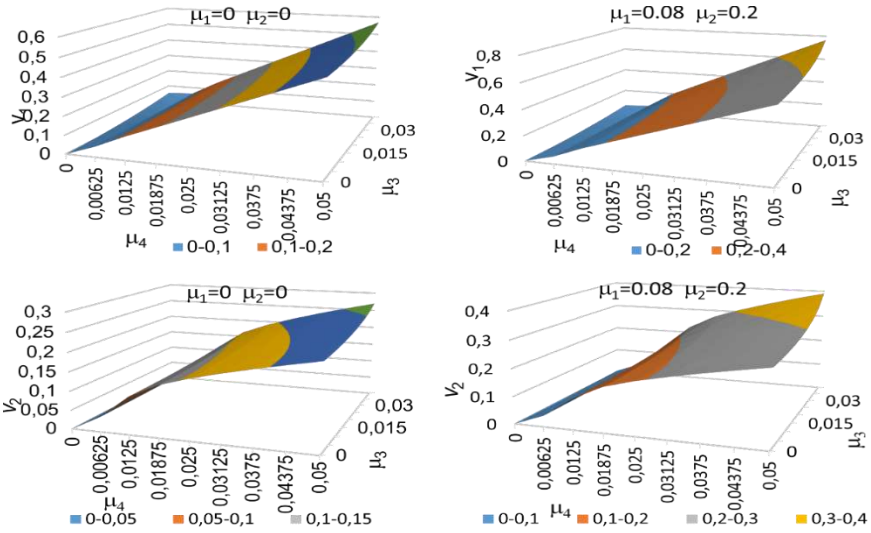

Fig. 8. Maximum and standard deviations

7. Efficiency analysis of parallel algorithms

Consider the effectiveness of the implementation on computer systems with coprocessors Intel Xeon Phi parallel algorithm (2), (3) modeling the effect of typical nonlinearities and unsteadiness on the output functions of the HDS. The data corresponding to the modeling of a nonlinear stabilization system with parameters (5) are presented in Table 1. The calculations were performed on a cluster of faculty of Computer Science and Informational Technologies and Volga Region Center of New Information Technologies of SSU. The four-dimensional grid of change of parameters μ_1, μ_2, μ_3 μ_4 dimension $6 \times 9 \times 9 \times 9$ was used.

Table 1. Modeling of the impact of model nonlinearities and non-stationary, sec.

Grid 6 × 9 × 9 × 9					
Processor, serial/parallel	Test 1	Test 2	Test 3	Test 4	Test 5
Intel Xeon E5-2603 v2, serial	16411	16325	16470	16531	16314
2 processors Intel Xeon E5-2603 v2, OpenMP.	2480	2471	2501	2492	2485
Coprocessor Intel Xeon Phi 5110P, OpenMP	1667	1635	1673	1649	1655
2 coprocessors Intel Xeon Phi 5110P, MPI-MAP/OpenMP	1023	1015	1032	1008	1040

As follows from the Table 1 results, in this case, the use of a single Intel Xeon Phi coprocessor is more efficient than the use of two quad-core CPUs. The most profitable strategy of using coprocessors is parallelization based on OpenMP inside the coprocessor and parallelization based on MPI-MAP between coprocessors. Similar data for the stabilization system with parameters for the stabilization system with a set of parameters (6) are presented in Table 2. And in this case, using one Intel Xeon Phi processor is more efficient than using two quad-core CPUs. The most profitable strategy for the use of coprocessors is the parallelization based on OpenMP within the coprocessor and parallelization based on MPI-MAP between the coprocessors.

Table 2. Modeling of the impact of model nonlinearities and non-stationary, sec.

Grid 6 × 9 × 9 × 9					
Processor, serial/parallel.	Test 1	Test 2	Test 3	Test 4	Test 5
Intel Xeon E5-2603 v2, serial.	9637	9597	9645	9657	9675
2 processors Intel Xeon E5-2603 v2, OpenMP	1443	1470	1430	1412	1467
Coprocessor Intel Xeon Phi 5110P, OpenMP	1052	1042	1063	1037	1055
2 coprocessors Intel Xeon Phi 5110P, MPI-MAP/OpenMP	703	699	710	707	705

Analogical evidence of the effectiveness of the implementation of the parallel algorithm (2), (3) using a single Intel Xeon Phi coprocessor (OpenMP) and two Intel Xeon Phi coprocessors (MPI-MAP – OpenMP) for a more detailed meshes, changing parameters μ_1, μ_2, μ_3 μ_4 are given in Table 3 for stabilization system with parameters (5) and in Table 4 for the stabilization system with parameters (6).

Table 3 - Modeling of the impact of model nonlinearities and non-stationary, sec.

Processor, serial/parallel.	Test 1	Test 2	Test 3	Test 4	Test 5
Grid 6 × 16 × 16 × 16					
Coprocessor Intel Xeon Phi 5110P, OpenMP	8953	9005	8934	8902	8985
2 coprocessors Intel Xeon Phi 5110P, MPI-MAP/ OpenMP	4726	4753	4715	4703	4744
Grid 12 × 16 × 16 × 16					
Coprocessor Intel Xeon Phi 5110P, OpenMP	18132	18243	18025	18187	18053
2 coprocessors Intel Xeon Phi 5110P, MPI-MAP/ OpenMP	9478	9529	9435	9501	9439

Table 4 - Modeling of the impact of model nonlinearities and non-stationary, sec.

Processor, technology of parallelization	Test 1	Test 2	Test 3	Test 4	Test 5
Grid 6 × 16 × 16 × 16					
Coprocessor Intel Xeon Phi 5110P, OpenMP	5671	5634	5654	5754	5698
2 coprocessors Intel Xeon Phi 5110P, MPI-MAP/ OpenMP	2988	2969	2967	3031	3002
Grid 12 × 16 × 16 × 16					
Coprocessor Intel Xeon Phi 5110P, OpenMP	11205	11278	11154	11174	11237
2 coprocessors Intel Xeon Phi 5110P, MPI-MAP/ OpenMP	5777	5809	5735	5741	5798

As follows from the Table 3 and 4 results, with an increase in the average number of nodes on the grid measurement, the multiplicative contribution to the acceleration of the MPI-MAP pattern quickly tends to the number of coprocessors used.

8. Conclusions

The proposed parallel algorithm is effective on cluster systems with nodes using processors with a large number of cores. In particular, it is effective on cluster systems with Intel Xeon Phi coprocessors.

References

[1] Andreichenko D.K., Andreichenko K.P. On the theory of hybrid dynamical systems. Journal of Computer and Systems Sciences International, vol. 39, no. 3, 2000, pp. 383-398.

[2] Andreichenko D.K., Andreichenko K.P. Modeling, analysis and synthesis of combined dynamical systems. Tutorial. Saratov, Rait-Ekspo Publ., 2013. 144 p. (in Russian) / Д.К. Андрейченко, К.П. Андрейченко. Моделирование, анализ и синтез комбинированных динамических систем. Учебное пособие. Саратов, Издательский дом «Райт-Экспо», 2013 г., 144 с.

[3] Andreichenko D.K., Andreichenko K.P., Melnichuk D.V. Pattern MPI-MAP and modeling of nonlinear hybrid dynamical systems. In Proc. of the IV International scientific conference on Problems of control, information processing and transmission, vol. 2, 2015, pp. 19-26 (in Russian) / Д.К. Андрейченко, К.П. Андрейченко, Д.В. Мельничук. Паттерн MPI-MAP и моделирование нелинейных комбинированных динамических систем. В сборнике трудов IV международной научной конференции «Проблемы управления, обработки и передачи информации», т. 2, 2015 г., pp. 19-26

[4] Andreichenko D.K., Andreichenko K.P., Melnichuk D.V., Portenko M.S. Adaptive Algorithm of Parametric Synthesis of Hybrid Dynamical Systems. Izvestiya of Saratov University. New Series. Series: Mathematics. Mechanics. Informatics, vol. 16, issue. 4, 2016, pp. 465–475 (in Russian) / Д.К. Андрейченко, К.П. Андрейченко, Д.В. Мельничук, М.С. Портенко. Адаптивный алгоритм параметрического синтеза комбинированных динамических систем. Известия Саратовского университета, новая серия, серия: Математика. Механика. Информатика, том 16, вып. 4, 2016 г., стр. 465-475.

[5] Kang Q., Träff J.L., Al-Bahrani R., Agraval A., Choundary A., Liao W. Scalable Algorithms for MPI Intergroup Allgather and Allgatherv. Parallel Computing, vol. 85, 2019, pp. 220-230.

[6] Dalcin L., Mortensen M., Keyes D.E. Fast parallel multidimensional FFT using advanced MPI. Journal of Parallel and Distributed Computing, vol. 128, 2019, pp. 137–150

[7] Avdeeva A.N., Puzikova V.V. Application of parallel algorithms for numerical simulation of quasi-one dimensional blood flow. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 2, 2018, pp. 301-316 (in Russian) DOI: 10.15514/ISPRAS-2018-30(2)-15 / Авдеева А.Н., Пузикова В.В. Применение параллельных алгоритмов при численном моделировании кровотока в квазиодномерном приближении. Труды ИСП РАН, том 30, вып. 2, 2018 г., стр. 301-316.

[8] Towara M., Schanen M., Naumann U. MPI-Parallel Discrete Adjoint OpenFOAM. Procedia Computer Science, vol. 51, 2015, pp. 19-28.

[9] Tamada Y. Memory efficient parallel algorithm for optimal DAG structure search using direct communication. Journal of Parallel and Distributed Computing, vol. 119, 2018, pp. 27–35.

[10] Chen C., Pouransari H., Rajamanickam S., Boman E.G., Darve E. A distributed-memory hierarchical solver for general sparse linear systems. Parallel Computing, vol. 74, 2018, pp. 49–64.

[11] Takahashi D. Computation of the 100 quadrillionth hexadecimal digit of π on a cluster of Intel Xeon Phi processors. Parallel Computing, vol. 75, 2018, pp. 1–10.

[12] Cheng X., He B., Lu M., Lau C.T. Many-core needs fine-grained scheduling: A case study of query processing on Intel Xeon Phi processors. Journal of Parallel and Distributed Computing, vol. 120, 2018, pp. 395–404.

[13] Lazarev D.O., Kuzyurin N.N. On-line algorithm for scheduling parallel tasks on related computational clusters with processors of different capacities and its average-case analysis. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 6, 2018, pp. 105-122 (in Russian). DOI: 10.15514/ISPRAS-2018-30(6)-6 / Лазарев Д.О., Кузюрин Н.Н. Алгоритм построения расписаний выполнения параллельных задач на группах кластеров с процессорам различной производительности и его анализ в среднем. Труды ИСП РАН, том 30, вып. 6, 2018 г., стр.105-122.

[14] Gandhi P. S., Borja P., Ortega R. Energy shaping control of an inverted flexible pendulum fixed to a cart. Control Engineering Practice, vol. 56, 2016, pp. 27–36.

Appendix A.

The rotation of the coordinate system is characterized by angles $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \alpha_3)^T$(in order $\alpha_3, \alpha_2, \alpha_1$), and

$$A(\boldsymbol{\alpha}) = \begin{bmatrix} \cos\alpha_3 & -\sin\alpha_3 & 0 \\ \sin\alpha_3 & \cos\alpha_3 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\alpha_2 & 0 & \sin\alpha_2 \\ 0 & 1 & 0 \\ -\sin\alpha_2 & 0 & \cos\alpha_2 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha_1 & -\sin\alpha_1 \\ 0 & \sin\alpha_1 & \cos\alpha_1 \end{bmatrix}$$

$$B(\boldsymbol{\alpha}) = \begin{bmatrix} 1 & 0 & -\sin\alpha_2 \\ 0 & \cos\alpha_1 & \cos\alpha_2\sin\alpha_1 \\ 0 & -\sin\alpha_1 & \cos\alpha_2\cos\alpha_1 \end{bmatrix}$$

In dimensionless variables and parameters the equations of motion of the HDS have the form

$$\boldsymbol{\Omega}_1 = B(\mu_1\boldsymbol{\beta}_1)\dot{\boldsymbol{\beta}}_1, \boldsymbol{\Omega}_2 = A^T(\mu_1\beta_2)\boldsymbol{\Omega}_1 + B(\mu_1\boldsymbol{\beta}_2)\dot{\boldsymbol{\beta}}_2, \omega_{0_x} = -\dot{\beta}_{0,2}\sin(\mu_1\beta_{0,3})$$

$$\omega_{0_y} = \dot{\beta}_{0,2}\cos(\mu_1\beta_{0,3}), \omega_{0_z} = \dot{\beta}_{0,3}, m_1\ddot{\mathbf{r}}_1 = A(\mu_1\boldsymbol{\beta}_1)\mathbf{Q}(0,t) - \mathbf{F}_e +$$

$$+a_x[(1+m_2)A(\mu_1\boldsymbol{\beta}_1)\boldsymbol{\Phi}(0,\boldsymbol{\beta}_0) + m_1\boldsymbol{\Phi}(\boldsymbol{\beta}_1,\boldsymbol{\beta}_0)]$$

$$\boldsymbol{\Phi}(\boldsymbol{\alpha},\boldsymbol{\beta}) = \mu_1^{-1}(A(\mu_1\boldsymbol{\alpha})A(\mu_1\boldsymbol{\beta}) - E)(1,0,0)^T =$$

$$= (\Phi_1(\boldsymbol{\alpha},\boldsymbol{\beta}), \Phi_2(\boldsymbol{\alpha},\boldsymbol{\beta}), \Phi_3(\boldsymbol{\alpha},\boldsymbol{\beta}))^T, E = \mathrm{diag}\{1,1,1\}, J^{(1)} = \mathrm{diag}\{J_{1k}, J_1, J_1\}$$

$$J_0(\dot{\boldsymbol{\Omega}}_1 + \dot{\boldsymbol{\omega}}_0) + J^{(1)}\dot{\boldsymbol{\Omega}}_1 + \mu_1\boldsymbol{\Omega}_1 \times (J_0\boldsymbol{\omega}_0 + J^{(1)}\boldsymbol{\Omega}_1) = \mathbf{M}(0,t) - (M_c^{(1)}, 0, 0)^T$$

$$J_0\left[\dot{\Omega}_{1_y} + \dot{\omega}_{0_y} + \mu_1\left(\Omega_{1_z}\omega_{0_x} - \Omega_{1_x}\omega_{0_z}\right)\right] =$$

$$= M_2^{(c)}\cos\left(\mu_1\beta_{0,3}\right) + M_3^{(c)}\sin\left(\mu_1\beta_{0,3}\right)\sin\left(\mu_1\beta_{0,2}\right)$$

$$J_0\left[\dot{\Omega}_{1_z} + \dot{\omega}_{0_z} + \mu_1\left(\Omega_{1_x}\omega_{0_y} - \Omega_{1_y}\omega_{0_x}\right)\right] = M_3^{(c)}\cos\left(\mu_1\beta_{0,2}\right)$$

$$\alpha_2 = -\mu_1^{-1}\arcsin\left(\mu_1\Phi_3(\boldsymbol{\beta}_2,\boldsymbol{\beta}_1)\right), f_1(z) = \mathrm{tg}z \qquad (7)$$

$$\alpha_3 = \frac{1}{\mu_1}\arcsin\frac{\mu_1\Phi_2(\boldsymbol{\beta}_2,\boldsymbol{\beta}_1)}{\cos(\mu_1\alpha_2)}, \alpha_1 = -\frac{1}{\mu_1}\arcsin\frac{\mu_1\Phi_3^{**}(\boldsymbol{\beta}_2,\boldsymbol{\beta}_1)}{\cos(\mu_1\alpha_2)}$$

$$m_2\mathbf{w}_2 = a_x m_2[\boldsymbol{\Phi}(0,\mu_1(0,\beta_{2,2},\beta_{2,3})^T) - \boldsymbol{\Phi}^*(0,\mu_1\beta_1)] - A(\mu_1(0,\beta_{2,2},\beta_{2,3})^T)\mathbf{Q}(1,t)$$

$$\dot{\mathbf{w}}_2 = A^T(\mu_1\boldsymbol{\beta}_1)\ddot{\mathbf{r}}_1 + \dot{\boldsymbol{\Omega}}_1 \times \mathbf{R}_2 + \mu_1(\boldsymbol{\Omega}_1 \cdot \mathbf{R}_2)\boldsymbol{\Omega}_1 - \mu_1\boldsymbol{\Omega}_1^2\,\mathbf{R}_2 - 2\mu_1\boldsymbol{\Omega}_1 \times \dot{\mathbf{r}}_2 + \ddot{\mathbf{r}}_2$$

$$J^{(2)}\dot{\boldsymbol{\Omega}}_2 + \mu_1\boldsymbol{\Omega}_2 \times J^{(2)}\Omega_2 = -A^T((\mu_1\beta_{2,1},0,0)^T)\mathbf{M}(1,t) +$$

$$+(a,0,0)^T \times A^T((\mu_1\beta_{2,1},0,0)^T)\mathbf{Q}(1,t), J^{(2)} = \mathrm{diag}\{J_{2k}, J_2, J_2\}$$

$$\mathbf{R}_2 = (1+a,0,0)^T + \mu_1\mathbf{r}_2, \Phi^*(\boldsymbol{\alpha},\boldsymbol{\beta}) = \mu_1^{-1}(A^T(\mu_1\boldsymbol{\alpha})A^T(\mu_1\boldsymbol{\beta}) - E)(1,0,0)^T =$$

$$= (\Phi_1^*(\boldsymbol{\alpha},\boldsymbol{\beta}), \Phi_2^*(\boldsymbol{\alpha},\boldsymbol{\beta}), \Phi_3^*(\boldsymbol{\alpha},\boldsymbol{\beta}))^T$$

$$u'_x = \mu_1^{-1}[(1 - \mu_1^2(u'^2_y + u'^2_z))^{1/2} - 1], L_{21} = \mu_1 u'_y, L_{31} = \mu_1 u'_z$$

$$L_{11} = (1 - L_{21}^2 - L_{31}^2)^{1/2}, L_{33} = (1 - L_{31}^2)^{1/2}, L_{12} = -L_{21}/L_{33}, L_{22} = L_{11}/L_{33}$$

$$L_{32} = 0, L_{13} = -L_{31}L_{22}, L_{23} = L_{31}L_{12}, \kappa_1 = u'_z(L_{12}L'_{22} - L_{22}L'_{12})$$

$$\kappa_2 = u'_z L_{22}L'_{11} - u''_y L_{23} - u''_z L_{33}, \kappa_3 = -\frac{u'_y L'_{11}}{L_{33}} + u''_y L_{22}$$

$$\ddot{u}_y + (A^T(\mu_1\boldsymbol{\beta}_1)\ddot{\mathbf{r}}_1) \cdot (0,1,0)^T - (\mu_1\dot{\Omega}_{1_x}u_z - \dot{\Omega}_{1_z}(x + \mu_1 u_x)) +$$

$$+\mu_1[(x + \mu_1 u_x)\Omega_{1_x} + \mu_1 u_z\Omega_{1_z}]\Omega_{1_y} - \mu_1^2(\Omega_{1_x}^2 + \Omega_{1_z}^2)u_y + 2\mu_1(\Omega_{1_x}\dot{u}_z -$$

$$-\Omega_{1_z}\dot{u}_x) = L_{21}(Q'_1 + \mu_1(\kappa_2 Q_3 - \kappa_3 Q_2)) + L_{22}(Q'_2 - \mu_1(\kappa_1 Q_3 - \kappa_3 Q_1)) +$$

$$+L_{23}(Q'_3 + \mu_1(\kappa_1 Q_2 - \kappa_2 Q_1)) - a_x[\Phi_2^*(0,\boldsymbol{\beta}_1) + ((m_2 + 1 - x)u'_y)'] \qquad (8)$$

$$\ddot{u}_z + (A^T(\mu_1\boldsymbol{\beta}_1)\ddot{\mathbf{r}}_1) \cdot (0,0,1)^T + \mu_1\dot{\Omega}_{1_x}u_y - \dot{\Omega}_{1_y}(x + \mu_1 u_x) +$$

$$+\mu_1[(x + \mu_1 u_x)\Omega_{1_x} + \mu_1 u_y\Omega_{1_y}]\Omega_{1_z} - \mu_1^2(\Omega_{1_x}^2 + \Omega_{1_y}^2)u_z - 2\mu_1(\Omega_{1_x}\dot{u}_y -$$

$$-\Omega_{1_y}\dot{u}_x) = L_{31}(Q'_1 + \mu_1(\kappa_2 Q_3 - \kappa_3 Q_2)) + L_{33}(Q'_3 + \mu_1(\kappa_1 Q_2 - \kappa_2 Q_1)) -$$

$$-a_x\left[\Phi_3^*(0,\boldsymbol{\beta}_1) + \left((m_2 + 1 - x)u'_z\right)'\right]$$

$$Q''_1 - \mu_1^2(\kappa_2^2 + \kappa_3^2)Q_1 = \mu_1\{-a_x(m_2 + 1 - x)(\kappa_2^2 + \kappa_3^2) + \kappa'_3 Q_2 - \kappa'_2 Q_3 +$$
$$+2\kappa_3 Q'_2 - 2\kappa_2 Q'_3 - \mu_1\kappa_1(\kappa_2 Q_2 + \kappa_3 Q_3) - ((\dot{u}'_x)^2 + (\dot{u}'_y)^2 + (\dot{u}'_z)^2) +$$
$$+(\Omega_{1_x} L_{11} + \Omega_{1_y} L_{21} + \Omega_{1_z} L_{31})^2 - (\Omega_{1_x}^2 + \Omega_{1_y}^2 + \Omega_{1_z}^2) +$$
$$+2[L_{11}(\Omega_{1_y}\dot{u}'_z - \Omega_{1_z}\dot{u}'_y) - L_{21}(\Omega_{1_x}\dot{u}'_z - \Omega_{1_z}\dot{u}'_x) + L_{31}(\Omega_{1_x}\dot{u}'_y - \Omega_{1_y}\dot{u}'_x)]\}$$

$$u_y(0,t) = 0, u'_y(0,t) = 0, u_y(1,t) = y_2 - a\Phi_2(0,\boldsymbol{\beta}_2), u'_y(1,t) = \cos(\mu_1\beta_{2,2}) \cdot$$
$$\cdot \mu_1^{-1}\sin(\mu_1\beta_{2,3}), u_z(0,t) = 0, u'_z(0,t) = 0, u_z(1,t) = z_2 - a\Phi_3(0,\boldsymbol{\beta}_2),$$
$$u'_z(1,t) = -\mu_1^{-1}\sin(\mu_1\beta_{2,2}), x_2 = u_x(1,t) + a\Phi_1(0,\boldsymbol{\beta}_2),$$
$$Q'_1(0,t) + \mu_1\big(\kappa_2(0,t)Q_3(0,t) - \kappa_3(0,t)Q_2(0,t)\big) = \quad (9)$$
$$= \ddot{\mathbf{r}}_1 \cdot A(\mu_1\boldsymbol{\beta}_1)(1,0,0)^T + a_x\Phi_1^*(0,\boldsymbol{\beta}_1)$$
$$Q'_1(1,t) + \mu_1(\kappa_2(1,t)Q_3(1,t) - \kappa_3(1,t)Q_2(1,t)) =$$
$$= a_x\Phi_1^*(\boldsymbol{\beta}_2,\boldsymbol{\beta}_1) + \mu_1 a(\Omega_{2_\eta}^2 + \Omega_{2_\zeta}^2) + (1,0,0)^T \cdot A^T(\mu_1\boldsymbol{\beta}_2)\mathbf{w}_2$$

$$M_1 = I_k\left(\beta_{2,1} + \gamma\dot{\beta}_{2,1} - \int_0^1 \kappa_1 dx\right), M_2 = \kappa_2 - \gamma\dot{u}''_z, M_3 = \kappa_3 + \gamma\dot{u}''_y \quad (10)$$
$$Q_2 = -M'_3 + \mu_1(\kappa_2 M_1 - \kappa_1 M_2), Q_3 = M'_2 + \mu_1(\kappa_3 M_1 - \kappa_1 M_3)$$
$$\boldsymbol{\beta}_1(0) = \dot{\boldsymbol{\beta}}_1(0) = \boldsymbol{\beta}_2(0) = \dot{\boldsymbol{\beta}}_2(0) = \beta_{0,2}(0) = \dot{\beta}_{0,2}(0) = \beta_{0,3}(0) = \dot{\beta}_{0,3}(0) =$$
$$= \mathbf{r}_1(0) = \dot{\mathbf{r}}_1(0) = y_2(0) = \dot{y}_2(0) = z_2(0) = \dot{z}_2(0) = \quad (11)$$
$$= u_y(x,0) = \dot{u}_y(x,0) = u_z(x,0) = \dot{u}_z(x,0) = 0$$

Here (7) are ordinary differential equations, (8) are partial differential equations, (9) are boundary conditions, (10) are constraint's conditions, (11) are initial conditions, $()' = \partial()/\partial x$.

Информация об авторе / Information about author

Дмитрий Вадимович МЕЛЬНИЧУК получил степень магистра по направлению «Прикладная математика и информатика» в Саратовском национальном исследовательском государственном университете имени Н.Г. Чернышевского, Саратов, Россия. Его исследовательские интересы включают математическое моделирование, моделирование управляемых комбинированных динамических систем, параллельные алгоритмы и параллельные вычислительные технологии.

Dmitry Vadimovich MELNICHUK received a master's degree in «Applied mathematics and Informatics» at Saratov State University, Saratov, Russia. His research interests include mathematical modeling, simulation of controlled hybrid dynamic systems, parallel algorithms, and parallel computing.