ISSN 1818-1015
eISSN 2313-5417

2021 Volume 28 No 4

MODELING AND ANALYSIS OF INFORMATION SYSTEMS

SCIENTIFIC JOURNAL

Start date of publication — 1999
Published quarterly

FOUNDER
P.G. Demidov Yaroslavl State University

EDITORIAL OFFICE
14 Sovetskaya str., Yaroslavl 150003, Russian Federation

Website: http://mais-journal.ru
E-mail: mais@uniyar.ac.ru
Phone: +7 (4852) 79-77-73

2021 Том 28 № 4

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ

НАУЧНЫЙ ЖУРНАЛ

Издается с 1999 года
Выходит 4 раза в год

УЧРЕДИТЕЛЬ
федеральное государственное бюджетное образовательное учреждение высшего образования
«Ярославский государственный университет им. П. Г. Демидова»

РЕДАКЦИЯ
ул. Советская, 14, Ярославль, 150003, Российская Федерация
Website: http://mais-journal.ru
E-mail: mais@uniyar.ac.ru
Телефон: +7 (4852) 79-77-73

Свидетельство о регистрации СМИ ПИ № ФС 77–66186 от 20.06.2016 выдано Федеральной службой по надзору в сфере связи, информационных технологий и массовых коммуникаций. Подписной индекс — 31907 в Объединенном каталоге «Пресса России». Технический редактор, компьютерная вёрстка – М. С. Каряева. Корректор английского текста – М. С. Комар. Подписано в печать 10.12.2021. Дата выхода в свет 30.12.2021. Формат 200×265 мм. Объем 146 с. Тираж 40 экз. Свободная цена. Заказ 068/021. Адрес типографии: ул. Советская, 14, оф. 109, Ярославль, 150003, Россия. Адрес издателя: Ярославский государственный университет им. П. Г. Демидова, ул. Советская, 14, Ярославль, 150003, Россия.
Содержание предназначено для детей старше 12 лет.

Contents

Содержание

От редакторов выпуска

В. А. Захаров[1,2], *Н. В. Шилов*[3]

1 ВШЭ
2 ИСП РАН
3 Университет Иннополис

From the Editors of the Issue

V. A. Zakharov[1,2], *N. V. Shilov*[3]

1 HSE
2 ISP RAS
3 Innopolis University

04–05 ноября 2021 г. в Иннополисе прошел двенадцатый международный научно–исследовательский семинар **«Семантика, спецификация и верификация программ: теория и приложения»** (12-th Workshop on Program Semantics, Specification and Verification: Theory and Applications, PSSV 2021). Ввиду непростой эпидемической обстановки в стране заседания семинара проводились в формате видеоконференции. Организатором семинара выступила лаборатория программной инженерии университета Иннополис (рук. лаборатории Н. В. Шилов).

Первая сессия семинара была посвящена юбилею выдающегося ученого, специалиста в области информационных технологий и программирования, заведующего отделом технологии программирования Института системного программирования им. В. П. Иванникова, профессора Московского государственного университета им. М. В. Ломоносова и Высшей Школы Экономики Александра Константиновича Петренко. В своем докладе, озаглавленном «The position of formal methods in nowadays software industrial development», А. К. Петренко проанализировал тенденции развития и практического применения формальных методов в разработке промышленного программного обеспечения, рассказал о своем опыте использования этих методов в решении разнообразных задач системного программирования, начиная от работ по созданию программного обеспечения для советской космической программы «Буран» до недавних прикладных исследований по проектированию и верификации критических по безопасности и надежности программных систем.

Редколлегия журнала присоединяется к теплым пожеланиям, высказанным участниками семинара PSSV 2021 в адрес Александра Константиновича Петренко.

На юбилейной сессии семинара также прозвучали доклады ближайших коллег

А. С. Камкин: *High-Level Synthesis of Computing Systems: Motivation, Challenges, and Existing Solutions*,
В. В. Кулямин: *Formal Security Models*,
А. В. Хорошилов: *Verification of operating systems*.

Программа семинара PSSV 2021 включала

- 5 регулярных докладов

  Anton Gnatenko, Vladimir Zakharov: *Satisfiability and model checking for one extension of Linear Time Temporal Logic*,
  Alexander Bolotov, Alex Abuin Yepes, Paqui Lucio and Montserrat Hermo: *Certifying Proofs and Models for CTL Satisfiability and Model Checking*,
  Hans de Nivelle: *A Recursive Inclusion Checker for Recursively Defined Subtypes*,
  Thomas Baar, Horst Schulte: *On the Need to Support Vectors and Matrixes in KeYmaera*,
  Dmitry Kondratyev: *Towards automatic deductive verification of C programs with Sisal loops using C-lightVer system*,
- 7 кратких сообщений

  Thanh-Hai Tran, Igor Konnov and Josef Widder: *EDTL4CSRS: Event-Driven Temporal Logic for Control Software Requirements Specification*,
  Julio Cesar Carrasquel: *Validating Real Behavior of Agents in Trading Systems using Nested Petri Nets*,

Anton Zavyalov, Sergey Staroletov: *Flovver: A Graphical Functional Language with a Compiler Focused on Recursion Optimization*,
Mohammadsadegh Mohagheghi, Khayyam Salehi: *Statistical Verification of Qualitative Reachability Properties for Markov Decision Processes*,
Andrei Klimov: *On Recent Achievements in Theory of Names to be Used in Semantics of Object-Oriented Languages*,
Andrew Mironov: *Mathematical model and method for verifying parallel programs*,
Nikolay Shilov, Dmitry Kondratyev, Boris Faifel: *Platform-independent model of fix-point arithmetic for verification of the standard mathematical functions*,

- 4 лекции приглашенных докладчиков
  Nataliya O. Garanina (Laboratory of Theoretical Programming of A.P. Ershov Institute of Informatics Systems, Novosibirsk, Russia): *The Optimization Problem with Model Checking*,
  Dmitry A. Kondratyev (Laboratory of Theoretical Programming of A.P. Ershov Institute of Informatics Systems, Novosibirsk, Russia): *Automatic deductive verification of C programs using the C-lightVer system*,
  Alexandr V. Naumchev (Innopolis University, Russia): *Security audit of code using contracts and program proving*,
  Nikolai D. Kudasov (Innopolis University, Russia): *Nameless and scope-safe: de Bruijn notation as a nested datatype*.

В своих выступлениях участники семинара рассказали о результатах недавно завершенных научных работ и о продолжающихся исследованиях, посвященных развитию и применению математических методов для разработки программного обеспечения, вычислительной и телекоммуникационной аппаратуры, созданию новых математических моделей в области информатики, развитию и внедрению перспективных средств программирования. Большое внимание было уделено методам дедуктивной проверки правильности программ, задачам верификации моделей программ, новым методам анализа типов данных, а также вопросам построения формальных моделей информационных систем. Данный выпуск журнала включает 6 статей участников семинара PSSV 2021, рекомендованных программным комитетом семинара к публикации в журнале.

T. Baar и H. Schulte исследовали вопрос о применении программно-инструментального средства верификации программ на основе логики Хоара KeYmaeraX для обоснования свойств поведения гибридных систем. Эти системы определяются совокупностью дискретных и непрерывных переменных, значения которых могут изменяться скачком или постепенно. Одной из многообещающих областей применения KeYmaeraX являются системы управления с обратной связью. На примере одной из таких управляющих систем авторы показывают, как можно получить строгое математическое доказательство асимптотической устойчивости ее поведения с использованием средства дедуктивной верификации KeYmaeraX. В статье также обсуждаются некоторые открытые вопросы, связанные с формализацией требования асимптотической устойчивости гибридных систем.

В статье Н. О. Гараниной и С. П. Горлача представлен новый подход к автоматизации поиска оптимальных параметров структуры многопроцессорных программ, предназначенных для параллельной обработки данных. Для решения этой задачи авторы предлагают применять средства верификации моделей вычислительных систем, способные формировать контрпримеры по результатам проверки требований правильного поведения программ. Описанный в статье метод поиска оптимальных параметров конструкции вычислительных систем при помощи построения контрпримеров предусматривает представление исполнения абстрактной программы на абстрактной

модели многопроцессорной системы, формулировку свойства оптимальности в виде требования безопасности вычислений полученной абстрактной модели, применение алгоритма верификации формальной модели и выбор оптимальных значений параметров настройки вычислительной системы на основании анализа контрпримеров, полученных в ходе верификации.

В статье А.Р. Гнатенко и В.А. Захарова установлены оценки вычислительной сложности задач верификации автоматных моделей последовательных реагирующих систем и проверки выполнимости формул параметризованного расширения темпоральной логики линейного времени, которое авторы описали в своих более ранних работах. При помощи теоретико-автоматного подхода удалось показать, что эффективные решения обеих рассматриваемых задач можно получить, используя лишь полиномиально ограниченный объем памяти. В свете ранее полученных результатов это означает, что обе задачи являются PSPACE-полными.

Статья Д.А. Кондратьева посвящена исследованию одной из задач, которая возникла при выполнении долгосрочного проекта, проводимого в Институте систем информатики СО РАН, по созданию программно-инструментального комплекса C-lightVer для дедуктивной верификации С-программ. Эта задача связана с попытками расширить область применения проектируемого средства дедуктивной верификации и использовать его для проверки правильности программ, выполненных в системе облачного параллельного программирования CPPS, которая также разрабатывается в ИСИ СО РАН.

В статье А.М. Миронова излагается новая математическая модель распределенных вычислительных систем, предназначенная для верификации программ, которые выполнены с использованием программного интерфейса параллельного программирования MPI. Параллельная программа моделируется системой вычислительных процессов, взаимодействующих путем асинхронной передачи и приема сообщений по каналам связи. Главным преимуществом предложенной автором модели является возможность моделирования и верификации параллельных программ с неограниченно большим числом последовательных взаимодействующих процессов. На основе предложенной модели была проведена верификация MPI программы перемножения матриц.

H. de Nivelle предложил в своей статье оригинальный логический подход к построению систем типов данных, допускающий введение зависимых и индуктивно определяемых типов. Эта работа является частью проекта по разработке нового языка программирования, подходящего для применения в области математической логики. Так как логические формулы представляют собой древовидные структуры с множеством разнотипных конструкторов, алгоритмы их обработки должны учитывать согласование типов различных узлов таких деревьев. Наиболее часто такое согласование осуществляется путем сопоставления с образцом. Автор статьи предлагает иной подход: вместо сопоставления деревьев с разными шаблонами, соответствующими определениями типов, он предлагает использовать аппарат логического (табличного) вывода, исходя из системы определений типов и подтипов.

MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 28, NO. 4, 2021
journal homepage: www.mais-journal.ru

THEORY OF COMPUTING

Notes on Recent Achievements in Proving Stability using KeYmaeraX

T. Baar[1], H. Schulte[1]

DOI: 10.18255/1818-1015-2021-4-326-336

[1]HTW Berlin, 75A Wilhelminenhofstraße, Berlin 12459, Germany.

MSC2020: 68N30
Research article
Full text in English

Received November 15, 2021
After revision December 1, 2021
Accepted December 8, 2021

KeYmaeraX is a Hoare-style theorem prover for hybrid systems. A hybrid system can be seen as an aggregation of both discrete and continuous variables, whose values can change abruptly or continuously, respectively. KeYmaeraX supports only variables having the primitive type bool or real.
Due to the mixture of discrete and continuous system elements, one promising application area for KeYmaeraX are closed-loop control systems. A closed-loop control system consists of a plant and a controller. While the plant is basically an aggregation of continuous variables whose values change over time accordingly to physical laws, the controller can be seen as an algorithm formulated in a classical programming language.
In this paper, we review some recent extensions of the proof calculus applied by KeYmaeraX that make formal proofs on the stability of dynamic systems more feasible. Based on an example, we first introduce to the topic and prove asymptotic stability of a given system in a hand-written mathematical style. This approach is then compared with a formal encoding of the problem and a formal proof established in KeYmaeraX. We also discuss open problems such as the formalization of asymptotic stability.

Keywords: cyber physical system; control theory; lyapunov function; imperative programming language

INFORMATION ABOUT THE AUTHORS

Thomas Baar correspondence author	orcid.org/0000-0002-8443-1558. E-mail: thomas.baar@htw-berlin.de Professor.
Horst Schulte	orcid.org/0000-0001-5851-3616. E-mail: horst.schulte@htw-berlin.de Professor.

For citation: T. Baar and H. Schulte, "Notes on Recent Achievements in Proving Stability using KeYmaeraX", *Modeling and analysis of information systems*, vol. 28, no. 4, pp. 326-336, 2021.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 28, № 4, 2021
сайт журнала: www.mais-journal.ru

THEORY OF COMPUTING

Замечания о последних достижениях в доказательстве устойчивости с использованием KeYmaeraX

Т. Баар[1], Х. Шульте[1]

DOI: 10.18255/1818-1015-2021-4-326-336

[1]Университет прикладных технических и экономических наук г. Берлина, ул. Вилхелминенхофштрассе, 75А, г. Берлин 12459 Германия.

УДК 004.942
Научная статья
Полный текст на английском языке

Получена 15 ноября 2021 г.
После доработки 1 декабря 2021 г.
Принята к публикации 8 декабря 2021 г.

KeYmaeraX – это доказательство теорем в стиле Хоара для гибридных систем. Гибридную систему можно рассматривать как совокупность дискретных, так и непрерывных переменных, значения которых могут изменяться резко или непрерывно соответственно. KeYmaeraX поддерживает только переменные, имеющие примитивный тип bool или real.

Благодаря сочетанию дискретных и непрерывных элементов системы, одной из перспективных областей применения KeYmaeraX являются системы управления с замкнутым контуром. Система управления с замкнутым контуром состоит из установки и контроллера. В то время как установка в основном представляет собой совокупность непрерывных переменных, значения которых меняются со временем в соответствии с физическими законами, контроллер можно рассматривать как алгоритм, сформулированный на классическом языке программирования.

В этой статье мы рассмотрим некоторые недавние расширения исчисления доказательств, применяемые KeYmaeraX, которые делают формальные доказательства устойчивости динамических систем более выполнимыми. Основываясь на примере, мы сначала познакомимся с темой и докажем асимптотическую устойчивость данной системы.

Ключевые слова: киберфизическая система; теория управления; функция Ляпунова; императивный язык программирования

ИНФОРМАЦИЯ ОБ АВТОРАХ

Томас Баар
автор для корреспонденции
orcid.org/0000-0002-8443-1558. E-mail: thomas.baar@htw-berlin.de
Профессор.

Хорст Шульте
orcid.org/0000-0001-5851-3616. E-mail: horst.schulte@htw-berlin.de
Профессор.

Для цитирования: T. Baar and H. Schulte, “Notes on Recent Achievements in Proving Stability using KeYmaeraX”, *Modeling and analysis of information systems*, vol. 28, no. 4, pp. 326-336, 2021.

1. Introduction

The verification of software has made huge progress over the last 20 years, but is still considered to be challenging [1]. Many software verification systems such as KeY, FramaC, VeriFast, Dafny and others rely on a classical Hoare-style calculus [2] for verifying pre-/post-conditions.

The theorem prover KeYmaeraX and its underlying Differential Dynamic Logic [3] were successful in extending a traditional Hoare-style calculus with capabilities to reason also on the dynamics of continuous functions, which are specified by ordinary differential equations (ODEs). More precisely, KeYmaeraX allows to verify so-called *hybrid programs*[1] to be correct in the sense of fulfilling given pre-/post-condition contracts. The language of hybrid programs is a very simple traditional imperative while-language with *assignment*, *sequential composition*, *conditional execution*, and *iteration* as basic programming constructs. In addition, there is support for non-determinism (non-deterministic choice, non-deterministic iteration) and a very special construct bridging the gap to evolving functions called *evolving state*. More details on KeYmaeraX and its supported syntax can be found in tutorial [4]. In [5, 6], we have analyzed some deficiencies of the input syntax and present a number of introductory examples.

Control theory is an engineering discipline aiming at analyzing dynamic systems in general. A dynamic system is modeled by a set of state variables, which typically change their value continuously over time. In many cases, this change is described by ordinary differential equations (ODEs).

Well understood and widely applied are linear dynamic systems, whose dynamic can be described as $\dot{x} = Ax$, where x is a vector of *state variables*, A the so-called *system matrix*, and $\dot{x}$ denotes the *derivation of* x. The stability of linear systems is well-understood and proving it for a given concrete system usually requires few mathematical arguments: Common techniques are finding a *Lyapunov function* [7, 8] or analyzing the system matrix A together with *its eigenvalues*.

When modeling real world processes, pure linear systems are often not sufficient. However, the overall system can be modeled in many cases as a system switching through multiple linear subsystems [9]. The switching condition can be simple (as in our running examples shown below) or more elaborate. In any case, we can formulate such conditions using an imperative programming language.

While classical control theory is very successful in analyzing pure linear systems, its mathematical approaches do not work well when applied to composed system. If the composition scheme is restricted to few switching schemes, the mathematical analysis remains feasible but becomes cumbersome[9]. A more elegant solution for this problem could be to encode the composition rules for aggregating the resulting system in terms of an imperative programming language. To analyze such composed systems, one would obviously need a notion of the underlying semantics of the used programming language.

The theorem prover KeYmaeraX is able to capture both the continuous dynamics of systems as well as algorithms for switching the current mode. We will review in this paper some recent achievements that will make the formal verification of system stability more feasible. We also discuss some shortcomings of the dynamic logic and the proof calculus underlying KeYmaeraX.

This paper is organized as follows: Section 2 presents two linear systems and then two different compositions of it. Though the composition algorithms differ only very slightly, the two resulting systems differ dramatically. One of them remains stable, the other one becomes instable. In Section 2, we also give a mathematical proof for the stability of the first composed system. In Section 3, we provide an encoding of the running examples for KeYmaeraX and outline the formal proof within KeYmaeraX. We also report on yet unresolved obstacles such as the formulation of the verification goal (asymptotic stability of the overall system) within the Hoare-style logic supported by KeYmaeraX. In Section 4 we review related work and Section 5 concludes the paper.

[1] There is a graphical version of these programs (basically the Control Flow Graph) called *hybrid automaton*.

2. Running Example

Given is a periodic linear time invariant (LTI) dynamic system in state space form [10]

$$\dot{x}(t) = \begin{pmatrix} 0 & 1 \\ -a & 0 \end{pmatrix} x(t) = A\,x(t)\,, \qquad x(t=0) = x_0 \tag{1}$$

with the state vector $x = (\,x_1\,,\,x_2\,)^T$, the system matrix A, the parameter $a \in \mathbb{R}_0$ and the initial condition as $x_0 \in \mathbb{R}^2$. Based on the theory of ordinary differential equations (ODEs) [11], the periodicity of the solution of (1) with

$$x(t) = x(t+T)\,, \quad T = \frac{2\pi}{\sqrt{a}} \tag{2}$$

is related to the eigenvalues of A denoted as $\lambda = \mathrm{eig}(A) \in \mathbb{C}^2$ with $\lambda_1 = -j\sqrt{a}$ and $\lambda_2 = +j\sqrt{a}$. Let us now consider two systems

$$Sys1 \equiv \dot{x}(t) = \underbrace{\begin{pmatrix} 0 & 1 \\ -a_1 & 0 \end{pmatrix}}_{A_1} x(t)\,, \qquad Sys2 \equiv \dot{x}(t) = \underbrace{\begin{pmatrix} 0 & 1 \\ -a_2 & 0 \end{pmatrix}}_{A_2} x(t)\,, \tag{3}$$

where the parameters a_i for $i = 1, 2$ satisfy the condition

$$0 < a_1 < 1 < a_2 \quad . \tag{4}$$

For illustration, we consider the numerical examples of the form (3) satisfying (4) with

$$a_1 = \frac{1}{4}\,, \qquad a_2 = \frac{9}{4}\,.$$

Figure 1 and 2 show the particular solution and phase diagram for the initial value $x_0 = (\,0\,,\,3\,)^T$. The analytically determined period in (2) of

$$T_1 = \frac{2\pi}{\sqrt{a_1}} = 4\,\pi\,, \qquad T_2 = \frac{2\pi}{\sqrt{a_2}} = \frac{4\,\pi}{3}$$

corresponds to the numerical solution. Let us now examine the dynamics of a system which switches

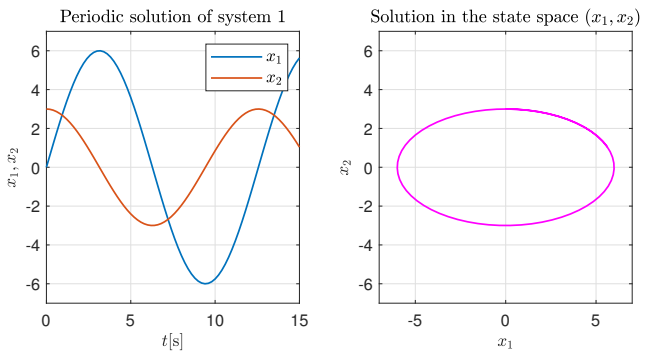


Fig. 1: Particular solution $x(t)$ and in the state space of $Sys1$ for $x_0 = (\,0\,,\,3\,)^T$

between the two systems (3) as a function of the current state x. For this purpose, the switching function

$$switch(x) = x_1\,x_2 \tag{5}$$

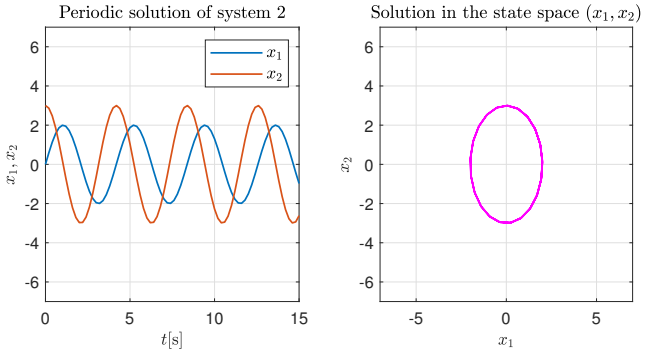


Fig. 2: Particular solution $x(t)$ and in the state space of $Sys2$ for $x_0 = (0, 3)^T$

is defined to divide the state space into two non-convex subspaces

$$X_1 = \{ x(t) \,|\, switch(x) < 0 \}, \qquad X_2 = \{ x(t) \,|\, switch(x) \geq 0 \}. \tag{6}$$

Based on these definitions, two variants of a switching system can now be specified. The first switching system $SwSys1$ is given as

$$SwSys1 \equiv \dot{x}(t) = \begin{cases} A_1 x(t), & switch(x) < 0 \\ A_2 x(t), & switch(x) \geq 0 \end{cases}. \tag{7}$$

The second system $SwSys2$ inverts the switching condition from the first system and is given as

$$SwSys2 \equiv \dot{x}(t) = \begin{cases} A_1 x(t), & switch(x) \geq 0 \\ A_2 x(t), & switch(x) < 0 \end{cases}. \tag{8}$$

Analogous to the previous LTI systems, the switching systems have the same initial condition x_0. The particular solution of $SwSys1$ (7) is given in Figure 3. One can clearly see that this system is asymptotic stable for the given initial condition, i.e. $\lim_{t \to \infty} x(t) \longrightarrow 0$ for $x_0 = (0, 3)^T$. By quadrant switching, two periodic systems are joined to form an asymptotically stable system.

The opposite effect occurs in the second switching system $SwSys2$ (8) if the switching condition is reversed. Here, the amplitudes of $x_1(t), x_2(t)$ fastly grow and exceed any bounds. Note that the diagrams of Figure 4 have on their axis much greater values than all the other diagrams.

In order to understand the effect, it is worthwhile to analyze the phase diagrams for both $SwSys1$ and $SwSys2$ (cmp. Fig. 5). As one can see at the left part, system $SwSys1$ is enforced in the right-upper and left-lower quadrants (note that $switch(x) \geq 0$ holds) to have a greater value change for x_2 than for x_1. For example, when entering the right-upper quadrant, the value for x_1 is 0 and let v be the value for x_2. When leaving this quadrant, the value for x_2 is now 0 and x_1 has a value, but this is smaller than v. An analogous behavior we have in the right-lower and left-upper quadrant, where x_1 changes more than x_2. In summary, whenever the trajectory of $SWSys1$ crosses the diagram axes ($x_1 = 0$ or $x_2 = 0$), the non-zero value of the coordinate (alternating x_1, x_2) form a monotonically decreasing series.

For system $SwSys2$, the opposite is true: When crossing the axes, the non-zero component of the coordinate becomes larger. Thus, $SwSys2$ is not stable.

To sum up, combining two periodic systems $Sys1, Sys2$ to a switched system can result both in an asymptotic stable system $SwSys1$ or in an unstable system $SwSys2$. The difference in the definition of $SwSys1$ and $SwSys2$ is rather marginal. Therefore, it would be very helpful to have a verification tool able to check formally, whether the resulting system is stable or not.

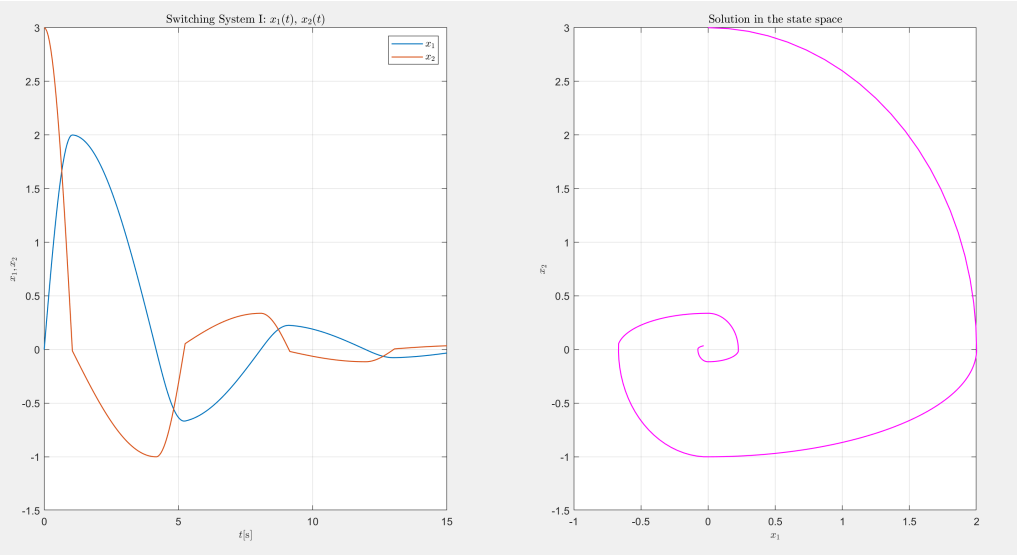


Fig. 3: Particular solution of $SwSys1$ (7) for $x_0 = (0, 3)^T$

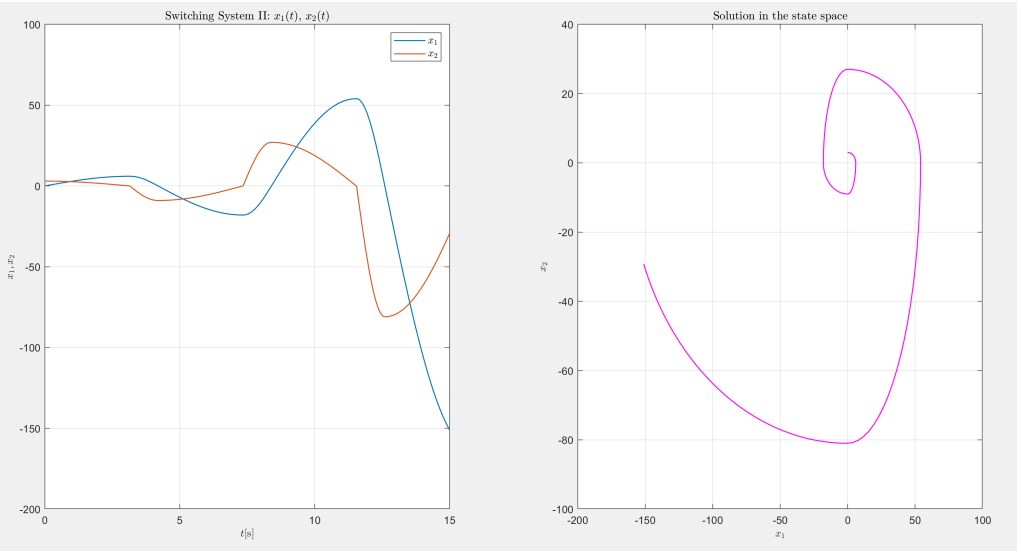


Fig. 4: Particular solution of $SwSys2$ (8) for $x_0 = (0, 3)^T$

2.1. Lyapunov Stability Analysis

A system is called to be *stable*, iff the system will never leave an ϵ region around the origin (i.e. $|x| < \epsilon$) when started in an appropriate δ region ($|x| < \delta$). Note that δ can be freely chosen once ϵ has been fixed. Both δ and ϵ must be positive ($\epsilon > 0, \delta > 0$). A formalization of this definition is presented below in (16).

A system is called to be *asymptotic stable*, iff in addition $lim_{t\to\infty} x(t) = 0$ holds. Figure 6 shows a system, which is stable but not asymptotic stable.

We proceed with a mathematical proof for the stability of the first switching system $SwSys1$ using Lyapunov's direct method [7, 8, 12]. We start with considering one - heuristically chosen - Lyapunov function candidate for the original systems $Sys1, Sys2$

$$V(x) = x_1^2 + x_2^2 \qquad \forall x \neq 0 \tag{9}$$

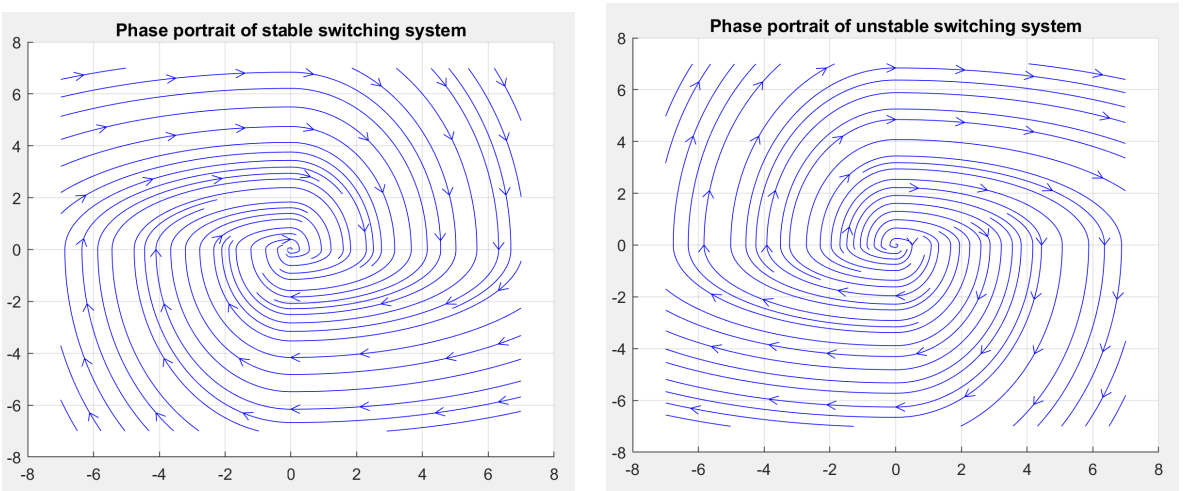


Fig. 5: Phase diagram for $SwSys1/SwSys2$

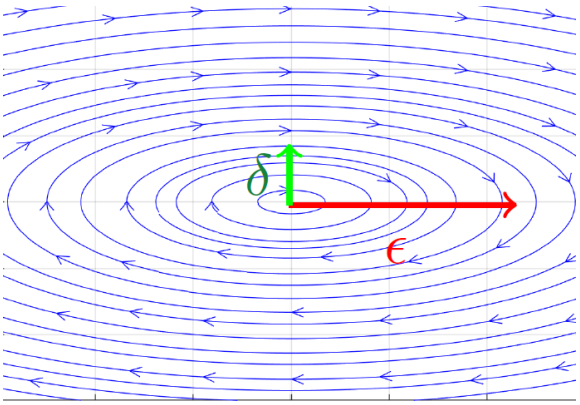


Fig. 6: Definition of Stability

The time derivative of the function $V(x)$ results in

$$\dot{V}(x) = 2\, x_1\, \dot{x}_1 + 2\, x_2\, \dot{x}_2\ . \tag{10}$$

Substituting $\dot{x}_1$ and $\dot{x}_2$ by the right hand side of the LTI systems in (3) with $0 < a_1 < 1 < a_2$ two Lyapunov function candidates are obtained for $Sys1, Sys2$:

$$\dot{V}_1(x) = 2\, x_1\, x_2\, (1 - a_1)\,, \qquad \dot{V}_2(x) = 2\, x_1\, x_2\, (1 - a_2) \tag{11}$$

Each separate function $\dot{V}_i(x)$ does not satisfy the requirement of a Lyapunov function with $\dot{V}_i(x) < 0.\ \forall x$. This is valid only segment-wise and with the introduced switching function (5) and switching condition (6) of x it follows that V_{SwSys1} is a Lyapunov function with

$$\dot{V}_{SwSys1} = \begin{cases} 2\, x_1\, x_2\, (1 - a_1) < 0\,, & switch(x) \leq 0 \\ 2\, x_1\, x_2\, (1 - a_2) < 0\,, & switch(x) \geq 0 \end{cases} \tag{12}$$

where $\dot{V}_{SwSys1} < 0$ holds for all $x \neq 0$. The switching condition in (12) corresponds to the one in system $SwSys1$ (7) which therefore proves the asymptotic stability.

3. Running Example for KeYmaeraX

In this section, we will present an elegant and succinct specification of both the linear and the switched system presented in the previous section using the input formalism of KeYmaeraX. In KeYmaeraX, a dynamic system is described in form of a program α.

As a first attempt, the linear system $Sys1$ might look as a program could as follows:

$$\alpha_{Sys1} \equiv \{x' = A_1 x\} \tag{13}$$

Unfortunately, KeYmaeraX does not allow the usage of state matrix A_1 since all constants and variables have to have a primitive type (real, bool). Also, the state vector x cannot be directly used and must be split into its components x_1, x_2. Thus, we have to rewrite our first attempt (13) as

$$\alpha_{Sys1} \equiv \{x1' = x2, x2' = -a_1 * x1\} \tag{14}$$

The system $Sys1$ can be described in KeYmaeraX by just one evolving state. For the system $SwSys1$, we combine the two evolving states representing $Sys1$ and $Sys2$ using the non-deterministic choice construct (operator ++) and enclose this by a non-deterministic iteration (operator $*$):

$$\begin{aligned} \alpha_{SwSys1} \equiv \quad & (\\ & \quad \{x1' = x2, x2' = -a_1 * x1 \;\&\; switch(x) \leq 0\} \\ & ++ \\ & \quad \{x1' = x2, x2' = -a_2 * x1 \;\&\; switch(x) \geq 0\} \\ &)* \end{aligned} \tag{15}$$

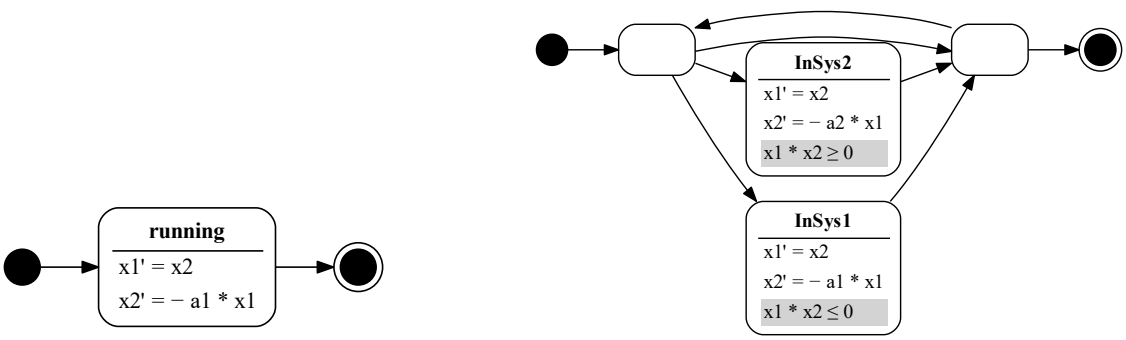


Fig. 7: System descriptions of $Sys1$, $SwSys1$ in KeYmaeraX (as hybrid automaton)

The graphical version of the programs (13) and (15) are shown in Figure 7.

3.1. Proving stability using KeYmaeraX

In terms of differential dynamic logic, the stability of system $SwSys1$ formulates as:

$$\forall \epsilon \, (0 < \epsilon \longrightarrow \exists \delta \, (0 < \delta \wedge x_1^2 + x_2^2 < \delta \longrightarrow [\alpha_{SwSys1}] \, x_1^2 + x_2^2 < \epsilon)) \tag{16}$$

Establishing a formal proof using KeYmaeraX for such a property of system $SwSys1$ remained for quite a long time rather a challenge. This was due to the fact that the proof's fundamental argument - the Lyapunov function $V(x) = x_1^2 + x_2^2$ - could not be encoded directly when establishing the proof.

However, Tan and Platzer report in their very recent paper [13], how the proof calculus has been recently conservatively extended, i.e. new proof rules have been derived and can be used now directly within proof tactics. One of the new proof rules looks as follows:

$$\mathsf{Lyap}_{>}\ \frac{\vdash f(0) = 0 \land v(0) = 0 \qquad \vdash \exists \gamma > 0 \forall x(|x|^2 \leq \gamma^2 \longrightarrow v > 0 \land v' < 0)}{\vdash AStab(x' = f(x))}$$

The formal proof for the stability of the switched system $SwSys1$ has been published in [14] and is available online[2].

Please note, that (16) only formalizes stability but not asymptotic stability. To formulate asymptotic stability, one would need to encode a situation that is far in the future ($lim_{t\to\infty}$) as asymptotic stability means that there will be a point in time, after which the system will always remain within an ϵ region. How can we catch this point in time in a formula defining asymptotic stability? In order to formalize asymptotic stability, one has to choose - probably - a different form than the usual invariant $\phi \longrightarrow [\alpha]\psi$ that has been successfully applied for formalizing stability. This problem has to remain as an open question here. Note, however, that also the conclusion of [14] considers the formalization of asymptotic stability within KeYmaeraX as an unsolved problem and future work.

4. Related Work

In his landmark paper published in 1892 (see [7] for a French and [8] for an English translation), A.M. Lyapunov identifies and describes mathematical tools for analyzing dynamic systems. A Lyapunov function for a given system is an energy measure that has to be decreasing/non-increasing as the system evolves over time. Once an appropriate Lyapunov function is found, it can witness the stability of the system.

Much research has gone into finding suitable Lyapunov functions automatically. Some numerical approaches [15], [16] are based on sum-of-square programming techniques while other approaches exploit Gröbner basis [17], Lie derivatives [18], or constraint solving techniques [19]. Switched systems [20], [9], [21] can vary considerably in their switching mechanics (see also [14] for an overview). A stability proof for switched system often requires to find more sophisticated Lyapunov functions taken all different system modes into account. In [22], a more relaxed notion of stability is presented together with a verification methods based on model checking.

5. Conclusion

In this paper, we report on experiences we gained when merging verification techniques from two engineering disciplines: control theory and software engineering.

Control theory has developed numerous techniques to verify certain properties of modeled systems. One of the most important properties is stability. One fundamental verification technique is finding a Lyapunov function as an upper bound for the system's state change. When the Lyapunov function decreases monotonously, the system changes will become smaller and smaller over time and the system converts towards a stable point. Another often used verification technique for linear system is the analysis of the eigenvalues of the system matrix.

It becomes more and more popular in control theory to combine rather trivial systems by some glue code (written in an imperative programming language) in order to form more complex systems. Here, the behavior of the overall system depends also from the semantics of the used imperative programming language.

The traditional, math-based verification techniques usually fail to verify such composed systems, because they do not have a notion of programming constructs. KeYmaeraX is a theorem prover that covers both areas: continuously and abruptly evolving systems. Very recently, KeYmaeraX was extended by dedicated verification support for stability properties of dynamic systems. Thus, it is now possible to formally verify

[2] see `https://github.com/LS-Lab/KeYmaeraX-projects/blob/master/stability/switchedsystems.kyx`

the stability also of such systems, which are composed of subsystems and which switch between these subsystems. The switching decision can be any algorithm encoded by a simple imperative programming language.

From a practical point of view, the most urgent future work is to find an elegant encoding of the notion of asymptotic stability within the formalism supported by KeYmaeraX.

References

[1] P. Baudin, F. Bobot, D. Bühler, L. Correnson, F. Kirchner, N. Kosmatov, A. Maroneze, V. Perrelle, V. Prevosto, J. Signoles, and N. Williams, "The dogged pursuit of bug-free C programs: the Frama-C software analysis platform", *Communications of the ACM (CACM)*, vol. 64, no. 8, pp. 56–68, Aug. 2021.

[2] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming", *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[3] A. Platzer, *Logical Foundations of Cyber-Physical Systems*. Springer, 2018, ISBN: 978-3-319-63587-3.

[4] J.-D. Quesel, S. Mitsch, S. Loos, N. Aréchiga, and A. Platzer, "How to Model and Prove Hybrid Systems with KeYmaera: A Tutorial on Safety", *STTT*, vol. 18, no. 1, pp. 67–91, 2016. DOI: 10.1007/s10009-015-0367-0.

[5] T. Baar and S. Staroletov, "A Control Flow Graph Based Approach to Make the Verification of Cyber-Physical Systems Using KeYmaera Easier", *Modeling and Analysis of Information Systems. 2018;25(5)*, pp. 465–480, 2018.

[6] T. Baar, "A Metamodel-Based Approach for Adding Modularization to KeYmaera's Input Syntax", in *PSI, Novosibirsk*, Springer, 2019, pp. 125–139. DOI: 10.1007/978-3-030-37487-7_11.

[7] A. Liapounoff, "Problème général de la stabilité du mouvement", *Annales de la faculté des sciences de Toulouse*, vol. 9, no. 2, pp. 203–474, 1907, French translation of original work published in 1892. [Online]. Available: http://www.numdam.org/item?id=AFST_1907_2_9__203_0.

[8] A. M. Lyapunov, "The general problem of the stability of motion", *International Journal of Control*, vol. 55, no. 3, pp. 531–773, 1992, English translation of original work published in 1892.

[9] D. Liberzon, *Switching in Systems and Control*. Birkhäuser, 2003. DOI: 10.1007/978-1-4612-0017-8.

[10] R. E. Kalman, "On the general theory of control systems", in *Proc. 1st World Congress of the International Federation of Automatic Control*, 1960, pp. 481–493.

[11] V. I. Arnol'd, *Gewöhnliche Differentialgleichungen*. Springer-Verlag, 1979.

[12] J. L. Salle and S. Lefschetz, *Die Stabilitätstheorie von Ljapunov*. Mannheim: BI Hochschultaschenbücher, 1974.

[13] Y. K. Tan and A. Platzer, "Deductive Stability Proofs for Ordinary Differential Equations", in *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*, J. F. Groote and K. G. Larsen, Eds., ser. Lecture Notes in Computer Science, vol. 12652, Springer, 2021, pp. 181–199. DOI: 10.1007/978-3-030-72013-1_10.

[14] Y. K. Tan and A. Platzer, "Switched Systems as Hybrid Programs", in *7th IFAC Conference on Analysis and Design of Hybrid Systems, ADHS 2021, Brussels, Belgium, July 7-9, 2021*, R. M. Jungers, N. Ozay, and A. Abate, Eds., ser. IFAC-PapersOnLine, vol. 54, Elsevier, 2021, pp. 247–252. DOI: 10.1016/j.ifacol.2021.08.506.

[15] U. Topcu, A. K. Packard, and P. J. Seiler, "Local stability analysis using simulations and sum-of-squares programming", *Autom.*, vol. 44, no. 10, pp. 2669–2675, 2008. DOI: 10.1016/j.automatica.2008.03.010.

[16] M. Anghel, F. Milano, and A. Papachristodoulou, “Algorithmic Construction of Lyapunov Functions for Power System Stability Analysis”, *IEEE Trans. Circuits Syst. I Regul. Pap.*, vol. 60-I, no. 9, pp. 2533–2546, 2013. DOI: 10.1109/TCSI.2013.2246233.

[17] K. Forsman, “Construction of Lyapunov functions using Grobner bases”, in *Proceedings of the 30th IEEE Conference on Decision and Control*, 1991, pp. 798–799. DOI: 10.1109/CDC.1991.261424.

[18] J. Liu, N. Zhan, and H. Zhao, “Automatically Discovering Relaxed Lyapunov Functions for Polynomial Dynamical Systems”, *Math. Comput. Sci.*, vol. 6, no. 4, pp. 395–408, 2012. DOI: 10.1007/s11786-012-0133-6.

[19] S. Sankaranarayanan, H. B. Sipma, and Z. Manna, “Constructing invariants for hybrid systems”, in *International Workshop on Hybrid Systems: Computation and Control*, Springer, 2004, pp. 539–554.

[20] M. S. Branicky, “Multiple Lyapunov functions and other analysis tools for switched and hybrid systems”, *IEEE Trans. Autom. Control.*, vol. 43, no. 4, pp. 475–482, 1998. DOI: 10.1109/9.664150.

[21] Z. Sun and S. S. Ge, *Stability Theory of Switched Dynamical Systems*. Springer, Communications and Control Engineering, 2011. DOI: 10.1007/978-0-85729-256-8.

[22] A. Podelski and S. Wagner, “Model Checking of Hybrid Systems: From Reachability Towards Stability”, in *Hybrid Systems: Computation and Control, 9th International Workshop, HSCC 2006, Santa Barbara, CA, USA, March 29-31, 2006, Proceedings*, J. P. Hespanha and A. Tiwari, Eds., ser. Lecture Notes in Computer Science, vol. 3927, Springer, 2006, pp. 507–521. DOI: 10.1007/11730637_38.

MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 28, NO. 4, 2021
journal homepage: www.mais-journal.ru

THEORY OF COMPUTING

Autotuning Parallel Programs by Model Checking

N. O. Garanina[1,2], S. P. Gorlatch[3]

DOI: 10.18255/1818-1015-2021-4-338-355

[1]A.P. Ershov Institute of Informatics Systems (IIS), Siberian Branch of the Russian Academy of Sciences, 6 Acad. Lavrentjev pr., Novosibirsk 630090, Russia.
[2]Institute of Automation and Electrometry SB RAS, 1, Academician Koptyug ave., Novosibirsk 630090, Russia.
[3]University of Munster, 62 Einsteinstr, Muenster 48149, Germany.

MSC2020: 68W10
Research article
Full text in Russian

Received November 15, 2021
After revision December 1, 2021
Accepted December 8, 2021

The paper presents a new approach to autotuning data-parallel programs. Autotuning is a search for optimal program settings which maximize its performance. The novelty of the approach lies in the use of the model checking method to find the optimal tuning parameters by the method of counterexamples. In our work, we abstract from specific programs and specific processors by defining their representative abstract patterns. Our method of counterexamples implements the following four steps. At the first step, an execution model of an abstract program on an abstract processor is described in the language of a model checking tool. At the second step, in the language of the model checking tool, we formulate the optimality property that depends on the constructed model. At the third step, we find the optimal values of the tuning parameters by using a counterexample constructed during the verification of the optimality property. In the fourth step, we extract the information about the tuning parameters from the counter-example for the optimal parameters. We apply this approach to autotuning parallel programs written in OpenCL, a popular modern language that extends the C language for programming both standard multi-core processors (CPUs) and massively parallel graphics processing units (GPUs). As a verification tool, we use the SPIN verifier and its model representation language Promela, whose formal semantics is good for modelling the execution of parallel programs on processors with different architectures.

Keywords: optimization problem; auto-tuning of parallel programs; parallel programs; GPU programming; model checking; counterexamples; OpenCL; SPIN; Promela

INFORMATION ABOUT THE AUTHORS

Natalia Olegovna Garanina
correspondence author
orcid.org/0000-0001-9734-3808. E-mail: garanina@iis.nsk.su
Ph.D. in Mathematics, senior research fellow.

Sergei Petrovich Gorlatch
orcid.org/0000-0003-3857-9380. E-mail: gorlatch@uni-muenster.de
Prof. Dr. Habil.

Funding: This work was supported by the DAAD research scholarship of Dr. Natalia Garanina #91735805, by State assignment #AAAA-A19-119120290056-0, and by the DFG project PPP-DL at the University of Muenster, Germany.

For citation: N. O. Garanina and S. P. Gorlatch, "Autotuning Parallel Programs by Model Checking", *Modeling and analysis of information systems*, vol. 28, no. 4, pp. 338-355, 2021.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 28, № 4, 2021
сайт журнала: www.mais-journal.ru

THEORY OF COMPUTING

Подход к автонастройке параллельных программ методом проверки моделей

Н. О. Гаранина[1,2], С. П. Горлач[3]

DOI: 10.18255/1818-1015-2021-4-338-355

[1]Институт систем информатики имени А. П. Ершова СО РАН, пр. Лаврентьева, д. 6, г. Новосибирск, 630090 Россия.
[2]Институт автоматики и электрометрии СО РАН, пр. Акад. Коптюга, д. 1, г. Новосибирск, 630090 Россия.
[3]Университет Мюнстера, Эйнштейн-штрассе, д. 62, Мюнстер, 48149 Германия.

УДК 004.822, 681.51
Научная статья
Полный текст на русском языке

Получена 15 ноября 2021 г.
После доработки 1 декабря 2021 г.
Принята к публикации 8 декабря 2021 г.

В этой статье представлен новый подход к автонастройке программ, параллельных по данным. Автонастройка – это поиск оптимальных параметров настройки программы, при которых её производительность оказывается максимальной. Новизна подхода состоит в использовании метода проверки моделей для поиска оптимальных параметров настройки методом контрпримеров. В нашей работе мы абстрагируемся от конкретных программ и конкретных процессоров, задавая их представительные абстрактные шаблоны. Наш метод контрпримеров состоит в реализации следующих четырёх шагов. На первом шаге на языке инструмента проверки моделей задаётся модель исполнения абстрактной программы на абстрактном процессоре. На втором шаге на языке инструмента проверки моделей формулируем свойство оптимальности, зависящее от построенной модели. На третьем шаге подбираем оптимальные значения параметров настройки посредством использования контрпримеров, построенных в ходе верификации свойства оптимальности. На четвёртом шаге извлекаем информацию о параметрах настройки из контрпримера для оптимальных параметров. Мы применяем этот подход к автонастройке параллельных программ, написанных на языке OpenCL – современном популярном языке, который расширяет язык C для программирования как обычных многоядерных процессоров (CPU), так и массивно-параллельных графических процессоров (GPU). В качестве инструмента верификации мы используем верификатор SPIN и его язык представления моделей Promela, формальная семантика которого позволяет моделировать исполнение параллельных программ на процессорах с различной архитектурой.

Ключевые слова: задача оптимизации; автонастройка параллельных программ; параллельные программы; программирование GPU; проверка моделей; контрпримеры; OpenCL; SPIN; Promela

ИНФОРМАЦИЯ ОБ АВТОРАХ

Наталья Олеговна Гаранина
автор для корреспонденции
orcid.org/0000-0001-9734-3808. E-mail: garanina@iis.nsk.su
канд. физ.-мат. наук, с.н.с.

Сергей Петрович Горлач
orcid.org/0000-0003-3857-9380. E-mail: gorlatch@uni-muenster.de
профессор, канд. физ.-мат. наук.

Финансирование: Представленное исследование выполнено в рамках гранта DAAD № 91735805, государственного задания № АААА-А-19-119120290056-0, и проекта DFG #PPP-DL в Университете Мюнстера, Германия.

Для цитирования: N. O. Garanina and S. P. Gorlatch, “Autotuning Parallel Programs by Model Checking”, *Modeling and analysis of information systems*, vol. 28, no. 4, pp. 338-355, 2021.

Введение

Наша работа посвящена развитию методов автонастройки программ для современных архитектур параллельных процессоров. *Автонастройка* автоматически находит оптимальные значения так называемых *параметров настройки*, являющихся критичными для производительности исполнения программ на многоядерных процессорах (CPU) и многоядерных графических процессорах (GPU): оптимальные значения параметров настройки позволяют достичь максимальной производительности и/или минимального энергопотребления данной параллельной программы на конкретной архитектуре. Типичными примерами параметров настройки являются: количество параллельных потоков, количество потоков в одной рабочей группе, размер данных в локальной памяти процессора и т. п. Система автонастройки должна автоматически генерировать пространство поиска параметров и исследовать его, чтобы найти оптимальную (или достаточно хорошую) комбинацию значений параметров настройки. Подбор оптимальных параметров в системе автонастройки основан на выборе конфигурации параметров из пространства поиска и запуске программы с этими параметрами на реальном процессоре. Стадия перебора параметров и запуска является самой затратной по времени – как правило, поиск достаточно качественного решения занимает несколько часов, а иногда и сутки машинного времени.

В связи с высокой актуальностью проблемы автонастройки, к настоящему времени предложен и реализован ряд конкретных систем автонастройки: ATLAS, PATUS, FFTW, MILEPOST, CHiLL, OSKI, ActiveHarmony, Apollo, OpenTuner, CLTune, ATF [1—11]. Эти системы используют различные методы поиска оптимального решения, включая моделирование отжига, случайный поиск, статистические методы экспериментального проектирования, генетические алгоритмы, поиск с использованием машинного обучения и динамическое программирование. Но лишь некоторые из них предлагают исчерпывающий поиск, который находит гарантированно оптимальный результат. При этом, этот поиск использует прямолинейный алгоритм перебора всех возможных вариантов и поэтому, как правило, используется редко из-за высокой ресурсоёмкости и временных затрат.

В этой работе мы предлагаем новую технику автонастройки, основанную на исчерпывающем поиске с использованием инструментов проверки модели (model checking) и моделировании абстрактных процессоров. Обычно инструменты проверки модели исследуют каждый элемент пространства поиска, используя различные сокращения явного перебора этого пространства, такие как символьное кодирование, редукция частичных порядков и абстракция [12]. Эти сокращения позволяют исследовать бóльшие пространства поиска за более короткое время. Поэтому мы предполагаем, что инструменты проверки моделей будут находить гарантированно оптимальные значения параметров настройки за меньшее время, чем традиционные программы автонастройки. Кроме того, моделирование абстрактных процессоров позволяет не зависеть от наличия конкретных процессоров при поиске оптимальных параметров параллельной программы.

Проверка модели является формальным методом верификации программ. Этот метод проверяет выполнимость свойства параллельной системы для всех сценариев её поведения. Если проверяемое свойство не выполняется в некотором сценарии, то этот сценарий является *контрпримером*: последовательностью действий системы и условий, влекущих невыполнимость свойства. Контрпримеры являются основой подхода к решению задач оптимизации и, в частности, составления оптимальных расписаний для реагирующих систем [13—17]. В этой статье мы адаптируем технику контрпримеров для поиска оптимальных конфигураций параметров настройки.

Мы будем решать задачу автонастройки программ, написанных на популярном языке параллельных вычислений OpenCL [18], предложенного как новый стандарт для программирования как CPU, так и GPU. Основные компоненты нашего абстрактного процессора соответствуют схеме графического процессора, т.к. автонастройка наиболее часто используется именно для GPU-программ, поскольку архитектура GPU особенно сложна для предсказания конечной производительности

программ на этой архитектуре. В качестве инструмента проверки моделей мы будем использовать верификатор SPIN [19] – надежную и удобную программу проверки моделей. Его язык моделирования Promela имеет C-подобный синтаксис и темпоральную логику LTL для формулирования свойств программы. Семантика языка объединяет модели параллелизма CSP [20] и акторную модель [21], что идеально подходит для моделирования параллельного исполнения программ на различных типах графических процессоров. Также немаловажно, что в описании моделей на языке Promela допускаются вставки кода на языке C.

Оставшаяся часть работы организована следующим образом. Раздел 1 посвящён описанию техники контрпримеров в задачах планирования/оптимизации и нашей адаптации этой техники для задачи поиска оптимальных параметров автонастройки. В разделе 2 рассмотрены основные понятия языка OpenCL и нашей абстрактной модели графического процессора, необходимые для решения задачи автонастройки в контексте проверки моделей, а также идеи их моделирования на языке Promela. Раздел 3 содержит описание предлагаемой нами техники поиска оптимальных параметров автонастройки с использованием инструмента SPIN. В заключительном разделе 4 изложены выводы и представлен план будущих исследований.

1. Проверка моделей в задачах планирования, оптимизации и автонастройки

Проверка моделей – это формальный метод верификации модели системы относительно заданной спецификации её свойств. Этот метод проверяет свойства модели системы для всех сценариев её поведения. Модель системы и свойство задаются с помощью формальных языков. Например, модель системы можно задать как структуру Крипке, а свойство – формулой темпоральной логики LTL. Если свойство не выполняется в модели, то метод позволяет найти контрпример, т.е. условия и последовательность действий модели, влекущих невыполнимость свойства. Контрпримеры являются основой техники получения оптимальных расписаний для параллельных систем [13—17]. Кроме того, известный Дагштуль-семинар №14482 [22] был посвящён сочетанию методов автоматического планирования и проверки моделей.

Кратко изложим нашу идею использования проверки моделей в задаче планирования (оптимизации), которую мы предлагаем применять для автонастройки. *Задача планирования* состоит в поиске *плана*, т.е. поиска последовательности действий в некоторой среде, которая приводит к заданной цели. При этом *среду* мы представляем как дискретное множество состояний и переходов между ними. Мы считаем, что состояния среды дискретны, каждое состояние полностью наблюдаемо и множество состояний конечно. Определено *начальное состояние* и определена *цель* как множество заключительных состояний. Известны возможные действия по изменению состояний, и при этом каждое действие приводит к единственному новому состоянию, а время выполнения действий не учитывается. *Решением* задачи планирования будет являться путь от начального состояния до целевого состояния, который, вообще говоря, может удовлетворять некоторому набору ограничений. Если оценивать качество решения, то есть решать *задачу оптимизации*, то необходимо определить *функцию стоимости пути*. В зависимости от критерия оптимальности, это может быть наибольшая или наименьшая стоимость среди всех возможных решений задачи планирования. Таким образом, для алгоритма планирования необходимы методы представления среды и переходов между состояниями, а также метод управления перебором в пространстве состояний.

Очевидно, что метод проверки моделей полностью соответствует вышеизложенной постановке задач планирования и оптимизации. В этой постановке средой является пространство состояний модели системы, в которых выделяется множество начальных состояний. Состояния цели можно задать с помощью булевой формулы относительно переменных системы. Тогда, чтобы найти решение задачи планирования с помощью метода проверки моделей, нужно задать свойство *невозможности достижения цели*. В случае, если цель всё-таки достижима, то метод проверки моделей построит контрпример, демонстрирующий возможность достижения цели. Поскольку такой контр-

пример является последовательностью действий модели, приводящей к состоянию, в котором цель достижима, то он и будет планом, ведущим к цели. Если целевые состояния заданы булевой формулой φ, тогда свойство невозможности достижения цели на языке логики LTL выглядит как $\mathbf{G}\neg\varphi$. Если имеются ограничения на пути к цели φ, которые можно выразить булевой формулой ψ, то LTL-формула для свойства невозможности достижения цели при заданных ограничениях выглядит как $\neg(\psi\mathbf{U}\varphi)$.

В задаче оптимизации предполагается, что цель достижима и есть численная оценка пути достижения цели. Поэтому, для решения этой задачи с помощью проверки моделей, в модель системы, представляющую среду, нужно ввести оценочную переменную, значения которой в состоянии модели зависят от пути, ведущего в это состояние. Тогда для получения оптимальных значений этой переменной, нужно задать свойство *невозможности достижения цели при заданной оценке*. Пусть целевые состояния заданы булевой формулой φ и, например, (минимальная) оценка задана неравенством $x < N$, где x – оценочная переменная, а N – число. Тогда это свойство выражается следующей формулой логики LTL: $\mathbf{G}(\varphi \longrightarrow x \geq N)$. Если это свойство не выполняется в модели, это означает, что цель достижима при значениях оценки x, меньших заданного порога N. Значит, если уменьшить порог, то можно получить лучший план, который приводит к цели. Таким образом, для получения оптимального значения оценочной функции необходимо последовательно выполнять проверку моделей с изменяющимся значением порога до тех пор, пока метод не перестанет конструировать контрпримеры. Порог, для которого минимальное его изменение ведёт к отсутствию контрпримера, будет оптимальным значением численной оценки, а контрпример, который порождается для этого порога, является оптимальным планом. Для автоматического составления оптимальных расписаний оценочной переменной может быть переменная, хранящая глобальное время работы системы. В этом случае, как правило, необходимо найти минимальное время работы системы. Вышеописанную технику можно обобщить для численных оценок нескольких параметров.

В этой статье мы адаптируем технику контрпримеров проверки модели для задачи поиска оптимальных параметров автонастройки исполнения параллельных программ на языке OpenCL на абстрактном графическом процессоре следующим образом.

В задаче параллельных вычислений целью является вычисление результата для заданных входных данных. Эта цель может быть достигнута различными путями в зависимости от распределения параллельной обработки данных по вычислительным элементам и использования локальной памяти процессора. Задача оптимизации состоит в поиске последовательности действий, которая за кратчайшее время позволяет получить результат вычисления. Поэтому свойство Φ_o невозможности достижения цели при заданной оптимальной оценке формулируется как: “Параллельная программа *не может* завершиться через T единиц времени”. Свойство Φ_o нужно верифицировать в модели, которая представляет исполнение параллельной программы вычислений в заданном процессоре, с помощью инструмента проверки модели. Если это свойство не выполняется, это означает, что вычисления действительно завершаются в течение времени T. Инструмент проверки модели конструирует контрпример, который описывает условия завершения работы программы за время T, включая конфигурацию параметров настройки, которые отвечают за распределение параллельной обработки данных по вычислительным элементам и интенсивность использования локальной памяти. Далее нужно уменьшить время завершения и проверять это свойство снова и снова, пока не найдётся *минимальное* время выполнения программы MT: если мы уменьшим это время на одну единицу времени, программа проверки модели докажет, свойство Φ_o выполняется, т.е. программа действительно не может завершиться за время T. Начальное значение времени завершения может быть задано с помощью симуляции модели программы. Как правило, инструменты проверки моделей позволяют симулировать сценарии выполнения модели и можно выявить значение вре-

мени исполнения параллельной программы в каком-либо из сценариев. Уменьшать полученное значение можно, например, методом бисекции.

Итак, предлагаемый нами *метод контрпримеров для поиска оптимальной конфигурации параметров настройки* состоит из следующих шагов.

1. Представление на языке инструмента проверки моделей параметров настройки и исполнения программы на выбранном процессоре.
2. Формулирование свойства Φ_o невозможности достижения цели на языке программы проверки моделей.
3. Поиск минимального времени завершения программы MT.
4. Извлечение информации об оптимальной конфигурации параметров из контрпримера для минимального времени MT.

В следующих разделах мы решаем задачу поиска параметров настройки для программ OpenCL, исполняющихся на различных графических процессорах. Для решения этой задачи методом контрпримеров мы выбрали в качестве средства проверки моделей инструмент SPIN [19], поскольку язык Promela, как и OpenCL, тоже близок к языку C, и при этом он допускает встраивание кода C в проверяемые модели. Кроме того, формальная семантика Promela, основанная на исчислении взаимодействующих процессов CSP [20] и алгебре акторов [21], допускает формализацию абстрактного графического процессора. Вышеперечисленные 4 шага поиска параметров настройки OpenCL с помощью SPIN мы подробно опишем в разделе 3.

В следующем разделе мы описываем наш подход к абстрагированию понятий языка OpenCL [18] и графического процессора, необходимые для построения Promela-модели исполнения абстрактной программы OpenCL на абстрактном процессоре.

2. Абстрактный графический процессор и язык OpenCL

Язык *OpenCL* используется для реализации параллельных вычислений на т.н. *OpenCL-устройствах*, которыми могут быть как многоядерные процессоры (CPU), так и графические процессоры (GPU). В нашей работе мы рассматриваем только графические процессоры (далее – *процессоры*), поскольку OpenCL разрабатывался в первую очередь для таких устройств, и как правило, реализация программ OpenCL на них более эффективна. Однако наш подход достаточно общий и может быть адаптирован для CPU.

2.1. Абстрактный графический процессор

В этой работе мы рассматриваем элементы архитектуры нашего абстрактного графического процессора `Abstract Processor`, ориентируясь на архитектуру Ферми корпорации Nvidia, которая является ведущим производителем GPU-процессоров [23]. Другие графические процессоры устроены похожим образом.

Процессор связан с CPU, откуда загружаются данные в *глобальную память* процессора GPU. Наш `Abstract Processor` включает m *мультипроцессоров SM* (Streaming Multiprocessor). Каждый мультипроцессор содержит ряд процессорных элементов (обычно их число составляет 2^n). В частности, мультипроцессор с архитектурой Fermi включает 32 универсальных *вычислителя* (cores), 16 элементов для работы с данными *LSU* (load/store units) и 4 элемента для работы со специальными функциями *SFU* (Special Function Units). В дальнейшем мы предполагаем, что абстрактный мультипроцессор содержит 2^n процессорных элемента, которые, для простоты моделирования, являются только универсальными вычислителями.

Процессорные элементы работают одновременно. Их вычислениями управляют встроенные *диспетчеры*. Они запускают вычислители группами – так называемыми *варпами* (warp). Будем считать,

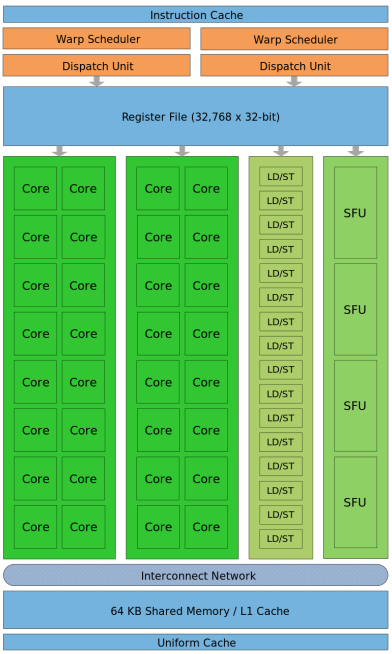


Fig. 1. Fermi architecture of multiprocessor

Рис. 1. Мультипроцессор архитектуры Ферми

что в абстрактный мультипроцессор встроены 2^k диспетчеров и размер варпа составляет 2^{n-k}[1]. Диспетчеры для работы с варпами используют список инструкций обработки данных. Процессорные элементы выбранного диспетчером варпа одновременно выполняют свою очередную инструкцию. Когда все вычислители варпа завершают свою работу, диспетчер выбирает следующий варп.

Мультипроцессор имеет быструю *локальную память*, к которой есть доступ у всех процессорных элементов. Туда могут загружаться данные из глобальной памяти процессора. Отношение скоростей доступа к локальной памяти и глобальной составляет примерно 1/100.

На рисунке 1 представлен мультипроцессор архитектуры Ферми. Здесь оранжевым цветом обозначены компоненты, отвечающие за диспетчеризацию, оттенками зелёного цвета – процессорные элементы, а синим – модули локальной памяти.

Итак, наш `Abstract Processor` имеет следующие компоненты: процессорные элементы, диспетчеры и мультипроцессоры, а также CPU-процесс, который распределяет вычисления по мультипроцессорам. Мы будем представлять их с помощью отдельных Promela-процессов. Все эти процессы иерархически связаны отношениями подчинения: CPU-процесс запускает мультипроцессоры, мультипроцессоры активируют своих диспетчеров, а диспетчеры, в свою очередь, регулируют работу своего варпа, т.е. набора процессорных элементов. Для синхронизации своей работы они используют каналы сообщений ёмкостью 1, по которым пересылаются команды начала и окончания работы, а также сообщения о завершении вычислений. Различие в скоростях доступа к локальной и глобальной памяти моделируется с помощью констант, задающих объём времени обращения к памяти. Для моделирования глобального времени в системе мы используем специальный процесс, использующий число активных процессорных элементов для определения момента, когда счётчик времени увеличивается. В этой работе мы не учитываем время коммуникации между компонентами процессора, которое на практике существенно меньше, чем другие временные задержки. Мы изложим детали нашей реализации на Promela абстрактного процессора в разделе 3.

[1]Мультипроцессор с архитектурой Fermi включает два диспетчера и в одном его варпе может быть 16 вычислителей или LSU, или 4 SFU.

2.2. Ключевые понятия программы на языке OpenCL

Программа на OpenCL состоит из двух логических частей: *хост-программа* и *ядро*. Хост-программа исполняется на CPU, соединенном с графическим процессором (GPU), а ядро – параллельно всеми процессорными элементами этого GPU. Хост-программа обеспечивает исполнение ядра: компилирует ядро, работает с данными для него: резервирует глобальную память процессора, копирует в неё данные, и выгружает данные результата из этой памяти в память CPU. Ядро производит обработку данных и вычисления.

Типичная программа на OpenCL обрабатывает массивы и выдаёт результат обработки. Поэтому, как правило, множество ядер параллельно вычисляют выражения на основе доступных данных, зависящих от индекса массива, и присваивают результат вычисления отдельной переменной или элементу массива. Отметим, что массив может быть многомерным, и тогда его элементы будут иметь соответствующий многомерный индекс. Каждое вычисление (ядро) выполняет один *рабочий элемент*. Рабочий элемент имеет идентификатор, соответствующий индексу массива. Таким образом, рабочие элементы выполняют одни и те же вычисления, но для разных данных, т.е. элементов массива. Этот метод распараллеливания известен в литературе как параллелизм по данным (data parallelism). Если вычисления одной итерации цикла зависят от вычислений другой итерации, то есть от индекса массива, то в OpenCL предусмотрены средства синхронизации вычислений: *барьеры* (barrier). Множество рабочих элементов разбивается на *рабочие группы* либо явно программистом, либо по умолчанию конкретной реализацией OpenCL. Элементы одной группы должны исполняться на одной вычислительной единице (например, мультипроцессоре).

В OpenCL память разделяется на 4 вида: *глобальная, постоянная, локальная и приватная*. К глобальной памяти имеют доступ все рабочие элементы и хост-программа. Постоянная память – это неизменяемая часть глобальной памяти. К локальной памяти имеют доступ элементы одной рабочей группы. Приватной памятью может пользоваться один рабочий элемент. Поскольку предполагается, что доступ к локальной памяти на устройстве, для которого компилируется программа, существенно быстрее доступа к глобальной памяти, то ядро часто задаёт, какие данные и в каком объёме будут загружаться в локальную память. Эти данные называются *плитками* (tiles).

Язык C служит основой языка OpenCL. Основные ограничения OpenCL по сравнению с C:

- отсутствуют многие стандартные функции, например, `printf` или `malloc`;
- нет массивов переменной длины;
- рекурсия не допускается;
- нет указателей на функции.

Схема абстрактной программы на OpenCL выглядит следующим образом:

```
int main ( void ) {
 1. Initialize OpenCL
 2. Compile kernel source code
 3. Reserve global memory on the device and copy data to the device
 4. Execute kernel (instances)
 5. Copy data from the device
}
```

Листинг 1: Абстрактная программа на OpenCL

В дальнейшем мы будем рассматривать следующую схему типичного ядра программы на OpenCL:

```
__kernel void abstract_kernel (__global float* N_g, __global float* R_g,
    int size){
1. __local float N_l[ TS ];
2. int idx_g = get_global_id (0);
3. int idx_l = get_local_id (0);
4. float result = 0;
5. for (int i = 0; i < size / TS; ++i) {
```

```
    // access to global memory
6.      N_l[idx_l] = f(i, idx_l, size, tile, N_g);
    // waiting for local co-workers
7.      barrier (CLK_LOCAL_MEM_FENCE);
8.      if b(idx_l) // access to local memory
9.           for (int k=0; k < TS; ++k) result = g1(k, idx_l, N_l, result);
10.     else for (int k=0; k < TS; ++k) result = g2(k, idx_l, N_l, result);
    // waiting for local co-workers
11.     barrier (CLK_LOCAL_MEM_FENCE);}
 // copy the result of this working item to global memory
12. R_g[h(idx_g, size)] = result;
```

Листинг 2: Схема типичного ядра OpenCL

Параллельные по данным вычисления – это всегда вычисления для массивов (возможно, многомерных). Поэтому для каждого индекса массива отдельный рабочий элемент вычисляет локальный результат исполнения кода ядра. В нашем примере программы в листинге 2 входными данными является массив `N_g` размера `size`. Выходными данными (результатом) здесь являются элементы массива `R_g`. Для уменьшения обращений к глобальной памяти в строке 1 объявляется локальный массив `N_l` размера `TS`, элементы которого зависят от входных данных. Каждый из элементов рабочей группы, который получил значение своего индекса `idx_l` в строке 2, с помощью функции `f` параллельно вычисляет (или копирует) своё значение локального массива `N_l[idx_l]` в строке 6, но пользоваться этим значением могут все элементы его группы, поэтому в строке 7 необходимо дождаться завершения работы с глобальными данными всех элементов данной рабочей группы с помощью оператора синхронизации `barrier`. Далее в строках 9-10 итеративные вычисления результата `result` зависят только от локальных данных и индекса элемента группы. Кроме того, в зависимости от индекса, эти вычисления могут проводиться по-разному: в строке 8 булева функция `b(idx_l)` регулирует варианты вычисления (функция `g1` или `g2`). В строке 11 рабочий элемент дожидается завершения вычислений всех одногруппников с текущими локальными данными, и на следующей итерации цикла строки 5, локальные данные снова вырабатываются на основе очередной порции глобальных данных. Когда глобальные данные будут обработаны полностью, результат работы элемента группы сохраняется в глобальной памяти в элементе массива `R_g`, индекс которого зависит от размера входных данных и глобального индекса `idx_g`, полученного элементом группы `idx_l` в строке 2. Список *Возможных деталей* программ на OpenCL, которые не показаны в примере схемы ядра `abstract_kernel` включает следующие пункты:

1) массивы могут быть многомерными;
2) может быть несколько входных массивов, и кроме массивов могут быть другие входные переменные и константы, размещённые в глобальной памяти;
3) выходными данными может быть несколько массивов или чисел;
4) нет изменений в глобальной памяти, которые требуют синхронизации всех процессов всех рабочих групп.

Однако пример листинга 2 легко дополнить этими деталями, не выходя за рамки его функций и операторов.

Наш подход к использованию языка Promela использует тот факт, что поскольку язык Promela, как и язык OpenCL, близок к языку C, то программы на OpenCL можно транслировать в язык Promela с учётом следующих ограничений. Так как SPIN предполагает абстрагирование от вычислительных аспектов программ и предназначен для проверки взаимодействия, синхронизации и координации параллельных процессов, все данные Promela-модели должны иметь конечный тип. Более того, при задании на Promela абстрактного ядра, мы абстрагируемся от конкретных вычислений, и для решения нашей задачи поиска оптимальных параметров мы учитываем лишь время этих вычислений, которое зависит от количества и соотношения обращений к глобальной и локальной памяти.

Поэтому строки вычислений 6, 9, 10 и 12 листинга 2 в Promela-модели абстрактного ядра заменяются на код, реализующий течение времени, необходимого для этих вычислений. По этой же причине мы игнорируем параметры и локальные переменные ядра, задающие содержание данных для вычислений (строки 1-4). Однако количество этих вычислений, т. е. `size` должно быть учтено. В силу абстрагирования от вычислений в Promela-модели ядра, пункты 1-3 списка Возможных деталей также не учитываются при моделировании, а значит, пример ядра `abstract_kernel` является достаточно представительным с точностью до количества циклов и структур управления программы, так как для моделирования вычислений имеет значение только размер входных данных.

Кроме того, важным аспектом, который мы учитываем при моделировании, является синхронизация элементов рабочих групп относительно изменений в локальной и глобальной памяти. Для обеспечения локальной синхронизации в модель вводятся процессы барьера, отвечающие за отдельную рабочую группу, а для глобальной синхронизации – процесс барьера для всех рабочих элементов. В нашем примере ядра глобальный барьер отсутствует, но его реализация на Promela аналогична реализации локального барьера.

Роль хост-программы в нашем моделировании на Promela сводится к процессу “компиляции” ядра, т. е. распределению вычислений в заданном графическом процессоре. Детали реализации на Promela исполнения абстрактного ядра `abstract_kernel` и его хост-программы на абстрактном процессоре `Abstract Processor` изложены в следующем разделе, где описывается применение метода проверки моделей для решения задачи оптимизации производительности программ OpenCL.

3. Поиск оптимальных параметров программ OpenCL с помощью SPIN

Для реализации первого шага поиска оптимальной конфигурации параметров настройки (раздел 1), т. е. моделирования на языке инструмента проверки моделей исполнения программы на выбранном процессоре, будем опираться на следующую общепринятую семантику исполнения программ OpenCL [18].

Согласно этой семантике, компилятор выделяет для исполнения хост-программы один хост-процессор (CPU), соединенный с графическим процессором (GPU). Последний объединяет m мультипроцессоров SM, а элементы одной рабочей группы исполняются на одном мультипроцессоре. Каждый рабочий элемент группы последовательно исполняется на одном вычислителе мультипроцессора. Следовательно, одновременно могут исполняться 2^n рабочих элемента, по 2^{n-k} на каждый варп, где 2^k – число диспетчеров. Если размер рабочей группы больше 2^n, то множество рабочих элементов разбивается на несколько варпов и два диспетчера мультипроцессора выбирают поочерёдно по одному варпу для исполнения очередной инструкции рабочих элементов группы. За один такт выполняется ровно одна инструкция элемента. Поскольку в коде ядра могут встречаться условные операторы, зависящие от номера рабочего элемента, то в этом случае варп может выполниться только частично, а инструкции оставшихся элементов варпа выполняются в следующих тактах, пока все элементы варпа не исполнят свою очередную инструкцию. После этого диспетчер выбирает следующий варп. Рабочие элементы могут использовать данные как из локальной памяти мультипроцессора, так и из глобальной памяти. При этом данные локальной памяти не доступны элементам других рабочих групп, исполняемых на других мультипроцессорах. Поскольку доступ к локальной памяти значительно быстрее, чем к глобальной, разумно ожидать, что разбиение на группы максимизирует использование локальной памяти, то есть в одной группе оказываются элементы, которые в основном используют данные, вырабатываемые внутри группы.

Таким образом, мы выделяем следующие параметры, влияющие на производительность параллельных вычислений для программы на OpenCL. Эти параметры зависят от размера входных данных `size`, обрабатываемых программой. Количество рабочих элементов зависит от размера данных.

- *Размер рабочей группы* WG (определяется в хост-программе). При оптимально выбранном размере рабочей группы все вычислители всех мультипроцессоров GPU загружены полностью и равномерно, что приводит к сокращению общего времени вычислений.
- *Размер плиток* TS (определяется в ядре). При оптимальном выборе размера данных, периодически загружаемых в быструю локальную память, достигается минимизация числа обращений к медленной глобальной памяти, что также приводит к сокращению времени вычислений.

Итак, мы решаем задачу поиска оптимальных размеров рабочих групп WG и размеров плиток TS для абстрактного ядра abstract_kernel (листинг 2) и его хост-программы, исполняемых на абстрактном процессоре Abstract Processor (раздел 2.1).

Шаг 1 метода контрпримеров

Этот шаг является самым трудоёмким этапом нашего метода, так как необходимо учесть все детали исполнения параллельных программ OpenCL на абстрактном графическом процессоре. На этом шаге мы определяем в Promela-модели *PM* этого исполнения следующие Promela-процессы.

Процесс рабочего элемента группы pex выполняет вычисления экземпляра ядра abstract_kernel. Общее число этих процессов, создающихся в Promela-модели, зависит от размера входных данных. Число одновременно работающих процессов pex зависит от количества мультипроцессоров и величины варпов, что значительно меньше общего числа процессов. Каждый из них запускается своим диспетчером, Promela-процесс которого описан ниже. Процесс pex связан каналами синхронизации pex_b и b_pex с локальным барьером группы, и pex_d и d_pex – со своим диспетчером. В эти каналы рабочий элемент получает команды запуска и останова, а также сообщает о завершении (этапа) вычислений. Мы абстрагируемся от конкретных вычислений, которые проводит ядро, поэтому в модели используем только время выполнения вычислений. При этом время вычислений, использующих локальную память, мы считаем равным одной условной единице времени, а время вычислений, использующих глобальную память, равным GMT условным единицам времени. Таким образом, в процессе pex смоделировано только количество шагов вычислений, совершаемых ядром (цикл for в строке 5), зависящее от размера входных данных и обращений к глобальной памяти (строки 6-11, 23-28) и к локальной памяти (строки 14-19). По завершении шага вычислений pex сообщает об этом событии процессу, реализующему глобальное время посредством увеличения счётчика работающих в данный момент процессов NRP_work. Отметим, что в силу блокирующей семантики языка Promela, процесс может перейти к следующему этапу своих вычислений только, когда глобальное время time увеличится на 1 (строки 10, 16, 19 и 27). Синхронизация по локальному барьеру происходит дважды, как и в исходном ядре: строка 7 ядра соответствует строке 13 модели, а строка 11 – строке 21. Процесс pex завершает работу (строка 29), когда выгружает результат своей работы в глобальную память.

```
proctype pex ( byte me; chan pex_b; chan b_pex; chan d_pex; chan pex_d) {
1.  do
2.  :: d_pex ? go, me ->
3.     start_time = time;
4.     cur_time = time;
5.     for ( i : 0 .. size/TS){
6.          do  // access to global memory
7.          :: time > start_time + GMT -> break;
8.          :: else -> atomic { cur_time = time;
9.                              NRP_work++;}
10.                    time == cur_time + 1;
11.         od;
12.         b_pex ! done;
          // waiting for local co-workers
13.         pex_b ? go, me;
          // 'if' access to local memory
14.         atomic { cur_time = time;
```

```
15.                   NRP_work++;}
16.          time == cur_time + 1;
         // 'else' access to local memory
17.          atomic { cur_time = time;
18.                   NRP_work++;}
19.          time == cur_time + 1;
20.          b_pex ! done;
         // waiting for local co-workers
21.          pex_b ? go, me;
       }
22.    start_time = time;
   // copy the result of this working item to global memory
23.    do
24.    :: time > start_time + GMT -> break;
25.    :: else -> atomic { cur_time = time;
26.                        NRP_work++;}
27.                 time == cur_time + 1;
28.    od;
29.    pex_d ! done;
30. :: d_pex ? stop, me -> break;
31. od;
}
```

Листинг 3: Promela-процесс для рабочего элемента группы pex

Процесс локальной синхронизации `barriere` синхронизирует процессорные элементы одного мультипроцессора, с которыми он связан каналами `pex_b` и `b_pex`. Он принимает от процессов `pex` сообщение о приостановке их работы и подсчитывает в переменной `i` число процессов, которые ожидают завершения работы с локальной памятью других элементов группы, выполняющих в данный момент вычисления. Когда оно окажется равным числу процессорных элементов `PE`, барьер считается пройденным и в строке 13 процесс барьера позволяет процессорным элементам продолжить вычисления.

```
proctype barriere (chan pex_b; chan b_pex) {
1. do
2. :: pex_b ? done ->
3.             i = 1;
4.             do
5.             :: i < PE -> atomic {
6.                             pex_b ? done;
7.                             NRP--;
8.                             i++;}
9.             :: else -> break;
10.            od;
11.            atomic {
12.              NRP = NRP + PE;
13.              for (i : 1..PE) { b_pex ! go, i;} }
14. :: pex_b ? stop -> break;
15. od;
}
```

Листинг 4: Promela-процесс для локальной синхронизации `barrier`

Процесс диспетчера `dispatcher` запускает в строке 5 один варп, т.е. `WR` процессов `pex`, исполняющих ровно одну операцию одновременно. При этом он обновляет количество `NRP` запущенных в данный момент процессоров (строка 4). На это число влияет число процессов диспетчеров `ND` в одном мультипроцессоре и число `NM` мультипроцессоров, которые зависят от выбранного процессора. В зависимости от размера рабочей группы `WG`, может быть несколько варпов, приписанных одному диспетчеру. В таком случае, диспетчер активирует их поочерёдно, пока все элементы рабочей группы не выполнят все свои вычисления (строки 6-16). Мы считаем, что диспетчер запускает варпы целое число раз (`WG % (WR*ND) = 0` в строке 6). Процесс `dispatcher` завершает работу, когда

все приписанные к нему процессы pex завершаются и сообщает об этом своему мультипроцессору (строка 19).

```
proctype dispatcher (byte me; chan pex_b; chan b_pex; chan mul_d; chan
    d_mul) {
 chan d_pex = [1] of {mtype : action, byte};
 chan pex_d = [1] of {mtype : action};
1. do
2. :: mul_d ? go, me ->
3.              atomic {
4.                NRP = NRP + WR;
5.                for (i : 1..WR){ run pex (i, pex_b, b_pex, d_pex, pex_d);}}
6.              for (j : 1..WG/(WR*ND)) {
7.                   atomic {
8.                     NRP = NRP + WR;
9.                     for (i : 1..WR) { d_pex ! go, i;} }
10.                  i = 0;
11.                  do
12.                  :: i < WR -> atomic {
13.                                  pex_d ? done;
14.                                  NRP--;
15.                                  i++; }
16.                  :: else -> break;
17.                  od; }
18.           for (i : 1..WR) { d_pex ! stop, i;}
19.           d_mul ! done;
20. :: mul_d ? stop, me -> break;
21. od;
}
```

Листинг 5: Promela-процесс для диспетчера

Процесс мультипроцессора `muproc` запускает свои процессы `dispatcher` числом `ND` в строках 4-5 и фиксирует завершение их работы в строках 8-13, после чего сообщает процессу `host` о завершении вычислений в своей рабочей группе. Отметим, что мультипроцессор запускает единый для своей рабочей группы процесс локальной синхронизации `barriere` с каналами, в который есть доступ у всех процессорных элементов этой группы, поскольку они передаются как параметр.

```
proctype muproc (byte me; chan m_hst; chan hst_m) {
 chan pex_b = [1] of {mtype : action};
 chan b_pex = [1] of {mtype : action, byte};
 chan d_mul = [1] of {mtype : action};
 chan mul_d = [1] of {mtype : action, byte};
1.  do
2.  :: hst_m ? go, me ->
3.             run barriere (px_bar, bar_px);
4.             atomic { for (i : 1..ND) {
5.                        run dispatcher (i, pex_b, b_pex, mul_d, d_mul); }}
6.             for (i : 1..ND) { mul_d ! go, i;}
7.             i = 0;
8.             do
9.             :: i < WR -> atomic {
10.                           d_mul ? done;
11.                           i++;}
12.            :: else -> break;
13.            od;
14.            for (i : 1..ND) { mul_d ! stop, i;}
15.            px_bar ! stop;
16.            m_hst ! done, me;
17. :: hst_m ? stop, me -> break;
18. od;
}
```

Листинг 6: Promela-процесс для мультипроцессора

Процесс хост-программы `host` запускает мультипроцессоры `muproc`. Если число рабочих групп `WGs` оказалось больше числа мультипроцессоров в выбранном процессоре, то вычисления, разбитые на рабочие группы, запускаются последовательно (строки 13-31). Здесь мы не требуем, чтобы число рабочих групп делилось нацело на число мультипроцессоров `NM`. Процесс `host` фиксирует завершение работы мультипроцессоров присваиванием глобальной переменной `DONE` значение `true`. Это означает, что параллельные вычисления завершились.

```
proctype host () {
 chan m_hst = [1] of {mtype : action, byte};
 chan hst_m = [1] of {mtype : action, byte};
1.  DONE = false;
2.  if
3.  :: WGs <= NM ->
4.                atomic { for (i : 1..WGs) {run muproc (i, m_hst, hst_m);}}
5.                for (i : 1..WGs) {hst_m ! go, i};
6.                i = 0;
7.                do
8.                :: i < WGs -> atomic {
9.                                m_hst ? done;
10.                               i++; }
11.               :: else -> break;
12.               od;
13. :: else -> atomic { for (i : 1..NM) {run muproc (i, m_hst, hst_m);}}
14.             for (i : 1..NM) {hst_m ! go, i};
15.             i = 0;
16.             do
17.             :: i < WGs -> atomic {
18.                             m_hst ? done, j;
19.                             hst_m ! stop, j;
20.                             run muproc (j, m_hst, hst_m);
21.                             hst_m ! go, j;
22.                             i++; }
23.             :: else -> break;
24.             od;
25.             i = 0;
26.             do
27.             :: i < NM -> atomic {
28.                            m_hst ? done, j;
29.                            i++;}
30.             :: else -> break;
31.             od;
33. fi;
33. DONE = true;
}
```

Листинг 7: Promela-процесс для хост-программы

Основной процесс `main` выбирает значения параметров настройки `WG` и `TS` и запускает процессы `host` и `clock`. Количество мультипроцессоров `NM`, количество диспетчеров `ND` для одного мультипроцессора, размер варпа `WR` и количество вычислителей `PE` являются глобальными константами и объявляются в начале описания модели. Это даёт возможность настройки Promela-модели для различных архитектур реальных графических процессоров. В этой модели для простоты мы считаем, что размер данных `size` является степенью числа 2. Поэтому числа `WG` и `TS` также будут какой-либо случайной степенью 2. Эти степени выбираются в строках 3 и 6. Поскольку Promela не поддерживает операцию возведения в степень, выбранные случайным образом числа можно получить посредством соответствующего побитового сдвига `size`.

```
active proctype main() {
byte d;
 // let size = 2^n
1.  byte n = 10;
```

```
2.   size = 1024;
 // WG selection
3.   select ( d  : 1 .. n-1 );
4.   WG = size >> (n - d);
 // Number of Working Groups
5.   WGs = size/WG;
 // tile size selection
6.   select ( d  : 1 .. n/2 );
7.   TS = size >> (n - d);
8.   Topt = 100;
9.   atomic { run host(); run clock(); }
}
```

Листинг 8: Promela-процесс для выбора параметров настройки и запуска вычислений

Процесс часов `clock` реализует подсчёт глобального времени. Этот процесс увеличивает глобальную переменную счётчика времени `time`, когда все запущенные в настоящий момент процессы `pex` (их число `NRP`) сообщили посредством увеличения разделяемой переменной `NRP_work`, что они находятся в состоянии вычисления очередного значения. Процесс `clock` останавливается, когда глобальная переменная `DONE` принимает значение `true`. Значение `time` в этот момент является временем, затраченным на вычисления.

```
proctype clock () {
1. do
2. :: DONE -> break;
3. :: NRP != 0 && NRP_work == NRP -> atomic { NRP_work = 0; time++; }
4. od;
}
```

Листинг 9: Promela-процесс для подсчёта глобального времени

Таким образом, мы задали модель исполнения программы OpenCL на абстрактном графическом процессоре, т.е. выполнили первый шаг метода контрпримеров для поиска оптимальных параметров настройки (раздел 1).

Шаг 2 метода контрпримеров

На втором шаге метода при формулировании свойства невозможности достижения цели Φ_o^a мы будем использовать значение переменной `DONE`, фиксирующей окончание вычислений, и финальное значение `time`. Языком спецификации свойств в SPIN является темпоральная логика LTL и поэтому $\Phi_o^a = \mathbf{G}(\texttt{DONE} \longrightarrow (\texttt{time} \geq MT))$, что соответствует высказыванию "Всегда при завершении параллельных вычислений время работы программы больше минимального времени MT."

Шаг 3 метода контрпримеров

Третий шаг метода, который заключается в поиске минимального времени, за которое программа параллельных вычислений может завершиться, начинается с запуска верификатора SPIN с построенной моделью PM и формулой Φ_o^a для некоторого значения минимального времени MT. Затем мы уменьшаем MT до тех пор, пока SPIN не перестанет генерировать контрпримеры, т.е. пока не согласится с тем, что программу за время, меньшее заданного MT, выполнить нельзя. Начальное значение MT можно задать, используя возможность симуляции моделей в SPIN. При симуляции SPIN воспроизводит один из конечных сценариев работы системы, фиксируя значения используемых в модели переменных по завершении симуляции. Поэтому можно воспользоваться значением `time`, соответствующим концу работы программы в симулируемом сценарии. Для уменьшения значения MT в следующих запусках SPIN мы используем метод бисекции.

Шаг 4 метода контрпримеров

Последний шаг нашего подхода – анализ контрпримера для извлечения оптимальной конфигурации параметров настройки. Для анализа контрпримера SPIN предоставляет возможность

запуска симуляции, соответствующей переходам контрпримера. В задаче автонастройки не нужно осуществлять поиск оптимального пути вычислений, поэтому собственно переходы контрпримера анализировать нет необходимости. Для решения нашей задачи нужно извлечь лишь значения параметров настройки `WG` и `TS`, которые, как и значения других переменных, известны в конце симуляции.

Таким образом, мы показали, как можно использовать инструмент проверки моделей SPIN для решения задачи поиска оптимальных параметров программ на языке OpenCL, исполняемых на абстрактном графическом процессоре. Абстрактный графический процессор можно специализировать путём задания конкретных значений числа мультипроцессоров, диспетчеров и процессорных элементов, а также, возможно, изменением алгоритма взаимодействия диспетчера с процессорными элементами.

4. Заключение

Результаты нашей работы направлены на развитие методов автонастройки за счет разработки новой техники, исчерпывающего поиска оптимальных параметров настройки параллельных программ на основе проверки моделей. В представленной статье мы предложили подход к поиску оптимальных параметров исполнения программ на языке OpenCL на абстрактном графическом процессоре, который обобщает архитектуры GPU, используемые на практике. Наш подход использует метод контрпримеров, основанный на проверке моделей. Метод предполагает представление исполнения программ и свойства оптимальности на языке инструмента проверки моделей. В качестве такого инструмента мы выбрали верификатор SPIN с языком Promela для моделирования исполнения программ и темпоральной логикой LTL для представления свойства оптимальности.

Наше моделирование исполнения программ OpenCL в Promela абстрагируется от конкретных вычислений, сохраняя логику взаимодействия и синхронизации параллельных процессов программы в выбранном графическом процессоре. Отметим, что поскольку язык Promela имеет формальную семантику, то Promela-модель заданной OpenCL-программы можно считать формальной операционной семантикой взаимодействия и синхронизации параллельных процессов программы в выбранном графическом процессоре. Более того, варьирование параметров и алгоритмов взаимодействия компонентов абстрактного процессора позволяет задавать и исследовать конкретные графические процессоры. Это даёт возможность поиска оптимальных параметров настройки программ при физическом отсутствии реальных графических процессоров, в то время как традиционные системы автонастройки такой возможности предоставить не могут.

В будущем мы планируем добавить в модель время коммуникации между процессами, а также рассмотреть другие параметры настройки, в частности, количество рабочих элементов. Кроме того, мы адаптируем метод поиска оптимальных значений из работы [14], который позволяет избежать многократного запуска верификатора SPIN в методе контрпримеров. Существенным ограничением изложенной реализации нашего метода является малый размер Promela-модели, который допускает одновременную активность только 255 параллельных процессов, в то время как в реальных графических процессорах их могут быть тысячи. Поэтому мы планируем разработать подход к масштабированию результатов нашего метода контрпримеров, в котором может использоваться то, что размер реальных данных и количество процессорных элементов реальных графических процессоров является степенью числа 2. Развитием этой темы будет применение нашего подхода к реальным OpenCL-программам и конкретным графическим процессорам.

References

[1] J. Ansel, S. Kamil, K. Veeramachaneni, and et al., “OpenTuner: An extensible framework for program autotuning”, in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 303–316.

[2] J. Beckingsale, O. Pearce, I. Laguna, and T. Gamblin, “Apollo: Reusable models for fast, dynamic tuning of input-dependent code”, in *Proc. of 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2017.

[3] C. Chen, J. Chame, and M. Hall, “CHILL: A framework for composing high-level loop transformations”, *Technical Report 08-897. Los Angeles, CA*, pp. 136–150, 2008.

[4] M. Christen, O. Schenk, and H. Burkhart, “PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures”, in *Proc. of 2011 IEEE International Parallel Distributed Processing Symposium*, IEEE, 2011.

[5] R. Whaley and J. Dongarra, “Automatically tuned linear algebra software”, in *Proc. of the ACM/IEEE Conference on Supercomputing*, IEEE, 1998.

[6] M. Frigo and S. G. Johnson, “The design and implementation of FFTW3”, *IEEE*, vol. 93(2), 2005.

[7] G. Fursin, Y. Kashnikov, A. Memon, and et al., “Milepost GCC: machine learning enabled self-tuning compiler”, *Int J Parallel Prog*, vol. 39(3), pp. 296–327, 2011.

[8] C. Nugteren and V. Codreanu, “CLTUne: A generic auto-tuner for OpenCL kernels”, in *Proc. of 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, IEEE, 2015.

[9] A. Rasch and S. Gorlatch, “ATF: A Generic, Directive-Based Auto-Tuning Framework”, *Concurrency and Computation: Practice and Experience*, vol. 31(5), 2018.

[10] C. Tapus, I. Chung, and J. Hollingsworth, “Active harmony: towards automated performance tuning”, in *Proc. of 2002 ACM/IEEE Conference on Supercomputing*, IEEE, 2002.

[11] R. Vuduc, J. W. Demmel, and K. A. Yelick, “OSKI: A library of automatically tuned sparse matrix kernels”, in *Proc. of Scientific Discovery through Advanced Computing Conference*, IEEE, 2005.

[12] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, *Handbook of Model Checking*. Springer International Publishing, 2018, ch. 1. Introduction to Model Checking, pp. 1–13.

[13] T. C. Ruys and E. Brinksma, “Experience with Literate Programming in the Modelling and Validation of Systems”, in *Proc. of the 4th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’98)*, Springer, 1998, pp. 393–408.

[14] T. Ruys, “Optimal Scheduling Using Branch and Bound with SPIN 4.0”, in *Proc. of Model Checking Software. SPIN 2003*, Springer, 2003.

[15] E. Brinksma, A. Mader, and A. Fehnker, “Verification and optimization of a PLC control schedule”, *International Journal on Software Tools for Technology Transfer*, vol. 4, pp. 21–33, 2002.

[16] A. Wijs, J. V. D. Pol, and E. M. Bortnik, “Solving scheduling problems by untimed model checking: The clinical chemical analyser case study.”, in *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, 2005, pp. 54–61.

[17] R. Malik and P. Pena, “Optimal Task Scheduling in a Flexible Manufacturing System using Model Checking”, *IFAC-PapersOnLine*, vol. 51, no. 7, pp. 230–235, 2018.

[18] *The OpenCL Specification*. Khronos OpenCL working group, 2021.

[19] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[20] C. A. R. Hoare, *Communicating sequential processes*. Prentice-Hall, 1985.

[21] M. Gaspari and G. Zavattaro, “An Algebra of Actors”, in *Proc. of Formal Methods for Open Object-Based Distributed Systems. FMOODS 1999*, Springer, 1999.

[22] A. Cimatti, S. Edelkamp, M. Fox, D. Magazzeni, and E. Plaku, "Automated Planning and Model Checking (Dagstuhl Seminar 14482)", *Dagstuhl Reports*, vol. 4, no. 11, pp. 227–245, 2015, ISSN: 2192-5283. DOI: 10.4230/DagRep.4.11.227. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2015/4973.

[23] P. N. Glaskowsky, *NVIDIA's Fermi: The First Complete GPU Computing Architecture*. NVIDIA Corporation, 2009.

MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 28, NO. 4, 2021
journal homepage: www.mais-journal.ru

THEORY OF COMPUTING

On the Satisfiability and Model Checking for one Parameterized Extension of Linear-time Temporal Logic

A. R. Gnatenko[1], V. A. Zakharov[1,2]

DOI: 10.18255/1818-1015-2021-4-356-371

[1]National Research University Higher School of Economics (HSE), 20 Myasnitskaya str., Moscow 101000, Russia.
[2]Ivannikov Institute for System Programming of the RAS, 25 Alexander Solzhenitsyn str., Moscow 109004, Russia.

MSC2020: 68W10
Research article
Full text in Russian

Received November 15, 2021
After revision December 1, 2021
Accepted December 8, 2021

Sequential reactive systems are computer programs or hardware devices which process the flows of input data or control signals and output the streams of instructions or responses. When designing such systems one needs formal specification languages capable of expressing the relationships between the input and output flows. Previously, we introduced a family of such specification languages based on temporal logics *LTL*, *CTL* and CTL^* combined with regular languages. A characteristic feature of these new extensions of conventional temporal logics is that temporal operators and basic predicates are parameterized by regular languages. In our early papers, we estimated the expressive power of the new temporal logic *Reg-LTL* and introduced a model checking algorithm for *Reg-LTL*, *Reg-CTL*, and $Reg\text{-}CTL^*$. The main issue which still remains unclear is the complexity of decision problems for these logics. In the paper, we give a complete solution to satisfiability checking and model checking problems for *Reg-LTL* and prove that both problems are PSPACE-complete. The computational hardness of the problems under consideration is easily proved by reducing to them the intersection emptyness problem for the families of regular languages. The main result of the paper is an algorithm for reducing the satisfiability of checking *Reg-LTL* formulas to the emptiness problem for Buchi automata of relatively small size and a description of a technique that allows one to check the emptiness of the obtained automata within space polynomial of the size of input formulas.

Keywords: temporal logics; regular language; transducer; model checking; satisfiability checking; Buchi automata; emptiness problem

INFORMATION ABOUT THE AUTHORS

Anton Romanovich Gnatenko
correspondence author
orcid.org/0000-0003-1499-2090. E-mail: gnatenko.cmc@gmail.com
Master student.

Vladimir Anatolyevich Zakharov
correspondence author
orcid.org/0000-0002-3794-9565. E-mail: zakh@cs.msu.su
Prof. Dr.Sc.

For citation: A. R. Gnatenko and V. A. Zakharov, "On the Satisfiability and Model Checking for one Parameterized Extension of Linear-time Temporal Logic", *Modeling and analysis of information systems*, vol. 28, no. 4, pp. 356-371, 2021.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 28, № 4, 2021
сайт журнала: www.mais-journal.ru

THEORY OF COMPUTING

О верификации моделей и проверке выполнимости формул одного параметрического расширения темпоральной логики линейного времени

А. Р. Гнатенко[1], В. А. Захаров[1,2]

DOI: 10.18255/1818-1015-2021-4-356-371

[1]Национальный исследовательский университет "Высшая школа экономики", ул. Мясницкая, д. 20, г. Москва, 101000 Россия.
[2]Институт системного программирования им. В. П. Иванникова РАН, ул. А. Солженицына, д. 25, г. Москва, 109004 Россия.

УДК 517.9
Научная статья
Полный текст на русском языке

Получена 15 ноября 2021 г.
После доработки 1 декабря 2021 г.
Принята к публикации 8 декабря 2021 г.

К последовательным реагирующим системам относятся компьютерные программы и вычислительные устройства, которые обрабатывают потоки входных данных или сигналов управления и генерируют на выходе последовательности команд или результатов вычислений. Для проектирования таких систем полезно иметь формальные языки спецификаций, способные выражать отношения между входными и выходными потоками данных. В предшествующих работах нами было предложено семейство таких языков спецификаций, представляющих собой расширение темпоральных логик LTL, CTL и CTL^* за счет использования регулярных языков в качестве параметров темпоральных операторов. Мы провели сравнительный анализ выразительных возможностей нового расширения темпоральной логики линейного времени $Reg\text{-}LTL$ и предложили алгоритмы верификации моделей для новых расширений логик $Reg\text{-}LTL$, $Reg\text{-}CTL$, и $Reg\text{-}CTL^*$. Однако вопрос о сложности задач верификации моделей и проверки выполнимости формул указанных логик оставался открытым. В этой статье мы восполняем этот пробел в наших исследованиях и показываем, что для темпоральной логики $Reg\text{-}LTL$ обе задачи являются Pspace-полными. Вычислительная трудность рассматриваемых задач легко доказывается сведением к ним проблемы пустоты пересечения семейств регулярных языков. Основным результатом статьи является алгоритм сведения задачи проверки выполнимости формул логики $Reg\text{-}LTL$ к проблеме пустоты автоматов Бюхи сравнительно небольшого размера и описание стратегии, позволяющей проверять пустоту полученных автоматов с использованием объема памяти, полиномального относительно размера исходных формул.

Ключевые слова: темпоральные логики; регулярный язык; автомат-преобразователь; верификация моделей; проверка выполнимости; автоматы Бюхи; проблема пустоты

ИНФОРМАЦИЯ ОБ АВТОРАХ

Антон Романович Гнатенко
автор для корреспонденции
orcid.org/0000-0003-1499-2090. E-mail: gnatenko.cmc@gmail.com
студент магистратуры.

Владимир Анатольевич Захаров
автор для корреспонденции
orcid.org/0000-0002-3794-9565. E-mail: zakh@cs.msu.su
профессор, доктор физ.-мат. наук.

Для цитирования: A. R. Gnatenko and V. A. Zakharov, "On the Satisfiability and Model Checking for one Parameterized Extension of Linear-time Temporal Logic", *Modeling and analysis of information systems*, vol. 28, no. 4, pp. 356-371, 2021.

Введение

Амир Пнуели был, вероятно, первым, кто обратил внимание на возможность использования темпоральных логик для описания поведения последовательных и параллельных программ [1]. Темпоральные формулы хорошо подходят для этой цели, когда поведение вычислительной системы рассматривается как совокупность последовательностей событий. За полвека, прошедшие со времени публикации пионерской работы Пнуели, появилось и было подробно изучено большое разнообразие темпоральных логик, таких как темпоральная логика линейного времени (LTL) [2], логика деревьев вычислений (CTL) и ее обобщение (CTL^*) [3], метрическая темпоральная логика (MTL) [4] и пр. Логика LTL является довольно выразительным формальным языком для рассуждения об устройстве бесконечных последовательностей событий, сигналов, состояний вычисления. Формулы LTL строятся из символов конечного алфавита атомарных высказываний при помощи булевых связок и темпоральных модальностей, или операторов, таких как **F** (оператор происшествия, когда-нибудь), **G** (оператор неограниченной инвариантности, всегда), **U** (оператор ограниченной инвариантности, до тех пор, пока) и некоторых других. Например, для заданного алфавита элементарных событий $\{a, b\}$ формула $\mathbf{F}a$ выполняется для тех и только тех последовательностей событий, которые включают в себя хотя бы одно событие a (т.е., событие a когда-нибудь происходит), а формула $\mathbf{G}b$ требует, чтобы событие b происходило в каждый момент времени (событие b происходит всегда). При помощи формул LTL можно описывать разнообразные свойства вычислений, включая требования живости, безопасности, справедливости [5].

Основываясь на концепциях логики LTL, многие авторы предложили и исследовали ряд ее фрагментов и расширений, к числу которых относятся расширенная LTL [6], квантифицированная LTL [7], динамическая LTL [8, 9], темпоральные дескриптивные логики [10], пространственно-временные логики [11] и целый ряд других. Логики этого семейства представляют двоякий интерес. С одной стороны, они интересны с чисто теоретической точки зрения как некие разумные математические теории, имеющие определенные области применения. В связи с этим возникает необходимость изучить их выразительные возможности, аксиоматизацию, алгоритмические проблемы. С другой стороны, эти логики могут использоваться в качестве языков спецификации для различных типов программного и аппаратного обеспечения, когда приходится иметь дело с задачами верификации, статического анализа, проверкой качества функционирования систем (см. [12, 13]).

Существование большого разнообразия темпоральных логик, которым, казалось бы, недостает строгой систематизации, легко объяснимо. Для верификации разных типов программ, систем обработки информации и вычислительной и коммуникационной аппаратуры требуются модели совершенно разной природы и, следовательно, разные языки спецификации. Рассмотрим, например, последовательные реагирующие системы, представляющие собой компьютерные программы или аппаратные устройства, которые обрабатывают потоки входных данных или управляющих сигналов и выдают на выходе последовательности команд или результатов действий. Эти действия и порядок их выполнения зависят не только от текущего принятого сигнала, но и от всех предыдущих управляющих сигналов. Таким образом, поведение реагирующей системы характеризуется некоторым соответствием (бинарным отношением) между потоком сигналов и потоком действий. Поэтому при разработке формальных методов верификации реагирующих систем необходим, прежде всего, адекватный формальный язык, позволяющий описывать не последовательности событий, а отношения между последовательностями входных и выходных событий.

Хотя темпоральная логика широко используется уже долгое время для спецификации и верификации программ, не было предпринято никаких серьезных попыток разработать вариант темпорального языка, который подходил бы для работы с реагирующими системами такого рода. Ключевым моментом является то, что темпоральные формулы обычно интерпретируются на последовательностях событий, в то время как поведение реагирующей системы проявляется в парах

последовательностей — входных и выходных — с определенными отношениями между ними. Ни один из традиционных языков темпоральной спецификации вычислений не содержит специальных средств для выражения свойств таких бинарных отношений.

Чтобы преодолеть этот недостаток, мы предложили дополнить временные операторы параметрами, в качестве которых выступают регулярные языки, и получили таким образом *Reg*-варианты темпоральных логик *LTL*, *CTL*, и CTL^*, описанные в работах [14—16]). Например, формулу $\mathbf{G}\varphi$ ("свойство φ всегда выполняется") можно сделать более специфичной, записав ее как $\mathbf{G}_L\varphi$, что означает "свойство φ выходной последовательности выполняется всякий раз, когда входная последовательность представляет собой слово из языка L". Конечно, можно использовать любой тип языков для параметризации временных модальностей, но для того, чтобы традиционные задачи, связанные с применением логик, оставались алгоритмически разрешимыми мы ограничиваемся регулярными языками. В статье [17] было показано, что вопрос о выполнимости даже очень простых формул может оказаться алгоритмически неразрешимым, если позволить использовать в качестве параметров темпоральных операторов контекстно-свободные языки.

Следуя основным положениям теоретико-автоматного подхода к решению задач верификации моделей относительно темпоральных спецификаций (см. [13, 18]), авторы статьи [14] предложили метод трансляции формул логики *Reg-LTL* в автоматы Бюхи; таким образом, задачи верификации моделей и проверки выполнимости формул *Reg-LTL* были сведены к проблеме пустоты для автоматов Бюхи. В статье [16] этот метод трансляции был обобщен и адаптирован для работы с логикой $Reg\text{-}CTL^*$. Однако в обоих случаях размер получаемых в результате трансляции автоматов оценивался двойной экспонентой, зависящей от размера темпоральной формулы; это давало верхнюю оценку ExpSpace вычислительной сложности рассматриваемых задач. С другой стороны, в статье [16] было показано, что задача верификации моделей для логики $Reg\text{-}CTL^*$ является PSpace-трудной. Таким образом, между верхней и нижней оценками вычислительной сложности задач проверки выполнимости формул рассматриваемых параметризованных модификаций темпоральных логик образовался разрыв, и главная цель данной статьи — восполнить этот пробел в наших исследованиях указанных логик.

Мысль о том, что выразительные возможности темпоральной логики можно усилить за счет повышения «чувствительности» темпоральных операторов, не нова. Ограниченность *LTL* стала очевидной, как только было показано, что формулы этой логики позволяют описывать лишь такие регулярные языки, которые можно задавать без привлечения операции итерации Клини [19]. В статье [6] Вольпер привел пример очень простого свойства, которое невозможно охарактеризовать формулами логики *LTL* (в статье [17] приведено другое доказательство этого факта при помощи темпоральной разновидности игр Эренфойхта–Фреше [20]). Чтобы сделать язык логики *LTL* более выразительным, Вольпер предложил ввести новый вид темпоральных операторов, область действия которых определяется регулярными грамматиками [6]. Эта идея получила дальнейшее развитие, в котором для усиления темпоральных операторов вместо грамматик были задействованы конечные автоматы [21] и даже альтернирующие автоматы [22].

В статье [8] Хенриксен и Тиагаранжан предложили альтернативный подход к расширению логики *LTL*. Они использовали регулярные выражения для параметризации темпоральных операторов и получили динамическую темпоральную логику линейного времени (*DLTL*), которая очень похожа на логику *Reg-LTL*. Авторы статьи [8] разработали разрешающую процедуру для этой логики, позволяющую за экспоненциальное время транслировать логические формулы в автоматы Бюхи, размер которых так же экспоненциально зависит от размера формул. Хотя для интерпретации формул *DLTL* используются одиночные последовательности событий, мы воспользовались предложенным в статье [8] методом трансляции, адаптировали его к нашей логике *Reg-LTL* и получили полиномиальные по объему используемой памяти алгоритмы верификации моделей и проверки

выполнимости формул логики *Reg-LTL*. Эти алгоритмы и являются основным результатом нашей статьи. Принимая во внимание нижние оценки сложности указанных задач, полученные ранее в статьях [15—17], мы приходим к выводу о том, что задачи верификации моделей и проверки выполнимости формул темпоральных логик *Reg-CTL*, *Reg-LTL*, и *Reg-CTL** являются Pspace-полными.

Оглавление нашей статьи таково. В разделе 1 описаны автоматы-преобразователи, которые служат формальной моделью последовательных реагирующих систем. Синтаксис и семантика темпоральной логики *Reg-LTL*, используемой для спецификации поведения автоматов-преобразователей, представлены в разделе 2. В разделе 3 рассказывается о новом методе трансляции формул *Reg-LTL* в автоматы Бюхи. Построенному на основе этой трансляции алгоритму проверки выполнимости формул *Reg-LTL* посвящен раздел 4. В нем показано, что для проверки выполнимости формул достаточно использовать объем памяти, полиномиальный относительно размера формул. Далее, в разделе 5 говорится о том, каким образом описанный метод трансляции формул *Reg-LTL* в автоматы Бюхи применять для верификации моделей последовательных реагирующих систем относительно формальных спецификаций их поведения. В заключительном разделе 6 мы подводим итоги проведенных исследований задач спецификации и верификации моделей последовательных реагирующих систем с использованием темпоральных логик.

1. Модель последовательных реагирующих систем

Область применения моделей вычислений с конечным числом состояний (конечных автоматов) обширна. Простота устройства автоматов позволяет использовать их для автоматизации решения задач синтеза и анализа разнообразных управляющих систем. В частности, автоматы-преобразователи (машины с конечным числом состояний) хорошо подходят для моделирования последовательных реагирующих систем, включая интерпретаторы, системные драйверы, контроллеры сетевые коммутаторы. В этой главе мы приводим определения основных понятий, относящихся к этой модели вычислений.

Рассмотрим два конечных алфавита C и $\mathcal{A}$. Элементы множества C мы будем называть *сигналами*, а конечные последовательности этих элементов — *потоками сигналов*. Сигналы можно истолковывать как входные данные, управляющие команды или запросы, которые поступают на вход реагирующей системы из окружающей среды. Множество всех потоков сигналов будем обозначать записью C^*, а конкатенацию потоков $u, v \in C$ записывать как uv. Для обозначения пустого потока сигналов используем символ ε.

Символы алфавита $\mathcal{A}$ будем называть *элементарными действиями*, а конечные слова в алфавите $\mathcal{A}$ — *составными действиями* (далее, просто действиями). Элементарные действия являются абстракциями простейших операций, которые способна совершать реагирующая система, а составные действия являются последовательной композицией простейших операций. В общем случае разные действия могут давать одинаковый эффект, и поэтому их можно интерпретировать на полугруппе $(S, e, \circ)$, порожденной множеством элементарных действий $\mathcal{A}$ с операцией композиции $\circ$ и нейтральным элементом e. Элементы множества S будем называть состояниями данных. В результате применения составного действия $h = a_1 a_2 \ldots a_k$ к состоянию данных s будет получено новое состояние данных $s' = s \circ a_1 \circ a_2 \circ \cdots \circ a_k$.

В этой статье мы ограничимся рассмотрением самой простой интерпретации, когда $(S, e, \circ)$ является свободной полугруппой с множеством образующих $\mathcal{A}$. В этом случае множество состояний данных S совпадает с множеством составных действий $\mathcal{A}^*$, операцией $\circ$ является конкатенация слов в алфавите $\mathcal{A}$, а начальным состоянием данных e является пустое слово в алфавите $\mathcal{A}$. Более общий случай моделей реагирующих систем над произвольными полугруппами рассматривался в работах [23—25].

Автомат-преобразователь над множествами сигналов C и элементарных действий $\mathcal{A}$ задается пятеркой $\pi = (Q, C, \mathcal{A}, q_{init}, T)$, где

- Q — конечное множество *состояний управления*;
- $q_{init} \in Q$ — *начальное состояние*;
- $T \subseteq Q \times C \times Q \times \mathcal{A}^*$ — конечное *отношение переходов*.

Четверку $(q', c, q'', h) \in T$ будем называть *переходом*. Каждый такой переход соответствует простейшему шагу вычисления реагирующей системы: если система, пребывающая в состоянии q', получает на входе сигнал c, то она выполняет действие h и переходит в состояние q''. Как обычно, мы будем использовать запись $q' \xrightarrow{c,\ h} q''$ вместо $(q', c, q'', h) \in T$.

Прогон преобразователя π — это бесконечная последовательность переходов

$$\rho = q_1 \xrightarrow{c_1, h_1} q_2 \xrightarrow{c_2, h_2} q_3 \xrightarrow{c_3, h_3} \dots$$

Прогон называется *начальным*, если $q_1 = q_{init}$.

Прогоны автоматов-преобразователей — это и есть абстрактные образы вычислений реагирующих систем. Однако наблюдаемое поведение реагирующей системы выглядит совсем иначе. Стороннему наблюдателю не доступно устройство автомата-преобразователя, и поэтому он может обозревать лишь поступающие на его вход сигналы и регистрировать результаты выполнения действий после каждого входного сигнала. Поэтому при анализе поведения реагирующих систем вместо прогонов мы будем иметь дело с наблюдаемыми трассами, последовательностями пар, состоящими из поступивших на вход сигналов и достигнутых состояний данных.

Трассой над множествами сигналов C и элементарных действий $\mathcal{A}$ называется всякая пара $tr = \langle s_0, \alpha \rangle$, в которой первая компонента $s_0 \in \mathcal{A}^*$, а вторая компонента α является бесконечной последовательностью пар $\alpha = \{(c_i, s_i)\}_{i \geqslant 1}$, $c_i \in C$, $s_i \in \mathcal{A}^*$. Множество всех возможных трасс обозначим записью $Traces$. Мы будем обозначать записью $tr|^i$ суффикс трассы tr, т.е. трассу $tr|^i = (s_i, \alpha|^i)$, в которой $\alpha|^i = (c_{i+1}, s_{i+1}), (c_{i+2}, s_{i+2}), \dots$.

Каждое состояние данных $s_0 \in \mathcal{A}^*$ и прогон $\rho = q_1 \xrightarrow{c_1, h_1} q_2 \xrightarrow{c_2, h_2} \dots$ автомата-преобразователя π порождают трассу $tr = \langle s_0, \alpha \rangle$, в которой последовательность пар $\alpha = (c_1, s_1), (c_2, s_2), \dots, (c_i, s_i), \dots$ удовлетворяет равенству $s_i = s_{i-1} h_i$ для любого $i \geq 1$. Множество $Traces(\pi)$ всех трасс, порожденных начальным состоянием данных e и всеми возможными начальными прогонами ρ_{init} автомата-преобразователя π, формализует понятие наблюдаемого поведения последовательной реагирующей системы, моделируемой автоматом π. Тогда свойством наблюдаемого поведения системы является всякое подмножество $Prop$ множества трасс $Traces$. Реагирующая система, моделируемая автоматом-преобразователем π, удовлетворяет свойству наблюдаемого поведения $Prop$ в том и только том случае, если выполняется отношение $Traces(\pi) \subseteq Prop$.

2. Язык спецификаций поведения реагирующих систем *Reg-LTL*

В основу формального языка спецификаций *Reg-LTL*, предназначенного для описания свойств наблюдаемого поведения реагирующих систем, положен язык темпоральной логики линейного времени *LTL*. Новый логический формализм имеет две отличительные особенности:

- атомарными формулами являются предикаты на множестве состояний данных $\mathcal{A}^*$, т.е. языки в алфавите $\mathcal{A}$;
- все темпоральные операторы параметризованы отдельными сигналами из множества C или множествами (языками) потоков сигналов из совокупности C^*.

В общем случае (см. [14]) базовыми предикатами и параметрами темпоральных операторов могут быть произвольные языки в алфавитах сигналов C и элементарных действий $\mathcal{A}$. Но для построения эффективных разрешающих процедур приходится ограничиваться регулярными языками. Как известно, регулярные языки можно задавать разными способами; один из наиболее простых и эффективных — задавать их при помощи конечных детерминированных автоматов-распознавателей.

В дальнейшем, упоминая о регулярных языках при описании синтаксиса логики *Reg-LTL*, мы будем подразумевать, что каждый регулярный язык представлен минимальным детерминированным автоматом-распознавателем (автоматом Рабина-Скотта), допускающим этот язык.

Формулы логики *Reg-LTL* определяются следующим образом:

1. всякий регулярный язык P в алфавите $\mathcal{A}$ является формулой (базовым предикатом);
2. если выражения φ_1, φ_2 — формулы, то $\neg\varphi_1$ and $\varphi_1 \wedge \varphi_2$ также являются формулами;
3. если выражения φ_1, φ_2 — формулы, $c \in \mathcal{C}$, и L — регулярный язык в алфавите $\mathcal{C}$, то выражения $\mathbf{X}_c\,\varphi_1$ и $\varphi_1 \mathbf{U}_L \varphi_2$ являются формулами.

Семантика языка *Reg-LTL* определяется отношением выполнимости формул на трассах, которые выступают в роли интерпретационных структур. Для заданной трассы $tr = \langle s_0, \alpha \rangle$, где $\alpha = \{(c_i, h_i)\}_{i \geqslant 1}$ мы будем использовать запись $tr \models \psi$ для обозначения того, что формула ψ выполняется на трассе tr. Отношение выполнимости формул $\models$ определяется индуктивно следующим образом:

1. $tr \models P \iff s_0 \in P$ для любого базового предиката P;
2. $tr \models \neg\varphi \iff$ если неверно, что $s \models \varphi$;
3. $tr \models \psi_1 \wedge \psi_2 \iff tr \models \psi_1$ и $tr \models \psi_2$;
4. $tr \models \mathbf{X}_c\varphi \iff c = c_1$ и $tr|^1 \models \varphi$;
5. $tr \models \varphi_1 \mathbf{U}_L \varphi_2 \iff$ существует такое $i \geqslant 0$, что поток сигналов $c_1 c_2 \ldots c_i$ принадлежит языку L и $tr|^i \models \varphi_2$, и при этом для любого j, $0 \leqslant j < i$, всякий раз, когда поток сигналов $c_1 c_2 \ldots c_j$ принадлежит языку L, справедливо соотношение $tr|^j \models \varphi_1$.

Как видно из приведенного определения, темпоральные операторы $\mathbf{X}$ и $\mathbf{U}$ имеют в логике *Reg-LTL* такой же смысл, как и в основополагающей логике *LTL* с той лишь разницей, что теперь при продвижении времени приходится учитывать, принадлежит ли поток сигналов трассы tr, на которой проверяются формулы, тем языкам (в статьях [14, 17] они называются шаблонами поведения окружающей среды), которыми параметризованы эти операторы.

При помощи отрицания и конъюнкции можно выразить формулами другие булевы связки, наподобие дизъюнкции $\vee$ или импликации $\longrightarrow$. Кроме того, точно так же, как и в логике *LTL* можно определить другие темпоральные операторы, такие как $\mathbf{F}_L\varphi \equiv true\,\mathbf{U}_L\varphi$ и $\mathbf{G}_L\varphi \equiv \neg\mathbf{F}_L\neg\varphi$.

Каждая формула φ логики *Reg-LTL* характеризует множество трасс $Traces(\varphi) = \{tr \in Traces \mid tr \models \varphi\}$. Будем говорить, что формула φ *выполнима*, если $Traces(\varphi) \neq \varnothing$; также будем говорить, что автомат-преобразователь π *удовлетворяет формуле* φ (обозначается записью $\pi \models \varphi$) если имеет место включение $Traces(\pi) \subseteq Traces(\varphi)$.

Выразительные возможности темпоральной логики *Reg-LTL* подробно исследованы в статье [17]. В частности, были выделены два фрагмента $\mathcal{LP}$-1-*LTL* и $\mathcal{LP}$-*n*-*LTL*, которые позволили сравнить параметризованную темпоральную логику *Reg-LTL* с темпоральной логикой *LTL* и монодической логикой *S*1*S*. Формулы первого из этих фрагментов $\mathcal{LP}$-1-*LTL* строятся над однобуквенным алфавитом $\mathcal{C} = \{c\}$ входных сигналов и произвольным конечным алфавитом элементарных действий $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$. В качестве параметров темпоральных операторов в формулах этого фрагмента разрешается использовать любые регулярные языки над алфавитом $\{c\}$, а в качестве базовых предикатов используются регулярные языки вида $P_i = \mathcal{A}^* a_i$. При таких ограничениях появляется возможность сравнивать семантики логик *LTL* и $\mathcal{LP}$-1-*LTL* на одном и том же классе моделей (трасс). В [17] было показано, что любая формула *LTL* имеет эквивалентную формулу $\mathcal{LP}$-1-*LTL*, но при этом есть такие формулы $\mathcal{LP}$-1-*LTL*, для которых не существует эквивалентных формул *LTL*. Фрагмент $\mathcal{LP}$-*n*-*LTL* определяется аналогично, с той лишь разницей, что алфавиты $\mathcal{C}$ и $\mathcal{A}$ совпадают, т. е. $\mathcal{C} = \mathcal{A} = \{a_1, a_2, \ldots, a_n\}$. В той же статье [17] удалось установить, что класс моделей, на которых выполняются формулы $\mathcal{LP}$-*n*-*LTL*, в точности совпадает с классом сверхязыков, распознаваемых конечными автоматами Маллера. Отсюда следует, что логики $\mathcal{LP}$-*n*-*LTL* и *S*1*S* имеют одинаковые выразительные возможности.

В следующем разделе мы приступим к описанию нашего метода решения двух основных задач, связанных с описанной здесь логикой *Reg-LTL*. Задача проверки выполнимости формул состоит в том, чтобы для произвольной заданной формулы логики *Reg-LTL* выяснить, является ли эта формула выполнимой. Задача верификации моделей состоит в том, чтобы для произвольного заданного автомата-преобразователя π и произвольной заданной формулы φ логики *Reg-LTL* выяснить, справедливо ли отношение $\pi \models \phi$. Как видно из результатов, представленных в статье [17], обе рассматриваемые задачи являются Pspace-трудными, и поэтому нашей целью является разработка алгоритмов, способных решить указанные задачи с использованием полиномиального объема памяти.

3. Проверка выполнимости формул при помощи автоматов Бюхи

Теоретико-автоматный подход к проверке выполнимости формул темпоральной логики линейного времени, первоначально предложенный в статье [18] и опирающийся на идеи работы [26], предполагает построение для проверяемой формулы φ автомата Бюхи B_φ, который удовлетворяет следующему требованию: φ выполнима тогда и только тогда, когда $Lang(B_\varphi) \neq \emptyset$. Тогда проверка выполнимости формул сводится к задаче проверки пустоты для автоматов Бюхи. Эта задача решается детерминированным алгоритмом за время, линейное относительно размера автомата, или недетерминированным алгоритмом с использованием памяти, объем которой оценивается логарифмом от размера автомата. Обычно автомат Бюхи B_φ строится так, чтобы язык $Lang(B_\varphi)$ состоял из всех ω-слов, представляющих интерпретации (трассы), на которых выполняется проверяемая формула φ. Для темпоральной логики *LTL* трансляция формул в автоматы Бюхи может быть осуществлена за время, экспоненциальное относительно размера формулы и с использованием полиномиального объема памяти. Размер самого автомата B_φ при этом оценивается экспонентой от размера формулы φ, однако проверку пустоты языка $Lang(B_\varphi)$ можно проводить «на лету» без построения автомата в явном виде. Так удается показать, что задача проверки выполнимости формул *LTL* принадлежит классу сложности Pspace.

Авторы статьи [14] применили указанный подход к решению задачи верификации моделей для логики *Reg-LTL*. Однако предложенный в этой статье метод трансляции формул *Reg-LTL* в автоматы Бюхи оказался слишком трудоемким: размер получавшихся автоматов оценивался двойной экспонентой от размера темпоральных формул. Поэтому для проверки выполнимости формул требовался объем памяти, экспоненциально зависящий от размера формул. В данной работе мы модифицируем этот алгоритм, воспользовавшись приемами, которые были предложены в статье [8], и получаем полиномиальные по объему памяти процедуры верификации моделей и проверки выполнимости формул рассматриваемой логики *Reg-LTL*.

3.1. Автоматы Рабина–Скотта и автоматы Бюхи

Конечный автомат Рабина–Скотта (далее автомат-распознаватель) над алфавитом Σ задается пятеркой $A = (Q, \Sigma, q_{init}, \delta, F)$, где Q — конечное множество *состояний*, включающее *начальное состояние* $q_{init} \in Q$ и подмножество *допускающих состояний* $F \subseteq Q$, а $\delta : Q \times \Sigma \mapsto Q$ — *функция переходов*. Эту функцию можно распространить на множество слов Σ^* обычным образом, полагая $\delta(q, \varepsilon) = q$, и $\delta(q, \sigma x) = \delta(\delta(q, \sigma), x)$ для всех состояний $q \in Q$, $\sigma \in \Sigma$ и слов $x \in \Sigma^*$. Размер автомата A — это число его состояний $|A| = |Q|$.

Автомат-распознаватель A *допускает* слово $x \in \Sigma^*$ тогда и только тогда, когда $\delta(q_{init}, x) \in F$. Множество всех слов $Lang(A)$, допускаемых автоматом A, называется *языком* этого автомата. Для произвольного регулярного языка L обозначим записью $|L|$ размер минимального автомата, для которого выполнено равенство $L = Lang(A)$.

Конечные автоматы могут работать не только на конечных словах. Обозначим записью Σ^ω множество всех бесконечных последовательностей букв алфавита Σ, которые будем называть ω-словами.

Недетерминированный *автомат Бюхи* над алфавитом Σ, как и автомат Рабина–Скотта, задается пятеркой $B = (Q, \Sigma, q_{init}, \Delta, F)$, которая включает в себя множество состояний Q, начальное состояние q_{init}, отношение переходов $\Delta \subseteq Q \times \Sigma \times Q$ и подмножество допускающих состояний $F \subseteq Q$. Однако функционирование автоматов Бюхи определяется иначе. Для заданного ω-слова $x = \sigma_0\sigma_1\sigma_2\cdots \in \Sigma^\omega$, *прогоном* автомата Бюхи B на слове x называется такая бесконечная последовательность состояний $q_0, q_1, q_2, \ldots$, в которой $q_0 = q_{init}$, и для каждого $i, i \geqslant 0$, выполняется включение $(q_i, \sigma_i, q_{i+1}) \in \Delta$. Такой прогон считается *допускающим*, если в нем бесконечно много раз встречаются состояния из допускающего подмножества F. Будем говорить, что ω-слово $x \in \Sigma^\omega$ *допускается* автоматом B, если существует допускающий прогон автомата B на слове x. Множество всех ω-слов, которые допускает автомат Бюхи B обозначим записью $Lang(B)$. Размером $|B|$ автомата Бюхи B назовем число его состояний.

3.2. Трансляция формул *Reg-LTL* в автоматы Бюхи

Рассмотрим (бесконечное) множество пар слов $\mathcal{C} \times \mathcal{A}^*$, где $\mathcal{C}$ — множество сигналов, а $\mathcal{A}$ — множество элементарных действий. Тогда бесконечная последовательность $\beta = \{(c_i, h_i)\}_{i\geqslant 1}, c_i \in \mathcal{C},\ h_i \in \mathcal{A}^*$, может рассматриваться как ω-слово в указанном алфавите. Это слово однозначно характеризует трассу $tr_\beta = \langle e, \alpha\rangle$, где $\alpha = \{(c_i, h_1h_2 \ldots h_i)\}_{i\geqslant 1}$. Наша цель состоит в том, чтобы для произвольной формулы φ логики *Reg-LTL* построить такой автомат Бюхи B_φ, размер которого не более чем экспоненциально зависит от размера формулы B_φ, удовлетворяющий при этом следующему требованию: для любого ω-слова $\beta \in (\mathcal{C} \times \mathcal{A}^*)^\omega$ верно соотношение $\beta \in Lang(B_\varphi) \iff tr_\beta \models \varphi$.

Для заданной формулы φ логики *Reg-LTL* выделим в этой формуле все регулярные языки $L_1, L_2, \ldots, L_k$ и $P_1, P_2, \ldots, P_m$, которые используются в формуле φ в качестве параметров темпоральных операторов и базовых предикатов соответственно. Рассмотрим минимальные по размеру автоматы Рабина-Скотта $A_i = (Q'_i, \mathcal{C}, q'_{init,i}, \delta'_i, F'_i), 1 \leq i \leq k$, и $D_j = (Q''_j, \mathcal{A}, q''_{init,j}, \delta''_j, \mathcal{F}''_j), 1 \leq j \leq m$, распознающие эти регулярные языки. Размером формулы φ будем называть величину $|\varphi| = \ell + \sum\limits_{i=1}^{k} |A_i| + \sum\limits_{j=1}^{m} |D_j|$, где ℓ — количество булевых связок в формуле φ.

При построении автомата Бюхи B_φ будем следовать принципам построения автоматов, характеризующих выполнимость темпоральных формул, которые изложены в монографиях [12, 13] применительно к формулам логики *LTL*.

Замыканием Фишера–Ладнера для формулы φ назовём наименьшее множество формул $Closure(\varphi)$, которое удовлетворяет следующим условиям:

- $\varphi \in Closure(\varphi)$,
- $\psi \in Closure(\varphi) \iff \neg\psi \in Closure(\varphi)$,
- если $\psi' \wedge \psi'' \in Closure(\varphi)$, то $\psi', \psi'' \in Closure(\varphi)$,
- если $\mathbf{X}_c\psi \in Closure(\varphi)$, то $\psi \in Closure(\varphi)$,
- если $\psi'\mathbf{U}_L\psi'' \in Closure(\varphi)$, то $\psi', \psi'' \in Closure(\varphi)$,

Подмножество $K \subseteq Closure(\varphi)$ называется *согласованным*, если K непротиворечиво как множество пропозициональных формул и при этом согласовано с семантикой темпоральных операторов, т. е. удовлетворяет условиям:

- для всякой формулы $\psi \in Closure(\varphi)$ верно $\psi \in K \iff \neg\psi \notin K$ (полагая при этом $\neg\neg\psi = \psi$),
- для всякой формулы $\psi' \wedge \psi'' \in Closure(\varphi)$ верно $\psi' \wedge \psi'' \in K \iff \psi', \psi'' \in K$,
- для всякой формулы $\psi'\mathbf{U}_L\psi'' \in Closure(\varphi)$ если $\psi'' \in K$ и $\varepsilon \in L$, то $\psi'\mathbf{U}_L\psi'' \in K$,

Совокупность всех согласованных подмножеств совокупности формул $Closure(\varphi)$ будем обозначать записью K_φ.

Положим $V = \bigcup_{i=1}^{k} Q'_i$, $W = \bigcup_{j=1}^{m} Q''_j$. Будем говорить, что множество состояний $R \subseteq W$ *достижимо*, если R содержит ровно по одному состоянию каждого из автоматов-распознавателей $D_j, 1 \leq j \leq m$, причем все эти состояния достижимы из соответствующих начальных состояний $q_{init,j}$

этих автоматов по одному и тому же слову, т.е. $R = \{\delta''_j(q''_{init,j}, h) : 1 \le j \le m\}$ для некоторого слова $h \in A^*$.

Автомат Бюхи $B_\varphi = (Q, C \times \mathcal{A}^*, Q_{init}, \Delta, F)$ над алфавитом $C \times \mathcal{A}^*$ определяется следующим образом.

(1) Состояниями этого автомата являются наборы из множества

$$Q \subseteq K_\varphi \times 2^V \times 2^{V \times \{0,1\}} \times 2^W \times \{0,1\} \times \{\circ, \bullet\}.$$

Каждый такой набор (K, C, S, R, b, θ) состоит из согласованного множества формул $K \in K_\varphi$, контрольного множества состояний $C \subseteq V$, выполняющего множества помеченных состояний $S \subseteq V \times \{0,1\}$, предикатного достижимого множества состояний $R \subseteq W$, фазы $b \in \{0,1\}$ и маркера $\theta \in \{\circ, \bullet\}$. Для того, чтобы набор (K, C, S, R, b, θ) принадлежал множеству состояний рассматриваемого автомата Бюхи B_φ необходимо и достаточно обеспечить соблюдение следующих требований:

- для каждого базового предиката P_j, $1 \le j \le m$, верно соотношение $P_j \in K \iff F''_j \cap R \neq \emptyset$;
- для каждой формулы $\psi \mathbf{U}_{L_i} \chi \in Closure(\varphi)$, где $1 \le i \le k$, выполняются условия
  - если $\chi \in K$, то $F'_i \cap C \neq \emptyset$,
  - если $q_{init,i} \in C$ и при этом либо $\varepsilon \notin L_i$, либо $\psi \in K$, то $\psi \mathbf{U}_{L_i} \chi \in K$,
  - если $\psi \mathbf{U}_{L_i} \chi \in K$, то либо верны оба включения $\varepsilon \notin L_i$ и $\psi \in K$, либо $(q_{init,i}, 1-b) \in S$,
  - если $(q, f) \in S$ для некоторого состояния $q \in Q'_i$ и некоторой фазы $f \in \{0,1\}$, то либо $q \in F'_i$ и $\chi \in K$, либо $\psi \in K$.

(2) Отношение переходов $\Delta \subseteq Q \times (C \times \mathcal{A}^*) \times Q$ содержит переход

$$\big((K, C, S, R, b, \theta), (c, h), (K', C', S', R', b', \theta')\big)$$

в том и только том случае, если выполнены следующие требования:

- для каждого j, $1 \le j \le m$, верно соотношение $q \in Q''_j \cap R \implies \delta''_j(q, h) \in R'$,
- для любой формулы $\mathbf{X}_d \psi \in Closure(\varphi)$ верно соотношение $\mathbf{X}_d \psi \in K \implies d = c, \psi \in K'$,
- $\psi \mathbf{U}_{L_i} \chi \in Closure(\varphi)$, где $1 \le i \le k$, выполняются условия
  - если $q' = \delta'_i(q, c) \in Q'_i \cap C'$ и при этом либо $q \notin F'_i$, либо $\psi \in K$, то $q \in C$,
  - если $(q, f) \in S$ для некоторого состояния $q \in Q_i$ и некоторой фазы f, и при этом либо $q \notin F'_i$, либо $\chi \notin K$, то $(\delta'_i(q, c), f) \in S'$,
  - если $\theta = \bullet$, то $b' = 1 - b$ и $\theta' = \circ$,
  - если $\theta = \circ$, то $b' = b$; при этом в случае, если $\chi \notin K$ или множество S содержит пару (q, b), для которой $q \notin F'_i$, то $\theta' = \circ$, а в противном случае $\theta' = \bullet$.

(3) Множество начальных состояний $Q_{init} = \{(K, C, S, R, 0, \bullet) : \varphi \in K, R —$ достижимое множество$\}$.

(4) Множество допускающих состояний $F = \{(K, C, S, R, b, \bullet) : K \in K_\varphi\}$.

Теорема 1. *Формула φ выполнима тогда и только тогда, когда $Lang(B_\varphi) \neq \emptyset$.*

Доказательство. (эскиз) Справедливость утверждения теоремы устанавливается, следуя той же схеме рассуждений, которая применяется при обосновании аналогичных теорем для темпоральных логик *LTL* [12, 13], *DLTL* [8], и, наконец, самой логики *Reg-LTL* [14].

Для каждого допускающего прогона $\rho = z_0, z_1, \ldots, z_i, \ldots$ автомата Бюхи B_φ на ω-слове β, где $z_i = (K_i, C_i, S_i, R_i, b_i, \theta_i)$, $i \ge 0$, индукцией по структуре подформул $\psi \in Closure(\varphi)$ можно показать, что для любого $i \ge 0$ имеет место соотношение $\psi \in K_i \iff tr_\beta|^i \models \psi$. Кроме того, для каждой трассы tr, на которой выполняется формула φ, можно показать, что для ω-слова β, порождающего эту трассу, автомат Бюхи B_φ имеет допускающее вычисление ρ, в котором для каждого состояния $z_i = (K_i, C_i, S_i, R_i, b_i, \theta_i)$ верно $K_i = \{\psi : \psi \in Closure(\varphi), tr|^i \models \psi\}$. Поскольку для каждого начального состояния $z_0 = (K_0, C_0, S_0, R_0, b_0, \theta_0)$ верно включение $\varphi \in K_0$, отсюда следует утверждение теоремы.

Вместо того, чтобы приводить технические детали этого обоснования, мы ограничимся объяснением той роли, которую играют в функционировании автомата B_φ все компоненты его состояний $z = (K, C, S, R, b, \theta)$.

Ясно, что K — это множество всех тех формул $\psi \in Closure(\varphi)$, совместная выполнимость которых должна быть подтверждена на какой-либо трассе, т.е. $tr \models \psi$. Остальные компоненты состояний автомата совместно с определением отношения переходов Δ призваны обеспечить соблюдение этого требования. Выполнимость базовых предикатов контролируют множества R_i, содержащие ровно по одному состоянию для каждого автомата $D_j, 1 \le j \le m$. Определение отношения Δ гарантирует, что в прогоне ρ на ω-слове $\{(c_i, h_i) : i \ge 0\}$ эти множества R_i изменяются в соответствии с действиями h_i. В этом же определении обеспечены условия выполнимости формул вида $\mathbf{X}_c\psi$, входящих в множество K.

Автомат следит за выполнимостью формул вида $\psi \mathbf{U}_{L_j}\chi$ при помощи аппарата фаз $b \in \{0, 1\}$; в каждом допускающем прогоне фазы чередуются. Маркер θ призван отслеживать завершение фаз. Равенство $\theta = \bullet$ означает, что текущая фаза успешно завершена. Информация о том, что в текущем состоянии автомат обеспечивает семантические условия выполнимости формулы $\psi \mathbf{U}_{L_j}\chi$ во время фазы b, хранится в виде пары $(q', b), q' \in Q'_j$, в выполняющем множестве S. Фаза b не будет завершена, пока хотя бы одна пара (q, b) находится в выполняющем множестве. В определении отношения переходов указано условие, при котором из выполняющего множества S удаляется пара (q, b); это происходит либо в том случае, когда поступивший поток сигналов не принадлежит языку L_j, который используется в качестве параметра темпорального оператора $\mathbf{U}_{L_j}$ (эта альтернатива проявляется в отношении $q \notin L_j$), либо когда в текущем состоянии z возникает необходимость обеспечить выполнимость граничного условия χ (эта альтернатива проявляется в отношении $\chi \in K$). Во время фазы b могут появляться новые формулы с оператором $\mathbf{U}$, выполнение которых также нужно проверить. Для этих формул информация о необходимости проверки их выполнения записывается в множество S с фазой $1 - b$. При этом проверка выполнимости этих формул начинается сразу же, но на ее завершение отводится время до конца следующей фазы.

Определение множества допускающих состояний гарантирует, что в том случае, если автомат B_φ имеет допускающий прогон на слове β, все фазы бесконечно часто завершаются, и, значит, выполнимость всех формул вида $\psi \mathbf{U}_{L_j}\chi$, появляющихся в множествах K, постоянно подтверждается на трассе tr_β. Так как для допускающего прогона верно включение $\varphi \in K_0$, описанный выше принцип функционирования автомата B_φ гарантирует, что в случае непустоты ω-языка $Lang(B_\varphi)$ существует такое ω-слово β, для которого $tr_\beta \models \varphi$. □

Как видно из приведенного описания автомата Бюхи B_φ, число его состояний оценивается сверху величиной $2^{poly(|\varphi|)}$. Ранее, в статье [14], теоретико-автоматный подход был применен для демонстрации того, что задача верификации моделей разрешима для логики *Reg-LTL*. Однако метод, который был предложен в этой статье, приводил к построению автоматов Бюхи, размер которых оценивался двойной экспонентой от размера проверяемой формулы. То устройство автомата B_φ, которое описано в этом разделе, оказывается существенно более компактным за счет использования в структуре состояний (K, C, S, R, b, θ) автоматов компонентов фазы и маркера. С их помощью оказалось возможным хранить подмножества C, S, R множеств состояний V и W автоматов-распознавателей параметров темпоральных операторов и базовых предикатов формулы φ; в работе [14] для тех же самых целей приходилось использовать семейства подмножеств V и W, что и приводило к «двойному экспоненциальному взрыву».

4. Сложность задачи проверки выполнимости формул *Reg-LTL*

Используя конструкцию автомата B_φ из предыдущего раздела, построим алгоритм проверки выполнимости формул φ логики *Reg-LTL*, которому требуется полиномиальный относительно раз-

мера формул объем памяти. Автомат B_φ, в том виде, в котором он описан в разделе 3, не вполне пригоден для алгоритмических целей, поскольку он работает над бесконечным алфавитом $C \times \mathcal{A}^*$. Но мы покажем, что для проверки пустоты достаточно исследовать лишь конечный фрагмент этого автомата, который, впрочем, может иметь большое число состояний. Чтобы обнаружить в этом фрагменте допускающий прогон автомата, используя при этом ограниченный объем памяти, мы воспользуемся приемом недетерминированного угадывания нужного прогона, а затем применим теорему Савича [27] и сделаем процесс угадывания детерминированным перебором, при котором происходит лишь квадратичное увеличение объема используемой памяти.

Первая трудность, которую нужно преодолеть, состоит в том, что переходы автомата B_φ происходят по элементам бесконечного алфавита $(c, h) \in C \times \mathcal{A}^*$, в котором действия h могут иметь произвольно большую длину. Однако как будет видно из последующей леммы, бесконечный алфавит $C \times \mathcal{A}^*$ можно разбить на конечное число классов так, что для элементов из одного и того же класса отношение переходов автомата определено одинаково. Это обусловлено тем, что, как это видно из определения B_φ, действия h оказывают влияние в переходах из одних состояний в другие только на компоненты предикатных множеств достижимых состояний R, а количество различных множеств такого вида конечно.

Лемма 1. *Пусть z и z' — два состояния автомата Бюхи $B_\varphi = (Q, C \times \mathcal{A}^*, Q_{init}, \Delta, F)$ над алфавитом $C \times \mathcal{A}^*$, и при этом для некоторого элемента (c, h) из множества $C \times \mathcal{A}^*$ существует переход $(z, (c, h), z') \in \Delta$. Тогда существует такое действие $g \in \mathcal{A}^*$ длины, не превосходящей $2^{|\varphi| \log |\varphi|}$, для которого также есть переход $(z, (c, g), z') \in \Delta$.*

Доказательство. Рассмотрим переход $(z, (c, h), z') \in \Delta$ между состояниями $z = (K, C, S, R, b, \theta)$ и $z' = (K', C', S', R', b', \theta')$ в автомате B_φ по элементу $(c, h) \in C \times \mathcal{A}^*$, и пусть $R = \{q''_1, q''_2, \ldots, q''_m\}$ — предикатное достижимое множество, в котором $q''_j, 1 \le j \le m$, — состояния автоматов-распознавателей базовых предикатов формулы φ. Тогда согласно определению отношения переходов Δ автомата Бюхи B_φ имеем $R' = \{\delta''_1(q''_1, h), \delta''_2(q''_2, h), \ldots, \delta''_m(q''_m, h)\}$.

Рассмотрим кратчайшее по длине действие $g = a_1 a_2 \ldots a_N \in \mathcal{A}^*$, для которого равенства $\delta''_j(q''_j, h) = \delta''_j(q''_j, g)$ выполняются для любого $j, 1 \le j \le m$, и покажем, что $N = O(2^{|\varphi| \log |\varphi|})$. Для этого достаточно заметить, что в силу минимальности g среди предикатных множеств состояний

$$R_\ell = \{\delta''_1(q''_1, a_1 a_2 \ldots a_\ell), \delta''_2(q''_2, a_1 a_2 \ldots a_\ell), \ldots, \delta''_m(q''_m, a_1 a_2 \ldots a_\ell)\},$$

где $0 \le \ell \le N$, не может быть двух одинаковых. Поэтому длина N действия g не превосходит количества всех возможных предикатных множеств, которые образованы состояниями автоматов $D_j, 1 \le j \le m$, распознающих базовые предикаты $P_j, 1 \le j \le m$, формулы φ. Поскольку число состояний $|Q''_j|$ каждого такого автомата-преобразователя $D_j, 1 \le j \le m$, а также и число m самих автоматов, не превосходит размера формулы $|\varphi|$, приходим к выводу о том, что $N \le 2^{|\varphi| \log |\varphi|}$. □

Лемма 2. *Подмножество состояний $R = \{q''_1, q''_2, \ldots, q''_m\}$ автоматов $D_j, 1 \le j \le m$, распознающих базовые предикаты $P_j, 1 \le j \le m$, формулы φ является достижимым множеством в том и только том случае, если существует такое действие $g \in \mathbf{A}^*$ длины, не превосходящей $2^{|\varphi| \log |\varphi|}$, для которого $q''_j = \delta''_j(q''_{init,j}, g)$ для всех $j, 1 \le j \le m$.*

Доказательство. Проводится по той же схеме, что и доказательство леммы 1. □

Опираясь на теорему 1 и леммы 1, 2, приходим к основному утверждению этого раздела.

Теорема 2. *Задача проверки выполнимости формул логики Reg-LTL принадлежит классу сложности* Pspace.

Доказательство. Из теоремы 1 следует, что для проверки выполнимости произвольной формулы φ логики *Reg-LTL* достаточно проверить, допускает ли хотя бы одно ω-слово автомат Бюхи $B_\varphi = (Q, \mathcal{C} \times \mathcal{A}^*, Q_{init}, \Delta, F)$, описанный в разделе 3. Как было отмечено в этом же разделе, число состояний автомата B_φ оценивается сверху величиной $2^{poly(|\varphi|)}$. Для проверки непустоты языка $Lang(B_\varphi)$ достаточно проверить (см. [12, 13]), существуют ли такие начальное состояние z_{init} и допускающее состояние z_{acc} автомата B_φ, что состояние z_{acc} достижимо как из z_{init}, так и из z_{acc}. Для этого, в свою очередь, достаточно проверить, существует ли такой прогон $\rho = z_0, z_1, \ldots, z_{\ell-1}, (z_\ell, \ldots, z_{\ell+r})^\omega$ автомата Бюхи B_φ, в котором z_0— начальное состояние, а z_ℓ — допускающее состояние, и при этом число $\ell + r$ различных состояний в этом прогоне не превосходит общего числа состояний автомата B_φ. При помощи леммы 2 можно проверить, используя лишь полиномиальный объем памяти, является ли наугад выбранное состояние z_0 начальным состоянием автомата B_φ. Недетерминированный алгоритм способен отыскать такой прогон ρ, выбирая наугад состояния автомата B_φ, проверяя выполнимость отношения перехода $(z_i, (c, h), z_{i+1}) \in \Delta$ между всеми парами последовательно выбранных состояний z_i и z_{i+1} хотя бы для какой-нибудь пары $(c, h) \in \mathcal{C} \times \mathcal{A}^*$, и проводя учет числа выбранных для прогона состояний. Лемма 1 показывает, что для проверки существования перехода в автомате B_φ из состояния z_i в состояние z_{i+1} достаточно ограничиться рассмотрением лишь таких пар (c, h), в которых длина N действия $h = a_1 a_2 \ldots a_N$ не превосходит величины $2^{|\varphi| \log |\varphi|}$. В этом случае проверку существования перехода можно также провести при помощи недетерминированной вспомогательной процедуры, которая выбирает наугад элементарные действия и подает их на входы всех тех автоматов-распознавателей $D_1, D_2, \ldots, D_m$, которые специфицируют базовые предикаты $P_1, P_2, \ldots, P_m$ формулы φ. На каждом шаге работы эта процедура проводит учет количества выбранных действий и хранит в памяти набор тех состояний $(q_1'', \ldots, q_m'')$, в которых пребывают автоматы $D_1, D_2, \ldots, D_m$. Очевидно, что такой процедуре требуется объем памяти, полиномиальный относительно размера формулы φ. Но тогда и сам алгоритм проверки существования допускающего прогона автомата Бюхи B_φ также может быть выполнен с использованием полиномиального относительно размера формулы φ объема памяти. Таким образом, показано, что задача проверки выполнимости формул логики *Reg-LTL* принадлежит классу сложности NPspace, который, как показано в статье [27], совпадает с классом сложности Pspace. □

Теорема 3. *Задача проверки выполнимости формул логики Reg-LTL является* Pspace*-трудной.*

Доказательство. Нижнюю оценку сложности задачи проверки выполнимости можно получить путем сведения к ней проблемы пустоты пересечения семейства регулярных языков, распознаваемых заданными детерминированными автоматами Рабин-Скотта. Pspace-полнота этой задачи была установлена в статье [28].

Пусть задано произвольное семейство детерминированных автоматов-распознавателей $M = \{A_1, A_2, \ldots, A_n\}$ и пусть $P_i = Lang(A_i)$ для каждого $i, 1 \le i \le n$. Рассмотрим формулу $\varphi_M = \mathbf{F}_{C^*}(P_1 \wedge P_2 \wedge \cdots \wedge P_n)$. Из определения отношения выполнимости для темпорального оператора **F** видно, что формула φ_M выполнима тогда и только тогда, когда $\bigcap_{i=1}^{n} P_i \neq \emptyset$. □

Следствие 1. *Задача проверки выполнимости формул логики Reg-LTL является* Pspace*-полной.*

5. Сложность задачи верификации моделей для логики *Reg-LTL*

Сложность задачи верификации моделей реагирующих систем относительно *Reg-LTL* спецификаций мы установим с помощью модификации методов, использованных в предыдущем разделе. Эта задача состоит в том, чтобы проверить выполнимость соотношения $\pi \models \varphi$ для произвольного заданного автомата-преобразователя π и произвольной заданной формулы φ логики *Reg-LTL*.

Теорема 4. *Задача верификации автоматов-преобразователей относительно формул логики Reg-LTL принадлежит классу сложности* Pspace.

Доказательство. Эта задача равносильна задаче проверки пустоты пересечения множества всех трасс $Traces(\pi)$, порождаемых автоматом-преобразователем π, и множеством всех трасс $Traces(\neg\varphi)$, на которых выполняется формула $\neg\varphi$. Для такой проверки построим автомат Бюхи $B_{\pi,\varphi}$, который удовлетворяет следующему требованию: $\pi \models \varphi \iff Lang(B_{\pi,\varphi}) = \emptyset$. Как было показано в теореме 1, для каждой формулы $\neg\varphi$ можно построить автомат Бюхи $\hat{B}_{\neg\varphi}$, удовлетворяющий требованию $Lang(\hat{B}_{\neg\varphi}) = \{tr = \langle e, \alpha\rangle : tr \in Traces, tr \notin Traces(\varphi)\}$ и имеющий $2^{poly(|\varphi|)}$ состояний. Отличие автомата $\hat{B}_{\neg\varphi}$ от автоматов Бюхи, описанных в разделе 3, состоит в том, что начальными состояниями в нем являются лишь такие наборы $z = (K, C, S, R_0, b, \theta)$, в которых предикатное достижимое множество $R_0 = \{q''_{init,j} : 1 \le j \le m\}$ состоит из начальных состояний всех автоматов, распознающих базовые предикаты формулы φ. Для каждого автомата-преобразователя $\pi = (Q, \mathcal{C}, \mathcal{A}, q_{init}, T)$ множество трасс этого автомата распознается автоматом Бюхи $B_\pi = (Q, \mathcal{C} \times \mathcal{A}^*, q_{init}, \Delta_T, Q)$, отношение переходов Δ_T которого удовлетворяет условию $(q, (c, h), q') \in \Delta_T \iff (q, c, q', h) \in T$. Тогда, как известно из теории автоматов (см. [12, 13]), множество трасс $Traces(\pi) \cap Traces(\neg\varphi)$ также распознается автоматом Бюхи $B_{\pi,\varphi}$, число состояний которого не превосходит произведения числа состояний автоматов B_π и $B_{\neg\varphi}$. Для проверки пустоты языка $Lang(B_{\pi,\varphi})$ применяется недетерминированный алгоритм поиска допускающего прогона, аналогичный алгоритму, описанному в доказательстве теоремы 1. □

Pspace-трудность задачи верификации моделей реагирующих систем относительно *Reg-LTL* спецификаций можно установить путем сведения к ней все той же проблемы пустоты пересечения семейства регулярных языков, подобно тому, как это было показано в теореме 3.

Следствие 2. *Задача верификации автоматов-преобразователей относительно формул логики Reg-LTL является* Pspace*-полной.*

6. Заключение

В данной работе дано полное решение задач проверки выполнимости формул темпоральной логики *Reg-LTL* и верификации автоматов-преобразователей относительно формул этой логики: 1) построены полиномиальные по объему используемой памяти разрешающие алгоритмы и 2) доказана Pspace-трудность обеих задач.

Можно обозначить также некоторые направления дальнейших исследований, представляющие теоретический и прикладной интерес. Для логики *Reg-LTL* задачи проверки выполнимости и верификации моделей оказались PSpace-полными, а в случае использования в качестве параметров произвольных контекстно-свободных языков — неразрешимыми. Однако остался неисследованным случай, когда в качестве языков для параметризации темпоральных операторов и базовых предикатов выбираются контекстно-свободные языки некоторых специальных семейств, для которых проблема проверки пустоты пересечения имеет эффективное решение. Интерес представляет вопрос о том, где проходит граница между разрешимостью и неразрешимостью и как изменяется при этом сложность указанных задач.

Помимо сложности проверки выполнимости существуют и другие математические вопросы, традиционно исследуемые для языков спецификаций: выразительность, аксиоматизируемость и другие. Выразительность некоторых фрагментов языка *Reg-LTL* изучалась в статье [17], однако в ней формулы этого языка интерпретировались над обычными, а не двойными трассами. В то же время, для двойных трасс существует своя теория регулярных множеств, и вопрос об отношении *Reg-LTL* с этими множествами заслуживает внимания.

References

[1] A. Pnueli, «The temporal logic of programs», in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, 1977, pp. 46–57.

[2] D. Gabbay, A. Pnueli, S. Shelach, and J. Stavi, «The temporal analysis of fairness», in *Proceedings of 7-th ACM Symposium on Principles of Programming Languages*, 1980, pp. 163–173.

[3] M. Ben-Ari, Z. Manna, and A. Pnueli, «The temporal logic of branching time», in *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1981, pp. 164–176.

[4] J. Ouaknine and J. Worrell, «Some recent results in Metric Temporal Logic», in *Proceedings of the 6th International Conference Formal Modeling and Analysis of Timed Systems*, 2008, pp. 1–13.

[5] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*. Springer, 1992.

[6] P. Wolper, «Temporal Logic Can Be More Expressive», *Information and Control*, vol. 56, pp. 72–99, 1983.

[7] T. French, «Quantified propositional temporal logic with repeating states», in *Proceedings of the 10th International Symposium on Temporal Representation and Reasoning*, 2003, pp. 155–165.

[8] J. Henriksen and P. S. Thiagarajan, «Dynamic linear time temporal logic», *Annals of Pure and Applied Logic*, vol. 96, pp. 187–207, 1999.

[9] G. De Giacomo and M. Y. Vardi, «Linear temporal logic and linear dynamic logic on finite traces», in *Proceedings of the 23th International Joint Conference on Artificial Intelligence*, 2013, pp. 854–860.

[10] A. Artale, R. Konchakov, V. Ryzhikov, and M. Zakharyaschev, «Tractable interval temporal propositional and description logics», in *Proceedings of the 29th AAAI Conference on Artificial Intelligence*, 2015, pp. 1417–1423.

[11] S. Merz, M. Wirsing, and J. Zappe, «A Spatio-Temporal Logic for the Specification and Refinement of Mobile Systems», in *Proceedings of the 6th International Conference on Fundamental Approaches to Software Engineering*, 2003, pp. 87–101.

[12] C. Baier and J. Katoen, *Principles of Model Checking*. MIT Press, 2008.

[13] E. M. Clarke, O. Gramberg, and D. A. Peled, *Model Checking*. MIT Press, 1999.

[14] D. G. Kozlova and V. A. Zakharov, «On the model checking of sequential reactive systems», in *Proceedings of the 25th International Workshop on Concurrency, Specification and Programming (CS&P 2016), CEUR Workshop Proceedings*, vol. 1698, 2016, pp. 376–389.

[15] A. R. Gnatenko and V. A. Zakharov, «On the verification of finite transducers over semigroups», *Proceedings of ISP RAS*, vol. 30, pp. 303–324, 2018.

[16] A. Gnatenko and V. Zakharov, «On the Model Checking Problem for Some Extension of CTL*», *Modeling and Analysis of Information Systems*, vol. 27, pp. 428–441, 2020.

[17] A. Gnatenko and V. Zakharov, «On the Expressive Power of Some Extensions of Linear Temporal Logic», *Modeling and Analysis of Information Systems*, vol. 25, pp. 506–524, 2018.

[18] M. Vardi and P. Wolper, «An Automata-Theoretic Approach to Automatic Program Verification», in *Proceedings of the First Symposium on Logic in Computer Science*, 1986, pp. 322–331.

[19] J. A. W. Kamp, *Tense Logic and the Theory of Linear Order, PhD thesis*. University of California, Los Angeles, 1968.

[20] K. Etessami and T. Wilke, «An Until Hierarchy and Other Applications of an Ehrenfeucht-Fraisse Game for Temporal Logic», *Information and Computation*, vol. 160, pp. 88–108, 2000.

[21] M. Leucker and C. Sanchez, «Regular Linear Temporal Logic», in *Proceedings of the 4-th International Colloquium on Theoretical Aspects of Computing*, 2007, pp. 291–305.

[22] O. Kupferman, N. Piterman, and M. Y. Vardi, «Extended Temporal Logic Revisited», in *Proceedings of 12-th International Conference on Concurrency Theory*, 2001, pp. 519–535.

[23] A. R. Gnatenko and V. A. Zakharov, «O slozhnosti verifikatsii avtomatov-preobrazovateley nad kommutativnymi polugruppami», in *Trudy 18 Mezhdunarodnoj konferentsii Problemy teoreticheskoy kibernetiki*, 2017, pp. 68–70.

[24] V. Zakharov and G. Temerbekova, «On the Minimization of Finite State Transducers over Semigroups», *Modeling and Analysis of Information Systems*, vol. 23, pp. 741–753, 2016.

[25] V. Zakharov, «Equivalence checking problem for finite state transducers over semigroups», in *Proceedings of the 6-th International Conference on Algebraic Informatics*, 2015, pp. 208–221.

[26] J. M. Fisher and R. E. Ladner, «Propositional Dynamic Logic of Regular Programs», *Journal of Computer and System Sciences*, vol. 18, pp. 194–211, 1979.

[27] W. J. Savitch, «Relationships between nondeterministic and deterministic tape complexities», *Journal of Computer and System Science*, vol. 4, pp. 177–192, 1970.

[28] D. Kozen, «Lower bounds for natural proof systems», in *Proceedings of the 18th International Symposium on the Foundations of Computer Science*, 1977, pp. 254–266.

MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 28, NO. 4, 2021
journal homepage: www.mais-journal.ru

THEORY OF COMPUTING

Towards Automatic Deductive Verification of C Programs with Sisal Loops Using the C-lightVer System

D. A. Kondratyev[1]

DOI: 10.18255/1818-1015-2021-4-372-393

[1]A.P. Ershov Institute of Informatics Systems, Siberian Branch of the Russian Academy of Sciences, 6, Acad. Lavrentjev pr., Novosibirsk 630090, Russia.

MSC2020: 68Q60
Research article
Full text in Russian

Received November 15, 2021
After revision December 1, 2021
Accepted December 8, 2021

The C-lightVer system is developed in IIS SB RAS for C-program deductive verification. C-kernel is an intermediate verification language in this system. Cloud parallel programming system (CPPS) is also developed in IIS SB RAS. Cloud Sisal is an input language of CPPS. The main feature of CPPS is implicit parallel execution based on automatic parallelization of Cloud Sisal loops. Cloud-Sisal-kernel is an intermediate verification language in the CPPS system. Our goal is automatic parallelization of such a superset of C that allows implementing automatic verification. Our solution is such a superset of C-kernel as C-Sisal-kernel. The first result presented in this paper is an extension of C-kernel by Cloud-Sisal-kernel loops. We have obtained the C-Sisal-kernel language. The second result is an extension of C-kernel axiomatic semantics by inference rule for Cloud-Sisal-kernel loops. The paper also presents our approach to the problem of deductive verification automation in the case of finite iterations over data structures. This kind of loops is referred to as definite iterations. Our solution is a composition of symbolic method of verification of definite iterations, verification condition metageneration and mixed axiomatic semantics method. Symbolic method of verification of definite iterations allows defining inference rules for these loops without invariants. Symbolic replacement of definite iterations by recursive functions is the base of this method. Obtained verification conditions with applications of recursive functions correspond to logical base of ACL2 prover. We use ACL2 system based on computable recursive functions. Verification condition metageneration allows simplifying implementation of new inference rules in a verification system. The use of mixed axiomatic semantics results to simpler verification conditions in some cases.

Keywords: deductive verification; C-lightVer; cloud parallel programming system; symbolic method of verification of definite iterations; loop invariant; ACL2

INFORMATION ABOUT THE AUTHORS

Dmitry A. Kondratyev | orcid.org/0000-0002-9387-6735. E-mail: apple-66@mail.ru
correspondence author | Junior researcher.

Funding: RSF, project No 18-11-00118.

For citation: D. A. Kondratyev, "Towards Automatic Deductive Verification of C Programs with Sisal Loops Using the C-lightVer System", *Modeling and analysis of information systems*, vol. 28, no. 4, pp. 372-393, 2021.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 28, № 4, 2021
сайт журнала: www.mais-journal.ru

THEORY OF COMPUTING

На пути к автоматической дедуктивной верификации С-программ с Sisal-циклами в системе C-lightVer

Д. А. Кондратьев[1]

DOI: 10.18255/1818-1015-2021-4-372-393

[1]Институт систем информатики им. А.П. Ершова Сибирского отделения Российской академии наук, проспект Академика Лаврентьева, д. 6, г. Новосибирск, 630090 Россия.

УДК 004.052.42
Научная статья
Полный текст на русском языке

Получена 15 ноября 2021 г.
После доработки 1 декабря 2021 г.
Принята к публикации 8 декабря 2021 г.

В Институте систем информатики СО РАН разрабатывается система C-lightVer для дедуктивной верификации С-программ. C-kernel является промежуточным языком верификации в данной системе. Система облачного параллельного программирования (CPPS) также разрабатывается в Институте систем информатики СО РАН. Cloud Sisal является входным языком системы CPPS. Главной особенностью системы CPPS является неявное параллельное исполнение, основанное на автоматическом распараллеливании циклов Cloud Sisal. Cloud-Sisal-kernel является промежуточным языком верификации в системе CPPS. Нашей целью является автоматическое распараллеливание такого надмножества языка С, которое позволяет реализовать автоматическую верификацию. Нашим решением является такое надмножество языка C-kernel, как язык C-Sisal-kernel. Первым результатом, представленным в данной статье, является расширение языка C-kernel циклами языка Cloud-Sisal-kernel. В результате был разработан язык C-Sisal-kernel. Вторым результатом, представленным в данной статье, является расширение аксиоматической семантики языка C-kernel правилом вывода для циклов языка Cloud-Sisal-kernel. В данной статье также представлен наш подход к проблеме автоматизации дедуктивной верификации в случае финитных итераций над структурами данных. Такие циклы называются финитными итерациями. Нашим решением является композиция символического метода верификации финитных итераций, метагенерации условий корректности и смешанной аксиоматической семантики. Символический метод верификации финитных итераций позволяет задавать правила вывода для таких циклов без инвариантов. Символическая замена финитных итераций рекурсивными функциями является основой данного метода. Полученные условия корректности с применениями рекурсивных функций соответствуют логической основе системы доказательства ACL2. Мы используем систему ACL2, основанную на вычислимых рекурсивных функциях. Метагенерация условий корректности позволяет упростить реализацию новых правил вывода в системе верификации. Использование смешанной аксиоматической семантики приводит в некоторых случаях к более простым условиям корректности.

Ключевые слова: дедуктивная верификация; C-lightVer; система облачного параллельного программирования; символический метод верификации финитных итераций; инвариант цикла; ACL2

ИНФОРМАЦИЯ ОБ АВТОРАХ

Дмитрий Александрович Кондратьев
автор для корреспонденции

orcid.org/0000-0002-9387-6735. E-mail: apple-66@mail.ru
Младший научный сотрудник.

Финансирование: РНФ, проект № 18-11-00118.

Для цитирования: D. A. Kondratyev, “Towards Automatic Deductive Verification of C Programs with Sisal Loops Using the C-lightVer System”, *Modeling and analysis of information systems*, vol. 28, no. 4, pp. 372-393, 2021.

Введение

Система C-lightVer [1—5] разрабатывается в Институте систем информатики СО РАН для верификации С-программ. Данная система основана на методе дедуктивной верификации [6—9]. Представительное подмножество языка С (C-light) [10] является входным языком системы C-lightVer. Модель памяти языка C-light основана на отображениях MeM и MD. MeM является отображением из имен объектов в их адреса, MD является отображением из адресов объектов в их значения. Операция upd позволяет создавать новое отображение MD, когда изменяется состояние памяти. C-kernel [11] является промежуточным языком верификации в системе C-lightVer. Трансляция из C-light в C-kernel основана на наборе правил [1]. Главной целью данной трансляции является локализация побочных эффектов.

Символический метод верификации финитных итераций [12] применяется к циклам специального вида (финитным итерациям). Тело финитной итерации исполняется один раз для каждого элемента структуры данных конечной размерности. Основой этого метода является символическая замена финитных итераций специальными рекурсивными функциями. Данный метод позволяет избежать задания инвариантов в случае финитных итераций. Система C-lightVer применяет этот метод к циклам языка С, соответствующим финитным итерациям [3—5].

Метагенерация условий корректности (УК) [2, 5, 13] позволяет использовать правила вывода УК как входные данные системы верификации. Входными данными метагенератора являются правила вывода УК и аннотированная программа. Заключения правил вывода представляют шаблоны, сопоставляемые с кодом на языке С. Поэтому, был разработан язык шаблонов [2] для задания правил вывода.

Метод смешанной аксиоматической семантики [1] позволяет использовать специальные версии правил вывода для определенных программных конструкций.

В системе C-lightVer для доказательства УК используется ACL2 [14]. Applicative Common Lisp (ACL) является входным языком системы ACL2. Система C-lightVer генерирует УК, записанные на языке ACL. Система ACL2 основана на вычислимых рекурсивных функциях [14]. Полученные УК с применениями рекурсивных функций соответствуют логической основе системы ACL2.

Важным этапом автоматизации верификации является этап автоматизации доказательства УК. Доказательство УК, содержащих операцию замены для финитной итерации, основано на индукции по длине структуры, над которой осуществляется данная итерация. При этом, использования классической индукции в системе ACL2 не достаточно для доказательства УК, содержащих операцию замены для финитной итерации, в случае, если итерация изменяет элементы структуры или содержит инструкции, подобные `break`. В системе C-lightVer была реализована стратегия автоматизации доказательства УК программ, постусловием которых является разбор случаев [4].

Система облачного параллельного программирования (CPPS) также разрабатывается в Институте систем информатики СО РАН [15]. Cloud Sisal [15—18] является входным языком системы CPPS. Данный язык является новой версией языка Sisal [19, 20]. Cloud Sisal является функциональным языком программирования, основанным на циклических выражениях. Главной особенностью системы CPPS является неявное параллельное исполнение, основанное на автоматическом распараллеливании циклов Cloud Sisal [15, 16, 21]. Cloud-Sisal-kernel [22] был разработан как промежуточный язык верификации для дедуктивной верификации Cloud Sisal программ. Автоматизация верификации циклов Cloud-Sisal-kernel является результатом использования символического метода верификации финитных итераций [12].

Нашей целью является автоматическое распараллеливание такого надмножества С, которое позволяет реализовать автоматическую верификацию. Нашим решением является такое надмножество C-kernel, как C-Sisal-kernel. Это расширение C-kernel циклами языка Cloud-Sisal-kernel. Реализация неявного параллельного исполнения программ на языке C-Sisal-kernel основано на опыте [15, 16, 21]

распараллеливания программ на языке Cloud Sisal. Отметим, что это задача другой группы исследователей [15—17, 21] из Института систем информатики СО РАН. Эта группа разрабатывает CPPS. Автоматическое распараллеливание C-Sisal-kernel программ является частью проекта CPPS.

Нашей задачей является расширение аксиоматической семантики языка C-kernel для реализации дедуктивной верификации C-программ с циклами языка Sisal. Дедуктивная верификация C-Sisal-kernel программ является частью системы C-lightVer. Мы применяем такие подходы как метод метагенерации УК [2, 5] и метод смешанной аксиоматической семантики [1] для решения этой задачи.

Первым результатом, представленным в данной статье, является расширения языка C-kernel циклами Cloud-Sisal-kernel. Полученный язык называется C-Sisal-kernel. Вторым результатом, представленным в данной статье, является расширение аксиоматической семантики C-kernel правилом вывода для циклов языка Cloud-Sisal-kernel. Эта семантика позволяет автоматизировать дедуктивную верификацию благодаря отсутствию инвариантов циклов. Данная статья также представляет наш комплексный подход к проблеме автоматизации дедуктивной верификации в случае финитных итераций над структурами данных. Нашим решением является композиция символического метода верификации финитных итераций [4, 5, 12], метода метагенерации УК [2, 5, 13] и метода смешанной аксиоматической семантики [1]. Мы полагаем, что наш подход может быть применен к различным языкам программирования, которые позволяют записывать программы с циклами, соответствующими финитным итерациям.

Данная статья имеет следующую структуру: предварительные сведения описаны в главах 1, 2 и 3, новые результаты описаны в главе 4 и эксперименты, демонстрирующие применение новых результатов, описаны в главе 5.

Обзор родственных работ. Во-первых, специальные виды циклов являются важной концепцией в различных парадигмах программирования. Например, функции *map* и *reduce* реализованы во многих функциональных языках программирования. Библиотека MapReduce [23] расширяет императивные и объектно-ориентированные языки программирования конструкциями, соответствующими функциональной парадигме программирования. Другим примером является специальная конструкция *loop* в языке Applicative Common Lisp [24]. Операционная семантика для расширения языка C-like конструкциями OpenMP была предложена в работе [25]. Такая особенность Sisal, как конструкции, подобные `break`, является преимуществом Sisal-циклов относительно рассмотренных видов итераций. Актуальность автоматизации дедуктивной верификации в случае циклов с инструкциями `break` продемонстрирована примерами из набора задач по верификации [26]. Мы полагаем, что символический метод верификации финитных итераций может быть применен к различным итерациям, упомянутым в этом разделе, чтобы избежать задания их инвариантов.

Во-вторых, расширение языков программирования конструкциями функциональной парадигмы программирования является актуальной задачей. Такие конструкции могут увеличить эффективность исполнения и упростить реализацию некоторых алгоритмов. Например, семантика конструкций функциональной парадигмы программирования в новейших стандартах C++ и Java описана в работах [27, 28].

В третьих, расширения языка C специальными видами циклов является актуальной задачей. Новейший стандарт C++20 [29] вводит диапазоны (ranges) и итерации над ними. Более того, диапазоны (ranges) напоминают триплеты языка Cloud Sisal. Но циклы *for* над этими диапазонами (ranges) не позволяют, в отличие от Sisal-циклов, использовать редукции. Отметим, что современный стандарт C11 [30] был расширен новыми конструкциями с формальной семантикой [31], но эти конструкции не являются видами итераций. Система RefinedC [32] использует эту семантику, реализованную в системе доказательства Coq, но пользователю RefinedC необходимо задавать инварианты циклов.

Наконец, верификация циклов, основанная на их замене рекурсивными функциями, представляет большой интерес для нас. Это актуальный подход, позволяющий избежать задания инвариантов циклов. Работы [33, 34] основаны на этом подходе. Другим примером данного подхода является символический метод верификации финитных итераций [12]. Этот метод был применен к специальным видам циклов for из языка C [3—5].

Главным альтернативным подходом является генерация инвариантов циклов. Например, автоматическая генерация инвариантов для P-разрешимых циклов была предложена в работе [35]. Правые части инструкций присваивания в теле P-разрешимого цикла должны иметь вид присваивания полиномиального выражения. Но некоторые Sisal-циклы не являются P-разрешимыми из-за неполиномиальных редукций и конструкций, подобных `break`. Другим примером является метод разностных инвариантов, реализованный в системе Diffy [36]. Но этот метод, в отличие от символического метода верификации финитных итераций, не подходит для циклов с инструкциями `break`. Другим подходом является задание предусловий и постусловий вместо инвариантов циклов [37]. В работе [38] также описано использование спецификаций специального вида вместо инвариантов циклов. Этот метод реализован в системе Frama-C [39]. Но эти методы [37, 38] основаны на задании спецификаций пользователем. Также есть другие альтернативные походы. Одним из них является натуральная семантика. Такая семантика была предложена для Sisal-циклов в работе [40]. Но предложенная натуральная семантика для языка Sisal больше подходит для разработки компиляторов, чем для дедуктивной верификации. Другой альтернативой является использование трансляционной семантики. Например, работа [41] основана на трансляции циклов языка Cloud Sisal в циклы языка C. Трансформационная семантика была предложена для MapReduce [42]. Однако мы полагаем, что аксиоматическая семантика является лучшим выбором для дедуктивной верификации, поэтому эти альтернативы не используются в представленном исследовании.

1. Языки, методы и модули, используемые в системе C-lightVer

1.1. Язык C-light и язык C-kernel

Входным языком системы C-lightVer является язык C-light [10]. Этот язык является представительным подмножеством языка C. Для языка C-light была разработана операционная семантика. Модель памяти, основанная на отображениях MeM и MD, используется в этой семантике. MeM является отображением из имен объектов в их адреса, MD является отображением из адресов объектов в их значения.

Операция upd позволяет создавать новое отображение MD, когда изменяется состояние памяти. Определим значение выражения $upd(MD,\ addr,\ val)$, где MD является отображением $address \longrightarrow value$, $addr$ является адресом и val является значением. Если MD содержит пару $(adr\ val')$ (где val' является некоторым значением), то отображение $upd(MD,\ addr,\ val)$ отличается от MD заменой пары $(adr\ val')$ на пару $(adr\ val)$. Если $addr$ не принадлежит области определения MD, тогда отображение $upd(MD,\ addr,\ val)$ отличается от MD добавлением пары $(adr\ val)$.

Язык C-kernel является очень ограниченным подмножеством языка C-light [11]. Для языка C-kernel определена аксиоматическая семантика. Это позволяет верифицировать C-kernel программы. Трансляция из C-light в C-kernel является первой стадией исполнения системы C-lightVer. Эта трансляция основана на наборе правил. Эти правила определяют трансляцию различных конструкций C-light в эквивалентные конструкции C-kernel.

Главной целью этой трансляции является локализация побочных эффектов. Поэтому, правила трансляции выносят сложные подвыражения (не переменные и константы) из выражений и операторов, используя задание вспомогательных переменных со значениями этих подвыражений. В итоге, все инструкции и выражения транслируются в форму, где только переменные и константы

являются их аргументами. В качестве примера рассмотрим правило трансляции *Ops*: если e_i не является переменной или константой, $e_{i+1}, \ldots, e_n$ – переменные или константы, f – функция или +, −, *, /, <, >, <=, =>, ! =, ==, тогда $f(e_1, \ldots, e_{i-1}, e_i, e_{i+1}, \ldots, e_n)$ транслируется в $(x = e_i, f(e_1, \ldots, e_{i-1}, x, e_{i+1}, \ldots, e_n))$, где x является новой переменной с тем же типом, что и e_i.

Так как язык C-kernel является подмножеством языка C-light, то его операционная семантика является той же самой, что и семантика языка C-light. Это позволило доказать, что правила трансляции сохраняют эквивалентность. УК полученной C-kernel программы генерируются на втором этапе исполнения системы C-lightVer.

1.2. Метагенератор условий корректности

Метагенератор УК является важным модулем системы C-lightVer. Входными данными метагенератора являются правила вывода УК и аннотированная программа. Заключения правил вывода представляют шаблоны, сопоставляемые с кодом на языке C. Поэтому, был разработан язык шаблонов [2] для задания правил вывода. Этот язык основан на логике первого порядка и грамматике языка C.

Правила вывода могут содержать нетерминальные символы, такие как неинтерпретированные предикатные символы или “фрагментные переменные”, обозначающие фрагменты кода [13]. Нетерминальный символ задает метаданные об определенной конструкции, соответствующей этому символу. Следовательно, в языке шаблонов есть специальные конструкции для представления этих метаданных. Например, конструкция `any_code(S)` может быть сопоставлена с любой последовательностью (включая пустую) инструкций на языке C. Специальный алгоритм [2] для сопоставления этих конструкций с аннотированным исходным кодом был реализован в метагенераторе. УК для аннотированной программы генерируются метагенератором с помощью использования входных правил вывода.

1.3. Символический метод верификации финитных итераций

Пусть $memb(S)$ обозначает мультимножество элементов структуры S и $empty(S) = true$ тогда и только тогда, когда $|memb(S)| = 0$. Определим

1. $choo(S)$ возвращает некоторый элемент из $memb(S)$, если $\neg empty(S)$;
2. $rest(S) = S'$, где $memb(S') = memb(S) \setminus \{choo(S)\}$, если $\neg empty(S)$.

Финитная итерация соответствует виду: **for x in S do v** := **body**(**v**, **x**), где S является структурой данных, x является переменной типа “элемент S”, v является вектором переменных цикла без x, $body$ представляет тело цикла, которое не изменяет x и завершается для каждого $x \in S$. Пусть v_0 обозначает начальные значения переменных из v. Определим операцию замены $rep(v, S, body)$ для цикла:

1. Если $empty(S)$, тогда $rep(v_0, S, body) = v_0$;
2. Если $\neg empty(S)$, тогда $rep(v_0, S, body) = body(rep(v_0, rest(S), body), choo(S))$.

В случае наличия инструкции `break` в финитной итерации предложено следующее решение: когда происходит выход из цикла из-за исполнения этой инструкции, мы полагаем, что итерации цикла продолжаются, но значение v остается неизменным. Полученная операция замены возвращает не только вектор v, но также структуру с булевским полем (*loop-break*). Значением по умолчанию поля *loop-break* является $false$. Полученная функция *rep* содержит условие исполнения этой инструкции `break`. Если данная инструкция `break` исполняется, тогда эта функция *rep* возвращает структуру со значением $true$ поля *loop-break*. Также эта функция *rep* проверяет значение поля *loop-break* из результата рекурсивного вызова. Если значением является $true$, тогда эта функция *rep* возвращает тот же результат, что и рекурсивный вызов. Значения переменных цикла не изменяются после исполнения инструкции `break` в этой реализации.

Этот метод позволяет избежать задания инвариантов в случае циклов, соответствующих финитным итерациям [12]. Например, циклы языка Cloud-Sisal-kernel соответствуют финитным итерациям [22]. Другим примером является специальный класс циклов for в языке C. Эти циклы могут содержать инструкции `break`. Этот класс называется допустимыми циклами [4]. Алгоритм проверки, принадлежит ли определенный цикл этому классу, реализован в системе C-lightVer. Но, конечно, этот класс не покрывает все возможные финитные итерации в языке C. Генерация операции замены для допустимых циклов языка C-kernel основана на трансляторе $c2acl2$ [5] из C-kernel в ACL [14].

1.4. Метод смешанной аксиоматической семантики

Метод смешанной аксиоматической семантики [1] позволяет использовать специальные версии правил вывода для определенных программных конструкций. Отметим, что в C-программах есть переменные, которые используются без применений операций взятия адреса и разыменования указателя. Количество таких переменных в C-программах может быть значительным. Для таких переменных может быть использована более простая модель памяти, чем модель, основанная на MeM и MD. Следовательно, более простые правила вывода могут быть применены к инструкциям над такими переменными. Это позволяет упростить УК.

1.5. Стратегии автоматизации доказательства УК

Рассмотрим стратегию автоматизации доказательства УК программ, постусловием которых является разбор случаев [4]. Эта стратегия применяется для содержащих финитные итерации аннотированных программ, постусловие которых имеет вид конъюнкции импликаций. Для каждой такой импликации полезно рассмотреть, эквивалентен ли описанный ее посылкой случай исполнению операции `break`.

Рассматриваемая стратегия генерирует леммы, где в посылку УК добавляется специальный конъюнкт. Такой конъюнкт генерируется для каждой импликации постусловия в двух видах: для значения $rep(\ldots).loop\text{-}break$ и для отрицания значения $rep(\ldots).loop\text{-}break$. Данный конъюнкт является проверкой эквивалентности посылки импликации постусловия и значения/отрицания значения $rep(\ldots).loop\text{-}break$. Если система ACL2 автоматически доказывает какую-либо из сгенерированных лемм, то эта лемма добавляется в теорию предметной области. Система ACL2 может использовать леммы, добавленные в теорию предметной области, для доказательства других теорем и УК.

Так как значение поля $rep(\ldots).loop\text{-}break$ является информацией о срабатывании инструкции `break`, то данная стратегия разработана для такой аннотированной программы, где в постусловии описаны случаи срабатывания/отсутствия срабатывания инструкции `break`.

2. Краткое описание языка Cloud-Sisal-kernel

Язык Cloud Sisal основан на специальных видах циклов [15, 17, 18]. Эти виды циклов позволяют упростить распараллеливание и векторизацию операций над массивами [15, 16, 21]. Язык Cloud-Sisal-kernel был разработан как промежуточный язык верификации для дедуктивной верификации Cloud Sisal программ [22]. Для языка Cloud-Sisal-kernel была предложена аксиоматическая семантика. Этот язык включает базовые конструкции Cloud Sisal, такие как триплеты, циклические выражения, циклические выражения с условиями завершения и выражения замещения элементов массивов.

Триплетом является структура вида [lower boundary .. upper boundary .. step]. Она задает арифметическую прогрессию между заданными границами с фиксированным шагом. Диапазоном является структура, основанная на декартовом произведении триплетов.

2.1. Циклические выражения

Рассмотрим циклическое выражение, управляемое диапазоном:

$for\ var_1\ in\ triplet_1\ cross\ \ldots\ var_n\ in\ triplet_n\ do\ returns\ reduction\ expr\ end\ for$

где var_i и $triplet_j$ являются переменными и триплетами соответственно, $expr$ является редуцируемым выражением, которое может зависеть от этих переменных, $reduction$ является редукцией, декартово произведение обозначается ключевым словом `cross`. Этот цикл совершает итерации над декартовым произведением триплетов. Значением цикла после определенной итерации является значение редукции, примененное к значению $expr$ на этой итерации и к значению цикла после предыдущей итерации. Рассмотрим главные редукции:

- $array\ (value)\ of$ возвращает массив (последнее значение) редуцируемого выражения;
- $sum\ (product)\ of$ вычисляет сумму (произведение) значений редуцируемых выражений;
- $greatest\ (least)\ of$ вычисляет наибольшее (наименьшее) значение редуцируемого выражения.

Обозначим как $loop\_e$ такое циклическое выражение с такими подвыражениями.

Рассмотрим иллюстративный пример Cloud Sisal программы:

```
function f (a: array of (array of integer), n,m: integer returns integer)
  for i in 0..n-1..1 cross j in 0..m-1..1 do
  returns sum of if (a[i, j] > 0) then a[i, j]*a[i, j]*a[i, j] else 0
                 end if end for end function
```

где

- `0..n-1..1` является конечной арифметической прогрессией, где `0` является начальным значением, `1` является шагом прогрессии и `n-1` является верхней границей;
- `0..m-1..1` является подобной последовательностью, где `m-1` является верхней границей;
- `sum of` является редукцией, которая вычисляет сумму редуцируемых выражений;
- `if (a[i,j] > 0) then a[i,j]*a[i,j]*a[i,j] else 0 end if` является редуцируемым выражением, которое вычисляет значение куба положительного элемента матрицы.

Эта функция вычисляет сумму значений кубов положительных элементов матрицы.

2.2. Циклические выражения с условиями завершения

Рассмотрим циклическое выражение с конструкцией $while$:

$for\ var_1\ in\ triplet_1\ cross\ \ldots\ var_n\ in\ triplet_n$
$while\ condition\ do\ returns\ reduction\ expr\ end\ for$

где конструкция $while$ соответствует следующей инструкции, записанной на языке программирования C: if (!$condition$) {**break**; }. Эта конструкция не задает вложенный цикл.

2.3. Выражения замещения элементов массивов

Выражение замещения элементов массива имеет следующий вид:

$$source\_array[var_1\ \texttt{in}\ triplet_1\ \texttt{cross}\ \ldots\ \texttt{cross}\ var_n\ \texttt{in}\ triplet_n\ :=\ expr]$$

где $source\_array$ является именем исходного массива, $var_1, \ldots, var_n$ являются переменными, триплетами являются $triplet_1, \ldots, triplet_n$, $expr$ является замещающим выражением (возможно зависящим от $v_{1..n}$). Выражение замещение элементов массива выполняет неявные итерации над диапазоном, получаемым в результате декартового произведения триплетов. Результатом этого выражения является новый массив, который совпадает с $source\_array$, за исключением элементов с индексами из диапазона, чьи значения заменяются на значение замещающего выражения.

Обозначим как $array\_rep\_expr$ такое выражение замещения элементов массива с такими подвыражениями.

3. Семантика языка Cloud-Sisal-kernel

Входной язык системы ACL2 [14] является аппликативным и строго функциональным диалектом языка Common Lisp. Так как Cloud Sisal также является функциональным языком, то в работе [22] предложена трансляция выражений Cloud Sisal в выражения ACL2. Функция *sisal2acl2* реализует эту трансляцию. Определение этой функции описано в работе [22]. Рассмотрим краткое описание этой трансляции.

3.1. Трансляция базовых конструкций языка Cloud-Sisal-kernel на язык ACL

Функция *sisal2acl2* конвертирует массивы языка Cloud-Sisal-kernel в списки ACL2. Две операции используются для обработки индексируемых последовательностей. Рассмотрим функции *nth* и *update-nth*, которые реализуют эти операции. Если i является индексом и l является списком, тогда (*nth i l*) является значением i-го элемента из списка l. Если выражение *expr* является выражением ACL2, тогда (*update-nth i expr l*) является новым списком, который совпадает со списком l за исключением i-го элемента, чьим значением является *expr*.

Функция *sisal2acl2* транслирует триплеты Cloud-Sisal-kernel в применение функции *triplet*. Эта функция принимает в качестве аргументов нижнюю границу *low*, верхнюю границу *high* и шаг *step* и возвращает список значений арифметической прогрессии, заданной триплетом.

Функция *sisal2acl2* использует транслятор *range2acl2*. Он транслирует декартово произведение триплетов в применение функции *cartesian_product* в общем случае. Функция *cartesian_product* возвращает список кортежей. Отметим, что также используется функция *partition* для моделирования заголовка цикла в случае одиночного триплета. Функция *range2acl2* транслирует одиночный триплет в применение функции *partition*.

Мы используем специальные конструкции, предоставляемые библиотекой *fty* в системе ACL2, для задания новых типов. Если *e* – имя нового типа, тогда генерируется макрос *make-e* с помощью библиотеки *fty*. Этот макрос является конструктором для типа *e*. Мы используем этот подход для создания структуры *loop-expr* в главе 4.

Также эти конструкции позволяют задать специальные структуры для моделирования контекста цикла. Тип структуры *environment_id* генерируется для моделирования контекста. Поля этой структуры соответствуют переменным контекста. Мы генерируем функцию *create_environment_id* для упрощения создания объектов типа *environment_id*. Функции *context_variables*, *context2vector* и *context2string* позволяют получать информацию о переменных контекста от компилятора CPPS [15, 16, 21] и конвертировать список таких переменных в строковое представление.

3.2. Трансляция циклов Cloud-Sisal-kernel на язык ACL

Аксиоматическая семантика циклических выражений Cloud-Sisal-kernel основана на символическом методе верификации финитных итераций [12]. Следовательно, функция *sisal2acl2* транслирует цикл в применение функции *rep_id*, где *id* является уникальным идентификатором.

Отметим, что в работе [22] не была задана аксиоматическая семантика для циклических выражений с условиями завершения.

Циклы языка Cloud-Sisal-kernel задают итерации над кортежами значений переменных из заголовка цикла. Эти кортежи получаются декартовым произведением триплетов из заголовка циклов. Следовательно, трансляция определена с помощью следующего применения функции *range2acl2*:

$$
\begin{aligned}
&sisal2acl2(loop\_e) \;=\; (rep\_id \\
&(reverse\ range2acl2(triplet_1\ cross\ \ldots\ triplet_{n-1}\ cross\ triplet_n)) \\
&\quad (create\_environment\_id\ context2string\ context2vector(\\
&\qquad context\_variables(expr) \setminus \{var_1,\ var_2,\ \ldots\ var_{n-1},\ var_n\}))),
\end{aligned}
$$

где *id* является уникальным идентификатором. Применение функции *reverse* гарантирует корректный порядок применения редукции. Операция разности множеств удаляет из контекста переменные диапазона. Строковые представления переменных становятся аргументами функции *create_environment_id*.

Генерация определения функции *rep_id* также описана в работе [22]. Функция *sisal2acl2* реализует эту генерацию как трансляцию тела цикла в применение конструкции $b*$ языка ACL2. Рассмотрим общую форму этой конструкции: $(b*\ (\ldots(var\ expr)\ldots)\ result\_expr)$, где выражение $(var\ expr)$ обозначает связывание переменной *var* со значением выражения *expr*, которое может зависеть от ранее связанных переменных. Значением блока $b*$ является значение выражения *result_expr*, которое также может зависеть от связываемых переменных. Отметим, что значение переменных из заголовка цикла задано в том кортеже из декартового произведения, над которым исполняется текущая итерация. Рассмотренные конструкции позволяют создать в блоке $b*$ переменные, соответствующие переменным из заголовка цикла, и связать их значения с соответствующими значениями из кортежа, над которым исполняется текущая итерация. Исполнение конструкций, завершающих исполнение, моделируется истинностью условия *when*, которое может быть использовано как условие связываний.

Отметим, что переменные заголовка цикла языка Cloud-Sisal-kernel не зависят друг от друга. Поэтому, мы не используем такую возможность блока $b*$, как зависимость переменных от ранее определенных переменных. Мы используем блок $b*$, чтобы задать переменные, соответствующие переменным контекста, и переменные, соответствующие переменным заголовка цикла. Эти переменные не зависят друг от друга. Так как в языке системы ACL2 отсутствуют замыкания, то задание в блоке $b*$ таких переменных позволяет моделировать возможности замыкания. Введение таких переменных позволяет записать внутри блока $b*$ редуцируемое выражение, зависящее от переменных контекста цикла и от переменных заголовка цикла.

Функция *sisal2acl2* использует транслятор *reduct2acl2* для генерации применения редуцирующей функции как *result_expr* конструкции $b*$. Функция *reduct2acl2* реализует трансляцию редукций. Эта функция принимает три аргумента: имя редукции, редуцируемое выражение на текущей итерации $expr_1$ и значение цикла после предыдущей итерации ($expr_2$). Мы также используем вспомогательную функцию *reduction_init*, которая принимает в качестве аргумента имя редукции и возвращает значение редукции по умолчанию. Эти функции позволяют реализовать редукцию в теле функции *rep*. Рассмотрим список некоторых редуцирующих функций, полученных в результате применения транслятора *reduct2acl2*:

- $reduct2acl2(array\ of, expr_1, expr_2) = (cons\ expr_1\ expr_2)$
- $reduct2acl2(sum\ of, expr_1, expr_2) = (+\ expr_1\ expr_2)$
- $reduct2acl2(product\ of, expr_1, expr_2) = (*\ expr_1\ expr_2)$

Редуцирующая функция в теле функции *rep_id* применяется к редуцируемому выражению и результату рекурсивного вызова функции *rep_id*.

3.3. Трансляция выражений замещения элементов массивов на язык ACL

Семантика выражений замещения элементов массивов также основана на символическом методе верификации финитных итераций. Эти выражения транслируются в применение рекурсивных функций *update_elements_id*, где *id* является уникальным идентификатором. Рассмотрим трансляцию этих выражений с помощью функции *sisal2acl2*:

$sisal2acl2(array\_rep\_expr) =$
$(update\_elements\_id$
$(reverse\ range2acl2(triplet_1\ cross\ \ldots\ cross\ \ldots\ triplet_{n-1}\ cross\ triplet_n))$
$(create\_environment\_id$
$context2string($
$context2vector($
$context\_variables(expr) \setminus \{var_1,\ var_2,\ \ldots\ var_{n-1},\ var_n\})))$
$source\_array),$

Отметим, что применение функции $update\_elements\_id$ похоже на применение функции rep_id в случае циклических выражений. Определение функции $update\_elements\_id$ также схоже с определением функции rep_id за исключением результирующего выражения в блоке $b*$ с применениями выражений $update\text{-}nth$ вместо редуцирующей функции.

3.4. Аксиоматическая семантика языка Cloud-Sisal-kernel

Для задания аксиоматической семантики языка Cloud-Sisal-kernel используется метод слабейшего предусловия [7]. Данный метод, как и символический метод верификации финитных итераций, основан на замене переменных их значениями, вычисленными символически. Если $wp(S, Q)$ является слабейшим предусловием программы S с постусловием Q и P является формулой, тогда тройка $\{wp(S, Q)\}\ S\ \{Q\}$ частично корректна и из истинности формулы $P \longrightarrow wp(S, Q)$ следует частичная корректность тройки $\{P\}\ S\ \{Q\}$. Данное свойство позволяет использовать метод слабейшего предусловия для генерации УК.

Аксиоматическая семантика выражений языка Sisal описана в работе [22]. Для задания аксиоматической семантики язык спецификаций был расширен термом $result$. Данный терм можно использовать в постусловии Sisal-выражения для обозначения значения этого выражения. Пусть $R(y \longleftarrow expr)$ обозначает одновременную замену в R всех свободных вхождений переменной y на на $expr$. Если $expr$ является выражением языка Sisal, тогда

$$\mathbf{wp(expr, Q) = Q(result \longleftarrow sisal2acl2(expr))}$$

Так как слабейшее предусловие Sisal-выражений основано на применении транслятора $sisal2acl2$, то аксиоматическая семантика языка Cloud-Sisal-kernel основана на трансляционной семантике. Отметим, что в работе [22] не была задана аксиоматическая семантика для циклических выражений с условиями завершения.

4. Расширение языка C-kernel циклическими выражениями языка Cloud-Sisal-kernel

4.1. Синтаксис триплетов, циклических выражений, циклических выражений с условиями завершения и выражений замещения элементов массивов в языках C-light и C-kernel

Язык C-light был расширен таким новым видом выражений, как триплет. Триплет имеет следующий синтаксис: lower boundary .. upper boundary .. step, где $lower\ boundary$, $upper\ boundary$ и $step$ – целочисленные выражения. Рассмотрим разницу между операционной семантикой этих выражений в языке C-light и семантикой триплетов Cloud-Sisal-kernel.

Триплеты, добавленные в язык C-light, возвращают структуру типа $tripl$:

typedef struct tripl{int scalar; int * vector; }tripl;

Так как триплет моделирует последовательность чисел, то поле *vector* этой структуры является указателем на динамический массив, заполненный этой последовательностью (выделенная память должна быть освобождена программистом после использования), поле *scalar* хранит длину полученного массива. Пустой триплет соответствует структуре, где поле *scalar* равно 0 и поле *vector* равно *NULL*.

Также язык C-light был расширен циклическими выражениями с синтаксисом, сходным с синтаксисом *loop_e*. Опишем разницу между семантикой этих выражений в языке C-light и семантикой циклов Cloud-Sisal-kernel. Циклические выражения, добавленные в язык C-light, возвращают структуру типа *loop_expr*:

typedef struct loop_expr{**int scalar**; **int** * **vector**; }**loop_expr**;

Если циклическое выражение возвращает целочисленное значение, тогда результат сохраняется в поле *scalar* соответствующей структуры и поле *vector* хранит *NULL*. Если циклическое выражение возвращает целочисленный массив, то поле *vector* рассматриваемой структуры является указателем на динамический массив, соответствующий результату (выделенная память должна быть освобождена программистом после использования), длина полученного массива хранится в поле *scalar*.

Отметим, что определение структуры *loop_expr* совпадает с определением структуры *tripl*. Поэтому, возможно использовать единую структуру для хранения триплетов и результатов циклических выражений. Но было принято решение использовать для хранения триплетов и результатов циклических выражений структуры с разными наименованиями. Это решение является шагом на пути к улучшению типизации в расширении языка C-light и в расширении языка C-kernel. Также это решение может упростить написание кода на данных языках.

Также язык C-light был расширен циклическими выражениями с условиями завершения. В качестве условий завершения используются выражения языка C-light.

Также язык C-light был расширен выражениями замещения элементов массивов с синтаксисом, сходным с синтаксисом *array_rep_expr*. Но эти выражения замещения элементов массивов были реализованы как инструкции языка C-light. Поэтому, будем называть их инструкциями замещения элементов массивов.

Язык, полученный расширением языка C-light конструкциями языка Cloud-Sisal-kernel, называется C-Sisal-light. Язык C-kernel также был расширен конструкциями языка Cloud-Sisal-kernel, в которые и транслируются соответствующие конструкции C-Sisal-light. Но подвыражения этих конструкций, например, редуцируемые выражения циклов, могут измениться в результате трансляции.

Так как триплеты, циклические выражения и выражения замещения элементов массивов могут содержать побочные эффекты (например, выделение динамической памяти), то правила трансляции, которые локализуют побочные эффекты (например, правило *Ops*), были модифицированы. Триплеты, циклические выражения и выражения замещения элементов массивов были добавлены в область действия этих правил. В результате редуцируемые подвыражения циклов транслируются из C-Sisal-light в C-kernel. Также транслятор *c2acl2* [5] из C-kernel в ACL был модифицирован для трансляции триплетов и циклических выражений.

Язык, полученный расширением языка C-kernel конструкциями языка Cloud-Sisal-kernel, называется C-Sisal-kernel.

4.2. Семантика триплетов, циклических выражений, циклических выражений с условиями завершения и выражений замещения элементов массивов в языке C-Sisal-kernel

Аксиоматическая семантика языка C-kernel была расширена правилами вывода для триплетов, циклических выражений и инструкций замещения элементов массивов. Все инструкции, содер-

жащие триплеты, имеют вид следующего присваивания из-за применения правил трансляции, локализующих побочные эффекты: $var\ =\ low\ ..high\ ..step$; , где var является переменной типа $tripl$, low, $high$ и $step$ являются целочисленными переменными или константами. Эта инструкция является присваиванием триплета переменной. Рассмотрим правило вывода для этой инструкции:

$$\{P\}\ \mathbf{A};\ \{Q(var\ \leftarrow c2acl2(low\ ..hight\ ..step),\ MD \leftarrow upd(MD, var.vector, c2acl2(low\ ..high\ ..step).vector))\}$$

$$\{P\}\ \mathbf{A}; \mathbf{var}\ =\ \mathbf{low}\ ..\mathbf{high}\ ..\mathbf{step};\ \{Q\}$$

где A являются инструкциями программы до рассматриваемого присваивания. Мы используем обратное прослеживание (метод слабейшего предусловия [7]): мы движемся от конца программы к ее началу и элиминируем самый правый оператор (на верхнем уровне), применяя соответствующее правило вывода смешанной аксиоматической семантики [1] языка C-kernel. Отметим, что новое постусловие получается из исходного Q одновременной заменой var на значение триплета и заменой MD на новое состояние памяти. Структура $tripl$ была задана в ACL2, используя конструкции библиотеки fty системы ACL2. Транслятор $c2acl2$ генерирует применение конструктора структуры $tripl$, который устанавливает в качестве значения поля $vector$ значение функции $triplet$.

Все инструкции, содержащие циклические выражения, имеют вид следующего присваивания из-за применения правил трансляции, которые локализуют побочные эффекты: $var\ =\ loop\_e$;. Эта инструкция является присваиванием циклического выражения переменной. Рассмотрим следующее правило вывода для этой инструкции:

$$\{P\}\ \mathbf{A};\ \{Q(var\ \leftarrow c2acl2(loop\_e),\ MD \leftarrow upd(MD, var.vector, c2acl2(loop\_e).vector))\}$$

$$\{P\}\ \mathbf{A};\ \mathbf{var}\ =\ \mathbf{loop\_e}\ \{Q\}$$

где A определяется по аналогии с правилом для триплетов. Отметим, что структура $loop\_expr$ была задана на языке ACL, используя конструкции библиотеки fty системы ACL2. Эта структура аналогична структуре $loop\_expr$, заданной на языке C-light. Определение транслятора $c2acl2$ для этого выражения основано на модификации транслятора $sisal2acl2$ для циклических выражений:

$$\begin{aligned}
&c2acl2(loop_e)\ =\ (make\text{-}loop_expr\\
&:\ scalar\ (rep_id\\
&\quad (reverse\ range2acl2(triplet_1\ cross\ \ldots\ cross\ triplet_n))\\
&\quad (create_environment_id\ context2string(context2vector(\\
&\qquad context_variables(expr) \setminus \{var_1,\ var_2,\ \ldots\ var_{n-1},\ var_n\}))))\\
&:\ vector\ nil)
\end{aligned}$$

где id является уникальным идентификатором. Это результат трансляции в случае целочисленного значения редукции. Если редукция возвращает целочисленный массив, то результат редукции связывается с полем $vector$ вместо поля $scalar$, поле $scalar$ связывается с длиной результата редукции. Результатом трансляции является использование конструктора структуры $loop\_expr$, который задает значения полей, используя значение функции rep. Рассмотрим алгоритм генерации определения функции rep_id:

$$\begin{aligned}
&(defun\ rep_id\ (range_tuples\ environment)\\
&\ (b*\ ((when\ (endp\ range_tuples))\ reduction_init(reduction))\\
&\quad (tuple\ (car\ range_tuples))\\
&\quad (var_1\ (car\ tuple))
\end{aligned}$$

```
   ...
   (var_n (car (cdr (cdr ... (cdr tuple) ...))))
   (previous_iter_result (rep_id (cdr range_tuples) environment)))
 reduct2acl2(reduction, c2acl2(expr), previous_iter_result)))
```

где *range_tuples* является списком, соответствующим результату декартового произведения триплетов. Каждый элемент этого списка является кортежом значений переменных заголовка цикла. Редукция реализована, используя результат транслятора *reduct2acl2*. Применения *car* и *cdr* к *range_tuples* соответствуют применениям *choo* и *rest* к *S* в определении символического метода верификации финитных итераций. Это демонстрирует, что циклические выражения являются финитными итерациями над декартовым произведением триплетов. Главным изменением этого алгоритма относительно генерации *rep_id* для циклов Cloud-Sisal-kernel [22] является применение транслятора *c2acl2* [5] вместо функции *sisal2acl2*.

Метод смешанной аксиоматической семантики позволяет задать версию правила вывода для циклических выражений в случае скалярного результата и отсутствия выделения динамической памяти. Запишем эту версию правила вывода на языке шаблонов [2] для использования этого правила в качестве входа метагенератора УК:

```
{P} A {substitution(Q, var, c2acl2(
   for int_var(var_element_1) in triplet_val(triplet_element_1)
   repeat(cross int_var(var_element_2) in triplet_val(triplet_element_2))
   returns reduction int_val(expr) end for)}
|- {any_predicate(P)} any_code(A)
var = for int_var(var_element_1) in triplet_val(triplet_element_1)
   repeat(cross int_var(var_element_2) in triplet_val(triplet_element_2))
   returns reduction int_val(expr) end for {any_predicate(Q)}
```

где `repeat` является конструкцией, которая позволяет задавать шаблоны, основанные на повторах, `elements` с индексами [5] является конструкцией, которая позволяет задавать векторы переменных и выражений одинакового типа. Конструкция `int_var` сопоставляется с целочисленной переменной, конструкция `int_val` сопоставляется с целочисленным значением и конструкция `triplet_val` сопоставляется с триплетом. Правило вывода для присваивания триплету в случае пустого триплета и отсутствия выделения динамической памяти является схожим с рассмотренным правилом.

Правило вывода для циклического выражения с условием завершения является таким же, как правило вывода для циклического выражения. Главным отличием является алгоритм генерации определения функции *rep_id*, расширенный генерацией проверки условия завершения и генерацией проверки результата предыдущей итерации. Отметим, что такой алгоритм отсутствует для циклов с условиями завершения в языке Cloud-Sisal-kernel. Рассмотрим такую генерацию определения функции *rep_id*:

```
(defun rep_id (range_tuples environment)
 (b *
  (when (endp range_tuples))(update : loop-break nil reduction_init(reduction)))
  (tuple (car range_tuples))
  (var_1 (car tuple))
  ...
  (var_n (car (cdr (cdr ... (cdr tuple) ...))))
  (previous_iter_result (rep_id (cdr range_tuples) environment))
  ((when (not previous_iter_result.loop-break)) previous_iter_result)
  ((when (not c2acl2(condition))) (update : loop-break t previous_iter_result)))
 (update : loop-break nil
  reduct2acl2(reduction, c2acl2(expr), previous_iter_result))))
```

где мы связываем *previous_iter_result* с результатом предыдущей итерации. Если список кортежей диапазона равен *nil*, тогда значением функции является структура со значением редукции по умолчанию и со значением поля *loop-break*, которое равно $false$. Если значение поля *loop-break* из результата предыдущей итерации равно $true$, тогда значением этой функции является *previous_iter_result*. Если условие завершения выполнено, то значением функции является результат предыдущей итерации за исключением значения поля *loop-break*, которое равно $true$. Эти проверки реализованы в первой, во второй и в третьей конструкциях *when* соответственно. Структура с результатом применения редукции и со значением поля *loop-break*, равным $false$, является результатом этой функции в других случаях.

Рассмотрим правило вывода для инструкции замещения элементов массива:

$$\{P\}\ \mathbf{A};\ \{Q(source\_array \leftarrow c2acl2(array\_rep\_expr),\ MD \leftarrow upd(MD, source\_array, c2acl2(array\_rep\_expr)))\}$$

$$\{P\}\ \mathbf{A};\ \mathbf{array\_rep\_expr};\ \{Q\}$$

где A определяется по аналогии с правилом для триплетов. Определение транслятора *c2acl2* для этой инструкции схоже с определением транслятора *sisal2acl2* для выражений замещения элементов массивов. Рассмотрим алгоритм генерации определения функции *update_elements_id* в случае языка C-Sisal-kernel:

```
(defun update_elements_id (range_tuples environment source_array)
   (b* ((when (endp range_tuples)) source_array)
       (tuple (car range_tuples))
       (var_1 (car tuple))
       ...
       (var_n (car (cdr (cdr ... (cdr tuple) ... )))))
     (update-nth var_1
     ...
     (update-nth var_{n-1}
     (update-nth var_n c2acl2(expr)
       (nth var_{n-1}
       ...
       (nth var_1
         (update_elements_id(cdr range_tuples) environment
           source_array))) ...))))
```

где *source_array* является списком вложенных списков в случае многомерного массива. Мы используем вложенные применения функций *nth* и *update-nth* для обновления элемента в многомерном массиве. Главным изменением этого алгоритма относительно генерации *update_elements_id* в случае Cloud-Sisal-kernel является применение транслятора *c2acl2* вместо функции *sisal2acl2*. Кроме того, мы задали это правило вывода на языке шаблонов [2] для использования этого правила в качестве входа метагенератора УК:

```
{P} A {simultaneous_substitution(Q, source_array, c2acl2(
   source_array[int_var(var_element_1) in triplet_val(triplet_element_1)
   repeat(cross int_var(var_element_2) in triplet_val(triplet_element_2))
   := int_val(expr)]),
   MD, (upd MD source_array c2acl2(
   source_array[int_var(var_element_1) in triplet_val(triplet_element_1)
   repeat(cross int_var(var_element_2) in triplet_val(triplet_element_2))
   := int_val(expr)]))}
```

```
|- {any_predicate(P)} any_code(A)
source_array[int_var(var_element_1) in triplet_val(triplet_element_1)
   repeat(cross int_var(var_element_2) in triplet_val(triplet_element_1))
   := int_val(expr)] {any_predicate(Q)}
```

где `simultaneous_substitution` является конструкцией, которая позволяет выполнять одновременную замену. Наша реализация транслятора *c2acl2* и конструкции `triplet_val` в метагенераторе позволяет задавать новые правила вывода как входные данные системы C-lightVer. Данный подход упростил расширение системы верификации C-lightVer рассмотренными правилами вывода.

5. Эксперименты

Опишем эксперименты, демонстрирующие применение представленной семантики для автоматической дедуктивной верификации C-Sisal-kernel программ.

5.1. Произведение элементов матрицы

Автоматическая дедуктивная верификация суммирования элементов матрицы описана в работе [22]. Данная программа была задана на языке Cloud-Sisal-kernel. Рассмотрим автоматическую верификацию схожей программы, заданной на языке C-Sisal-kernel. Эта программа вычисляет произведение элементов целочисленной матрицы:

```
int product_matrix_elements (int** a, int n, int m){
  loop_expr prod = for i in 0..n-1..1 cross j in 0..m-1..1 do
                     returns product of a[i][j] end for;
  return prod.scalar;}
```

Входными данными этой функции является целочисленная матрица a с n строками и m столбцами. Рассмотрим предусловие этой функции, заданное на языке ACL:

```
(and (integerp n) (integerp m) (< 0 n) (< 0 m) (integer-matrixp n m a))
```

Предикат *integerp* проверяет, является ли значение его аргумента целым числом. Целочисленная матрица задана нами в системе ACL2 как список целочисленных списков. Длины всех вложенных списков равны. Предикат *integer-matrixp* проверяет, соответствует ли его аргумент этой структуре.

Постусловием является формула `(= prod (product-matrix n m a))`. Функция *product-matrix* задана нами на языке ACL. Циклическое выражение транслируется в следующую структуру, основанную на применении функции *rep*_1:

```
   (make-loop_expr
     :scalar
       (rep_1
        (reverse
         (cartesian_product (triplet 0 (- n 1) 1) (triplet 0 (- m 1) 1)))
        (create_environment_1 a))
     :vector nil
   ).scalar
```

Рассмотрим определение функции *create_environment*_1 и определение функции *rep*_1:

```
(defun create_environment_1(a) (make-environment_1 :a a))
(defun rep_1 (range_tuples environment)
  (b* (((when (endp range_tuples)) 1)
```

```
      (tuple (car range_tuples))
      (i (car tuple))
      (j (car (cdr tuple)))
      (a environment.a)
      (previous_iter_result (rep_1 (cdr range_tuples) environment)))
   (* (nth j (nth i a)) previous_iter_result)))
```

Метод смешанной аксиоматической семантики [1] и метод метагенерации УК [2] позволили применить правило вывода без *MeM* и *MD*. Полученное слабейшее предусловие основано на замене переменной *prod* в постусловии на результат трансляции цикла. Рассмотрим результат этой замены:

```
(=
 (make-loop_expr
   :scalar
     (rep_1 (reverse
       (cartesian_product (triplet 0 (- n 1) 1) (triplet 0 (- m 1) 1)))
       (create_environment_1 a))
   :vector nil
 ).scalar
 (product-matrix n m a))
```

Рассмотрим полученное УК:

```
(implies
  (and (integerp n) (integerp m) (< 0 n) (< 0 m) (integer-matrixp n m a))
  (=
   (make-loop_expr
     :scalar
       (rep_1 (reverse
         (cartesian_product (triplet 0 (- n 1) 1) (triplet 0 (- m 1) 1)))
         (create_environment_1 a))
     :vector nil
   ).scalar
   (product-matrix n m a)))
```

Система ACL2 автоматически доказала это УК, используя индукцию по n и m. Это доказательство основано на использовании лемм о функциях *integer-matrixp* и *product-matrix*. Следовательно, функция `product_matrix_elements` соответствует спецификациям.

5.2. Изменение знака первого отрицательного элемента массива на противоположный

Рассмотрим программу `negate_first` из набора задач по верификации [26]. Эта программа изменяет знак первого отрицательного элемента массива на противоположный. Главной проблемой верификации программы `negate_first` является использование в теле цикла конструкции, подобной конструкции `break`. Рассмотрим функцию `negate_first`, записанную на языке C-Sisal-kernel:

```
void negate_first(int n, int* a){
    loop_expr index = for i in 0..n-1..1 while a[i] >= 0 do
                      returns value of i end for;
    if (a[index.scalar] < 0) {a[index.scalar] = -a[index.scalar];}}
```

Предусловие этой функции задано на языке ACL:

```
(and (integer-listp a) (integer-listp a_0) (equal a a_0)
     (integerp n) (< 0 n) (<= n (length a_0)))
```

Рассмотрим постусловие этой функции, заданное на языке ACL:

```
(and (implies (not (found-negative n a_0)) (equal a a_0))
     (implies (found-negative n a_0)
              (equal a (update-nth (count-index n a_0)
                                   (- (nth (count-index n a_0) a_0)) a_0)))))
```

В постусловии используется предикат *found-negative*, заданный нами. Этот предикат проверяет наличие отрицательного элемента в массиве. Также в постусловии используется функция *count-index*, заданная нами. Эта функция вычисляет индекс первого отрицательного элемента массива в случае наличия в массиве такого элемента. Значение этой функции не определено в других случаях.

Данное постусловие является конъюнкцией импликаций. Первая импликация соответствует случаю отсутствия отрицательных элементов в массиве. Полученный массив совпадает с исходным массивом в данном случае. Вторая импликация соответствует случаю наличия отрицательного элемента в массиве. В этом случае полученный массив соответствует исходному за исключением первого отрицательного элемента.

Циклическое выражение транслируется в применение функции *rep*_1 к применениям функций *partition* и *create_environment*_1. Рассмотрим определение функции *create_environment*_1 и определение функции *rep*_1:

```
(defun create_environment_1(a) (make-environment_1 :a a))
(defun rep_1 (range_tuples environment)
  (b* (((when (endp range_tuples)) (update :loop-break nil 0))
       (tuple (car range_tuples))
       (i (car tuple))
       (a environment.a)
       (previous_iter_result (rep_1 (cdr range_tuples) environment))
       ((when previous_iter_result.loop-break) previous_iter_result)
       ((when (not (>= (nth i a) 0))) (update :loop-break t
                                              previous_iter_result)))
    i ))
```

где поле *loop-break* является истинным тогда и только тогда, когда конструкция `while` завершает исполнение циклического выражения.

Метод смешанной аксиоматической семантики [1] и метод метагенерации УК [2] позволили применить правило вывода без *MeM* и *MD*. Наличие инструкции *if* привело к генерации двух УК. Рассмотрим одно из них:

```
(implies
 (and (integer-listp a) (integer-listp a_0) (equal a a_0)
      (integerp n) (< 0 n) (<= n (length a_0))
      (>=
       (nth
        (make-loop_expr :scalar
                        (rep_1
                         (reverse (partition (triplet 0 (- n 1) 1)))
                         (create_environment_1 a))
                        :vector nil).scalar
        a)
       0)
 (and
  (implies (not (found-negative n a_0))
```

```
   (equal
    (update-nth
     (make-loop_expr :scalar
                     (rep_1
                      (reverse (partition (triplet 0 (- n 1) 1)))
                      (create_environment_1 a))
                     :vector nil).scalar
     (- (nth (make-loop_expr :scalar
                             (rep_1
                              (reverse (partition (triplet 0 (- n 1) 1)))
                              (create_environment_1 a))
                             :vector nil).scalar

             a))
     a)
    a_0))
  (implies (found-negative n a_0)
   (equal
    (update-nth
     (make-loop_expr :scalar
                     (rep_1
                      (reverse (partition (triplet 0 (- n 1) 1)))
                      (create_environment_1 a))
                     :vector nil).scalar
     (- (nth (make-loop_expr :scalar
                             (rep_1
                              (reverse (partition (triplet 0 (- n 1) 1)))
                              (create_environment_1 a))
                             :vector nil).scalar

             a))
     a)
    (update-nth (count-index n a_0)
                (- (nth (count-index n a_0) a_0)) a_0)))))
)
```

Второе УК схоже с рассмотренным. Структура УК соответствует импликации из предусловия в постусловие с заменой вхождений *a* на выражения *update-nth*, где вместо индекса используется значение функции *rep_1*.

Стратегия автоматизации доказательства УК программ, постусловием которых является разбор случаев [4], позволила системе ACL2 доказать леммы о том, что первая импликация в постусловии эквивалентна случаю отсутствия завершения цикла конструкцией `while` и вторая импликация в постусловии эквивалентна случаю завершения цикла конструкцией `while`. Эти леммы были добавлены в теорию предметной области. Система ACL2 автоматически доказала оба УК, используя индукцию по *n* и эти леммы. Следовательно, функция `negate_first` соответствует спецификациям.

Заключение

В данной статье представлены следующие основные результаты:

- Новый язык программирования C-Sisal-kernel:
  - синтаксис языка C-Sisal-kernel;
  - семантика языка C-Sisal-kernel.
- Эксперименты по автоматической верификации C-Sisal-kernel программ:

– произведение элементов матрицы;
– изменение знака первого отрицательного элемента массива на противоположный.

В данной статье также представлен комплексный подход к автоматизации дедуктивной верификации в случае финитных итераций над структурами данных. Данный подход представляет собой композицию символического метода верификации финитных итераций, метода метагенерации УК и метода смешанной аксиоматической семантики. Символический метод верификации финитных итераций позволяет решить проблему инвариантов для циклов определенного вида. Метод метагенерации УК позволяет упростить добавление новых правил вывода в систему верификации. Метод смешанной аксиоматической семантики позволяет получить более простые УК в некоторых случаях.

Мы планируем расширить язык C-Sisal-kernel такими циклами Cloud Sisal, как выражения конструирования массива. Для решения этой задачи мы планируем применить представленный комплексный подход к автоматизации дедуктивной верификации.

Также мы планируем изменить реализацию синтаксического анализа выражений *upd* из правил вывода. Этот анализ является частью метагенератора. Применение анализа к выражению *upd* из правила вывода не зависит от выражений *upd* в других правилах вывода. Следовательно, мы можем реализовать такой анализ, используя Sisal-итерации над строками, соответствующими правилам вывода. Автоматическое распараллеливание этой части метагенератора будет основано на использовании языка C-Sisal-kernel. Такая реализация может позволить выполнить эксперименты по применению системы верификации к этой части собственного исходного кода. Отметим, что формальная верификация генераторов УК является актуальной задачей [2, 43]. Планируемое самоприменение системы C-lightVer будет шагом на пути к решению этой задачи.

References

[1] I. V. Maryasov, V. A. Nepomniaschy, A. V. Promsky, and D. A. Kondratyev, “Automatic C Program Verification Based on Mixed Axiomatic Semantics”, *Automatic Control and Computer Sciences*, vol. 48, no. 7, pp. 407–414, 2014.

[2] D. A. Kondratyev and A. V. Promsky, “Developing a self-applicable verification system. Theory and practice”, *Automatic Control and Computer Sciences*, vol. 49, no. 7, pp. 445–452, 2015.

[3] D. Kondratyev, “Implementing the Symbolic Method of Verification in the C-Light Project”, in *Perspectives of System Informatics*, ser. LNCS, vol. 10742, Springer, 2018, pp. 227–240.

[4] D. A. Kondratyev, I. V. Maryasov, and V. A. Nepomniaschy, “The Automation of C Program Verification by the Symbolic Method of Loop Invariant Elimination”, *Automatic Control and Computer Sciences*, vol. 53, no. 7, pp. 653–662, 2019.

[5] D. A. Kondratyev and A. V. Promsky, “The Complex Approach of the C-lightVer System to the Automated Error Localization in C-Programs”, *Automatic Control and Computer Sciences*, vol. 54, no. 7, pp. 728–739, 2020.

[6] C. A. R. Hoare, “An axiomatic basis for computer programming”, *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[7] K. R. Apt and E.-R. Olderog, “Fifty years of Hoare’s logic”, *Formal Aspects of Computing*, vol. 31, no. 6, pp. 751–807, 2019.

[8] R. Hähnle and M. Huisman, “Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools”, in *Computing and Software Science*, ser. LNCS, vol. 10000, Springer, 2019, pp. 345–373.

[9] K. R. Apt and E.-R. Olderog, “Assessing the Success and Impact of Hoare’s Logic”, in *Theories of Programming: The Life and Works of Tony Hoare*, 2021, pp. 41–76.

[10] V. A. Nepomniaschy, I. S. Anureev, I. N. Mikhailov, and A. V. Promskii, "Towards verification of C programs. C-light language and its formal semantics", *Programming and Computer Software*, vol. 28, no. 6, pp. 314–323, 2002.

[11] V. A. Nepomniaschy, I. S. Anureev, and A. V. Promskii, "Towards Verification of C Programs: Axiomatic Semantics of the C-kernel Language", *Programming and Computer Software*, vol. 29, no. 6, pp. 338–350, 2003.

[12] V. A. Nepomniaschy, "Symbolic method of verification of definite iterations over altered data structures", *Programming and Computer Software*, vol. 31, no. 1, pp. 1–9, 2005.

[13] M. Moriconi and R. L. Schwartz, "Automatic construction of verification condition generators from hoare logics", in *International Colloquium on Automata, Languages, and Programming*, ser. LNCS, vol. 115, Springer, 1981, pp. 363–377.

[14] J. S. Moore, "Milestones from the Pure Lisp Theorem Prover to ACL2", *Formal Aspects of Computing*, vol. 31, no. 6, pp. 699–732, 2019.

[15] V. Kasyanov and E. Kasyanova, "Methods and System for Cloud Parallel Programming", in *Proceedings of the 21st International Conference on Enterprise Information Systems*, vol. 1, 2019, pp. 623–629.

[16] V. N. Kasyanov and A. P. Stasenko, "Sisal 3.2 Language Structure Decomposition", in *Proceedings of the European Computing Conference*, ser. LNEE, vol. 28, Springer, 2009, pp. 533–543.

[17] A. Stasenko, "Sisal 3.2 Language Features Overview", in *International Conference on Parallel Computing Technologies*, ser. LNCS, vol. 6873, Springer, 2011, pp. 110–124.

[18] V. Kasyanov, "Sisal 3.2: functional language for scientific parallel programming", *Enterprise Information Systems*, vol. 7, no. 2, pp. 227–236, 2013.

[19] J. T. Feo, D. C. Cann, and R. R. Oldehoeft, "A report on the sisal language project", *Journal of Parallel and Distributed Computing*, vol. 10, no. 4, pp. 349–366, 1990.

[20] J.-L. Gaudiot, T. DeBoni, J. Feo, W. Böhm, W. Najjar, and P. Miller, "The Sisal Project: Real World Functional Programming", in *Compiler Optimizations for Scalable Parallel Systems*, ser. LNCS, vol. 1808, Springer, 2001, pp. 45–72.

[21] K. Pyzhov and R. Idrisov, "Back-end translator for Sisal 3.1 compiler", *Bulletin of the Novosibirsk Computing Center*, no. 35, pp. 101–119, 2013.

[22] D. A. Kondratyev and A. V. Promsky, "Towards verification of scientific and engineering programs. The CPPS project", *Journal of Computational Technologies*, vol. 25, no. 5, pp. 91–106, 2020.

[23] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters", in *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, vol. 6, 2004.

[24] M. Kaufmann and J. S. Moore, "Iteration in ACL2", in *Proceedings of the Sixteenth International Workshop on the ACL2 Theorem Prover and its Applications*, ser. EPTCS, vol. 327, 2020, pp. 16–31.

[25] S. Blom, S. Darabi, M. Huisman, and M. Safari, "Correct program parallelisations", *International Journal on Software Tools for Technology Transfer*, vol. 23, no. 5, pp. 741–763, 2021.

[26] B. Jacobs, J. Kiniry, and M. Warnier, "Java Program Verification Challenges", in *Formal Methods for Components and Objects*, ser. LNCS, vol. 2852, Springer, 2003, pp. 202–219.

[27] D. R. Cok, "Reasoning about Functional Programming in Java and C++", in *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, 2018, pp. 37–39.

[28] D. R. Cok and S. Tasiran, "Practical Methods for Reasoning About Java 8's Functional Programming Features", in *Verified Software: Theories, Tools, and Experiments*, ser. LNCS, vol. 11294, Springer, 2018, pp. 267–278.

[29] *ISO/IEC 14882:2020: Programming language C++*. ISO/IEC, 2020.

[30] *ISO/IEC 9899:2011: Programming language C*. ISO/IEC, 2011.

[31] R. Krebbers and F. Wiedijk, "A Typed C11 Semantics for Interactive Theorem Proving", in *Proceedings of the 2015 Conference on Certified Programs and Proofs*, 2015, pp. 15–27.

[32] M. Sammler, R. Lepigre, R. Krebbers, K. Memarian, D. Dreyer, and D. Garg, "RefinedC: automating the foundational verification of C code with refined ownership types", in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 158–174.

[33] M. O. Myreen and M. J. C. Gordon, "Transforming Programs into Recursive Functions", *Electronic Notes in Theoretical Computer Science*, vol. 240, pp. 185–200, 2009.

[34] R. Blanc, V. Kuncak, E. Kneuss, and P. Suter, "An overview of the Leon verification system: verification by translation to recursive functions", in *Proceedings of the 4th Workshop on Scala*, 2013, pp. 1–10.

[35] A. Humenberger, M. Jaroschek, and L. Kovács, "Invariant Generation for Multi-Path Loops with Polynomial Assignments", in *Verification, Model Checking, and Abstract Interpretation*, ser. LNCS, vol. 10747, Springer, 2018, pp. 226–246.

[36] S. Chakraborty, A. Gupta, and D. Unadkat, "Diffy: Inductive Reasoning of Array Programs Using Difference Invariants", in *Computer Aided Verification*, ser. LNCS, vol. 12760, Springer, 2021, pp. 911–935.

[37] T. Tuerk, "Local reasoning about while-loops", in *Proceedings of the Theory Workshop at VSTTE 2010*, 2010, pp. 29–39.

[38] A. Blanchard, F. Loulergue, and N. Kosmatov, "Towards Full Proof Automation in Frama-C Using Auto-active Verification", in *NASA Formal Methods*, ser. LNCS, vol. 11460, Springer, 2019, pp. 88–105.

[39] P. Baudin, F. Bobot, D. Bühler, L. Correnson, F. Kirchner, N. Kosmatov, A. Maroneze, V. Perrelle, V. Prevosto, J. Signoles, and N. Williams, "The dogged pursuit of bug-free C programs: the Frama-C software analysis platform", *Communications of the ACM*, vol. 64, no. 8, pp. 56–68, 2021.

[40] I. Attali, D. Caromel, and A. Wendelborn, "A Formal Semantics and an Interactive Environment for Sisal", in *Tools and Environments for Parallel and Distributed Systems*, ser. SOFT, vol. 2, Springer, 1996, pp. 229–256.

[41] D. Kondratyev and A. Promsky, "Proof Strategy for Automated Sisal Program Verification", in *Software Technology: Methods and Tools*, ser. LNCS, vol. 11771, Springer, 2019, pp. 113–120.

[42] B. Beckert, T. Bingmann, M. Kiefer, P. Sanders, M. Ulbrich, and A. Weigl, "Relational Equivalence Proofs Between Imperative and MapReduce Algorithms", in *Verified Software. Theories, Tools, and Experiments*, ser. LNCS, vol. 11294, Springer, 2018, pp. 248–266.

[43] G. Parthasarathy, P. Müller, and A. Summers, "Formally Validating a Practical Verification Condition Generator", in *Computer Aided Verification*, ser. LNCS, vol. 12760, Springer, 2021, pp. 704–727.

MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 28, NO. 4, 2021
journal homepage: www.mais-journal.ru

THEORY OF COMPUTING

A Mathematical Model of Parallel Programs and an Approach Based on it to Verification of MPI Programs

A. M. Mironov[1]

DOI: 10.18255/1818-1015-2021-4-394-412

[1]Lomonosov Moscow State University, 1 Leninskie Gory, Moscow 119991, Russia.

MSC2020: 68Q60
Research article
Full text in Russian

Received November 15, 2021
After revision December 1, 2021
Accepted December 8, 2021

The paper presents a new mathematical model of parallel programs, on the basis of which it is possible, in particular, to verify parallel programs presented on a certain subset of the parallel programming interface MPI. This model is based on the concepts of a sequential and distributed process. A parallel program is modeled as a distributed process in which sequential processes communicate by asynchronously sending and receiving messages over channels. The main advantage of the described model is the ability to simulate and verify parallel programs that generate an indefinite number of sequential processes. The proposed model is illustrated by the application of verification of the matrix multiplication MPI program.

Keywords: parallel programs; MPI; distributed processes; verification

INFORMATION ABOUT THE AUTHORS

Andrew M. Mironov	orcid.org/0000-0002-9132-7804. E-mail: amironov66@gmail.com
correspondence author	Associate professor.

Funding: Grant from the Ministry of Digital Development, Communications and Mass Media of the Russian Federation and JSC "Russian Venture Company" (contract No. 004/20 from 20.03.2020, IGK 0000000007119P190002) .

For citation: A. M. Mironov, "A Mathematical Model of Parallel Programs and an Approach Based on it to Verification of MPI Programs ", *Modeling and analysis of information systems*, vol. 28, no. 4, pp. 394-412, 2021.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 28, № 4, 2021
сайт журнала: www.mais-journal.ru

THEORY OF COMPUTING

Математическая модель параллельных программ и основанный на ней подход к верификации MPI-программ

А. М. Миронов[1]

DOI: 10.18255/1818-1015-2021-4-394-412

[1]Московский государственный университет им. М. В. Ломоносова, Ленинские горы, д. 1, г. Москва, 119991 Россия.

УДК 519.681.2
Научная статья
Полный текст на русском языке

Получена 15 ноября 2021 г.
После доработки 1 декабря 2021 г.
Принята к публикации 8 декабря 2021 г.

В работе излагается новая математическая модель параллельных программ, на базе которой можно в частности верифицировать параллельные программы, представленные на некотором подмножестве программного интерфейса параллельного программирования MPI. Данная модель основана на понятиях последовательного и распределенного процесса. Параллельная программа моделируется распределенным процессом, в котором последовательные процессы взаимодействуют путем асинхронной передачи и приема сообщений через каналы. Главным преимуществом изложенной модели является возможность моделирования и верификации параллельных программ, порождающих неопределенное число последовательных процессов. Изложенная модель проиллюстрирована применением к верификации MPI программы перемножения матриц.

Ключевые слова: параллельные программы; MPI; распределенные процессы; верификация

ИНФОРМАЦИЯ ОБ АВТОРАХ

Андрей Михайлович Миронов
автор для корреспонденции

orcid.org/0000-0002-9132-7804. E-mail: amironov66@gmail.com
доцент.

Финансирование: Грант министерства цифрового развития, связи и массовых коммуникаций РФ и АО «Российская венчурная компания» (договор No004/20 от 20.03.2020, ИГК 0000000007119P190002).

Для цитирования: A. M. Mironov, “A Mathematical Model of Parallel Programs and an Approach Based on it to Verification of MPI Programs ”, *Modeling and analysis of information systems*, vol. 28, no. 4, pp. 394-412, 2021.

1. Введение

Параллельные программы – это программы, предназначенные для исполнения на многопроцессорных вычислительных системах (МПВС). Проблема разработки корректных и безопасных параллельных программ представляет в настоящее время исключительно высокую актуальность. Формальное обоснование свойств корректности и безопасности параллельных программ (называемое также **верификацией** параллельных программ) является сложной математической задачей. Существующие методы решения данной задачи пригодны лишь для достаточно ограниченного класса параллельных программ.

Одним из наиболее широко используемых средств для описания параллельных программ является программный интерфейс MPI (Message Passing Interface). В настоящей работе вводится новая математическая модель параллельных программ, на основе которой можно решать задачи верификации параллельных программ, представленных на некотором подмножестве MPI. Введенная модель иллюстрируется применением к решению задачи верификации MPI-программы умножения матриц. Наиболее близким из существующих в настоящее время подходов к моделированию параллельных программ к тому подходу, который излагается в настоящей работе, является формализм расширенных конечных автоматов (Extended Finite State Machine – EFSM) [1].

Наиболее важной особенностью предлагаемого подхода является возможность его применения для верификации параллельных программ, которые могут порождать неопределенное число процессов. Среди других подходов к моделированию и верификации таких программ следует отметить подход в [2]. В этой работе представлен инструмент ParTypes для моделирования и верификации параллельных программ, порождающих неопределенное число процессов. К сожалению, данный инструмент не работает для параллельных программ со свойством wildcard receives, которое имеет следующий смысл: в параллельной программе допускается использование действий приема сообщений от произвольного процесса (т.е. при выполнении действия такого типа номер процесса, от коротого принимается сообщение, можно определить лишь после выполнения этого действия). В частности, при помощи подхода, лежащего в основе данного инструмента, невозможно верифицировать MPI-программу для умножения матриц, рассматриваемую в настоящей статье. Среди работ, посвященных верификации MPI-программ, следует также отметить работу [3], в которой рассматривается пример верификации MPI-программы, основанный на применении аппарата машин абстрактных состояний (ASM) [4]. Существуют и другие подходы с использованием символьного исполнения и проверки модели, см. например [5—12], однако все эти подходы пригодны лишь для анализа параллельных программ, порождающих заранее заданное количество процессов.

В настоящей работе рассматривается решение задачи верификации конкретной MPI-программы, вычисляющей произведение двух матриц. Отметим, что верифицируется лишь модель данной программы, в которой умножение чисел предполагается не приближенным, а точным. Подход к верификации, представленный в работе, связан с преобразованием MPI-программы в распределенный процесс. В работе не описывается общий метод верификации, пригодный для верификации достаточно широкого класса MPI-программ, и нет описания перспектив его автоматизации. Тем не менее, преимуществом данной работы является пример решения задачи верификации такой MPI-программы, которую невозможно верифицировать на основе использования других известных моделей параллельных программ.

2. Необходимые сведения по MPI

2.1. MPI-программы

MPI (Message Passing Interface) представляет собой набор функций, типов и констант, позволяющих создавать программы (называемые **MPI-программами**) для МПВС.

Выполнение MPI-программы на МПВС заключается в том, что на каждом из узлов, входящих в эту систему, порождается вычислительный процесс, соответствующей этой MPI-программе. Все процессы, порожденные MPI-программой на узлах МПВС, функционируют параллельно, и могут обмениваться информацией друг с другом посредством **передачи сообщений**. Совокупность всех процессов, порожденных MPI-программой, обозначается записью `MPI_COMM_WORLD`.

Каждый из процессов, порожденных MPI-программой, имеет номер (называемый **рангом**), являющийся числом из множества $\{0, \dots, m-1\}$, где m – количество процессов, порожденных MPI-программой.

MPI содержит функции `MPI_Comm_rank` и `MPI_Comm_size`, позволяющие каждому из процессов, порождаемых MPI-программой, узнать свой ранг и количество запущенных процессов, порожденных этой MPI-программой, соответственно. Эти функции имеют следующий формат.

- `int rank;`
  `MPI_Comm_rank (MPI_COMM_WORLD, &rank);`
  После выполнения данной функции значение переменной `rank` будет равно рангу процесса, вызвавшего эту функцию.
- `int nprocs;`
  `MPI_Comm_size (MPI_COMM_WORLD, &nprocs);`
  После выполнения данной функции значение переменной `nprocs` будет равно количеству процессов, порожденных MPI-программой.

Процессы, порожденные какой-либо MPI-программой, могут выполняться по-разному из-за того, что результаты вызова `MPI_Comm_rank` в этих процессах будут различными.

2.2. MPI-функции передачи сообщений

Под **сообщением** в MPI понимается массив данных определенного типа, а под **передачей сообщения (ПС)** – действие, в результате которого сообщение одного процесса пересылается другому процессу. В настоящем тексте мы будем рассматривать лишь **асинхронный** режим ПС, который заключается в том, что процесс, пославший сообщение, после посылки сообщения не приостанавливает свою работу (переходя в режим ожидания доставки этого сообщения), а посланное сообщение помещается в очередь, из которой оно затем будет взято процессом-получателем.

Мы будем рассматривать MPI-функции ПС следующих двух видов.

- Функции **попарной ПС (ППС)**.
  В ППС участвуют только два процесса, один из них является отправителем сообщения, а другой – получателем этого сообщения.
- Функция **коллективной ПС (КПС)**.
  В рассматриваемой функции КПС участвуют все процессы, порожденные MPI-программой, процесс с рангом 0 является отправителем сообщения, а остальные процессы являются получателями.
  Сообщения, посылаемые функциями КПС, не могут быть получены функциями ППС, и наоборот.

Ниже мы приводим описания некоторых MPI-функций ПС. В объяснениях смысла этих функций после имени каждого аргумента мы указываем в скобках его тип.

1. Посылка сообщения (ППС):

$$\begin{array}{ll} \texttt{MPI\_Send} & (p, n, \tau, \\ & r, l, \texttt{MPI\_COMM\_WORLD}); \end{array} \tag{1}$$

   Выполнение данной функции заключается в посылке сообщения процессу с рангом r (`int`). Посылаемое сообщение представляет собой массив из n (`int`) элементов типа τ (`MPI_Datatype`),

начало которого находится по адресу p (`void *`). К посылаемому сообщению присоединяется тег (т.е. метка) l (`int`).

2. Получение сообщения (ППС):

$$\begin{aligned} \texttt{MPI\_Recv} \quad & (p, n, \tau, \\ & \texttt{MPI\_ANY\_SOURCE}, \texttt{MPI\_ANY\_TAG}, \\ & \texttt{MPI\_COMM\_WORLD}, q); \end{aligned} \tag{2}$$

Выполнение данной функции заключается в получении сообщения, которое должно быть размещено в участке памяти по адресу p. Предполагается, что получаемое сообщение представляет собой массив из не более n элементов типа τ.
q (`MPI_Status *`) – адрес структуры, в которой должна быть размещена информация о принятом сообщении. Эта структура содержит поля: `MPI_SOURCE` (в нем должен быть размещен ранг отправителя), `MPI_TAG` (в нем должен быть размещен тег принятого сообщения), и другие поля.

3. Рассылка сообщения из процесса с рангом 0 остальным процессам (КПС):

$$\begin{aligned} \texttt{MPI\_Bcast} \quad & (p, n, \tau, \\ & 0, \texttt{MPI\_COMM\_WORLD}); \end{aligned} \tag{3}$$

Эта функция выполняется следующим образом.

- В процессе с рангом 0 происходит посылка всем остальным процессам сообщения, которое представляет собой массив из n (`int`) элементов типа τ (`MPI_Datatype`), начало которого находится по адресу p (`void *`).
- В остальных процессах происходит получение от процесса с рангом 0 сообщения, которое представляет собой массив из n элементов типа τ, и должно быть размещено в участке памяти по адресу p.

3. MPI-программа умножения матриц

В этом пункте мы излагаем пример MPI-программы умножения матриц. Задача умножения матриц заключается в том, чтобы по заданным матрицам A и B вычислить их произведение $C = AB$. Данный пример используется ниже для иллюстрации применения излагаемой в настоящей работе модели параллельных программ для решения задачи верификации MPI-программ.

3.1. Неформальное описание MPI-программы умножения матриц

Неформально работу излагаемой в этом пункте MPI-программы для умножения матриц A и B можно описать следующим образом. Мы будем называть процесс с рангом 0 **менеджером**, а остальные процессы – **работниками**. Работа менеджера состоит из следующих действий:

- рассылка второй матрицы (B) всем работникам,
- назначение задач работникам, и
- прием результатов от работников.

Каждая задача работнику заключается в вычислении одной строки матрицы $C = AB$. Менеджер назначает эту задачу путем посылки работнику сообщения, содержащего одну строку матрицы A. Тег этого сообщения равен номеру посылаемой строки.

Как только менеджер получает от работника результат (т.е. сообщение с вычисленной строкой произведения, его тег равен номеру этой строки), он посылает этому работнику

- новую задачу (если еще есть неназначенные задачи), или
- сообщение с тегом 0 (если неназначенных задач нет).

3.2. MPI-программа умножения матриц

Излагаемая ниже MPI-программа для умножения матриц использует вспомогательную функцию `vecmat` умножения строки `vector[L]` на матрицу `matrix[L][M]` и записи результата в массив `result[M]`. Данная функция имеет следующий вид:

```
void vecmat (double vector[L],
             double matrix[L][M],
             double result[M])
{ int j, k;
  for (j = 0; j < M; j++)
    for (k = 0, result[j] = 0.0; k < L; k++)
      result[j] += vector[k]*matrix[k][j];
}
```

В излагаемой ниже MPI-программе будем использовать следующие обозначения: `input(a,b);` и `output(c);` являются сокращенными записями операторов чтения из файла матриц-сомножителей и записи в файл матрицы-произведения соответственно.

MPI-программа умножения матриц A и B имеет следующий вид (в этой программе слева от каждой строки мы указываем ее номер, это необходимо для описания соответствия между компонентами данной программы и компонентами модели, соответствующей этой программе):

```
01 #define comm MPI_COMM_WORLD
02
03 int main(int argc, char *argv[])
04 { int rank, nprocs, i, j;
05   MPI_Status status;
06
07   MPI_Init(&argc, &argv);
08   MPI_Comm_size(comm, &nprocs);
09   MPI_Comm_rank(comm, &rank);
10
11   if (rank == 0)
12   { int count;
13     double a[N][L], b[L][M], c[N][M], tmp[M];
14
15     input(a, b);
16     MPI_Bcast(b, L*M, MPI_DOUBLE,
17               0, comm);
18     for (count = 0;
19          count < nprocs-1 && count < N;
20          count++)
21       MPI_Send(&a[count][0], L, MPI_DOUBLE,
22                count+1, count+1, comm);
23     for (i = 0; i < N; i++)
24     { MPI_Recv(tmp, M, MPI_DOUBLE,
25                MPI_ANY_SOURCE, MPI_ANY_TAG,
26                comm, &status );
27       for (j = 0; j < M; j++)
```

```
28          c[status.MPI_TAG-1][j] = tmp[j];
29        if (count < N)
30        { MPI_Send(&a[count][0], L, MPI_DOUBLE,
31                   status.MPI_SOURCE, count+1, comm);
32          count++;
33        }
34      }
35      for (i = 1; i < nprocs; i++)
36        MPI_Send(NULL, 0, MPI_INT,
37                 i, 0, comm);
38      output (c);
39    }
40
41    else
42    { double b[L][M], in[L], out[M];
43
44      MPI_Bcast(b, L*M, MPI_DOUBLE,
45                0, comm);
46      while (1)
47      { MPI_Recv(in, L, MPI_DOUBLE,
48                 0, MPI_ANY_TAG,
49                 comm, &status);
50        if (status.MPI_TAG == 0) break;
51        vecmat(in, b, out);
52        MPI_Send(out, M, MPI_DOUBLE,
53                 0, status.MPI_TAG, comm);
54      }
55    }
56
57    MPI_Finalize();
58    return 0;
59  }
```

3.3. Вспомогательные обозначения

Для удобства моделирования и анализа данной программы мы введем специальные обозначения для некоторых используемых в ней объектов:

- массивы `a[N][L]`, `b[L][M]`, `c[N][M]` будем обозначать символами A, B, C и интерпретировать их как соответствующие матрицы,
- сообщение, пересылаемое функциями `MPI_Send` в строках 21, 22 и 30, 31 программы, будем понимать как соответствующую строку матрицы A, и обозначать ее записью A_i, где i = `count` + 1,
- массив `c[status.MPI_TAG-1][M]`, в который копируется сообщение, принятое функцией `MPI_Recv` в строках 24, 25, 26, будем понимать как соответствующую строку матрицы C, и обозначать ее записью C_l, где l =`status.MPI_TAG`,
- из определения функции `vecmat` следует, что массив `out`, вычисляемый в результате выполнения функции `vecmat` в строке 51, соответствует произведению строки Y, соответствующей массиву `in`, на матрицу B, будем обозначать в модели MPI-программы данный массив `out` записью YB.

3.4. Спецификация программы умножения матриц

Спецификация изложенной выше MPI-программы умножения матриц представляет собой следующее утверждение: после завершения выполнения этой программы должно быть верно утверждение $C = AB$, которое эквивалентно утверждению

$$\forall\, i = 1, \dots, N \quad C_i = A_i B. \tag{4}$$

4. Модель параллельных программ

В этом пункте мы излагаем модель параллельных программ. Данная модель позволяет формально представлять MPI-программы, в которых используются описанные выше операторы пересылки сообщений.

Основными понятиями данной модели являются понятия последовательного и распределенного процессов. Последовательный процесс является моделью вычислительного процесса, порождаемого параллельной программой на каком-либо узле МПВС, а распределенный процесс является моделью всей параллельной программы. Предложенная модель является теоретической основой для решения задач верификации параллельных программ.

4.1. Вспомогательные понятия

4.1.1. Типы, переменные, функциональные символы, значения, термы, формулы, связывания

Предполагаем, что заданы множества $\mathcal{T}$, $\mathcal{X}$ и $\mathcal{F}$, элементы которых называются **типами**, **переменными**, и **функциональными символами (ФС)**, соответственно. Каждому элементу x множеств $\mathcal{X}$ и $\mathcal{F}$ сопоставлен некоторый тип $\tau_x \in \mathcal{T}$. $\forall f \in \mathcal{F}$ τ_f имеет вид

$$(\tau_1, \dots, \tau_n) \longrightarrow \tau, \quad \text{где } \tau_1, \dots, \tau_n, \tau \in \mathcal{T}. \tag{5}$$

$\forall \tau \in \mathcal{T}$ задано множество $\mathcal{D}_\tau$ **значений** типа τ. Символ $\mathcal{D}$ обозначает множество значений всех типов.

$\mathcal{T}$ содержит следующие типы:

- **B** (булевский тип), $\mathcal{D}_\mathrm{B} = \{0, 1\}$,
- **N** (натуральный тип), $\mathcal{D}_\mathrm{N} = \{0, 1, \dots\}$,
- **C**, значения этого типа называются **каналами**.

Множество $\mathcal{E}$ **термов** определяется индуктивно. Каждому терму e сопоставлен тип $\tau_e \in \mathcal{T}$. Определение терма имеет следующий вид:

- $\forall \tau \in \mathcal{T}$ каждое значение типа τ является термом типа τ,
- каждая переменная $x \in \mathcal{X}$ является термом типа τ_x,
- если $f \in \mathcal{F}$, $e_1, \dots, e_n$ – термы, и $\tau_f = (\tau_{e_1}, \dots, \tau_{e_n}) \longrightarrow \tau$, то запись $f(e_1, \dots, e_n)$ является термом типа τ.

$\forall f \in \mathcal{F}$, если τ_f имеет вид (5), то этому ФС сопоставлена функция (обозначаемая тем же символом f) вида $\mathcal{D}_{\tau_1} \times \dots \times \mathcal{D}_{\tau_n} \longrightarrow \mathcal{D}_\tau$.

Если $e \in \mathcal{E}$ и $\mathcal{X}_e = \emptyset$, то с e связано **значение** $value(e) \in \mathcal{D}_{\tau_e}$, которое определяется индуктивно:

- если $e \in \mathcal{D}_{\tau_e}$, то $value(e) = e$, и
- если $e = f(e_1, \dots, e_n)$, то $value(e) = f(value(e_1), \dots, value(e_n))$.

Ниже, как правило, будем обозначать значения термов без переменных теми же записями, которыми обозначаются сами эти термы.

Будем считать, что

- $\forall n \geq 1$ $\mathcal{F}$ содержит ФС $tuple_n$, позволяющий строить кортежи: для каждого списка термов $e_1, \dots, e_n$ множество $\mathcal{E}$ содержит терм $tuple_n(e_1, \dots, e_n) \in \mathcal{E}$, который будем обозначать $(e_1, \dots, e_n)$ и интерпретировать как кортеж термов $e_1, \dots, e_n$,

- $\mathcal{F}$ содержит ФС *channel* типа $\mathbf{N} \longrightarrow \mathbf{C}$, $\forall\, i \geq 0$ канал $channel(i)$ будем называть ***i*–м каналом**, терм вида $channel(e)$ будем обозначать записью c_e,
- $\mathcal{D}_\mathbf{C}$ содержит канал $\circ$, который будем называть **широковещательным каналом**, он отличается от всех каналов вида c_i.

Будем использовать следующие обозначения:

- $\mathcal{B}$ и $\mathcal{C}$ обозначают множества $\mathcal{E}_\mathbf{B}$ и $\mathcal{E}_\mathbf{C}$ соответственно, термы из $\mathcal{B}$ называются **формулами**,
- $\forall\, e \in \mathcal{E}\ \ \mathcal{X}_e = \{x \in \mathcal{X} \mid x \text{ входит в } e\}$,
- $\forall\, X \subseteq \mathcal{X}\ \ \mathcal{E}(X) = \{e \in \mathcal{E} \mid \mathcal{X}_e \subseteq X\}$, $\mathcal{B}(X) = \mathcal{E}(X) \cap \mathcal{B}$,
- $\forall\, E \subseteq \mathcal{E}\ \ \forall\, \tau \in \mathcal{T}\ \ E_\tau = \{e \in E \mid \tau_e = \tau\}$.

Ниже для каждой рассматриваемой функции вида $f : E \longrightarrow E'$, где $E, E' \subseteq \mathcal{E}$, будем предполагать, что $\forall\, e \in E\ \ \tau_{f(e)} = \tau_e$.

Запись $\mathcal{D}^*$ обозначает совокупность всех кортежей вида $(d_1, \ldots, d_n)$, где $n \geq 0$ и $d_1, \ldots, d_n \in \mathcal{D}$, в случае $n = 0$ соответствующий кортеж называется **пустым** и обозначается символом ε. Элементы $\mathcal{D}^*$ будем называть **очередями**. $\forall\, D = (d_1, \ldots, d_n) \in \mathcal{D}^*$ запись $|D|$ обозначает число компонентов в D (т.е. n). Если $D = (d_1, \ldots, d_n) \in \mathcal{D}^*$ и $1 \leq i \leq n$, то запись D_i обозначает i-компоненту этого кортежа (т.е. d_i). $\forall\, M \subseteq \mathcal{D}^*$, $\forall\, i \geq 1$ запись M_i обозначает множество i-х компонентов кортежей из M. Если кортеж $D = (d_1, \ldots, d_n) \in \mathcal{D}^*$ непуст, то записи $head(D)$ и $tail(D)$ обозначают значение $d_1 \in \mathcal{D}$ и кортеж $(d_2, \ldots, d_n) \in \mathcal{D}^*$ соответственно.

4.1.2. Связывания

Связыванием называется произвольная функция $\theta : \mathcal{X} \longrightarrow \mathcal{E}$.

Будем использовать следующие обозначения:

- множество всех связываний обозначается символом Θ,
- $\forall\, X \subseteq \mathcal{X}\quad \Theta(X) = \{\theta \in \Theta \mid \forall\, x \in \mathcal{X} \setminus X\ \ \theta(x) = x\}$,
- $\forall\, \theta \in \Theta,\ \forall\, e \in \mathcal{E}$ запись e^θ обозначает терм, получаемый из e заменой $\forall\, x \in \mathcal{X}_e$ каждого вхождения x в e на терм $\theta(x)$,
- $\forall\, \theta, \theta' \in \Theta$ запись $\theta\theta'$ обозначает связывание, определяемое следующим образом: $\forall\, x \in \mathcal{X}\ \ (\theta\theta')(x) = (x^\theta)^{\theta'}$.

4.2. Последовательные процессы

В этом пункте определяется понятие последовательного процесса. Данное понятие является моделью вычислительного процесса, порождаемого параллельной программой на каком-либо узле МПВС.

4.2.1. Действия

Элементарные действия (ЭД) – это записи следующих видов:

$$c!e, \quad c?e, \quad e := e', \quad [\![\varphi]\!], \quad \text{где } c \in \mathcal{C},\ e, e' \in \mathcal{E}, \tau_e = \tau_{e'}, \varphi \in \mathcal{B},$$

которые называются **посылкой** сообщения e в канал c, **приемом** сообщения e из канала c, **присваиванием** и **условным переходом** соответственно.

Действие – это конечная последовательность ЭД, в которой имеется не более одной посылки или приема. Действие называется **посылкой** или **приемом**, если одно из входящих в него ЭД – посылка или прием, соответственно. Действие, не являющееся посылкой или приемом, называется **внутренним действием**. Каждое ЭД можно рассматривать как действие, состоящее из одного этого ЭД.

Множество всех действий обозначается символом $\mathcal{A}$. $\forall\, \alpha \in \mathcal{A}$ множество всех переменных, входящих в α, обозначается записью $\mathcal{X}_\alpha$.

Если $\theta \in \Theta$ и $\alpha \in \mathcal{A}$, то запись α^θ обозначает действие, получаемое из α заменой каждой переменной x в α на терм x^θ.

4.2.2. Понятие последовательного процесса

Последовательный процесс (ПП) – это тройка (P, X, φ), компоненты которой имеют следующий смысл:

- P – граф с выделенной вершиной P^0 (называемой **начальной вершиной**), каждому ребру которого сопоставлена метка $\alpha \in \mathcal{A}$,
- $X \subseteq \mathcal{X}$ – множество **входных переменных** ПП P, и
- $\varphi \in \mathcal{B}$ – **начальное условие** ПП P.

Для каждого ПП (P, X, φ)

- данный ПП может сокращенно обозначаться тем же символом P, что и соответствующий ему граф, множество вершин графа P также обозначается символом P,
- X_P и φ_P обозначают вторую и третью компоненту P соответственно, $\mathcal{X}_P$ обозначает множество всех переменных, входящих в P,
- $\hat{X}_P$ обозначает множество $\mathcal{X}_P \setminus X_P$ **внутренних** переменных,
- $\mathcal{A}_P$ обозначает совокупность меток всех ребер графа P,
- $P^{v \to v'}$ обозначает произвольное ребро графа P из v в v'.

ПП является формальным описанием поведения динамической системы, работа которой заключается в последовательном выполнении действий, связанных с посылкой сообщений в каналы или приемом сообщений из каналов, а также с изменением значений внутренних переменных.

Будем предполагать, что для каждого ПП P

- каждая вершина графа P является элементом множества $\mathcal{D}$,
- P содержит внутреннюю переменную at_P, и для каждого ребра графа P, если v и v' – начало и конец соответственно этого ребра, то первое ЭД в метке этого ребра имеет вид $[\![at_P = v]\!]$, а последнее – $at_P := v'$, эти ЭД не будут указываться явно.

4.2.3. Состояние последовательного процесса

Состояние ПП P – это пара

$$s = (\theta^s, \{[c]^s \mid c \in \mathcal{D}_{\mathrm{C}}\}), \tag{6}$$

компоненты которой интерпретируются следующим образом:

- $\theta^s \in \Theta(\mathcal{X}_P)$ – связывание в s, причем $\forall\, x \in \mathcal{X}_P \;\; \mathcal{X}_{x^{\theta^s}} = \emptyset$,
- $\forall\, c \in \mathcal{D}_{\mathrm{C}} \;\; [c]^s \in \mathcal{D}^*$ – очередь непрочитанных значений в канале c в состоянии s ($[c]^s$ называется **содержимым** канала c в s).

Множество всех состояний ПП P обозначается записью Σ_P.

Для каждого состояния $s \in \Sigma_P$ и каждого терма $e \in \mathcal{E}(\mathcal{X}_P)$ будем обозначать значение e^{θ^s} записью e^s.

Состояние ПП P называется **начальным** (и обозначается 0_P), если оно имеет вид $(\theta, \{\varepsilon \mid c \in \mathcal{D}_{\mathrm{C}}\})$, где $\varphi_P^\theta = 1$ и $at_P^s = P^0$.

Состояние ПП P называется **терминальным**, если из вершины at_P^s не выходит ни одного ребра.

4.2.4. Переходы в последовательных процессах

В этом пункте мы определяем понятие перехода в ПП P, соответствующего какому-либо действию $\alpha \in \mathcal{A}_P$. Этот переход понимается как пара s, s' состояний из Σ_P, связь между которыми можно понимать следующим образом: если ПП P в текущий момент времени находился в состоянии s, то после последовательного выполнения ЭД, входящих в α, новым состоянием ПП P является состояние s'.

Будем обозначать записью $s \xrightarrow{\alpha} s'$ утверждение о том, что пара s, s' является переходом, соответствующим действию α, и определим это утверждение следующим образом: если действие $\alpha \in \mathcal{A}_P$

является последовательностью ЭД вида $\alpha_1 \dots \alpha_n$, то утверждение $s \xrightarrow{\alpha} s'$ верно, если

$$\exists\, s_1, \dots, s_{n-1} \in \Sigma_P \,:\, s \xrightarrow{\alpha_1} s_1,\ s_1 \xrightarrow{\alpha_2} s_2,\ \dots,\ s_{n-1} \xrightarrow{\alpha_n} s',$$

где утверждение $s \xrightarrow{\alpha} s'$ в том случае, когда α – ЭД, определяется отдельно для каждого возможного вида ЭД α (после формального определения данного утверждения для каждого конкретного вида α мы неформально интерпретируем изменение состояния в результате этого перехода):

(a) если $\alpha = c!e$, то $\theta^{s'} = \theta^s$, и

$$[c^s]^{s'} = ([c^s]^s, e^s), \quad \forall\, c' \in \mathcal{D}_{\mathrm{C}} \setminus \{c^s\} \quad [c']^{s'} = [c']^s,$$

в данном случае P посылает в канал c^s значение e^s, после чего

- связывание переменных ПП P не изменилось, и содержимое всех каналов, кроме канала c^s, также не изменилось,
- содержимое канала c^s увеличилось путем добавления к очереди $[c^s]^s$ значения e^s,

(b) если $\alpha = c?e$, то $[c^s]^s \neq \emptyset$ и

$$\begin{gathered}\exists\, \theta \in \Theta(\hat{X}_P) \,:\, (e^\theta)^s = head([c^s]^s), \theta^{s'} = \theta\theta^s,\\ [c^s]^{s'} = tail([c^s]^s), \quad \forall\, c' \in \mathcal{D}_{\mathrm{C}} \setminus \{c^s\} \quad [c']^{s'} = [c']^s\end{gathered}$$

в данном случае P принимает из канала c^s значение $head([c^s]^s)$ и изменяет текущее связывание так, чтобы значение терма e при новом связывании совпадало с принятым значением, после чего

- содержимое канала c^s стало $tail([c^s]^s)$,
- содержимое остальных каналов не изменилось,

(c) если $\alpha = (e := e')$, то

$$\begin{gathered}\exists\, \theta \in \Theta(\hat{X}_P) \,:\, (e^\theta)^s = (e')^s, \theta^{s'} = \theta\theta^s,\\ \forall\, c \in \mathcal{D}_{\mathrm{C}} \quad [c]^{s'} = [c]^s,\end{gathered} \tag{7}$$

в данном случае P изменяет текущее связывание так, чтобы значение терма e при новом связывании совпадало бы со значением терма e' при старом связывании, содержимое каналов не изменяется,

(d) если $\alpha = [\![\varphi]\!]$, то $\varphi^s = 1$, $\theta^{s'} = \theta^s$, и верно (7),
в данном случае связывание и содержимое каналов не изменяется.

Пустым переходом в ПП P называется произвольная пара состояний $s, s' \in \Sigma_P$, такая, что $\theta^s = \theta^{s'}$. Пустые переходы обозначаются записями вида $s \longrightarrow s'$.

Если пара s, s' является переходом в ПП, то s называется началом этого перехода, а s' – его концом.

4.2.5. Редукция графов последовательных процессов

Пусть заданы ПП P и вершина $v \in P$, не являющаяся начальной. Если множества всех ребер в P с концом в v и с началом в v имеют вид

$$\{v_i \xrightarrow{\alpha_i} v \mid i = 1, \dots, n\} \quad \text{и} \quad \{v \xrightarrow{\alpha'_i} v'_i \mid i = 1, \dots, n'\},$$

соответственно, где v отличается от всех вершин v_i и v'_i, и либо все действия $\alpha_1, \dots, \alpha_n$ внутренние, либо все действия $\alpha'_1, \dots, \alpha'_{n'}$ внутренние, то к данному графу можно применить операцию **редукции**, которая заключается в преобразовании данного графа путем

- удаления вершины v и связанных с нею ребер, и
- добавления ребер вида $v_i \xrightarrow{\alpha_i\alpha'_{i'}} v'_{i'}$, где $i = 1, \dots, n$, $i' = 1, \dots, n'$, и $\alpha_i\alpha'_{i'}$ – конкатенация последовательностей α_i и $\alpha'_{i'}$.

4.2.6. Переименования

Переименованием называется произвольная инъективная функция вида $\eta : X \longrightarrow X'$, где $X, X' \subseteq \mathcal{X}$.

Для каждого переименования $\eta : X \longrightarrow X'$, каждого $e \in \mathcal{E}$ и каждого ПП P записи e^η и P^η обозначают терм или ПП соответственно, получаемые из e или P заменой $\forall\, x \in X$ каждого вхождения x на $\eta(x)$.

Если P – ПП, и η – переименование вида $\eta : \hat{X}_P \longrightarrow \mathcal{X} \setminus X_P$, то будем рассматривать ПП P и P^η как равные.

4.3. Распределенные процессы

В этом пункте вводится понятие распределенного процесса, которое можно использовать для моделирования параллельных программ.

4.3.1. Понятие распределенного процесса

Распределенный процесс (РП) – это семейство ПП

$$\mathcal{P} = \{P_i \mid i \in I\}, \tag{8}$$

где компоненты семейства $\{\hat{X}_{P_i} \mid i \in I\}$ дизъюнктны и не пересекаются с множеством $X_\mathcal{P} \stackrel{\text{def}}{=} \bigcup_{i \in I} X_{P_i}$ (если это условие не выполняется, то заменим каждый ПП P_i на равный ему, в смысле, указанном в конце пункта 4.2.6, так, чтобы это условие выполнялось). Ниже будем считать, что данное условие всегда выполнено, даже если внутренние переменные в ПП P_i и $P_{i'}$ из $\mathcal{P}$, где $i \neq i'$, имеют одинаковые обозначения

Запись $\mathcal{X}_\mathcal{P}$ обозначает множество всех переменных, входящих в $\mathcal{P}$.

4.3.2. Понятие состояния распределенного процесса

Состоянием РП $\mathcal{P} = \{P_i \mid i \in I\}$ называется пара

$$s = (\theta^s, \{c^s \mid c \in \mathcal{D}_\mathrm{C}\}), \tag{9}$$

где $\theta^s \in \Theta(\mathcal{X}_\mathcal{P})$, и $\forall\, i \in I$

$$s_i \stackrel{\text{def}}{=} (\theta^s_i, \{c^s \mid c \in \mathcal{D}_\mathrm{C}\}) \in \Sigma_{P_i}$$

где $\theta^s_i \in \Theta(\mathcal{X}_{P_i})$, $\forall\, x \in \mathcal{X}_{P_i} \;\; x^{\theta^s_i} = x^{\theta^s}$.

Множество всех состояний РП $\mathcal{P}$ обозначается записью $\Sigma_\mathcal{P}$.

$\forall\, s \in \Sigma_\mathcal{P}$, $\forall\, e \in e \in \mathcal{E}(\mathcal{X}_\mathcal{P})$ будем обозначать значение e^{θ^s} записью e^s.

Состояние $s \in \Sigma_\mathcal{P}$ называется

- **начальным** (и обозначается $0_\mathcal{P}$), если $\forall\, i \in I \;\; s_i = 0_{P_i}$,
- **терминальным**, если $\forall\, i \in I$ состояние s_i терминально,
- **тупиковым**, если оно нетерминально, и $\forall\, i \in I$ не существует непустого перехода ПП P_i с началом s_i.

4.3.3. Переходы в распределенных процессах

Пусть заданы РП $\mathcal{P} = \{P_i \mid i \in I\}$, индекс $i \in I$ и действие $\alpha \in \mathcal{A}_{P_i}$.

Переходом в РП $\mathcal{P}$, соответствующим действию α ПП P_i, называется произвольная пара состояний $s, s' \in \Sigma_\mathcal{P}$, такая, что

$$s_i \stackrel{\alpha}{\longrightarrow} s'_i, \;\; \forall\, i' \in I \setminus \{i\} \;\; s_{i'} \longrightarrow s'_{i'}. \tag{10}$$

Будем обозначать свойство (10) записью $P_i^{\upsilon \to \upsilon'} : s \longrightarrow s'$, где υ, υ' – начало и конец соответственно ребра графа P_i с меткой α. Если пара s, s' является переходом в РП $\mathcal{P}$, то будем обозначать это записью $s \longrightarrow s'$.

Связь между состояниями $s, s' \in \Sigma_{\mathcal{P}}$, удовлетворяющими свойству (10), можно интерпретировать следующим образом: если РП $\mathcal{P}$ в текущий момент времени находился в состоянии s, и, начиная с этого момента, ПП P_i последовательно выполнял ЭД, входящие в α, а остальные ПП из $\mathcal{P}$ в течение всего этого времени не выполняли никаких действий, то после завершения выполнения последовательности ЭД, входящих в α, новым состоянием РП $\mathcal{P}$ является состояние s'.

Множество $\Sigma_{\mathcal{P}}$ можно рассматривать как граф, в котором существует ребро из s в s' с меткой α_{P_i} тогда и только тогда, когда верно (10).

Выполнение РП $\mathcal{P}$ представляет собой последовательность состояний $s_0, s_1, \ldots$ такой, что $s_0 = 0_{\mathcal{P}}$, и каждая пара s_i, s_{i+1} соседних состояний в этой последовательности является переходом в $\mathcal{P}$.

Состояние s РП $\mathcal{P}$ называется **достижимым**, если существует путь из $0_{\mathcal{P}}$ в s. Ниже $\Sigma_{\mathcal{P}}$ обозначает множество достижимых состояний $\mathcal{P}$.

4.4. Метод построения модели MPI-программы в виде распределенного процесса

Будем рассматривать только такие MPI-программы, которые содержат

- операторы присваивания, условного перехода, цикла,
- посылки и приема сообщений, определенные в пункте 2.2, и
- служебные MPI-функции (`MPI_Init` и т.п.), упомянутые в программе из пункта 3.2.

Если с MPI-программой П связаны множество $X_{\text{П}}$ входных переменных и начальное условие $\varphi_{\text{П}} \in \mathcal{B}$, то РП $\mathcal{P}_{\text{П}}$, являющийся моделью данной программы, имеет вид

$$\mathcal{P}_{\text{П}} = \{P_i \mid i = 0, 1, \ldots\}, \tag{11}$$

где $\forall\, i = 0, 1, \ldots$ ПП P_i строится следующим образом.

- Множество $\mathcal{X}_{P_i}$ состоит из переменных, входящих в $X_{\text{П}}$, и i-х копий внутренних переменных, входящих в П, $X_{P_i} = X_{\text{П}}$, $\varphi_{P_i} = \varphi_{\text{П}}$.
- Граф P_i строится путем
  - замены в программе П переменной, являющейся вторым аргументом функции `MPI_Comm_rank` (в MPI-программе, приведенной в пункте 3.2, это переменная `rank`) на константу i,
  - удаления неисполняемых частей получившейся программы, и
  - преобразования получившейся программы в графовую форму, аналогично тому, как по программе в операторной форме строится представление этой программы в виде блок-схемы (с тем отличием, что в блок-схемах действия связаны с вершинами, а в нашей модели они связаны с ребрами).

    Присваивания, условные операторы, операторы цикла преобразуются в действия ПП P_i стандартным образом, эти преобразования м.б. поняты из рассматриваемого ниже примера построения модели MPI-программы из пункта 3.2.

Используемые в П функции передачи сообщений представляются в графе P_i следующими действиями:

- функция (1) представляется действием $c_r!(e, i, l)$, где
  - e – терм, значение которого должно быть равно содержимому участка памяти, пересылаемому этой функцией,
  - r и l – соответствующие аргументы функции (1) (номер получателя и тег, соответственно),
- функция (2) представляется в P_i действием $c_i?(e, s, l)$, где
  - e – терм, значение которого после выполнения данного действия должно быть равно принятому сообщению,
  - s и l – новые переменные,

  и если последний аргумент в функции (2) имеет имя q, то выражения в P_i вида q`.MPI_SOURCE` и q`.MPI_TAG` заменяются на s и l, соответственно,

- представление функции (3) зависит от i:
  - при $i = 0$ функция (3) представляется действием $\circ!e$, где e – терм, значение которого должно быть равно содержимому участка памяти, пересылаемому этой функцией,
  - при $i \neq 0$ функция (3) представляется действием $\circ?e$, где e – терм, значение которого после выполнения данного действия должно быть равно принятому сообщению.

К построенному графу P_i можно применить операцию редукции, описанную в пункте 4.2.5.

4.5. Вспомогательные переменные

Для облегчения анализа РП $\mathcal{P}_\Pi$ можно добавить к действиям ПП, входящим в этот РП, присваивания вида $\iota := e$, в которых

- ι – новая переменная (называемая **вспомогательной** переменной),
- $e \in \mathcal{E}(\mathcal{X}_{\mathcal{P}_\Pi} \sqcup \mathcal{I})$, где $\mathcal{I}$ – множество вспомогательных переменных.

Присваивания указанного выше вида $\iota := e$ не являются реально выполняемыми действиями, они предназначены лишь для выражения зависимостей между значениями переменных во время выполнения РП. В некоторых случаях использование вспомогательных переменных позволяет компактно выразить свойства анализируемого РП и существенно упростить процедуру его анализа.

Последовательные процессы, получаемые из тех ПП, которые входят в РП $\mathcal{P}_\Pi$, добавлением присваиваний вида $\iota := e$, где ι – вспомогательная переменная, будем называть **дополненными** ПП.

5. Моделирование и верификация MPI-программы умножения матриц

В этом пункте мы рассмотрим пример применения описанных выше понятий для моделирования и верификации MPI-программы умножения матриц, представленной в пункте 3.1.

5.1. Построение модели MPI-программы умножения матриц

В этом пункте мы определяем РП $\mathcal{P}_\Pi = \{P_i \mid i = 0, 1, \ldots\}$, моделирующий MPI-программу П умножения матриц в пункте 3.1. Входными переменными П являются A, B (матрицы-сомножители), N (число строк в A), `nprocs`.

5.1.1. Упрощения

$\forall\, i \geq 0$ при построении ПП P_i используются следующие упрощения:

- оператор ввода матриц-сомножителей `input(a, b)` в строке 15 выполняется один раз в начальный момент выполнения процесса с рангом 0, поэтому в модели ПП P_0 можно опустить действия, соответствующие этому оператору, считая, что X_{P_0} содержит переменные A и B, значения которых равны матрицам-сомножителям,
- операторы КПС в P_0 в строках 16-17 и в P_i ($\forall\, i \geq 1$) в строках 44-45 выполняются один раз, в начальный момент выполнения этих ПП, поэтому можно заменить соответствующие действия вида $\circ!e$ и $\circ?e$ на предположение о том, что B входит в X_{P_i} ($\forall\, i \geq 1$).

Для сокращения обозначений в излагаемых ниже ПП вместо входной переменной `nprocs` будем использовать входную переменную n, значение которой равно `nprocs` – 1.

5.1.2. Последовательный процесс P_0

Для построения ПП P_0 используется только часть MPI-программы, расположенная в строках 12-39.

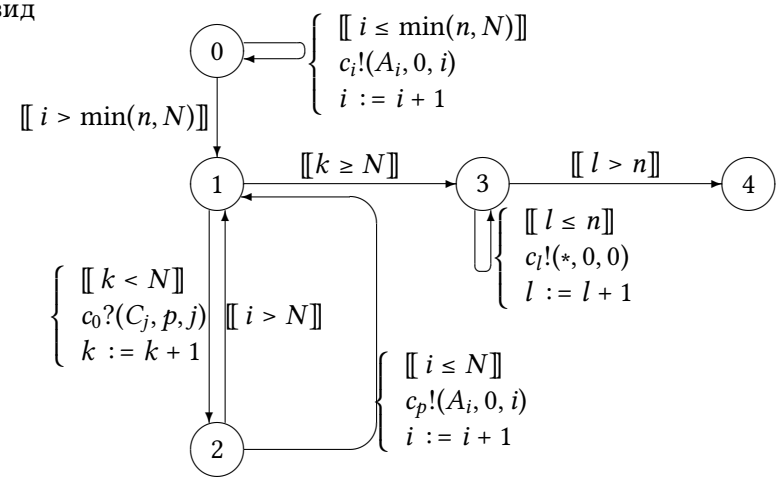
В ПП P_0 A и C являются именами массивов, компоненты которых являются строками соответствующих матриц и индексированы числами $1, \ldots, N$, i–е компоненты этих массивов обозначаются A_i и C_i.

P_0 имеет следующие переменные:

$$X_{P_0} = \{A, B, N, n\},\ \hat{X}_{P_0} = \{C, i, j, k, l, p\}.$$

Начальное условие: $\varphi_{P_0} = (N \geq 1) \wedge (i = 1) \wedge (k = 0) \wedge (l = 1)$.

ПП P_0 имеет вид



(12)

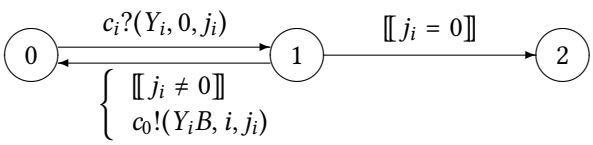
В ПП (12)

- – ребро $P_0^{0\to 0}$ соответствует циклу в строках 18-22 программы, переменная i в метке этого ребра соответствует выражению `count + 1` в программе,
  – сообщение, посылаемое функцией `MPI_Send` в строках 21-22 программы, представлено тройкой $(A_i, 0, i)$, третья компонента которой – тег этого сообщения (т.е. номер соответствующей строки в матрице A),
  – ребро $P_0^{0\to 1}$ соответствует выходу из этого цикла,
- – рёбра $P_0^{1\to 2}$ и $P_0^{2\to 1}$ соответствуют циклу в строках 23-34 программы,
  – переменная k соответствует переменной `i` в этом цикле,
  – оператор `MPI_Recv` в строках 24-26 и цикл в строках 27-28 программы заменены на единое действие: прием сообщения и его запись в соответствующую строку матрицы C,
  – ребро $P_0^{1\to 3}$ соответствует выходу из этого цикла,
- – ребра $P_0^{3\to 3}$ и $P_0^{3\to 4}$ соответствуют циклу в строках 35-37,
  – символ $*$ в метке ребра $P_0^{3\to 3}$ изображает пустую строку.

5.1.3. Последовательный процесс P_i для $i > 0$

Для построения ПП P_i, где $i \geq 1$, используется только часть MPI-программы, расположенная в строках 42-55.

P_i имеет следующие переменные: $X_{P_i} = \{B\}, \hat{X}_{P_i} = \{Y_i, j_i\}$, где значением B является вторая матрица-сомножитель, и значениями Y_i – строки действительных чисел. ПП P_i имеет следующий вид:



(13)

5.2. Верификация распределенного процесса, моделирующего MPI-программу умножения матриц

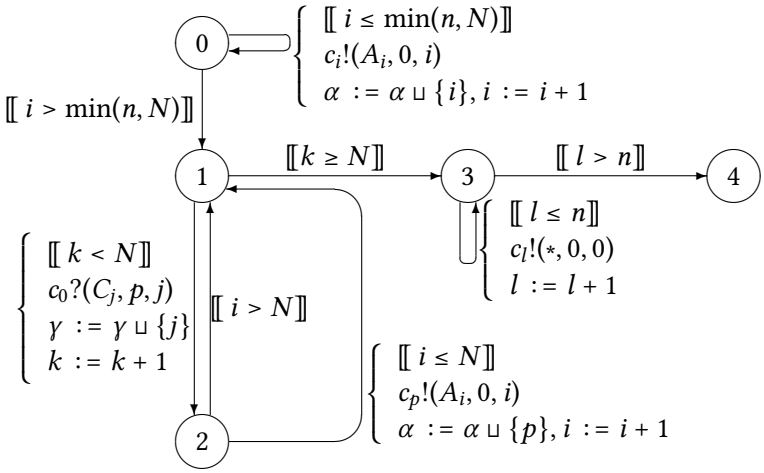
5.2.1. Вспомогательные переменные

Для доказательства утверждения о том, что определенный в пункте 5.1 РП $\mathcal{P}_\Pi$, моделирующий MPI-программу П умножения матриц в пункте 3.1, удовлетворяет своей спецификации, выражаемой свойством (4), мы введем вспомогательные переменные α, β, γ (и связанные с ними действия, не влияющие на выполнение РП), значения которых будут иметь следующий смысл: $\forall\, s \in \Sigma_{P_\Pi}$

- $\alpha^s \subseteq \{1, \ldots, n\}$, α^s состоит из номеров каналов из $\{c_1, \ldots, c_n\}$ с непустым содержимым в состоянии s,
- $\beta^s \subseteq \{1, \ldots, N\}$, β^s состоит из номеров строк матрицы A, для которых в состоянии s вычисляется их произведение на B,
- $\gamma^s \subseteq \{1, \ldots, N\}$, γ^s равно множеству номеров всех строк, которые P_0 записал в C до момента прихода РП $\mathcal{P}_\Pi$ в состояние s.

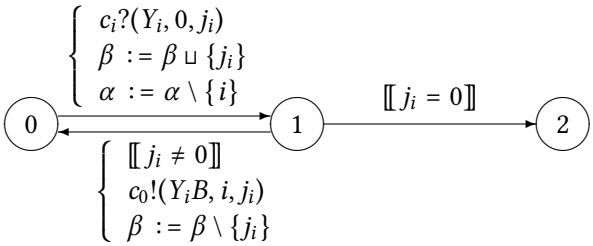
Начальные значения α, β и γ равны $\emptyset$.

Дополненный ПП P_0 имеет вид



(14)

$\forall\, i = 1, \ldots, n$ дополненный ПП P_i имеет вид



(15)

Использование в (14) и (15) символа $\sqcup$ для операций объединения множеств выражает утверждение (вытекающего из нижеследующей теоремы 1) о том, что всякий раз при выполнении этих операций их аргументы в действительности являются дизъюнктными множествами.

5.2.2. Теоремы, обосновывающие корректность $\mathcal{P}_\Pi$

Теорема 1.

$\forall\, s \in \Sigma_{\mathcal{P}_\Pi}$, если $at^s_{P_0} \neq 3, 4$, то верны следующие утверждения:

1. $\alpha^s \subseteq \{1, \ldots, i^s - 1\}$,
2. $i^s - 1 \leq N$
3. $|\gamma^s| = k^s \leq N$,
4. если $at^s_{P_0} = 1$ и $k^s < N$, то $k^s < i^s - 1$,
5. $[c_0]^s_2 \cap \alpha^s = \emptyset$,
6. $\forall\, i = 1, \ldots, n$
   (a) $|[c_i]^s| = \begin{cases} 1, \text{ если } i \in \alpha^s, \text{ и} \\ 0 \text{ иначе,} \end{cases}$
   (b) $at^s_{P_i} = 1 \implies (i \notin \alpha^s) \wedge (j^s_i \notin \gamma^s)$
7. $at^s_{P_0} = 2 \implies p^s \notin \alpha^s, p^s \in \{1, \ldots, i^s - 1\}$,

8. $[c_1]_3^s \sqcup \ldots \sqcup [c_n]_3^s \sqcup \beta^s \sqcup [c_0]_3^s \sqcup \gamma^s = \{1, \ldots, i^s - 1\}$,
9. $\forall\, p \in \alpha^s \;\; [c_p]^s$ имеет вид $\{(A_i, 0, i)\}$, где $i \in \{1, \ldots, N\}$,
10. каждый элемент $[c_0]^s$ имеет вид $(A_i B, p, i)$, где $i \in \{1, \ldots, N\}$,
11. $\forall\, j \in \gamma^s \;\; C_j = A_j B$,
12. $k^s = N \implies \gamma^s = \{1, \ldots, N\}$.

Доказательство.

Истинность всех вышеперечисленных утверждений обосновывается индуктивно: все они верны в начальном состоянии $0_{\mathcal{P}_\Pi}$, и сохраняют свою истинность после каждого перехода $s \longrightarrow s'$ РП $\mathcal{P}_\Pi$, где $at_{P_0}^{s'} \neq 3, 4$. ■

Теорема 2.

В $\Sigma_{\mathcal{P}_\Pi}$ нет тупиковых состояний.

Доказательство.

Пусть $\Sigma_{\mathcal{P}_\Pi}$ содержит тупиковое состояние s. Нетрудно доказать, что $at_{P_0}^s$ не м.б. равно 0, 2, 3, и $\forall\, i = 1, \ldots, n$ $at_{P_i}^s$ не м.б. равно 1. Таким образом, для $at_{P_0}^s$ есть два возможных значения: 1 и 4.

1. Пусть $at_{P_0}^s = 1$. Из тупиковости s следует, что $k^s < N$, откуда опять используя тупиковость s получаем: $[c_0]^s = \varnothing$.

   Из утверждения 4 в теореме 1 следует, что $k^s < i^s - 1$, откуда на основании утверждения 8 теоремы 1 и равенства $[c_0]^s = \varnothing$ получаем:

   $$[c_1]_3^s \sqcup \ldots \sqcup [c_n]_3^s \sqcup \beta^s \neq \varnothing. \tag{16}$$

   Если $\exists\, i \in \{1, \ldots, n\}$: $[c_i]^s \neq \varnothing$, то из тупиковости s следует, что $at_{P_i}^s \neq 0$, поэтому $at_{P_i}^s = 2$, откуда нетрудно получить, что $at_{P_0}^s = 4$. Однако это возможно лишь в случае $k^s \geq N$, что противоречит установленному выше неравенству $k^s < N$.

   Поэтому $\forall\, i \in \{1, \ldots, n\}$ $[c_i]^s = \varnothing$, и из (16) следует, что $\beta \neq \varnothing$. Однако анализом ПП P_i ($i = 0, \ldots, n$) нетрудно установить, что это возможно только если $\exists\, i \in \{1, \ldots, n\}$: $at_{P_i}^s = 2$. Как было установлено в предыдущем абзаце, данное равенство невозможно.
2. Пусть $at_{P_0}^s = 4$. Тогда $k^s \geq N$, откуда следует, что $|\gamma^s| = k^s = N$.

   Пусть s' – первое состояние на пути π из $0_{\mathcal{P}_\Pi}$ в s, такое, что $k^{s'} = N$. Нетрудно видеть, что $at_{P_0}^{s'} = 2$. Из утверждений 2 и 8 теоремы 1 следует, что $i^{s'} - 1 = N$ и

   $$[c_0]^{s'} = [c_1]^{s'} = \ldots = [c_n]^{s'} = \beta^{s'} = \varnothing,$$
   $$at_{P_1}^{s'} = 0, \ldots, at_{P_n}^{s'} = 0.$$

   Единственный переход из s' соответствует действию $[\![i > N]\!]$ и имеет вид $P_0^{2 \to 1} : s' \longrightarrow s''$, единственный переход из s'' соответствует действию $[\![k \geq N]\!]$ и имеет вид $P_0^{1 \to 3} : s'' \longrightarrow s'''$. Нетрудно видеть, что все состояния в хвосте пути π, начинающемся с s''', не являются тупиковыми.

В обоих случаях мы получили противоречие, которое является следствием предположения о том, что в $\Sigma_{\mathcal{P}_\Pi}$ содержит тупиковое состояние. Следовательно, в $\Sigma_{\mathcal{P}_\Pi}$ нет тупиковых состояний. ■

Теорема 3.

Любое выполнение определенного выше РП $\mathcal{P}$ завершается после конечного числа шагов.

Доказательство.

Пусть существует бесконечное выполнение π РП $\mathcal{P}$.

Докажем, что число переходов в π, соответствующих действиям ПП P_0, конечно. Если бы число таких переходов было бесконечным, то

- в π нет состояний s, обладающих свойством $at^s_{P_0} = 3$,
- существует состояние $s_1 \in \pi$, такое, что $at^{s_1}_{P_0} = 1$,
- каждый переход в π, начиная с s_1, соответствующий какому-либо действию P_0, имеет вид либо $P_0^{1\to 2} : s \longrightarrow s'$, либо $P_0^{2\to 1} : s \longrightarrow s'$, и число переходов вида $P_0^{1\to 2} : s \longrightarrow s'$ бесконечно, что невозможно по причине того, что при каждом таком переходе увеличивается значение переменной k, которое, согласно утверждению 3 теоремы 1, ограничено сверху значением N.

Пусть $s' \in \pi$ – состояние, начиная с которого π не содержит переходов, соответствующих действиям P_0, и π' – часть пути π, начинающаяся с s'. Нетрудно видеть, что

$$\exists\, i \in \{1, \ldots, n\}\colon \pi' \text{ содержит бесконечно много переходов вида } P_i^{0\to 1} : s \longrightarrow s'. \tag{17}$$

Т. к. π' не содержит переходов, соответствующих действиям P_0, то значение $|[c_i]^s|$ не может увеличиваться, и согласно (17) оно бесконечно уменьшается, что невозможно. ■

Из доказанных теорем следует, что каждое выполнение РП $\mathcal{P}_\Pi$ является конечным и завершается в некотором терминальном состоянии s. Из $at^s_{P_0} = 4$ следует, что $|\gamma^s| = k^s = N$, откуда согласно утверждениям 11 и 12 теоремы 1 следует, что РП $\mathcal{P}_\Pi$ удовлетворяет спецификации, изложенной в пункте 3.4.

6. Заключение

В настоящей работе мы представили новую математическую модель параллельных программ, предложили подход к верификации таких программ и рассмотрели пример верификации MPI-программы умножения матриц на основе предложенной модели. Основное преимущество предложенного подхода – возможность его применения для программ, генерирующих неограниченный набор последовательных процессов.

Проблемы для дальнейших исследований, связанных с предлагаемой моделью, могут быть следующими:

- расширить предложенную модель, введя концепции для моделирования синхронной передачи сообщений и другие механизмы для организации параллельного выполнения,
- ввести язык спецификации свойств распределенных процессов, на котором свойства параллельных программ выражаются в терминах наблюдаемой эквивалентности, и разработать алгоритм для распознавания наблюдаемой эквивалентности распределенных процессов.

References

[1] G. J. Holzmann, *Design and Validation of Computer Protocols*. Prentice Hall, 1990.

[2] H. A. Lopez, E. R. Marques, F. Martins, N. Ng, C. Santos, V. T. Vasconcelos, and N. Yoshida, “Protocol-based verification of message-passing parallel programs”, in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA 2015*, 2015, pp. 280–298.

[3] I. Grudenic and N. Bogunovic, “Modeling and Verification of MPI Based Distributed Software”, in *Recent Advances in Parallel Virtual Machine and Message Passing Interface. EuroPVM/MPI 2006*, ser. LNCS, vol. 4192, Springer, 2006, pp. 123–132.

[4] A. Blass and Y. Gurevich, “Abstract State Machines Capture Parallel Algorithms”, in *ACM Transactions on Computational Logic*, 2003, pp. 578–651.

[5] S. F. Siegel, “Model Checking Nonblocking MPI Programs”, in *International Workshop on Verification, Model Checking, and Abstract Interpretation, VMCAI*, ser. LNCS, vol. 4349, Springer, 2007, pp. 44–58.

[6] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke, “Combining symbolic execution with model checking to verify parallel numerical programs”, *ACM Transactions on Software Engineering and Methodology*, vol. 17, no. 2, pp. 1–34, 2008.

[7] S. Vakkalanka, G. Gopalakrishnan, and R. M. Kirby, “Dynamic Verification of MPI Programs with Reductions in Presence of Split Operations and Relaxed Orderings”, in *International Conference on Computer Aided Verification CAV 2008*, ser. LNCS, vol. 5123, Springer, 2008, pp. 66–79.

[8] G. Gopalakrishnan, R. M. Kirby, S. Siegel, R. Thakur, W. Gropp, E. Lusk, B. R. De Supinski, M. Schulz, and G. Bronevetsky, “Formal analysis of MPI-based parallel programs”, *Communications ACM*, vol. 54, no. 12, pp. 82–91, 2011.

[9] V. Forejt, S. Joshi, D. Kroening, G. Narayanaswamy, and S. Sharma, “Precise Predictive Analysis for Discovering Communication Deadlocks in MPI Programs”, *ACM Transactions on Programming Languages and Systems*, vol. 39, no. 4, pp. 1–27, 2017.

[10] Z. Luo, M. Zheng, and S. F. Siegel, “Verification of MPI programs using CIVL”, in *EuroMPI*, 2017, 6:1–6:11.

[11] W. Hong, Z. Chen, H. Yu, and J. Wang, “Evaluation of model checkers by verifying message passing programs”, in *Science China Information Sciences*, vol. 62, 2019.

[12] H. Yu, Z. Chen, X. Fu, J. Wang, Z. Su, J. Sun, C. Huang, and W. Dong, “Symbolic Verification of Message Passing Interface Programs”, in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE ’20*, 2020, pp. 1248–1260.

MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 28, NO. 4, 2021
journal homepage: www.mais-journal.ru

THEORY OF COMPUTING

A Recursive Inclusion Checker for Recursively Defined Subtypes

H. de Nivelle[1]

DOI: 10.18255/1818-1015-2021-4-414-433

[1]School of Engineering and Digital Sciences, Nazarbayev University, 53 Kabanbay Batyr str., Nur-Sultan 010000, Kazakhstan.

MSC2020: 68Q99,03B70
Research article
Full text in English

Received November 15, 2021
After revision December 1, 2021
Accepted December 8, 2021

We present a tableaux procedure that checks logical relations between recursively defined subtypes of recursively defined types and apply this procedure to the problem of resolving ambiguous names in a programming language. This work is part of a project to design a new programming language suitable for efficient implementation of logic. Logical formulas are tree-like structures with many constructors having different arities and argument types. Algorithms that use these structures must perform case analysis on the constructors, and access subtrees whose type and existence depend on the constructor used. In many programming languages, case analysis is handled by matching, but we want to take a different approach, based on recursively defined subtypes. Instead of matching a tree against different constructors, we will classify it by using a set of disjoint subtypes. Subtypes are more general than structural forms based on constructors, we expect that they can be implemented more efficiently, and in addition can be used in static type checking. This makes it possible to use recursively defined subtypes as preconditions or postconditions of functions. We define the types and the subtypes (which we will call adjectives), define their semantics, and give a tableaux-based inclusion checker for adjectives. We show how to use this inclusion checker for resolving ambiguous field references in declarations of adjectives. The same procedure can be used for resolving ambiguous function calls.

Keywords: programming language design; type systems; theorem proving; compiler construction

INFORMATION ABOUT THE AUTHORS

Hans de Nivelle
correspondence author

orcid.org/0000-0001-9343-6679. E-mail: hans.denivelle@nu.edu.kz
Associate Professor, Dr. Habil.

Funding: Faculty Development Competitive Research Grant Program (FDCRGP) 021220FD1651.

For citation: H. de Nivelle, "A Recursive Inclusion Checker for Recursively Defined Subtypes", *Modeling and analysis of information systems*, vol. 28, no. 4, pp. 414-433, 2021.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 28, № 4, 2021
сайт журнала: www.mais-journal.ru

THEORY OF COMPUTING

Рекурсивная проверка включения для рекурсивно определенных подтипов

Х. де Нивелле[1]

DOI: 10.18255/1818-1015-2021-4-414-433

[1]Школа инженерии и цифрових наук, Назарбаев Университет, ул. Кабанбай Батыра, д. 53, г. Нур-Султан, 010000 Казахстан.

УДК 510.57
Научная статья
Полный текст на английском языке

Получена 15 ноября 2021 г.
После доработки 1 декабря 2021 г.
Принята к публикации 8 декабря 2021 г.

Мы представляем табличную процедуру, которая проверяет логические отношения между рекурсивно определенными подтипами рекурсивно определенных типов, и применяем эту процедуру к проблеме разрешения неоднозначных имен в языке программирования. Эта работа является частью проекта по разработке нового языка программирования, подходящего для эффективной реализации логики. Логические формулы представляют собой древовидные структуры со множеством конструкторов, имеющих различные свойства и типы аргументов. Алгоритмы, использующие эти структуры, должны выполнять анализ вариантов для конструкторов и получать доступ к поддеревьям, тип и существование которых зависят от используемого конструктора. Во многих языках программирования анализ прецедентов обрабатывается путем сопоставления, но мы хотим использовать другой подход, основанный на рекурсивно определенных подтипах. Вместо сопоставления дерева с различными конструкторами мы будем классифицировать его с помощью набора непересекающихся подтипов.
Подтипы являются более общими, чем структурные формы, основанные на конструкторах, мы ожидаем, что они могут быть реализованы более эффективно и, кроме того, могут использоваться при статической проверке типов. Это позволяет использовать рекурсивно определенные подтипы в качестве предварительных условий или постусловий функций. Мы определяем типы и подтипы (которые мы будем называть прилагательными), определяем их семантику и даем проверку включения прилагательных на основе таблиц. Мы показываем, как использовать эту проверку включения для разрешения неоднозначных ссылок на поля в объявлениях прилагательных. Та же процедура может быть использована для разрешения неоднозначных вызовов функций.

Ключевые слова: проектирование языков программирования; системы типов; доказательство теорем; построение компилятора

ИНФОРМАЦИЯ ОБ АВТОРАХ

Ханс де Нивелле
автор для корреспонденции

orcid.org/0000-0001-9343-6679. E-mail: hans.denivelle@nu.edu.kz
Ассоциированный профессор, доктор физ.-мат. наук.

Финансирование: Программа поддержки исследовательской деятельности профессорско-преподавательского состава 021220FD1651.

Для цитирования: H. de Nivelle, “A Recursive Inclusion Checker for Recursively Defined Subtypes”, *Modeling and analysis of information systems*, vol. 28, no. 4, pp. 414-433, 2021.

1. Introduction

Our goal is to develop a programming language for implementation of logic that is convenient to use on one hand, and efficient on the other hand. Traditionally, functional languages like OCaml ([1]), Haskell ([2]) or Scala ([3]) are considered the most suitable for implementation of logic.

Functional languages have inductive types, and use matching for accessing subtrees. Matching can be viewed as simultaneously inspecting a tree and extracting subtrees into local variables. In our proposed language we want to keep inductive types, but add a mechanism for definition of subtypes, which we will use both as a replacement for matching, and as a basis for static overload resolution. The subtypes make it possible to replace matching by usual field access as used in imperative languages like Java or C^{++}. We believe that field access will be more efficient than matching.

We want our language also to support static overloading of function names. In C, defined functions always must have distinct names. In C^{++} and Java, the same name can be reused for different argument types. The compiler will select the definition that fits best to the arguments. This process is called *overload resolution*. In this paper, we study overload resolution only for field names (which is already present in C). The same technique is applicable to function calls.

We want some imperative features, because efficient run time handling of containers (also called *collections*) is difficult in functional setting. In order to obtain this, data must have efficient low level representation in memory. We allow a restricted form of assignment, only to top level variables and substructures at nesting depth one. In this way, side effects due to sharing can be avoided.

We will start by explaining the subtype system. At run time, it is used for inspecting data (formulas) so that we know which fields exist. The advantage of using subtypes for run time classification is that, once one has the mechanism for defining and verifying them, they are more general than structure matching, and can be used at many other points in the programming language, also at compile time.

As an example of our approach, suppose one has a propositional formula f, and one wants to know if it constructed by **and**. One can either match it into **and**(F_1, F_2), or define a subtype 'constructed by **and**', and define two fields f_1, f_2 that can be accessed only on formulas refined by the subtype 'constructed by **and**'. The advantage of subtypes is that they can be easily combined by Boolean operators. For example, it is easy to define the subtype 'constructed by **and** or **or**', or 'atom' and use this as a condition in case analysis.

We will call the subtypes *adjectives*. In addition to adjectives that correspond to simple structural matchings, our adjective system also allows recursively defined subtypes. This makes it possible to define for example negation normal form or conjunctive normal form as a subtype of formulas. Although such recursively defined subtypes can be used for run time classification, they are not intended for this purpose, because evaluating them at run time would be costly. They are intended to be used for overload resolution at compile time. If one restricts oneself to adjectives of constant depth (like the ones above), they can be efficiently checked in constant time at run time.

Adjectives are somewhat similar to liquid types ([4]) or refinement types ([5]) but they have a different aim. Adjectives are not intended for checking operations whose safety depends on arithmetic. Because of this, adjectives do not need SMT solving. We prefer the term *adjective* over *subtype* or *refinement type*, because adjectives can overlap, and are frequently created for one time use. We also want to use adjectives for static type checking and overload resolution. In order to show that this is possible, we define a procedure that resolves ambiguous field references in the adjectives themselves.

We start by defining the type and adjective system in this section. In Section 2 we give the semantics, which can be used for evaluating adjectives ondata, so that they can be used as a replacement for matching. In Section 3, we give a tableaux-based procedure for deciding satisfiability of adjectives. This procedure can be used for checking exhaustiveness of a given case analysis, for checking exclusiveness of cases, and for static type checking. In Section 4 we will apply the tableaux procedure on the problem of resolving

ambiguous field references in the adjective declarations themselves. A similar algorithm could be used for resolving ambiguous function calls. We define primitive types:

Definition 1.1. *The primitive types are* bool, char, nat, double, *and* selector. *We assume that all primitive types, with the exception of* selector, *have an order* $<$ *defined on them.*

Type selector is a global enumeration type that consists of named constants. It is intended for defining subtypes (see Definition 1.2 below), or representing logical operators. Actually, these uses are the same because the logical operator of a formula defines a subtype at the same time. Enumeration types in Rust ([6]) play a similar role. We write elements of selector as identifiers preceded by a question mark, e.g. `?and`,`?or`,`?implies`,`?equiv`, or `?zero`,`?succ`. We give examples of their use below, after the definition of the type system.

Next we define non-primitive types, and after that follow adjectives. Strictly seen, the definition of adjectives should come first, because types contain adjectives while adjectives do not contain types. We find, however, that this order would be unintuitive, because it would be hard to understand the usefulness of adjectives, without having seen the definition of types. Because of this, we start by introducing types:

Definition 1.2. Simple types *are recursively defined as follows:*

- *A primitive type is a simple type.*
- *An identifier is a simple type (assuming that it has been defined as type, see Definition 1.4 below).*
- *If T is a simple type, A is an adjective, then $T{\circ}A$ is also a simple type.*

We recursively define compound types*:*

1. *If $v_1, \ldots, v_n$ are identifiers, $V_1, \ldots, V_n$ are simple types with $n \geq 0$, then $(\, v_1 \colon V_1, \ldots, v_n \colon V_n \,)^*$ is a compound type.*
2. *If C is a compound type, v is an identifier and V is a simple type, then $v \colon V,\ C$ is a compound type as well.*
3. *If $C_1, \ldots, C_m$ with $m \geq 1$ are compound types, $A_1, \ldots, A_m$ are adjectives, s is an identifier, then $s?(A_1 \Longrightarrow C_1, \ldots, A_m \Longrightarrow C_m)$ is a compound type as well.*

In case 1, the fields $v_1, \ldots, v_n$ are called repeated *fields. All other fields defined in cases 2 and 3 are called* scalar fields*. In case 3, idenifier s (which must refer to an earlier defined field) is called the* pivot *field. Case 3 defines compound types where the existence of later fields depends on the adjective that applies to s.*

We distinguish between simple and compound types, because we want every compound type to have a name, which solves some technical problems. It has no consequences for expressiveness, because one can always define a compound type, and use the name as simple type. This can be automatically done by the compiler, if needed. The meaning of $T{\circ}A$ is: Type T refined by adjective A.

Compound types have a tree structure, where every path is a possible realization of the type. The tree branches whenever case 3 is used. In that case, identifier s should refer to a field that occurs before it on the same path. We cannot impose this in Definition 1.2, because trees are constructed bottom up, and the field s must be defined above it in an application of case 2. We give a simple example of a compound type:

Example 1.3.

$$T = \quad s\colon \text{selector}{\circ}(?\text{simp}|?\text{vect}),\ s?(\ ?\text{simp} \Longrightarrow a\colon A,\ b\colon B,\ ()^*,\qquad ?\text{vect} \Longrightarrow c\colon C,\ (d\colon D,\ e\colon E\,)^*\ \ )$$

Example 1.3 is purely technical. Type T is not intended to be meaningful. Also note that Definition 1.2 is awkward to use when defining types in practice because it is intended as technical representation. It is not intended as the syntax with which the programmer will define types. The first field s is a selector, which must be either ?simp or ?vect. In case it is ?simp, there are two more scalar fields a of type A, and b of type B. In case s equals ?vect, there is one more scalar field c of type C, and an unbounded number of repeated fields

d of type D and e of type E. They must be accessed with array notation $d[i]$ or $e[i]$. They are implemented as vectors in C^{++}, which means that the repeated part can dynamically grow and shrink, and an object of type T will be reallocated when it runs out of capacity. Because repeated fields are always last, offsets of scalar fields do not depend on the number of repeated fields present.
For the moment, we assume that different fields of different types have distinct names. This is an unrealistic restriction for a real programming language, because field names can be reused in different types. In Section 4 we will allow reuse of field names between different options of a type, and between different types. We will give a procedure that is able to resolve such ambiguous field references into unambiguous references.

Type definitions need to be stored in a mapping. It is convenient to use two distinct mappings, one for the simple types, and one for the compound types:

Definition 1.4. *We define two mappings Σ_S and Σ_C. The first mapping maps identifiers to simple types, and the second mapping maps identifiers to compound types. If Σ_S contains a value for v, we write $\Sigma_S(v)$ for the value. Similarly, if Σ_C contains a value for v, we write $\Sigma_C(v)$ for the value. We assume that Σ_S and Σ_C do not contain a value for the same variable v. We also assume that Σ_S is cycle free.*

An example of Σ_S not being cycle-free would be when $\Sigma_S(v_1) = v_2 \circ A_2$, and $\Sigma_S(v_2) = v_1 \circ A_1$. Cycles inside Σ_C are unproblematic, see for example the definition of prop below in Example 1.10.

This completes the definition of the types, next we define adjectives:

Definition 1.5. *We recursively define adjectives, starting with primitive adjective constructors:*

- *If c is a constant of one of the primitive types* bool, char, nat, double, *then c and $c^{\geq}$ are adjectives.*
- *If c is a constant of type* selector, *then c is an adjective.*
- **empty** *is an adjective.*
- *If v is an identifier, then v is an adjective.*

Next we define recursive adjective constructors:

- *If f is an identifier, and A is an adjective, then $f(A)$ is an adjective.*
- *If A is an adjective, then* **first**(A) *and* **rest**(A) *are adjectives.*
- *If A is an adjective, then $\forall A$ and $\exists A$ are adjectives.*

Adjectives of any type can be combined by propositional operators:

- *If $A_1, \ldots, A_n$ are adjectives, then $A_1 \vee \cdots \vee A_n$, and $A_1 \wedge \cdots \wedge A_n$ are also adjectives.*
- *If A is an adjective, $\neg A$ is also an adjective.*

The intuitive meaning of the adjective c is : equal to c. The intuitive meaning of $c^{\geq}$ is : greater or equal to c. The intuitive meaning of $f(A)$ is: whose f-field satisfies A. If f is a repeated field, then A must be equal to **empty** or have one of the forms $\forall A$, $\exists A$, **first**(A), or **rest**(A). The meaning of **empty** is that no repeated fields are present. The meaning of $f(\forall A)$ is that all repetitions of f must satisfy A. Similarly $f(\exists A)$ means that at least one repetition of f must satisfy A. The meaning of **first**(A) is that $f[1]$ must satisfy A. The meaning of **rest**(A) is that all repetitions of f, except for possibly the first, must satisfy A.

Definition 1.6. *We assume a mapping Σ_A that maps identifiers to pairs of form (T, A), where T is a simple type, and A is an adjective. If Σ_A contains a value for v, we write $\Sigma_A(v)$ for the value.*

The intuitive meaning of $\Sigma_A(v) = (T, A)$ is: Identifier v is defined on type T as adjective A.

Definition 1.7. *We write $A[v]$ for an adjective that contains v somewhere not inside a subformula of form $f(\cdot)$. We define a* cycle *as a sequence of identifiers, s.t.*

$$\Sigma_A(v_1) = (\, T_2, A_2[v_2]\,), \ldots, \Sigma_A(v_{n-1}) = (\, T_n, A_n[v_n]\,),\ \Sigma_A(v_n) = (\, T_1, A_1[v_1]\,).$$

Cycles are logically problematic when they involve negation. Because the intended meaning of an adjective is an inductively defined predicate, an adjective defined through a cycle involving negation, would be ill-defined. This is similar to the situation in logic programming ([7]). Although it would be possible to adapt Definitions 2.2 and 3.8 to monotonic cycles, we are not aware of a meaningful use of it. Therefore, we think it is better to forbid cycles altogether. In the rest of this paper, we will assume that Σ_A is cycle free. We give some examples of type and adjective definitions:

Example 1.8. *The type of complex numbers can be defined as follows:*

$$\text{complex} := \text{re}\colon \text{double},\ \text{im}\colon \text{double},\ (\,)^*.$$

Complex numbers have two scalar fields re *and* im*, and no repeated fields.*

Example 1.9. *Natural numbers can be defined as follows:*

$$\text{nat} := \text{sel}\colon \text{selector}, \text{sel?} \begin{pmatrix} \text{?zero} & \Longrightarrow & (\,)^* \\ \text{?succ} & \Longrightarrow & \text{pred}\colon \text{nat},\ (\,)^* \end{pmatrix}$$

Adjectives odd and even can be defined on nat *in mutual recursion:*

$$\begin{array}{lcl} \text{even} & := & (\ \text{nat},\ \text{sel(?zero)} \vee (\ \text{sel(?succ)} \wedge \text{pred(odd)}\ )\ ) \\ \text{odd} & := & (\ \text{nat},\ \text{sel(?succ)} \wedge \text{pred(even)}\ ) \end{array}$$

Example 1.10. *We define propositional logic:*

$$\text{prop} := \text{op}\colon \text{selector},\ \text{op?} \begin{pmatrix} \text{?var} & \Longrightarrow & (\ c\colon \text{char}\ )^* \\ \text{?not} & \Longrightarrow & \text{sub}\colon \text{prop},\ (\,)^* \\ \text{?implies}\vee\text{?equiv} & \Longrightarrow & \text{sub}_1\colon \text{prop},\ \text{sub}_2\colon \text{prop},\ (\,)^* \\ \text{?and}\vee\text{?or} & \Longrightarrow & (\ \text{sub}_n\colon \text{prop}\ )^* \end{pmatrix}$$

There is no need to define $\top$ or $\bot$ separately, because $\top = \bigwedge \emptyset$, and $\bot = \bigvee \emptyset$. A variable consists of ?var combined with a finite number of characters. A negation consists of ?not combined with a single subformula called sub. A conjunction or disjunction consists of ?and or ?or, followed by an arbitrary number of subformulas called sub_n. We define a few adjectives on prop:

Example 1.11. *'Constructed by* ?and*' can be expressed as* op(?and)*. 'Constructed by ?*and *or ?*or*' can be expressed as* op(?and∨?or)*. The property of being an atom can be expressed by*

$$\text{atom} := (\ \text{prop}, \text{op(var)}\ ).$$

The property of being a literal can be expressed by

$$\text{literal} := (\ \text{prop}, \text{atom} \vee (\ \text{op(?not)} \wedge \text{sub(?atom)}\ )\ ).$$

Negation normal form (NNF) can be expressed by:

$$\text{nnf} := (\ \text{prop}, \bigvee \begin{Bmatrix} \text{literal} \\ \text{op(?and}\vee\text{?or)} \wedge \text{sub}_n(\ \forall\text{nnf}\ ) \end{Bmatrix}\ ).$$

Conjunctive normal form (CNF) can be defined as follows:

$$\text{cnf} := (\ \text{prop}, \text{op(?or)} \wedge \text{sub}_n(\forall(\text{op(?and)} \wedge \text{sub}_n(\ \forall\text{literal}\ )))\ ).$$

Case 3 in Definition 1.2 introduces fields whose existence depends on the adjectives fulfilled by the pivot s. For type prop, the sub field only exists when the op field equals ?not, which can be expressed by the adjective op(not). Similarly, field sub_1 exists only when op(?implies∨?equiv). The following definition specifies how to obtain preconditions from the definition of the compound type.

Definition 1.12. *Let f be a exact identifier that is declared as field in a compound type C. We recursively define the* precondition *of f, written as* $\text{PREC}(C, f)$*, as follows:*

- *if C has form* $(\ v_1 \colon V_1, \ldots, v_n \colon V_n\ )^*$*, then* $\text{PREC}(f, C) = \top$.
- *If C has form $v \colon V,\ C'$, and $f = v$, then* $\text{PREC}(f, C) = \top$*. If f is declared in C', then* $\text{PREC}(f, C) = \text{PREC}(f, C')$.
- *If C has form $s?(A_1 \Longrightarrow C_1, \ldots, A_m \Longrightarrow C_m)$, and $f = s$, then* $\text{PREC}(f, C) = A_1 \vee \cdots \vee A_m$*. Otherwise, f must be declared in exactly one C_j. We define* $\text{PREC}(f, C) = s(A_j) \wedge \text{PREC}(f, C_j)$.

Since f can occur in Σ_C only once, it is possible to write $\text{PREC}(f)$ *instead of* $\text{PREC}(f, C)$*, where C is the compound type that contains f.*

2. Semantics of Types and Adjectives

We will describe the semantics of adjectives and types using a simplified high level representation of data. We assume that data are represented by trees whose subtrees are labeled with field names. This representation is convenient for defining the semantics without having to consider low level details like memory lay out. In the implementation we use a low level representation, where scalar fields have a fixed offset, and repeated fields have an offset that can be computed by multiplying the index with the size of the repeated part, and adding a base offset.

Definition 2.1. *We define the set of data trees D, and the set $\overline{D}$ of finite sequences of data trees, in simultaneous recursion:*

- *If d is an element of one of the primitive types T, defined in Definition 1.1, then $d \in D$.*
- *If $d_1, \ldots, d_n$ $(n \geq 0)$ is a finite sequence of data trees, then $(d_1, \ldots, d_n) \in \overline{D}$. We will write $\|\overline{D}\|$ for the length n of $\overline{D}$.*
- *If $f_1, \ldots, f_n$ are pairwise distinct identifiers, and each $d_i \in D \cup \overline{D}$, then $\{(f_1, d_1), \ldots, (f_n, d_n)\} \in D$.*

Data trees of the third type can be viewed as partial functions mapping each f_i to d_i, such that d_i is either a data tree or a sequence of data trees. When d is of the third type, we write $d.f$ for the value attached to f when it exists. If $d.f \in \overline{D}$, we write $d.f[i]$ for its i-th element, assuming that $i \leq \|d.f\|$.

We will define when a data tree has a given type, and when a data tree (or sequence of data trees) satisfies a given adjective. Types may contain adjectives, but adjectives do not contain types. Therefore, we start by defining when a data tree (or sequence of data trees) makes an adjective true:

Definition 2.2. *We define in simultaneous recursion when a data tree makes an adjective true, and when a sequence of data trees makes an adjective true. We use the notation $d \vDash A$ for both cases.*
We first consider the cases where $d \in D$:

- *If c is a constant, then $d \vDash c$ iff $d = c$, and $d \vDash c^{\geq}$ if $d \geq c$.*
- *If f is an identifier, s.t. $d.f$ exists, then*
  - *if $d.f$ is a single data tree, then $d \vDash f(A)$ iff $d.f \vDash A$.*
  - *if $d.f$ is a sequence of data trees, $d \vDash f(A)$ iff $d.f \vDash A$.*

  The two cases appear to be the same, but $d.f$ have different types, hence we prefer to list them separately.
- *If v is an identifier, s.t. $\Sigma_A(v)$ is defined, and $d \in D$, then $d \vDash v$ iff $d \vDash \Sigma_A(v)$.*

Next we list the cases where $d \in \overline{D}$:

- $(d_1, \ldots, d_n) \vDash$ **empty** *iff* $n = 0$.
- $(d_1, \ldots, d_n) \vDash \forall A$ *iff for every d_i we have $d_i \vDash A$.*

- $(d_1, \ldots, d_n) \vDash \exists A$ *iff there exists a* d_i *such that* $d_i \vDash A$.
- $(d_1, \ldots, d_n) \vDash \mathbf{first}(A)$ *iff* $n \geq 1$ *and* $d_1 \vDash A$.
- $(d_1, \ldots, d_n) \vDash \mathbf{rest}(A)$ *iff* $n \geq 1$ *and* $(d_2, \ldots, d_n) \vDash A$.

The definitions for the propositional connectives are standard, both for $d \in D$ *and* $d \in \overline{D}$:

- $d \vDash A_1 \vee \cdots \vee A_n$ *if there is an* i $(1 \leq i \leq n)$, *s.t.* $d \vDash A_i$.
- $d \vDash A_1 \wedge \cdots \wedge A_n$ *for all* i $(1 \leq i \leq n)$, *one has* $d \vDash A_i$.
- $d \vDash \neg A$ *iff not* $d \vDash A$.

We define when a tree has a certain type:

Definition 2.3. *We recursively define when a data tree* d *has type* T. *We use notation* $d\colon T$. *We first list the cases where* T *is simple:*

- *For a primitive type* T, *we define* $d\colon T$ *as* $d \in T$.
- *If* v *is an identifier that is defined in* Σ_S, *then* $d\colon v$ *iff* $d\colon \Sigma_S(v)$.
- *If* v *is an identifier that is defined in* Σ_C, *then* $d\colon v$ *iff* $d\colon \Sigma_C(v)$.
- $d\colon (T \circ A)$ *iff* $d\colon T$ *and* $d \vDash A$.

Next we list the cases where T *is compound:*

- $d\colon (\ v_1\colon V_1, \ldots, v_n\colon V_n\ )^*$ *iff either*
  - $n = 0$, *or*
  - $n > 0$ *and* $d.v_1, \ldots, d.v_n$ *are all defined, are all in* $\overline{D}$, *have the same length* $L = \|d.v_1\| = \cdots = \|d.v_n\|$, *and for every* i $(1 \leq i \leq n)$ *and* j $(1 \leq j \leq L)$, *we have* $d.v_i[j]\colon V_i$.
- $d\colon (v\colon V,\ C)$ *iff* $d.v$ *is defined,* $d.v\colon V$, *and* $d\colon C$.
- $d\colon s?(A_1 \Longrightarrow C_1, \ldots, A_m \Longrightarrow C_m)$ *iff* $d.s$ *is defined, in* D, *and there is exactly one* $1 \leq i \leq m$, *s.t.* $d.s \vDash A_i$ *and* $d\colon C_i$.

Example 2.4. *Following up on Example 1.8, we can see that* $\{\ (\mathrm{re}, 1.0), (\mathrm{im}, 2.0)\ \}$ *has type* complex.

Example 2.5. *Using the declaration of* prop *in Example 1.10, the propositional formula* p *will be represented by a data tree* d_p *with fields*

$$\begin{array}{lcl} d_p.\mathrm{op} & = & ?\mathrm{atom} \\ d_p.\mathrm{c} & = & (\text{'}p\text{'}) \end{array}$$

The propositional formula q *will be represented by* d_q *with fields*

$$\begin{array}{lcl} d_q.\mathrm{op} & = & ?\mathrm{atom} \\ d_q.\mathrm{c} & = & (\text{'}q\text{'}) \end{array}$$

Similarly, the propositional formula r *will be represented by* d_r *with*

$$\begin{array}{lcl} d_r.\mathrm{op} & = & ?\mathrm{atom} \\ d_r.\mathrm{c} & = & (\text{'}r\text{'}) \end{array}$$

The formula $q \vee r$ *will be represented by* d *with*

$$\begin{array}{lcl} d.\mathrm{op} & = & ?\mathrm{or} \\ d.\mathrm{sub} & = & (d_q, d_r) \end{array}$$

Finally, the formula $p \longrightarrow (q \vee r)$ *will be represented by* d' *with*

$$\begin{array}{lcl} d.\mathrm{op} & = & ?\mathrm{implies} \\ d.\mathrm{sub}_1 & = & d_p \\ d.\mathrm{sub}_2 & = & d' \end{array}$$

3. A Tableaux Calculus

In this section we define a calculus for checking satisfiability of adjectives. This is sufficient to answer all logical questions that are needed for the implementation of our programming language. In order to show that adjectives $A_1, \ldots, A_n$ cover all possible cases in a switch, it is sufficient to show that $A \wedge \neg A_1 \wedge \cdots \wedge \neg A_n$ is unsatisfiable, where A is the adjective part of the type of the switch expression. In order to show that a switch has no overlapping cases, it is sufficient to show that $A \wedge A_i \wedge A_j$ is unsatisfiable for all distinct i and j. In order to show that a function f_1 defined on adjective A_1 is a better fit than function f_2 defined on adjective A_2, it is sufficient to show that $A_1 \wedge \neg A_2$ is unsatisfiable, while $A_2 \wedge \neg A_1$ is satisfiable.

In order to use our calculus, types need to be decomposed into their adjective part, and the part that specifies their implementation. The calculus needs the adjective part of a type, but does not use the implementation. We will define two functions that decompose a simple type into their adjective and implementation components.

Definition 3.1. *An* implementation type *is either a primitive type, or an identifier defined in Σ_C, i.e. the name of a compound type. For a simple type T, we define* $\mathrm{ADJ}(T)$ *and* $\mathrm{IMPL}(T)$ *as follows:*

- *If T is primitive then* $\mathrm{ADJ}(T) = \top$, *and* $\mathrm{IMPL}(T) = t$.
- *If v is an identifier defined in Σ_C, then* $\mathrm{ADJ}(v) = \top$, *and* $\mathrm{IMPL}(v) = v$.
- $\mathrm{ADJ}(T{\circ}A) = \mathrm{ADJ}(T) \wedge A$, *and* $\mathrm{IMPL}(T{\circ}A) = \mathrm{IMPL}(T)$.

For example, if $T = \mathrm{prop}{\circ}\mathrm{nnf}{\circ}\mathrm{cnf}$, then $\mathrm{ADJ}(T) = \mathrm{nnf} \wedge \mathrm{cnf}$, and $\mathrm{IMPL}(T) = \mathrm{prop}$.

Theorem 3.2. *If T is a simple type, and d is a data tree, then $d : T$ iff $d : \mathrm{IMPL}(T)$ and $d \vDash \mathrm{ADJ}(T)$.*

The theorem can be easily verified by applying the rules of Definition 2.2.

At this point, we can define a simple interface of our tableaux procedure. Its input is just a single type T. The procedure establishes that there exists no data term d with type T. The applications that we mentioned in the introduction, can be obtained as follows:

- Establish that there exists no data term d of type T that satisfies adjective A : Call the procedure with $T{\circ}A$.
- Establish that every data term d of type T, that satisfies A, must also satisfy B : Call the procedure with $T{\circ}A{\circ}\neg B$.

The procedure works on the following normal form, which is similar to negation normal form:

Definition 3.3. *We recursively define when an adjective is in* path normal form *(PNF):*

- *If c is a constant of a primitive type, then c, $\neg c$, $c^{\geq}$ and $\neg c^{\geq}$ are in PNF. In the last two cases, c cannot have type* selector.
- **empty** *and* $\neg$**empty** *are in PNF.*
- *If v is an identifier, then v and $\neg v$ are in PNF.*
- *If A is in PNF, then $f(A)$,* **first**(A), **rest**(A), $\forall A$, *and* $\exists A$ *are in PNF.*
- *If $A_1, \ldots, A_n$ $(n \geq 0)$ are in PNF, then $A_1 \vee \cdots \vee A_n$ and $A_1 \wedge \cdots \wedge A_n$ are in PNF.*

Definition 3.4. *An adjective A can be brought into PNF by calling* $\mathrm{PNF}(A, \mathbf{pos})$, *where* $\mathrm{PNF}(A, p)$ *is defined by cases as follows:*
If A has form c, $c^{\geq}$, **empty**, *or v, then*

$$\mathrm{PNF}(A, \mathbf{pos}) = A$$
$$\mathrm{PNF}(A, \mathbf{neg}) = \neg A$$

For the remaining cases:

$$\begin{aligned}
&\mathrm{PNF}(A_1 \vee \cdots \vee A_n, \mathbf{pos}) = \mathrm{PNF}(A_1, \mathbf{pos}) \vee \cdots \vee \mathrm{PNF}(A_n, \mathbf{pos})\\
&\mathrm{PNF}(A_1 \vee \cdots \vee A_n, \mathbf{neg}) = \mathrm{PNF}(A_1, \mathbf{neg}) \wedge \cdots \wedge \mathrm{PNF}(A_n, \mathbf{neg})\\
&\mathrm{PNF}(A_1 \wedge \cdots \wedge A_n, \mathbf{pos}) = \mathrm{PNF}(A_1, \mathbf{pos}) \wedge \cdots \wedge \mathrm{PNF}(A_n, \mathbf{pos})\\
&\mathrm{PNF}(A_1 \wedge \cdots \wedge A_n, \mathbf{neg}) = \mathrm{PNF}(A_1, \mathbf{neg}) \vee \cdots \vee \mathrm{PNF}(A_n, \mathbf{neg})\\
&\mathrm{PNF}(f(A), p) = f(\,\mathrm{PNF}(A, p)\,) \textit{ for } p \in \{\mathbf{pos}, \mathbf{neg}\}\\
&\mathrm{PNF}(\mathbf{first}(A), p) = \mathbf{first}(\,\mathrm{PNF}(A, p)\,) \textit{ for } p \in \{\mathbf{pos}, \mathbf{neg}\}\\
&\mathrm{PNF}(\mathbf{rest}(A), p) = \mathbf{rest}(\,\mathrm{PNF}(A, p)\,) \textit{ for } p \in \{\mathbf{pos}, \mathbf{neg}\}\\
&\mathrm{PNF}(\exists A, \mathbf{pos}) = \exists\,\mathrm{PNF}(A, \mathbf{pos})\\
&\mathrm{PNF}(\exists A, \mathbf{neg}) = \forall\,\mathrm{PNF}(A, \mathbf{neg})\\
&\mathrm{PNF}(\forall A, \mathbf{pos}) = \forall\,\mathrm{PNF}(A, \mathbf{pos})\\
&\mathrm{PNF}(\forall A, \mathbf{neg}) = \exists\,\mathrm{PNF}(A, \mathbf{neg})\\
&\mathrm{PNF}(\neg A, \mathbf{pos}) = \mathrm{PNF}(A, \mathbf{neg})\\
&\mathrm{PNF}(\neg A, \mathbf{neg}) = \mathrm{PNF}(A, \mathbf{pos})
\end{aligned}$$

The following theorem can be straightforwardly proven, using Definition 2.2:

Theorem 3.5. *Let A be an adjective. For every data tree d, we have $d \models A$ iff $d \models \mathrm{PNF}(A, \mathbf{pos})$. Similarly, $d \models \neg A$ iff $d \models \mathrm{PNF}(A, \mathbf{neg})$.*

The path normal form of $\neg(A \vee f(\mathbf{rest}(\forall \neg B)))$ equals $\neg A \wedge f(\mathbf{rest}(\exists B))$.

Definition 3.6. *We define* a path *as a finite sequence $(f_1, \ldots, f_n)$ with $(n \geq 0)$, where each f_i is either an identifier (representing a field name), or an element of $\{\exists, \forall, \mathbf{first}, \mathbf{rest}\}$. We write ϵ for the empty path, and use notation $\pi.f$ for extending path π with f. If path π' can be obtained from path π by zero or more extensions, we call π a prefix of π'. We call π a* strict prefix of π' *if at least one extension was used.*

We assume that there exists a total order $<$ on paths with the property that if π_1 is a strict prefix of π_2, then $\pi_1 < \pi_2$. Such an order can be easily obtained by fixing a total order on all possible f_i, and using the alphabetic, lexicographic extension.

Definition 3.7. *An* adjective stack *S is a finite sequence of triples (π_i, A_i, λ_i), where each π_i is a path, each A_i is an adjective, and each $\lambda_i \in \mathcal{N}$. We write $S[\pi]$ for the set $\{\, A \mid \exists \lambda \text{ s.t. } (\pi, A, \lambda) \text{ occurs in } S \,\}$ and $S_\lambda[\pi]$ for the set $\{\, A \mid (\pi, A, \lambda) \text{ occurs in } S \,\}$.*

The attribute λ_i stores the level of A_i on path π_i. More precisely, an adjective that was obtained from a formula on a strict prefix of π_i will have level 0. An adjective that was obtained from an adjective with level λ on the same path, receives level $\lambda + 1$.

We define the tableaux procedure. In order to obtain termination, the procedure uses a blocking rule. Whenever a new path π is entered, it checks that there is no strict prefix π' of π containing a set of adjectives that are included in the formulas on the current path. For example, if path (f) contains A, B, and path (f, g) contains formulas A, B, C, we can close the branch. This rule is correct, because for every data tree d, the subtree $d.f.g$ is a subtree of $d.f$. If it is possible that $d.f.g \models A \wedge B \wedge C$, then one can replace $d.f$ by $d.f.g$ and obtain a smaller data tree, for which $d.f \models A \wedge B$.

The tableaux procedure always tries to extend in deterministic fashion first. If that does not result in a conflict, it selects a path π and tries all possible non-deterministic choices on this path. It keeps on trying until it either has explored all choices, or obtained a consistent stack S.

Definition 3.8. *We define a procedure that tries to establish that no data term d can have type T. The procedure starts by creating a stack with one element:*

$$S = (\,(\epsilon, \mathrm{ADJ}(T), 0)\,).$$

After that, it calls $b = \text{deterministic}(0)$. *If* $b = \bot$, *it returns* $\bot$. *Otherwise, it returns* **nondeterministic**(ϵ).
We list the subprocedures, starting with the procedure **deterministic**(d). *All subprocedures have access to the stack* S. *We write* $\|S\|$ *for the size of the stack, and write the* i*-th element in the form* $S_i = (\pi_i, A_i, \lambda_i)$.
Procedure **deterministic**(d) *must be called with a natural number* $1 \leq d \leq \|S\|$. *It returns* $\bot$ *or* $\top$, *with* $\bot$ *indicating contradiction. The implementation is as follows:*

1. *Set* $c = d$.
2. *If* $c \leq \|S\|$, *then*
   - *If* A *has form* v *with* v *an identifier that has no definition in* Σ_A, *then return* $\bot$.
   - *If* A *has form* $\neg c^{\geq}$ *with* c *the minimal element in its type, then return* $\bot$.
   - *For all* $i < c$ *with* $\pi_i = \pi_c$, *check whether* A_i *and* A_c *are in conflict, using the rules listed below. If they are in conflict, return* $\bot$. *Otherwise try the next* π_i. *The rules are:*
     - *A complementary pair* $A, \neg A$ *is in conflict.*
     - *A pair* c_1, c_2 *of any primitive type with* $c_1 \neq c_2$ *is in conflict.*
     - *A pair of form* $\neg c_1^{\geq}$, $c_2^{\geq}$ *with* $c_1 \leq c_2$ *is in conflict.*
     - *A pair of form* $c_1^{\geq}$, c_2 *with* $c_1 > c_2$, *is in conflict.*
     - *A pair of form* $\neg c_1^{\geq}, c_2$ *with* $c_1 \leq c_2$ *is in conflict.*
   - *Assign* $c = c + 1$ *and go back to step 1.*
3. *If* $d \leq \|S\|$, *check if one of the deterministic extension rules in the table below is applicable on* A_d. *If yes, then for every deterministic consequence* A', *push* $(\pi_d, A', \lambda_d + 1)$ *to* S.

v	$\Longrightarrow$	$\text{PNF}(A, \textbf{pos})$ *if* Σ_A *contains a definition of form* $v := (T, A)$
$\neg v$	$\Longrightarrow$	$\text{PNF}(A, \textbf{neg})$ *if* Σ_A *contains a definition of form* $v := (T, A)$
$\forall A$	$\Longrightarrow$	$\textbf{empty} \vee (\,\textbf{first}(A) \wedge \textbf{rest}(\forall A)\,)$
$\exists A$	$\Longrightarrow$	$\neg\textbf{empty} \wedge (\,\textbf{first}(A) \vee \textbf{rest}(\exists A)\,)$
$A_1 \wedge \cdots \wedge A_n$	$\Longrightarrow$	A_i *for each* $1 \leq i \leq n$

4. *If* A_d *has form* $f(A')$, *then push* $(\pi_d.f, A', 0)$ *and* $(\pi_d.f, \text{ADJ}(T), 0)$ *to* S, *where* $\text{ADJ}(T)$ *is the adjective component of the type* T *of* f.
5. *Set* $d = d + 1$. *Goto step 1.*

Procedure **nondeterministic**(π) *must be called with a path* π *that occurs in* S. *It returns* $\bot$ *or* $\top$, *with* $\bot$ *indicating contradiction. The implementation is as follows:*

1. *For every strict prefix* π' *of* π, *do:*
   (a) *if* $S_0[\pi'] \subseteq S[\pi]$, *then return* $\bot$. *(This is the blocking rule, mentioned above.)*
2. *Call* **nondeterministic**$(\pi, 1)$.

Procedure **nondeterministic**(π, n) *must be called with a path* π *that occurs in* S, *and with* $1 \leq n \leq \|S\|$. *It returns* $\bot$ *or* $\top$, *with* $\bot$ *indicating contradiction. The implementation is as follows:*

1. *Find the smallest* $n' \geq n$, *such that* $A_{n'}$ *has form* $A_1 \vee \cdots \vee A_m$ *with* $m \geq 2$.
2. *If no* n' *was found in the previous step, then find the next path* $\pi' > \pi$ *occurring in* S, *using the alphabetic lexicographic order* $>$. *Call* $b =$ **nondeterministic**(π') *and return* b. *If no path could be found, return* $\bot$.
3. *Set* $n = n'$. *We know that* S_n *has form* $(\pi,\ A_1 \vee \cdots \vee A_m, \lambda)$, *with* $m \geq 2$. *For* i *from* 1 *to* n, *do the following:*
   - *Set* $s = \|S\|$. *Push* $(\pi, A_i, \lambda + 1)$ *on the stack.*
   - *Call* $b =$ **deterministic**(s). *If* $b \neq \bot$, *call* $b =$ **nondeterministic**$(\pi, n + 1)$. *If* $b = \top$ *return* $\top$.
   - *Restore* S *to length* s.
4. *Set* $n = n + 1$. *Restart at step 1.*

Procedure **nondeterministic**(π) implements a loop checker which guarantees termination. Its correctness is based on the fact that, if a satisfying data tree can be found, it can be pruned into a data tree in which the calculus does not repeat initial states. We state the following without proof:

Theorem 3.9. *The tableaux calculus of Definition 3.8 terminates.*

In order to establish correctness of Theorem 3.9, it is sufficient establish that every branch is finite. During search, the procedure will only introduce subadjectives of the initial type, combined with subadjectives of identifiers defined in Σ_A. Therefore, blocking must eventually happen.

Theorem 3.10. *The procedure of Definition 3.8 is complete.*

Доказательство. The proof will be a bit informal. We have to prove that if no data tree d with $d\colon T$ exists, then the procedure of Definition 3.8 will reject T. We will prove the converse: If T is not rejected, then there exists a data tree d, s.t. $d\colon T$. Suppose that T is not rejected. This implies that the procedure terminates with an open branch. Let $S = (\pi_1, A_1, \lambda_1), \ldots, (\pi_n, A_n, \lambda_n)$ be the state of the stack with which the tableaux procedure terminated.
For a path occurring in S, let

$$S(\pi) = \{A \mid S \text{ contains a tuple } (\pi_i, A_i, \lambda_i), \text{ s.t. } \pi_i = \pi \text{ and } A_i \text{ has form } c,\ c^{\geq} \text{ or } \neg c^{\geq}\}.$$

For an arbitrary constant c, define

$$\begin{aligned} I(c) &= \{c\} \\ I(c^{\geq}) &= \{c' \mid c' \geq c\}, \\ I(\neg c^{\geq}) &= \{c' \mid c' < c\}. \end{aligned}$$

It is easily checked that for every π occurring in S, we have $\bigcap\{I(A) \mid A \in S(\pi)\} \neq \emptyset$, because otherwise the branch would be closed. Hence, we can select a constant c_π from each such set. Let A_S be the adjective

$$A_S = \bigwedge\{\pi(c_\pi) \mid \pi \text{ occurs in } S\}.$$

The adjective A_S states that for a given data tree d, selecting field sequence π will result in constant c_π. It is easy to construct a data tree d_S, s.t. $d_S \vDash A_S$ by starting with the required constants, and combining them as required by A_S. We show by backward induction (that is from n towards 1), that

$$d_S \vDash \pi_i(A_i) \wedge \cdots \wedge \pi_n(A_n).$$

Assume that we already have established that $d_S \vDash \pi_{i+1}(A_{i+1}) \wedge \cdots \wedge \pi_n(A_n)$. We proceed by case analysis on the form of A_i. We need to consider only the forms of A_i that do not close the branch.

- if A_i is an identifier v, then we know that v has a definition (T, A') in Σ_A, since otherwise the branch would have been closed. Hence we know that PNF$(A', \mathbf{pos})$ occurs among $A_{i+1}, \ldots, A_n$. As a consequence, $d_S \vDash$ PNF$(A', \mathbf{pos})$. By Theorem 3.5, we know that $d_S \vDash A'$.
- If A_i is a negated identifier $\neg v$, then if v has no definition in Σ_A, we have $d_S \nvDash \pi_i(v)$, so that $d_S \vDash \pi_i(\neg v)$. If v has a definition (T, A') in Σ_A, then PNF$(A', \mathbf{neg})$ occurs among $A_{i+1}, \ldots, A_n$. By induction, $d_S \vDash$ PNF$(A', \mathbf{neg})$. By Theorem 3.5, we have $d_S \vDash \neg A'$.
- The other cases can be obtained by inspecting the cases in Definition 2.2.

□

Theorem 3.11. *The procedure of Definition 3.8 is sound.*

Доказательство. In order to prove soundness, one must prove that if the procedure rejects a type T, then there is no data tree d with $d\colon T$. We will prove the converse: If there is a data tree d with $d\colon T$, then T will not be rejected by the procedure.

This would be trivial, if the blocking rule would not exist. It is easy to show that if $d\colon T$, there exists a stack $S = (\pi_1, A_1, \lambda_1), \ldots, (\pi_n, A_n, \lambda_n)$, with $(\pi_1, A_1, \lambda_1) = (\epsilon, \text{ADJ}(T), 0)$, that represents an open branch of the tableaux procedure when the blocking rule is not used.

Now assume that S will be closed when the blocking rule can be used. We will show that there exists a shorter stack S' which also starts with $S'_1 = (\epsilon, \mathrm{ADJ}(T), 0)$, and which also represents an open branch of the tableaux procedure when the blocking rule cannot be used. Repeating this process will result in a stack that does not contain any more applications of the blocking rule.

Let S be a stack with $S_1 = (\epsilon, \mathrm{ADJ}(T), 0)$, taken from an open branch of the tableaux procedure, when blocking is not used. Suppose that somewhere in S, the blocking rule would be applicable. This means that there exist π and π', s.t. π' is a strict prefix of π, and $S_0[\pi'] \subseteq S[\pi]$. We prune S as follows:

- Remove every (π_i, A_i, λ_i), s.t. π' is a prefix of π_i and π_i is a strict prefix of π.
- For every (π_i, A_i, λ_i), *s.t.* which π is a prefix of π_i write π_i as $\pi \cdot \pi'_i$, and replace it by $\pi' \cdot \pi'_i$.

It can be checked that the resulting S' is shorter, because there is at least one (π_i, A_i, λ_i) with $\pi_i = \pi'$ which will be removed. Moreover, S' still represents an open branch of the tableaux procedure without blocking. Hence we can continue the procedure and obtain an open branch on which the blocking rule is not applicable. □

4. Resolving Overloads in Types and Adjectives

Until now we have insisted that field and adjective names are always unique. In Example 1.10, we used sub, sub_1, sub_2, and sub_n as different variations of the name sub, dependent on whether we were taking a subformula of a negated formula, a formula constructed by a binary operator, or a formula constructed by an n-ary operator.

Similarly, we did not consider reuse of adjective names between different types. In reality, it is perfectly possible to have different types of formulas, for example modal, first-order and propositional, and define different nnf adjectives on each of them.

Modern programming languages like C^{++} or Java allow the use of ambiguous names which are made unambiguous by the compiler. For example, in C^{++}, one can define different print operators << on different types, and the compiler will pick the right one, when the programmer writes <<. This is called *overload resolution*. Without overload resolution, the programmer needs to invent a new name for every type that needs to be printed. For example, in C one has to include the type in the name of a print function, like `printfol` or `printmodal`.

We want overload resolution in our programming language: it should be possible that the user defines different nnf adjectives on different types and reuses the same name 'nnf' for them. Similarly, we want that the user can call all variations sub, sub_1, sub_2 and sub_n, just 'sub'.

Concretely, we want overload resolution on field names, adjective names, and function names. There will be no overload on type names, because we think it is unfeasable, and allowing ambiguous type names would result in ambiguous code.

In order to handle overload resolution, we introduce what we call *inexact identifiers*. Inexact identifiers are the identifiers that are used by the programmer in the program. We will call the identifiers that we have been using until now, *exact identifiers*. We assume a function $v^?$ that maps exact identifiers v to their inexact representations. The inexact representations are used in the program. As an example, one can introduce an inexact name sub and set $\mathrm{sub}^? = \mathrm{sub}^?_1 = \mathrm{sub}^?_2 = \mathrm{sub}^?_n = \mathrm{sub}$. Whenever identifier 'sub' occurs in the program, the compiler has to find out which of the exact variations of sub is meant. We define inexact identifiers:

Definition 4.1. *We assume an infinite set of inexact identifiers. We assume a map $v^?$ that maps exact identifiers v to inexact identifiers.*

Now we explain how inexact identifiers are used in the program. In order to do that, we modify the definitions of type and of adjective. In Definition 1.2, in the definition of compound types, we make the following modifications:

1. the identifiers $v_1, \ldots, v_n$, are inexact.

2. the identifier v is inexact.
3. identifier s is inexact.

In Definition 1.5, we will allow the identifiers v and f to be inexact. Apart from that, there are no changes.

Example 4.2. *We define propositional and multimodal logic:*

$$\text{ident} := (\,\text{c}\colon \text{char}\,)^*.$$

$$\text{prop} := \text{op}\colon \text{selector},\ \text{op?} \begin{pmatrix} \text{?var} & \Rightarrow & \text{ident}, (\,)^* \\ \text{?not} & \Rightarrow & \text{sub}\colon \text{prop},\ (\,)^* \\ \text{?implies} \vee \text{?equiv} & \Rightarrow & \text{sub}_1\colon \text{prop},\ \text{sub}_2\colon \text{prop},\ (\,)^* \\ \text{?and} \vee \text{?or} & \Rightarrow & (\,\text{sub}\colon \text{prop}\,)^* \end{pmatrix}$$

Similarly, one can define multimodal logic with inexact identifiers:

$$\text{modal} := \text{op}\colon \text{selector},\ \text{op?} \begin{pmatrix} \text{?var} & \Rightarrow & \text{ident}, (\,)^* \\ \text{?not} & \Rightarrow & \text{sub}\colon \text{prop},\ (\,)^* \\ \text{?implies} \vee \text{?equiv} & \Rightarrow & \text{sub}_1\colon \text{prop},\ \text{sub}_2\colon \text{prop},\ (\,)^* \\ \text{?and}, \text{?or} & \Rightarrow & (\,\text{sub}\colon \text{prop}\,)^* \\ \text{?box} \vee \text{?dia} & \Rightarrow & \text{sub}\colon \text{prop},\ (\,)^* \end{pmatrix}$$

In Example 4.2, the inexact field name sub occurs both in prop and in modal. In type prop, it occurs one time as scalar field, and one time as repeated field. If one knows that that $f.\text{op} = \text{?var}$, one can access $f.\text{sub}$. If one knows that $f.\text{op} \in \{\text{?and}, \text{?or}\}$, one can access $f.\text{sub}[i]$. Fieldname sub also occurs three times in type modal, two times as a scalar field, and one time as a repeated field.

Example 4.3. *The adjectives* atom *and* literal *can be defined both on* prop *and on* modal*:*

$$\begin{array}{lcl} \text{atom}\colon \text{prop} & := & \text{op(?var)} \\ \text{atom}\colon \text{modal} & := & \text{op(?var)} \\ \text{literal}\colon \text{prop} & := & \text{atom} \vee \text{op(?not)} \wedge \text{sub(?atom)} \\ \text{literal}\colon \text{modal} & := & \text{atom} \vee \text{op(?not)} \wedge \text{sub(?atom)} \end{array}$$

Similarly, NNF *can be defined both on* prop *and on* modal*:*

$$\text{nnf}\colon \text{prop} := \bigvee \begin{Bmatrix} \text{literal} \\ \text{op(?and} \vee \text{?or)} \wedge \text{sub}(\forall \text{nnf}) \end{Bmatrix}$$

$$\text{nnf}\colon \text{prop} := \bigvee \begin{Bmatrix} \text{literal} \\ \text{op(?and} \vee \text{?or)} \wedge \text{sub}(\forall \text{nnf}) \\ \text{op(?box} \vee \text{?dia)} \wedge \text{sub(nnf)} \end{Bmatrix}$$

In the example, there are different occurrences of sub, and it has to be determined which of the possible definitions is being referred to. Before we start discussing the treatment of inexact identifiers, note that defined identifiers in Σ_S, Σ_C and Σ_A are always exact. Moreover, we do not allow the use of ambiguous type names. This means that when a defined or built-in type is used, it must be referred to by its exact name.

Resolving of inexact identifiers starts with a preprocessing step on Σ_S and Σ_C. During this step, declared fields are replaced by unique exact identifiers, and any adjectives used in field declarations are moved to Σ_A. This is done by replacing them with a unique exact identifier, and defining this identifier in Σ_A. After the preprocessing step, all inexact identifiers are in adjectives in Σ_A.

Adjectives occurring in the types of function declarations will be dealt with in the same way. They are replaced with identifiers, defined in Σ_A, somewhat similar to the way subformulas are replaced in the CNF transformation ([8]). In our case the goal is not efficiency, but to make overload resolution possible in the first place.

We allow reuse of the same inexact field name in a compound type, as long as the different occurrences of the field name are in different states of the type. For example in prop in Example 4.2, it is possible to reuse fieldname sub between formulas built by ?not and formulas built by ?and∨?or, but renaming both fields sub_1 and sub_2 into sub at the same time would be impossible. It would be still possible to rename one of them into sub.

We first discuss how type definitions in Σ_S and Σ_C are processed.

Definition 4.4. *Let $D = (\, w_1 \colon W_1, \ldots, w_n \colon W_n \,)$ be a sequence of declarations. Let $w_{n+1} \colon W_{n+1}$ be a single declaration. We define*

$$D + (w_{n+1} \colon W_{n+1}) \;=\; (\, w_1 \colon W_1, \ldots, w_{n+1} \colon W_{n+1} \,).$$

Definition 4.5. *We define procedure* **preprocsimple**(T) *that preprocesses a simple type T. The result is again a simple type. It makes additions to Σ_A in the process.*
The implementation of **preprocsimple**(T) *is as follows:*

- *Let $A = \text{ADJ}(T)$, and let $T' = \text{IMPL}(T)$. If $A = \top$, then return T'. If $A \neq \top$, then create a new exact identifier α, assign $\Sigma_A(\alpha) := (T', A)$, and return $T' \circ \alpha$.*

Definition 4.6. *We define procedure* **preproccompound**(C, D) *that preprocesses a compound type C within declaration context D. The second argument is is used for checking that no inexact identifier is reused on a possible realization of C.*

- *If C has form $(\, v_1 \colon V_1, \ldots, v_n \colon V_n \,)^*$, then*
  - *if there exist $(w \colon W) \in D$ and $1 \leq i \leq n$, s.t. $w^? = v_i$, then create an error.*
  - *if there exist $1 \leq i < j \leq n$, s.t. $v_i = v_j$, then create an error.*
  - *Otherwise, create new, exact identifiers e_i for each v_i, and set $e_i^? = v_i$.*
  - *Set $V_i' =$* **preprocsimple**(V_i).
  - *Return $(\, e_1 \colon V_1', \ldots, e_n \colon V_n' \,)^*$.*
- *If C has form $(v \colon V,\ C')$ then if there is a $(w \colon W) \in D$, s.t. $w^? = v$, create an error.*
  Otherwise, let e be a new, exact identifier for v. Set $e^? = v$. Set $V' =$ **preprocsimple**(V). *Return*

$$e \colon V',\ \textbf{preproccompound}(C', D + (e \colon V')).$$

- *If C has form $s?(A_1 \Longrightarrow C_1, \ldots, A_m \Longrightarrow C_m)$, then there must a be unique $(w \colon W) \in D$, s.t. $w^? = s$.*
  - *If no such w exists, or w is not unique, then create an error.*
  - *Otherwise, create new exact identifiers $\alpha_1, \ldots, \alpha_m$, and set $\Sigma_A(\alpha_j) = (W, A_j)$, for each $1 \leq j \leq m$.*
  - *After that, return*

$$w?(\, \alpha_1 \Longrightarrow \textbf{preproccompound}(C_1, D), \ldots, \alpha_m \Longrightarrow \textbf{preproccompound}(C_m, D)\,).$$

We now apply procedures **preprocsimple** and **preproccompound** as follows:

- For every identifier v in the domain of Σ_S, replace $\Sigma_S(v)$ by **preprocsimple**$(\, \Sigma_S(v)\,)$.
- For every identifier v in the domain of Σ_C, replace $\Sigma_C(v)$ by **preproccompound**$(\, \Sigma_C(v),\ (\,)\,)$.

Example 4.7. *After being replaced by* **preproccompound**, *the definition of* prop *will have the following form in Σ_C :*

$$\text{prop} \;:=\; \text{op} \colon \text{selector},\ \begin{cases} \alpha_1 & \Longrightarrow & \text{ident},\ (\,)^* \\ \alpha_2 & \Longrightarrow & \text{sub} \colon \text{prop},\ (\,)^* \\ \alpha_3 & \Longrightarrow & \text{sub}_1 \colon \text{prop},\ \text{sub}_2 \colon \text{prop},\ (\,)^* \\ \alpha_4 & \Longrightarrow & (\, \text{sub}_n \colon \text{prop}\,)^* \end{cases}$$

$$\begin{aligned}
&\alpha_1 : \text{selector} := (\,\text{selector}, ?\text{var}\,)\\
&\alpha_2 : \text{selector} := (\,\text{selector}, ?\text{not}\,)\\
&\alpha_3 : \text{selector} := (\,\text{selector}, ?\text{implies} \vee ?\text{equiv}\,)\\
&\alpha_4 : \text{selector} := (\,\text{selector}, ?\text{and} \vee ?\text{or}\,)
\end{aligned}$$

At this moment, we have moved all inexactness into Σ_A. Adjective definitions in Σ_A have form $\Sigma_A(v) = (T, A)$, where v is an exact identifier, T is the type on which the defined adjective can be aplied, and A is the adjective. Unfortunately, T is a simple type, which also may contain inexact adjectives. As an example where this could occur, one could define adjective cnf (Example 1.10) on **prop**$\circ$nnf instead of prop. The definition would have form $\Sigma_A(\text{cnf}) = (\,\textbf{prop}\circ\text{nnf},\ A'\,)$, where A' is an inexact version of the expression given in Example 1.10, and nnf has a definition of form $\Sigma_A(\text{nnf}) = (\,\text{prop}, A''\,)$, where A'' is the expression of Example 4.3.

In order to solve this, we apply a preprocessing stage on Σ_A, similar to the preprocessing of Σ_S. We replace all adjectives occuring in domains by exact identifiers, while adding the definitions to Σ_A.

- For every identifier v in the domain of Σ_A, write $\Sigma_A(v)$ as (T, A). After that, replace $\Sigma_A(v)$ by $(\,\textbf{preprocsimple}(T), A)$.

Note that the call of **preprocsimple** may implicitly add new identifiers to Σ_A. These new identifiers need not be considered because their domain is an implementation type without adjectives. Hence there is no risk of non-termination. Unfortunately, there potentially exists a circular dependency between preconditions of adjectives that is hard to detect, as illustrated by the following example:

Example 4.8. *Suppose that Σ_A contains adjective definitions with the following circular structure:*

$$\begin{aligned}
v_1 &:= (\,T_1\circ\alpha_1, \cdots)\\
v_2 &:= (\,T_2\circ\alpha_2, \cdots)\\
&\cdots\\
\alpha_1 &:= (\,T_1, \cdots\ v_2^?\ \cdots)\\
\alpha_2 &:= (\,T_2, \cdots\ v_1^?\ \cdots)
\end{aligned}$$

If at some other point an occurrence of $v_1^?$ needs to be resolved, then v_1 is an overload candidate. In order to decide if v_1 should be used, one has to resolve the adjective of $\Sigma_A(\alpha_1)$, which contains $v_2^?$. In order to decide if v_2 is an overload candidate for this occurrence of $v_2^?$, one has to resolve the adjective in $\Sigma(\alpha_2)$, which again contains $v_1^?$. This circular dependency is not solvable. Note that using $v_1^?$ or $v_2^?$ inside the right hand side of v_1 or v_2 is unproblematic.

The circular dependency in Example 4.8 is unsolvable, so the compiler has to reject it. Unfortunately, it is difficult to detect, because it only exists if the use of $v_2^?$ in $\Sigma(\alpha_1)$ is applied on implementation type T_2, and the use of $v_1^?$ in $\Sigma(\alpha_2)$ is applied on implementation type T_1. If for example the use of $v_2^?$ in $\Sigma(\alpha_1)$ is not on T_2, then v_2 is not a candidate for overload, and there is no problem.

We solve this problem by starting to resolve an adjective definition, and whenever we encounter a precondition that has to be resolved first, we recursively try to solve this precondition. If this iteratively results in returning to the original definition, we reject Σ_A. In order to detect when we returned to the original definition, we maintain a set E of adjective definitions that we have encountered already. A circular dependency occurs when we need to resolve the precondition of an identifier that already occurs in E.

At this stage, all inexact identifiers are confined in Σ_A, in the second components A of the definitions $\Sigma_A(v) = (T, A)$. We give the procedure:

Definition 4.9. *Let Σ_A be the map of inexact adjective definitions. We define procedure* RESOLVE(v) *that tries to resolve the overloads in $\Sigma_A(v)$.*

It uses a set E of identifiers that were already encountered. Initially, $E = \emptyset$. The purpose of E is to detect circular dependencies of the type shown in Example 4.8. The implementation is as follows:

- *If v has no definition in $\Sigma_A(v)$, then the result is an error.*
- *If $v \in E$, then the result is also an error. This means that a circular dependency was detected.*
- *Otherwise, add v to E.*
- *Write $\Sigma_A(v)$ in the form (U, A). If U has form $(T \circ w)$, then call* RESOLVE(w).
- *Let $A' =$ RESOLVE$_T(\emptyset, A)$, and replace $\Sigma_A(v)$ by A'. Note that this function calls* RESOLVE$_T$, *that will be defined in Definition 4.10.*

Function RESOLVE(v) possibly calls itself in order to make w exact. This will be detected, because we will have $v \in E$. There is no way to detect such circularity a priori, because its existence depends on the way overloads are resolved.

The second procedure RESOLVE$_T(\Gamma, A)$ tries to resolve the fields and adjectives in the non-exact adjective A in context Γ. Context Γ is needed because it may contain preconditions of fields.

In case more than one overload candidate exists, we will take the nearest fit. This approach is used by C^{++} and Java. For example, if some class A is a subclass of B, which is a subclass of C, and some function f has definitions $f(A)$ and $f(C)$, then the call $f(a)$ will be resolved as $f(A)$, while the call $f(b)$ will be resolved as $f(C)$. We have no notion of subclass, but we have implication between adjectives. Using implication between adjectives, the rule becomes as follows: If an application of some inexact identifier f has different overload candidates $f_1, \dots, f_n$, with $f_1^? = \cdots f_n^? = f$, s.t. each f_i is defined on adjective A_i, then we resolve f as follows: If there is a unique A_i, s.t. A_i implies all of $A_1, \dots, A_n$, then f will be resolved as f_i. In the definition, we will write $\Gamma \vDash A$, which means that $\Gamma, \neg A$ is unsatisfiable. It should be noted that this is always in the context of a fixed Σ_S, Σ_C, and Σ_A. We do not want to use the notation $\Sigma_S, \Sigma_C, \Sigma_A, \Gamma \vDash A$, because it becomes too long.

Definition 4.10. *Let T be an implementation type, let Γ be a set of exact adjectives applicable on T, and let A be an inexact adjective.* RESOLVE$_T(\Gamma, A)$ *tries to resolve the overloads in A when applied on T in context Γ. If it succeeds it returns the exact version of A. We define* RESOLVE$_T(\Gamma, A)$ *by cases on the form of A.*

- *If A is an (inexact) identifier v, then let $v_1, \dots, v_n$ be the adjectives defined in Σ_A, that have $v_i^? = v$ and for which $\Sigma_A(v_i)$ has form (T, A_i) or $(T \circ w_i, A_i)$.*
  Set $C = \emptyset$. For each $i \in \{1, \dots, n\}$, do the following:
  - *If w_i is absent, add i to C.*
  - *Otherwise, call* RESOLVE(w_i). *(This is the first version, defined in Definition 4.9.) If after that, $\Gamma \vDash w_i$, then add i to C.*

  At this moment C is a subset of $\{1, \dots, n\}$ containing the candidates that can still be considered as overloads for v. For $i \in C$ do:
  - *for $j \in C \backslash \{i\}$ do:*
    - *If $\Gamma, w_i \vDash w_j$, then remove j from C.*

  If $\|C\| \neq 1$, then create an error message. Otherwise, return v_i where i is the unique element of C.
- *If A is a constant c, then if c has primitive type T, return c. Otherwise create an error.*
- *If A has form $c^{\geq}$, and c has a primitive type that is not* selector, *then return $c^{\geq}$. Otherwise, create an error.*
- *If A has form $f(A')$, then let $f_1, \dots, f_n$ be the fields (scalar or repeated) defined on type T, that have $f_i^? = f$. Note that T must be a defined compound type, because primitive types have no fields. Set $C = \{1, \dots, n\}$. For each $i \in C$ do:*
  - *let* PREC$(f_i) = g_{i,1}(A_{i,1}) \wedge \cdots \wedge g_{i,k_i}(A_{i,k_i})$ *with $k_i \geq 0$, be the precondition of field f_i as defined in Definition 1.12. Each $A_{i,j}$ is either a implementation type or has form $T_{i,j} \circ w_{i,j}$ with $w_{i,j}$ an exact identifier defined on implementation type $T_{i,j}$. For every $w_{i,j}$ $(1 \leq j \leq k_i)$ that is present, call* RESOLVE$(w_{i,j})$, *defined in Definition 4.9.*
  - *After that, check that $\Gamma \vDash g_{i,1}(A_{i,1}) \wedge \cdots \wedge g_{i,k_n}(A_{i,k_i})$. If not, then remove i from C.*

If $\|C\| \neq 1$, *then the result is error. Otherwise, let* i *be the unique element of* C. *Assume that the declaration of* f_i *has form* $f_i\colon W$. *If* f_i *is a scalar field, then return*

$$f_i(\,\mathrm{RESOLVE}_{\mathrm{IMPL}(W)}(\mathrm{ADJ}(W), A')\,).$$

if f_i *is a repeated field, then return*

$$f_i(\,\mathrm{RESOLVE}^*_{\mathrm{IMPL}(W)}(\mathrm{ADJ}(W), A')\,).$$

In the latter case, we called RESOLVE* *defined below.*

- *If* A *has form* $A_1 \vee \cdots \vee A_n$, *then*

$$\mathrm{RESOLVE}_T(\Gamma, A) = \mathrm{RESOLVE}_T(\Gamma, A_1) \vee \cdots \vee \mathrm{RESOLVE}_T(\Gamma, A_n).$$

- *If* A *has form* $A_1 \wedge \cdots \wedge A_n$, *then set* $\Gamma_1 = \Gamma$. *For* $i = 1$ *to* n *do the following:*
  - *Let* $A'_i = \mathrm{RESOLVE}_T(\Gamma_i, A_i)$.
  - *Set* $\Gamma_{i+1} = \Gamma_i \cup \{A'_i\}$.

  After that, return $A'_1 \wedge \cdots \wedge A'_n$.

Next we define RESOLVE* *which resolves repeated fields.*

- *If* A *has form* $\forall A'$, *then return* $\forall\mathrm{RESOLVE}_T(\emptyset, A')$.
- *If* A *has form* $\exists A'$, *then return* $\exists\mathrm{RESOLVE}_T(\emptyset, A')$.
- *If* $A =$ **empty**, *and* T *is not a compound type, then the result is an error. Otherwise return* **empty**.

In order to make nnf of Example 4.3 exact, one has to start by calling RESOLVE(nnf). Procedure RESOLVE will insert nnf into E, and call $\mathrm{RESOLVE}_{\mathrm{prop}}(\emptyset, A)$, with A the expanded definition of nnf. Since A is a disjunction, $\mathrm{RESOLVE}_{\mathrm{prop}}(\emptyset, A)$ will process the disjuncts independently.

The first disjunct equals literal. Procedure $\mathrm{RESOLVE}_{\mathrm{prop}}(\emptyset, \mathrm{literal})$ will establish that literal is the unique overload and return literal. It will not look at the definition of literal. If one wants to resolve the overloads in the definition of literal, one must call RESOLVE(literal) separately.

The second disjunct equals op(?and∨?or) ∧ sub(∀nnf). $\mathrm{RESOLVE}_{\mathrm{prop}}(\ \emptyset, \mathrm{op}(?\mathrm{and}\vee?\mathrm{or}) \wedge \mathrm{sub}(\forall\mathrm{nnf})\ )$ will recursively call $\mathrm{RESOLVE}_{\mathrm{prop}}(\ \mathrm{op}(?\mathrm{and}\vee?\mathrm{or})\ )$, which will resolve op (inexact) into op (exact) and call $\mathrm{RESOLVE}_{\mathrm{selector}}(?\mathrm{and}\vee?\mathrm{or})$, which will return ?and∨?or since both are constants. After that, it will call

$$\mathrm{RESOLVE}_{\mathrm{prop}}(\ \mathrm{op}(\ ?\mathrm{and}\vee?\mathrm{or}),\ \mathrm{sub}(\forall\mathrm{nnf})\ ).$$

There are two possible overloads for sub, namely sub for the ?not case, and $\mathrm{sub}_\forall$. We have PREC(sub) = op(?not), and $\mathrm{PREC}(\mathrm{sub}_\forall)$ = op(?and∨?or). Since only the latter is provable from the premiss, $\mathrm{sub}_\forall$ will be picked. Since $\mathrm{sub}_\forall$ is a repeated field, the procedure will recursively call $\mathrm{RESOLVE}^*_{\mathrm{prop}}(\emptyset, \forall\mathrm{nnf})$, which will recursively call $\mathrm{RESOLVE}_{\mathrm{prop}}(\emptyset, \mathrm{nnf})$. This call will return nnf without expanding it, after which the original call of $\mathrm{RESOLVE}_{\mathrm{prop}}$ will construct the complete exact overload

$$\bigvee \left\{ \begin{array}{l} \mathrm{literal} \\ \mathrm{op}(?\mathrm{and}\vee?\mathrm{or}) \vee \mathrm{sub}(\forall\mathrm{nnf}) \end{array} \right.$$

5. Conclusions

Our goal is to develop and implement an efficient programming language in which it is convenient to implement algorithms on trees whose forms are very different.

In order to obtain this, we have defined a flexible type system together with a way of refining these types by means of adjectives. The adjectives are intended as a replacement for matching in functional languages. In

order to make this replacement possible, we have given a precise semantics for adjectives, so that adjectives can be evaluated on concrete data.

We provided a terminating tableaux calculus for deciding propositional relations between adjectives, and applied this procedure to overload resolution in imprecise formulations of adjectives. The overload resolution procedure replaces ambiguous field references in adjective definitions by exact field references. A similar algorithm can be used for resolving ambiguous overloads in function calls.

6. Acknowledgements

This work gained from discussions with Cláudia Nalon. We thank Nazarbayev University for supporting this research through the Faculty Development Competitive Research Grant Program (FDCRGP) number 021220FD1651.

Список литературы

[1] Y. Minsky, A. Madhavapeddy и J. Hickey, *Real World OCaml (functional programming for the masses)*. O'Reilly, 2013, ISBN: 978-1-449-32391-2.

[2] G. Hutton, *Programming in Haskell (second edition)*. Cambridge University Press, 2016, ISBN: 978-1-316-62622-1.

[3] M. Odersky, L. Spoon и B. Venners, *Programming in Scala Third Edition*. Artima Press, 2007-2016, ISBN: 0-9815316-8-7.

[4] P. Rondon, M. Kawaguchi и R. Jhala, "Liquid Types", в *Programming Language Design and Implementation (PLDI)*, R. Gupta и S. Amarasinghe, ред., сер. ACM, ACM, 2009, с. 159—169.

[5] R. Jhala и N. Vazou, "Refinement Types: A Tutorial", *Foundations and Trends in Programming Languages*, т. 6, № 3–4, с. 159—317, 2021, ISSN: 2325-1107. DOI: 10.1561/2500000032. url: http://dx.doi.org/10.1561/2500000032.

[6] S. Klabnik и C. Nichols, *The Rust Programming Language*. No Starch Press, 2019, ISBN: 978-1718500440.

[7] Michael Gelfond и Vladimar Lifschitz, "The Stable Model Semantics for Logic Programming", в *Fifth International Conference and Symposium on Logic Programming*, R. Kowalski и K. Bowen, ред., MIT Press, 1988, с. 1070—1080.

[8] G. Tseytin, "On the complexity of Derivation in the Propositional Calculus", в *Studies in Constructive Mathematics and Mathematical Logic, Part II*, A. Slisenko, ред., Zapiski Nauchnykh Seminarov, 1970, с. 115—125.

References

[1] Y. Minsky, A. Madhavapeddy, and J. Hickey, *Real World OCaml (functional programming for the masses)*. O'Reilly, 2013, ISBN: 978-1-449-32391-2.

[2] G. Hutton, *Programming in Haskell (second edition)*. Cambridge University Press, 2016, ISBN: 978-1-316-62622-1.

[3] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala Third Edition*. Artima Press, 2007-2016, ISBN: 0-9815316-8-7.

[4] P. Rondon, M. Kawaguchi, and R. Jhala, "Liquid Types", in *Programming Language Design and Implementation (PLDI)*, R. Gupta and S. Amarasinghe, Eds., ser. ACM, ACM, 2009, pp. 159–169.

[5] R. Jhala and N. Vazou, "Refinement Types: A Tutorial", *Foundations and Trends in Programming Languages*, vol. 6, no. 3–4, pp. 159–317, 2021, ISSN: 2325-1107. DOI: 10.1561/2500000032. [Online]. Available: http://dx.doi.org/10.1561/2500000032.

[6] S. Klabnik and C. Nichols, *The Rust Programming Language*. No Starch Press, 2019, ISBN: 978-1718500440.

[7] Michael Gelfond and Vladimar Lifschitz, "The Stable Model Semantics for Logic Programming", in *Fifth International Conference and Symposium on Logic Programming*, R. Kowalski and K. Bowen, Eds., MIT Press, 1988, pp. 1070–1080.

[8] G. Tseytin, "On the complexity of Derivation in the Propositional Calculus", in *Studies in Constructive Mathematics and Mathematical Logic, Part II*, A. Slisenko, Ed., Zapiski Nauchnykh Seminarov, 1970, pp. 115–125.

MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 28, NO. 4, 2021
journal homepage: www.mais-journal.ru

ALGORITHMS

Solving Linear Programming Problems by Reducing to the Form with an Obvious Answer

G. D. Stepanov[1]

DOI: 10.18255/1818-1015-2021-4-434-451

[1]Voronezh State Pedagogical University, 86 Lenin str, Voronezh 394043, Russia.

MSC2020: 90C05
Research article
Full text in Russian

Received July 11, 2021
After revision December 1, 2021
Accepted December 8, 2021

The article considers a method for solving a linear programming problem (LPP), which requires finding the minimum or maximum of a linear functional on a set of non-negative solutions of a system of linear algebraic equations with the same unknowns. The method is obtained by improving the classical simplex method, which when involving geometric considerations, in fact, generalizes the Gauss complete exclusion method for solving systems of equations. The proposed method, as well as the method of complete exceptions, proceeds from purely algebraic considerations. It consists of converting the entire LPP, including the objective function, into an equivalent problem with an obvious answer.

For the convenience of converting the target functional, the equations are written as linear functionals on the left side and zeros on the right one. From the coefficients of the mentioned functionals, a matrix is formed, which is called the LPP matrix. The zero row of the matrix is the coefficients of the target functional, a_{00} is its free member. The algorithms are described and justified in terms of the transformation of this matrix. In calculations the matrix is a calculation table.

The method under consideration by analogy with the simplex method consists of three stages. At the first stage the LPP matrix is reduced to a special 1-canonical form. With such matrices one of the basic solutions of the system is obvious, and the target functional on it is a_{00}, which is very convenient. At the second stage the resulting matrix is transformed into a similar matrix with non-positive elements of the zero column (except a_{00}), which entails the non-negativity of the basic solution. At the third stage the matrix is transformed into a matrix that provides non-negativity and optimality of the basic solution.

For the second stage the analog of which in the simplex method uses an artificial basis and is the most time-consuming, two variants without artificial variables are given. When describing the first of them, along the way, a very easy-to-understand and remember analogue of the famous Farkas lemma is obtained. The other option is quite simple to use, but its full justification is difficult and will be separately published.

Keywords: linear programming; Gauss method; simplex method; LPP matrix; resolving element; choice rule; Farkas lemma; systems of linear equations; non-negative solution

INFORMATION ABOUT THE AUTHORS

Gleb D. Stepanov
correspondence author

orcid.org/0000-0003-3237-849X. E-mail: stpnv42@mail.ru
PhD, associate professor.

For citation: G. D. Stepanov, "Solving Linear Programming Problems by Reducing to the Form with an Obvious Answer", *Modeling and analysis of information systems*, vol. 28, no. 4, pp. 434-451, 2021.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 28, № 4, 2021
сайт журнала: www.mais-journal.ru

ALGORITHMS

Решение задач линейного программирования приведением к виду с очевидным ответом

Г. Д. Степанов[1]

DOI: 10.18255/1818-1015-2021-4-434-451

[1]Воронежский государственный педагогический университет, ул. Ленина, д. 86, г. Воронеж, 394043 Россия.

УДК 519.852
Научная статья
Полный текст на русском языке

Получена 11 июля 2021 г.
После доработки 1 декабря 2021 г.
Принята к публикации 8 декабря 2021 г.

В статье рассматривается способ решения задачи линейного программирования (ЗЛП), которая требует найти минимум или максимум линейного функционала на множестве неотрицательных решений системы линейных алгебраических уравнений с теми же неизвестными. Способ получен при усовершенствовании классического симплекс-метода, который, привлекая геометрические соображения фактически обобщает метод полных исключений Гаусса решения систем уравнений. Предлагаемый способ, как и метод полных исключений, исходит из чисто алгебраических соображений. Он заключается в преобразовании всей ЗЛП, включая целевую функцию, в эквивалентную задачу с очевидным ответом.

Ради удобства преобразования целевого функционала, уравнения записываются как линейные функционалы в левой части и нули в правой. Из коэффициентов упомянутых функционалов составляется матрица, которая называется матрицей ЗЛП. Нулевая строка матрицы – коэффициенты целевого функционала, a_{00} – его свободный член. Описание и обоснование алгоритмов ведется в терминах преобразования этой матрицы. При вычислениях матрица является расчетной таблицей.

Рассматриваемый метод, по аналогии с симплекс-методом, состоит из трех этапов. На первом этапе матрица ЗЛП приводится к специальному 1-каноническому виду. При таких матрицах одно из базисных решений системы очевидно, и на нем целевой функционал равен a_{00}, что очень удобно. На втором этапе полученная матрица преобразуется в аналогичную матрицу с неположительными элементами нулевого столбца (кроме a_{00}), что влечет неотрицательность базисного решения. На третьем этапе матрица преобразуется в матрицу, обеспечивающую неотрицательность и оптимальность базисного решения.

Для второго этапа, аналог которого в симплекс-методе использует искусственный базис и является наиболее трудоемким, приводятся два варианта без искусственных переменных. При описании первого из них, попутно, получен очень простой для понимания и запоминания аналог знаменитой леммы Фаркаша. Другой вариант совсем прост в применении, но его полное обоснование сложно и будет опубликовано отдельно.

Ключевые слова: линейное программирование; метод Гаусса; симплекс-метод; матрица ЗЛП; разрешающий элемент; правило выбора; лемма Фаркаша; системы линейных уравнений; неотрицательное решение

ИНФОРМАЦИЯ ОБ АВТОРАХ

Глеб Дмитриевич Степанов
автор для корреспонденции

orcid.org/0000-0003-3237-849X. E-mail: stpnv42@mail.ru
канд.физ.-мат.наук, доцент.

Для цитирования: G. D. Stepanov, “Solving Linear Programming Problems by Reducing to the Form with an Obvious Answer”, *Modeling and analysis of information systems*, vol. 28, no. 4, pp. 434-451, 2021.

В статье дается метод решения задачи линейного программирования (ЗЛП), основанный на преобразовании исходной задачи к эквивалентной ЗЛП, для которой ответ практически очевиден. Его можно считать усовершенствованным симплекс-методом с более простой для понимания, запоминания и реализации вычислительной схемой, которая, в частности, не использует искусственных переменных. Для понимания метода достаточно знаний материала средней школы, поскольку при изложении легко обойтись даже без таких элементарных понятий линейной алгебры, как линейная зависимость и линейная независимость, определитель, ранг матрицы, а также и без геометрической интерпретации.

Отметим, что метод был получен в 1996 году специально для студентов с недостаточной математической подготовкой, когда в форс-мажорной ситуации автору было поручено провести занятия по исследованию операций у студентов, зачисленных,в качестве эксперимента с экстернатом, сразу на третий курс Воронежского педуниверситета. (При этом автор учитывал ощущение неоправданной сложности общепринятого изложения, возникшее у него еще при первом знакомстве с предметом по монографии [1] в далеком 1962г.) Лишь спустя многие годы автор обратил внимание на то, что полученный метод, судя по всему, специалистам незнаком: регулярно публикуются учебные пособия, в которых излагается традиционный симплекс-метод с усложненной вычислительной таблицей, с трудными для понимания и запоминания условиями оптимальности, с использованием искусственных переменных, увеличивающих размеры задач, а преподаватели соответствующих дисциплин утверждают, что "проще нельзя".

Наличие слишком большого количества публикаций по линейному программированию (только в библиографии [2] около 1500 наименований) не позволяет нам быть уверенными в новизне нашего подхода, но зато нет сомнений в том, что он был бы полезен и во многих прежних, и в новых исследованиях по данной тематике, а тем более в практике преподавания. В частности, в пункте 6 дается простой для понимания и запоминания аналог леммы Фаркаша, а в пункте 7 анонсируется простой метод нахождения неотрицательного решения систем линейных алгебраических уравнений.

Иллюстрация использования рассматриваемого в статье подхода к решению ЗЛП дана в четырех примерах пункта 8.

1. В ЗЛП, для непосредственного решения которых предназначен симплекс-метод (см.[1—8]), речь идет о минимизации или максимизации линейной функции от n неизвестных на множестве неотрицательных решений системы из m ($m < n$) линейных алгебраических уравнений от тех же неизвестных.

В отличии от классических формулировок будем считать, что оптимизируемая функция может быть не только однородной линейной формой, но и неоднородным линейным функционалом, у которого допускается и неравный нулю свободный член (аналогичные целевые функционалы используются и в [9, 10]). Указанное чисто формальное расширение задачи позволяет преобразовывать всю задачу, а не только систему уравнений, как это делается в классическом симплекс-методе. Систему уравнений будем записывать в виде, при котором в левой части уравнений стоят линейные функционалы, а в правой – нули. Конечно, такое представление, получающееся из традиционного переносом (с изменением знака) свободных членов в левую часть, несколько непривычно, но зато удобно при преобразовании оптимизируемого функционала.

Итак, рассматривается линейный функционал

$$a_{00} + a_{01}x_1 + a_{02}x_2 + \ldots + a_{0n}x_n\ , \tag{1}$$

и система линейных алгебраических уравнений

$$\begin{gathered} a_{10} + a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = 0, \\ a_{20} + a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = 0, \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ a_{m0} + a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n = 0. \end{gathered} \tag{2}$$

В ЗЛП(min)/ЗЛП(max) требуется на множестве всех неотрицательных решений системы (2) найти минимум/максимум функционала (1) и какое-нибудь из неотрицательных решений, на котором этот минимум/максимум достигается. Если нужный экстремум отсутствует, то надо указать причину этого: либо система (2) несовместна, либо у системы нет неотрицательных решений, либо на непустом множестве неотрицательных решений функционал (1) не ограничен снизу (для ЗЛП(min)) или сверху (для ЗЛП(max)).

Соответствующую (1) и (2) матрицу

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & \ldots & a_{0n} \\ a_{10} & a_{11} & a_{12} & \ldots & a_{1n} \\ a_{20} & a_{21} & a_{22} & \ldots & a_{2n} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ a_{m0} & a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix} \tag{3}$$

будем называть матрицей ЗЛП. В наших алгоритмах такая матрица, по существу, будет расчетной таблицей.

Матрицы ЗЛП, имеющие вид

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & \ldots & a_{0n-m} & 0 & 0 & \ldots & 0 \\ a_{10} & a_{11} & a_{12} & \ldots & a_{1n-m} & 1 & 0 & \ldots & 0 \\ a_{20} & a_{21} & a_{22} & \ldots & a_{2n-m} & 0 & 1 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ a_{m0} & a_{m1} & a_{m2} & \ldots & a_{mn-m} & 0 & 0 & \ldots & 1 \end{pmatrix}, \tag{4}$$

или приводимые к такому виду перестановкой столбцов с номерами большими нуля, будем называть матрицами канонической структуры или короче 1-каноническими. Конечно, перестановок столбцов, приводящих 1-каноническую матрицу к виду (4), может быть несколько. При выделенной каким-либо образом одной из них будем обозначать через $s_1, s_2, \ldots, s_n$ соответствующую перестановку исходных номеров столбцов. При этом номера $r_1 = s_{n-m+1}, r_2 = s_{n-m+2}, \ldots, r_m = s_n$, сами столбцы и переменные с этими номерами будем называть базисными, а номера $s_1, s_2, \ldots, s_{n-m}$, сами столбцы и переменные с этими номерами будем называть внебазисными.

Очевидно, что вектор $(x_1, x_2, \ldots, x_n)$, в котором все внебазисные переменные равны нулю, а базисные переменные $x_{r_i} = -a_{i0}$ $(i = 1, 2, \ldots m)$, является решением системы линейных ограничений. Такое решение называется базисным. Следующая основная лемма дает условия, при которых ответ для ЗЛП очевиден.

Лемма 1. *Пусть матрица ЗЛП имеет каноническую структуру и X – множество неотрицательных решений системы ограничений. Тогда*

1^0. *На базисном решении целевой функционал равен a_{00}.*

2^0. *Базисное решение неотрицательно тогда и только тогда, когда*

$$a_{10} \le 0, a_{20} \le 0, \ldots, a_{m0} \le 0. \tag{5}$$

3^0. *Если выполнено (5) и условие*

$$a_{01} \ge 0, a_{02} \ge 0, \ldots, a_{0n} \ge 0, \tag{6}$$

то a_{00} является минимумом целевого функционала на X.

4^0. *Если выполнено (5) и условие*

$$a_{01} \le 0, a_{02} \le 0, \dots, a_{0n} \le 0, \tag{7}$$

то a_{00} является максимумом целевого функционала на X.

5^0. *Если X непусто и существует внебазисный номер $q \neq 0$ такой, что*

$$a_{1q} \le 0, a_{2q} \le 0, \dots, a_{mq} \le 0, \tag{8}$$

то при $a_{0q} < 0$ целевой функционал на X неограничен снизу, а при $a_{0q} > 0$ – неограничен сверху.

Доказательство. Справедливость свойств $1^0, 2^0$ очевидна.

Для обоснования 3^0 заметим, что при выполнении (6) минимум целевого функционала на множестве всех неотрицательных векторов равен a_{00}. Такое же значение функционал принимает на базисном решении, которое, ввиду (5), тоже неотрицательно. Вместе это означает справедливость 3^0.

Свойство 4^0 доказывается аналогично свойству 3^0.

Для обоснования свойства 5^0 обозначим через y=$(y_1, y_2, \dots, y_n)$ какое-нибудь неотрицательное решение системы линейных ограничений и рассмотрим зависящие от параметра t вектора $z(t) = (z_1(t), z_2(t), \dots, z_n(t))$ и $x(t) = y + z(t)$, где

$$z_j(t) = \begin{cases} t & \text{, если} \quad j = q, \\ 0 & \text{, если} \quad j \in \{s_1, s_2, \dots, s_{n-m}\} \setminus q, \\ -ta_{iq} & \text{, если} \quad j = r_i \quad (i = 1, 2, \dots, m). \end{cases}$$

При положительных t вектора $x(t)$ и $z(t)$ являются, соответственно, неотрицательными решениями системы линейных ограничений и ее однородной части. Целевой функционал на $z(t)$ равен ta_{0q} и с ростом t неограниченно убывает, если $a_{0q} < 0$, или неограниченно возрастает, если $a_{0q} > 0$. Этот линейный функционал, ввиду независимости y от t, с ростом t ведет себя на $x(t)$ также, как на $z(t)$, что означает справедливость 5^0. **Лемма 1 доказана.**

Замечание. ЗЛП(min) и ЗЛП(max) легко сводятся друг к другу умножением целевого функционала на (-1). Тем не менее, мы даем формулировки для каждой из них. В матрицах ЗЛП, при формулировке леммы 1 и формулировках алгоритмов решения ЗЛП, прослеживается некоторая аналогия между строками и столбцами. Такая аналогия была бы еще более заметной, если в формулировку леммы добавить аналоги свойств $2^0 - 5^0$, относящиеся к задаче оптимизации линейного функционала на множестве неположительных решений.

2. Уточним используемое нами понятие эквивалентных ЗЛП. Две системы линейных алгебраических уравнений называются эквивалентными, если множества их решений совпадают. Две ЗЛП(min) (или две ЗЛП(max)) называются эквивалентными, если их системы уравнений эквивалентны и на каждом решении таких систем значения целевых функционалов этих задач совпадают.

Преобразовывать ЗЛП в эквивалентную задачу с очевидным ответом будем с помощью шагов исключения переменных, в основном аналогичных используемым в методе Гаусса решения систем алгебраических уравнений. Каждый шаг исключения выполняется несколькими элементарными операциями, к которым будем относить перестановку уравнений в системе ограничений, умножение уравнений системы на ненулевые множители, добавление к одним уравнениям линейных комбинаций других, вычеркивание из системы ограничений тривиальных уравнений вида 0 = 0 и, наконец, добавление к целевому функционалу линейных комбинаций функционалов, стоящих в левой части уравнений системы ограничений. В терминах матрицы ЗЛП эти преобразования означают перестановку строк с номерами большими нуля, умножение строк с номерами большими нуля

на ненулевые множители, добавление к одним строкам (включая строку с номером 0) линейной комбинации строк с номерами большими нуля.

Легко понять, что перечисленные элементарные операции преобразуют ЗЛП в эквивалентные. То, что они не меняют множество решений системы линейных ограничений задачи, очевидно. Добавление же к целевому функционалу указанной линейной комбинации может менять значение целевого функционала на каких-то векторах, но не на решениях системы линейных ограничений, поскольку на этих решениях добавляемые функционалы равны нулю.

Цель шага исключения с ведущим элементом $a_{pq} \neq 0$, где $1 \le p \le m$ и $1 \le q \le n$, – преобразовать ЗЛП в эквивалентную ЗЛП с матрицей, у которой $a_{pq} = 1$ и $a_{iq} = 0$ при $i \in \{0, 1, \dots, m\} \setminus p$. (Такой шаг переводит номер q в разряд базисных, делает $r_p = q$). Для этого p-е уравнение умножается на $1/a_{pq}$, затем полученное уравнение при каждом $i \in \{1, \dots, m\} \setminus p$ умножается на a_{iq} и вычитается из i-го уравнения, а из целевого функционала вычитается функционал, стоящий в левой части полученного p-го уравнения умноженный на a_{0q}.

Если временно обозначить через $a^{\text{н}}_{ij}$ элементы матрицы ЗЛП после шага исключения, то элементы новой матрицы выражаются через старые элементы по формулам

$$\begin{aligned} a^{\text{н}}_{pj} &= a_{pj}/a_{pq} && (j \in \{0, 1, \dots, n\}), \\ a^{\text{н}}_{ij} &= a_{ij} - a_{pj}a_{iq}/a_{pq} && (i \in \{0, 1, \dots, m\} \setminus p,\ \ j \in \{0, 1, \dots, n\}). \end{aligned} \tag{9}$$

Если перед шагом исключения матрица ЗЛП была 1-канонической, то и после этого шага она будет 1-канонической. При этом прежний базисный номер r_p станет новым значением внебазисного номера s_q, а номер q станет новым значением r_p, т.е. станет базисным номером.

В заключение пункта отметим, что аналоги леммы 1 и формул (9) можно было бы сформулировать и при общепринятом способе задания системы уравнений со свободными членами в правой части уравнений. В лемме пришлось бы заменить знаки неравенств (5) на противоположные, что не повлияло бы на восприятие формулировки, а в (9) для элемента a_{00} уже пришлось бы использовать формулу, отличную от формулы для других элементов, что усложнило бы дальнейшее изложение.

3. Перейдем к обсуждению алгоритма решения ЗЛП. Поскольку ЗЛП и ее матрица однозначно определяют друг друга, далее обычно будем говорить о преобразованиях матрицы ЗЛП, имея в виду, что при этом преобразуется и сама ЗЛП. Лемма 1 позволяет увидеть и реализовать разные стратегии решения. Классическому симплекс-методу соответствует стратегия, состоящая из трех этапов преобразования матрицы ЗЛП. (На аналогичных этапах симплекс-метода преобразуется расширенная матрица системы уравнений и вычисляются некие дополнительные характеристики и оценки.)

Этап 1. Матрица 3 преобразуется в 1-каноническую матрицу.

Этап 2. Полученная матрица преобразуется в аналогичную матрицу, удовлетворяющую условию (5).

Этап 3. Полученная матрица, удовлетворяющая (5), преобразуется в аналогичную матрицу, дополнительно удовлетворяющую условию (6), если решается ЗЛП(min), или (7), если решается ЗЛП(max).

Наиболее простым является этап 1, очень похожий на основную часть метода полных исключений Гаусса решения систем уравнений. На этом этапе следует поочередно при $p = 1, \dots, m$ найти $a_{pq} \neq 0$ ($q \ge 1$), запомнить $r_p = q$ и преобразовать матрицу по формулам (9). Если при каждом p ненулевой разрешающий элемент будет существовать, то система ограничений (2) совместна и матрица (3) будет преобразована в матрицу нужной структуры. Однако может случиться, что при некотором p все $a_{pq} = 0$ ($q = 1, \dots, n$). В таком случае, если $a_{p0} \neq 0$, то система ограничений (2) несовместна, что завершает процесс поиска решения ЗЛП. Если же при некотором p все $a_{pq} = 0$ ($q = 0, 1, \dots, n$), то

p-е уравнение тривиально и его следует вычеркнуть (уменьшить на единицу номера последующих уравнений и значение m), а затем продолжить выполнение этапа 1 с того же номера p. Трудоемкость этапа 1 равна $O(nm^2)$ арифметических операций.

В симплекс-методе наиболее трудоемкий этап 2 с помощью искусственного базиса сводится к использованию этапа 3, часто называемому основной процедурой. По этой причине мы тоже этап 3 обсудим раньше этапа 2.

К моменту начала выполнения этапа 3 матрица ЗЛП уже приведена к каноническому виду и удовлетворяет условию (5). Это означает, что базисное решение уже неотрицательно и на нем целевой функционал равен a_{00}. Этап представляет собой итерационный цикл, в котором поочередно по определенному правилу в матрице выбирается разрешающий элемент a_{pq}, а затем выполняется шаг преобразования матрицы по формулам (9). Цикл завершается, когда при выборе очередного разрешающего элемента обнаруживается (с учетом свойств 3^0, 4^0, 5^0 леммы 1), что либо требуемый экстремум уже найден, либо отсутствует. Ясно, что наибольшего внимания в обсуждении этапа 3 заслуживает именно правило выбора разрешающего элемента.

Общее соображение, которое лежит в основе выбора разрешающего элемента, заключается в стремлении сохранить при каждом преобразовании матрицы неотрицательность базисного решения вместе с улучшением на нем значения целевого функционала, т.е. с уменьшением a_{00}, когда решается ЗЛП(min), или с увеличением a_{00}, когда решается ЗЛП(max). Добиться такого улучшения в вырожденном случае (так называют случай, когда в (5) некоторые неравенства нестрогие) удается не всегда, а в невырожденном случае вопрос решается следующим правилом.

Правило 1. *В качестве номера разрешающего столбца надо брать q, при котором либо $a_{0q} < 0$, если решается ЗЛП(min), либо $a_{0q} > 0$, если решается ЗЛП(max). После выбора q в качестве номера разрешающей строки надо брать такое p, что $a_{pq} > 0$ и*

$$a_{p0}/a_{pq} \geq a_{i0}/a_{iq} \quad \textit{при} \quad a_{iq} > 0 \ (i \neq 0, p). \tag{10}$$

Если выбирать a_{pq} по этому правилу, то условие (10) гарантирует выполнение (5) и после шага преобразования матрицы. Значение целевого функционала на новом базисном решении будет равно $a_{00} - a_{p0}a_{0q}/a_{pq}$, а значит улучшится в невырожденном случае и не ухудшится в вырожденном.

Если при выборе разрешающего элемента не найдется q, требуемого правилом 1 (т.е. либо выполнено (6) при решении ЗЛП(min), либо (7) при решении ЗЛП(max)), то по лемме 1 искомый экстремум найден и задача решена. При уже выбранном q может не существовать требуемого p, т.е. может выполняться условие (8). Тогда нужный экстремум отсутствует из-за неограниченности функционала и решение тоже завершено. В остальных случаях итерационный цикл продолжается. Естественно возникает вопрос о конечности этого цикла.

Здесь надо заметить, что правилом 1 значения q и p определяются неоднозначно. В невырожденном случае цикл завершится при любом уточнении правила 1, т.к. при каждом преобразовании матрицы значение a_{00} улучшается, а значит зацикливание исключено. В вырожденном случае при неудачном уточнении этого правила зацикливание возможно. Опыт использования симплекс метода показывает, что это происходит довольно редко, но есть примеры, когда наиболее естественные уточнения правила 1 в одних вырожденных случаях приводят к успешному завершению этапа, а в других – к зацикливанию (см. пример 1 пункта 8).

4. В литературе по линейному программированию можно встретить несколько приемов, позволяющих избежать зацикливания симплекс-метода: метод возмущений или ε-метод (см. [1, 3, 10, 11]), лексикографический метод (см. [4—6, 8]), метод Бленда (см. [7, 12, 13]). Последний метод является наиболее простым, и именно его аналогом воспользуемся для исключения зацикливания.

Правило 2. *При решении ЗЛП(min) (ЗЛП(max)) в качестве номера разрешающего столбца следует брать наименьшее q, при котором* $a_{0q} < 0$ (> 0). *После выбора q в качестве номера разрешающей строки следует брать p с наименьшим* r_p, *из числа удовлетворяющих* $a_{pq} > 0$ *и (10).*

Лемма 2. *Пусть матрица ЗЛП имеет каноническую структуру и удовлетворяет условию (5). Тогда итерационный цикл, каждая итерация которого заключается в выборе разрешающего элемента* a_{pq} *по правилу 2 и преобразовании матрицы по формулам (9), конечен. В результате либо будет найден нужный экстремум, либо будет установлено отсутствие такого экстремума из-за неограниченности целевого функционала.*

Доказательство (от противного). Для определенности будем считать, что речь идет о ЗЛП(min). Предположим, что существуют 1-канонические матрицы, при которых имеет место зацикливание. Далее будем говорить о цикле из матриц с минимально возможными значениями n и m. Для такого цикла каждый из n столбцов и каждая из m строк, номера которых больше 0, должны периодически оказываться разрешающими, так как иначе их вычеркиванием можно уменьшить размеры матриц, не влияя на наличие цикла. В частности, у одних матриц цикла номер n является базисным, а у других – нет, но зато у всех матриц $a_{10} = a_{20} = \ldots = a_{m0} = 0$.

Элементы матрицы из цикла, при преобразовании которой исходный номер n переходит из разряда внебазисных в разряд базисных, будем помечать штрихом. По правилу 2 у этой матрицы A' элемент $a'_{0n} < 0$ и $a'_{0j} \geq 0$ при $j \in \{1, 2, \ldots, n-1\}$.

Для матрицы из цикла, при преобразовании которой исходный номер n переходит из разряда базисных в разряд внебазисных, сохраним обозначение A. В разрешающем столбце этой матрицы только сам элемент a_{pq} может стать строго положительным, так как иначе по правилу 2 из базиса выводился бы номер $< n$. Поэтому у вектора $z = (z_1, z_2, \ldots z_n)$ с координатами

$$z_j = \begin{cases} 1 & \text{, если } \quad j = q, \\ 0 & \text{, если } \quad j \in \{1, 2, \ldots, n\} \setminus q \setminus \{s_1, s_2, \ldots, s_m\}, \\ -a_{iq} & \text{, если } \quad j = s_i \ (i = 1, 2, \ldots, m), \end{cases}$$

элемент $z_n = z_{s_p} = -a_{pq} < 0$, а остальные координаты неотрицательны.

Легко проверить, что вектор z является решением системы линейных ограничений ЗЛП с матрицей A. Поэтому, в силу эквивалентности ЗЛП с матрицами A и A', значения их целевых функционалов на z должны быть равны, т.е. должно выполняться равенство

$$a_{00} + \sum_{j=1}^{n} a_{0j} z_j = a'_{00} + \sum_{j=1}^{n} a'_{0j} z_j.$$

Однако поскольку $a_{0j} = 0$ при $j \in \{s_1, s_2, \ldots, s_m\}$, $z_q = 1$ и $z_j = 0$ при $j \in \{1, 2, \ldots, n\} \setminus q \setminus \{s_1, s_2, \ldots, s_m\}$, то значение левой части равно $a_{00} + a_{0q} < a_{00}$. Значение правой части больше a_{00}, т.к. $a'_{0j} \geq 0$ и $z_j \geq 0$ при $1 \leq j \leq n-1$, $a'_{0n} < 0$, $z_n < 0$, $a'_{00} = a_{00}$. Полученное противоречие завершает доказательство конечности итерационного цикла, т.е. при решении ЗЛП(min), через конечное число шагов обязательно выполнится либо условие (6), либо условие (8).

Случай ЗЛП(max) можно рассмотреть аналогичным образом при минимальных изменениях в знаках неравенств. В этом случае цикл завершится выполнением (7) или (8). **Лемма 2 доказана.**

В завершение пункта отметим, что хотя итерационный цикл с правилом 2 конечен, но на вопрос о трудоемкости этапа 3, а значит и всего нашего метода, как и всего симплекс-метода, получить столь однозначный ответ не удается. Дело в том, что число всевозможных наборов базисных номеров в общем случае равно C_n^m, т.е. при больших n и m очень велико. В [7] даются специально подобранные примеры, когда при выборе разрешающего элемента по аналогу правила 1 итерационный цикл перебирает практически все комбинации базисных номеров, и там же утверждается, что такие примеры удается построить для всех известных способов уточнения этого правила. Это

означает, что есть ЗЛП, для которых симплекс-метод весьма неэффективен. С другой стороны, при решении практических задач симплекс-метод работает очень быстро, оказывается исключительно эффективным и нет упоминаний, что когда-нибудь на практике встретилась задача, требующая перебора такого большого числа комбинаций.

5. Перейдем к обсуждению этапа 2, который должен 1-каноническую матрицу преобразовывать в аналогичную матрицу, удовлетворяющую условию (5). Мы уже отмечали, что практически во всех известных нам работах ([1—8] и др.), где излагается симплекс-метод , этап 2 сводится к этапу 3 (или объединяется с этапом 3, что менее удобно). Для этого используется метод искусственных переменных Ордена. Исключение составляют работы [9, 10], в которых ограничения заданы неравенствами. Для таких специфичных ЗЛП добавление искусственных переменных одновременно превращает неравенства в равенства и обеспечивает неотрицательность базисного решения. К исключениям еще относится редко где упоминаемая публикация [14], в которой использован прием, предполагающий возможность зацикливания, но чаще всего приводящий к нужному результату.

Метод Ордена, адаптированный к нашим формулировкам, выглядит следующим образом. Каждое уравнение с $a_{i0} > 0$, умножается на -1 и к нему добавляется искусственная переменная (для каждого такого уравнения своя), а остальные уравнения оставляются без изменений. Кроме этой искусственной системы линейных ограничений, теперь уже удовлетворяющей условия (5), формируется искусственный целевой функционал, в виде суммы искусственных переменных, добавленных к уравнениям. Ясно, что у полученной искусственной ЗЛП минимум на неотрицательных решениях не может быть меньше нуля. Если этот минимум равен нулю, то у неотрицательных решений искусственной системы линейных ограничений, на которых он достигается, все искусственные переменные равны нулю.

Полученная искусственная ЗЛП(min) приводится к эквивалентной задаче с 1-канонической матрицей вычитанием из ее целевого функционала функционалов модифицированных уравнений. В результате получается эквивалентная ЗЛП, имеющая матрицу канонической структуры и удовлетворяющая условию (5). После этого минимум искусственной ЗЛП отыскивается с помощью основной процедуры, т.е. процедуры этапа 3. Если найденный минимум окажется положительным, то исходная ЗЛП не имеет неотрицательных решений системы ограничений. Если же минимум равен нулю, то в полученном базисном решении все искусственные переменные равны нулю и либо сразу ни одна из них не входит в число базисных, либо дополнительными преобразованиями, не нарушающими (5), они легко исключаются из числа базисных. Например, если искусственная переменная с номером $r > n$ является базисной, то $r = s_p$ при некотором p, и $a_{p0} = 0$, поскольку искусственная переменная равна нулю. Выбрав в p-й строке разрешающий элемент $a_{pq} \neq 0$ с $q \leq n$ и выполнив преобразование по формулам (9), мы, не меняя в матрице нулевой столбец, исключим искусственную переменную из числа базисных, заменив ее на обычную переменную.

После исключения всех искусственных переменных следует вычеркнуть из полученной матрицы столбцы, соответствующие искусственным переменным, а нулевую строку заменить строкой коэффициентов целевого функционала исходной задачи. Выполнение этапа 2 завершается исключением из этой строки всех элементов с базисными номерами.

Заметим, что мы изложили алгоритм с искусственными переменными, в основном, чтобы не быть голословными, критикуя общепринятый метод, используемый уже в течение многих десятилетий. Основным недостатком этого метода является увеличение размеров матрицы задачи, что крайне нежелательно при больших m и n. Выбранный нами способ представления ЗЛП позволяет увидеть, что необходимость такого увеличения отсутствует.

Специфика способа в том, что все строки матрицы, включая строку с номером 0, составлены из коэффициентов функционалов. Строка коэффициентов целевого функционала отличается от

других лишь тем, что в ней не выбираются разрешающие элементы. Учитывая это обстоятельство, с помощью основной процедуры можно постепенно увеличивать количество строк матрицы с $a_{i0} \le 0$ ($i \ne 0$), в результате чего либо выполнится условие (5), либо в ходе процесса появится строка, не имеющая элементов противоположных знаков, что означает отсутствие неотрицательных решений системы ограничений.

Опишем подробнее процесс увеличения количества строк с $a_{i0} \le 0$.

Если в канонической матрице все $a_{i0} > 0$, то выберем разрешающий элемент a_{1q} с $a_{1q} < 0$ (при отсутствии такого элемента система линейных ограничений не имеет неотрицательных решений) и преобразуем всю матрицу по формулам (9). После этого в матрице уже есть строки с $a_{i0} \le 0$ и, поскольку перестановка строк является эквивалентным преобразованием, можно считать, что $a_{i0} \le 0$ при $1 \le i < k$ и $a_{i0} > 0$ при $k \le i \le m$. Попытаемся с помощью основной процедуры, минимизировать a_{k0} на множестве неотрицательных решений первых $k-1$ уравнений. Для этого на каждом шаге процедуры разрешающие элементы надо выбирать в строках с номерами $1, \dots, k-1$, а номер q разрешающего столбца надо выбирать по k-й строке. При этом, чтобы матрица оставалась 1-канонической, следует преобразовывать элементы всей матрицы.

Пусть при таком применении основной процедуры минимум будет найден, т.е. будет выполнен аналог (6). Тогда либо неравенство $a_{k0} > 0$ сохранится и у исходной ЗЛП нет неотрицательных решений системы линейных ограничений, т.к. в k-й строке нет элементов разных знаков, либо k-я строка тоже попадет в число строк с $a_{i0} \le 0$.

Если же основная процедура установит неограниченность a_{k0} (обнаружится $a_{kq} < 0$ и $a_{iq} \le 0$ при всех $1 \le i \le k-1$), то выбрав a_{kq} разрешающим элементом и сделав шаг исключения опять получим $a_{k0} \le 0$. **Описание этапа 2 и всего алгоритма завершено.**

6. В данном пункте коротко обсудим некоторые следствия изложенных алгоритмов. Эти следствия касаются систем линейных алгебраических уравнений и играют важную роль в теории двойственности.

Сначала обратим внимание, что термин базисное решение относится именно к системе уравнений. Нетрудно заметить, что к базисным относятся те и только те решения системы, при которых столбцы матрицы системы, соответствующие ненулевым компонентам решения, линейно независимы.

Алгоритм этапа 1, по существу, совпадает с основной частью метода Гаусса полных исключений. Этот алгоритм либо приводит матрицу системы линейных ограничений к виду, из которого сразу находится одно из базисных решений, либо устанавливает факт отсутствия решений системы, который проявляется тем, что некоторая линейная комбинация уравнений системы представляет собой линейное уравнение, у которого свободный член отличен от нуля, а все коэффициенты при неизвестных равны нулю.

Таким образом, из существования решения системы следует существование базисного решения, и, вообще, справедлива следующая альтернатива.

Лемма 3. *Всякая система линейных уравнений либо имеет (базисное) решение, либо существует линейная комбинация уравнений системы, которая не имеет решения.*

Это простое утверждение можно переформулировать еще несколькими способами. Например, можно опустить взятое нами в скобки слово "базисное"или сформулировать лемму в виде критерия.

Лемма 4. *Система линейных алгебраических уравнений имеет решение в том и только том случае, когда всякая линейная комбинация уравнений тоже имеет решение.*

Аналогичным образом, второй вариант реализации этапа 2 (с учетом лемм 3,4) дает конструктивное доказательство эквивалентности существования неотрицательных базисных и обычных решений, а также доказательство следующей альтернативы.

Теорема 1. *Система линейных уравнений либо имеет неотрицательное решение, либо существует линейная комбинация уравнений системы, которая не имеет неотрицательного решения.*

Эту теорему тоже можно переформулировать еще несколькими способами. В форме критерия она примет следующий вид.

Теорема 2. *Система линейных алгебраических уравнений имеет неотрицательное решение тогда и только тогда, когда всякая линейная комбинация ее уравнений тоже имеет неотрицательное решение.*

Эти лаконичные, интуитивно понятные и удобные для запоминания теоремы на самом деле являются аналогами очень известной теоремы, которая обычно называется леммой Фаркаша (см., например, [2, 4—7, 10, 15, 16]). Теорема 1 является аналогом этой леммы в альтернативной форме, а теорема 2 – в форме критерия. Для сравнения с теоремой 2, приведем дословно одну из формулировок леммы Фаркаша из [2] (стр. 138).

Пусть A и b соответственно матрица и вектор-столбец. Существование неотрицательного решения $x \geq 0$ системы $Ax = b$ эквивалентно тому, что $yb \geq 0$ для любого вектора-строки y, удовлетворяющего $yA \geq 0$.

Ясно, что здесь условие "$yb \geq 0$ для любого вектора-строки y, удовлетворяющего $yA \geq 0$" является одним из способов выразить, что у каждой линейной комбинации уравнений системы есть неотрицательные решения. Конечно, можно было бы в этом условии знаки $\geq$ поменять на $\leq$ или заменить все условия каким-либо другим эквивалентом существования у скалярных уравнений неотрицательного решения.

Аналогичным образом теорему 1 можно было бы сравнить с теоремой 7.1 из [2], которая там называется основной теоремой о линейных неравенствах и связывается с именами Фаркаша, Минковского, Каратеодори и Вейля.

Вообще, стоит отметить, что лемма Фаркаша отличается исключительным многообразием формулировок и их доказательств. Некоторые из этих формулировок довольно сложны, ввиду используемой терминологии, обозначений и особых акцентов. Надеемся, что простые формулировки теорем 1 и 2 будут способствовать лучшему пониманию более сложных формулировок.

7. Приведенный в пункте 5 второй вариант отыскания неотрицательного базисного решения более экономно использует вычислительные ресурсы, но оказалось, что есть существенно более простой способ, полученный уже при написании данной статьи и анонсированный в [17].

Еще в 1996 году, экспериментируя с программой, которая при преобразовании 1-канонической матрицы ЗЛП позволяла выбирать разрешающие элементы a_{pq} в диалоговом режиме, автор заметил, что неотрицательные базисные решения очень быстро отыскивались интуитивным подбором a_{pq}. Фактически процесс сводился к итерационному циклу, где сначала выбиралось p, при котором $a_{p0} > 0$, затем выбиралось q, при котором $a_{pq} < 0$, а затем выполнялось преобразование матрицы по формулам (9). Цикл заканчивался, когда либо выполнялось условие (5), либо появлялась строка, в которой все a_{pj} были неотрицательны и $a_{p0} > 0$, что означает отсутствие неотрицательных решений. Очевидно, что если каким-то образом уточнить это правило выбора (например, в качестве p и q брать наименьшие из номеров, при которых $a_{p0} > 0$ и $a_{pq} < 0$), то цикл либо закончится по одной из указанных двух причин, либо произойдет зацикливание. Однако в ходе численных экспериментов зацикливание не встречалось, и автор даже рекомендовал студентам использовать такой эвристический прием при выполнении лабораторных работ.

Уже когда работа над данной статьей завершалась, удалось получить и обосновать очень просто формулируемое правило, которое вообще исключает зацикливание.

Правило 3. *Для 1-канонической матрицы в качестве номера разрешающей строки из всех p, при которых $a_{p0} > 0$, надо брать номер с наименьшим r_p. При уже выбранном p надо брать наименьшее q, при котором $a_{pq} < 0$.*

Теорема 3. *Для 1-канонических матриц итерационный цикл, тело которого состоит из выбора разрешающего элемента по правилу 3 и последующего шага исключения, конечен.*

Еще раз обратим внимание, что в теореме, по сути, речь идет о конечности простого алгоритма решения важной задачи отыскания неотрицательного базисного решения, которая до сих пор во всех известных нам публикациях решалась методом искусственного базиса или еще более сложными методами. Несмотря на простоту приведенного правила выбора разрешающего элемента, доказательство теоремы весьма непросто, связано с обоснованием еще нескольких алгоритмов решения ЗЛП и требует написания отдельной статьи.

8. В этом пункте приведем четыре примера, иллюстрирующих решение ЗЛП с использованием рассмотренного в статье подхода. Напомним, что алгоритм заключается в приведении ЗЛП к виду, при котором выполнены условия леммы 1, и состоит из трех этапов, некоторые из которых могут не понадобиться. Каждый этап представляет из себя цикл, в теле которого соответствующим образом выбирается разрешающий элемент, а затем матрица преобразуется по формулам (9). Нумерация строк и столбцов начинается с нуля. Разрешающий элемент берется из строк и столбцов с номерами больше нуля. Значение целевого функционала на текущем базисном решении присутствует в таблице в явном виде как элемент a_{00}. Мы будем на каждой итерации помечать разрешающий столбец звездочкой в нулевой строке, разрешающую строку – звездочкой в нулевом столбце, а сам разрешающий элемент – двумя звездочками.

Пример 1.

Требуется решить ЗЛП(min) и ЗЛП(max) с 1-канонической матрицей

$$\begin{pmatrix} 1 & 0 & 0 & 5 & 12 & 18 & -41 \\ 0 & 0 & 1 & -3 & -5 & 7 & 16 \\ 0 & 1 & 0 & 1 & 2 & -3 & 7 \end{pmatrix}.$$

Эти задачи вырожденные, условие (5) выполняется автоматически и требуется выполнять лишь этап 3. Поскольку при исключении переменных свободный член целевого функционала не меняется, то максимум и минимум могут иметь лишь значение $a_{00} = 1$, хотя могут и отсутствовать.

Сначала решим ЗЛП(min), выбирая в качестве p и q наименьшие номера, удовлетворяющие правилу 1.

1	0	0	5	12	-18*	-41
0*	0	1	-3	-5	7**	16
0	1	0	1	2	-3	-7
1	0	18/7	-19/7*	-6/7	0	1/7
0	0	1/7	-3/7	-5/7	1	16/7
0	1	3/7	-2/7	-1/7	0	-1/7

Уже на второй итерации, после выбора $q = 3$ в столбце с этим номером нет положительных элементов. Это означает, что выполнено условие (8) и целевая функция неограничена снизу.

Применяя аналогичный способ уточнения правила 1 для решения ЗЛП(max), получаем следующую последовательность итераций.

1	0	0	5*	12	-18	-41
0	0	1	-3	-5	7	16
0*	1	0	1**	2	-3	-7
1	-5	0	0	2*	-3	-6
0*	3	1	0	1**	-2	-5
0	1	0	1	2	-3	-7
1	-11	-2	0	0	1*	4
0	3	1	0	1	-2	-5
0*	-5	-2	1	0	1**	3
1	-6	0	-1	0	0	1*
0*	-7	-3	2	1	0	1**
0	-5	-2	1	0	1	3
1	1*	3	-3	-1	0	0
0	-7	-3	2	1	0	1
0*	16**	7	-5	-3	1	0
1	0	41/16*	-43/16	-13/16	-1/16	0
0*	0	1/16**	-3/16	-5/16	7/16	1
0	1	7/16	-5/16	-3/16	1/16	0
1	0	0	5	12	-18	-41
0	0	1	-3	-5	7	16
0	1	0	1	2	-3	-7

После шестой итерации мы получили исходную матрицу, т.е. произошло зацикливание. Чтобы этого не случилось, надо было использовать правило 2, которое в любом случае исключает зацикливание. Применяя это правило к нашей задаче, убеждаемся, что вплоть до шестой итерации выбираются те же разрешающие элементы. На шестой итерации, после выбора $q = 2$, по правилу 2 надо выбрать $p = 2$, т.к. $r_1 = 6$, а $r_2 = 1$. Таким образом, начиная с шестой итерации, вычисления будут иметь следующий вид.

1	0	41/16*	-43/16	-13/16	-1/16	0
0	0	1/16	-3/16	-5/16	7/16	1
0*	1	7/16**	-5/16	-3/16	1/16	0
1	-41/7	0	-6/7	2/7*	-3/7	0
0	-1/7	0	-1/7	-2/7	3/7	1
0	16/7	1	-5/7	-3/7	1/7	0

На седьмой итерации выбирается $q = 4$, а разрешающего элемента выбрать не удается, т. к. в четвертом столбце вне нулевой строки нет положительных элементов. Это означает, что выполнено условие (8) для ЗЛП(max) и целевая функция сверху тоже неограничена. **Конец примера.**

Пример 2. Требуется решить ЗЛП(min) и ЗЛП(max) с матрицей

$$\begin{pmatrix} 0 & -2 & -2 & 1 & -2 & -1 & 0 & -1 \\ 1 & -2 & -1 & -1 & -2 & -1 & -2 & 1 \\ 2 & -2 & -1 & 1 & -1 & 1 & -1 & 0 \\ 3 & 1 & 0 & -1 & -2 & -1 & -1 & -2 \end{pmatrix}.$$

Этап 1 совпадает с основной частью метода полных исключений Гаусса.

0	-2*	-2	1	-2	-1	0	-1
1*	-2**	-1	-1	-2	-1	-2	1
2	-2	-1	1	-1	1	-1	0
3	1	0	-1	-2	-1	-1	-2
-1	0	-1	2*	0	0	2	-2
-1/2	1	1/2	1/2	1	1/2	1	-1/2
1*	0	0	2**	1	2	1	-1
7/2	0	-1/2	-3/2	-3	-3/2	-2	-3/2
-2	0	-1*	0	-1	-2	1	-1
-3/4	1	1/2	0	3/4	0	3/4	-1/4
1/2	0	0	1	1/2	1	1/2	-1/2
17/4*	0	-1/2**	0	-9/4	0	-5/4	-9/4
-21/2	0	0	0	7/2	-2	7/2	7/2
7/2	1	0	0	-3/2	0	-1/2	-5/2
1/2	0	0	1	1/2	1	1/2	-1/2
-17/2	0	1	0	9/2	0	5/2	9/2

Теперь матрица ЗЛП является 1-канонической. Этап 2 будем реализовывать способом, изложенном в пункте 7, т. е. итерационным циклом с правилом 3.

-21/2	0	0	0	7/2*	-2	7/2	7/2
7/2*	1	0	0	-3/2**	0	-1/2	-5/2
1/2	0	0	1	1/2	1	1/2	-1/2
-17/2	0	1	0	9/2	0	5/2	9/2
-7/3	7/3	0	0	0	-2	7/3	-7/3*
-7/3	-2/3	0	0	1	0	1/3	5/3
5/3	1/3	0	1	0	1	1/3	-4/3
2*	3	1	0	0	0	1	-3**
-35/9	0*	-7/9	0	0	-2	14/9	0
-11/9	1	5/9	0	1	0	8/9	0
7/9*	-1**	-4/9	1	0	1	-1/9	0
-2/3	-1	-1/3	0	0	0	-1/3	1
-35/9	0	-7/9	0	0	-2	14/9	0
-4/9	0	1/9	1	1	1	7/9	0
-7/9	1	4/9	-1	0	-1	1/9	0
-13/9	0	1/9	-1	0	-1	-2/9	1

Получили 1-каноническую матрицу ЗЛП, удовлетворяющую условию (5). Этап 3 будем реализовывать итерационным циклом с правилом 2. Сначала решим ЗЛП(min).

-35/9	0	-7/9*	0	0	-2	14/9	0
-4/9	0	1/9	1	1	1	7/9	0
-7/9*	1	4/9**	-1	0	-1	1/9	0
-13/9	0	1/9	-1	0	-1	-2/9	1
-21/4	7/4	0	-7/4*	0	-15/4	7/4	0
-1/4*	-1/4	0	5/4**	1	5/4	3/4	0
-7/4	9/4	1	-9/4	0	-9/4	1/4	0
-5/4	-1/4	0	-3/4	0	-3/4	-1/4	1
-28/5	7/5	0	0	7/5	-2*	14/5	0
-1/5*	-1/5	0	1	4/5	1**	3/5	0
-11/5	9/5	1	0	9/5	0	8/5	0
-7/5	-2/5	0	0	3/5	0	1/5	1
-6	1	0	2	3	0	4	0
-1/5	-1/5	0	1	4/5	1	3/5	0
-11/5	9/5	1	0	9/5	0	8/5	0
-7/5	-2/5	0	0	3/5	0	1/5	1

Условие (6) выполнено. По лемме 1 минимум целевого функционала на неотрицательных решениях равен $a_{00} = -6$. Он достигается на неотрицательном базисном решении $x = (0, 11/5, 0, 0, 1/5, 0, 7/5)$.

Решим ЗЛП(max).

-35/9	0	-7/9	0	0	-2	14/9*	0
-4/9*	0	1/9	1	1	1	7/9**	0
-7/9	1	4/9	-1	0	-1	1/9	0
-13/9	0	1/9	-1	0	-1	-2/9	1
-3	0	-1	-2	-2	-4	0	0
-4/7*	0	1/7	9/7	9/7	9/7	1	0
-5/7	1	3/7	-8/7	-1/7	-8/7	0	0
-11/7	0	1/7	-5/7	2/7	-5/7	0	1

Теперь условие (7) выполнено. Максимум целевого функционала на неотрицательных решениях равен $a_{00} = -3$. Он достигается на неотрицательном базисном решении $x = (5/7, 0, 0, 0, 0, 4/7, 11/7)$. **Задачи примера 2 решены.**

Пример 3. Требуется решить ЗЛП(min) с матрицей

$$\begin{pmatrix} -2 & 1 & 1 & -2 & -2 & 0 & -1 & 0 \\ 0 & 0 & -1 & 1 & -2 & 0 & -2 & -2 \\ 1 & 1 & -2 & -1 & 1 & 0 & -2 & -1 \\ -1 & -1 & 1 & 1 & -1 & 1 & 0 & -2 \\ 1 & -2 & 2 & 3 & -4 & 1 & 0 & -3 \end{pmatrix}.$$

Этап 1. Опять используем схему полных исключений Гаусса.

-2	1	1^{*}	-2	-2	0	-1	0
0^{*}	0	-1^{**}	1	-2	0	-2	-2
1	1	-2	-1	1	0	-2	-1
-1	-1	1	1	-1	1	0	-2
1	-2	2	3	-4	1	0	-3
-2	1^{*}	0	-1	-4	0	-3	-2
0	0	1	-1	2	0	2	2
1^{*}	1^{**}	0	-3	5	0	2	3
-1	-1	0	2	-3	1	-2	-4
1	-2	0	5	-8	1	-4	-7
-3	0	0	2^{*}	-9	0	-5	-5
0	0	1	-1	2	0	2	2
1	1	0	-3	5	0	2	3
0^{*}	0	0	-1^{**}	2	1	0	-1
3	0	0	-1	2	1	0	-1
-3	0	0	0	-5	2	-5	-7
0	0	1	0	0	-1	2	3
1	1	0	0	-1	-3	2	6
0	0	0	1	-2	-1	0	1
3^{*}	0	0	0	0	0	0	0

В последней строке элемент нулевого столбца положителен, а остальные элементы равны нулю. Это означает, что система линейных ограничений поставленной задачи не имеет решений. **Конец примера 3.**

Пример 4. Решить ЗЛП(min) с 1-канонической матрицей

$$\begin{pmatrix} -1/2 & 0 & 0 & 0 & 3/2 & 0 & 1/2 & -5/2 \\ 7/8 & 1 & 0 & 0 & -7/8 & -5/4 & -9/8 & -3/8 \\ -3/8 & 0 & 1 & 0 & 11/8 & 5/4 & 5/8 & -1/8 \\ 3/8 & 0 & 0 & 1 & 5/8 & 3/4 & 3/8 & 1/8 \end{pmatrix}.$$

Поскольку матрица ЗЛП является 1-канонической, решение начинается сразу с этапа 2, который опять будем реализовывать итерационным циклом с правилом 3.

-1/2	0	0	0	3/2*	0	1/2	-5/2
7/8*	1	0	0	-7/8**	-5/4	-9/8	-3/8
-3/8	0	1	0	11/8	5/4	5/8	-1/8
3/8	0	0	1	5/8	3/4	3/8	1/8
1	12/7	0	0	0	-15/7*	-10/7	-22/7
-1	-8/7	0	0	1	10/7	9/7	3/7
1*	11/7	1	0	0	-5/7**	-8/7	-5/7
1	5/7	0	1	0	-1/7	-3/7	-1/7
-2	-3	-3*	0	0	0	2	-1
1	2	2	0	1	0	-1	-1
-7/5	-11/5	-7/5	0	0	1	8/5	1
4/5*	2/5	-1/5**	1	0	0	-1/5	0
-14	-9	0	-15	0	0	5*	-1
9*	6	0	10	1	0	-3**	-1
-7	-5	0	-7	0	1	3	1
-4	-2	1	-5	0	0	1	0
1	1	0	5/3	5/3	0	0	-8/3
-3	-2	0	-10/3	-1/3	0	1	1/3
2*	1	0	3	1	1	0	0
-1	0	1	-5/3	1/3	0	0	-1/3

В строке с номером 2 все элементы неотрицательны, а элемент нулевого столбца строго положителен. Это означает, что у системы линейных ограничений вообще нет неотрицательных решений. **Конец примера 4.**

В заключение отметим, что выбор разрешающего элемента на этапе 1 и на этапе 2, при использовании правила 3, не зависит от целевого функционала. Поэтому на этих этапах при ручных вычислениях, для экономии, нулевую строку можно не преобразовывать, а преобразовать ее тогда, когда уже будет выполняться условие (5), т.е. непосредственно перед этапом 3.

References

[1] S. I. Gass, *Linear Programming: Methods and Applications*. McGraw-Hill: New York, 1958.

[2] A. Schrijver, *Theory of linear and integer programming*. Moscow: Mir, 1991.

[3] D. B. Yudin and E. G. Holstein, *Linear Programming: Theory and finite methods*. Moscow: Fizmatlit, 1963.

[4] G. B. Dantzig, *Linear Programming and Extensions*. Princeton University Press, 1963.

[5] T. S. Hu, *Integer programming and network flows*. Addison-Wesley, 1969.

[6] S. A. Ashmanov, *Linear Programming*. Moscow: Nauka, 1981.

[7] C. H. Papadimitriou and K. Steiglitz, *Combinatorial optimization: algorithms and complexity*. Prentice-Hall, 1982.

[8] F. P. Vasiliev and A. Y. Ivanitsky, *Linear Programming*. MCCME, 2020.

[9] E. Stiefel, “Note on Jordan elimination, linear programming and Tchebycheff approximation”, *Numerische Mathematik*, vol. 2, no. 1, pp. 1–17, 1960.

[10] S. I. Zukhovitsky and L. I. Avdeeva, *Linear and Convex Programming*. Moscow: Nauka, 1967.

[11] A. Charnes, "Optimality and degeneracy in linear programming", *Econometrica: Journal of the Econometric Society*, vol. 20, no. 2, pp. 160–170, 1952.

[12] R. G. Bland, "New finite pivoting rules for the simplex method", *Mathematics of Operations Research*, vol. 2, no. 2, pp. 103–107, 1977.

[13] H. W. Kuhn, *Class Notes*. Princeton University, 1976.

[14] A. S. Barsov, *What is Linear Programming*. Moscow: Fizmatgiz, 1959.

[15] V. G. Karmanov, *Mathematical Programming*. Moscow: Fizmatlit, 2004.

[16] Y. G. Evtushenko, A. A. Tret'yakov, and E. E. Tyrtyshnikov, "New approach to Farkas' theorem of the alternative", in *Doklady Mathematics*, vol. 99, Springer, 2019, pp. 208–210.

[17] G. D. Stepanov, "A Simple Algorithm for Finding a Non-negative Basic Solution of a System of Linear Algebraic Equations", *Modeling and analysis of information systems*, vol. 28, no. 3, pp. 234–237, 2021.

MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 28, NO. 4, 2021
journal homepage: www.mais-journal.ru

COMPUTER SYSTEM ORGANIZATION

An Algorithm for Estimating the Signal Frequency at the Output of a Channel with a Controlled Information Flow under Phase Noise Conditions

L. N. Kazakov[1], E. P. Kubyshkin[1], I. V. Lukyanov[2]

DOI: 10.18255/1818-1015-2021-4-452-461

[1]P. G. Demidov Yaroslavl State University, 14 Sovetskaya str., Yaroslavl 150003, Russia.
[2]OOO "NPP "LAMA" ", 89 Serova str, Rybynsk 152907, Russia.

MSC2020: 68W25, 68W40
Research article
Full text in Russian

Received November 15, 2021
After revision December 1, 2021
Accepted December 8, 2021

Research in the field of efficient frequency estimation algorithms is of great interest. The reason for this is the redistribution of the role of additive and phase noise in many modern radio-engineering applications. An example is the area of measuring radio devices, which usually operate at high signal-to-noise ratios (SNR). The estimation error is largely determined not by the broadband noise, but by the frequency and phase noise of the local oscillators of the receiving and transmitting devices. In particular, earlier works [1] proposed an efficient computational algorithm for estimating the frequency of a quasi-harmonic signal based on the iterative calculation of the autocorrelation sequence (ACS). In [2], this algorithm was improved and its proximity to the Rao-Cramer boundary was shown (the sources of this noise are master oscillators and frequency synthesizers). Possibilities of frequency estimation in radio channels make it possible to significantly expand the functionality of the entire radio network. This can include, for example, the problem of adaptive distribution of information flows of a radio network. This also includes the tasks of synchronization and coherent signal processing. For these reasons, more research is needed on this algorithm, the calculation of theoretical boundaries and their comparison with the simulation results.

Keywords: Rao-Cramer boundary; autocorrelation function; frequency instability; phase noise; Fisher information

INFORMATION ABOUT THE AUTHORS

Leonid Nikolaevich Kazakov	orcid.org/0000-0002-9348-4440. E-mail: kazakov@uniyar.ac.ru Doctor of Science, Professor .
Evgenii Pavlovich Kubyshkin correspondence author	orcid.org/0000-0003-1796-0190. E-mail: kubysh.e@yandex.ru Doctor of Science, Professor.
Ilya Victorovich Lukyanov	orcid.org/0000-0002-1751-8557. E-mail: il-lukyanov@yandex.ru Research associate.

Funding: The reported study was funded by YSU Programme according to the research project № P2-K-1-G-4/2021.

For citation: L. N. Kazakov, E. P. Kubyshkin, and I. V. Lukyanov, "An Algorithm for Estimating the Signal Frequency at the Output of a Channel with a Controlled Information Flow under Phase Noise Conditions", *Modeling and analysis of information systems*, vol. 28, no. 4, pp. 452-461, 2021.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 28, № 4, 2021
сайт журнала: www.mais-journal.ru

COMPUTER SYSTEM ORGANIZATION

Алгоритм оценивания частоты сигнала на выходе канала с управляемым информационным потоком в условиях фазового шума

Л. Н. Казаков[1], Е. П. Кубышкин [1], И. В. Лукьянов[2]

DOI: 10.18255/1818-1015-2021-4-452-461

[1]Ярославский государственный университет им. П. Г. Демидова, ул. Советская, д. 14, г. Ярославль, 150003 Россия.
[2]ООО «НПП «ЛАМА», пр-т Серова, д. 89, г. Рыбинск, 152907 Россия.

УДК 519.7, 621.396.96
Научная статья
Полный текст на русском языке

Получена 15 ноября 2021 г.
После доработки 1 декабря 2021 г.
Принята к публикации 8 декабря 2021 г.

Большой интерес вызывают исследования в области эффективных алгоритмов оценивания частоты. Причиной этому является перераспределение во многих современных радиотехнических приложениях роли аддитивного и фазового шумов. Примером может служить область измерительных радиоприборов, работающих, как правило, при высоких отношениях сигнал/шум (ОСШ). Ошибка оценки в большей степени определяется не широкополосным шумом, а частотными и фазовыми шумами гетеродинов приемных и передающих устройств. В частности, в более ранних работах [1] предложен эффективный вычислительный алгоритм оценивания частоты квазигармонического сигнала, основанный на итерационном вычислении АКП. В [2] этот алгоритм доработан и показана его близость к границе Рао-Крамера (источниками этих шумов являются задающие генераторы и синтезаторы частоты). Возможности оценивания частоты в радиоканалах позволяют существенно расширить функционал всей радиосети. Сюда можно отнести, например, задачу адаптивного распределения информационных потоков радиосети. Сюда же можно отнести задачи синхронизации и когерентной обработки сигналов. По этим причинам необходимы дополнительные исследования этого алгоритма, расчет теоретических границ и их сравнение с результатами моделирования.

Ключевые слова: граница Рао-Крамера; автокорреляционная функция; частотная нестабильность; фазовый шум; информация Фишера

ИНФОРМАЦИЯ ОБ АВТОРАХ

Леонид Николаевич Казаков — orcid.org/0000-0002-9348-4440. E-mail: kazakov@uniyar.ac.ru
доктор технических наук, профессор.

Евгений Павлович Кубышкин — orcid.org/0000-0003-1796-0190. E-mail: kubysh.e@yandex.ru
автор для корреспонденции
доктор физико-математических наук, профессор.

Илья Викторович Лукьянов — orcid.org/0000-0002-1751-8557. E-mail: il-lukyanov@yandex.ru
научный сотрудник .

Финансирование: Исследование выполнено в рамках Программы развития ЯрГУ, проект № П2-К-1-Г-4/2021.

Для цитирования: L. N. Kazakov, E. P. Kubyshkin, and I. V. Lukyanov, “An Algorithm for Estimating the Signal Frequency at the Output of a Channel with a Controlled Information Flow under Phase Noise Conditions”, *Modeling and analysis of information systems*, vol. 28, no. 4, pp. 452-461, 2021.

Введение

В [1] предложен эффективный вычислительный алгоритм оценивания частоты квазигармонического сигнала, основанный на итерационном вычислении АКП. В [2] этот алгоритм доработан и показана его близость к границе Рао-Крамера (ГРК). В этих публикациях, как и в большинстве других известных работ, посвященных алгоритмам оценивания частоты, представлены результаты моделирования в условиях аддитивного белого гауссовского шума (АБГШ). Однако в измерительных радиосистемах, работающих, как правило, при высоких отношениях сигнал/шум (ОСШ), ошибка оценки в большей степени определяется не широкополосным шумом, а частотными и фазовыми шумами гетеродинов приемных и передающих устройств. Источниками этих шумов являются задающие генераторы и синтезаторы частоты [3]. По этим причинам необходимы дополнительные исследования этого алгоритма, расчет теоретических границ и их сравнение с результатами моделирования.

1. МОДЕЛИ ФАЗОВОГО ШУМА

Модель комплексного сигнала на выходе канала с АБГШ и фазовым шумом можно записать в виде

$$s(t) = A\exp\left(j[2\pi f_0 t + \varphi(t)]\right) + \dot{w}(t),$$

где t – время; A – амплитуда; f_0 – номинальная частота; $\varphi(t)$ – фазовая функция сигнала; $\dot{w}(t)$ – АБГШ.

Под частотным и фазовым шумами понимаются функции:

$$\Delta f(t) = f_0 - \hat{f}(t),\ \Delta\phi(t) = \varphi(t) - \hat{\varphi}(t),$$

где $\hat{f}(t)$, $\hat{\varphi}(t)$ – фактические мгновенные значения частоты и фазы сигнала.

Частотные и фазовые шумы генераторов характеризуются следующими функциями [3, 4]:

– спектральная плотность фазовых флуктуаций

$$S_\varphi(f) = \lim_{T\to+\infty}\frac{2}{T}\left|\int_{-T/2}^{T/2}\Delta\varphi(t)\exp\left(-j2\pi f t\right)dt\right|^2;$$

– спектральная плотность частотных флуктуаций

$$S_F(f) = \lim_{T\to+\infty}\frac{2}{T}\left|\int_{-T/2}^{T/2}\Delta f(t)\exp\left(-j2\pi f t\right)dt\right|^2 = f^2 S_\varphi(f);$$

– спектральная плотность относительного отклонения частоты

$$S_Y(f) = \left(\frac{f}{f_0}\right)^2, S_\varphi(f) = \left(\frac{1}{f_0}\right)^2 S_F(f),$$

где T – интервал наблюдения.

К интегральным характеристикам частотной нестабильности относятся паразитные отклонения фазы (ПОФ) и паразитные отклонения частоты (ПОЧ), определяемые выражениями:

$$\sigma_\varphi(f_H, f_B) = \sqrt{\int_{f_H}^{f_B} S_\varphi(f)df},\ \sigma_f(f_H, f_B) = \sqrt{\int_{f_H}^{f_B} S_f(f)df}.$$

Общепризнанной моделью представления спектральной плотности фазовых и частотных флуктуаций является степенная модель [3]:

$$S_M(f) = \sum_a h_a f^a,$$

где a характеризует вид шума; h_a характеризует мощность шума.

В таблице 1 представлена классификация основных частотных и фазовых шумов генераторов [3, 4].

Table 1. Classification of frequency and phase noise

Таблица 1. Классификация частотного и фазового шума

Тип шума	Характеристика наклона	
	$S_Y(f) = h_a f^a$	$S_\varphi(f) = f_0^2 h_\beta f^\beta$
	a	β=a-2
Частотный шум случайных блужданий	-2	-4
Частотный фликкер-шум	-1	-3
Белый частотный шум	0	-2
Фазовый фликкер-шум	1	-1
Белый фазовый шум	2	0

2. АНАЛИЗ ТЕОРЕТИЧЕСКИХ ГРАНИЦ АЛГОРИТМА

Представим дискретную модель сигнала в условиях фазового шума и АБГШ в виде

$$x[n] = A\exp(j\varphi[n])\exp(j[2\pi f_0 n + \varphi_0]) + \dot{w}[n], \quad (1)$$

где n = 0, 1, ..., $N\lambda 1$ – номер отсчета; N – размер выборки; $\varphi[n]$ – отсчеты фазового шума; f_0 – нормированная номинальная частота; φ_0 – начальная фаза; $\dot{w}[n]$ – комплексные отсчеты АБГШ с дисперсией σ_w^2.

При исследовании алгоритмов оценивания интерес представляет сравнение среднеквадратической ошибки (СКО) оценки с некоторой теоретической границей. При этом СКО оценки определяется в соответствии с выражением

$$\varepsilon = \sqrt{\frac{1}{K}\sum_{i=0}^{K-1}\left(\hat{f}_i - f_0\right)^2}, \quad (2)$$

где K – число экспериментов; $\hat{f}_i$ – оценка нормированной частоты.

Общепризнанным количественным критерием алгоритмов оценивания является сравнение их СКО с границей Рао-Краммера (ГРК). Анализ публикаций показал, что на сегодняшний день для модели (1) в аналитическом виде (с рядом допущений) ГРК получена только для случая, когда $\varphi[n]$ является нормальным процессом [5]. Математические сложности при выводе ГРК обусловлены нестационарностью процессов, классифицированных в таблице 1, и невозможностью аналитического представления их автокорреляционных функций.

Рассмотрим эту проблему подробнее. Будем считать, что $\varphi[n]$ в общем случае является нестационарным процессом с ковариационной матрицей $\mathbf{Q}_\varphi[n]$ и математическим ожиданием $\mu_\varphi[n]$, где n означает зависимость статистических параметров от времени наблюдения.

ГРК в общем случае определяется элементами главной диагонали обратной матрицы Фишера [6] с элементами

$$[I(\Theta)]_{ij} = \left[\frac{\partial\mu(\Theta)}{\partial\Theta_i}\right]^T Q^{-1}(\Theta)\left[\frac{\partial\mu(\Theta)}{\partial\Theta_j}\right] + \frac{1}{2}tr\left[Q^{-1}(\Theta)\frac{\partial Q(\Theta)}{\partial\Theta_i}Q^{-1}(\Theta)\frac{\partial Q(\Theta)}{\partial\Theta_j}\right], \quad (3)$$

где $\boldsymbol{\Theta}$ – вектор оцениваемых параметров; μ и $\boldsymbol{\Theta}$ – математическое ожидание и ковариационная матрица процесса (1).

Для нестационарных процессов $\varphi[n]$ (когда показатель $\beta < 0$) аналитический расчет по (3) невозможен. Для этого требуется представление автокорреляционной функции процесса $\varphi[n]$ в стационарном приближении, что ставит под сомнение корректность полученного критерия.

А. Приближенный расчет СКО оценки по спектральной плотности фазового шума

Другим путем получения теоретического критерия, является использование выражений, представленных в работе [7]. Считая аддитивный шум и фазовые флуктуации независимыми процессами, а измерительную систему линейной, такой показатель можно записать в виде

$$\sigma_{\hat{f}}^2 = \frac{N_I}{N_I - 1}\int_{f_H}^{f_B} S_\phi(f)\frac{\sin^2(\pi\tau f)}{(\pi\tau)^2}\left[1 - \frac{\sin^2(T_\Pi\pi f N_I)}{N_I^2\sin^2(T_\Pi\pi f)}\right]df + \frac{6f_\partial}{(w\pi)^2 R_l\tau\left((\tau f_\partial)^2 - 1\right)}, \quad (4)$$

где первое слагаемое характеризует дисперсию частотных флуктуаций в соответствии с [7]; второе слагаемое характеризует ГРК оценки частоты в канале с АБГШ в соответствии с [6]; τ – интервал измерения; T_Π – период повторения измерений; N – количество измерений; R_l – ОСШ, выраженное в единицах; f_∂ – частота дискретизации.

Б. Расчет ГРК для белого фазового шума

Как было отмечено выше, ГРК для случая белого фазового шума может быть рассчитана аналитически. Положим в (1) $\varphi[n]$ центрированным нормальным процессом с дисперсией $\Delta = \sigma_\phi^2$. Для малых значений фазовых флуктуаций в соответствии с [5] модель (1) можно переписать в виде

$$x[n] \cong A(1 + j\varphi[n])\exp(j[2\pi f_0 n + \varphi_0]) + \dot{w}[n].$$

Воспользовавшись результатами Свинглера [6], ГРК можно записать в виде

$$var(\hat{f}) \cong \frac{6(2R_l\Delta + 1}{(2\pi)^2 R_l(2R_l\Delta^2 + 1)(N^3 - N)}. \quad (5)$$

Отметим, что оценка (5) при высоких уровнях ПОФ становится заниженной. Это связано с тем, что используемые при ее расчете формулы приближенного вычисления тригонометрических функций дают большую ошибку при больших значениях аргумента.

Другим путем получения приближенного выражения ГРК является представление аддитивного шума в виде белого фазового шума. В [8, 9] показано, что при высоких ОСШ модель (1) можно записать в виде

$$x[n] \cong A\exp j[2\pi f_0 n + \varphi_0 + \varphi[n] + \eta[n]],$$

где $\eta[n]$ – центрированный нормальный процесс с дисперсией ${\sigma_\eta}^2 = \sigma_w^2/2A^2$.

Тогда оценку ГРК можно получить, представив фазовую функцию в виде процесса

$$\phi[n] = 2\pi f n + \varphi_0 + \varphi[n] + \eta[n].$$

Приняв допущение о независимости $\varphi[n]$ и $\eta[n]$, выразим функцию правдоподобия и информационную матрицу Фишера:

$$L(\phi|f,\varphi_0) = \frac{1}{(2\pi\sigma^2)^{\frac{N}{2}}}\exp\left(-\frac{1}{2\sigma^2}\sum_{n=0}^{N-1}(\phi[n]-2\pi fn-\varphi_0)^2\right);$$

$$I(\Theta) = \begin{bmatrix} -E\left[\frac{\partial^2 \ln L(\phi|f,\varphi_0)}{\partial f^2}\right] & -E\left[\frac{\partial^2 \ln L(\phi|f,\varphi_0)}{\partial f\partial\varphi_0}\right] \\ -E\left[\frac{\partial^2 \ln L(\phi|f,\varphi_0)}{\partial f\partial\varphi_0}\right] & -E\left[\frac{\partial^2 \ln L(\phi|f,\varphi_0)}{\partial\varphi_0^2}\right] \end{bmatrix} = \frac{1}{\sigma^2}\begin{bmatrix} 2\pi\sum_{n=0}^{N-1} n^2 & 2\pi\sum_{n=0}^{N-1} n \\ 2\pi\sum_{n=0}^{N-1} n & N] \end{bmatrix} =$$

$$= \frac{1}{\sigma^2}\begin{bmatrix} \frac{N(N-1)(N-2)}{6} & \frac{N(N-2)}{2} \\ \frac{N(N-2)}{2} & N] \end{bmatrix},$$

где $\sigma^2 = \sigma_\eta^2 + \sigma_\varphi^2$.

Найдем обратную матрицу Фишера $I^{-1}(\Theta)$, и получим ГРК:

$$var(\hat{f}) \cong \frac{12\left(\sigma_\eta^2+\sigma_\varphi^2\right)}{(2\pi)^2 N(N^2-1)} = \frac{6\left(1+2R_l\sigma_\varphi^2\right)}{(2\pi)^2 R_l N(N^2-1)}, \quad (6)$$

где $R_l = \frac{A^2}{\sigma_w^2}$ – ОСШ.

В отсутствии фазового шума выражение (6) вырождается в ГРК на выходе канала с АБГШ [6], а при малом уровне ПОФ это выражение очень близко к результатам Свинглера (5).

3. РЕЗУЛЬТАТЫ МОДЕЛИРОВАНИЯ

Результаты сравнения СКО алгоритма (2) с критерием (4) при воздействии процессов в соответствии с таблицей 1 представлены на рисунках 1–5. Размерность выборки N=2048, количество экспериментов K=10000. Моделирование фазового шума выполнялось методом с учетом спектральной маски фазового шума, описанным в [10, 11]. ПОФ определялось по формуле

$$\sigma_\varphi(f_H, f_d) = \sqrt{\int_{f_H}^{f_d} S_\varphi(f)df},$$

где f_d – частота дискретизации; $f_H = 1/T$, T – полный интервал наблюдения.

На рис. 1–3: 1–5 – кривые СКО исследуемого алгоритма для ПОФ 90°, 45°, 20°, 10°, 5° соответственно, 6–10 теоретические границы в соответствии с (4), 11 – кривая ГРК в отсутствии фазового шума.

На рис. 4, 5: 1–4 – кривые СКО исследуемого алгоритма для ПОФ 20°, 10°, 5°, 1° соответственно, 5–8 теоретические границы в соответствии с (4), 9 – кривая ГРК в отсутствии фазового шума.

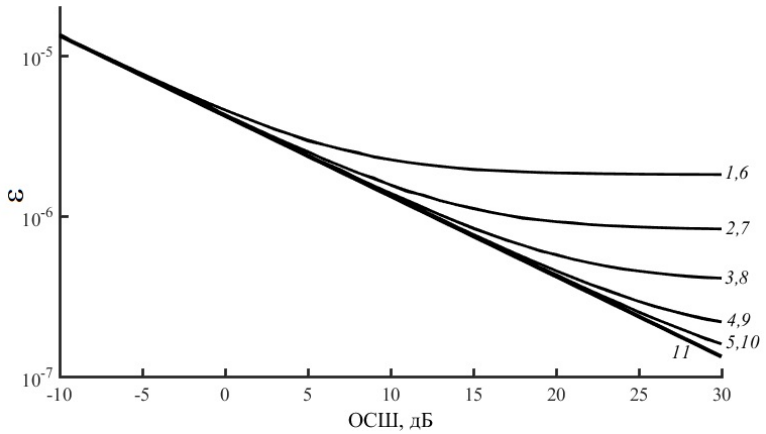


Fig. 1. The dependence of the standard error on the OSH and POF. Frequency noise of random walks

Рис. 1. Зависимости СКО от ОСШ и ПОФ. Частотный шум случайных блужданий

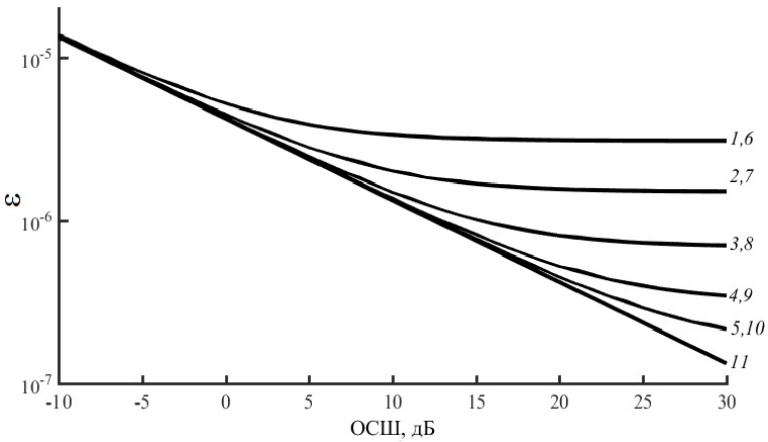


Fig. 2. The dependence of the standard error on the OSH and POF. Frequency flicker-noise

Рис. 2. Зависимости СКО от ОСШ и ПОФ. Частотный фликкер-шум

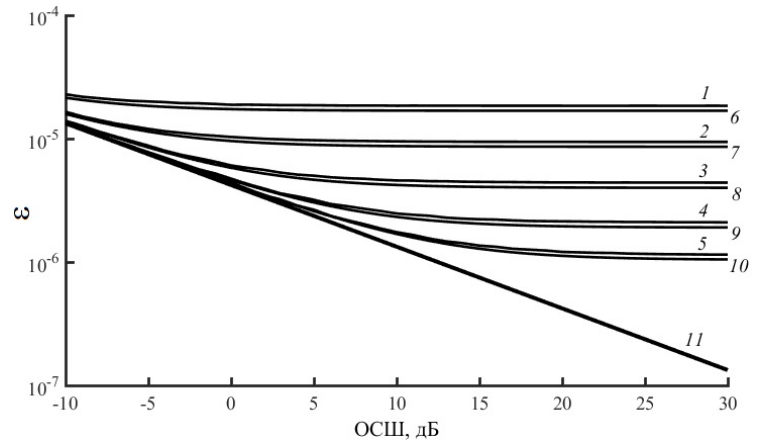


Fig. 3. The dependence of the standard error on the OSH and POF. White frequency noise

Рис. 3. Зависимости СКО от ОСШ и ПОФ. Белый частотный шум

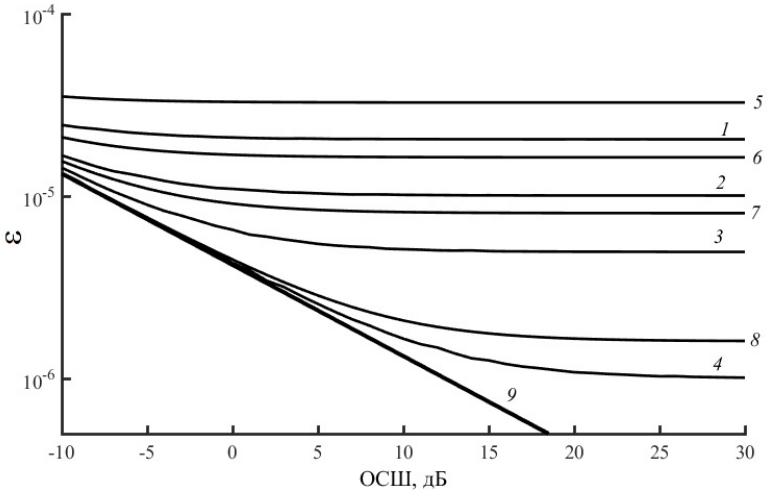


Fig. 4. The dependence of the standard error on the OSH and POF. Phase flicker-noise

Рис. 4. Зависимости СКО от ОСШ и ПОФ. Фазовый фликкер-шум

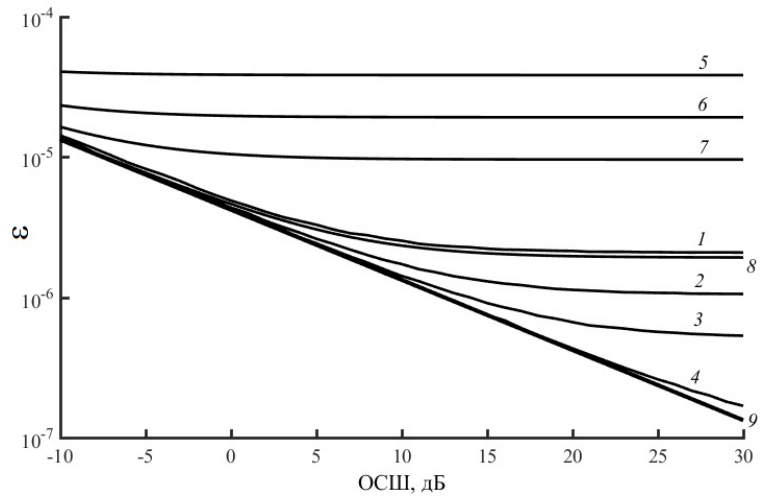


Fig. 5. The dependence of the standard error on the OSH and POF. White phase noise

Рис. 5. Зависимости СКО от ОСШ и ПОФ. Белый фазовый шум

По рис. 4–5 можно сделать следующие выводы:

- в канале с частотными шумами СКО алгоритма близка к значениям по (4);

- в канале с фазовыми шумами оценка (4) является завышенной к СКО алгоритма. Это связано с тем, что (4) не является теоретической предельной границей. Поскольку ГРК для канала с белым фазовым шумом получена аналитически (выражения (5), (6)) целесообразно провести сравнение алгоритма с этими выражениями.

Результаты сравнения СКО алгоритма с выражениями (5) и (6) представлены на рис. 6, 7.

На рис. 6: 1–5 – кривые СКО исследуемого алгоритма для ПОФ 15°, 10°, 5°, 2°, 1° соответственно, 6–10 теоретические границы в соответствии с (5), 11 – кривая ГРК в отсутствии фазового шума.

На рис. 7: 1–5 – кривые СКО исследуемого алгоритма для ПОФ 15°, 10°, 5°, 2°, 1° соответственно, 6–10 теоретические границы в соответствии с (6), 11 – кривая ГРК в отсутствии фазового шума.

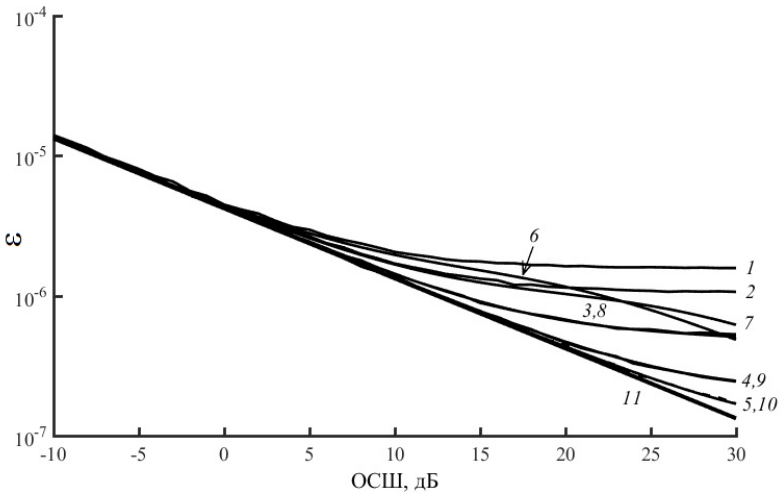


Fig. 6. The dependence of the standard error on the OSH and POF. White phase noise

Рис. 6. Зависимости СКО от ОСШ и ПОФ. Белый фазовый шум

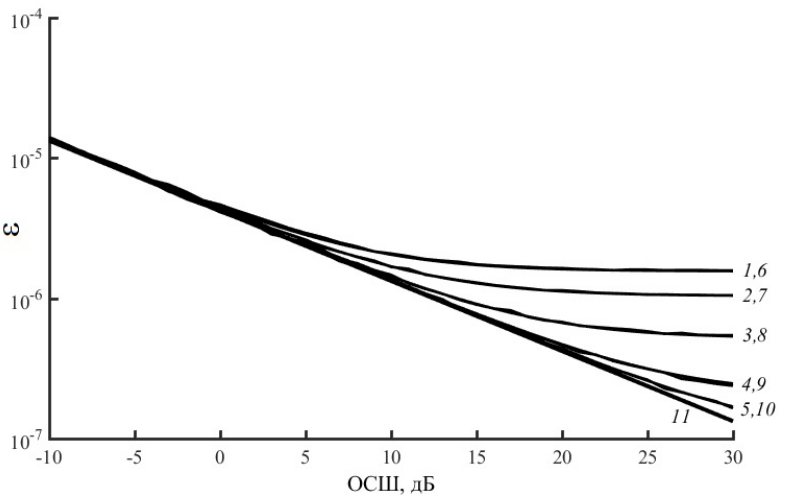


Fig. 7. The dependence of the standard error on the OSH and POF. White phase noise

Рис. 7. Зависимости СКО от ОСШ и ПОФ. Белый фазовый шум

По представленным на рис. 6, 7 кривым видно, что при малых значениях ПОФ СКО алгоритма очень близка к рассмотренным приближениям ГРК. Однако, как было отмечено выше, при высоких уровнях ПОФ граница, полученная Свинглером становится заниженной. Выражение (6) дает более точное приближение.

Выводы

1. Представлены выражения, приближенно описывающие СКО алгоритма оценки частоты, основанного на итерационном вычислении АКП, в условиях частотных шумов и АБГШ. На основании этих выражений алгоритм может использоваться для анализа долговременной частотной нестабильности коротких импульсно-модулированных сигналов.
2. Предложена формула приближенного вычисления ГРК в условиях белого фазового шума. Результаты моделирования показали, что предложенная формула точнее описывает СКО алгоритма при высоких значениях ПОФ, чем известная формула Свинглера.
3. В канале с белым фазовым шумом и АБГШ алгоритм, основанный на итерационном вычислении АКП обеспечивает оценку с СКО близкой к границе Рао-Крамера.

References

[1] A. A. Nikiforov, “Identification and assessment of information parameters of navigation systems with code division”, in *Thesis for the degree of candidate of technical sciences*, In Russian, Bauman Moscow State Technical University, 2014.

[2] V. G. Volkov, Y. N. Krivov, and I. V. Lukyanov, “Improved frequency estimation algorithm based on iterative computation of autocorrelation sequence”, *Journal of Radio Electronics*, no. 10, 2016, In Russian. [Online]. Available: http://jre.cplire.ru/jre/oct16/6/text.html.

[3] A. V. Ryzhkov and V. N. Popov, *Frequency synthesizers in radio communication technique*. Moscow: Radio and Communication, 1991, p. 264, In Russian.

[4] W. J. Riley, *Handbook of Frequency Stability Analysis*. USA — Washington: National Institute of Standards and Technology, 2008.

[5] D. N. Swingler, “Approximations to the Cramer-Rao lower bound on frequency estimates for complex sinusoids in the presence of sampling jitter”, *Signal Processing*, vol. 48, no. 1, pp. 77–83, 1996.

[6] S. M. Kay, *Fundamentals of Statistical Signal Processing: Estimation Theory (v.1)*. NJ: Prentice Hall, Upper Sadle River, 1998, p. 595.

[7] J. A. Barnes *et al.*, “Characterization of frequency stability”, in *IEEE Transactions on Instrumentation and Measurement*, vol. IM-20, 1971, pp. 105–120.

[8] A. Tretter, “Estimating the Frequency of a Noisy Sinusoid by Linear Regression”, *IEEE Transactions on Information theory*, vol. 31, no. 6, pp. 832–835, 1985.

[9] S. M. Kay, “A Fast and Accurate Single Frequency Estimator”, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 37, no. 12, pp. 1987–1990, 1989.

[10] Y. V. Kryukov, E. V. Rogozhnikov, and D. A. Pokamestov, “Phase noise model taking into account the spectral mask of frequency synthesizers and signal generators”, *Bulletin of the Tomsk Polytechnic University. Information Technology*, vol. 325, no. 5, pp. 45–51, 2014, In Russian.

[11] J. A. Barnes, “Atomic Timekeeping and the Statistics of Precision Signal Generators”, *Proceeding of the IEEE*, vol. 54, no. 2, pp. 207–220, 1966.