ISSN 1818-1015
eISSN 2313-5417

2020 Volume 27 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

2020 Том 27 № 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». Корректор – М. С. Каряева, корректор английских текстов – М. С. Комар. Подписано в печать 17.12.2020. Дата выхода в свет 30.12.2020. Формат 200×265 мм. Объем 146 с. Тираж 46 экз. Свободная цена. Заказ 083/020. 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

03–04 ноября 2020 г. в Москве прошел одиннадцатый международный научно–исследовательский семинар **«Семантика, спецификация и верификация программ: теория и приложения»** (11-th Workshop on Program Semantics, Specification and Verification: Theory and Applications, PSSV 2020). Заседания семинара проводились в формате видеоконференции. Организатором семинара выступила лаборатория процессно-ориентированных информационных систем (руководитель: проф. И.А. Ломазова) факультета компьютерных наук Национального исследовательского университета «Высшая школа экономики». Программа семинара PSSV 2020 включала:

- 5 регулярных докладов
  Natalia Garanina, Igor Anureev, Vladimir Zyubin, Andrei Rozov, Tanya Luach, Sergei Gorlatch: *Reasoning about Programmable Logic Controllers*,
  Aleksandr Tvardovskii, Nina Yevtushenko: *Deriving homing sequences for Finite State Machines with timed guards*,
  Evgenii Vinarskii, Vladimir Zakharov: *Studying the properties of timed finite state machines*,
  Anton Gnatenko, Vladimir Zakharov: *Using an extension of CTL* for specification and verification of sequential reactive systems*,
  Nikolay Shilov: *BDI-agent in a Cloud of Services*,

- 3 кратких сообщения
  Vladimir Zyubin, Igor Anureev, Natalia Garanina, Sergey Staroletov, Andrei Rozov, Tatiana Liakh: *EDTL4CSRS: Event-Driven Temporal Logic for Control Software Requirements Specification*,
  Vladimir Kukharenko, Kirill Ziborov, Alexandr Naumchev, Rafael Sadykov, Ruslan Rezin: *Verification of HotStuff protocol with TLA+/TLC for industry*,
  Catarina Gamboa, Paulo Santos, Alcides Fonseca: *Extending Java with Refinements*,

- 5 лекций приглашенных докладчиков
  Ilya Sergey (Yale-NUS College and National University of Singapore): Deductive Synthesis of Heap-Manipulating Programs: *Sound, Expressive, Fast*,
  Samvel K. Shoukourian (IT Educational and Research Center, Yerevan State University, Armenia): *Polynomial algorithm for equivalence problem of deterministic multitape finite automata*,
  Natasha Alechina (Utrecht University, The Netherlands): *State of the Art in Logics for Verification of Resource-Bounded Multi-Agent System*,
  Leonid Merkin: *Leading Research Center for Blockchain Technology of Innopolis University*,
  Ekaterina Komendantskaya (Heriot-Watt University Edinburgh, UK): *Refinement types for Verification of Neural Networks*,

а также мемориальную сессию, посвященную памяти выдающегося советского и украинского математика Александра Адольфовича Летичевского (1935–2019). В завершение семинара был организован круглый стол для обмена мнениями о характерных особенностях современных языков программирования и перспективах их дальнейшей эволюции. Своими взглядами на эти вопросы

с участниками семинара поделились: Андрей Бреслав (JetBrains), Алексей Недоря (Huawei), Антон Полухин (Yandex), Алексей Незванов (НИУ ВШЭ), Евгений Зуев, Алексей Канатов, Николай Кудасов (Университет Иннополис) и Владимир Ицыксон (СПбГУ). Финансовую поддержку организаторам конференции оказала всемирно известная компания Jet Brains.

В выступлениях участников семинара были представлены результаты завершенных и продолжающихся исследований, посвященных развитию и применению математически обоснованных методов для разработки программного обеспечения, вычислительной и телекоммуникационной аппаратуры. Значительное внимание было уделено методам дедуктивной проверки правильности программ, верификации моделей программ, применения методов теории автоматов к тестированию программ, а также ряду вопросов построения и анализа распределенных алгоритмов и формальных моделей информационных систем. Данный выпуск журнала включает 7 статей участников семинара PSSV 2020.

Н. В. Евтушенко и А. С. Твардовский посвятили свою статью решению одной из задач идентификации состояний конечных автоматов, а именно, задачи построения безусловных и адаптивных установочных последовательностей для автоматов с временными ограничениями. Предложенное авторами решение опирается на метод конечно-автоматной абстракции временного автомата, то есть описания временного автомата соответствующим конечным автоматом, который сохраняет некоторые свойства временного автомата, необходимые для анализа его поведения.

Временные конечные автоматы также исследуются в статье Е. М. Винарского и В. А. Захарова. В ней предложены две операционные семантики для новой модели конечных временных автоматов, применяемой для описания вычислений последовательных реагирующих систем (контроллеров, сетевых коммутаторов, системных драйверов), работающих в реальном времени. Показано, что эти семантики хорошо взаимосвязаны друг с другом и открывают возможность применения ранее известных методов верификации систем вычислений реального времени для анализа поведения последовательных реагирующих систем.

В статье Н. О. Гараниной, И. С. Ануреева, В. Е. Зюбина и других членов научно-исследовательской группы из Новосибирска рассматривается задача верификации программы для программируемых логических контроллеров (ПЛК). В качестве формальной модели программы ПЛК авторы используют систему переходов гиперпроцессов и предлагают вниманию читателей новую разновидность темпоральной логики, предназначенную для спецификации вычислений ПЛК с учетом особенностей поведения этой разновидности систем обработки информации.

Изучение новых разновидностей темпоральных логик также проводится в статье А. Р. Гнатенко и В. А. Захарова. В ней рассматривается задача верификации последовательных реагирующих систем, требования правильного поведения которых описываются при помощи нового расширения обобщенной темпоральной логики деревьев вычислений. Авторы описывают и исследуют алгоритм проверки выполнимости формул этой логики на моделях конечных автоматов-преобразователей.

Новые распределенные алгоритмы для мультиагентных систем предложены в статье Н. О. Гараниной и Н. В. Шилова. Один из алгоритмов предназначен для обнаружения неисправностей коммутаторов в телекоммуникационных системах, а другой – для разрешения конфликтов, возникающих при выделении ресурсов в облачных сервисах.

В двух статьях этого номера рассказывается о целях, задачах и результатах выполнения масштабного проекта создания российской верифицированной системы распределенного реестра (СРР), отвечающей требованиям безопасности. В статье В. А. Кухаренко, К. В. Зиброва, Р. Ф. Садыкова, А. В. Наумчева, Р. М. Резина и Л. А. Меркина описан реализующийся в настоящее время индустриальный проект между крупным российским авиаперевозчиком и несколькими ведущими университетами, результатом которого должна быть верифицированная программная СРР для более гибкой заправки воздушных судов (ВС). В качестве предварительного результата авторы статьи рассказывают об

опыте формальной спецификации и верификации протокола HotStuff – отказоустойчивого протокола для гарантированного достижения консенсуса в присутствии византийских процессов и лидера. В статье Л. А. Меркина, Р. М. Резина и Н. Васильева описана архитектура CPP InnoChain, основной целью которой является реализуемость 5-ти уровней формальной верификации программного обеспечения системы InnjChain, включая операционное окружение. Предлагаемая архитектура открывает возможности для использования CPP InnoChain в критических по надежности приложениях, в частности, в системе управления заправкой воздушных судов.

В последней статье номера А. Басин, М. В. Буздалов и А. А. Шалыто исследуют вопросы повышения эффективности вычислений эволюционных эвристических алгоритмов решения оптимизационных задач. Авторы статьи предлагают новый метод настройки параметров одного класса генетических алгоритмов. Действенность предложенного метода подтверждается результатами экспериментальных исследований, представленными в этой работе, а также хорошо согласуется с теоретическими оценками времени работы алгоритмов, использующих эти методы настройки параметров.

MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 27, NO. 4, 2020
journal homepage: www.mais-journal.ru

THEORY OF COMPUTING

Deriving Homing Sequences for Finite State Machines with Timed Guards

A. S. Tvardovskii[1], N. V. Yevtushenko[2,3]

DOI: 10.18255/1818-1015-2020-4-376-395

[1]National Research Tomsk State University, 36 Lenin Ave, Tomsk 634050, Russia.
[2]Ivannikov Institute for System Programming of the RAS, 25 Alexander Solzhenitsyn St., Moscow 109004, Russia.
[3]National Research University Higher School of Economics, 20 Myasnitskaya St., Moscow 101000, Russia.

MSC2020: 68Q45
Research article
Full text in Russian

Received November 9, 2020
After revision November 30, 2020
Accepted December 16, 2020

State identification is the well-known problem in the theory of Finite State Machines (FSM) where homing sequences (HS) are used for the identification of a current FSM state, and this fact is widely used in the area of software and hardware testing and verification. For various kinds of FSMs, such as partial, complete, deterministic, non-deterministic, there exist sufficient and necessary conditions for the existence of preset and adaptive HS and algorithms for their derivation. Nowadays timed aspects become very important for hardware and software systems and for this reason classical FSMs are extended by clock variables. In this work, we address the problem of checking the existence and derivation of homing sequences for FSMs with timed guards and show that the length estimation for timed homing sequence coincides with that for untimed FSM. The investigation is based on the FSM abstraction of a Timed FSM, i.e. on a classical FSM which describes behavior of corresponding TFSM and inherits some of its properties. When solving state identification problems for timed FSMs, the existing FSM abstraction is properly optimized.

Keywords: Finite State Machine; timed guards; FSM abstraction; homing sequence

INFORMATION ABOUT THE AUTHORS

Aleksandr Sergeevich Tvardovskii correspondence author	orcid.org/0000-0001-7705-7214. E-mail: tvardal@mail.ru PhD.
Nina Vladimirovna Yevtushenko	orcid.org/0000-0002-4006-1161. E-mail: evtushenko@ispras.ru Doctor of technical sciences, professor.

Funding: The reported study was funded by RFBR, project number 19-07-00327.

For citation: A. S. Tvardovskii and N. V. Yevtushenko, "Deriving Homing Sequences for Finite State Machines with Timed Guards", *Modeling and analysis of information systems*, vol. 27, no. 4, pp. 376-395, 2020.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 27, № 4, 2020
сайт журнала: www.mais-journal.ru

THEORY OF COMPUTING

Синтез установочных последовательностей для автоматов с временными ограничениями

А. С. Твардовский[1], Н. В. Евтушенко[2,3]

DOI: 10.18255/1818-1015-2020-4-376-395

[1]Национальный исследовательский Томский Государственный университет, пр. Ленина, д. 36, г. Томск, 634050 Россия.
[2]Институт системного программирования РАН, ул. А. Солженицына, д. 25, г. Москва, 109004 Россия.
[3]Национальный исследовательский университет «Высшая школа экономики», ул. Мясницкая, д. 20, г. Москва, 101000 Россия.

УДК 519.7
Научная статья
Полный текст на русском языке

Получена 9 ноября 2020 г.
После доработки 30 ноября 2020 г.
Принята к публикации 16 декабря 2020 г.

Идентификация состояний является хорошо известной задачей теории конечных автоматов, и установочные последовательности, которые позволяют идентифицировать текущее состояние конечного автомата, широко используются в областях тестирования и верификации программного и аппаратного обеспечения. Для автоматов различных классов, полностью определенных и частичных, детерминированных и недетерминированных, установлены необходимые и достаточные условия существования безусловных и адаптивных установочных последовательностей и предложены алгоритмы их синтеза, если такая последовательность существует. В настоящее время при верификации и тестировании программного и аппаратного обеспечения необходимо учитывать временные аспекты, что приводит к расширению автоматных моделей временными переменными. В настоящей работе мы исследуем задачи проверки существования и синтеза безусловных и адаптивных установочных последовательностей для автоматов с временными ограничениями и показываем, что оценки на длину таких последовательностей совпадают с оценками для классических конечных автоматов. Предлагаемый подход основан на использовании конечно-автоматной абстракции временного автомата, то есть описании временного автомата соответствующим конечным автоматом, который сохраняет свойства временного автомата относительно установочных последовательностей.

Ключевые слова: конечный автомат; временные ограничения; конечно-автоматная абстракция; установочная последовательность

ИНФОРМАЦИЯ ОБ АВТОРАХ

Александр Сергеевич Твардовский
автор для корреспонденции
orcid.org/0000-0001-7705-7214. E-mail: tvardal@mail.ru
кандидат физико-математических наук.

Нина Владимировна Евтушенко
orcid.org/0000-0002-4006-1161. E-mail: evtushenko@ispras.ru
доктор технических наук, профессор.

Финансирование: Исследование выполнено при финансовой поддержке РФФИ в рамках научного проекта № 19-07-00327.

Для цитирования: A. S. Tvardovskii and N. V. Yevtushenko, “Deriving Homing Sequences for Finite State Machines with Timed Guards”, *Modeling and analysis of information systems*, vol. 27, no. 4, pp. 376-395, 2020.

Введение

Проблемы идентификации состояний в автоматных моделях исследуются с середины 20-го века [1—5], в первую очередь, для классических конечных автоматов [1, 2]. Установочные, различающие и синхронизирующие последовательности, позволяющие идентифицировать состояния в конечно-автоматных моделях, широко используются в областях тестирования и верификации программного и аппаратного обеспечения [2, 4, 6—10]. Установочные и синхронизирующие последовательности позволяют определить состояние, достигнутое автоматом после подачи такой последовательности [3, 11—14]. Данный факт используется как для установки исследуемой системы в известное состояние при активном тестировании, так и для определения текущего состояния системы в ходе мониторинга при пассивном тестировании [8, 15]. В обоих случаях знание достигнутого автоматом состояния позволяет более эффективно провести дальнейший анализ исследуемой системы.

Задача синтеза установочных последовательностей формулируется для конечных автоматов, в которых вывод о достигнутом состоянии может быть сделан на основе соответствующей выходной реакции автомата после подачи установочной последовательности [8, 14, 16]. Понятие синхронизирующей последовательности формулируется для конечных полуавтоматов и отражает последовательность действий, которая переводит полуавтомат в известное состояние независимо от начального состояния полуавтомата [4, 14]. Задача синтеза установочных последовательностей хорошо изучена для классических конечных автоматов, однако недостаточно исследована для различных расширений таких автоматов.

Для различных классов классических конечных автоматов определены необходимые и достаточные условия существования безусловных [8, 11, 12, 14] и условных (адаптивных) [17—19] установочных последовательностей и предложены алгоритмы их синтеза, если такие последовательности существуют. Безусловная установочная последовательность представляет собой известную до начала эксперимента последовательность входных воздействий, по реакции на которую можно определить достигнутое автоматом состояние. Для адаптивной последовательности следующее входное воздействие зависит от реакции автомата на предыдущие входные воздействия, что усложняет проведение эксперимента, но позволяет сократить длину установочной последовательности в ряде случаев [17]. Также известно, что для недетерминированных конечных автоматов вероятность существования адаптивной установочной последовательности значительно выше, чем вероятность существования безусловной установочной последовательности.

Функционирование современного программного и аппаратного обеспечения часто зависит от временных аспектов, что мотивирует исследования в области временных автоматов. Существуют различные подходы к внесению временных аспектов в модель конечного автомата, такие как входные таймауты, выходные таймауты (задержки на переходах), входные и выходные временные ограничения [20—24]. Все эти подходы объединяет внесение в модель автомата временной переменной, от значения которой зависит поведение автомата. В настоящей работе мы исследуем задачу проверки существования и синтеза безусловных и адаптивных установочных последовательностей для автоматов с одной временной переменной и (входными) временными ограничениями [21, 23]. А именно, мы адаптируем классические конечно-автоматные алгоритмы для построения установочных последовательностей для полностью определенных временных автоматов.

Предлагаемый в работе подход основан на использовании конечно-автоматной абстракции [23], т.е. конечного автомата, в ряде случаев адекватно описывающего поведение временного автомата. Как показано в работе, такая абстракция позволяет проверить существование временной установочной последовательности с использованием классических алгоритмов и оценить длину кратчайшей установочной последовательности, если таковая существует. Известный подход к построению конечно-автоматной абстракции сопровождается существенным увеличением входного алфави-

та или множества состояний относительно исходного временного автомата [23]. Поскольку такая абстракция является избыточной при решении ряда задач [21, 25, 26], в том числе задач, связанных с установочными и тестовыми последовательностями, мы предлагаем подход к оптимизации конечно-автоматной абстракции.

Результаты данной работы были частично представлены на международном семинаре PSSV-2020 (XI Workshop Program Semantics, Specification and Verification: Theory and Applications) и опубликованы в [26]. В настоящей статье мы расширяем полученные результаты для адаптивных установочных последовательностей, а также формулируем более общий алгоритм оптимизации конечно-автоматной абстракции, используемой для проверки существования и построения установочных последовательностей.

Структура работы следующая. Раздел 1 содержит определения классического и временного автомата, а также описание существующего подхода к синтезу конечно-автоматной абстракции. В разделе 2 исследуется проблема существования и синтеза безусловных и адаптивных установочных последовательностей для автоматов с временными ограничениями. Раздел 3 описывает предлагаемый в работе подход к оптимизации конечно-автоматной абстракции при синтезе установочных последовательностей.

1. Основные определения и обозначения

1.1. Конечный автомат

Под конечным автоматом [1] понимается четвёрка $\mathsf{P} = (P, I, O, h_{\mathsf{P}})$, где P — конечное непустое множество состояний, I — конечное непустое множество входных символов (входной алфавит), O — конечное непустое множество выходных символов (выходной алфавит), $I \cap O = \emptyset$, $h_{\mathsf{P}} \subseteq (P \times I \times O \times P)$ — отношение переходов. Кортеж (p, i, o, p') описывает переход из состояния p в состояние p' под действием входного символа i с выдачей выходного символа o. В соответствии с введённым определением, мы рассматриваем так называемые неинициальные автоматы, в которых каждое состояние может быть начальным, если явно не указано обратное.

Если в автомате P для любой пары $(p, i) \in P \times I$ существует переход $(p, i, o, p') \in h_{\mathsf{P}}$, то автомат называется полностью определённым, иначе – частичным. Если в автомате P для любой пары $(p, i) \in P \times I$ существует не более одного перехода $(p, i, o, p') \in h_{\mathsf{P}}$, то автомат называется детерминированным, иначе – недетерминированным. Полностью определённый недетерминированный автомат называется наблюдаемым, если для каждой тройки $(p, i, o) \in P \times I \times O$ существует не более одного перехода $(p, i, o, p') \in h_{\mathsf{P}}$, т.е. следующее состояние автомата для пары (входной символ / выходной символ) определяется единственным образом.

В настоящей работе мы рассматриваем полностью определённые наблюдаемые, возможно, недетерминированные конечные автоматы. Исключением являются тестовые примеры, которые используются для представления адаптивных входных последовательностей. Тестовые примеры, которые представляют собой инициальные наблюдаемые автоматы специального вида, являются частичными автоматами при $|I| > 1$ и подробно рассматриваются в разделе 2.2.

Для автомата P допустимая последовательность пар (входной символ / выходной символ) $i_1/o_1, i_2/o_2, \ldots, i_n/o_n$ в состоянии p_1 называется вход-выходной последовательностью в состоянии p_1, если в P существует соответствующая последовательность переходов $(p_1, i_1, o_1, p_2), (p_2, i_2, o_2, p_3), \ldots, (p_n, i_n, o_n, p_{n+1})$. При этом $\alpha = i_1, i_2, \ldots, i_n$ — входная последовательность, а $\gamma = o_1, o_2, \ldots, o_n$ — соответствующая выходная последовательность (реакция автомата). Вход-выходная последовательность в этом случае может быть обозначена как α/γ, и в автомате существует переход $(p_1, \alpha, \gamma, p_{n+1})$.

Для состояния p и вход-выходной пары i/o вводится понятие i/o-преемника состояния p: i/o-преемник состояния p в автомате P содержит каждое состояние p' такое, что в автомате существует переход (p, i, o, p'). Для вход-выходной последовательности α/γ, α/γ-преемником состояния p автомата P называется множество всех состояний p', в которые автомат переходит из состояния p по вход-выходной последовательности α/γ. Заметим, что α/γ-преемник состояния p может быть пустым, если в автомате не существует ни одного перехода вида (p, α, γ, p'), и в этом случае мы иногда говорим, что α/γ-преемник состояния p не существует. В наблюдаемом полностью определенном автомате α/γ-преемник состояния p является одноэлементным или не существует. Для подмножества P' состояний автомата, α/γ-преемник определяется как объединение α/γ-преемников состояний из P'. Автомат $\mathsf{Q} = (Q, I_Q, O_Q, h_\mathsf{Q})$ является подавтоматом автомата $\mathsf{P} = (P, I, O, h_\mathsf{P})$, если $Q \subseteq P, I_Q \subseteq I, O_Q \subseteq O$ и $h_\mathsf{Q} \subseteq h_\mathsf{P}$. В данной работе мы рассматриваем частный случай подавтоматов, в которых $Q = P$.

1.2. Автомат с временными ограничениями

В настоящей работе мы рассматриваем автомат с временными ограничениями, который представляет собой конечный автомат, дополненный одной временной переменной и временными интервалами на переходах [21, 23, 24].

Под автоматом с временными ограничениями понимается четвёрка $\mathsf{S} = (S, I, O, \lambda_\mathsf{S})$, где S — конечное непустое множество состояний, I — входной алфавит, O — выходной алфавит, $\lambda_\mathsf{S} \subseteq (S \times I \times O \times S \times \Pi)$ – конечное отношение переходов, Π – множество интервалов из промежутка $[0; \infty)$ вида $\langle a, b \rangle$, где $\langle \in \{[, (\}, \rangle \in \{],)\}$, a и b — целые неотрицательные числа или ∞. Соответственно, кортеж (s, i, o, s', g) описывает переход из состояния s в состояние s' под действием входного символа i, поступившего в момент времени $t, t \in g$, после перехода автомата в текущее состояние, с выдачей выходного символа o. Временная переменная увеличивает своё значение при ожидании входного символа и сбрасывается в 0 при выполнении перехода автоматом.

Временной автомат S называется полностью определённым, если поведение автомата в любом состоянии определено для любого входного символа, и для любой пары $(s, i) \in S \times I$, то есть для любого входного символа i, поступающего на вход автомата в состоянии s, объединение всех временных интервалов g из переходов $(s, i, o, s', g) \in \lambda_\mathsf{S}$ равно $[0; \infty)$. Аналогично классическим автоматам, частичный автомат может быть доопределён до полностью определённого автомата различными способами в зависимости от интерпретации неопределенных переходов [1, 27]. Если для любых двух кортежей $(s, i, o_1, s_1, g_1), (s, i, o_2, s_2, g_2) \in \lambda_\mathsf{S}$ справедливо, что $g_1 \bigcap g_2 = \emptyset$, то временной автомат называется детерминированным, иначе – недетерминированным. Полностью определённый недетерминированный временной автомат называется наблюдаемым, если для каждой пары переходов $(s, i, o, s_1, g_1), (s, i, o, s_2, g_2) \in \lambda_\mathsf{S}$, такой что $g_1 \bigcap g_2 \neq \emptyset$, справедливо, что $s_1 = s_2$, т.е. следующее состояние определяется единственным образом по соответствующей выходной реакции.

Временным входным символом называется пара (i, t), где i – входной символ, t – текущее значение временной переменной, т.е. время поступления входного символа, отсчитываемое от момента выдачи автоматом последнего выходного символа или начала эксперимента.

Для временного автомата S последовательность временных входных символов $\alpha = (i_1, t_1), (i_2, t_2), \ldots, (i_n, t_n)$ называется временной входной последовательностью. Реакция на временную входную последовательность α автомата S в состоянии s_1 вычисляется итеративно и представляет собой последовательность выходных символов γ. Для входного символа (i_1, t_1) автоматом выполняется переход $(s_1, i_1, o_1, s_2, g_1)$, для которого $t_1 \in g_1$; после выполнения перехода временная переменная сбрасывается в 0. Процедура повторяется в достигнутом состоянии s_2 со следующим временным входным символом (i_2, t_2) и так далее. Аналогично классическим автоматам, последовательность пар (временной входной символ / выходной символ) называется временной вход-выходной последовательностью автомата S в состоянии s_1, если в S существует соответствующая последовательность

переходов $(s_1, i_1, o_1, s_2, g_1), (s_2, i_2, o_2, s_3, g_2), \ldots, (s_n, i_n, o_n, s_{n+1}, g_n)$, где $t_i \in g_i$, и такая последовательность обозначается $\alpha/\gamma = (i_1, t_1)/o_1 \ldots (i_n, t_n)/o_n$.

Например, если на временной автомат S (рисунок 1) в состоянии s_1 подается входной символ i_1 при значении временной переменной 1.7, т.е. на автомат поступает временной входной символ $(i_1, 1.7)$, то автомат перейдёт в состояние s_3 с выходным символом o_2, после чего временная переменная «сбросится» в 0.

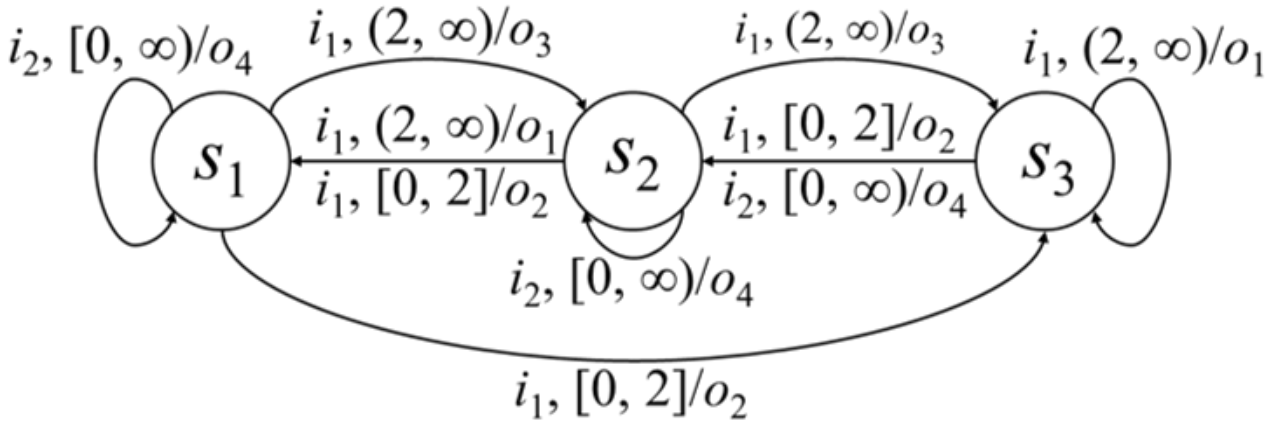


Fig. 1. FSM with timed guards S

Рис. 1. Автомат с временными ограничениями S

Понятие α/γ-преемника для временных автоматов определяется аналогично классическим автоматам. По определению, $(i, t)/o$-преемник состояния s содержит каждое состояние s' такое, что в автомате существует переход (s, i, o, s', g), где $t \in g$. Для временной вход-выходной последовательности α/γ, α/γ-преемником состояния s временного автомата S называется множество всех состояний s', в которые автомат переходит из состояния s по последовательности α/γ. В наблюдаемом автомате α/γ-преемник состояния s является одноэлементным или не существует. Для подмножества S' состояний временного автомата α/γ-преемник определяется как объединение α/γ-преемников состояний из S'.

В настоящей работе мы рассматриваем задачу синтеза безусловных и адаптивных установочных последовательностей для полностью определённых наблюдаемых, возможно недетерминированных автоматов с временными ограничениями.

1.3. Конечно-автоматные абстракции временного автомата

Известно [23], что поведение временных автоматов в ряде случаев можно адекватно описать при помощи конечного автомата, т.е. с использованием некоторой конечно-автоматной абстракции. Конечно-автоматная абстракция полностью определенного автомата с временными ограничениями $\mathsf{S} = (S, I, O, \lambda_{\mathsf{S}})$ с наибольшей конечной границей для временных входных интервалов B_{S} определяется как полностью определённый конечный автомат $\mathsf{A}_{\mathsf{S}} = (S, I_A, O, h_{\mathsf{AS}})$, где $I_A = \{(i, [0, 0]), (i, (0, 1)), \ldots, (i, (B_{\mathsf{S}} - 1, B_{\mathsf{S}})), (i, [B_{\mathsf{S}}, B_{\mathsf{S}}]), (i, (B_{\mathsf{S}}, \infty)) : i \in I\}$. Для состояния s автомата A_{S} и входного символа (i, g_j), множество h_{AS} содержит переход $(s, (i, g_j), o, s')$, если и только если существует переход $(s, i, o, s', g) \in \lambda_{\mathsf{S}}$, такой что $g_j \subseteq g$. Таким образом, при построении абстракции к каждому входному символу добавляются открытые интервалы времени длины 1 и целочисленные значения временной переменной.

Для наблюдаемого недетерминированного автомата S на рисунке 1, соответствующая конечно-автоматная абстракция представлена на рисунке 2. Строки таблицы соответствуют входным символам, столбцы – состояниям, в ячейке на пересечении состояния s и входного символа i содержится множество пар (выходной символ o / io-преемник s'), таких что $(s, i, o, s') \in h_{\mathsf{AS}}$.

Временная входная последовательность α для автомата с временными ограничениями может быть переведена в соответствующую последовательность входных символов α_{FSM} для конечно-автоматной абстракции. При этом каждый временной входной символ (i, t) представляется входным символом (i, t), если t целое, либо символом $(i, (a, b))$, где $t \in (a, b)$, в противном случае. Аналогично, входная последовательность конечно-автоматной абстракции может быть переведена во временную входную последовательность. При этом, для входного символа $(i, (a, b))$ значением временной переменной соответствующего входного временного символа может быть выбрано любое значение $t \in (a, b)$.

i/s	s_1	s_2	s_3
$(i_1, [0, 0])$	s_3/o_2	s_1/o_2	s_2/o_2
$(i_1, (0, 1))$	s_3/o_2	s_1/o_2	s_2/o_2
$(i_1, [1, 1])$	s_3/o_2	s_1/o_2	s_2/o_2
$(i_1, (1, 2))$	s_3/o_2	s_1/o_2	s_2/o_2
$(i_1, [2, 2])$	s_3/o_2	s_1/o_2	s_2/o_2
$(i_1, (2, \infty))$	s_2/o_3	$s_1/o_1, s_3/o_3$	s_3/o_1

i/s	s_1	s_2	s_3
$(i_2, [0, 0])$	s_1/o_4	s_2/o_4	s_2/o_4
$(i_2, (0, 1))$	s_1/o_4	s_2/o_4	s_2/o_4
$(i_2, [1, 1])$	s_1/o_4	s_2/o_4	s_2/o_4
$(i_2, (1, 2))$	s_1/o_4	s_2/o_4	s_2/o_4
$(i_2, [2, 2])$	s_1/o_4	s_2/o_4	s_2/o_4
$(i_2, (2, \infty))$	s_1/o_4	s_2/o_4	s_2/o_4

Fig. 2. Flow table of the FSM abstraction of the TFSM in figure 1

Рис. 2. Конечно-автоматная абстракция временного автомата на рисунке 1

Отметим некоторые основные свойства конечно-автоматной абстракции [23].

- Конечно-автоматная абстракция полностью определённого временного автомата является полностью определённым автоматом.
- Конечно-автоматная абстракция детерминированного (недетерминированного) временного автомата является детерминированным (недетерминированным) автоматом.
- Конечно-автоматная абстракция наблюдаемого (ненаблюдаемого) временного автомата является наблюдаемым (ненаблюдаемым) автоматом.

Такая конечно-автоматная абстракция достаточно полно описывает поведение временного автомата и, в частности, может быть использована при синтезе тестов с гарантированной полнотой [25]. В [23], было доказано утверждение о соответствии между вход-выходными последовательностями автомата с временными ограничениями и его конечно-автоматной абстракции для инициальных автоматов, в которых выделено единственное начальное состояние. Аналогичное утверждение можно сформулировать и для неинициальных временных автоматов.

Утверждение 1. Временная вход-выходная последовательность α/γ существует в состоянии s неинициального временного автомата S, если и только если для конечно-автоматной абстракции $\mathsf{A_S}$ существует вход-выходная последовательность α_{FSM}/γ в состоянии s.

2. Установочные последовательности для временных автоматов

Установочная последовательность позволяет по реакции автомата установить достигнутое по такой последовательности состояние, независимо от начального состояния. Такие последовательности могут быть безусловными, если следующий входной символ не зависит от реакции автомата на предыдущие входные воздействия, или адаптивными, если следующий входной символ зависит от реакции автомата на предыдущие входные символы. Для детерминированных полностью определенных приведенных сильно связных автоматов всегда существует безусловная установочная последовательность, причем длина кратчайшей установочной последовательности для автомата с n состояниями не превосходит величины $n(n-1)/2$, и переход к адаптивным установочным последовательностям не позволяет сократить длину последовательности. В то же время, для недетерминированных автоматов безусловная установочная последовательность существует не всегда, и такая кратчайшая последовательность может иметь экспоненциальную длину относительно числа состояний автомата. Для недетерминированных автоматов адаптивные установочные последовательности существуют чаще, чем безусловные [28], и в ряде случаев имеют меньшую длину. В следующих разделах мы адаптируем известные конечно-автоматные методы построения установочных последовательностей к построению таких последовательностей для временных автоматов.

2.1. Безусловные установочные последовательности для временных автоматов

Временная входная последовательность α является установочной для полностью определенного временного автомата $\mathsf{S} = (S, I, O, \lambda_{\mathsf{S}})$, если для любой выходной реакции γ на α, α/γ-преемник множества S является одноэлементным или не существует, т.е. является пустым множеством. Для наблюдаемых полностью определённых конечных автоматов установочная последовательность может быть построена по усечённому дереву преемников [1, 12, 13].

При рассмотрении временного автомата необходимо учитывать, что автомат может не только находиться в произвольном состоянии, но и обладать любым значением временной переменной в начале эксперимента. Тем не менее, поскольку любое входное воздействие сбрасывает значение временной переменной в 0, то достаточно рассмотреть проблему синтеза установочной последовательности в предположении, что временная переменная равна 0 в момент начала установочного эксперимента.

Утверждение 1 и взаимно однозначное соответствие между состояниями временного автомата и его конечно-автоматной абстракцией позволяет сформулировать следующее утверждение.

Утверждение 2. Временная входная последовательность α является установочной последовательностью для полностью определённого автомата с временными ограничениями S, если и только если соответствующая входная последовательность α_{FSM} является установочной последовательностью для конечно-автоматной абстракции A_{S}.

Доказательство. Пусть α является установочной последовательностью для автомата с временными ограничениями $\mathsf{S} = (S, I, O, \lambda_{\mathsf{S}})$. По определению, α является установочной последовательностью для S, если и только если для каждой вход-выходной последовательности $\alpha/\gamma = (i_1, t_1)/o_1 \ldots (i_k, t_k)/o_k$ справедливо, что α/γ-преемник S является одноэлементным или не существует. По определению конечно-автоматной абстракции и в силу утверждения 1, в автомате S существуют вход-выходная последовательность α/γ и последовательность переходов $(s_1, i_1, o_1, s_2, g_1), \ldots, (s_k, i_k, o_k, s', g_k) \in \lambda_{\mathsf{S}}$ если и только если в конечно-автоматной абстракции A_{S} существуют вход-выходная последовательность α_{FSM}/γ и последовательность переходов $(s, (i_1, g'_1), o_1, s_1), \ldots, (s_k, (i_k, g'_k), o_k, s') \in h_{\mathsf{AS}}$, где $g'_j \subseteq g_j$ и $1 \le j \le k$. Соответственно, в конечно-автоматной абстракции A_{S} каждый α_{FSM}/γ-преемник множества S является одноэлементным множеством или не существует, если и только если α_{FSM} является установочной последовательностью для A_{S}.

Следовательно, установочная последовательность для автомата с временными ограничениями может быть построена как установочная последовательность для соответствующей конечно-автоматной абстракции. Далее входная последовательность абстракции преобразуется во временную входную последовательность.

Оценим длину установочных последовательностей для автоматов с временными ограничениями, которая определяется как количество временных входных символов в последовательности.

Состояния s и p полностью определённых детерминированных (временных) автоматов S и P называются эквивалентными, если реакции автоматов в этих состояниях совпадают на любую (временную) входную последовательность, в противном случае, состояния s и p различимы. Полностью определенный детерминированный автомат называется приведённым, если любые два различные его состояния не являются эквивалентными. Полностью определенный детерминированный автомат называется сильно связным, если для любой пары состояний s_1 и s_2 существует последовательность α, такая что α/γ-преемник состояния s_1 есть состояние s_2.

Известно [1], что для полностью определённых детерминированных приведённых сильно связных конечных автоматов установочная последовательность всегда существует и имеет длину не более $n(n-1)/2$, где n – число состояний автомата. Это свойство выполняется и для временного автомата.

Утверждение 3. Пусть S – полностью определённый детерминированный автомат с временными ограничениями.

1) Автомат S является сильно связным, если и только если конечно-автоматная абстракция $\mathsf{A_S}$ является сильно связным автоматом.
2) Автомат S является приведённым по состояниям, если и только если конечно-автоматная абстракция $\mathsf{A_S}$ является приведённым конечным автоматом.

Доказательство.

1) По определению конечно-автоматной абстракции $\mathsf{A_S}$, между множеством её состояний и множеством состояний соответствующего временного автомата S существует взаимно однозначное соответствие, более того, автомат $\mathsf{A_S}$ содержит переход $(s,(i,g_i),o,s') \in h_{\mathsf{AS}}$ если и только если $(s,i,o,s',g) \in \lambda_{\mathsf{S}}$, где $g_i \subseteq g$. Соответственно, временная входная последовательность α переводит автомат с временными ограничениями S из состояния s в состояние s', если и только если входная последовательность α_{FSM} переводит автомат $\mathsf{A_S}$ из состояния s в состояние s'.
2) Автомат с временными ограничениями S приведённый, если и только если для каждой пары различных состояний $s_1, s_2 \in S$ существует входная временная последовательность $\alpha(s_1,s_2)$, различающая эти состояния. Различающая временная входная последовательность $\alpha(s_1,s_2)$ существует для каждой пары состояний $s_1, s_2 \in S$, если и только если существует различающая последовательность $\alpha(s_1,s_2)_{FSM}$ для состояний s_1 и s_2 конечно-автоматной абстракции $\mathsf{A_S}$ (утверждение 1). Следовательно, различающая последовательность $\alpha(s_1,s_2)_{FSM}$ существует для каждой пары различных состояний конечно-автоматной абстракции $\mathsf{A_S}$, если и только если $\mathsf{A_S}$ является приведённым автоматом.

Поскольку между состояниями временного автомата и состояниями соответствующей конечно-автоматной абстракции существует взаимно однозначное соответствие, то на основе утверждений 2 и 3, можно сформулировать следующее утверждение о длине установочных последовательностей для временных автоматов.

Утверждение 4. Пусть S – полностью определённый детерминированный приведённый сильно связный автомат с временными ограничениями. Установочная последовательность всегда существует для автомата S и длина кратчайшей установочной последовательности не превышает $n(n-1)/2$, где n – количество состояний автомата.

Действительно, поскольку конечно-автоматная абстракция A_S полностью определённого детерминированного приведённого сильно связного автомата с временными ограничениями S является полностью определённым детерминированным приведённым и сильно связным автоматом (утверждения 1 и 3), то и A_S, и соответствующий временной автомат S обладают установочной последовательностью длины не больше $n(n-1)/2$.

Таким образом, установочная последовательность всегда существует для полностью определённого детерминированного приведённого сильно связного автомата с временными ограничениями. Более того, поскольку верхняя оценка не может быть уменьшена для конечных детерминированных автоматов [3], эту оценку нельзя уменьшить и для временных детерминированных автоматов с соответствующими свойствами.

Отметим, что для недетерминированных конечных автоматов длина кратчайшей безусловной установочной последовательности может превышать величину $2^{n-1}-1$ [13], и, таким образом, для полностью определённого недетерминированного наблюдаемого автомата с временными ограничениями, который имеет n состояний, длина кратчайшей безусловной установочной последовательности может также превышать величину $2^{n-1}-1$.

Аналогично результатам для классических конечных автоматов, установочная последовательность не всегда существует для недетерминированных наблюдаемых полностью определённых автоматов с временными ограничениями. Проверка существования и синтез установочных последовательностей для таких временных автоматов могут быть выполнены аналогично классическому алгоритму на основе построения усечённого дерева преемников [13].

Алгоритм 1 проверки существования и построения установочных последовательностей для автомата с временными ограничениями.

Вход: полностью определённый наблюдаемый возможно недетерминированный автомат с временными ограничениями $\mathsf{S} = (S, I, O, \lambda_\mathsf{S})$.

Выход: установочная последовательность α для временного автомата S или сообщение “Для автомата S не существует установочной последовательности”.

1. Строится конечно-автоматная абстракция $\mathsf{A}_\mathsf{S} = (S, I_A, O, h_{\mathsf{AS}})$ для временного автомата S.
2. Для автомата A_S строится усечённое дерево преемников. Каждая вершина дерева преемников помечается подмножеством пар различных состояний; рёбра дерева помечаются входными символами. Корню дерева ставится в соответствие множество всех пар различных состояний. Из вершины, помеченной множеством P уровня $j, j \geq 0$, существует ребро, помеченное входным символом i, в вершину уровня $(j+1)$, помеченную множеством Q, где Q содержит пару состояний s_1, s_2, если и только если состояния s_1 и s_2 являются i/o-преемниками некоторой пары состояний из P для некоторого выходного символа o. Если таких пар нет, то вершина уровня $(j+1)$ помечается пустым множеством. Вершина уровня $k, k > 0$, помеченная множеством P, является листом, если множество P пусто (Правило усечения 1) или P содержит непустое множество R, помечающее вершину уровня $j, j < k$ (Правило усечения 2).
3. Если построенное дерево преемников не имеет вершин, помеченных пустым множеством, то выдается сообщение “Для автомата S не существует установочной последовательности”. Иначе, кратчайшая последовательность входных символов α_{FSM}, помечающая путь из корня дерева в помеченную пустым множеством вершину, преобразуется в соответствующую временную входную последовательность, которая является кратчайшей установочной последовательностью для автомата S.

Корректность алгоритма 1 следует непосредственно из утверждения 2.

Для конечно-автоматной абстракции A_S (рисунок 2) временного автомата S (рисунок 1) соответствующее дерево преемников для подавтомата с входными символами, покрашенными серым цветом, представлено на рисунке 3. Непосредственной проверкой можно убедиться, что кратчайшей последовательностью входных символов, помечающей путь из корня дерева в вершину с пустым множеством, является последовательность $(i_1, (2, \infty)), (i_1, (2, \infty)), (i_1, (2, \infty))$ длины 3, которая может быть преобразована во временную установочную последовательность $(i_1, 2.1), (i_1, 2.3), (i_1, 2.2)$. Таким образом, эта последовательность является установочной последовательностью для временного автомата на рисунке 1. Обоснованность использования именно такого подавтомата при построении кратчайшей установочной последовательности обсуждается в разделе 3.

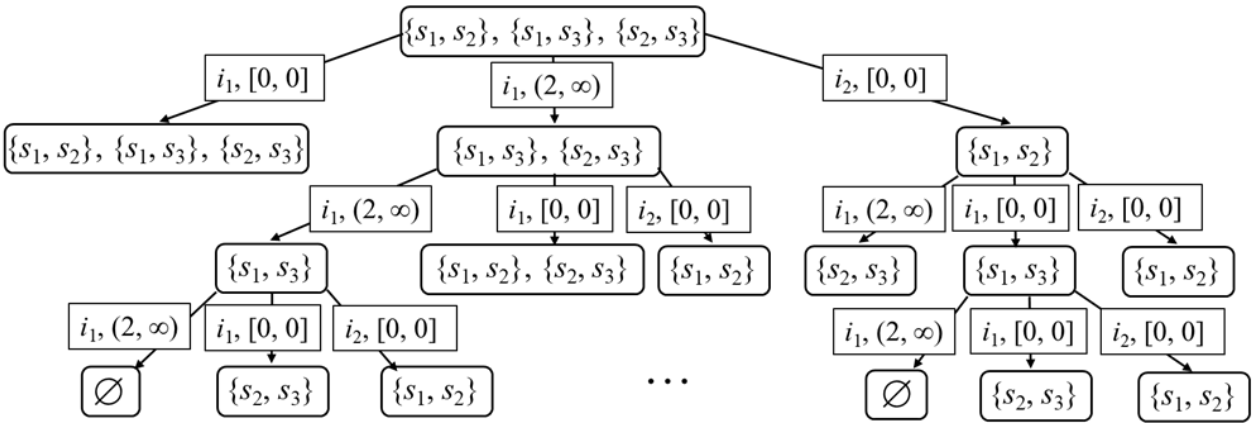


Fig. 3. The successor tree for a submachine of the FSM abstraction A_S in figure 2

Рис. 3. Дерево преемников для подавтомата конечно-автоматной абстракции A_S на рисунке 2

2.2. Адаптивные установочные последовательности

Безусловные установочные последовательности не всегда существуют для недетерминированных автоматов, и длина кратчайшей безусловной установочной последовательности (если таковая существует) может достигать экспоненциальной величины. Однако для неинициальных наблюдаемых полностью определенных автоматов, которые рассматриваются в настоящей статье, задача проверки существования и синтеза адаптивной установочной последовательности имеет полиномиальную сложность [13]. Для таких автоматов адаптивная установочная последовательность может существовать и в том случае, когда отсутствует безусловная установочная последовательность [28, 29]. Более того, в ряде случаев, кратчайшая адаптивная последовательность может оказаться короче, чем безусловная установочная последовательность. В настоящем разделе мы исследуем задачу синтеза адаптивных установочных последовательностей для временных автоматов.

2.2.1. Установочный тестовый пример

Для адаптивных последовательностей следующий входной символ может зависеть от реакции автомата на предыдущий входной символ и в данном разделе мы вводим понятие адаптивной установочной последовательности для временных автоматов. Адаптивная входная последовательность часто представляется тестовым примером [28, 29]. Пусть I и O — входной и выходной алфавиты некоторого конечного автомата P, тестовый пример $\mathsf{TC}(I, O)$ для автомата P представляет собой конечный автомат $\mathsf{TC}(I, O) = (T, I, O, h_{\mathsf{TC}}, t_0)$ с ациклическим графом переходов, в каждом состоянии которого либо определен переход только по одному входному символу со всеми выходными символами, либо не определён ни один переход; состояние без исходящих переходов называется

терминальным. Тестовый пример называется установочным [28] для полностью определённого возможно недетерминированного конечного автомата $\mathsf{P} = (P, I, O, h_{\mathsf{P}})$, если для любой вход-выходной последовательности α/γ тестового примера $\mathsf{TC}(I, O)$, переводящей автомат из начального состояния в терминальное, α/γ-преемник множества всех состояний P автомата P является одноэлементным или не существует.

Аналогично синтезу безусловных установочных последовательностей, понятие тестового примера для автомата с временными ограничениями может быть введено в предположении, что временная переменная имеет нулевое значение в начале эксперимента, поскольку может быть сброшена в 0 подачей произвольного входного символа. Под временным тестовым примером для автомата с временными ограничениями $\mathsf{S} = (S, I, O, \lambda_{\mathsf{S}})$ будем понимать тестовый пример $\mathsf{TTC}(I_A, O) = (T, I_A, O, h_{\mathsf{TTC}}, q_0)$ над входным алфавитом $I_A = \{I \times [a, a], (a, a+1), [B_{\mathsf{S}}, B_{\mathsf{S}}], (B_{\mathsf{S}}, \infty) : 0 \le a < B_{\mathsf{S}}\}$ и выходным алфавитом O. Временной тестовый пример $\mathsf{TTC}(I_A, O)$ представляет адаптивную входную последовательность для временного автомата $\mathsf{S} = (S, I, O, \lambda_{\mathsf{S}})$, которая может быть подана на автомат S следующим образом. Пусть входной символ (i_1, g) определён в начальном состоянии q_0 автомата $\mathsf{TTC}(I_A, O)$, тогда первым подаётся временной входной символ (i_1, t), где $t \in g$, и автомат $\mathsf{TTC}(I_A, O)$ переходит в $(i_1, g)/o$-преемник q_1 начального состояния q_0 в соответствии с выходным символом o, выданным автоматом S. Процедура продолжается до тех пор, пока тестовый пример не достигнет терминального состояния. Временной тестовый пример $\mathsf{TTC}(I_A, O)$ называется установочным, т.е. представляет адаптивную установочную последовательность для временного автомата S, если для каждой вход-выходной последовательности $\alpha_{FSM}/\gamma = (i_1, g_1)/o_1 \ldots (i_k, g_k)/o_k$ тестового примера $\mathsf{TTC}(I_A, O)$ из начального состояния в терминальное и любых $t_j \in g_j, j = 1 \ldots k$, $\alpha/\gamma = (i_1, t_1)/o_1 \ldots (i_k, t_k)/o_k$, α/γ-преемник множества состояний S во временном автомате S является одноэлементным или не существует. Таким образом, после подачи адаптивной установочной последовательности достигнутое автоматом состояние может быть однозначно определено.

Отметим, что для автомата с временными ограничениями S, временной тестовый пример $\mathsf{TTC}(I_A, O)$ имеет входной алфавит конечно-автоматной абстракции A_{S}. Более того, по определению временного тестового примера установочный тестовый пример конечно-автоматной абстракции является установочным для исходного временного автомата. Таким образом, справедливо следующее утверждение.

Утверждение 5. Пусть $\mathsf{S} = (S, I, O, \lambda_{\mathsf{S}})$ – полностью определённый недетерминированный наблюдаемый автомат с временными ограничениями. Временной тестовый пример $\mathsf{TTC}(I_A, O)$ является установочным для автомата S, если и только если он является установочным тестовым примером для конечно-автоматной абстракции A_{S}.

Действительно, каждой вход-выходной последовательности α_{FSM}/γ тестового примера $\mathsf{TTC}(I_A, O)$ из начального состояния в терминальное, для любого состоянии s временного автомата S и его абстракции A_{S} существуют как временная вход-выходная последовательность α/γ, так и соответствующая вход-выходная последовательность α_{FSM}/γ автомата A_{S} (в силу утверждения 1). Далее, аналогично утверждению 2, можно показать, что, для каждой вход-выходной последовательности α_{FSM}/γ тестового примера $\mathsf{TTC}(I_A, O)$ из начального состояния в терминальное, α/γ-преемник множества состояний S является одноэлементным или не существует во временном автомат S, если и только если α_{FSM}/γ-преемник множества состояний S является одноэлементным или не существует в конечно-автоматной абстракции A_{S}.

Для слабо инициальных автоматов, в которых не каждое состояние может быть начальным, длина кратчайшей адаптивной установочной последовательности может достигать экспоненциальной величины относительно числа состояний автомата [30]. Однако известно, что для неинициального полностью определенного наблюдаемого автомата с n состояниями, т.е. автомата, в котором каждое состояние может быть начальным, длина кратчайшей адаптивной установочной последовательности не превосходит величины n^3 [17].

Иными словами, имеет место следующее утверждение.

Утверждение 6. Если S – неинициальный полностью определённый недетерминированный наблюдаемый автомат с временными ограничениями, который имеет n состояний и обладает адаптивной установочной последовательностью, то длина кратчайшей адаптивной установочной последовательности для автомата S не превосходит величины n^3.

В следующем подразделе мы адаптируем алгоритм из [28] к нахождению кратчайшей адаптивной установочной последовательности для полностью определенных наблюдаемых неинициальных временных автоматов.

2.2.2. Синтез адаптивных временных установочных последовательностей

Адаптивная установочная последовательность может быть построена по соответствующему установочному автомату [28, 31], и далее мы адаптируем данный подход к синтезу кратчайшей временной адаптивной установочной последовательности для автомата с временными ограничениями S на основе конечно-автоматной абстракции.

Для полностью определённого наблюдаемого, возможно недетерминированного конечного автомата $\mathsf{P} = (P, I, O, h_{\mathsf{P}}), |P| > 1$, и натурального числа $L \geq 0$ установочный автомат P^L_{home} [28, 31] строится итеративно, начиная с P^0_{home}. Входной и выходной алфавиты P^L_{home} совпадают с таковыми для автомата P, в то время как состояния P^L_{home} соответствуют подмножествам состояний автомата P мощности не менее двух; также в P^L_{home} добавляется специальное состояние F. Установочный автомат P^0_{home} состоит из начального состояния P, содержащего все состояния автомата P, и специального состояния F. Переходы из состояний P и F ведут в состояние F по всем вход-выходным парам. Начальное состояние P является помеченным состоянием.

Пусть автомат $\mathsf{P}^l_{home}, l \geq 0$, с помеченными состояниями уже построен, и b – помеченное состояние автомата P^l_{home}, соответствующее в автомате P подмножеству состояний b мощности не менее двух, все переходы из которого ведут в состояние F; эти переходы удаляются при построении автомата P^{l+1}_{home}. Новые переходы добавляются по следующим правилам:

- В автомате P^{l+1}_{home} существует переход (b, i, o, F), если и только если существует $o' \in O$, такой что непустой io'-преемник множества b совпадает с b.
- Если для любого $o' \in O$ непустой io'-преемник множества b не совпадает с b, то в автомате P^{l+1}_{home} существует переход (b, i, o, b'), если и только если b' является io-преемником множества b мощности не менее двух. Если b' не является состоянием автомата P^l_{home}, то b' добавляется в множество состояний автомата P^{l+1}_{home}, становится помеченным состоянием в автомате P^{l+1}_{home}, и в автомат P^{l+1}_{home} добавляются переходы из состояния b' в состояние F по каждой вход-выходной паре.
- Переход из состояния b по входному символу i считается неопределённым, если и только если каждый io-преемник множества b не существует или является одноэлементным.
- После определения всех переходов состояние b становится непомеченным в автомате P^{l+1}_{home}.

Если автомат P^{l+1}_{home} совпадает с автоматом P^l_{home}, то автомат P^l_{home} совпадает с автоматом P^k_{home} для любого $k > l$.

Автомат P обладает адаптивной установочной последовательностью длины $L > 0$, если и только если автомат P^L_{home} не имеет полностью определённого подавтомата [28]. Проверить автомат P^L_{home} на наличие полностью определённого подавтомата можно путём итеративного удаления из P^L_{home} состояний с неопределёнными входными символами. При удалении состояния b один из неопределённых входных символов отмечается как $i(b)$. Состояния с неопределёнными переходами удаляются до тех пор, пока либо в установочном автомате не закончатся такие состояния, либо в начальном состоянии P^L_{home} появится неопределённый входной символ. Если в начальном состоянии установочного автомата P^L_{home} появился неопределённый входной символ, то автомат P не имеет полностью определённого подавтомата. Соответствующий установочный тестовый пример T высоты L может быть построен на основе сохранённых неопределённых входных символов автомата P^L_{home}. В этом случае начальное состояние t_0 автомата T соответствует начальному состоянию P автомата P^L_{home}. Входной символ $i(P), P = b$, представляет собой единственный входной символ, определённый в состоянии $t = b$ автомата T, и переход $(t, i(b), o, t')$ существует в T, если и только если в P^L_{home} (до удаления состояний) существует переход $(b, i(b), o, b')$, где $t' = b'$. Если для некоторого $o \in O$ не существует перехода $(b, i(b), o, b')$ в автомате P^L_{home}, то в T добавляется переход $(t, i(b), o, D)$, где D – тупиковое состояние. Переходы для других состояний установочного тестового примера T формируются аналогично. Состояние T становится терминальным, если не имеет исходящих переходов. Если автомат P^L_{home} обладает полностью определенным подавтоматом и $L = n^3$, то адаптивной установочной последовательности для автомата P не существует.

Другой способ проверки существования адаптивной установочной последовательности и построения установочного тестового примера, если такая последовательность существует, предложен в [29], однако алгоритм не гарантирует построения кратчайшей установочной последовательности.

Таким образом, алгоритм построения кратчайшей установочной последовательности для временного полностью определенного наблюдаемого автомата содержит следующие шаги.

Алгоритм 2 проверки существования и построения кратчайшей адаптивной установочной последовательности для автомата с временными ограничениями

Вход: полностью определённый недетерминированный наблюдаемый автомат с временными ограничениями $S = (S, I, O, \lambda_S)$

Выход: установочный тестовый пример $TTC(I_A, O)$ для временного автомата S или сообщение “Для автомата S не существует адаптивной установочной последовательности”

1. Строится конечно-автоматная абстракция A_S автомата S; $L := 1$.
2. Если $L \le n^3$ строится автомат $(A_S)^L_{home}$ и Шаг 3, иначе выдаётся сообщение “Для автомата S не существует адаптивной установочной последовательности”, и алгоритм завершает работу.
3. Если $(A_S)^{L-1}_{home} = (A_S)^L_{home}$, то выдаётся сообщение “Для автомата S не существует адаптивной установочной последовательности”, и алгоритм завершает работу; иначе, Шаг 4;
4. Если автомат $(A_S)^L_{home}$ не содержит полностью определенных подавтоматов, то Шаг 5, иначе, $L = L + 1$ и Шаг 2;
5. Строится тестовый пример $TTC(I_A, O)$ высоты L.

Корректность алгоритма 2 непосредственно следует из утверждений 5 и 6.

Для конечно-автоматной абстракции A_S (рисунок 2) временного автомата S (рисунок 1), фрагмент установочного автомата Q^2_{home} для подавтомата Q с выделенными серым цветом входными символами представлен на рисунке 4.

Соответствующий установочный тестовый пример, представляющий адаптивную установочную последовательность, изображён на рисунке 5 и имеет высоту два, в то время как кратчайшая безусловная установочная последовательность состоит из трёх входных символов.

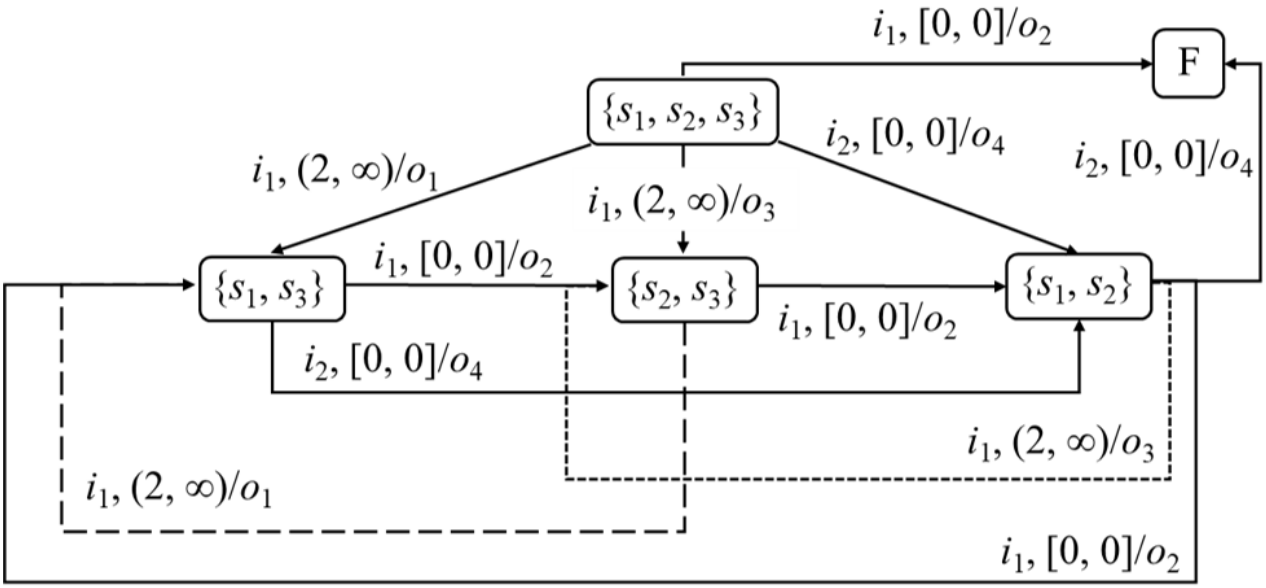


Fig. 4. A fragment of a Homing FSM for submachine Q of the FSM abstraction A_S in figure 2

Рис. 4. Фрагмент установочного автомата для подавтомата Q конечно-автоматной абстракции A_S на рисунке 2

3. Оптимизация конечно-автоматной абстракции автоматов с временными ограничениями

Согласно алгоритмам 1 и 2, сложность построения (адаптивной) установочной последовательности существенно зависит от числа входных символов автомата. Отметим, что при использовании конечно-автоматной абстракции число абстрактных входных символов существенно превышает количество входных символов исходного автомата с временными ограничениями, а именно, для временного автомата с мощностью входного алфавита $|I|$, число входных символов абстракции может достигнуть $(2(B_S+1) * |I|)$. Увеличение числа входных символов конечно-автоматной абстракции приводит к увеличению числа вершин в усечённом дереве преемников и увеличению входного алфавита установочного автомата. В настоящем разделе мы рассматриваем, каким образом можно сократить входной алфавит конечно-автоматной абстракции при проверке существования и построения адаптивной и безусловной установочных последовательностей для автомата с временными ограничениями.

Существуют различные возможности для оптимизации описанной в разделе 1.3 абстракции [21, 25], которые позволяют сократить входной алфавит при построении проверяющих тестов. В работе [26] показано, что при построении установочных последовательностей для временного автомата можно использовать одну из таких оптимизаций, а именно, рассматривать в абстракции только абстрактные входные символы, которые соответствуют целочисленным моментам времени. Однако, последнее возможно только при условии, что все временные интервалы в исходном временном автомате закрыты слева и далее мы предлагаем более общий алгоритм оптимизации, который заключается в построении *HS*-абстракции временного автомата.

Рассмотрим конечно-автоматную абстракцию (рисунок 2) временного автомата на рисунке 1. Непосредственной проверкой можно убедиться, что для некоторых пар различных абстрактных входных символов, строки таблицы переходов совпадают. Например, для входных символов $(i_1, [0, 0])$ и $(i_1, (0, 1))$, в любом состоянии автомата, множества пар (выходная реакция / следующее состояние) совпадают, и далее мы показываем, что для поиска установочных последовательностей достаточно рассмотреть только один из них. В работе [26] нами показано, что если в полностью определенном

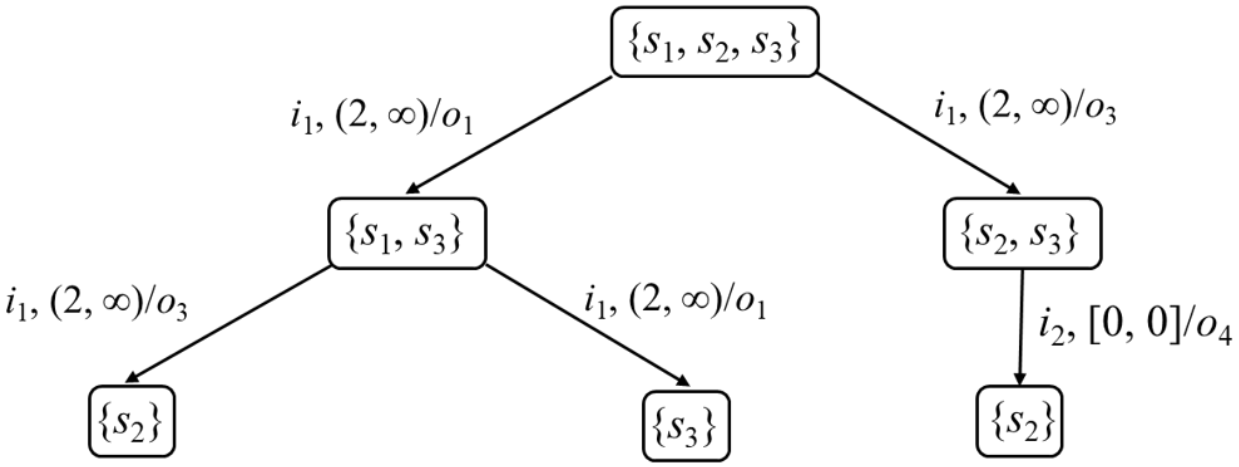


Fig. 5. A Homing test case for Timed FSM S in figure 1

Рис. 5. Установочный тестовый пример для временного автомата S на рисунке 1

автомате $\mathsf{P} = (P, I, O, h_{\mathsf{P}})$ существуют входные символы $i, i' \in I$, такие что в любом состоянии p, $(p, i, o, p') \in h_{\mathsf{P}}$, если и только если $(p, i', o, p') \in h_{\mathsf{P}}$, то справедливо следующее: автомат P обладает установочной последовательностью α, если и только такая последовательность существует для подавтомата, в котором отсутствует входной символ i'. Утверждение 5 иллюстрирует, что данное свойство сохраняется и для адаптивных установочных последовательностей, что позволяет ввести понятие *HS*-абстракции временного автомата, которая сохраняет свойства временного автомата относительно адаптивных и безусловных установочных последовательностей. Более того, в отличие от [26] мы также рассматриваем случай, когда множество пар (выходная реакция / следующее состояние) для любого состояния по одному входному символу включает в себя соответствующее множество для другого входного символа.

Утверждение 7. Пусть $\mathsf{P} = (P, I, O, h_{\mathsf{P}})$ – конечный полностью определенный автомат, где $i, i' \in I$, и для каждого состояния s справедливо, что если $(p, i, o, p') \in h_{\mathsf{P}}$, то $(p, i', o, p') \in h_{\mathsf{P}}$. Конечный автомат P обладает кратчайшей безусловной (адаптивной) установочной последовательностью α, если и только если безусловная (адаптивная) входная последовательность α', полученная из α заменой каждого входного символа i' на i, является кратчайшей установочной последовательностью для подавтомата $\mathsf{P}' = (P, I \setminus i', O, h_{\mathsf{P}'})$, где $h_{\mathsf{P}'}$ получен из h_{P} удалением переходов по входному символу i'.

Доказательство. Пусть для конечного автомата P существует кратчайшая установочная последовательность $\alpha = i_1 \dots i_l$. Следовательно, для каждой вход-выходной последовательности $\alpha/\gamma = i_1/o_1 \dots i_l/o_l$, α/γ-преемник множества состояний P является одноэлементным или не существует. Поскольку для входных символов i и i' множество пар $\{(o, p') : (p, i, o, p') \in h_{\mathsf{P}}\}$ содержится в множестве пар $\{(o, p') : (p, i', o, p') \in h_{\mathsf{P}}\}$ для любого состояния p, то для последовательности α', полученной из α заменой каждого входного символа i' на i, α/γ-преемник множества P содержит в себе α'/γ-преемник P, т.е. является одноэлементным или не существует. Таким образом, α' – кратчайшая установочная последовательность для автомата P', если и только если α – кратчайшая установочная последовательность для автомата P.

Следствие. Конечный автомат P обладает кратчайшей безусловной (адаптивной) установочной последовательностью α длины l, если и только если подавтомат $P' = (P, I \setminus i', O, h_{P'})$ обладает кратчайшей установочной последовательностью α' длины l.

Иными словами, если для пары входных символов i' и i для каждого состояния p имеет место условие: из $(p, i, o, p') \in h_{\mathsf{P}}$, следует что $(p, i', o, p') \in h_{\mathsf{P}}$, то при проверке существования и построении безусловной (адаптивной) установочной последовательности вместо конечно-автоматной абстракции достаточно рассмотреть ее полностью определенный подавтомат без входных символов i', обладающих описанным выше свойством, что приводит нас к следующем определению конечно-автоматной абстракции.

HS-абстракция $\mathsf{A}_{\mathsf{S}}^{\mathsf{HS}} = (S, I_A^{\mathsf{HS}}, O, h_{\mathsf{AS}}^{\mathsf{HS}})$ временного автомата S представляет собой полностью определенный подавтомат конечно-автоматной абстракции $\mathsf{A}_{\mathsf{S}} = (S, I_A, O, h_{\mathsf{AS}})$, получаемый итеративным удалением из A_{S} входных символов по следующему правилу. Входной символ i удаляется из конечно-автоматной абстракции, если существует входной символ $i' \neq i$, такой что для любого состояния s справедливо $\left\{(o, s') : (s, i', o, s') \in h_{\mathsf{AS}}\right\} \subseteq \left\{(o, s') : (s, i, o, s') \in h_{\mathsf{AS}}\right\}$. Входная последовательность α_{HFSM} конечно-автоматной абстракции $\mathsf{A}_{\mathsf{S}}^{\mathsf{HS}}$ может быть преобразована во временную входную последовательность исходного автомата аналогично такому преобразованию для конечно автоматной абстракции A_{S}.

Согласно сформулированным выше утверждениям, *HS*-абстракция временного автомата является конечным автоматом и «наследует» многие свойства исходного временного автомата, такие как полнота, наблюдаемость, приведенность по состояниям и связность. Кроме того, полностью определенный наблюдаемый временной автомат обладает адаптивной или безусловной установочной последовательностью, если и только если его *HS*-абстракция обладает такими установочными последовательностями.

Утверждение 8. Пусть $\mathsf{S} = (S, I, O, \lambda_{\mathsf{S}})$ — полностью определённый наблюдаемый автомат с временными ограничениями. Временная входная последовательность α является кратчайшей безусловной (адаптивной) установочной последовательностью для автомата S, если и только если соответствующая входная последовательность α_{HFSM} является кратчайшей безусловной (адаптивной) установочной последовательностью для *HS*-абстракции $\mathsf{A}_{\mathsf{S}}^{\mathsf{HS}}$.

Доказательство. Пусть α — кратчайшая установочная последовательность длины l для автомата с временными ограничениями S. Согласно утверждениям 2 и 5, α является кратчайшей временной безусловной (адаптивной) установочной последовательностью для временного автомата S, если и только если α_{FSM} является кратчайшей безусловной (адаптивной) установочной последовательностью для конечно-автоматной абстракции A_{S}. В силу утверждения 7, α_{FSM} является кратчайшей безусловной (адаптивной) установочной последовательностью для автомата A_{S}, если и только если соответствующая входная последовательность α_{HFSM} является кратчайшей безусловной (адаптивной) установочной последовательностью для *HS*-абстракции $\mathsf{A}_{\mathsf{S}}^{\mathsf{HS}}$.

Таким образом, входной алфавит конечно-автоматной абстракции может быть существенно сокращен при проверке существования или построении кратчайшей установочной последовательности по усечённому дереву преемников или установочному автомату.

Заключение

В настоящей работе была исследована задача синтеза установочных последовательностей для автоматов с временными ограничениями. Показано, как для проверки существования и синтеза адаптивных и безусловных временных установочных последовательностей можно адаптировать существующие алгоритмы для классических конечных автоматов. Оценки длин установочных последовательностей для детерминированных и недетерминированных временных автоматов совпадают с таковыми для классических автоматов. Полученные результаты основываются на построении по временному автомату соответствующей конечно-автоматной абстракции и в работе также предлагается подход к оптимизации известного метода синтеза таких абстракций.

Интересным продолжением данной работы является проверка существования и синтеза установочных последовательностей для автоматов с таймаутами, для которых переход автомата между состояниями возможен не только по входному воздействию, но и при достижении временной переменной определённого значения (таймаута). Такое расширение, с одной стороны, приводит к увеличению числа состояний в конечно-автоматной абстракции, а с другой – требует нового определения установочной последовательности.

References

[1] A. Gill, *Introduction to the Theory of Finite-State Machines*. McGraw-Hill, 1962.

[2] D. Lee and M. Yannakakis, “Testing finite state machines: state identification and verification”, *IEEE Transactions on Computers*, vol. 43, no. 3, pp. 306–320, 1994.

[3] T. N. Hibbard, “Least upper bounds on minimal terminal state experiments for two classes of sequential machines”, *Journal of the ACM*, vol. 8, no. 4, pp. 601–612, 1961.

[4] Z. Kohavi, *Switching and Finite Automata Theory*. McGraw-Hill, 1978.

[5] P. Starke, *Abstract Automata*. American Elsevier, 1972.

[6] G. Bochmann and A. Petrenko, “Protocol testing: review of methods and relevance for software testing”, in *In Proc. of International Symposium on Software Testing and Analysis*, 1994, pp. 109–124.

[7] G.-V. Jourdan, H. Ural, and H. Yenigun, “Reduced checking sequences using unreliable reset”, *Information Processing Letters*, vol. 115, no. 5, pp. 532–535, 2015.

[8] N. Kushik, J. López, A. Cavalli, and N. Yevtushenko, “Improving Protocol Passive Testing through ”Gedanken” Experiments with Finite State Machines”, in *IEEE International Conference on Software Quality, Reliability and Security*, 2016, pp. 315–322.

[9] F. Hennie, “Fault-detecting experiments for sequential circuits”, in *Proceedings of Fifth Annual Symposium on Circuit Theory and Logical Design*, 1965, pp. 95–110.

[10] T. S. Chow, “Testing software design modeled by finite-state machines”, *IEEE Transactions on Software Engineering*, vol. 4, no. 3, pp. 178–187, 1978.

[11] H.-E. Wang, K.-H. Tu, J.-H. R. Jiang, and N. Kushik, “Homing Sequence Derivation with Quantified Boolean Satisfiability”, in *Testing Software and Systems*, ser. LNCS, vol. 10533, 2017, pp. 230–242.

[12] H. Yenigun, N. Yevtushenko, N. Kushik, and J. López, “The effect of partiality and adaptivity on the complexity of FSM state identification problems”, *Trudy ISP RAN/Proceedings ISP RAS*, vol. 30, no. 1, pp. 7–24, 2018.

[13] N. Kushik and N. Yevtushenko, “On the Length of Homing Sequences for Nondeterministic Finite State Machines”, in *Implementation and Application of Automata*, ser. LNCS, vol. 7982, Springer, 2013, pp. 220–231.

[14] S. Sandberg, "Homing and Synchronizing Sequences", in *Model-Based Testing of Reactive Systems*, ser. LNCS, vol. 3472, Springer, 2005, pp. 5–33.

[15] E. Bayse, A. R. Cavalli, M. Núñez, and F. Zaïdi, "A passive testing approach based on invariants: application to the WAP", *Computer Networks*, vol. 48, no. 2, pp. 247–266, 2005.

[16] N. Kushik, K. El-Fakih, and N. Yevtushenko, "Preset and Adaptive Homing Experiments for Nondeterministic Finite State Machines", in *Implementation and Application of Automata*, ser. LNCS, vol. 6807, Springer, 2011, pp. 215–224.

[17] N. Kushik, K. El-Fakih, and N. Yevtushenko, "Adaptive Homing and Distinguishing Experiments for Nondeterministic Finite State Machines", in *Testing Software and Systems*, ser. LNCS, vol. 8254, Springer, 2013, pp. 33–48.

[18] N. Kushik and H. Yenigün, "Heuristics for Deriving Adaptive Homing and Distinguishing Sequences for Nondeterministic Finite State Machines", in *Testing Software and Systems*, ser. LNCS, vol. 9447, Springer, 2015, pp. 243–248.

[19] H. Yenigün, N. Yevtushenko, and N. Kushik, "The complexity of checking the existence and derivation of adaptive synchronizing experiments for deterministic FSMs", *Information Processing Letters*, vol. 127, pp. 49–53, 2017.

[20] M. Krichen and S. Tripakis, "Conformance testing for real-time systems", *Formal Methods in System Design*, vol. 34, no. 3, pp. 238–304, 2009.

[21] K. El-Fakih, N. Yevtushenko, and H. Fouchal, "Testing timed finite state machines with guaranteed fault coverage", in *Proc. of the 21st IFIP WG 6.1 Int. Conf. on Testing of Software and Communication Systems and 9th Int. FATES Workshop*, 2009, pp. 66–80.

[22] M. Merayo, M. Nunez, and I. Rodriguez, "Formal testing from timed finite state machines", *Computer Networks: The International Journal of Computer and Telecommunications Networking*, vol. 52, no. 2, pp. 432–460, 2008.

[23] D. Bresolin, K. El-Fakih, T. Villa, and N. Yevtushenko, "Deterministic timed finite state machines: Equivalence checking and expressive power", in *International conference GANDALF*, 2014, pp. 203–216.

[24] M. Gromov, K. El-Fakih, N. Shabaldina, and N. Yevtushenko, "Distinguishing non-deterministic timed finite state machines", in *In Formal Techniques for Distributed Systems*, ser. LNCS, vol. 5522, Springer, 2009, pp. 137–151.

[25] K. Tvardovskii A.S.and El-Fakih, M. Gromov, and N. Yevtushenko, "Testing Timed Nondeterministic Finite State Machines with the Guaranteed Fault Coverage", *Automatic Control and Computer Sciences*, vol. 51, no. 7, pp. 724–730, 2017.

[26] A. Tvardovskii and N. Yevtushenko, "Deriving homing sequences for Finite State Machines with timed guards", *System Informatics*, vol. 17, pp. 1–10, 2020.

[27] J. Hartmanis and R. Stearns, *Algebraic structure theory of sequential machines*. Prentice-Hall, 1966.

[28] E. Vinarskii, A. Tvardovskii, L. Evtushenko, and N. Yevtushenko, "Deriving adaptive homing sequences for weakly initialized nondeterministic FSMs", 2019, pp. 1–5.

[29] N. Yevtushenko, V. Kuliamin, and N. Kushik, "Evaluating the Complexity of Deriving Adaptive Homing, Synchronizing and Distinguishing Sequences for Nondeterministic FSMs", in *Testing Software and Systems*, ser. LNCS, vol. 11812, Springer, 2019, pp. 86–103.

[30] E. Vinarskii and N. Yevtushenko, "Evaluating Length of a Shortest Adaptive Homing Sequence for Weakly Initialized FSMs", 2020, pp. 1–5.

[31] N. Kushik and N. Yevtushenko, “Adaptive Homing is in P”, *Electronic Proceedings in Theoretical Computer Science*, vol. 180, pp. 73–78, 2015.

MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 27, NO. 4, 2020
journal homepage: www.mais-journal.ru

THEORY OF COMPUTING

On the Modeling of Sequential Reactive Systems by Means of Real Time Automata

E. M. Vinarskii[1], V. A. Zakharov[1]

DOI: 10.18255/1818-1015-2020-4-396-411

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

MSC2020: 68Q45
Research article
Full text in Russian

Received November 16, 2020
After revision December 1, 2020
Accepted December 16, 2020

Sequential reactive systems include hardware devices and software programs which operate in continuous interaction with the external environment, from which they receive streams of input signals (data, commands) and in response to them form streams of output signals. Systems of this type include controllers, network switches, program interpreters, system drivers. The behavior of some reactive systems is determined not only by the sequence of values of input signals, but also by the time of their arrival at the inputs of the system and the delays in computing the output signals. These aspects of reactive system computations are taken into account by real-time models of computation which include, in particular, real-time finite state machines (TFSMs). However, in most works where this class of real-time automata is studied a simple variant of TFSM semantics is used: the transduction relation computed by a TFSM is defined so that the elements of an output stream, regardless of their timestamps, follow in the same order as the corresponding elements of the input stream. This straightforward approach makes the model easier to analyze and manipulate, but it misses many important features of real-time computation. In this paper we study a more realistic semantics of TFSMs and show how to represent it by means of Labeled Transition Systems. The use of the new TFSM model also requires new approaches to the solution of verification problems in the framework of this model. For this purpose, we propose an alternative definition of TFSM computations by means of Labeled Transition Systems and show that the two definitions of semantics for the considered class of real-time finite state machines are in good agreement with each other. The use of TFSM semantics based on Labeled Transition Systems opens up the possibility of adapting well known real-time model checking techniques to the verification of sequential reactive systems.

Keywords: reactive system; finite state machine; verification; security property; labeled transition system; simulation

INFORMATION ABOUT THE AUTHORS

Evgeney Maximovich Vinarskii	orcid.org/0000-0002-7328-0942. E-mail: vinevg2015@gmail.com –
Vladimir Anatolyevich Zakharov correspondence author	orcid.org/0000-0002-3794-9565. E-mail: zakh@cs.msu.ru doctor of sciences, professor.

Funding: The reported study was funded by RFBR, project number 18-01-00854.

For citation: E. M. Vinarskii and V. A. Zakharov, “On the Modeling of Sequential Reactive Systems by Means of Real Time Automata”, *Modeling and analysis of information systems*, vol. 27, no. 4, pp. 396-411, 2020.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 27, № 4, 2020
сайт журнала: www.mais-journal.ru

THEORY OF COMPUTING

О моделировании последовательных реагирующих систем при помощи автоматов, работающих в реальном времени

Е. М. Винарский[1], В. А. Захаров[1]

DOI: 10.18255/1818-1015-2020-4-396-411

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

УДК 519.7
Научная статья
Полный текст на русском языке

Получена 16 ноября 2020 г.
После доработки 1 декабря 2020 г.
Принята к публикации 16 декабря 2020 г.

Последовательные реагирующие системы включают в себя устройства и программы, вычисления которых состоят в непрерывном взаимодействии с внешней средой, от которой они получают потоки входных сигналов (данных, команд) и в ответ на них формируют потоки выходных сигналов. К системам такого вида относятся контроллеры, сетевые коммутаторы, интерпретаторы программ, системные драйверы. Поведение некоторых реагирующих систем определяется не только последовательностью значений входных сигналов, но также временем их поступления на вход системы и задержками вычисления выходных сигналов. Эти особенности вычисления реагирующих систем хорошо учитываются вычислительными моделями реального времени, к числу которых относятся, в частности, конечные автоматы-преобразователи реального времени (Timed Finite State Machines, TFSMs). Однако в большинстве работ, посвященных изучению этой модели вычислений, используется упрощенная семантика TFSMs: элементы выходного потока, независимо от сопутствующих пометок времени, располагаются в том же порядке, в котором следуют соответствующие им элементы входного потока. Такое упрощение семантики делает модель менее адекватной для многих приложений, но зато облегчает решение задач анализа и преобразования таких автоматов. В данной статье мы изучаем модель TFSM с более реалистичной семантикой. Переход к новой модели TFSM требует и новых подходов к решению задачи верификации автоматов в этой модели. Для этой цели мы предлагаем альтернативное определение вычислений TFSM с использованием размеченных систем переходов и показываем, что два определения семантики для рассматриваемого класса автоматов реального времени хорошо взаимосвязаны друг с другом. Использование семантики TFSM, основанной на размеченных системах переходов, открывает возможность применения ранее известных методов верификации систем вычислений реального времени для анализа поведения последовательных реагирующих систем.

Ключевые слова: реагирующая система; конечный автомат; верификация; свойство безопасности; размеченная система переходов; симуляция

ИНФОРМАЦИЯ ОБ АВТОРАХ

Евгений Максимович Винарский | orcid.org/0000-0002-7328-0942. E-mail: vinevg2015@gmail.com
–
Владимир Анатольевич Захаров | orcid.org/0000-0002-3794-9565. E-mail: zakh@cs.msu.ru
автор для корреспонденции | доктор физико-математических наук, профессор.

Финансирование: Работа выполнена при финансовой поддержке гранта РФФИ N 18-01-00854.

Для цитирования: E. M. Vinarskii and V. A. Zakharov, “On the Modeling of Sequential Reactive Systems by Means of Real Time Automata”, *Modeling and analysis of information systems*, vol. 27, no. 4, pp. 396-411, 2020.

Введение

Последовательные реагирующие системы включают в себя устройства и программы, вычисления которых состоят в непрерывном взаимодействии с внешней средой; они получают извне потоки входных данных (запросов, управляющих сигналов, команд и др.) и в ответ формируют потоки выходных данных (откликов). К системам такого вида относятся контроллеры, адаптеры, сетевые коммутаторы, интерпретаторы программ, системные драйверы. Одной из самых простых математических моделей последовательных реагирующих систем являются конечные автоматы-преобразователи (Finite State Machines, FSMs), позволяющие адекватно описывать поведение синхронных систем. Вычисления этих систем разделены на такты, на каждом такте на вход системы поступает запрос и на выходе формируется соответствующий отклик, зависящий от поступившего запроса и текущего состояния системы. Модель FSM была предложена еще в начале развития теории автоматов (см. [1]). Она хорошо изучена, и на основе этой модели разработаны эффективные алгоритмы синтеза и анализа поведения синхронных управляющих систем с памятью (см., например, [2]).

Вместе с тем, во многих приложениях приходится иметь дело с асинхронными реагирующими системами, вычисления которых протекают не столь регулярно, как в синхронных системах. Поведение асинхронных систем определяется не только последовательностью значений входных сигналов, но также временем их поступления на входы системы, задержками вычисления выходных сигналов, продолжительностью пребывания системы в том или ином состоянии управления и др. Чтобы использовать конечные автоматы-преобразователи для описания поведения асинхронных реагирующих систем, необходимо снабдить автоматы дополнительными возможностями для моделирования течения физического времени.

Существуют различные подходы к расширению концепции FSM в зависимости от используемых для этой цели средств. Вероятно, наиболее развитым расширением концепции конечных автоматов, работающих в реальном времени, является модель временны́х автоматов (Timed Automata, TA), впервые предложенная в статье [3]. Каждый TA может иметь несколько специальных вещественных переменных — таймеров. В зависимости от значения таймеров определяются условия пребывания TA в состояниях управления — инварианты состояний TA, — а также условия осуществления переходов из одних состояний управления в другие состояния — предохранители переходов TA. Ход физического времени моделируется увеличением показаний таймеров на одну и ту же величину (вещественное число). Автомату также предоставлена возможность сброса показаний некоторых таймеров при совершении переходов. В рамках модели TA можно описывать многие особенности вычислений, протекающих в реальном времени, и поэтому эта модель нашла широкое применение для решения задач проектирования и верификации вычислительных систем реального времени. В работах [3—5] было показано, что многие алгебраические свойства классических конечных автоматов присущи и временны́м автоматам. Однако было обнаружено также (см. [6, 7]), что большинство задач анализа поведения временны́х автоматов оказываются вычислительно трудными. Это объясняется, в первую очередь, тем, что TA предоставлена большая свобода в обращении с таймерами.

Сложные манипуляции с таймерами могут быть неизбежными, если возникает необходимость синхронизировать несколько событий по ходу вычисления. Однако во многих практически важных случаях поведение систем управления не так сложно, и реакция системы определяется исключительно текущим состоянием управления и параметрами входных данных. Поэтому, чтобы избежать трудностей, возникающих при работе с такой сложной вычислительной моделью, как TA, авторы [8, 9] предложили снабдить FSM всего лишь одним таймером, показания которого сбрасываются при каждом переходе. Расширения модели автоматов-преобразователей такого вида получили название TFSM (Timed Finite State Machine). Были выделены три семейства TFSM в зависимости

от допустимого *modus operandi* с таймерами: (i) TFSM с синхронизированными предохранителями, которые запускают переходы только тогда, когда временна́я метка входа удовлетворяет соответствующему временно́му ограничению [8]; (ii) TFSM с тайм-аутами, которые заставляют машину выполнять переход по истечении предписанного времени ожидания входного сигнала [9]; (iii) TFSM с временны́ми предохранителями и тайм-аутами [10]. В статье [10] было установлено, что эти три семейства TFSM обладают сходными вычислительными возможностями — машины одного семейства способны моделировать вычисления машин другого семейства. Кроме того, в [10] был предложен метод абстракций TFSM — особая модификация метода регионов [3], — при помощи которой оказалось возможным эффективно сводить многие задачи синтеза и анализа TFSM к аналогичным задачам для классических конечных автоматов (см. [10—13]).

Для внешнего наблюдателя поведение реагирующей системы проявляется в преобразовании потока входных данных (запросов) с пометками времени их подачи на вход системы в поток выходных данных (откликов), которые также помечены временем их появления на выходе системы. Естественно ожидать, что элементы выходного потока следуют в порядке возрастания пометок времени. Однако отличительная особенность всех тех классов TFSM, которые были введены и исследованы в статьях [8—12], состоит в том, что в выходном потоке отклики, независимо от сопутствующих пометок времени, располагаются в том же порядке, в котором следуют соответствующие им запросы. Такое упрощение операционной семантики TFSM облегчает решение задач анализа и синтеза, но, вместе с тем, делает указанные модели TFSM непригодными для некоторых приложений.

Обратимся, в частности, к примеру контроллера программно-конфигурируемой сети (ПКС) [14]. Программно-конфигурируемая сеть состоит из множества взаимосвязанных коммутаторов, находящихся под управлением контроллера. Заголовки пакетов, поступающих во входной буфер коммутатора, сопоставляются с таблицей коммутации, чтобы выбрать подходящее правило, которое либо пересылает пакет в какой-либо выходной буфер коммутатора, либо отбрасывает его. Если для поступившего пакета в таблице коммутации нет подходящего правила, то коммутатор обращается к контроллеру. Контроллер ПКС можно рассматривать как последовательную реагирующую систему, которая получает на вход запросы на предоставление новых правил коммутации и формирует в ответ, руководствуясь некоторой политикой маршрутизации пакетов, управляющие команды, модифицирующие таблицы коммутации. Эти команды доставляются коммутаторам по каналам сети с некоторой задержкой, и поэтому порядок выполнения команд, вносящих изменения в таблицы коммутации, может отличаться от того порядка, в котором они были сформированы контроллером ПКС. Возможна такая ситуация, описанная в [15], когда контроллер получает пару запросов R_1 и R_2 на добавление новых правил от коммутаторов сети C_1 и C_2 в моменты времени t_1 и t_2 соответственно, где $t_1 < t_2$. В ответ на каждый из этих запросов R_i контроллер формирует команду A_i добавления в таблицы коммутации новых правил, и эта команда может быть доставлена коммутатору C_i и выполнена им с некоторой задержкой τ_i. Если $t_2 + \tau_2 < t_1 + \tau_1$, то в течение некоторого времени ПКС может пребывать в ошибочной конфигурации, в которой коммутатор C_1 обрабатывает пакеты по старым правилам, а коммутатор C_2 уже применяет к пакетам новые правила. Однако достижимость этой ошибочной конфигурации ПКС не может быть обнаружена при помощи модели TFSM, использующей операционную семантику, которая была введена в работах [8, 9], поскольку эта модель будет показывать, что входная последовательность запросов $(R_1, t_1), (R_2, t_2)$ преобразуется в выходную последовательность $(A_1, t_1 + \tau_1), (A_2, t_2 + \tau_2)$, в которой порядок следования команд отличается от порядка их выполнения.

Чтобы справиться с этим недостатком обычных TFSM и сделать их поведение более “реалистичным”, в статье [16] были внесены некоторые изменения в семантику TFSM. Новая модель TFSM отражает важную особенность вычислений реагирующих систем: порядок следования откликов в выходном потоке определяется временем их генерации, а не порядком следования соответствующих им

запросов во входном потоке: если запрос b поступает на вход машины после сигнала a, то возможно, что ответ на запрос a будет следовать за ответом на запрос b. Усовершенствованная операционная семантика TFSM позволяет явно обнаруживать описанное выше ошибочное поведение контроллера ПКС. Вместе с тем, новая семантика TFSM привносит дополнительные трудности в анализ поведения автоматов. Например, как показано в статье [16], даже такое простое свойство как свойство строгой детерминированности вычислений оказывается нелокальным в новой модели TFSM, и его проверка является вычислительно трудной задачей. Существенные трудности возникают также с проверкой свойств безопасности вычислений, поскольку вычисления TFSM в новой операционной семантике не обладают свойством монотонности: если входная последовательность α' продолжает входную последовательность α, то нельзя гарантировать, что соответствующая ей выходная последовательность β будет являться началом более протяженной последовательности β', которая является откликом автомата на входную последовательность α'. Отсутствие свойства монотонности значительно усложняет задачу верификации TFSM, и, в частности, препятствует применению метода абстракций, который столь успешно использовался в работах [10—13] для верификации, минимизации и тестирования TFSM, работающих в более простой операционной семантике.

Для преодоления этой принципиальной трудности, возникающей при решении задач верификации TFSM с модифицированной операционной семантикой, мы предлагаем в этой статье альтернативный подход к определению семантики TFSM, который, по нашему мнению, лучше подходит для традиционных методов проверки. Для определения вычислений TFSM мы воспользуемся размеченными системами переходов (Labeled Transition Systems, LTS), которые строятся на основе концепции конфигурации TFSM. Новая операционная семантика TFSM позволяет увидеть явную взаимосвязь этой модели вычислений с моделью ТА [3]. Однако пространство состояний в системе переходов $LTS(\mathcal{T})$, соответствующей TFSM $\mathcal{T}$, оказывается несчетным. Наша долгосрочная цель — разработать алгоритм, который для каждой TFSM $\mathcal{T}$ строит конечную $LTS_{fin}(\mathcal{T})$, которая симуляционно эквивалентна системе переходов $LTS(\mathcal{T})$ и, таким образом, может использоваться для проверки свойств вычислений TFSM $\mathcal{T}$. В этой статье мы представляем некоторые предварительные результаты наших исследований.

В разделах 1 и 2 мы вводим понятие TFSM с модифицированной операционной семантикой и показываем на простом примере, как можно использовать новую модель TFSM для адекватного описания поведения контроллеров ПКС. В разделе 3 мы определяем семантику для рассматриваемых TFSM на основе размеченных систем переходов, изучаем ее взаимосвязь с теоретико-автоматной семантикой и представляем некоторые результаты, которые устанавливают взаимосвязь рассматриваемой модели TFSM и модели временны́х автоматов [3]. В заключительном разделе мы обсуждаем некоторые новые задачи, которые возникают на основании полученных результатов.

1. Основные определения и обозначения

В этом разделе приводятся определения основных понятий, связанных с моделью вычислений конечных автоматов преобразователей, работающих в реальном времени (TFSM), и определяется операционная семантика TFSM, основанная на понятии вычисления автомата.

Рассмотрим два непустых конечных алфавита A (*входной алфавит*) и B (*выходной алфавит*). Элементы первого алфавита обозначают запросы, поступающие на вход реагирующей системы, а элементы второго алфавита — это отклики, которые реагирующая система вырабатывает в ответ на запросы. Конечную последовательность символов входного алфавита $\alpha = a_1, a_2, \ldots, a_n \in A^*$ будем называть *входным словом*, а конечную последовательность символов выходного алфавита $\beta = b_1, b_2, \ldots, b_n \in B^*$ – *выходным словом*.

Реакция конечного асинхронного автомата-преобразователя на входной символ a зависит от следующих факторов:

- текущего состояния автомата,

- времени поступления входного символа a,
- времени, которое требуется автомату для обработки этого входного символа (запроса) и формирования выходного символа (отклика).

Для моделирования времени будем использовать вещественную переменную t, принимающую значения в множестве неотрицательных вещественных чисел $\mathbb{R}_0^+$. Значение этой переменной будет обозначать время поступления входных символов и время выдачи выходных символов. Всякую возрастающую последовательность неотрицательных вещественных чисел $\tau = t_1, t_2, \ldots, t_n, \ldots$ будем называть *временно́й последовательностью*.

Допустимым интервалом (задержкой) является всякий интервал вида $g = \langle u, v \rangle$, где $0 < u \le v$, $\langle$ — один из ограничителей (или [, $\rangle$ — один из ограничителей) или]. Допустимые интервалы и задержки являются параметрами переходов TFSM. Допустимый интервал перехода TFSM из состояния s — это промежуток времени, в течение которого этот переход является активными, начиная с того момента, когда автомат оказывается в состоянии s. Задержка — это промежуток времени, в течение которого TFSM формирует отклик на входной запрос, начиная с момента поступления этого запроса. *Левым концом* (*правым концом*) допустимого интервала (задержки) $\langle u, v \rangle$ назовем число u (соответственно, число v). Задержку вида $[d, d]$ с одинаковыми концами будем называть *точечной* и использовать для ее обозначения запись d вместо $[d, d]$).

Пусть $w = x_1, x_2, \ldots x_n$ — входное (выходное) слово и $\tau = t_1, t_2, \ldots, t_n$ — временная последовательность. Тогда всякую пару (x_i, t_i), где $1 \le i \le n$, назовем *временны́м символом*, а последовательность временны́х символов

$$\alpha = (x_1, t_1), (x_2, t_2), \ldots, (x_n, t_n)$$

назовём *входным (выходным) временны́м словом* и обозначим записью (w, τ). Последнее значение t_n в последовательности τ обозначим записью $t(\alpha)$.

Пусть задано некоторое множество допустимых интервалов G и некоторое множество задержек D. Тогда *конечным автоматом-преобразователем, работающим в реальном времени* (сокращенно, TFSM, Timed Finite State Machine) над алфавитами A и B, множеством допустимых интервалов G и множеством задержек D, назовём набор $\mathcal{T} = (S, \rho, s_0)$, в котором:

- S — конечное *множество состояний управления*;
- $\rho \subseteq (S \times A \times G \times B \times D \times S)$ — конечное множество *трансдукций*;
- $s_0 \in S$ — *начальное состояние управления*.

Всякий кортеж $(s, a, g, b, d, s') \in \rho$ называется *трансдукцией* в $\mathcal{T}$. Трансдукция является элементарным действием TFSM $\mathcal{T}$. Это действие выполняется, если спустя некоторое время $t \in g$ с тех пор, как автомат оказался в состоянии управления s, на его вход поступает символ a; тогда автомат немедленно переходит в состояние s' и выдает выходной символ b спустя некоторое время $\delta \in d$ после поступления входного символа a. Таким образом, вычисление TFSM $\mathcal{T}$ состоит в преобразовании входного временно́го слова в выходное временно́е слово. Более формально это преобразование определяется так.

Вычислением TFSM $\mathcal{T} = (S, \rho, s_0)$ на входном временно́м слове

$$\alpha = (a_1, t_1), (a_2, t_2), \ldots, (a_n, t_n)$$

называется такая конечная последовательность

$$r = s_0 \xrightarrow{(a_1,t_1)/(b_1,\tau_1)} s_1 \xrightarrow{(a_2,t_2)/(b_2,\tau_2)} \ldots \xrightarrow{(a_n,t_n)/(b_n,\tau_n)} s_n,$$

что для каждого $i \in \{1, \ldots, n\}$ существует трансдукция $(s_{i-1}, a_i, g_i, b_i, d_i, s_i) \in \rho$, удовлетворяющая следующим двум условиям:

1. $t_i - t_{i-1} \in g_i$ (полагая при этом $t_0 = 0$);

2. $\tau_i = t_i + \delta$, где $\delta \in d_i$.

Будем говорить, что вычисление r TFSM $\mathcal{T}$ *преобразует* входное временно́е слово α в такое выходное временно́е слово $\beta = (b_{j_1}, \tau_{j_1}), (b_{j_2}, \tau_{j_2}), \ldots, (b_{j_n}, \tau_{j_n})$, которое является перестановкой последовательности пар $\gamma = (b_1, \tau_1), (b_2, \tau_2), \ldots, (b_n, \tau_n)$. Данное определение вычисления TFSM на входном временно́м слове отличается от аналогичного определения, используемого в статьях [10—13], тем, что в указанных работах результатом вычисления TFSM считается не временно́е слово β, а последовательность выходных временны́х символов γ, которая не обязана быть упорядоченной по возрастанию пометок времени $\tau_1, \tau_2, \ldots$.

Основной функциональной характеристикой TFSM $\mathcal{T}$ является *отношение временно́й трансдукции* $TR(\mathcal{T})$, которое представляет собой множество всех таких пар (α, β), где:

- α – входное временно́е слово;
- β – выходное временно́е слово, которое получено в результате преобразования входного временно́го слова α некоторым вычислением r TFSM $\mathcal{T}$.

Введенное здесь отношение временно́й трансдукции отличается от обычного для автоматов-преобразователей отношения словарной трансдукции, в котором входное временно́е слово α преобразуется в указанную выше последовательность выходных временны́х символов γ, а не в соответствующее этой последовательности выходное временно́е слово β.

Если все задержки в множестве D являются точечными, то TFSM $\mathcal{T}$ будем называть *TFSM с точечными задержками*. В данной работе мы рассматриваем только TFSM с точечными задержками. На рис. 1 приведен пример TFSM с точечными задержками. Этот автомат имеет вычисление

$$r = s_0 \xrightarrow{(a_1,1.5)/(b_1,4.5)} s_1 \xrightarrow{(a_2,2.7)/(b_2,3.7)} s_2 \xrightarrow{(a_3,5)/(b_3,7)} s_3,$$

которое преобразует входное временно́е слово $\alpha = (a_1, 1.5), (a_2, 2.7), (a_3, 5)$ в выходное временно́е слово $\beta = (b_2, 3.7), (b_1, 4.5), (b_3, 7)$.

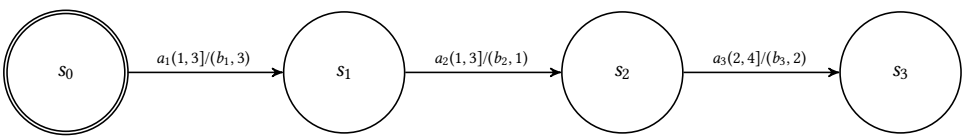


Fig. 1. TFSM with point delays

Рис. 1. TFSM с точечными задержками

2. Свойство безопасности для систем реального времени

Ошибочное поведение информационной системы может проявляться по-разному. В самом простом случае вычисление системы считается ошибочным, если оно достигает некоторого недопустимого состояния; тогда факт ошибки может быть зафиксирован, и никакие последующие действия системы не могут отменить этого факта. Поведение системы, которое не содержит такого рода ошибок, обычно называют безопасным. Для реагирующей системы недопустимым может считаться такое состояние вычисления, при достижении которого обнаруживается, что реакция системы, представленная последовательностью откликов, полученных в ответ на последовательность запросов, поданных на вход системы, не удовлетворяет некоторым заданным ограничениям. Если реагирующая система является синхронной, то эта реакция может быть определена непосредственно по заданной конечной последовательности запросов. Вычисления любой синхронной реагирующей системы устроены так, что ее реакция на последующие запросы не может изменить результаты выполнения тех ее действий, которые были инициированы предшествующими запросами. Благодаря этой особенности, требования безопасности вычислений синхронных реагирующих систем можно определять в терминах свойств отношений трансдукции, вычисляемых этими системами.

Однако поведение асинхронных реагирующих систем, работающих в реальном времени, значительно сложнее. Поступление запроса на вход асинхронной реагирующей системы и появление соответствующего этому запросу отклика на выходе системы может отделять некоторый промежуток времени. В течение этого промежутка времени система может отреагировать на последующие запросы и, тем самым, изменить результат выполнения ранее инициированных ею откликов. Образным примером этого эффекта может служить поведение человека, который, уронив вначале на пол стеклянный шар, успевает затем положить на место падения предмета мягкий ковер, и шар остается целым. В данном случае между принятием решения о выполнении действия по запросу “уронить шар на пол” и завершением осуществления этого действия реагирующая система успевает выполнить действие по следующему запросу “переместить ковер”, и, тем самым, предотвратить нарушение требования безопасности.

В этом разделе мы покажем, какие трудности возникают при проверке свойств безопасности для рассматриваемой модели TFSM, в которой семантика автоматов определяется отношением временно́й трансдукции.

Вычисление реагирующей системы обычно можно охарактеризовать бесконечной последовательностью событий (ω-словом). Тогда согласно определению [17] множество ω-слов P_{safe} называется *свойством безопасности вычислений*, если каждое ω-слово $\alpha \notin P_{safe}$ имеет такой конечный префикс w, что для любого ω-слова α', имеющего тот же префикс w, также выполняется отношение $\alpha' \notin P_{safe}$. Согласно этому определению если вычисление α не является безопасным, то ошибка может быть обнаружена после конечного числа шагов вычисления w, и ее уже нельзя будет устранить впоследствии. Вычисления реагирующих систем, работающих в реальном времени, представляются в виде бесконечных последовательностей событий с возрастающими пометками времени (временны́х ω-слов). Тогда свойство безопасности вычислений для реагирующих систем, работающих в реальном времени, можно определить так. Подмножество временны́х ω-слов $P_{safe} \subseteq (A \times \mathcal{R}^+)^\omega$ является *свойством безопасности вычислений в реальном времени*, если каждое временно́е ω-слово $\alpha \notin P_{safe}$ обладает конечным префиксом w таким, что для любого временно́го ω-слова α', имеющего то же временно́е слово w в качестве префикса, также выполняется отношение $\alpha' \notin P_{safe}$. Это определение можно использовать и в случае TFSM, заменив временны́е ω-слова α парами временны́х ω-слов (α, β), где β — результат преобразования входного временно́го ω-слова α.

В основу алгоритмов верификации свойств безопасности положены следующие соображения. Согласно приведенному выше определению свойства безопасности P_{safe} существует множество конечных слов L_{safe}, которое для любого ω-слова α и конечного слова β удовлетворяет условию: $\alpha \notin P_{safe}$ тогда и только тогда, когда α имеет такой префикс β, что $\beta \in L_{safe}$. Тогда для обнаружения того, что некоторое вычисление реагирующей системы нарушает требование безопасности P_{safe}, необходимо и достаточно убедиться, что какое-либо начало этого вычисления характеризуется некоторым словом $\beta \in L_{safe}$. Язык L_{safe} определяется свойством P_{safe} неоднозначно, для его описания можно использовать логические или алгебраические формулы, различные виды автоматов и это придает большую гибкость и разнообразие алгоритмам верификации свойств безопасности вычислений реагирующих систем. Однако при верификации моделей TFSM, описанных в предыдущем разделе, этот подход сталкивается с принципиальными трудностями.

Рассмотрим модель коммутатора ПКС C, который получает от контроллера ПКС команды на обновление правил в таблицах коммутации пакетов или запросы на предоставление информации об использовании правил коммутации и отправляет в ответ подтверждения о выполнении команд или запрашиваемую информацию. Может случиться так (см. [15]), что контроллер ПКС при получении входного временного слова $(R_1, t_1), (R_2, t_2)$, где $t_1 < t_2$, выдаёт входное временное слово $(A_2, \tau_1), (A_1, \tau_2)$, где $\tau_2 < \tau_1$, т.е. второе правило будет установлено раньше первого. При моделировании вычислений коммутатора ПКС необходимо учитывать, что выполнение команд занимает некоторое время, и

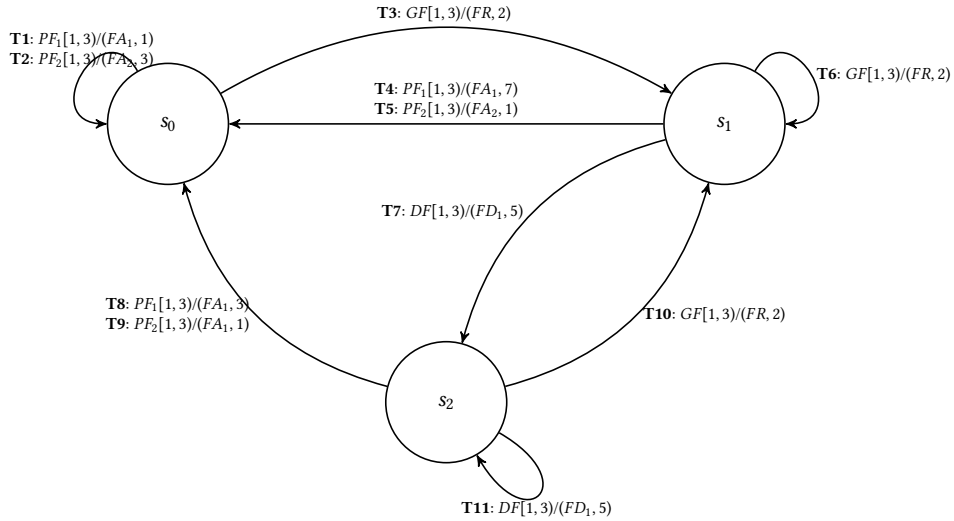


Fig. 2. Model of SDN switch $\mathcal{C}$

Рис. 2. Модель коммутатора ПКС $\mathcal{C}$

поэтому подтверждения о внесенных изменениях в таблицы коммутации доставляются контроллеру с некоторой задержкой. TFSM, описывающий поведение коммутатора $\mathcal{C}$, изображен на рис. 2. Отметим, что $\mathcal{C}$ обрабатывает следующие входные запросы:

- PF_i — добавить правило коммутации пакетов r_i в таблицу коммутатора;
- GF_i — предоставить информацию об использовании правила коммутации r_i;
- DF_i — удалить из таблицы правило коммутации пакетов r_i.

При этом коммутатор $\mathcal{C}$ отправляет следующие данные:

- FA_i — подтверждение о добавлении правила r_i в таблицу коммутации;
- FR_i — информация о частоте применения правила r_i;
- FD_i — подтверждение об удалении правила r_i из таблицы коммутации.

Помимо корректной обработки поступающих команд и запросов коммутатор также должен обеспечить своевременное предоставление информации контроллеру. Это означает, в частности, что подтверждения о выполненной модификации таблиц коммутации должны быть доставлены контроллеру в том же порядке, в каком поступали команды на проведение соответствующих модификаций. Как видно из приведенных выше определений, это требование является свойством безопасности вычислений коммутатора (P_{safe}). Рассмотрим вычисление предложенной модели коммутатора $\mathcal{C}$ на входном временно́м слове

$$\alpha = (GF_1, 1.5), (PF_1, 3.7), (PF_2, 5.5).$$

TFSM $\mathcal{C}$ преобразует входное слово α в выходное временно́е слово

$$\beta = (FR_1, 3.5), (FA_2, 8.5), (FA_1, 10.7).$$

Таким образом, реакция автомата явно показывает, что требование соответствия порядка следования сообщений-подтверждений порядку поступления команд нарушено и, следовательно, это вычисление нарушает требование безопасности. Тем не менее, входное слово α можно продолжить таким образом:

$$\alpha' = (GF_1, 1.5), (PF_1, 3.7), (PF_2, 5.5), (PF_1, 6.7),$$

что TSFM $\mathcal{C}$ преобразует его в выходное временное слово

$$\beta' = (FR_1, 3.5), (FA_1, 7.7), (FA_2, 8.5), (FA_1, 10.7),$$

которое удовлетворяет свойству безопасности P_{safe}. Данный пример показывает, что операционная семантика TFSM, которая определена в терминах преобразования временны́х слов, мало пригодна для эффективного решения на ее основе задачи верификации свойств безопасности вычислений TFSM. Это объясняется, прежде всего, тем, что отношение временно́й трансдукции не обладает свойством монотонности: если вычисление r_2 TFSM $\mathcal{T}$ является продолжением вычисления r_1, и при этом каждое из вычислений $r_i, i = 1, 2$, преобразует входное временно́е слово α_i в выходное временно́е слово и β_i, то слово α_1 является префиксом слова α_2, но слово β_1 не обязано быть префиксом слова β_2. Вычисления TFSM в той операционной семантике, которая использовалась в работах [10—13], обладают свойством монотонности, и поэтому для этих моделей реагирующих систем таких трудностей проверки свойств безопасности вычислений не возникает.

Для преодоления этого затруднения целесообразно определить альтернативную операционную семантику TFSM при помощи размеченных систем переходов (LTS), которая не имела бы отмеченного выше недостатка. Далее необходимо установить взаимосвязь между двумя введенными семантиками TFSM — теоретико-автоматной семантикой и семантикой, основанной на LTS, — и показать корректность альтернативной операционной семантики. После этого можно приступать к разработке алгоритмов верификации свойств безопасности вычислений TFSM, основываясь на размеченных системах переходов, соответствующих этим автоматам-преобразователям.

3. Размеченные системы переходов для TFSM

Операционная семантика временны́х автоматов-преобразователей, определенная в разделе 1, хорошо подходит для моделирования реагирующих систем, поскольку она отражает многие важные эффекты вычислений, выполняемых в реальном времени. Однако эта семантика препятствует применению традиционных методов и инструментов анализа поведения систем реального времени, поскольку в ней отсутствует понятие состояния вычислений как моментального снимка вычислительного процесса. Чтобы преодолеть этот недостаток, мы вводим понятие конфигурации, которая позволяет представлять поведение временного конечного трансдьюсера с помощью размеченной системы переходов на множествах таких конфигурациях.

3.1. Конфигурации размеченной системы переходов

Конфигурация TFSM $\mathcal{T}$ содержит информацию о текущем состоянии управления, показании таймера TFSM, а также о всех тех откликах (символах выходного алфавита), которые уже были сформированы ранее, но не были опубликованы, поскольку срок их задержки еще не истек. На множестве конфигураций TFSM $\mathcal{T}$ определены переходы трех типов: продвижение времени, получение входного символа (запроса) и выдача выходного символа (отклика).

Конфигурация TFSM $\mathcal{T} = (S, \rho, s_0)$ — это всякая тройка вида $q = \langle s, t, TC \rangle$, где

- $s \in S$ — состояние управления $\mathcal{T}$, которое будем обозначать записью $q.s$;
- $t \in \mathcal{R}_0^+$ — *входная метка времени*, которую будем обозначать записью $q.t$;
- $TC = \{(b_1, \tau_1), (b_2, \tau_2), \ldots, (b_n, \tau_n)\}$ — *выходной временно́й контекст* конфигурации q, который представляет собой множество выходных временны́х символов и обозначается записью $q.c$.

Выходная метка времени $q.t$ обозначает время, прошедшее с момента начала вычисления или приема автоматом последнего входного символа, а выходной временно́й контекст $q.c$ — это множество откликов TFSM $\mathcal{T}$, приготовленных для выдачи на выходе автомата, с указанием текущего времени их задержки. Множество всех конфигураций TFSM $\mathcal{T}$ обозначим записью $Q(\mathcal{T})$.

Размером конфигурации q назовем количество $|q.c|$ элементов в выходном временно́м контексте конфигурации q. Обозначим записью $q.T$ значение $min(\tau_1, \ldots, \tau_n)$, если $|q.c| = n > 0$; если $|q.c| = 0$, то будем полагать $q.T = \infty$. Значение $q.T$ — это время, спустя которое TFSM способна выдать очередной выходной символ без взаимодействия с внешней средой. Если $q.c = \{(b_1, \tau_1), \ldots, (b_n, \tau_n)\}$,

$b \in B$, и $\delta \le q.T$, то условимся обозначать записью $q + \delta$ конфигурацию

$$\langle q.s, t + \delta, \{(b_1, \tau_1 - \delta), \dots, (b_n, \tau_n - \delta)\}\rangle,$$

записью $in(q, b, d)$ — конфигурацию $\langle q.s, 0, q.c \cup \{(b, d)\}\rangle$, а записью $out(q, b)$ — конфигурацию $\langle q.s, q.t, q.c \setminus (b, 0)\rangle$.

При сопоставлении введенного здесь понятия конфигурации TFSM и понятия состояния временного автомата (ТА) [3] можно заметить, что параметры $t, \tau_1, \dots, \tau_n$ конфигурации q играют роль показаний таймеров состояния ТА, а совокупность параметров $s, b_1, \dots, b_n$ выполняет функции состояния управления ТА.

3.2. Размеченная система переходов TFSM

Размеченная система переходов $\mathcal{LTS}(\mathcal{T})$ TFSM $\mathcal{T} = (S, \rho, s_0)$ над входным алфавитом A, выходным алфавитом B, множествами допустимых интервалов G и задержек D — это тройка $(Q(\mathcal{T}), q_0, \rho_{\mathcal{T}})$, где

- $q_0 = \langle s_0, 0, \varnothing\rangle$ — *начальная конфигурация*,
- $\rho_{\mathcal{T}} \subseteq (Q \times \mathcal{R}^+ \times Q) \cup (Q \times A \times Q) \cup (Q \times B \times Q)$ — *отношение переходов*, определённое для каждой конфигурации $q = \langle s, t, \{(b_1, \tau_1), (b_2, \tau_2), \dots, (b_m, \tau_m)\}\rangle$ следующим образом:
  1) для любого числа $\delta \in \mathcal{R}^+$ такого, что $\delta < q.T$, в множестве $\rho_{\mathcal{T}}$ существует *переход продвижения времени* $q \xrightarrow{\delta} q + \delta$;
  2) для каждой трансдукции $(s, a, \langle u, v\rangle, c, d, s')$ TFSM $\mathcal{T}$ такой, что $t \in \langle u, v\rangle$, в множестве $\rho_{\mathcal{T}}$ существует *переход приема запроса* $q \xrightarrow{a} in(q, c, d)$;
  3) для каждой пары $(b, 0) \in q.c$ в множестве $\rho_{\mathcal{T}}$ существует *переход выдачи отклика* $q \xrightarrow{b} out(q, b)$.

Произвольную последовательность переходов $\pi = q_1 \xrightarrow{x_1} q_2 \xrightarrow{x_2} \cdots q_k \xrightarrow{x_k} q_{k+1}$, каждый из которых исходит из той же конфигурации, которой достиг предыдущий переход, назовем *путем* в $\mathcal{LTS}(\mathcal{T})$. Если конфигурация q_1, из которой исходит путь π, — это начальная конфигурация $\mathcal{LTS}(\mathcal{T})$, то такой путь назовем *начальным путем*. Особо выделим также пути, которые опустошают выходной временно́й контекст конфигураций без прочтения входных символов. *Завершающим путем* из конфигурации q будем называть путь $\overline{\pi}(q)$ $\mathcal{LTS}(\mathcal{T})$, который удовлетворяет следующему рекурсивному определению:

1) если $q.c = \varnothing$, то $\overline{\pi}(q)$ — пустая последовательность переходов;
2) если $q.c \ne \varnothing$ и $(b, q.T) \in q.c$ — любой из выходных временны́х символов с наименьшей задержкой $q.T$, то $\overline{\pi}(q) = q \xrightarrow{q.T} q + q.T \xrightarrow{b} \overline{\pi}(out(q + q.T, b))$.

Путь π будем называть *приведенным*, если в нем не встречаются несколько идущих подряд переходов продвижения времени. Очевидно, что любые два идущих подряд перехода продвижения времени $q \xrightarrow{\delta_1} q' \xrightarrow{\delta_2} q''$ можно заменить одним переходом $q \xrightarrow{\delta_1+\delta_2} q''$. Проделав это преобразование для всех пар идущих подряд переходов продвижения времени в произвольном пути π, который ведет из конфигурации q_1 в конфигурацию q_k, получим приведенный путь π' из q_1 в q_k, в котором переходы продвижения времени перемежаются переходами приема запросов и выдачи откликов. Этот приведённый путь π' определяется по пути π однозначно; назовем его *редукцией* пути π.

Начальный путь π назовем *полным*, если последняя конфигурация пути q_k удовлетворяет условию $q_k.c = \varnothing$. Каждому пути π в $\mathcal{LTS}(\mathcal{T})$ поставим в соответствие пару, состоящую из входного и выходного временны́х слов $TR(\pi) = (\alpha, \beta)$, по следующим правилам:

1. Если π — это пустой путь, то $TR(\pi) = (\varepsilon, \varepsilon)$.
2. Пусть π' — путь в $\mathcal{LTS}(\mathcal{T})$ из конфигурации q_0 в конфигурацию q', для которого $TR(\pi') = (\alpha', \beta')$, и пусть путь π получен из пути π' добавлением перехода $E = q' \xrightarrow{x} q$. Если переход E — это переход

- продвижения времени, то $TR(\pi) = TR(\pi')$;
- приема запроса, то $TR(\pi) = (\alpha, \beta')$, где $\alpha = \alpha', (x, t(\alpha') + q'.t)$;
- выдачи отклика, то $TR(\pi) = (\alpha', \beta)$, где $\beta = \beta', (x, t(\alpha') + q'.t)$.

Исходя из этого определения, нетрудно заметить, что для любого пути π и его редукции π' верно равенство $TR(\pi) = TR(\pi')$.

Записью $TR(\mathcal{LTS}(\mathcal{T}))$ обозначим множество всех пар $TR(\pi)$, соответствующих полным путям π в $\mathcal{LTS}(\mathcal{T})$. Как видно из предыдущего замечания, при изучении отношения $TR(\mathcal{LTS}(\mathcal{T}))$ можно ограничиться рассмотрением одних лишь приведённых путей.

Покажем, что размеченная система переходов $\mathcal{LTS}(\mathcal{T})$ полностью характеризует поведение TFSM $\mathcal{T}$.

Утверждение 1. *Для любого TFSM $\mathcal{T}$ верно равенство $TR(\mathcal{T}) = TR(\mathcal{LTS}(\mathcal{T}))$.*

Доказательство. Вначале докажем индукцией по n, что для любого вычисления TFSM $\mathcal{T}$, которое преобразует входное временно́е слово $\alpha = (a_1, t_1), \ldots, (a_n, t_n)$ в выходное временно́е слово $\beta = (b_1, \tau_1), \ldots, (b_n, \tau_n)$ и оканчивается в состоянии s_n, существует приведенный полный путь π в $\mathcal{LTS}(\mathcal{T})$, который ведет в конфигурацию $(s_n, \tau_n - t_n, \emptyset)$ и удовлетворяет равенству $TR(\pi) = (\alpha, \beta)$. Отсюда будет следовать включение $TR(\mathcal{T}) \subseteq TR(\mathcal{LTS}(\mathcal{T}))$.

Базис индукции. Пусть TFSM $\mathcal{T}$ имеет вычисление $r = s_0 \xrightarrow{(a_1,t_1)/(b_1,\tau_1)} s_1$. Тогда согласно определению вычисления TFSM, существует трансдукция $(s_0, a_1, g, b_1, d, s_1) \in \rho$, удовлетворяющая условиям $t_1 \in g$ и $\tau_1 = t_1 + d$. Тогда согласно определению отношения переходов в $\mathcal{LTS}(\mathcal{T})$ существует полный путь $\pi = q_0 \xrightarrow{t_1} q_1 \xrightarrow{a_1} q_2 \xrightarrow{d} q_3 \xrightarrow{b_1} q_4$. Нетрудно видеть, что $TR(\pi) = ((a_1, t_1), (b_1, \tau_1))$ и при этом $q_4 = (s_1, d, \emptyset)$.

Индуктивный переход. Пусть TFSM $\mathcal{T}$ имеет вычисление r', которое преобразует входное временно́е слово $\alpha' = (a_1, t_1), \ldots, (a_n, t_n)$ в выходное временное слово $\beta' = (b_1, \tau_1), \ldots, (b_n, \tau_n)$, достигает состояния управления s_n, и может быть продолжено выполнением некоторой трансдукции $s_n \xrightarrow{(a,t)/(b,\tau)} s_{n+1}$. Тогда по индуктивному предположению вычислению r' соответствует некоторый полный редуцированный путь π' в $\mathcal{LTS}(\mathcal{T})$, для которого имеет место равенство $TR(\pi') = (\alpha, \beta)$. Рассмотрим в этом пути последний переход приема запроса $q \xrightarrow{a_n} q'$, и пусть k — это наименьшее число, для которого выполнено неравенство $t_n \le \tau_k$. Тогда путь π' можно разделить на две половины π'_1 и π'_2, первая из которых оканчивается переходом приема запроса $q \xrightarrow{a_n} q'$. Так как это последний переход приема запроса в пути π', вторая половина π'_2 пути π' имеет вид

$$\pi'_2 = q' \xrightarrow{\tau_k - t_n} q'_1 \xrightarrow{b_k} q'_2 \xrightarrow{\tau_{k+1} - \tau_k} q'_3 \xrightarrow{b_{k+1}} \cdots \xrightarrow{\tau_n - \tau_{n-1}} q'_{m-1} \xrightarrow{b_n} q'_m,$$

и состоит только из переходов продвижения времени и переходов выдачи откликов $b_k, \ldots, b_n$. При этом для всех конфигураций q'_i этой половины пути выполняется равенство $q'_i.s = s_n$.

Предположим, что параметры t и τ выполненной трансдукции $s_n \xrightarrow{(a,t)/(b,\tau)} s_{n+1}$ удовлетворяют неравенствам $\tau_i \le t \le \tau_{i+1}$ и $\tau_j \le \tau \le \tau_{j+1}$ для некоторой пары чисел i, j, $k \le i \le j \le n$. Тогда вычисление r TFSM $\mathcal{T}$, которое продолжает вычисление r' указанным выполнением трансдукции преобразует входное временно́е слово $\alpha = (a_1, t_1), \ldots, (a_n, t_n), (a, t)$ в выходное временно́е слово

$$\beta = (b_1, \tau_1), \ldots, (b_j, \tau_j), (b, \tau), (b_{j+1}, \tau_{j+1}), \ldots, (b_n, \tau_n).$$

Но, как можно заметить, для указанных значений параметров t и τ, вставив переход приема запроса a и переход выдачи отклика b в последовательность переходов пути π'_2, можно построить

следующий приведенный путь из конфигурации q' в $\mathcal{LTS}(\mathcal{T})$:

$$\begin{aligned}\pi_2 \quad = \quad & q' \xrightarrow{\tau_k - t_n} q'_1 \xrightarrow{b_k} q'_2 \xrightarrow{\tau_{k+1}-\tau_k} q'_3 \xrightarrow{b_{k+1}} \cdots \xrightarrow{\tau_i - \tau_{i-1}} q'_{2i-1} \xrightarrow{b_i} q'_{2i} \\ & \xrightarrow{t-\tau_i} q_{2i+1} \xrightarrow{a} q_{2i+2} \xrightarrow{\tau_{i+1}-t} q_{2i+3} \xrightarrow{b_{i+1}} \cdots \xrightarrow{b_j} q_{2j} \\ & \xrightarrow{\tau-\tau_j} q_{2j+1} \xrightarrow{b} q_{2j+2} \xrightarrow{\tau_{j+1}-\tau} q_{2j+3} \xrightarrow{b_{j+1}} \cdots \xrightarrow{b_n} q_{m+4}.\end{aligned}$$

Сцеплением путей π'_1 и π_2 получаем последовательность переходов π в $\mathcal{LTS}(\mathcal{T})$, являющуюся полным путем, которому соотвествует пара временны́х слов (α, β).

Аналогичным образом можно построить путь π в $\mathcal{LTS}(\mathcal{T})$, которому соответствует пара временны́х слов (α, β), и в тех случаях, когда параметры t и τ выполненной трансдукции $s_n \xrightarrow{(a,t)/(b,\tau)} s_{n+1}$ удовлетворяют неравенствам $\tau_n \le t$ или $\tau_n \le \tau$.

Тем самым, обоснование индуктивного перехода завершено, и вместе с этим получено обоснование включения $TR(\mathcal{T}) \subseteq TR(\mathcal{LTS}(\mathcal{T}))$.

Для обоснования включения $TR(\mathcal{LTS}(\mathcal{T})) \subseteq TR(\mathcal{T})$ достаточно показать, что каждый полный редуцированный путь π в $\mathcal{LTS}(\mathcal{T})$, для которого $TR(\pi) = (\alpha, \beta)$, соответствует вычислению TFSM $\mathcal{LTS}(\mathcal{T})$, которое преобразует входное временно́е слово α в выходное временное слово β. Но, как видно из определения множества пар $TR(\pi) = (\alpha, \beta)$, вычисление r, соответствующее пути π, однозначно определяется переходами приема запросов вида $q \xrightarrow{a} in(q, c, d)$ в этом пути. При этом временны́е пометки входного слова α, на котором проводится вычисление r, и временны́е пометки соответствующего выходного слова β однозначно определяются переходами продвижения времени в пути π. Таким образом, вычисление r восстанавливается по пути π в однозначно. □

Основное преимущество $\mathcal{LTS}(\mathcal{T})$ заключается в том, что эта система переходов имеет точно такое же устройство, какое имеют трассовые модели, используемые для верификации временны́х автоматов [3]. Тем самым, мы установили взаимосвязь рассматриваемой модели TFSM и модели временны́х автоматов, и это позволяет применять инструментальные средства верификации моделей программ, подобные комплексу верификации Uppaal [18], для анализа поведения TFSM с модифицированной операционной семантикой. Однако такой анализ сопряжен с определенными трудностями: пространство состояний $\mathcal{LTS}(\mathcal{T})$ может быть континуально-бесконечным. Отчасти это связано с тем, что размер конфигураций в $\mathcal{LTS}(\mathcal{T})$ в общем случае может быть неограниченно большим. Тем не менее, при определенных условиях этой неограниченности размера конфигураций можно избежать.

Пусть u, v — пара вещественных чисел, удовлетворяющих неравенству $0 < u \le v$. TFSM $\mathcal{T} = (S, \rho, s_0)$ называется (u, v)-*прогрессивным*, если для каждой трансдукции $(s, a, g, b, d, s') \in \rho$ ее допустимый интервал $g = (u', v')$ удовлетворяет условиям $u \le u'$ и $v' \le v$, т.е. допустимые интервалы всех трансдукций $\mathcal{T}$ вложены в интервал (u, v). Тогда справедливо следующее утверждение.

Утверждение 2. *Если $\mathcal{T}$ является (u, v)-прогрессивной TFSM с точечными задержками, то существует такое целое число ℓ, что $|q.c| < \ell$ выполняется для любой конфигурации q, достижимой в $\mathcal{LTS}(\mathcal{T})$ из начальной конфигурации q_0.*

Доказательство. Пусть в трансдукциях $\mathcal{T}$ минимальная левая граница допустимого интервала равна $u > 0$, а максимальная точечная задержка равна $d_{max} > 0$.

Рассмотрим $\ell = \lceil \frac{d_{max}}{u} \rceil$. Предположим, что из начальной конфигурации q_0 достижима конфигурация q, для которой $|q.c| = \ell + 1$. Это означает, что TFSM $\mathcal{T}$ имеет вычисление

$$\begin{aligned}r \quad = \quad & s_0 \xrightarrow{(a_1,t_1)/(b_1,d_1)} \cdots \xrightarrow{(a_{n-\ell-1},t_{n-\ell-1})/(b_{n-\ell-1},d_{n-\ell-1})} s_{n-\ell-1} \xrightarrow{(a_{n-\ell},t_{N-\ell})/(b_{n-\ell},d_{N-\ell})} \\ & \cdots \xrightarrow{(a_{n-1},t_{N-1})/(b_{N-1},d_{n-1})} s_{n-1} \xrightarrow{(a_n,t_n)/(b_n,d_n)} s_n,\end{aligned}$$

которому в размеченной системе переходов $\mathcal{LTS}(\mathcal{T})$ соответствует путь из начальной конфигурации q_0 в конфигурацию

$$q = \langle s_n, t_n, \{(b_{j_1}, d_{j_1} - \sum_{i=j_1}^{n} t_i), (b_{j_2}, d_{j_2} - \sum_{i=j_2}^{n} t_i), \dots, (b_{j_{\ell+1}}, d_{j_{\ell+1}} - \sum_{i=j_{\ell+1}}^{n} t_i)\}\rangle,$$

где $j_1, \dots, j_{\ell+1}$ такие, что $1 \le j_1 < \dots < j_{\ell+1} \le n$, следовательно, $j_1 < n - \ell - 1$.

Тогда для временно́й метки символа b_{j_1} в этой конфигурации верны неравенства

$$d_{j_1} - \sum_{i=j_1}^{n} t_i \le d_{max} - \sum_{i=j_1}^{n} t_i \le d_{max} - \sum_{i=j_1}^{n} u \le d_{max} - u(\ell + 1).$$

Поскольку $\ell = \lceil \frac{d_{max}}{u} \rceil$, приходим к заключению о том, что $d_{j_1} - \sum_{i=j_1}^{n} t_i < 0$, вопреки тому, что все временные метки в конфигурациях являются неотрицательными. □

Ещё одна трудность при анализе размеченной системы переходов $\mathcal{LTS}(\mathcal{T})$ обусловлена возможностью продолжить любой путь только переходами продвижения времени. Чтобы обнаружить и исключить из дальнейшего рассмотрения такие избыточные пути, мы выделим из всего множества конфигураций так называемые *блокирующие конфигурации* [17]. Для каждого состояния s TFSM $\mathcal{T}$ обозначим записью $up(s)$ максимальную верхнюю границу v допустимых интервалов $g = (u, v)$ всех трансдукций (s, a, g, b, d, s'), исходящих из состояния управления s. Конфигурация q называется *блокирующей конфигурацией*, если $q.c = \emptyset$ и $q.t > up(q.s)$. Все остальные конфигурации в $Q(\mathcal{T})$ называются *прогрессивными*. Из определения следует, что только прогрессивные конфигурации имеют значение для анализа преобразования входных временны́х слов в выходные. Если путь в $\mathcal{LTS}(\mathcal{T})$ достигает блокирующей конфигурации, то не существует такого продолжения пути в котором присутствуют переходы приема запросов или выдачи откликов. Поэтому анализировать пути, исходящие из таких конфигураций, нет необходимости. Следующее утверждение предлагает алгоритм для обнаружения *блокирующих конфигураций* в (u, v)-прогрессивных TFSM.

Утверждение 3. *Для каждого состояния s в (u, v)-прогрессивной TFSM $\mathcal{T}$ существует такая величина $c_s \ge 0$, что произвольная конфигурация $q = (s, TC)$ является блокирующей тогда и только тогда, когда $q.t > c_s$.*

Доказательство. Пусть s — состояние управления TFSM $\mathcal{T}$, и пусть $v = up(s)$. Рассмотрим множество конфигураций, $Q_s = \{q \in Q \mid (q.s = s) \wedge (q.t \ge v) \wedge (q.c = \emptyset)\}$ и соответствующее множество временных меток $TS_s = \{t \mid (q \in Q_s) \wedge (q.t = t)\}$. Легко видеть, что TS_s имеет точную нижнюю грань c_s, которая удовлетворяет условиям утверждения. □

Однако количество прогрессивных конфигураций может быть континуально-бесконечно. Поэтому следующим шагом в наших дальнейших исследованиях будет отыскание такого отношения эквивалентности ~ на множестве прогрессивных конфигураций $\mathcal{LTS}(\mathcal{T})$, чтобы фактор-модель $\mathcal{LTS}_{fin}(\mathcal{T}) = \mathcal{LTS}(\mathcal{T})/\sim$ была конечной и при этом симуляционно эквивалентной системе переходов $LTS(\mathcal{T})$. Построенная таким образом модель $\mathcal{LTS}_{fin}(\mathcal{T})$ открывает возможности применения традиционных методов для верификации реагирующих систем реального времени, представленных временны́ми конечными автоматами-преобразователями.

4. Заключение

Основной результат статьи — это описание и исследование новой операционной семантики для модели TFSM, предложенной ранее в работе [16]. Благодаря этому удалось показать, что вычисления TFSM можно моделировать конечными временны́ми автоматами. В связи с этим возникает ряд вопросов, касающихся сложности такого моделирования. Например, интересно выяснить, какого количества таймеров и состояний достаточно иметь конечному временно́му автомату для моделирования заданной TFSM. В том случае, если будет обнаружено, что сложность временно́го автомата, моделирующего вычисления TFSM, существенно превосходит размер этой TFSM, сведение задачи верификации TFSM к аналогичной задаче для временны́х автоматов может оказаться нецелесообразным, и для этой цели будет выгоднее разрабатывать независимые методы анализа поведения автоматов-преобразователей.

В связи с этим можно заметить, что использование методов верификации, опирающихся на размеченные системы переходов осложняется тем, что количество конфигураций в $LTS(\mathcal{T})$, которая представляет всевозможные вычисления TFSM $\mathcal{T}$, может быть несчётно. Поэтому необходимо создать такие методы преобразования бесконечной системы переходов $LTS(\mathcal{T})$ в конечную систему переходов $\mathcal{LTS}_{fin}(\mathcal{T})$, связанную с $LTS(\mathcal{T})$ отношениями. В наших дальнейших исследованиях мы собираемся найти подходящее отношение симуляции ~, которое (i) сохраняет преобразование входных временных слов в выходные временные слова, осуществляемое TFSM $\mathcal{T}$, и (ii) предложить эффективный алгоритм построения $\mathcal{LTS}_{fin}(\mathcal{T})$ такой, что $LTS(\mathcal{T})$ ~ $\mathcal{LTS}_{fin}(\mathcal{T})$. Мы предполагаем, что такое отношение может быть найдено при помощи определения понятия шаблона конфигурации, которое будет играть ту же роль для анализа TFSM, что для анализа свойств временных автоматов играет система регионов (см. [3]).

Авторы выражают признательность рецензенту за ценные замечания, позволившие улучшить облик этой статьи.

References

[1] A. Gill *et al.*, «Introduction to the Theory of Finite-state Machines», 1962.

[2] A. Y. Savelev, «Prikladnaya teoriya cifrovyh avtomatov», 1987, In Russian.

[3] R. Alur and D. Dill, «A theory of timed automata», *Theoretical computer science*, vol. 126, no. 2, pp. 183–235, 1994.

[4] E. Asarin, P. Caspi, and O. Maler, «Timed regular expressions», *Journal of the ACM*, vol. 49, no. 2, pp. 1–35, 2001.

[5] E. Asarin, P. Caspi, and O. Maler, «A Kleene theorem for timed automata», in *Proceedings of 12-th Annual IEEE Symposium on Logic in Computer Science (LICS'97)*, IEEE, 1997, pp. 160–171.

[6] R. Alur and P. Madhusudan, «Decision Problems for Timed Automata: A Survey, Formal Methods for the Design of Real-Time Systems», in *Proceedings of the 4-th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM'04)*, Springer, 2004, pp. 1–24.

[7] S. Lasota and I. Walukiewicz, «Alternating timed automata», *ACM Transactions on Computational Logic (TOCL)*, vol. 9, no. 2, pp. 1–26, 2008.

[8] M. Gromov, K. El-Fakih, N. Shabaldina, and N. Yevtushenko, «Distinguing Non-deterministic Timed Finite State Machines», in *Formal Techniques for Distributed Systems, Lecture Notes in Computer Science*, vol. 5522, Springer, 2009, pp. 137–151.

[9] M. G. Merayo, M. Nunez, and I. Rodriguez, «Formal testing from timed finite state machines», *Computer networks*, vol. 52, no. 2, pp. 432–460, 2008.

[10] D. Bresolin, K. El-Fakih, T. Villa, and N. Yevtushenko, «Deterministic Timed Finite State Machines: Equivalence Checking and Expressive Power», *Proceedings of the 5-th International Symposium on Games, Automata, Logics and Formal Verification*, pp. 203–216, 2014.

[11] A. Tvardovskii and N. Yevtushenko, «Minimizing timed Finite State Machines», *Tomsk State University Journal of Control and Computer Science*, vol. 29, no. 4, pp. 77–83, 2014.

[12] A. S. Tvardovskii, N. V. Yevtushenko, and M. L. Gromov, «Minimizing finite state machines with time guards and timeouts», *Proceedings of the Institute for System Programming of the RAS*, vol. 29, no. 4, pp. 139–154, 2017.

[13] A. S. Tvardovskii and N. V. Yevtushenko, «Deriving homing sequences for Finite State Machines with timed guards», *Sistemnaya informatika*, vol. 17, pp. 1–10, 2020.

[14] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, «OpenFlow: enabling innovation in campus networks», *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[15] E. Vinarskii, J. Lopez, N. Kushik, N. Yevtushenko, and D. Zeghlache, «A Model Checking Based Approach for Detecting SDN Races», in *Proceedings of the 31-st IFIP WG 6.1 International Conference on Testing Software and Systems (ICTSS)*, Springer, 2019, pp. 194–211.

[16] E. M. Vinarskii and V. A. Zakharov, «On the verification of strictly deterministic behavior of Timed Finite State Machines», *Proceedings of ISP RAS*, vol. 30, no. 3, pp. 325–340, 2018.

[17] C. Baier and J.-P. Katoen, *Principles of model checking*. Cambridge: MIT Press Cambridge, 2008.

[18] G. Behrmann, A. David, and K. G. Larsen, «A tutorial on Uppaal», in *Proceedings of the International School on Formal Methods for the Design of Computer, Communication, and Software Systems*, Springer, 2004, pp. 200–236.

MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 27, NO. 4, 2020
journal homepage: www.mais-journal.ru

THEORY OF COMPUTING

Temporal Logic for Programmable Logic Controllers

N. O. Garanina[1], I. S. Anureev[1], V. E. Zyubin[2], S. M. Staroletov[3], T. V. Liakh[2], A. S. Rozov[2], S. P. Gorlatch[4]

DOI: 10.18255/1818-1015-2020-4-412-427

[1]A. P. Ershov Institute of Informatics Systems (IIS), Siberian Branch of the Russian Academy of Sciences, 6, Acad. Lavrentjev ave., Novosibirsk 630090, Russia.
[2]Institute of Automation and Electrometry SB RAS, 1 Academician Koptyug ave., Novosibirsk 630090, Russia.
[3] Polzunov Altai State Technical University, 46 Lenina ave., Barnaul 656038, Russia.
[4]University of Munster, 2 Schlossplatz, Munster 48149, Germany.

MSC2020: 68N30
Research article
Full text in Russian

Received November 12, 2020
After revision December 2, 2020
Accepted December 16, 2020

We address the formal verification of the control software of critical systems, i.e., ensuring the absence of design errors in a system with respect to requirements. Control systems are usually based on industrial controllers, also known as Programmable Logic Controllers (PLCs). A specific feature of a PLC is a scan cycle: 1) the inputs are read, 2) the PLC states change, and 3) the outputs are written. Therefore, in order to formally verify PLC, e.g., by model checking, it is necessary to describe the transition system taking into account this specificity and reason both in terms of state transitions within a cycle and in terms of larger state transitions according to the scan-cyclic semantics. We propose a formal PLC model as a hyperprocess transition system and temporal cycle-LTL logic based on LTL logic for formulating PLC property. A feature of the cycle-LTL logic is the possibility of viewing the scan cycle in two ways: as the effect of the environment (in particular, the control object) on the control system and as the effect of the control system on the environment. For both cases we introduce modified LTL temporal operators. We also define special modified LTL temporal operators to specify inside properties of scan cycles. We describe the translation of formulas of cycle-LTL into formulas of LTL, and prove its correctness. This implies the possibility of model checking requirements expressed in logic cycle-LTL, by using well-known model checking tools with LTL as specification logic, e.g., Spin. We give the illustrative examples of requirements expressed in the cycle-LTL logic.

Keywords: formal verification; temporal logics; transition systems; programmable logic controllers (PLC).

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.
Igor Sergeevich Anureev	orcid.org/0000-0001-9574-128X. E-mail: anureev@iis.nsk.su Ph.D. in Mathematics, senior research fellow.
Vladimir Evgenyevich Zyubin	orcid.org/0000-0002-8198-3197. E-mail: zyubin@iae.nsk.su Dr. Sci., head of laboratory.
Sergey Mikhailovich Staroletov	orcid.org/0000-0001-5183-9736. E-mail: serg_soft@mail.ru Ph.D. in Mathematics, lecturer.
Tatiana Victorovna Liakh	orcid.org/0000-0001-9148-946X. E-mail: liakh@iae.nsk.su research engineer.
Andrey Sergeevich Rozov	orcid.org/0000-0002-7468-0411. E-mail: rozov@iae.nsk.su junior researcher.
Sergei Petrovich Gorlatch	orcid.org/0000-0003-3857-9380. E-mail: gorlatch@uni-muenster.de Prof. Dr. Habil.

Funding: The reported study was funded by State assignment No AAAA-A19-119120290056-0.

For citation: N. O. Garanina, I. S. Anureev, V. E. Zyubin, S. M. Staroletov, T. V. Liakh, A. S. Rozov, and S. P. Gorlatch, "Temporal Logic for Programmable Logic Controllers", *Modeling and analysis of information systems*, vol. 27, no. 4, pp. 412-427, 2020.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 27, № 4, 2020
сайт журнала: www.mais-journal.ru

THEORY OF COMPUTING

Темпоральная логика для программируемых логических контроллеров

Н. О. Гаранина[1], И. С. Анureев[1], В. Е. Зюбин[2], С. М. Старолетов[3], Т. В. Лях[2], А. С. Розов[2], С. П. Горлач[4]

DOI: 10.18255/1818-1015-2020-4-412-427

[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]Алтайский государственный технический университет им. И.И. Ползунова, пр. Ленина, д. 46, г. Барнаул, 656038 Россия.
[4]Университет Мюнстера, Шлосплац, д. 2, Мюнстер, 48149 Германия.

УДК 004.822, 681.51
Научная статья
Полный текст на русском языке

Получена 12 ноября 2020 г.
После доработки 2 декабря 2020 г.
Принята к публикации 16 декабря 2020 г.

Мы исследуем формальную верификацию управляющего программного обеспечения критических систем, т. е. проверку соответствия функционирования проектируемой системы предъявленным требованиям. Важнейший класс управляющего программного обеспечения составляют программы для программируемых логических контроллеров (ПЛК). Особенностью программ ПЛК является цикл управления: 1) считываются входы, 2) изменяются состояния ПЛК и 3) записываются выходы. Поэтому для формальной верификации программ ПЛК нужна возможность описывать учитывающие эту специфику системы переходов, а также определять свойства систем, моделирующих программы ПЛК, как относительно переходов внутри цикла, так и относительно более крупных переходов в соответствии с семантикой цикла управления. Мы предлагаем формальную модель программы ПЛК как систему переходов гиперпроцессов и темпоральную логику cycle-LTL для формализации свойств ПЛК. Особенностью логики cycle-LTL является возможность рассматривать свойства систем управления двояким образом: воздействие окружения на систему управления и воздействие системы управления на окружение. Мы определяем модификации стандартных темпоральных операторов логики LTL для каждого из этих случаев, а также для свойств внутри цикла управления. Рассмотрены примеры требований, определенных в нашей логике. Описана трансляция формул cycle-LTL в формулы LTL и показана её корректность. Доказана возможность сведения задачи верификации методом проверки моделей для требований, определенных в логике cycle-LTL, к задаче верификации требований, определенных в стандартной логике LTL.

Ключевые слова: формальная верификация; темпоральная логика; системы переходов; программируемые логические контроллеры (ПЛК).

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..
Игорь Сергеевич Анureев	orcid.org/0000-0001-9574-128X. E-mail: anureev@iis.nsk.su канд. физ.-мат. наук, с.н.с..
Владимир Евгеньевич Зюбин	orcid.org/0000-0002-8198-3197. E-mail: zyubin@iae.nsk.su д.т.н., заведующий лабораторией.
Сергей Михайлович Старолетов	orcid.org/0000-0001-5183-9736. E-mail: serg_soft@mail.ru канд. физ.-мат. наук, доцент.
Татьяна Викторовна Лях	orcid.org/0000-0001-9148-946X. E-mail: liakh@iae.nsk.su инженер-исследователь.
Андрей Сергеевич Розов	orcid.org/0000-0002-7468-0411. E-mail: rozov@iae.nsk.su м.н.с.
Сергей Петрович Горлач	orcid.org/0000-0003-3857-9380. E-mail: gorlatch@uni-muenster.de профессор, канд. физ.-мат. наук.

Финансирование: Исследование выполнено в рамках государственного задания № АААА-А19-119120290056-0.

Для цитирования: N. O. Garanina, I. S. Anureev, V. E. Zyubin, S. M. Staroletov, T. V. Liakh, A. S. Rozov, and S. P. Gorlatch, “Temporal Logic for Programmable Logic Controllers”, *Modeling and analysis of information systems*, vol. 27, no. 4, pp. 412-427, 2020.

Введение

В последнее десятилетие, в связи с решением задач, предъявляющих повышенные требования по безопасности к системам управления на основе программируемых логических контроллеров (ПЛК), стали разрабатываться решения по формальному моделированию программ для таких контроллеров и доказательству их свойств, в частности, выражаемых в виде формул темпоральных логик.

Долгосрочной целью нашей работы является формальная верификация программ для систем автоматического управления, написанных в рамках процесс-ориентированной парадигмы, и в частности программ, написанных на процесс-ориентированном языке Reflex [1, 2]. Формальная проверка программ для ПЛК, которые являются важными компонентами систем автоматического управления, является актуальной темой теоретических и практических работ [3, 4]. Предлагаемые подходы следуют различным формальным моделям ПЛК [3—7]. В общем случае, функционирование ПЛК состоит из бесконечной последовательности *циклов управления*. Каждый цикл управления включает в себя последовательность из трех фаз: чтение входов, исполнение и запись выходов.

Мы используем процесс-ориентированное моделирование программ для ПЛК на основе *гиперпроцессов* [2]. Этот метод моделирования позволяет естественно определять основные особенности программ для ПЛК, такие как цикл управления и таймеры; он описывает программу для ПЛК как синхронизированную систему взаимодействующих процессов, определяемых набором функциональных состояний и действий в этих состояниях. Согласно классификации [5], гиперпроцесс моделирует программу для ПЛК, которая абстрагируется от времени исполнения цикла управления[1] и использует входное и выходное управление таймерами. Такое моделирование обеспечивает естественную спецификацию для многопроцессорных систем управления, которая за счёт абстрагирования от времени цикла управления позволяет строго упорядочивать последовательность исполнения процессов в цикле управления. Такая сильная синхронизация процессов без чередования позволяет эффективно использовать формальные методы верификации. Гиперпроцесс является основой для предметно-ориентированного языка Reflex, который показал свою продуктивность в ряде промышленных проектов, в частности, в установке для выращивания монокристаллов кремния методом Чохральского [8] и вакуумной системе для большого солнечного вакуумного телескопа [9].

В этой статье мы в качестве метода формальной верификации используем метод проверки моделей. Поэтому гиперпроцесс представлен как система переходов специального вида, а свойства гиперпроцесса — как формулы темпоральной логики. Система переходов гиперпроцессов близка к параллельной многопоточной системе [10], обогащённой синхронизирующим счётчиком, функциональными состояниями процессов, таймерами и примитивами действий для их изменения. Время для программы ПЛК, определённой как гиперпроцесс, тактируется не только внутри фазы исполнения цикла управления (с каждым изменением значений, характеризующих состояния процессов), но и более крупно – относительно последовательности циклов управления, т.е. при чтении входов/записи выходов.

Логика спецификации требований для программы ПЛК должна позволять формулировать высказывания относительно этих двух типов тактирования. В этой статье мы разрабатываем логику *cycle-LTL*, которая обогащает темпоральную логику LTL [11] цикловыми темпоральными операторами для анализа состояний программы ПЛК в начале и в конце цикла управления, а также внутрицикловыми темпоральными операторами для определения свойств программы ПЛК. В этой логике для некоторого абстрактного алгоритма управления можно выразить следующие свойства: “*Если сработал датчик, то устройство включится в следующем цикле управления*” или “*Если температура выше критического значения, то процесс охлаждения всегда включён.*” Заметим, что если

[1] Среда считается достаточно медленной, чтобы считать нулевым время фаз ввода/вывода и исполнения цикла управления.

включение и выключение процесса охлаждения выполняется после действий других процессов в фазе исполнения цикла управления, то последнее свойство может быть нарушенным внутри этой фазы, но может сохраняться вне её. Этот пример иллюстрирует необходимость использования цикловых темпоральных операторов, в частности, оператора “всегда” $\mathbf{G}^{\mathbf{i}}$.

Приведём обзор формализмов, используемых в современных работах по теме исследования. Вначале было разработано расширение временного автомата – PLC-автомат [12] для описания поведения программ для ПЛК в автоматной форме, и далее работы, в основном, концентрировались на применении логических систем, а не автоматов.

В работе [13] сделана оценка методов верификации программ для ПЛК как в текстовой, так и в графической форме с использованием различных инструментов проверки моделей, и приводятся конкретные типы реальных программ. Также описываются текущие проблемы в этой области, в частности, проблема комбинаторного взрыва, моделирование таймеров, определение свойств для проверки и проблема моделирования цикла исполнения таких программ, которой и посвящена настоящая работа.

Известны методы представления программ ПЛК в виде LTL формул. Согласно подходу [14], значение каждой переменной должно изменяться один раз только в одном месте программы за одно полное выполнение при прохождении рабочего цикла ПЛК. Поэтому изменение значения каждой программной переменной представлено двумя явными и одной неявной LTL-формулами. Таким образом, программирование ПЛК предлагается сводить к построению LTL-спецификации для каждой программной переменной. В этом случае доказывается полнота по Тьюрингу языков стандарта IEC 61131-3 с использованием машин Минского.

В работе [15] представлены способы автоматического майнинга инвариантов с помощью причинно-следственного графа связей событий по работающей программе для ПЛК с целью получения TPTL [16] формул для свойств безопасности. Даётся ремарка, что во время выполнения трудно обеспечить соблюдение правила на основе LTL формулы, которое требует, чтобы за одним действием следовало другое, поскольку отсутствие требуемого события в течение ограниченного времени тестирования не предполагает его отсутствия в более позднее время.

В [17] предложен способ автоматического получения спецификаций по заданным шаблонам LTL формул путём прогона циклов исполнения IEC 61131-3 программ. В работе [18] осуществлён перевод кода ПЛК на входной язык верификатора SMV для проверки моделей согласно выявленным инвариантам.

Разработка предметно-ориентированных расширений темпоральных логик для спецификации требований к промышленным контроллерам началась в последнее время. Обзор по этой теме дан в [19], где, в частности, предложена логика ST-LTL [20]. Основные улучшения по сравнению с LTL – это использование предыдущих значений переменных вместо оператора следующего состояния, а также введение оператора *WhileNScanCycles* для работы со значениями управляющих переменных на N-м шаге, что существенно отличается от нашего подхода и приводит к более сложным спецификациям и доказательствам. В [21] вводится темпоральная логика реального времени MITL[a,b], где все темпоральные модальности ограничены интервалом времени [a,b] и далее логика STL (сигнальная темпоральная логика), основанная на фильтрации сигналов и перехода из действительных чисел в натуральные.

Таким образом, наш подход соответствует современным тенденциям развития подходов к формальной спецификации и верификации программ, которые концентрируются на расширении существующих логик в контексте моделирования цикла управления. Первая версия нашего подхода представлена в работе [22].

Оставшаяся часть статьи организована следующим образом. В разделе 1 мы даём определение системы переходов гиперпроцессов. В разделе 2 описаны синтаксис и семантика темпоральной

логики cycle-LTL. В разделе 3 показано, что любую формулу cycle-LTL можно выразить в языке темпоральной логики LTL, и что задача проверки моделей для cycle-LTL и системы переходов гиперпроцессов разрешима. Раздел 3 содержит заключение и план будущих работ.

1. Система переходов гиперпроцессов

Дадим неформальное описание гиперпроцесса [2]. *Гиперпроцесс* — это упорядоченный набор взаимодействующих процессов, которые выполняются последовательно в заданном порядке, образуя *цикл управления*. Этот цикл начинается с чтения входных данных из среды в системные входные переменные гиперпроцесса и заканчивается записью выходных данных (управляющих воздействий) в соответствующие выходные переменные. Значения выходных переменных вырабатываются в фазе исполнения. Мы считаем, что изменения в данных из окружения происходят достаточно медленно, чтобы все процессы успели сработать в фазе исполнения. Таким образом, один цикл управления можно рассматривать как логическую единицу времени гиперпроцесса. Все переменные гиперпроцесса являются глобальными. Особенностью процессов, образующих гиперпроцесс, являются *функциональные состояния* — метки, обозначающие определённую последовательность действий процесса. Среди функциональных состояний выделяются состояния нормального останова и ошибочного останова, в которых процесс не совершает никаких действий. Действия процессов могут изменять значения переменных гиперпроцесса (кроме входных переменных), ограниченно влиять на функциональные состояния других процессов, а также устанавливать и сбрасывать значения таймера. Действия могут иметь охранные условия, зависящие от переменных гиперпроцесса и функциональных состояний других процессов. Наше определение системы переходов гиперпроцессов основано на описании гиперпроцессов и операционной семантике языка Reflex [1, 2].

Определение 1. *(Система переходов гиперпроцессов, HTS)*
Система переходов гиперпроцессов это пятёрка $H = (P, S, s_{ini}, A, R)$, где

- $P = \{p_1, ..., p_n\}$ — упорядоченное множество процессов;
- S — непустое множество состояний;
- s_{ini} — начальное состояние;
- A — алфавит действий;
- R — отношение помеченных переходов $R : A \longrightarrow 2^{S \times S}$.

Перед определением компонент HTS, опишем элементы гиперпроцессов в целом.

Определение 2. *(Элементы гиперпроцессов)*
Элементами гипероцессов являются переменные, функциональные состояния, действия процессов и таймеры:

Переменные. $V = \{v_1, \dots, v_N\}$ — множество *переменных гиперпроцесса*. Значения переменных в состоянии гиперпроцесса определяются соответствующими функциями $v_i : S \longrightarrow D \cup \{\perp\}$. Мы различаем *входные переменные* V_I, *выходные переменные* V_O и *внутренние переменные* V_L: $V = V_I \cup V_O \cup V_L$.

Функциональные состояния. Для каждого $i \in [1..n]$ множеством *функциональных состояний процесса* p_i является $F_i = \{f_i^1, ..., f_i^{m_i}, stop, err\}$. Выделяются *неактивные* состояния $stop$ и err. Остальные состояния являются *активными*. Значение переменной функционального состояния f_i для каждого процесса p_i в состоянии гиперпроцесса описывается функцией $f_i : S \longrightarrow F_i$.

Действия. В активном функциональном состоянии f_i^j, процесс p_i выполняет действия из множества A. Эти действия формируют *тело функционального состояния*. $L_i^j \in \mathbb{N}$ – это число действий в теле f_i^j. Для каждого процесса a_i – это *счётчик действий*, а его значение – это *позиция*. Значение счётчика действий в состоянии гиперпроцесса является результатом функции $a_i : S \longrightarrow [1..L_i^j] \cup \{\perp\}$. Следующее действие процесса в функциональном состоянии определяется функциями $nxt^j : \mathbb{N} \times S \longrightarrow \mathbb{N} \cup \{\perp\}$. Эти функции неявно задают охранные условия для

действий, поскольку зависят от текущего состояния гиперпроцесса, в частности, от состояния активности других процессов. Если в текущем функциональном состоянии нет следующего действия, т.е. достигнут конец его тела, то $nxt^j(a_i) = \bot$. Функции $an^j : \mathbb{N} \cup \{\bot\} \longrightarrow A$ возвращают имя действия на позиции a_i.

Таймеры. *Таймер* процесса p_i – это переменная t_i, чьи значения в состоянии гиперпроцесса являются результатом функции $t_i : S \longrightarrow \mathbb{N} \cup \{\bot\}$. *Значение таймаута* на позиции a функционального состояния f_i^j – это $N_i^{j_a} \in \mathbb{N}$. Для простоты изложения, здесь мы рассматриваем только процессы, у которых не более одного запуска таймера на каждое функциональное состояние.

Определим компоненты системы переходов гиперпроцессов $H = (P, S, s_{ini}, A, R)$.

Множество состояний S

Состояние $s = (v, sp, pc) \in S$ включает следующие элементы:

- оценка переменных $v = (v_1(s), \dots, v_N(s))$: вектор значений переменных в состоянии s;
- состояние процессов $sp = ((f_1(s), a_1(s), t_1(s)), \dots, (f_n(s), a_n(s), t_n(s)))$, где для каждого процесса p_i в состоянии s, $f_i(s)$ — его текущее функциональное состояние, $a_i(s)$ — счётчик действий в состоянии $f_i(s)$, и $t_i(s)$ — значение его таймера;
- счётчик процессов $pc(s)$ со значениями в $[0..n + 1]$, где 0 зарезервирован для чтения входных данных, а $n + 1$ – для записи выходных данных.

Начальное состояние s_{ini}

$s_{ini} = (v_0, sp_0, pc_0)$, где

- $v_0 = (\bot_1, \dots, \bot_N)$,
- $sp_0 = ((f_1^1, 1, \bot)_1, (stop, \bot, \bot)_2, \dots, (stop, \bot, \bot)_n)$, и
- $pc_0 = 0$.

Алфавит действий A

Алфавит включает действия обновления входных и выходных данных, а также действия процессов: $A = \{inp, out, skip, end, upd, tout, reset, startP, stopP, start, set, next, stop, err\}$.

0. Цикловые действия.

inp – изменение значений входных переменных (чтение входов из окружения).

out – изменение значений выходных переменных (запись выходов в окружение).

1. Служебные действия.

$skip$ – процесс ничего не делает в неактивных состояниях.

end – по завершении действий процесса в активном состоянии, ход передаётся следующему процессу.

2. Действие обновления значений внутренних переменных.

upd – процесс изменяет значения некоторых внутренних переменных.

3. Действия с таймером.

$tout$ – процесс запускает таймер и выполняет действия в текущем состоянии до наступления таймаута;

$reset$ – процесс переустанавливает свой таймер в ноль.

4. Действия с функциональными состояниями.

$startP/stopP$ – процесс переводит целевой процесс p_k в функциональные состояния $f_k^1/stop$;

$start/set$ – процесс переходит в целевое функциональное состояние f_i^1/f;

$stop/err$ – процесс переходит в функциональное состояние $stop/err$;

$next$ – процесс переходит к следующему функциональному состоянию.

Отношение помеченных переходов R

Отношение переходов R задаёт семантику действий процессов и обновления входов. Пусть $i \in [1..n], j \in [1..m_i]$, $a \in [1..L_i^j]$. Мы будем использовать следующую нотацию для действий процессов.

Выражение

$$(f_i = f_i^j, a_i = a, T_i, Tg_i, pc = i) \xrightarrow{act} (y'_1 = new_1, \dots, y'_{m'} = new_{m'})$$

означает, что для всех состояний s, удовлетворяющих предусловию (выражению слева от стрелки), и всех состояний s', удовлетворяющих постусловию (выражению справа от стрелки), верно $R(act) = (s, s')$, и при этом

- $act = an^j(a)$, и $an^j(\perp) \in \{skip, end\}$;
- стартовое состояние s таково, что $pc(s) = i$, $f_i(s) = f_i^j$, $a_i(s) = a$, временное ограничение T_i может быть $t_i(s) \neq \perp$, $t_i(s) = \perp$, $t_i(s) < N_i^j$, или $t_i(s) = N_i^j$, а целевое ограничение Tg_i может быть $tgp_i = k$ или $tgs_i = k$, где tgp_i — это переменная из V_L со значениями в $[1..n]$ для задания целевого процесса, а tgs_i — это переменная в V_L со значениями в $[1..m_i]$ для задания целевого функционального состояния процесса p_i; неупомянутые слева элементы имеют произвольные значения;
- результирующее состояние s' специфицирует изменения элементов гиперпроцесса после действия act: $y_k(s') = new_k$ ($k \in [1..m']$); неупомянутые справа элементы не изменяются.

0. Действия обновления входных и выходных переменных.

Мы используем аналогичную нотацию для определения $R(inp)$ и $R(out)$.

В начале цикла управления значения входных данных считываются во входные переменные из V_I, значение каждого активированного таймера процесса увеличивается на 1, и счётчик процессов смещается к первому процессу:

– $(t_{i_1} \neq \perp, \dots, t_{i_k} \neq \perp, pc = 0) \xrightarrow{inp} (u'_1 = u_1, \dots, u'_K = u_K, t'_{i_1} = t_{i_1} + 1, \dots, t_{i_k} = t_{i_k} + 1, pc' = 1)$.

Действие чтения входных данных — единственное действие в системе, которое, вообще говоря, недетерминировано.

В конце цикла управления значения выходных данных записываются в выходные переменные из V_O, и счётчик процессов смещается на начало цикла управления:

– $(pc = n + 1) \xrightarrow{out} (w'_1 = w_1, \dots, w'_M = w_M, pc' = 0)$

Это действие детерминировано.

В определении отношения переходов R для действий процессов, мы рассматриваем процесс p_i, который выполняет действие номер a_i с именем $an^j(a_i)$ в активном функциональном состоянии $f_i = f_i^j$, или находится в неактивном состоянии.

1. Служебные действия.

Процесс p_i ничего не делает в неактивных состояниях $stop$ и err, и ход передаётся следующему процессу:

– $(f_i = stop, pc = i) \xrightarrow{skip} (pc' = i + 1)$.

– $(f_i = err, pc = i) \xrightarrow{skip} (pc' = i + 1)$.

Когда процесс p_i достигает конца тела активного функционального состояния f_i^j, он переходит к первому действию этого состояния, которое он будет выполнять в следующем цикле управления (если только другой процесс не переведёт его в иное состояние), и ход передаётся следующему процессу:

– $(f_i = f_i^j, a_i = \perp, pc = i) \xrightarrow{end} (a'_i = 1, pc' = i + 1)$.

2. Действие обновления значений переменных процесса.

С помощью этого действия, процесс p_i изменяет значения некоторых внутренних переменных $\{v_1, \dots, v_m\} \subseteq V_L$, и переходит к следующему действию $nxt^j(a)$ текущего функционального состояния:

$-\ (f_i = f_i^j, pc = i) \xrightarrow{upd} (v_1' = d_1, \dots, v_m' = d_m, a_i' = nxt^j(a))$;

Это действие детерминировано.

3. Действия с таймером.

В следующих случаях, процесс p_i запускает таймер t_i (если $t_i = \bot$), переходит к первому действию текущего функционального состояния f_i^j, и ход передаётся следующему процессу:

$-\ (f_i = f_i^j, t_i = \bot, pc = i) \xrightarrow{tout} (a_i' = 1, t_i' = 0, pc' = i + 1)$;

$-\ (f_i = f_i^j, t_i < N_i^j, pc = i) \xrightarrow{tout} (a_i' = 1, pc' = i + 1)$.

В случае наступления события таймаут, процесс p_i останавливает таймер t_i и переходит к следующим действиям текущего функционального состояния:

$-\ (f_i = f_i^j, t_i = N_i^j, pc = i) \xrightarrow{tout} (a_i' = nxt^j(a), t_i' = \bot)$.

Результат действия $reset$ аналогичен результату первого случая действия $tout$:

$-\ (f_i = f_i^j, t_i \neq \bot, pc = i) \xrightarrow{reset} (a_i' = 1, t_i' = 0, pc' = i + 1))$.

4. Действия с функциональными состояниями.

Следующие два действия процесса p_i переводят процесс p_k ($i \neq k$) в состояние $stop$, err или его первое функциональное состояние. Заметим, что модель гиперпроцесса удовлетворяет принципу ограниченной инкапсуляции: процесс не может перевести другой процесс в произвольное функциональное состояние.

$-\ (f_i = f_i^j, tgp_i = k, pc = i) \xrightarrow{startP} (f_k' = f_k^1, a_k' = 1, t_k' = \bot, a_i' = nxt^j(a))$;

$-\ (f_i = f_i^j, tgp_i = k, pc = i) \xrightarrow{stopP} (f_k' = stop, a_k' = \bot, t_k' = \bot, a_i' = nxt^j(a))$;

$-\ (f_i = f_i^j, tgp_i = k, pc = i) \xrightarrow{errP} (f_k' = err, a_k' = \bot, t_k' = \bot, a_i' = nxt^j(a))$;

С помощью следующих действий, процесс p_i устанавливает, что в следующем цикле управления он начнёт работу с первого действия соответствующего функционального состояния, и останавливает таймер:

$-\ (f_i = f_i^j, tgs_i = k, pc = i) \xrightarrow{set} (f_i' = f_i^k, a_i' = 1, t_i' = \bot)$;

$-\ (f_i = f_i^j, pc = i) \xrightarrow{start} (f_i' = f^1, a_i' = 1, t_i' = \bot)$;

$-\ (f_i = f_i^j, pc = i) \xrightarrow{next} (f_i' = f_i^{j+1}, a_i' = 1, t_i' = \bot)$;

Процесс p_i переходит в состояние нормального или ошибочного останова:

$-\ (f_i = f_i^j, pc = i) \xrightarrow{stop} (f_i' = stop, a_i' = \bot, t_i' = \bot)$;

$-\ (f_i = f_i^j, pc = i) \xrightarrow{err} (f_i' = err, a_i' = \bot, t_i' = \bot)$.

Согласно определению отношения переходов, процессы действуют последовательно в текущем цикле управления в порядке, заданном счётчиком процессов. Отметим, что отношение переходов недетерминировано только для действия чтения входных данных inp, которое не является действием какого-либо процесса. Пусть *выходные состояния* s_{out} — это состояния, в которых $pc(s_{out}) = 0$. В этих состояниях выходные переменные получили новые значения. Таким образом, выходные состояния отражают зависимости выходных данных (реакций системы управления) от входных данных (от объекта управления). Определим *входные состояния* s_{inp} как состояния сразу после чтения входов во входные переменные и перед тем, как процессы начали действовать: $R(inp) = (s_{out}, s_{inp})$. Таким образом, входные состояния отражают возможные зависимости входных данных (реакций объекта управления) от выходных данных системы управления на предыдущем цикле управления. В силу детерминизма действий процессов, по заданному входному состоянию однозначно определяет-

ся ближайшее выходное состояние, однако различные входы всё же могут приводить к одному и тому же выходу: например, определённый диапазон входных значений может обрабатываться одинаково. Пусть S_i – это множество входных состояний, а S_o – это множество выходных состояний.

Определим три вида путей в HTS. *Стандартный путь* $\pi = s_0, s_1, \ldots$ – это последовательность состояний $s_j \in S$ такая, что $\forall j \geq 0\ \exists a \in A : R(a) = (s_j, s_{j+1})$. Пусть $\pi(j)$ — это j-е состояние на пути π. *Входной путь* $\sigma = i_0, i_1, \ldots$ – это бесконечная последовательность входных состояний $i_j \in S_i$ такая, что для каждого $j \geq 0$ существует конечный стандартный путь π_j длины n_j с $\pi_j(0) = i_j$ и $\pi_j(n_j) = i_{j+1}$. *Выходной путь* $\rho = o_0, o_1, \ldots$ – это бесконечная последовательность выходных состояний $o_j \in S_o$ такая, что для каждого $j \geq 0$ существует конечный стандартный путь π_j длины n_j с $\pi_j(0) = o_j$ и $\pi_j(n_j) = o_{j+1}$. Произвольный путь $\chi = x_0, x_1, \ldots$ является *подпутём* пути $\chi' = x'_0, x'_1, \ldots$, если $x_0, x_1 \ldots$ является подпоследовательностью $x'_0, x'_1, \ldots$. Соответственно, χ' – это *надпуть* χ. Если путь χ проходит в H, то пишем $\chi \in H$.

Далее в статье χ — это стандартный, входной или выходной путь, π — стандартный путь, ρ — входной путь, σ — выходной путь. Дадим определение дополнительных путей, которые используются при задании семантики темпоральных операторов cycle-LTL.

Определение 3. *(Пути и их элементы)*

1. $\chi(k)$ — k-е состояние на пути χ;
2. χ^k — путь-суффикс χ, такой что $\chi^k(0) = \chi(k)$;
3. i^π_k — номер k-го входного состояния на пути π;
4. o^π_k — номер k-го выходного состояния на пути π;
5. π_ρ – это путь, содержащий заданный путь ρ так, что их начала совпадают, т.е. ρ подпуть π и $\pi_\rho(0) = \rho(0)$ (этот путь единственный, т.к. процессы обрабатывают входные данные детерминировано);
6. π_σ – это пути, содержащие заданный путь σ так, что их начала совпадают, т.е. σ подпуть π и $\pi_\sigma(0) = \sigma(0)$;
7. ρ^π — это входные пути, которые начинаются раньше заданного π, но содержатся в нём, начиная со своего второго состояния, т.е. $\pi = \pi^k_{\rho^\pi}$, где k такое, что $\rho^\pi(1) = \pi(i^\pi_1)$;
8. ρ^σ — это входные пути, надпути которых содержит заданный путь σ с ближайшего выходного состояния, т.е σ подпуть π_{ρ^σ} и $\sigma(0) = \pi_{\rho^\sigma}(o_1^{\pi_{\rho^\sigma}})$;
9. σ^π — это выходной путь, который начинается позже заданного π, но содержится в нём, и его начало – это первое выходное состояние π, т.е. $\pi_{\sigma^\pi} = \pi^k$, где k такое, что $\sigma^\pi(0) = \pi(o^\pi_1)$ (этот путь единственный, т.к. все его состояния лежат на заданном пути π);
10. σ^ρ — это выходной путь, надпуть которого содержит заданный путь ρ с ближайшего предшествующего входного состояния, т.е. σ подпуть π_ρ и $\sigma(0) = \pi_\rho(o_1^{\pi_\rho})$ (этот путь единственный, т.к. процессы для заданных входных данных вырабатывают детерминированный выход).

Рисунок 1 иллюстрирует понятия определения 3.

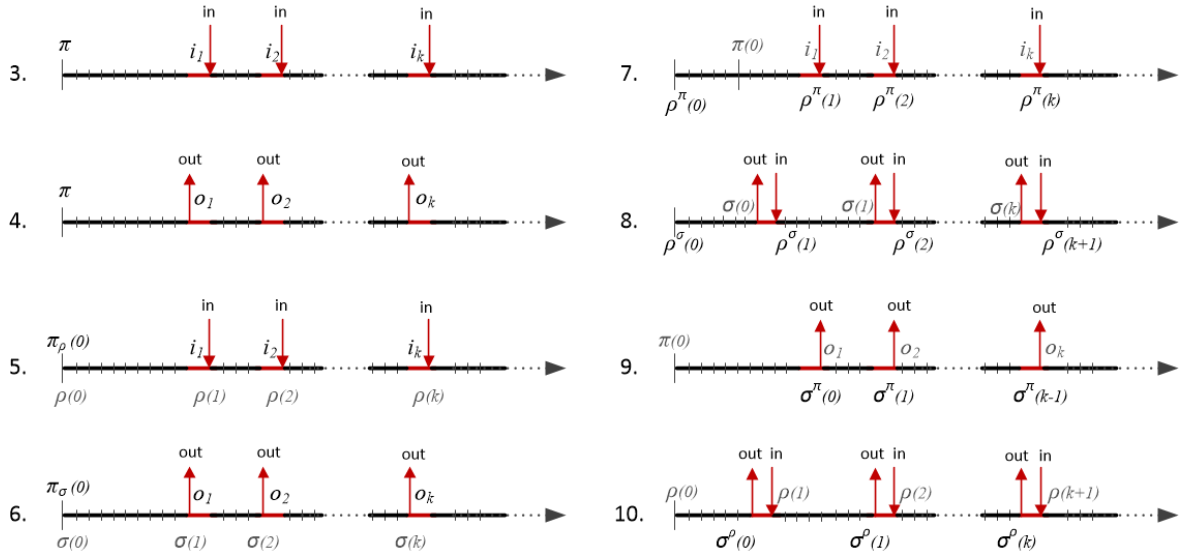


Fig. 1. Paths and their elements

Рис. 1. Пути и их элементы

2. Темпоральная логика cycle-LTL

Синтаксис логики cycle-LTL содержит атомарные высказывания P, булевы связки, стандартные темпоральные операторы LTL, входные, внутренние и выходные цикловые темпоральные операторы:

$$\varphi ::= P \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathbf{X}\varphi \mid \mathbf{F}\varphi \mid \mathbf{G}\varphi \mid \varphi\mathbf{U}\varphi \mid$$
$$\mathbf{X^i}\varphi \mid \mathbf{F^i}\varphi \mid \mathbf{G^i}\varphi \mid \varphi\mathbf{U^i}\varphi \mid \mathbf{X^e}\varphi \mid \mathbf{F^e}\varphi \mid \mathbf{G^e}\varphi \mid \varphi\mathbf{U^e}\varphi \mid \mathbf{X^o}\varphi \mid \mathbf{F^o}\varphi \mid \mathbf{G^o}\varphi \mid \varphi\mathbf{U^o}\varphi$$

Внутренние цикловые темпоральные операторы $\mathbf{X^e}$, $\mathbf{F^e}$, $\mathbf{G^e}$, и $\mathbf{U^e}$ используются для формулирования свойств, которые должны выполняться в течение фазы исполнения циклов управления, тогда как с помощью *входных и выходных цикловых операторов* $\mathbf{X^i}$, $\mathbf{F^i}$, $\mathbf{G^i}$, $\mathbf{U^i}$, $\mathbf{X^o}$, $\mathbf{F^o}$, $\mathbf{G^o}$, и $\mathbf{U^o}$ можно описывать свойства систем управления, которые выполняются в начале и в конце циклов управления.

Пусть Φ^s — это множество формул, начинающихся со стандартных LTL-операторов, Φ^e — множество формул, начинающихся с внутренних операторов, Φ^i — множество формул, начинающихся с входных операторов, Φ^o — множество формул, начинающихся с выходных операторов, и Φ^c — множество всех формул cycle-LTL.

Семантику логики cycle-LTL мы традиционно определяем в терминах отношения выполнимости $\models$ между формулой и путём в HTS: $H, \chi \models \xi$, где H — это система переходов гиперпроцессов, χ — бесконечный стандартный, входной или выходной путь, и $\xi \in \Phi^c$. Мы рассматриваем все виды путей для формул с цикловыми темпоральными операторами, и входные/выходные пути для стандартных LTL-формул. Семантику стандартных LTL-формул на стандартных путях можно посмотреть в [11]. Пусть H — это система переходов гиперпроцессов, π — бесконечный стандартный путь, ρ — входной путь, σ — выходной путь, и $\varphi, \psi \in \Phi^c$ — формулы cycle-LTL.

Семантика формул Φ^s.
Пусть $\xi \in \Phi^s$.

- $H, \rho \models \xi \iff H, \pi_\rho \models \xi$;
- $H, \sigma \models \xi \iff$ для всех π_σ верно $H, \pi_\sigma \models \xi$;

Семантика формул Φ^e.
Пусть $\xi \in \Phi^e$.

- $H, \rho \models \xi \iff H, \pi_\rho \models \xi$;
- $H, \sigma \models \xi \iff$ для всех π_σ верно $H, \pi_\sigma \models \xi$;
- $H, \pi \models \mathbf{X^e}\varphi \iff \pi(1) \notin S_i \cup S_o$ и $H, \pi^1 \models \varphi$;
- $H, \pi \models \mathbf{F^e}\varphi \iff$ существует $0 \le k < o_1^\pi$ такое, что $\pi(k) \notin S_i \cup S_o$ и $H, \pi^k \models \varphi$;
- $H, \pi \models \mathbf{G^e}\varphi \iff$ для всех $0 \le k < o_1^\pi$ $\pi(k) \notin S_i \cup S_o$ и $H, \pi^k \models \varphi$;
- $H, \pi \models \varphi \mathbf{U^e} \psi \iff$ существует $0 \le k < o_1^\pi$ такое что $\pi(k) \notin S_i \cup S_o$ и $H, \pi^k \models \psi$ и для всех $0 \le j < k$ $\pi(j) \notin S_i \cup S_o$ и $H, \pi^j \models \varphi$.

Семантика формул Φ^i.
Пусть $\xi \in \Phi^i$.

- $H, \pi \models \xi \iff$ для всех ρ^π верно $H, \rho^\pi \models \xi$;
- $H, \sigma \models \xi \iff$ для всех ρ^σ верно $H, \rho^\sigma \models \xi$;
- $H, \rho \models \mathbf{X^i}\varphi \iff H, \rho^1 \models \varphi$;
- $H, \rho \models \mathbf{F^i}\varphi \iff$ существует $k \ge 0$ такое, что $H, \rho^k \models \varphi$;
- $H, \rho \models \mathbf{G^i}\varphi \iff$ для всех $k \ge 0$ $H, \rho^k \models \varphi$;
- $H, \rho \models \varphi \mathbf{U^i} \psi \iff$ существует $k \ge 0$ такое что $H, \rho^k \models \psi$ и для всех $0 \le j < k$ $H, \rho^j \models \varphi$.

Семантика формул Φ^o.
Пусть $\xi \in \Phi^o$.

- $H, \pi \models \xi \iff H, \sigma^\pi \models \xi$;
- $H, \rho \models \xi \iff H, \sigma^\rho \models \xi$;
- $H, \sigma \models \mathbf{X^o}\varphi \iff H, \sigma^1 \models \varphi$;
- $H, \sigma \models \mathbf{F^o}\varphi \iff$ существует $k \ge 0$ такое, что $H, \sigma^k \models \varphi$;
- $H, \sigma \models \mathbf{G^o}\varphi \iff$ для всех $k \ge 0$ $H, \sigma^k \models \varphi$;
- $H, \sigma \models \varphi \mathbf{U^o} \psi \iff$ существует $k \ge 0$ такое что $H, \sigma^k \models \psi$ и для всех $0 \le j < k$ $H, \sigma^j \models \varphi$.

Проиллюстрируем спецификации на языке cycle-LTL для систем управления на примере тепловентилятора и устройства поддержания температуры:

1. *Если руки появились под сушилкой, она будет включена в ближайшем цикле управления*:
   $\mathbf{G^i}(hands = on \rightarrow \mathbf{X^i} dryer = on)$.
2. *Если температура выше максимального значения, то процесс охлаждения всегда включён*:
   $\mathbf{G^i}(temp > 95^\circ \rightarrow cooler = on)$.
3. *Нагревательный элемент включён, если температура опустилась ниже минимального значения*:
   $\mathbf{G^o}(temp < 5^\circ \rightarrow heater = on)$.
4. *Если включён нагревательный элемент, то со временем температура поднимется выше минимального значения*:
   $\mathbf{G^o}(heater = on \rightarrow \mathbf{F^i}\ temp \ge 5^\circ)$.

Отметим вариативность синтаксиса cycle-LTL при описании свойств гиперпроцессов. В формулах примеров 1 и 3 можно использовать как оператор $\mathbf{G^i}$ (в комплекте с $\mathbf{X^i}$), так и $\mathbf{G^o}$: в обоих случаях задаются свойства зависимости значений ближайшего выхода от входных данных. Выбор того или иного представления свойства системы зависит от предпочтений пользователя.

Вышеприведённые примеры свойств систем используют только циклические состояния и наблюдаемые входные-выходные переменные системы. Такого рода формулы могут использоваться для задания высокоуровневых свойств систем и алгоритмов управления, независимых от реализации, когда система управления рассматривается как чёрный ящик. Однако, если для некоторой реализации системы управления не выполнилось высокоуровневое свойство, то для дальнейшего анализа может понадобиться проверить низкоуровневые свойства системы, сформулированные в терминах состояний и действий процессов, её реализующих. Эти свойства системы определяются

в виде гипотез, использующих внутрицикловые операторы логики, что позволяет локализовать ошибку внутри исполнительной фазы цикла управления. В ряде случаев проверка таких гипотез менее трудозатратна, чем анализ контрпримера для высокоуровневого свойства. Рассмотрим систему, комбинирующую систему управления освещением и охранную сигнализацию. Следующие два свойства этой системы иллюстрируют причинно-следственную связь между низкоуровневыми гипотезами и высокоуровневыми свойствами: невыполнение низкоуровневой формулы 2 влечёт невыполнение высокоуровневой формулы 1.

1. *Когда обнаружен взлом, все лампы мигают в аварийном режиме*:
   $\mathbf{G^i}(alarm \rightarrow \mathbf{X^i} alarm\_light = on)$.
2. *Если сработал датчик взлома, то процесс внешнего взаимодействия охранной сигнализации в течение ближайшего цикла управления посылает сигнальное сообщение всем связанным системам, включая систему управления освещением*:
   $\mathbf{G^i}(alarm \rightarrow \mathbf{F^e} alarm\_message\_sent)$.

Выбор гипотезы 2 обусловлен предположением, что в случае отправки сигнального сообщения, оно будет вовремя получено и после получения сразу сработает включение аварийного режима работы ламп.

3. Перевод формул логики cycle-LTL в LTL

Мы считаем, что формулы логики cycle-LTL $\xi, \xi' \in \Phi^c$ *эквивалентны* ($\xi \equiv \xi'$), если и только если $H, \chi \vDash \xi \iff H, \chi \vDash \xi'$ для произвольной HTS H и произвольного пути χ. В этом разделе мы покажем, что для формул с цикловыми операторами $\Phi^e \cup \Phi^i \cup \Phi^o$ существуют эквивалентные им стандартные формулы LTL с дополнительными атомарными высказываниями.

Добавим в описание состояний HTS H две служебных булевых переменных $Input$ и $Output$. Пусть переменная $Input$ истинна только во входных состояниях из S_i, а переменная $Output$ истинна только в выходных состояниях из S_o. Тогда для соответствующих атомарных высказываний верно, что $H, \pi \vDash Input$ для всех путей π, таких что $\pi(0) \in S_i$, и $H, \pi \nvDash Input$ для всех путей π, таких что $\pi(0) \notin S_i$, а также $H, \pi \vDash Output$ для всех путей π, таких что $\pi(0) \in S_o$, и $H, \pi \nvDash Output$ для всех путей π, таких что $\pi(0) \notin S_o$. Пусть H — это система переходов гиперпроцессов, π — бесконечный стандартный путь, ρ — входной путь, σ — выходной путь, и $Exe = \neg(Input \vee Output)$. Поскольку доказательства утверждений ниже проводятся индукцией по структуре формулы, то можно считать, что далее подформулы формул с цикловыми операторами φ и ψ являются формулами LTL.

Утверждение 1. *cycle-LTL-формулы с начальными внутренними цикловыми операторами эквивалентны следующим LTL-формулам:*

- $\mathbf{X^e}\varphi \equiv \mathbf{X}(\varphi \wedge Exe)$;
- $\mathbf{F^e}\varphi \equiv (Exe)\mathbf{U}(\varphi \wedge Exe)$;
- $\mathbf{G^e}\varphi \equiv (\varphi \wedge Exe)\mathbf{U}(\neg Exe)$;
- $\varphi \mathbf{U^e} \psi \equiv (\varphi \wedge Exe)\mathbf{U}(\psi \wedge Exe)$.

Доказательство.

Проводится индукцией по структуре формулы. Рассмотрим сначала стандартные пути. Отметим, что $\pi(k) \notin S_i \cup S_o \iff H, \pi^k \vDash \neg(Input \vee Output) \iff H, \pi^k \vDash Exe$.

- $H, \pi \vDash \mathbf{X^e}\varphi \iff (\pi(1) \notin S_i \cup S_o) \bigwedge (H, \pi^1 \vDash \varphi) \iff$
  $(H, \pi^1 \vDash Exe) \bigwedge (H, \pi^1 \vDash \varphi) \iff H, \pi^1 \vDash \varphi \wedge Exe \iff H, \pi \vDash \mathbf{X}(\varphi \wedge Exe)$;
- $H, \pi \vDash \mathbf{F^e}\varphi \iff \exists\, 0 \le k < o_1^\pi : (\pi(k) \notin S_i \cup S_o) \bigwedge (H, \pi^k \vDash \varphi) \iff$
  $\exists\, 0 \le k < o_1^\pi : (H, \pi^k \vDash Exe) \bigwedge (H, \pi^k \vDash \varphi) \overset{def.o_1^\pi}{\iff}$
  $\exists\, 0 \le k < o_1^\pi : (H, \pi^k \vDash \varphi \wedge Exe) \bigwedge \forall\, 0 \le j \le k\ (H, \pi^j \vDash Exe) \iff H, \pi \vDash (Exe)\mathbf{U}(\varphi \wedge Exe)$;
- $H, \pi \vDash \mathbf{G^e}\varphi \iff \forall\, 0 \le k < o_1^\pi : (\pi(k) \notin S_i \cup S_o) \bigwedge (H, \pi^k \vDash \varphi) \iff$
  $\forall\, 0 \le k < o_1^\pi : (H, \pi^k \vDash Exe) \bigwedge (H, \pi^k \vDash \varphi) \bigwedge (H, \pi^{o_1^\pi} \vDash Output) \iff H, \pi \vDash (\varphi \wedge Exe)\mathbf{U}(\neg Exe)$;

- $H, \pi \vDash \varphi \mathbf{U^e} \psi \iff \exists\, 0 \le k < o_1^\pi : (\pi(k) \notin S_i \cup S_o) \bigwedge (H, \pi^k \vDash \psi) \bigwedge$
$\forall\, 0 \le j < k : (\pi(j) \notin S_i \cup S_o) \bigwedge (H, \pi^j \vDash \varphi) \iff$
$\exists\, 0 \le k < o_1^\pi : (H, \pi^k \vDash \psi \wedge Exe) \bigwedge \forall\, 0 \le j < k : (H, \pi^j \vDash \varphi \wedge Exe) \iff H, \pi \vDash (\varphi \wedge Exe)\mathbf{U}(\psi \wedge Exe)$.

Эквивалентность формул Φ^e и вышеприведённым формулам из Φ^s на стандартных путях доказана. Пусть $\xi \in \Phi^e$ и $\xi' \in \Phi^s$ — соответствующая LTL-формула. Покажем эквивалентность этих формул на входных и выходных путях.

- $H, \rho \vDash \xi \iff H, \pi_\rho \vDash \xi \iff H, \pi_\rho \vDash \xi' \iff H, \rho \vDash \xi'$;
- $H, \sigma \vDash \xi \iff \forall \pi_\sigma : H, \pi_\sigma \vDash \xi \iff \forall \pi_\sigma : H, \pi_\sigma \vDash \xi' \iff H, \sigma \vDash \xi'$. ■

Утверждение 2. *cycle-LTL-формулы с начальными входными цикловыми операторами эквивалентны следующим LTL-формулам:*

- $\mathbf{X^i}\varphi \equiv \mathbf{X}(\neg Input \mathbf{U}(Input \wedge \varphi))$;
- $\mathbf{F^i}\varphi \equiv \mathbf{F}(Input \wedge \varphi)$;
- $\mathbf{G^i}\varphi \equiv \mathbf{G}(Input \rightarrow \varphi)$;
- $\varphi \mathbf{U^i} \psi \equiv (Input \rightarrow \varphi)\mathbf{U}(Input \wedge \psi)$.

Доказательство.

Проводится индукцией по структуре формулы. Рассмотрим сначала входные пути.

- $H, \rho \vDash \mathbf{X^i}\varphi \iff H, \rho^1 \vDash \varphi \overset{def.\pi_\rho, Input}{\iff}$
$\forall 1 \le k < i_2^\pi : (H, \pi_\rho^k \vDash \neg Input) \bigwedge (H, \pi_\rho^{i_2^\pi} \vDash Input \wedge \varphi) \iff$
$H, \pi_\rho \vDash \mathbf{X}(\neg Input \mathbf{U}(Input \wedge \varphi)) \iff H, \rho \vDash \mathbf{X}(\neg Input \mathbf{U}(Input \wedge \varphi))$;
- $H, \rho \vDash \mathbf{F^i}\varphi \iff \exists k \ge 0 : H, \rho^k \vDash \varphi \overset{def.\rho}{\iff}$
$\exists k \ge 0 : (H, \rho^k \vDash \varphi) \bigwedge (H, \rho^k \vDash Input) \overset{def.\pi_\rho}{\iff} \exists j \ge 0 : H, \pi_\rho^j \vDash Input \wedge \varphi \iff$
$H, \pi_\rho \vDash \mathbf{F}(Input \wedge \varphi) \iff H, \rho \vDash \mathbf{F}(Input \wedge \varphi)$;
- $H, \rho \vDash \mathbf{G^i}\varphi \iff \forall k \ge 0\ H, \rho^k \vDash \varphi \overset{def.\rho}{\iff}$
$\forall k \ge 0\ (H, \rho^k \vDash \varphi) \bigwedge (H, \rho^k \vDash Input) \overset{def.\pi_\rho, Input}{\iff} \forall j \ge 0\ H, \pi_\rho^j \vDash (Input \rightarrow \varphi)$
$\iff H, \pi_\rho \vDash \mathbf{G}(Input \rightarrow \varphi); \iff H, \rho \vDash \mathbf{G}(Input \rightarrow \varphi)$;
- $H, \rho \vDash \varphi \mathbf{U^i} \psi \iff \exists k \ge 0 : (H, \rho^k \vDash \psi) \bigwedge \forall 0 \le j < k : (H, \rho^j \vDash \varphi) \overset{def.\rho}{\iff}$
$\exists k \ge 0 : (H, \rho^k \vDash Input \wedge \psi) \bigwedge \forall 0 \le j < k : (H, \rho^j \vDash Input \wedge \varphi) \overset{def.\pi_\rho, Input}{\iff}$
$\exists l \ge 0 : (H, \pi_\rho^l \vDash Input \wedge \psi) \bigwedge \forall 0 \le i < l : (H, \pi_\rho^i \vDash Input \rightarrow \varphi) \iff$
$H, \pi_\rho \vDash (Input \rightarrow \varphi)\mathbf{U}(Input \wedge \psi) \iff H, \rho \vDash (Input \rightarrow \varphi)\mathbf{U}(Input \wedge \psi)$.

Эквивалентность формул Φ^i и вышеприведённым формулам из Φ^s на входных путях доказана. Пусть $\xi \in \Phi^i$ и $\xi' \in \Phi^s$ — соответствующая LTL-формула. Покажем эквивалентность этих формул на стандартных и выходных путях. Вывод основан на определении путей π_ρ, ρ^π, σ^π и ρ^σ.

- $H, \pi \vDash \xi \iff \forall \rho^\pi : H, \rho^\pi \vDash \xi \iff \forall \pi'_{\rho^\pi} : H, \pi'_{\rho^\pi} \vDash \xi' \overset{ind.by.struc.\xi'}{\iff} H, \pi \vDash \xi'$;
- $H, \sigma \vDash \xi \iff \forall \rho^\sigma : H, \rho^\sigma \vDash \xi \iff \forall \pi'_{\rho^\sigma} : H, \pi'_{\rho^\sigma} \vDash \xi' \iff \forall \sigma'^{\pi'_{\rho^\sigma}} : H, \sigma'^{\pi'_{\rho^\sigma}} \vDash \xi' \overset{ind.by.struc.\xi'}{\iff} H, \sigma \vDash \xi'$. ■

Утверждение 3. *cycle-LTL-формулы с начальными выходными цикловыми операторами эквивалентны следующим LTL-формулам:*

- $\mathbf{X^o}\varphi \equiv \mathbf{X}(\neg Output \mathbf{U}(Output \wedge \varphi))$;
- $\mathbf{F^o}\varphi \equiv \mathbf{F}(Output \wedge \varphi)$;
- $\mathbf{G^o}\varphi \equiv \mathbf{G}(Output \rightarrow \varphi)$;
- $\varphi \mathbf{U^o} \psi \equiv (Output \rightarrow \varphi)\mathbf{U}(Output \wedge \psi)$.

Доказательство.
Проводится индукцией по структуре формулы. Рассмотрим сначала выходные пути.

- $H, \sigma \models \mathbf{X^i}\varphi \iff H, \sigma^1 \models \varphi \overset{def.\pi_\sigma, Output}{\iff}$
$$\forall\ 1 \le k < o_2^\pi\ (H, \pi_\sigma^k \models \neg Output) \bigwedge (H, \pi_\sigma^{o_2^\pi} \models Output \wedge \varphi) \iff$$
$$H, \pi_\sigma \models \mathbf{X}(\neg Output \mathbf{U}(Output \wedge \varphi)) \iff H, \sigma \models \mathbf{X}(\neg Output \mathbf{U}(Output \wedge \varphi));$$
- $H, \sigma \models \mathbf{F^o}\varphi \iff \exists\ k \ge 0 : H, \sigma^k \models \varphi \overset{def.\sigma}{\iff}$
$$\exists\ k \ge 0 : (H, \sigma^k \models \varphi) \bigwedge (H, \sigma^k \models Output) \overset{def.\pi_\sigma}{\iff}$$
$$\exists\ j \ge 0 : H, \pi_\sigma^j \models Output \wedge \varphi \iff H, \pi_\sigma \models \mathbf{F}(Output \wedge \varphi) \iff$$
$$H, \sigma \models \mathbf{F}(Output \wedge \varphi);$$
- $H, \sigma \models \mathbf{G^o}\varphi \iff \forall\ k \ge 0\ H, \sigma^k \models \varphi \overset{def.\sigma}{\iff}$
$$\forall\ k \ge 0\ (H, \sigma^k \models \varphi) \bigwedge (H, \sigma^k \models Output) \overset{def.\pi_\sigma, Output}{\iff}$$
$$\forall\ j \ge 0\ H, \pi_\sigma^j \models (Output \rightarrow \varphi) \iff H, \pi_\sigma \models \mathbf{G}(Output \rightarrow \varphi) \iff$$
$$H, \sigma \models \mathbf{G}(Output \rightarrow \varphi);$$
- $H, \sigma \models \varphi \mathbf{U^o} \psi \iff \exists\ k \ge 0\ (H, \sigma^k \models \psi) \bigwedge \forall\ 0 \le j < k\ (H, \sigma^j \models \varphi) \overset{def.\sigma}{\iff}$
$$\exists\ k \ge 0 : (H, \sigma^k \models Output \wedge \psi) \bigwedge \forall\ 0 \le j < k\ (H, \sigma^j \models Output \wedge \varphi) \overset{def.\pi_\sigma, Output}{\iff}$$
$$\exists\ l \ge 0 : (H, \pi_\sigma^l \models Output \wedge \psi) \bigwedge \forall\ 0 \le i < l\ (H, \pi_\sigma^i \models Output \rightarrow \varphi) \iff$$
$$H, \pi_\sigma \models (Output \rightarrow \varphi) \mathbf{U}(Output \wedge \psi) \iff H, \sigma \models (Output \rightarrow \varphi) \mathbf{U}(Output \wedge \psi).$$

Эквивалентность формул Φ^o и вышеприведённым формулам из Φ^s на входных путях доказана. Пусть $\xi \in \Phi^o$ и $\xi' \in \Phi^s$ — соответствующая LTL-формула. Покажем эквивалентность этих формул на стандартных и входных путях. Вывод основан на определении путей π_ρ, σ^π, ρ^π и σ^ρ.

- $H, \pi \models \xi \iff H, \sigma^\pi \models \xi \iff H, \pi'_{\sigma^\pi} \models \xi' \overset{ind.by.struc.\xi'}{\iff} H, \pi \models \xi';$
- $H, \rho \models \xi \iff H, \sigma^\rho \models \xi; \iff H, \pi'_{\sigma^\rho} \models \xi' \iff H, \rho'^{\pi'_{\sigma^\rho}} \models \xi' \overset{ind.by.struc.\xi'}{\iff} H, \rho \models \xi'.$ ■

Это утверждение является прямым следствием предыдущих:

Утверждение 4. *Для каждой формулы из $\Phi^e \cup \Phi^i \cup \Phi^o$ существует эквивалентная ей LTL-формула линейно большей длины.*

Задача проверки моделей для формул логики cycle-LTL и систем переходов гиперпроцессов состоит в определении семантики формул cycle-LTL в заданной HTS, т.е. в вычислении множества $\{\chi \mid H, \chi \models \xi'$, где H – HTS, $\chi \in H,\ \xi \in \Phi^c\}$. Отметим, что для любой HTS можно за полиномиальное время построить структуру Крипке, содержащую то же самое множество стандартных путей. В силу этого замечания, утверждения 4 и оценки сложности задачи проверки моделей для LTL [11] верна следующая теорема:

Теорема 1. *Существует алгоритм, решающий задачу проверки моделей для формул cycle-LTL и систем переходов гиперпроцессов, сложность которого полиномиально зависит от размера системы переходов гиперпроцессов, и экспоненциально зависит от длины формулы cycle-LTL.*

Заключение

В этой статье мы разработали систему переходов гиперпроцессов (HTS) для моделирования программного обеспечения программируемых логических контроллеров (ПЛК) и новую логику cycle-LTL для формулирования свойств ПЛК. Разработанная HTS-модель естественным образом отражает особенности программ ПЛК, такие как циклы управления и работу с таймерами. Предложенная темпоральная логика cycle-LTL позволяет описывать свойства программ ПЛК как относительно малых временных шагов внутренней фазы исполнения циклов управления, так и больших временных

шагов собственно циклов управления. Мы описали трансляцию формул логики cycle-LTL в формулы логики LTL и доказали её корректность. Эта трансляция демонстрирует, что формулирование свойств систем управления непосредственно в терминах логики LTL оказывается значительно более сложным и запутанным. Мы показали, что в силу корректности этой трансляции, задача проверки моделей для HTS и cycle-LTL является разрешимой.

Мы планируем использовать изложенные в этой статье результаты для обеспечения корректного перевода процесс-ориентированного языка Reflex в язык Promela, используемый верификатором SPIN[23]. Кроме того, мы будем разрабатывать и реализовывать специальный алгоритм проверки моделей для формул cycle-LTL и HTS. Мы ожидаем, что этот алгоритм во многих случаях будет иметь меньшую временную сложность, чем стандартный алгоритм проверки моделей для LTL, за счет использования таких особенностей HTS, как цикличность, строгий порядок действий процессов и детерминированность этих действий. Мы также планируем изучить возможность использования логики cycle-LTL для формулирования свойств систем переходов более общего вида.

References

[1] I. Anureev, «Operacionnaya semantica annotirovannih Reflex programm», *Modelirovanie i analiz informacionnyh sistem*, vol. 26, no. 6, pp. 181–192, 2019.

[2] I. Anureev, N. Garanina, T. Liakh, A. Rozov, and V. Zyubin, «Towards safe cyber-physical systems: the Reflex language and its transformational semantics», in *IEEE International Siberian Conference on Control and Communications*, IEEE, 2019, pp. 18–20.

[3] E. Brinksma and A. Mader, «Verification and Optimization of a PLC Control Schedule», in *SPIN 2000 – SPIN Model Checking and Software Verification*, ser. LNCS, vol. 1885, Springer, 2000, pp. 73–92.

[4] V. Gourcuff, O. de Smet, and J.-M. Faure, «Improving large-sized PLC programs verification using abstractions», *IFAC Proceedings Volumes*, vol. 41, no. 2, pp. 5101–5106, 2008.

[5] A. Mader, «A Classification of PLC Models and Applications», in *Discrete Event Systems*, ser. SECS, vol. 569, Springer, 2000, pp. 239–246.

[6] H. Wan, G. Chen, X. Song, and M. Gu, «Formalization and Verification of PLC Timers in Coq», in *Proc. of 33rd Annual IEEE International Computer Software and Applications Conference*, IEEE, 2009, pp. 315–323.

[7] J. Yoo, S. Cha, and E. Jee, «A Verification Framework for FBD Based Software in Nuclear Power Plants», in *Proc. of 15th Asia-Pacific Software Engineering Conference*, IEEE, 2008, pp. 385–392.

[8] D. Bulavskij, V. Zyubin, N. Karlson, V. Krivoruchko, and V. Mironov, «An Automated Control System for a Silicon Single-Crystal Growth Furnace», *Optoelectronics, instrumentation, and data processing*, vol. 32, no. 2, pp. 25–30, 1996.

[9] P. G. Kovadlo, A. Lubkov, A. Bevzov, and et al., «Automation system for the large solar vacuum telescope», *Optoelectronics, Instrumentation and Data processing*, vol. 52, pp. 187–195, 2016.

[10] A. Gupta, V. Kahlon, S. Qadeer, and T. Touili, *Handbook of Model Checking*. Springer International Publishing, 2018, ch. 18. Model Checking Concurrent Programs, pp. 573–577.

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

[12] H. Dierks, «PLC-automata: A new class of implementable real-time automata», in *International AMAST Workshop on Aspects of Real-Time Systems and Concurrent and Distributed Software*, vol. 1231, Springer, 1997, pp. 111–125.

[13] T. Ovatman, «An overview of model checking practices on verification of PLC software», *Software & Systems Modeling*, vol. 4, no. 15, pp. 937–960, 2016.

[14] E. Kuzmin, D. Ryabukhin, and V. A. Sokolov, «On the expressiveness of the approach to constructing PLC-programs by LTL-specification», *Automatic Control and Computer Sciences*, vol. 7, no. 50, pp. 510–519, 2016.

[15] M. Zhang, «Towards automated safety vetting of PLC code in real-world plants», in *IEEE Symposium on Security and Privacy*, IEEE, 2019, pp. 522–538.

[16] A. Rajeev and T. Henzinger, «A really temporal logic», *Journal of the ACM*, vol. 41, no. 1, pp. 181–203, 1994.

[17] J. Xiong, «A User-Friendly Verification Approach for IEC 61131-3 PLC Programs», *Electronics*, vol. 4, no. 9, 2020.

[18] B. Beckert, «Regression verification for programmable logic controller software», in *International Conference on Formal Engineering Methods*, vol. 9407, Springer, 2015.

[19] O. Ljungkrantz, «An empirical study of control logic specifications for programmable logic controllers», *Empirical Software Engineering*, vol. 3, no. 19, pp. 655–677, 2014.

[20] O. Ljungkrantz, K. Åkesson, M. Fabian, and C. Yuan, «A formal specification language for PLC-based control logic», in *8th IEEE International Conference on Industrial Informatics*, IEEE, 2010, pp. 1067–1072.

[21] O. Maler and D. Nickovic, «Monitoring temporal properties of continuous signals», *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pp. 152–166, 2004.

[22] N. Garanina, I. Anureev, V. Zyubin, A. Rozov, T. Liakh, and S. Gorlatch, «Reasoning about Programmable Logic Controllers», *System Informatics*, vol. 17, pp. 33–42, 2020.

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

MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 27, NO. 4, 2020
journal homepage: www.mais-journal.ru

THEORY OF COMPUTING

On the Model Checking Problem for Some Extension of CTL*

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

DOI: 10.18255/1818-1015-2020-4-428-441

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

MSC2020: 68Q45
Research article
Full text in Russian

Received November 16, 2020
After revision December 1, 2020
Accepted December 16, 2020

Sequential reactive systems include programs and devices that work with two streams of data and convert input streams of data into output streams. Such information processing systems include controllers, device drivers, computer interpreters. The result of the operation of such computing systems are infinite sequences of pairs of events of the request–response type, and, therefore, finite transducers are most often used as formal models for them. The behavior of transducers is represented by binary relations on infinite sequences, and so, traditional applied temporal logics (like HML, LTL, CTL, mu-calculus) are poorly suited as specification languages, since omega-languages, not binary relations on omega-words are used for interpretation of their formulae. To provide temporal logics with the ability to define properties of transformations that characterize the behavior of reactive systems, we introduced new extensions of these logics, which have two distinctive features: 1) temporal operators are parameterized, and languages in the input alphabet of transducers are used as parameters; 2) languages in the output alphabet of transducers are used as basic predicates. Previously, we studied the expressive power of new extensions Reg-LTL and Reg-CTL of the well-known temporal logics of linear and branching time LTL and CTL, in which it was allowed to use only regular languages for parameterization of temporal operators and basic predicates. We discovered that such a parameterization increases the expressive capabilities of temporal logic, but preserves the decidability of the model checking problem. For the logics mentioned above, we have developed algorithms for the verification of finite transducers. At the next stage of our research on the new extensions of temporal logic designed for the specification and verification of sequential reactive systems, we studied the verification problem for these systems using the temporal logic Reg-CTL*, which is an extension of the Generalized Computational Tree Logics CTL*. In this paper we present an algorithm for checking the satisfiability of Reg-CTL* formulae on models of finite state transducers and show that this problem belongs to the complexity class ExpSpace.

Keywords: reactive system; model checking; finite state transducer; temporal logic; regular language; specification

INFORMATION ABOUT THE AUTHORS

Anton Romanovich Gnatenko	orcid.org/0000-0003-1499-2090. E-mail: gnatenko.cmc@gmail.com student.
Vladimir Anatolyevich Zakharov correspondence author	orcid.org/0000-0002-3794-9565. E-mail: zakh@cs.msu.ru Dr. of Science, professor.

Funding: The reported study was funded by RFBR, project number 18-01-00854.

For citation: A. R. Gnatenko and V. A. Zakharov, "On the Model Checking Problem for Some Extension of CTL*", *Modeling and analysis of information systems*, vol. 27, no. 4, pp. 428-441, 2020.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 27, № 4, 2020
сайт журнала: www.mais-journal.ru

THEORY OF COMPUTING

О задаче верификации моделей программ для одного расширения логики CTL*

А. Р. Гнатенко[1], В. А. Захаров[1,2]

DOI: 10.18255/1818-1015-2020-4-428-441

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

УДК 519.7
Научная статья
Полный текст на русском языке

Получена 16 ноября 2020 г.
После доработки 1 декабря 2020 г.
Принята к публикации 16 декабря 2020 г.

К последовательным реагирующим системам относятся программы и устройства, которые работают с двумя потоками данных и осуществляют преобразование входных потоков данных в выходные потоки. К числу таких систем обработки информации относятся контроллеры, драйверы устройств, компьютерные интерпретаторы. Результатом работы таких вычислительных систем являются бесконечные последовательности пар событий типа запрос–отклик, и поэтому в качестве математических моделей для них наиболее часто используются конечные автоматы-преобразователи. Поведение автоматов-преобразователей представлено бинарными отношениями на бесконечных последовательностях, и традиционные прикладные темпоральные логики (HML, LTL, CTL, mu-исчисление) плохо подходят для этой цели, поскольку для интерпретации их формул используются omega-языки, а не бинарные отношения на omega-словах. Чтобы предоставить темпоральным логикам возможность определять свойства преобразований, которые характеризуют поведение реагирующих систем, мы ввели новые расширения этих логик, имеющие две отличительные особенности: 1) темпоральные операторы в расширениях этих логик параметризованы, и в качестве параметров используются языки в входном алфавите автоматов-преобразователей; 2) в качестве базовых предикатов используются языки в выходном алфавите автоматов-преобразователей. Ранее нами были исследованы выразительные возможности новых расширений Reg-LTL и Reg-CTL известных темпоральных логик линейного и ветвящегося времени LTL и CTL, в которых для параметризации темпоральных операторов и задания базовых предикатов разрешалось использовать только регулярные языки. Мы обнаружили, что такая параметризация увеличивает выразительные возможности темпоральной логики, но сохраняет разрешимость задачи проверки выполнимости формул на конечных моделях. Для указанных выше логик нами были разработаны алгоритмы верификации конечных автоматов-преобразователей. На следующем этапе изучения новых расширений темпоральной логики, предназначенных для спецификации и верификации последовательных реагирующих систем, мы обратились к задаче верификации этих систем с использованием темпоральной логики Reg-CTL*, которая является расширением обобщенной логики деревьев вычислений CTL*. В этой статье описан алгоритм проверки выполнимости формул Reg-CTL* на моделях конечных автоматов-преобразователей и показано, что эта задача принадлежит классу сложности ExpSpace.

Ключевые слова: реагирующая система; верификация моделей программ; конечный автомат-преобразователь; темпоральная логика; регулярный язык; спецификация

ИНФОРМАЦИЯ ОБ АВТОРАХ

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

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

Финансирование: Работа выполнена при финансовой поддержке гранта РФФИ N 18-01-00854.

Для цитирования: A. R. Gnatenko and V. A. Zakharov, “On the Model Checking Problem for Some Extension of CTL*”, *Modeling and analysis of information systems*, vol. 27, no. 4, pp. 428-441, 2020.

Введение

Конечные автоматы широко используются в информатике как модели последовательных вычислительных систем. В частности, автоматы-преобразователи с конечным числом состояний служат подходящей формальной моделью для различных программных и аппаратных систем, таких как контроллеры, системные драйверы, сетевые коммутаторы, интерпретаторы программ, которые работают с двумя потоками данных. Эти устройства и программы получают потоки данных на свои входы и преобразуют их в выходные потоки управляющих сигналов (команд, данных). К аппаратным устройствам этого типа относятся адаптеры, сетевые коммутаторы, контроллеры. В программировании автоматы-преобразователи используются в качестве формальных моделей различных программ и протоколов, которые обрабатывают строки символов, потоки команд и данных (см. [1—3]). Такие информационные системы объединяются под общим названием *последовательные реагирующие системы*. Их вычисления проводятся в дискретном времени; на каждом этапе вычислений система получает управляющий сигнал (или часть входных данных) из среды и генерирует (выполняет, выводит) последовательность действий (или часть выходных данных) в ответ. Эти выходные действия и порядок их выполнения зависят от всех предыдущих управляющих сигналов.

В этой статье в центре внимания будут находиться реагирующие системы с ограниченной памятью, которые работают в оперативном (on-line) режиме. В статье [4] мы рассмотрели ряд примеров, которые показывают, что автоматы-преобразователи с конечным числом состояний являются простой и адекватной математической моделью для представления последовательных реагирующих систем во многих приложениях. Поведение таких систем характеризуется не набором последовательностей событий, а взаимосвязью между двумя последовательностями событий. Типичное свойство поведения, требующее проверки, состоит в том, что для каждого входного слова, подпадающего под некоторый шаблон допустимого воздействия внешней среды реагирующая система всегда вычисляет выходное слово, которое подпадает под другой шаблон, определяющий ожидаемый тип отклика системы. Требования такого рода можно сформулировать с помощью темпоральной логики, адаптированной для рассуждений о парах последовательностей событий.

Такие темпоральные логики были введены и изучены в [4—6]. В статье [5] новое расширение $\mathcal{LP}$-LTL темпоральной логики линейного времени LTL было представлено как формальный язык для спецификации поведения последовательных реагирующих систем. В логике $\mathcal{LP}$-LTL темпоральные операторы параметризованы множествами слов (языками), которые представляют допустимые потоки управляющих сигналов, воздействующих на реагирующую систему. Базовые предикаты в $\mathcal{LP}$-LTL также являются языками в алфавите основных действий преобразователя; они представляют собой ожидаемую реакцию преобразователя на указанные воздействия окружающей среды. В статье [5] авторы исследовали задачу проверки модели для регулярного фрагмента *Reg*-LTL этой логики, когда только регулярные языки используются в качестве базовых предикатов и параметров темпоральных операторов. Было показано, что задача проверки выполнимости формул *Reg*-LTL на моделях конечных автоматов-преобразователей разрешима за двойное экспоненциальное время. В статье [4] мы изучили выразительные возможности логики $\mathcal{LP}$-LTL и сравнили ее с другими темпоральными логиками, используемыми для спецификации поведения реагирующих систем. В статье [6] был предложен табличный алгоритм верификации моделей для логики *Reg*-CTL, которая является регулярным расширением темпоральной логики деревьев вычислений CTL.

Здесь мы рассматриваем более общий язык спецификаций последовательных реагирующих систем, являющийся расширением обобщенной логики деревьев вычислений CTL^*, фрагментами которой являются упомянутые выше темпоральные логики LTL и CTL. Как показано в [7], алгоритм решения задачи верификации моделей для логики CTL^* получается комбинацией алгоритмов

решения аналогичной задачи для LTL и CTL. Однако в случае, когда темпоральные операторы параметризованы регулярными выражениями, ситуация оказывается более сложной и требует отдельного исследования. Это исследование проводится в настоящей статье. Мы вводим в рассмотрение расширенную логику $\mathcal{LP}$-CTL^* и её регулярный фрагмент Reg-CTL^* и решаем для этого фрагмента задачу верификации моделей программ. Основные результаты, полученные в данной работе, таковы:

1. Предложено новое расширение темпоральной логики деревьев вычислений $\mathcal{LP}$-CTL^* и определена формальная семантика этой логики на деревьях вычислений конечных автоматов-преобразователей.
2. Для этой логики разработан теоретико-автоматный алгоритм верификации моделей автоматов-преобразователей, использующий объём памяти, экспоненциальный относительно размеров проверяемых автомата и формулы.
3. Показано, что указанная задача верификации автоматов-преобразователей для логики Reg-CTL^* является PSpace-трудной и принадлежит классу сложности ExpSpace.

Полученные результаты основаны на методах, предложенных в работах [5, 6, 8], и продолжают направление исследований, инициированное и развитое в этих статьях. В разделе 1 мы вводим формальное определение автоматов-преобразователей, а также определяем синтаксис и семантику языка темпоральной логики Reg-CTL^*. Далее приведен краткий обзор некоторых ранее известных расширений обычных темпоральных логик. В разделе 3 описан алгоритм решения задачи верификации моделей для формул логики Reg-CTL^* и установлены оценки его сложности. В заключении мы сравниваем полученные в статье результаты с результатами решения аналогичной задачи для других расширений темпоральной логики, а также обсуждаем дальнейшие направления исследований в этой области.

1. Модели реагирующих систем и их спецификация

Последовательные реагирующие системы обработки информации, такие как адаптеры, контроллеры, драйверы устройств, интерпретаторы программ, сетевые коммутаторы работают с двумя потоками данных и преобразуют входные потоки данных (управляющих сигналов, команд) в выходные потоки данных (управляющих сигналов, команд). Для моделирования вычислений таких систем можно использовать конечные автоматы-преобразователи.

Пусть заданы конечные алфавиты $\mathcal{C}$ и $\mathcal{A}$. Элементы множества $\mathcal{C}$ называются *входными сигналами*, а элементы множества $\mathcal{A}$ — *элементарными действиями*. Обозначим записью $\mathcal{A}^*$ множество всех конечных слов над алфавитом $\mathcal{A}$, которые называются *составными действиями*. Для двух составных действий u и v мы пишем uv для их конкатенации и используем обозначение ε для пустого слова. Записи $Reg(\mathcal{C})$ и $Reg(\mathcal{A})$ будут обозначать семейства всех регулярных языков над алфавитами $\mathcal{C}$ и $\mathcal{A}$ соответственно.

Конечный *автомат-преобразователь* (или просто преобразователь) над алфавитами $\mathcal{C}$ и $\mathcal{A}$ — это пятёрка $\pi = (Q, \mathcal{C}, \mathcal{A}, q_{init}, T)$, где Q — конечное множество *управляющих состояний*, $q_{init} \in Q$ — *начальное состояние*, а $T \subseteq Q \times \mathcal{C} \times Q \times \mathcal{A}^*$ — *отношение переходов*. При этом требуется, чтобы отношение T было тотальным, т.е. чтобы для любого состояния $q \in Q$ и любого входного сигнала $c \in \mathcal{C}$ существовали такие $q' \in Q$ и $h \in \mathcal{A}^*$, что $(q, c, q', h) \in T$. *Размером* преобразователя π мы будем называть размер табличного описания его отношения переходов. Эта величина обозначается записью $|\pi|$.

Четвёрка (q, c, q', h) называется *переходом*; он означает, что преобразователь, находясь в состоянии q и получив входной сигнал c, может перейти в состояние q' и выполнить последовательность элементарных действий h. Таким образом, отношение переходов моделирует программу реагирующей системы. *Траектория* преобразователя π из состояния q_0 — это бесконечная последовательность переходов $tr = \{\tau_i\}_{i \geqslant 1}$, где $\tau_i = (q_{i-1}, c_i, q_i, h_i) \in T$ для всех i, $1 \leqslant i \leqslant n$. Последовательность

пар $(c_1, h_1), (c_2, h_2), \ldots$, которыми помечены переходы траектории tr, представляет собой всю информацию о работе реагирующей системы, доступную внешнему наблюдателю. Множество всех траекторий из состояния q обозначается записью $Tr_\pi(q)$. Для краткости вместо $Tr_\pi(q_{init})$ условимся использовать обозначение Tr_π.

Наряду с бесконечными траекториями мы будем также рассматривать конечные траектории. Конечной траекторией из состояния q называется любой конечный префикс некоторой траектории из этого состояния. Множество конечных траекторий из состояния q обозначим записью $Fin\ Tr_\pi(q)$.

Поведение преобразователя $\pi = (Q, \mathcal{C}, \mathcal{A}, q_{init}, T)$ описывается *графом вычислений* — ориентированным графом $\Gamma_\pi = (V_\pi, E_\pi)$ с множествами вершин $V = (Q \times \mathcal{A}^*)$ и помеченных дуг $E \subseteq V \times \mathcal{C} \times V$, согласованных с отношением переходов T следующим образом: для любой пары вершин $u' = (q', s')$ и $u'' = (q'', s'')$, для произвольного входного сигнала c имеет место соотношение

$$(u', c, u'') \in E \iff \exists h \in \mathcal{A}^* : (q', c, q'', h) \in T \wedge s'' = s'h.$$

Вершину графа вычислений (q_{init}, ε) будем называть *начальной* и обозначать записью v_{init}. Между траекториями преобразователя и путями в графе вычислений имеется соответствие, которое важно для последующих построений. Каждой паре, включающей траекторию $tr = \{(q_{i-1}, c_i, q_i, d_i)\}_{i \geqslant 1}$ из состояния q_0 и составное действие s_0, сопоставим путь $\rho = \{(v_{i-1}, c_i, v_i)\}_{i \geqslant 1}$ в графе вычислений Γ_π, где $v_0 = (q_0, s_0)$ и $v_i = (q_i, s_{i-1}h_i)$ для всех $i \geqslant 1$. Для каждой конечной траектории $tr = \{(q_{i-1}, c_i, q_i, h_i)\}_{i=1}^{k} \in Fin\ Tr_\pi$ обозначим записью $v_{init}[tr]$ вершину (q_k, h) графа Γ_π, где $h = h_1 h_2 \ldots h_k$. Мы будем использовать традиционное обозначение $\rho|^k$ для бесконечного суффикса пути $\rho = \{(v_{i-1}, c_i, v_i)\}_{i \geqslant 1}$, начинающегося из вершины v_k.

Задача верификации вычислительной системы состоит в том, чтобы выяснить обладает ли поведение системы требуемыми свойствами. Выбрав конечные преобразователи в качестве моделей последовательных реагирующих систем, мы можем полагать, что поведение преобразователя полностью характеризуется множеством его начальных траекторий Tr_π (или, ввиду описанного выше соответствия, множеством путей в графе Γ_π, начинающихся в вершине v_{init}). Таким образом, «свойством поведения» преобразователя, может считаться любое свойство его траекторий, т.е. произвольное подмножество $Property \subseteq Tr_\pi$.

Хорошо известно, что свойства бесконечных последовательностей состояний системы, также как и свойства систем переходов, наподобие графа вычислений, удобно описывать формулами темпоральных логик (LTL, CTL, CTL^*). Однако следует подчеркнуть, что траектория преобразователя содержит в себе не только последовательность состояний, но также входную и выходную последовательности, связь между которыми осуществляется реагирующей системой. Поэтому для спецификации поведения преобразователя необходим язык спецификаций, позволяющий формально описывать эту связь.

Желаемого эффекта можно достигнуть, снабдив модальные операторы темпоральной логики параметрами; в качестве параметров в общем случае разрешается использовать любой язык в алфавите $\mathcal{C}$. Далее мы определим синтаксис и семантику темпоральной логики $\mathcal{LP}$-CTL^*, которая является расширением обобщенной логики деревьев вычислений CTL^*, и будем исследовать наиболее интересный с практической точки зрения фрагмент Reg-CTL^* введенного расширения. В формулах Reg-CTL^* параметрами темпоральных операторов могут быть только регулярные языки. Логика обобщает Reg-CTL^* языки спецификаций Reg-LTL и Reg-CTL поведения автоматов-преобразователей, изученные в наших предыдущих статьях [4—6, 8].

Поведение преобразователя считается правильным, если реакция системы на определенные сигналы, поступающие из внешней среды, соответствует ожиданию. Желаемый тип этой реакции может быть описан множеством допустимых последовательностей элементарных действий, т.е. некоторым языком в алфавите действий $\mathcal{A}$. Любой такой язык $P \subseteq \mathcal{A}^*$ называется *базовым*

предикатом. При этом нас может интересовать отклик системы на последовательности входных сигналов, подпадающих под некоторый *шаблон поведения окружения*, который задан некоторым языком $L \subseteq C^*$.

Пусть задано семейство $\mathcal{L}$ шаблонов поведения окружения и семейство $\mathcal{P}$ базовых предикатов. Формулы логики $\mathcal{LP}$-CTL^* подразделяются на два типа — *формулы состояния* и *формулы пути*, — и определяются следующим образом:

1) всякий базовый предикат $P \in \mathcal{P}$ — это формула состояния;
2) если $\varphi_1,\ \varphi_2$ — формулы состояния, то $\neg\varphi_1$ и $\varphi_1 \wedge \varphi_2$ — формулы состояния;
3) если ψ — формула пути, то $\mathbf{A}\psi$ и $\mathbf{E}\psi$ — формулы состояния;
4) если φ — формула состояния, то φ — формула пути;
5) если $\psi_1,\ \psi_2$ — формулы пути, то $\neg\psi_1$ и $\psi_1 \wedge \psi_2$ — формулы пути;
6) если ψ, ψ_1, ψ_2 — формулы пути, $c \in C$, и $L \in \mathcal{L}$, то $\mathbf{X}_c\psi$ и $\psi_1\,\mathbf{U}_L\psi_2$ — формулы пути.

Формулы $\mathcal{LP}$-CTL^* интерпретируются на графах вычислений. Пусть Γ_π — граф вычислений преобразователя $\pi = (Q, C, \mathcal{A}, q_{init}, T)$. Запись $\Gamma_\pi, v \models \varphi$ обозначает, что формула состояния φ выполняется в вершине v графа Γ_π. Запись $\Gamma_\pi, \rho \models \psi$, в свою очередь, означает, что формула пути ψ выполняется на пути ρ этого графа.

Пусть $v = (q, s)$ — вершина графа вычислений Γ_π, а $\rho = (v_0, c_1, v_1), (v_1, c_2, v_2), \ldots$ — путь в этом графе. Отношение выполнимости $\models$ определяется индуктивно следующим образом:

1) $\Gamma_\pi, v \models P \iff s \in P$ для всякого базового предиката $P \in \mathcal{P}$;
2) $\Gamma_\pi, v \models \neg\varphi \iff$ неверно, что $\Gamma_\pi, v \models \varphi$;
3) $\Gamma_\pi, v \models \varphi_1 \wedge \varphi_2 \iff \Gamma_\pi, v \models \varphi_1$ и $\Gamma_\pi, v \models \varphi_2$;
4) $\Gamma_\pi, v \models \mathbf{E}\,\varphi \iff$ существует путь ρ_v из вершины v, для которого $\Gamma_\pi, \rho \models \varphi$;
5) для любой формулы состояния $\varphi:\ \Gamma_\pi, \rho \models \varphi \iff \Gamma_\pi, v_0 \models \varphi$;
6) $\Gamma_\pi, \rho \models \neg\psi \iff$ неверно, что $\Gamma_\pi, \rho \models \psi$;
7) $\Gamma_\pi, \rho \models \psi_1 \wedge \psi_2 \iff \Gamma_\pi, \rho \models \psi_1$ и $\Gamma_\pi, \rho \models \psi_2$;
9) $\Gamma_\pi, \rho \models \mathbf{X}_c\varphi_1 \iff c = c_1$ и $\Gamma_\pi, \rho|^1 \models \varphi_1$;
10) $\Gamma_\pi, \rho \models \varphi_1\,\mathbf{U}_L\varphi_2 \iff \exists i \geqslant 0:\ c_1c_2\ldots c_i \in L$, где $\Gamma_\pi, \rho|^i \models \varphi_2$, причём $\forall j,\ 0 \leqslant j < i$, если $c_1c_2\ldots c_j \in L$, то $\Gamma_\pi, \rho|^j \models \varphi_1$.

Формула $\mathbf{X}_c\varphi_1$ означает, что свойство φ_1 окажется выполненным на следующем шаге, причём этот шаг будет сделан в ответ на получение сигнала c. Формула $\varphi_1\,\mathbf{U}_L\varphi_2$ утверждает, что свойство φ_1 будет выполняться каждый раз, когда поведение окружения соответствует шаблону L, до тех пор, пока не выполнится свойство φ_2. При этом φ_2 также должно выполниться в момент времени, когда последовательность входных сигналов удовлетворяет шаблону. Для удобства записи можно определить другие темпоральные операторы, а также ввести квантор всеобщности для путей:

- Оператор **F**uture: $\mathbf{F}_L\varphi \equiv \mathbf{true}\,\mathbf{U}_L\varphi$: свойство φ выполнилось когда-нибудь при получении последовательности сигналов, удовлетворяющей шаблону L.
- Оператор **G**obally: $\mathbf{G}_L\varphi \equiv \neg\,\mathbf{F}\,\neg\varphi$, двойственный оператору **F**.
- Квантор всеобщности для путей: $\mathbf{A}\psi \equiv \neg\,\mathbf{E}\neg\psi$.

Если $v = v_{init}$, то вместо $\Gamma_\pi, v \models \varphi$ будем использовать запись $\pi \models \varphi$. Задача верификации преобразователя π относительно спецификации $\varphi \in \mathcal{LP}$-CTL^* заключается в проверке соотношения $\pi \models \varphi$.

В общем случае $\mathcal{L}$ и $\mathcal{P}$ могут быть произвольными семействами языков, и эта свобода выбора может приводить к неразрешимости задачи верификации преобразователей. Действительно, пусть $C = \mathcal{A} = \{a, b\}$, а преобразователь π обладает единственным состоянием $q_{init} = q$ и отношением переходов $T = \{(q, c, c, q):\ c \in C\}$. Тогда для любых двух языков $U, V \subseteq C^*$ верно следущее соотношение:

$$\pi \models \mathbf{EF}_U V \iff U \cap V \neq \varnothing.$$

Так как проблема проверки пустоты пересечения двух контекстно-свободных языков неразрешима [9], задача верификации преобразователей оказывается неразрешимой, если в качестве шаблонов поведения окружения и базовых предикатов разрешается использовать произвольные контекстно-свободные языки. Чтобы избежать этой ситуации, в качестве $\mathcal{L}$ и $\mathcal{P}$ мы будем использовать семейства регулярных языков. Выбор этого класса языков оправдан еще и тем, что регулярные языки широко используются в практических приложениях, и для работы с ними существуют эффективные и доступные программные средства.

Полагая $\mathcal{L} = Reg(C)$ и $\mathcal{P} = Reg(\mathcal{A})$, мы получим фрагмент логики $\mathcal{LP}$-CTL^*, который условимся обозначать записью Reg-CTL^*. В следующем разделе мы сравним этот фрагмент введенной нами логики с другими расширениями темпоральных логик, которые были введены и исследованы ранее. Далее для логики Reg-CTL^* будет описан алгоритм решения задачи верификации преобразователей.

2. Другие расширения темпоральных логик

Всесторонний анализ темпоральной логики LTL как формального языка для описания поведения вычислительных систем был впервые проведен в работе [10]. Вскоре после этого было предпринято несколько попыток улучшить выразительные возможности этой темпоральной логики. Модификации проводились в основном по двум направлениям: 1) добавление новых выразительных средств (кванторов, модальностей и т.д.) и 2) изменение семантики темпоральных операторов.

Например, авторы [11] снабдили синтаксис LTL количественной оценкой основных предикатов и обнаружили, что выразительные возможности введенного таким образом количественного расширения значительно превосходят описательные возможности простого LTL. С другой стороны, в статье [12] был предложен метод введения новых темпоральных операторов с использованием праволинейных грамматик. Слова, порождаемые этими грамматиками, определяют шаблоны, по которым проверяется выполнимость формул в рамках временного оператора. Аналогично, в статьях [13—15] было показано, что шаблоны для описания семантики новых темпоральных операторов могут быть определены с помощью конечных автоматов и регулярных выражений. Идея снабжения темпоральных операторов некоторыми параметрами не нова: почти такая же параметризация темпоральных операторов, как в этой статье, была введена в [16] для динамически расширяемого LTL. С тех пор было предпринято несколько попыток такого рода объединить регулярные языки и временные операторы (например, см. [17, 18] среди последних). Почти во всех этих случаях было обнаружено, что выразительные возможности этих расширений увеличиваются и становятся эквивалентными таковым для логики $S1S$, в то время как проблема проверки выполнимости для этих логик остается PSpace-полной.

Вводя новое расширение $\mathcal{LP}$-CTL^* языка темпоральной логики деревьев вычислений, мы не стремились всего лишь улучшить выразительность языка CTL^*. Наша цель состояла в том, чтобы предложить адекватный язык для описания поведения реагирующих систем, моделируемых преобразователями. При вычислении преобразователя выполняется согласованное формирование двух последовательностей — последовательности входных сигналов и последовательности выходных действий. Поэтому отличительной особенностью семантики $\mathcal{LP}$-CTL^* является определенная синхронизация параметров темпоральных операторов (их интерпретация определяется последовательностью входных сигналов) и значений истинности базовых предикатов (они зависят от последовательности выходных действий). Можно сказать, что семантика нашей логики определяется на двухмерных трассах, тогда как во всех ранее известных параметризованных расширениях темпоральной логики использовались только одномерные трассы. Эта семантическая особенность существенно влияет на алгоритмы проверки выполнимости формул.

3. Верификация реагирующих систем относительно формул логики *Reg-CTL**

Задача верификации конечных преобразователей была впервые исследована в работе [5], где был предложен алгоритм проверки выполнимости $\pi \models \varphi$ формул логики *Reg-LTL* на множествах вычислений (траекторий) конечных преобразователей. Предложенная в этой работе логика *Reg-LTL* является линейным фрагментом рассматриваемой здесь логики *Reg-CTL**, который содержит только формулы вида $\mathbf{A}\varphi$, где φ — бескванторная формула пути. Алгоритм верификации, описанный в [5], основан на трансляции пары (π, φ) в такой автомат Бюхи $B(\pi, \varphi)$, что выполнимость $\pi \models \varphi$ имеет место тогда и только тогда, когда автомат $B(\pi, \varphi)$ допускает хотя бы одно слово. Из разрешимости проблемы пустоты для автоматов Бюхи следует разрешимость задачи проверки выполнимости формул логики *Reg-LTL* на конечных моделях.

В этом разделе мы развиваем идею алгоритма, предложенного в работе [5], и представляем более общий алгоритм верификации преобразователей относительно произвольных формул φ логики *Reg-CTL**, основанный на рекурсивном применении алгоритма из [5] к подформулам состояния формулы φ. Для описания этого алгоритма потребуется ввести несколько дополнительных определений.

3.1. Автоматы распознаватели над конечными и бесконечными словами

Детерминированный конечный *автомат-распознаватель* (или просто автомат) над алфавитом Σ — это пятерка $A = (Q, \Sigma, q_{init}, \delta, F)$, где Q — конечное множество состояний, $q_{init} \in Q$ — начальное состояние, $F \subset Q$ — множество *принимающих состояний*, а $\delta : Q \times \Sigma \longrightarrow Q$ — функция переходов. В отличие от отношения переходов преобразователя, функция переходов δ однозначно определяет следующее состояние по данной паре $(q, \sigma) \in Q \times \Sigma$. Эту функцию можно продолжить на всё множество Σ^* стандартным образом: $\delta(q, \varepsilon) = q$, и $\delta(q, \sigma x) = \delta(\delta(q, \sigma), x)$ для всех $q \in Q, \sigma \in \Sigma$ и $x \in \Sigma^*$. Автомат $A[x] = (Q, \Sigma, \delta(q_{init}, x), \delta, F)$ будем называть *сдвигом* A *на* слово x. Автомат A *принимает* слово x в том и только том случае, когда $\delta(q_{init}, x) \in F$. Говорят, что A *распознает* язык $L(A) = \{x : \delta(q_{init}, x) \in F\}$. Для краткости мы будем обозначать и автомат, и распознаваемый им язык одним и тем же символом A и записывать, например, $x \in A$ и $A = \varnothing$ вместо $x \in L(A)$ и $L(A) = \varnothing$.

Размером $|A|$ автомата A назовем размер табличного описания его функции переходов δ. Размером регулярного языка L мы называем величину $|L|$, равную размеру *минимального* автомата, распознающего этот язык.

Вычисления конечных автоматов можно определить также для бесконечных слов (ω-слов) в алфавите Σ. Множество всех таких слов обозначим записью Σ^ω.

Недетерминированный *обобщенный автомат Бюхи* [19] над алфавитом Σ — это пятерка $B = (Q, \Sigma, q_{init}, \Delta, \mathcal{F})$, где, в отличие от определения конечного автомата, $\Delta \subseteq Q \times \Sigma \times Q$ является *отношением переходов*, а $\mathcal{F} = \{F_1, \ldots, F_m\}$ — *допускающим правилом*, состоящим из компонент $F_i \subseteq Q, 1 \leqslant i \leqslant m$. Для произвольного заданного ω-слова $x = \sigma_0\sigma_1\sigma_2\cdots \in \Sigma^\omega$ *прогоном* автомата B на x назовем бесконечную последовательность состояний $q_0, q_1, q_2, \ldots$, в которой $q_0 = q_{init}$, и для всех $i, i \geqslant 0$, выполняется включение $(q_i, \sigma_i, q_{i+1}) \in \Delta$. Прогон $q_0, q_1, q_2, \ldots$ автомата B называется *допускающим*, если для каждого допускающего правила $F_i \in \mathcal{F}, 1 \leq i \leq m$, этого автомата имеется бесконечно много таких значений j, для которых $q_j \in F_i$. Автомат B *принимает* ω-слово $x \in \Sigma^\omega$ если и только если существует допускающий прогон B на этом слове. Понятие языка автомата B определяется аналогично случаю обычного автомата. Размером автомата B называется размер $|B|$ табличного описания отношения Δ.

Хорошо известны (см., например, [7]) эффективные процедуры проверки языка автомата Бюхи на пустоту, основанные на поиске в графе переходов автомата компонент сильной связности и завершающие вычисления за время, линейное относительно размера автомата $|B|$. Эти процедуры положены в основу теоретико-автоматного подхода к верификации вычислительных систем

для темпоральных логик [20]. Именно этого подхода мы будем придерживаться при построении алгоритма верификации преобразователей для логики $Reg\text{-}CTL^*$.

3.2. Трансляция формул $Reg\text{-}CTL^*$ в автоматы

Перейдем к построению алгоритма верификации. На его вход подается преобразователь $\pi = (Q, C, \mathcal{A}, q_{init}, T)$ и формула $\varphi \in Reg\text{-}CTL^*$. Алгоритм должен проверять, справедливо ли соотношение $\pi \models \varphi$ для этих входных данных.

Предположим, что формула φ имеет вид $\mathbf{E}\psi$, где ψ — бесквантерная формула. Задача верификации для этого частного случая решена в статье [5], где предложен алгоритм трансляции пары (π, φ) в автомат Бюхи $B(\pi, \varphi)$, язык которого непуст тогда и только тогда, когда $\pi \models \varphi$. При этом для каждого базового предиката P, используемого в формуле φ и представляющего собой язык в алфавите $\mathcal{A}$, строится распознающий его автомат A_P. Заметим, что этим способом можно решить задачу и в случае, когда φ имеет вид булевой комбинации формул вида $\mathbf{E}\psi$, т.е. для формул вида $\varphi = \Phi(\mathbf{E}\psi_1, \dots, \mathbf{E}\psi_n)$, где $\Phi(x_1, \dots, x_n)$ — булева формула.

В общем случае формула φ является булевой комбинацией подформул вида $\mathbf{E}\psi(\mathbf{E}\xi_1, \dots, \mathbf{E}\xi_m)$, где $\xi_1, \dots, \xi_m$ — некоторые формулы пути, а $\psi = \psi(P_1, \dots, P_n)$ — *бесквантерная формула пути*, в которой на места вхождения базовых предикатов $P_1, \dots, P_n$ подставлены формулы состояния $\mathbf{E}\xi_1, \dots, \mathbf{E}\xi_m$.

Рассмотрим произвольную конечную траекторию $tr \in Fin\,Tr_\pi$ преобразователя π. Она соответствует некоторому конечному пути $\rho = v_{init}, v_1, v_2, \dots v_t$ в графе вычислений Γ_π. Истинность формул состояния $\mathbf{E}\xi_i, 1 \leqslant i \leqslant m$, в вершине v_t определяется конечной траекторией tr, которую можно рассматривать как конечное слово tr в конечном алфавите переходов T преобразователя π. Предлагаемый нами алгоритм основан на построении для каждой формулы $\mathbf{E}\xi_i$ автомата над алфавитом T, принимающего те и только те конечные траектории, на которых выполняется формула $\mathbf{E}\xi_i$. Построенные автоматы затем используются при конструировании автомата Бюхи для формулы $\mathbf{E}\psi(\mathbf{E}\xi_1, \dots, \mathbf{E}\xi_m)$.

Введем необходимые понятия для описания этого построения. Будем верифицировать преобразователь $\pi = (Q, C, \mathcal{A}, q_{init}, T)$ и полагать, что формула φ имеет вид $\mathbf{E}\psi(\mathbf{E}\xi_1, \dots, \mathbf{E}\xi_m)$. *Проверочным автоматом* для пары (π, φ) называется такой конечный автомат $A(\pi, \varphi) = (Q_A, T, q^A_{init}, \delta_A, F_A)$ над алфавитом T переходов преобразователя π, что для любой конечной траектории $tr \in Fin\,Tr_\pi$ выполняется соотношение

$$\Gamma_\pi, v_{init}[tr] \models \varphi \iff tr \in A(\pi, \varphi).$$

Таким образом, построение автомата $A(\pi, \varphi)$ дает решение задачи верификации: $\pi \models \mathbf{E}\varphi$ тогда и только тогда, когда $\varepsilon \in A(\pi, \varphi)$. Верны следующие утверждения.

Теорема 1. *Существует алгоритм построения для любого конечного преобразователя π и формулы φ вида $\mathbf{E}\psi(\mathbf{E}\xi_1, \dots, \mathbf{E}\xi_m)$ проверочного автомата $A(\pi, \varphi)$ с использованием объема памяти $|\pi| \cdot 2^{\mathrm{poly}(|\varphi|)}$.*

Следствие 1. *Задача верификации конечных преобразователей относительно формул $Reg\text{-}CTL^*$ принадлежит классу сложности* ExpSpace.

Оставшаяся часть раздела посвящена доказательству теоремы 1.

Доказательство. Пусть $\pi = (Q, C, \mathcal{A}, q_{init}, T)$, а φ — формула состояния. Проведём доказательство теоремы индукцией по длине формулы φ.

Базис индукции состоит в доказательстве утверждения теоремы для случая $\varphi = P$, где P — базовый предикат. Заметим, что согласно определению семантики формул, $P \equiv \mathbf{E}P$. Поэтому мы будем считать, что $\varphi \equiv \mathbf{E}P$. Пусть $A_P = (Q_P, \mathcal{A}, p_{init}, \delta_P, F_P)$ — автомат, распознающий язык P. Положим

$$A(\pi, \mathbf{E}\varphi) = (Q \times Q_P, T, (q_{init}, p_{init}), v, Q \times F_P),$$

где функция переходов $\upsilon: Q \times Q_P \times T \longrightarrow Q \times Q_P$ такова, что для всех состояний $q \in Q$, $p \in Q_P$ и любого перехода преобразователя $\tau = (q, c, q', h) \in T$ выполняется равенство $\upsilon(q, p, \tau) = (q', \delta_P(p, h))$.

Базис индукции, таким образом, обоснован. Заметим, что множество состояний проверочного автомата $A(\pi, \varphi)$ имеет вид $Q \times Q_P$.

Перейдем теперь к обоснованию индуктивного перехода. Любая формула состояния является булевой комбинацией формул вида $\mathbf{E}\psi(\mathbf{E}\xi_1, \ldots, \mathbf{E}\xi_n)$, где $\psi(P_1, \ldots, P_n)$ — бесквантерная формула пути с базовыми предикатами $P_1, \ldots, P_n$. Достаточно, таким образом, рассмотреть случай $\varphi = \mathbf{E}\psi(\mathbf{E}\xi_1, \ldots, \mathbf{E}\xi_n)$. В силу отмеченной выше эквивалентности $P \equiv \mathbf{E}P$ все базовые предикаты P в формуле ψ можно заменить на $\mathbf{E}P$. Поэтому можно считать, что список $\mathbf{E}\xi_1, \ldots, \mathbf{E}\xi_n$ исчерпывает *все* подформулы состояния, встречающиеся в φ (не считая тех подформул состояния, которые являются подформулами $\xi_i, 1 \leqslant i \leqslant n$).

Предположение индукции, которое мы будем использовать, таково: для формул состояния $\psi_i, 1 \leqslant i \leqslant n$, существуют и могут быть эффективно построены проверочные автоматы $A(\pi, \xi_1), \ldots, A(\pi, \xi_n)$, причём для каждого $i, 1 \leqslant i \leqslant n$, множество состояний $A(\pi, \xi_i)$ имеет вид $Q \times Q_{P_{i_1}} \times \ldots \times Q_{P_{i_m}}$, где $P_{i_1}, \ldots, P_{i_m}$ — все базовые предикаты, встречающиеся в формуле ξ_i.

Каждый автомат $A(\pi, \xi_i)$ распознает язык над алфавитом T. Таким образом, формула $\mathbf{E}\xi_i$ является аналогом базового предиката над этим алфавитом. С этой точки зрения φ — бесквантерная формула, к которой можно применить алгоритм, разработанный в статье [5]. Этот алгоритм построит автомат Бюхи $B(\pi, \varphi)$, работающий в алфавите T и обладающий следующим свойством: для всякой конечной траектории $tr \in Fin\,Tr_\pi: \Gamma_\pi, \upsilon_{init}[tr] \models \varphi$ тогда и только тогда, когда у автомата $B(\pi, \varphi)$ есть допускающий прогон на бесконечной траектории tr_{inf}, для которой tr является префиксом. Мы приводим здесь краткое описание процедуры построения автомата $B(\pi, \varphi)$.

Замыканием Фишера-Ладнера формулы $\varphi = \mathbf{E}\psi(\mathbf{E}\xi_1, \ldots, \mathbf{E}\xi_n)$ называется минимальное по включению множество формул $\mathrm{FL}(\varphi)$, для которого выполняются следующие свойства:

- $\psi \in \mathrm{FL}(\varphi)$;
- если $\neg\zeta \in \mathrm{FL}(\varphi)$, то $\zeta \in \mathrm{FL}(\varphi)$;
- если $\zeta \in \mathrm{FL}(\varphi)$, то $\neg\zeta \in \mathrm{FL}(\varphi)$ (мы отождествляем $\neg\neg\zeta$ и ζ);
- если $\zeta \wedge \chi \in \mathrm{FL}(\varphi)$, то $\zeta, \chi \in \mathrm{FL}(\varphi)$;
- если $\mathbf{X}_c\zeta \in \mathrm{FL}(\varphi)$, то $\zeta \in \mathrm{FL}(\varphi)$;
- если $\zeta\,\mathbf{U}_L\chi \in \mathrm{FL}(\varphi)$, то $\zeta, \chi \in \mathrm{FL}(\varphi)$.

Мы добавляем к классическому определению замыкания ещё одно требование:

- если $\zeta\,\mathbf{U}_L\chi \in \mathrm{FL}(\varphi)$, то для всех $\tau = (q, c, q', h) \in T: \zeta\,\mathbf{U}_{L[c]}\chi \in \mathrm{FL}(\varphi)$.

Множество формул $K \subseteq \mathrm{FL}(\varphi)$ называется *атомом*, если:

- K непротиворечиво с точки зрения пропозициональной логики, т.е.
  - если $\xi \wedge \chi \in K$, то $\xi \in K$ и $\chi \in K$;
  - $\xi \in K$ тогда и только тогда, когда $\neg\xi \notin K$;
  - если $\mathrm{true} \in \mathrm{FL}(\varphi)$, то $\mathrm{true} \in K$;
- K подчиняется семантике оператора $\mathbf{U}$, то есть для всех $\zeta\,\mathbf{U}_L\chi \in \mathrm{FL}(\varphi)$
  - если $\chi \in K$ и $\varepsilon \in L$ то $\zeta\,\mathbf{U}_L\chi \in K$;
  - если $\zeta\,\mathbf{U}_L\chi \in K$, и $\chi \notin K$, и $\varepsilon \in L$, то $\zeta \in K$.

Атом K, содержащий формулу ψ, называется *начальным*. Далее под формулами будут подразумеваться элементы множества $\mathrm{FL}(\varphi)$.

Пусть $\{P_i\}_{i=1}^m$ — это все базовые предикаты, встречающиеся в формуле $\varphi = \mathbf{E}\psi(\mathbf{E}\varphi_1, \ldots, \mathbf{E}\varphi_n)$, и пусть им соответствуют автоматы $A_{P_i} = (Q_{P_i}, \mathcal{A}, p_i^{init}, \delta_{P_i}, F_{P_i})$. Каждый базовый предикат встречается в одной из формул ξ_i. По предположению индукции для каждой из этих формул существует проверочный автомат $A(\pi, \xi_i) = (Q_i, T, q_i^{init}, \delta_i, F_i),\ 1 \leqslant i \leqslant n$. Каждое состояние автомата $A(\pi, \xi_i)$ представляет собой кортеж вида $(q, p_{j_1}, \ldots, p_{j_r}), q \in Q, p_{j_k} \in Q_{P_{j_k}}$. Это состояние можно получить из кортежа

$(q, p_1, \dots, p_m), p_j \in Q_{P_j}$, опустив все те элементы p_j, которые соответствуют базовым предикатам P_j, не встречающимся в формуле в ξ_i. Будем обозначать полученное таким образом состояние записью $q_i(q, p_1, \dots, p_m)$.

Опишем теперь конструкцию автомата Бюхи $B(\pi, \varphi)$. Каждое состояние этого автомата содержит состояние преобразователя π, набор состояний автоматов A_{P_i} и атом $K \subseteq \mathrm{FL}(\varphi)$. Устройство автомата $B(\pi, \varphi) = (S, T, \varnothing, \Delta, \mathcal{F})$ таково:

- $S = \left\{ \langle q, p_1, \dots, p_m, K \rangle \mid q \in Q, p_j \in Q_{P_j}, \text{а } K - \text{атом} \right\}$;
- $\mathcal{F} = \{F_{\alpha \mathbf{U}_L \beta} \mid \alpha \mathbf{U}_L \beta \in \mathrm{FL}(\varphi)\}$, где $F_{\alpha \mathbf{U}_L \beta}$ — множество всех тех состояний, которые удовлетворяют следующему требованию: для всех сдвигов $L' = L[x]$, где $x \in C^*$, если $\alpha \mathbf{U}_L \beta \in K$, то $\beta \in K$ и $\varepsilon \in L'$.
- отношение $\Delta \subseteq S \times T \times S$ подчиняется следующему требованию: для всякого состояния $s = \langle q, p_1, \dots, p_n, K \rangle \in S$ и перехода $\tau = (q, c, q', h) \in T$:
  - если $\mathbf{E}\varphi_i \in K$ для некоторого $i, 1 \leqslant i \leqslant n$, причем $q_i(s) \notin F_i$, или если существует $\mathbf{X}_a \xi \in K$, для которого $a \neq c$, то *не существует ни одного перехода* из s по τ, т.е. $(s, \tau, s') \notin \Delta$ для любого состояния $s' \in S$;
  - если для всех формул $\mathbf{E}\varphi_i \in K$ выполняется $q_i(s) \in F_i$, и для всех $\mathbf{X}_a \xi \in K$ выполняется $a = c$, то $(s, \tau, s') \in \Delta$ для каждого такого состояния $s' = \langle q', p'_1, \dots, p'_m, K' \rangle \in S$, которое удовлетворяет набору условий:
    1. $p'_i = \delta_{P_i}(p_i, h), 1 \leqslant i \leqslant n$;
    2. для всех $\mathbf{X}_c \xi : \mathbf{X}_c \xi \in K$ если и только если $\xi \in K'$;
    3. для всех $\zeta \mathbf{U}_L \chi$: $\zeta \mathbf{U}_L \chi \in K$ тогда и только тогда, когда либо $\varepsilon \in L$ и $\chi \in K$, либо $(\zeta \mathbf{U}_{L[c]} \chi) \in K'$ и при условии $\varepsilon \in L$ верно, что $\zeta \in K$.

Приведенная здесь конструкция автомата Бюхи $B(\pi, \varphi)$ в общих чертах воспроизводит известное построение автомата Бюхи по формуле линейной темпоральной логики LTL, описанное [7], с той лишь разницей, что при работе с формулами языка $Reg\text{-}CTL^*$ используются дополнительные конечные автоматы для базовых предикатов, а также проверочные автоматы для подформул состояния. И поэтому нетрудно заметить, что, следуя тому же ходу рассуждений с применением индукции по длине формулы φ и длине траектории $tr \in Fin\, Tr_\pi$, который подробно описан в книге [7], можно доказать следующее вспомогательное утверждение:

Лемма 1. *Для любого конечного преобразователя π, всякой формулы состояния $\varphi = \mathbf{E}\psi(\mathbf{E}\varphi_1, \dots, \mathbf{E}\varphi_n)$ и произвольной конечной траектории $tr \in Fin\, Tr_\pi$ выполняется соотношение: $\Gamma_\pi, q_{init}[tr] \models \varphi$ тогда и только тогда, когда автомат Бюхи $B(\pi, \varphi)$ имеет допускающий прогон из состояния $\langle q_{init}[tr], p_1^{init}[tr], \dots, p_m^{init}[tr], K \rangle$, для которого $\psi \in K$.*

Опираясь на лемму 1, можно предложить следующий способ построения проверочного автомата $A(\pi, \varphi)$. Этот автомат должен иметь следующее устройство $A(\pi, \varphi) = (Q_A, T, q_A^{init}, \delta_A, F_A)$, где

- $Q_A = Q \times Q_{P_1} \times \dots \times Q_{P_m}$;
- $q_A^{init} = (q_{init}, p_1^{init}, \dots, p_m^{init})$;
- значение функции переходов $\delta_A : Q_A \times T \longrightarrow Q_A$ определяется следующим образом: для любого состояния $q_A = (q', p_1, \dots, p_m)$ и перехода $\tau = (q', c, q'', h)$

$$\delta_A(q_A, \tau) \quad = \quad (q'', \delta_{P_1}(p_1, \tau), \dots, \delta_{P_m}(p_m, \tau));$$

- множество $F_A \subseteq Q_A$ состоит из всех состояний $q_A = (q, p_1, \dots, p_m)$, для которых существуют начальный атом $K \subseteq \mathrm{FL}(\varphi)$ и допускающий прогон $B(\pi, \varphi)$ из состояния $\langle q, p_1, \dots, p_m, K \rangle$.

Согласно лемме 1 автомат $A(\pi, \varphi)$ удовлетворяет определению проверочного автомата:

$$\Gamma_\pi, v_{init}[tr] \models \varphi \iff tr \in A(\pi, \varphi).$$

Оценим размер полученного проверочного автомата. Как видно из описания устройства проверочного автомата, его размер можно оценить сверху

$$|A(\pi,\varphi)| = \mathrm{poly}(|\pi| \cdot \prod_{i=1}^{m} |P_i|) \lesssim |\pi| \cdot 2^{\mathrm{poly}(|\varphi|)}.$$

Для вычисления множества принимающих состояний автомата $A(\pi,\varphi)$ достаточно для каждого состояния q_A проверить, существует ли такой начальный атом $K \subseteq \mathrm{FL}(\varphi)$, что автомат $B(\pi,\varphi)$ имеет допускающий прогон из (q_A, K).

Поскольку $|K| \leqslant |\mathrm{FL}(\varphi)| \leqslant 2^{\mathrm{poly}(|\varphi|)}$, для хранения описания одного состояния автомата $B(\pi,\varphi)$ требуется объем памяти, не превышающий величины $2^{\mathrm{poly}(|\varphi|)}$. Таким образом, существование допускающего прогона может быть проверено *недетерминированно* переборным поиском с использованием объема памяти, ограниченного величиной $2^{\mathrm{poly}(|\varphi|)}$. По теореме Савича [21] детерминизация алгоритма поиска приводит не более чем к квадратичному увеличению объема используемой памяти. Поэтому сложность по объему памяти предложенного алгоритма проверки выполнимости $\pi \vDash \varphi$ ограничена сверху величиной $|\pi| 2^{\mathrm{poly}(|\varphi|)}$. □

Нижняя оценка сложности задачи верификации

Теорема 2. *Задача верификации конечных преобразователей относительно формул логики Reg-CTL* является* PSpace*-трудной.*

Доказательство. Покажем, как можно свести известную (см. [22]) PSpace-полную проблему проверки непустоты пересечения языков, распознаваемых семейством конечных автоматов, к указанной задаче верификации. Пусть задано семейство конечных автоматов $A_1, A_2, \ldots, A_n$ над алфавитом Σ. Рассмотрим преобразователь $\pi = (q, \Sigma, \Sigma, q, T)$ с одним состоянием управления и множеством переходов $T = \{(q,\sigma,\sigma,q) : \sigma \in \Sigma\}$. Будем использовать автоматы A_i для представления базовых предикатов (регулярных языков) в формуле $\varphi = \mathbf{EF}_{\Sigma^*} \bigwedge_{i=1}^{n} A_i$. Эта формула выражает следующее требование: из начальной вершины графа Γ_π достижима вершина, в которой истинны все базовые предикаты $A_1, \ldots, A_n$. Таким образом, имеет место равносильность

$$\bigcap_{i=1}^{n} A_i \neq \varnothing \quad \Longleftrightarrow \quad \pi \vDash \mathbf{EF}_{\Sigma^*} \bigwedge_{i=1}^{n} A_i,$$

показывающая, что для проверки непустоты пересечения языков, распознаваемых семейством автоматов $A_1, A_2, \ldots, A_n$, достаточно проверить соотношение $\pi \vDash \varphi$. □

4. Заключение

В этой статье мы определили синтаксис и семантику нового расширения *Reg-CTL** темпоральной логики *CTL**, которую можно использовать для спецификаций поведения последовательных реагирующих систем. Для новой темпоральной логики была исследована и решена задача верификации моделей конечных автоматов-преобразователей. Решение получено с помощью теоретико-автоматного метода, следуя схемам построения алгоритмов верификации автоматов-преобразователей, предложенных в работах [5, 8] для других расширений темпоральных логик.

Описанный здесь алгоритм верификации автоматов-преобразователей для логики *Reg-CTL** имеет значительно большую сложность по объему памяти, нежели аналогичный алгоритм для *CTL** (см. [7]). Однако нам удалось показать, что рассматриваемая нами задача верификации является PSpace-трудной. Таким образом, пока существует экспоненциальный разрыв между верхней и нижней оценками сложности этой задачи. Мы рассчитываем построить более эффективный (полиномиальный по памяти) алгоритм верификации, основанный на результатах статьи [16], где

исследовали свойства логики, семантически близкие рассматриваемой нами логике *Reg-LTL*. Эти результаты дают основание предполагать, что задача верификации для *Reg-CTL**, по-видимому, является PSpace-полной. Доказательство этого предположения является одной из целей дальнейших исследований в области верификации реагирующих систем.

Разумеется, интересно также проверить практическую применимость предложенного в статьях [4—6, 8] формализма для спецификации поведения последовательных реагирующих систем. По-видимому, для этой цели можно обратиться к задаче верификации программируемых логических контроллеров (ПЛК) [23, 24] и сравнить выразительные возможности предложенного нами расширения темпоральных логик с другими формальными средствами спецификации и верификации ПЛК [25—28].

Авторы выражают признательность рецензенту за ценные замечания, позволившие улучшить облик этой статьи.

References

[1] R. Alur and P. Cerny, «Streaming transducers for algorithmic verification of single-pass list-processing programs», in *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2011, pp. 599–610.

[2] Q. Hu and L. D'Antoni, «Automatic program inversion using symbolic transducers», in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 376–389.

[3] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner, «Symbolic finite state transducers: Algorithms and applications», in *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2012, pp. 137–150.

[4] A. R. Gnatenko and V. A. Zakharov, «On the Expressive Power of Some Extensions of Linear Temporal Logic», *Automatic Control and Computer Sciences*, vol. 53, no. 7, pp. 663–675, 2019.

[5] D. G. Zakharov, V. A. Kozlova, and D. Kozlova, «On the model checking of sequential reactive systems», in *Proceedings of the 25-th International Workshop on Concurrency, Specification and Programming, Rostock, Germany, September 28-30*, vol. 1698, 2016, pp. 233–244.

[6] A. R. Gnatenko and V. A. Zakharov, «On the model checking of finite state transducers over semigroups», *Proceedings of ISP RAS*, vol. 30, no. 3, pp. 303–324, 2018.

[7] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model checking*. MIT press, 1999.

[8] A. R. Gnatenko, «On the complexity of model checking problem for finite state transducers over free semigroups», in *Proceedings of the Student Session of European Summer School on Logic, Language and Information, Riga*, 2019.

[9] J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[10] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi, «On the temporal analysis of fairness», in *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1980, pp. 163–173.

[11] Z. Manna and P. Wolper, «Synthesis of communicating processes from temporal logic specifications», in *Workshop on Logic of Programs*, Springer, vol. 131, 1981, pp. 253–281.

[12] P. Wolper, «Temporal Logic Can Be More Expressive», *Information and control*, vol. 56, no. 1-2, pp. 72–99, 1983.

[13] O. Kupferman, N. Piterman, and M. Y. Vardi, «Extended temporal logic revisited», in *Proceedings of 12-th International Conference on Concurrency Theory*, Springer, 2001, pp. 519–535.

[14] M. Y. Vardi and P. Wolper, «Yet another process logic», in *Lecture Notes in Computer Science*, Springer, vol. 14, 1983, pp. 501–512.

[15] M. Y. Vardi, «A Temporal Fixpoint Calculus», in *Proceedings of the 15-th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1988, pp. 250–259.

[16] J. G. Henriksen and P. S. Thiagarajan, «Dynamic linear time temporal logic», *Annals of Pure and Applied logic*, vol. 96, no. 1-3, pp. 187–207, 1999.

[17] M. Leucker and C. Sanchez, «Regular Linear Temporal Logic», in *Proceedings of the 4-th International colloquium on theoretical aspects of computing*, Springer, 2007, pp. 291–305.

[18] R. Mateescu, P. T. Monteiro, E. Dumas, and H. De Jong, «CTRL: Extension of CTL with regular expressions and fairness operators to verify genetic regulatory networks», *Theoretical Computer Science*, vol. 412, no. 26, pp. 2854–2883, 2011.

[19] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper, «Simple on-the-fly automatic verification of linear temporal logic», in *International Conference on Protocol Specification, Testing and Verification*, Springer, 1995, pp. 3–18.

[20] M. Y. Vardi and P. Wolper, «An automata-theoretic approach to automatic program verification», in *Proceedings of the First Symposium on Logic in Computer Science*, IEEE Computer Society, 1986, pp. 322–331.

[21] W. J. Savitch, «Relationships between nondeterministic and deterministic tape complexities», *Journal of computer and system sciences*, vol. 4, no. 2, pp. 177–192, 1970.

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

[23] A. Mader, «A classification of PLC models and applications», in *Discrete Event Systems*, vol. 569, Springer, 2000, pp. 239–246.

[24] T. Ovatman, A. Aral, D. Polat, and A. O. Unver, «An overview of model checking practices on verification of PLC software», *Software & Systems Modeling*, vol. 15, no. 4, pp. 937–960, 2016.

[25] N. Garanina, I. Anureev, V. Zyubin, A. Rozov, T. Liakh, and S. Gorlatch, «Reasoning about Programmable Logic Controllers», *System Informatics*, vol. 17, pp. 33–42, 2020.

[26] E. V. Kuzmin, D. A. Ryabukhin, and V. A. Sokolov, «On the expressiveness of the approach to constructing PLC-programs by LTL-specification», *Automatic Control and Computer Sciences*, vol. 50, no. 7, pp. 510–519, 2016.

[27] O. Ljungkrantz, K. Akesson, M. Fabian, and C. Yuan, «A formal specification language for PLC-based control logic», in *Proceedings of the 8th IEEE International Conference on Industrial Informatics*, IEEE, 2010, pp. 1067–1072.

[28] O. Ljungkrantz, K. Akesson, M. Fabian, and A. H. Ebrahimi, «An empirical study of control logic specifications for programmable logic controllers», *Empirical Software Engineering*, vol. 19, no. 3, pp. 655–677, 2014.

MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 27, NO. 4, 2020
journal homepage: www.mais-journal.ru

THEORY OF COMPUTING

Knowledge-based Algorithms for BDI-agents

N. V. Shilov[1], N. O. Garanina[2]

DOI: 10.18255/1818-1015-2020-4-442-453

[1]Innopolis University, 1 Universitetskaya, Innopolis, 420500, Russia.
[2]A.P. Ershov Institute of Informatics Systems (IIS), Siberian Branch of the Russian Academy of Sciences, 6 Acad. Lavrentjev ave., Novosibirsk 630090, Russia.

MSC2020: 93A16
Research article
Full text in Russian

Received November 20, 2020
After revision December 5, 2020
Accepted December 16, 2020

Multiagent algorithm is a knowledge-based distributed algorithm that solves some problems by means of cooperative work of agents. From an individual agent's perspective, a multiagent algorithm is a reactive and proactive knowledge/believe-based rational algorithm aimed to achieve an agent's own desires. In the paper we study a couple of knowledge-based multiagent algorithms. One particular algorithm is for a system consisting of agents that arrive one by one (in a non-deterministic order) to a resource center to rent (for a while) one of available desired resources. Available resources are passive, they form a cloud; each of the available resources is lent on demand if there is no race for this resource and returns to the cloud after use. Agents also form a cloud but leave the cloud immediately when they rent a desired resource. The problem is to design a knowledge-based multiagent algorithm, which allows each arriving agent eventually to rent some of desired resources (without race for these resources).

Keywords: multiagent systems; multiagent algorithms; BDI-agents; knowledge and belief

INFORMATION ABOUT THE AUTHORS

Nikolay Vyacheslavovich Shilov correspondence author	orcid.org/0000-0001-7515-9647. E-mail: shiloviis@mail.ru Ph.D. in Mathematics, head of laboratory.
Natalia Olegovna Garanina correspondence author	orcid.org/0000-0001-9734-3808. E-mail: garanina@iis.nsk.su Ph.D. in Mathematics, senior research fellow.

Funding: This research has been financially supported by the Ministry of Digital Development, Communications and Mass Media of the Russian Federation and Russian Venture Company (Agreement No. 004/20 dd. 20.03.2020, IGK 0000000007119P190002.

For citation: N. V. Shilov and N. O. Garanina, "Knowledge-based Algorithms for BDI-agents", *Modeling and analysis of information systems*, vol. 27, no. 4, pp. 442-453, 2020.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 27, № 4, 2020
сайт журнала: www.mais-journal.ru

THEORY OF COMPUTING

Алгоритмы для BDI-агентов, основанные на знаниях

Н. В. Шилов[1], Н. О. Гаранина[2]

DOI: 10.18255/1818-1015-2020-4-442-453

[1]Университет Иннополис, Университетская, д.1, г. Иннополис, 420500 Россия.
[2]Институт систем информатики имени А. П. Ершова СО РАН, пр. Лаврентьева, д. 6, г. Новосибирск, 630090 Россия.

УДК 004.8
Научная статья
Полный текст на русском языке

Получена 20 ноября 2020 г.
После доработки 5 декабря 2020 г.
Принята к публикации 16 декабря 2020 г.

Мультиагентный алгоритм — это распределённый алгоритм, основанный на знаниях, который решает некоторую проблему посредством совместной работы агентов. BDI-агент — это агент, обладающий убеждениями (Belief), желаниями (Desire) и намерениями (Intention). С точки зрения такого агента, мультиагентный алгоритм — это алгоритм, основанный на его знаниях и убеждениях, с помощью которого достигается выполнение его желаний посредством последовательного осуществления намерений. Мы считаем также, что агенты реактивны, проактивны и рациональны. В этой статье мы предлагаем и изучаем два мультиагентных алгоритма, которые основаны на знаниях. В частности, мы предлагаем мультиагентный алгоритм для следующей задачи аренды ресурсов. Система состоит из агентов, которые прибывают один за другим в произвольном порядке в ресурсный центр, чтобы арендовать один из предоставляемых ресурсов. Предоставляемые ресурсы пассивны, они образуют облако. Если за ресурс нет конкуренции, то он предоставляется по запросу, и возвращается в облако после использования. Агенты также образуют облако, но когда арендуют нужный ресурс, то сразу же покидают ресурсный центр. Задача состоит в разработке мультиагентного алгоритма, основанного на знаниях, обладающего следующим свойством корректности: каждый прибывающий в ресурсный центр агент рано или поздно арендует какой-либо из запрашиваемых ресурсов без конкуренции за этот ресурс в данный момент.

Ключевые слова: мультиагентные системы; мультиагентные алгоритмы; BDI-агенты; знания и мнения

ИНФОРМАЦИЯ ОБ АВТОРАХ

Николай Вячеславович Шилов	orcid.org/0000-0001-7515-9647. E-mail: shiloviis@mail.ru
автор для корреспонденции	канд. физ.-мат. наук, нач. лаб.
Наталья Олеговна Гаранина	orcid.org/0000-0001-9734-3808. E-mail: garanina@iis.nsk.su
автор для корреспонденции	канд. физ.-мат. наук, с.н.с.

Финансирование: Исследование выполнено при финансовой поддержке Министерства цифрового развития, связи и массовых коммуникаций РФ и АО «Российская венчурная компания» (договор №004/20 от 20.03.2020, ИГК 0000000007119P190002).

Для цитирования: N. V. Shilov and N. O. Garanina, “Knowledge-based Algorithms for BDI-agents”, *Modeling and analysis of information systems*, vol. 27, no. 4, pp. 442-453, 2020.

Введение: основные понятия

Термины “мультиагентный” и “знание” относятся к нескольким (иногда не связанным) концепциям, подходам и парадигмам исследований в теоретическом программировании, искусственном интеллекте, программной инженерии, анализе данных, информационных технологиях и т.д. В противоположность этому, понятие “облако” обычно используется лишь в контексте информационных технологий и программной инженерии, а со стороны искусственного интеллекта ему уделено незначительное внимание. В этой статье мы изучаем основанные на знаниях алгоритмы для системы BDI-агентов, которые запрашивают ресурсы, образующие облако.

Распределённая система [1, 2] состоит из нескольких автономных отдельных “компьютеров” (программ с распределённой памятью), которые обмениваются данными через сеть. Коммуникация в распределённой системе считается *справедливой*, если каждый компьютер, которому необходимо взаимодействовать с каким-либо другим компьютером, рано или поздно будет с ним взаимодействовать. Для обеспечения справедливости, очевидно, требуется планировщик или специальный механизм коммуникаций.

Мультиагентная система — это распределённая система, состоящая из агентов. *BDI-агент* [3] (далее просто *агент*) — это автономный реактивный и проактивный объект (в объектно-ориентированном смысле), внутренние состояния которого могут быть охарактеризованы в терминах убеждений (мнений) (B), желаний (D), и намерений (I).

Убеждения агента представляют его идеи/мнения о себе и окружении, т.е. других агентах и сети; эти убеждения могут быть неверными, неполными и даже несогласованными. *Желания* агента представляют собой его долгосрочные цели, обязательства и задачи, которые также могут быть противоречивыми. *Намерения* агента характеризуют его краткосрочное планирование. *Логическое всеведение* агента означает, что агент сразу знает все логические следствия, вытекающие из его знаний.

Мы различаем убеждение и знание согласно Платону, который полагал, что *знание* — это истинное мнение, имеющее подтверждение [4, 5]. В настоящей статье мы считаем, что убеждение агента становится знанием, если существует формальное доказательство истинности этого убеждения. Мы предполагаем логическое всеведение агента, и поэтому для того, чтобы убеждение стало знанием, достаточно истинности этого убеждения.

Реактивность агента означает, что он может изменить свои убеждения после общения и взаимодействия с другими агентами. *Проактивность* агента означает, что он может изменить свои намерения после изменения/обновления своих убеждений, т. е. планировать своё поведение в ближайшем будущем на основе имеющихся данных.

Каждый агент *автономен*, то есть изменение его личных убеждений, желаний и намерений не может быть предписано каким-либо другим агентом. *Рациональный* агент имеет чёткие предпочтения и всегда из возможных действий выбирает то, которое приводит к наилучшему личному результату.

Мультиагентный алгоритм — это распределённый алгоритм (протокол распределённой системы), который решает некоторую задачу посредством совместной работы агентов в мультиагентной системе. Родственной парадигмой в экономической науке является дизайн механизмов [6].

Отказоустойчивость мультиагентного алгоритма — это способность правильно решить задачу, несмотря на возможный сбой сети и/или некорректное поведение отдельных агентов.

Алгоритм, основанный на знаниях [7], — это алгоритм, в котором агент предпринимает действие только тогда, когда “знает”, что действие можно/нужно выполнить. В нашем случае это означает, что действие с общим ресурсом осуществляется только тогда, когда агент знает, что доступ к этому ресурсу является безопасным, т.е. при отсутствии конкуренции (гонок) за этот ресурс.

Классический пример мультиагентного алгоритма, основанного на знаниях — решение задачи о разрезании торта [3]. Ещё один популярный пример мультиагентного алгоритма, основанного на знаниях — решение головоломки о чумазых ребятишках [7].

Далее в разделах 1 и 2 мы предложим и изучим два новых (насколько нам известно) мультиагентных алгоритма, основанных на знаниях. Второй из этих алгоритмов (см. раздел 2) решает задачу о доступе к ресурсу в *облаке ресурсов*. Поэтому прежде, чем перейти к обсуждению алгоритмов, опишем, что такое *облачные вычисления*.

Согласно [8], "облачные вычисления — это модель для обеспечения повсеместного, удобного сетевого доступа по требованию к общему набору конфигурируемых вычислительных ресурсов (например, сетей, серверов, хранилищ, приложений и услуг), которые могут быть быстро заняты и освобождены с минимальными усилиями руководства или взаимодействия с поставщиком услуг". Определение включает пять основных характеристик, три модели обслуживания и четыре модели развертывания. Перечислим эти характеристики и модели.

- Характеристики: самообслуживание по запросу, широкий доступ к сети, пул ресурсов, масштабируемость (rapid elasticity), измеряемый сервис.
- Модели обслуживания: программное обеспечение как услуга (Software as a Service, SaaS), платформа как услуга (Platform as a Service, PaaS), инфраструктура как услуга (Infrastructure as a Service, IaaS).

1. Задача о хосте и P2P-серверах

В этом разделе мы рассмотрим задачу и мультиагентный алгоритм (протокол), которая до некоторой степени является модификацией задачи о чумазых ребятишках. Архитектура сети для этой задачи представлена на рисунке 1.

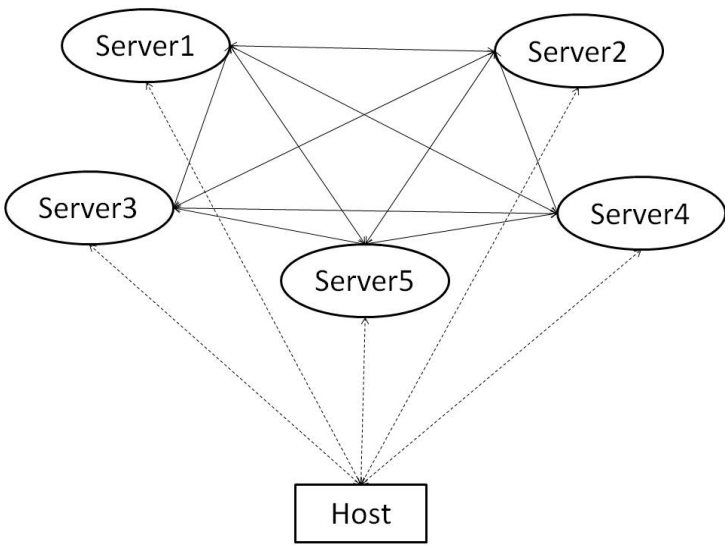


Fig. 1. Network for the problem of host and P2P-servers

Рис. 1. Архитектура сети для задачи о хосте и P2P-серверах

Постановка задачи

Система состоит из клиентского компьютера-хоста и n серверов, соединённых сетью. Хост соединяется с каждым сервером по выделенному каналу, и, кроме того, серверы связаны друг с другом в одноранговую сеть. Все соединения надёжны и мгновенны, хост подключён к каждому серверу напрямую, а каждый сервер имеет коммутатор, через который он подключается к одноранговой

сети. Таким образом, каждый сервер имеет свой индивидуальный коммутатор с $(n-1)$ сокетами для отдельных каналов к каждому из остальных $(n-1)$ серверов. По сети могут пересылаться оригинальные *сообщения* и *подтверждения* о получении.

Исправный коммутатор предоставляет своему владельцу-серверу i ($1 \le i \le n$), следующие два сервиса. Пусть j — это номер другого сервера: $1 \le j \le n$ и $j \ne i$.

1. Если владелец отправляет исходящее сообщение какому-либо серверу j, то коммутатор выбирает подходящий канал, пересылает сообщение, ожидает подтверждения от получателя j и затем уведомляет владельца о подтверждении.
2. Если коммутатор получает входящее сообщение или подтверждение от какого-либо сервера j, то он передаёт сообщение своему серверу-владельцу и, если было получено сообщение, то он высылает подтверждение отправителю сообщения серверу j.

Мы рассматриваем случай, когда коммутаторы ненадёжны: коммутатор всегда может предоставить первый сервис по отправке сообщений, но может раз и навсегда отказать в предоставлении второго сервиса для всех входящих сообщений и подтверждений со всех других серверов, т.е. перестать передавать полученное сообщение/подтверждение своему серверу-владельцу и посылать подтверждение серверу, отправившему сообщение. Напомним, связь с хостом осуществляется по специальному каналу без коммутатора, поэтому сбой коммутатора не может прервать связь с хостом.

У серверов нет аппаратных средств для проверки исправности своих коммутаторов. Кроме того, неясно, как сервер может проверять работоспособность своего коммутатора логически на основе анализа входящего трафика, так как следующие случаи отсутствия коммуникации неразличимы с поломкой коммутатора:

- если нет входящих сообщений, то, возможно, другим серверам нечего отправлять;
- если нет подтверждений для исходящих сообщений, то, возможно, вышли из строя коммутаторы всех остальных серверов.

Предположим, что в какой-то момент некоторые коммутаторы неисправны в описанном выше смысле, и множество таких коммутаторов не меняется.

Задача: разработать основанный на знаниях мультиагентный алгоритм для системы хоста и серверов, гарантирующий следующие свойства прогресса и безопасности [9].

1. Прогресс: каждый сервер рано или поздно узнает об исправности или неисправности своего коммутатора.
2. Безопасность: каждый сервер сообщит хосту истинное состояние (исправен или неисправен) своего коммутатора.

Решение, основанное на синхронизации

Введём следующие обозначения для временных интервалов:

- *SCR* (Server Communication Round) — это время, которое достаточно каждому серверу для отправки сообщений другим серверам (не более чем по одному сообщению), для ожидания получения подтверждений и для выполнения внутренней обработки данных (deliberation).
- *HSR* (Host-Server Round) — это время, которое достаточно хосту для объявления всем серверам (рассылки сообщений), для выполнения внутренней обработки данных серверами и хостом, и для приёма отдельных сообщений от серверов (не более чем по одному сообщению).

Задача может быть решена с помощью следующего протокола, аналогичного алгоритму задачи о чумазых ребятишках [7]. Протокол состоит из двух частей: клиентской части для исполнения на хосте и серверной части для исполнения на каждом сервере.

Клиентская часть

1. Сначала создать переменную для хранения списка отчётов от серверов и инициализировать ее пустым списком, а затем дождаться окончания временного интервала $HSR + SCR$.

2. Пока пуст список отчётов от серверов повторять следующий цикл:
   сначала разослать новый запрос отчётов всем серверам, а затем собрать в переменной для хранения отчётов все полученные отчёты с серверов, которые поступят до истечения очередного временного интервала HSR.
3. Пополнить переменную для хранения отчётов всеми отчётами, которые поступят от серверов до истечения следующего временного интервала HSR, а затем остановиться (завершить работу алгоритма).

Серверная часть

1. Отправить всем серверам сообщение, а затем подсчитать в своей приватной переменной FS количество коммутаторов других серверов, от которых не пришло подтверждения до истечения временного интервала $HSR + SRT$.
2. Пока $FS > 0$ повторить следующий цикл:
   если до истечения временного интервала HSR получен новый запрос об отчёте от хоста,
   то уменьшить значение своей приватной переменной FS и дождаться окончания текущего временного интервала HSR
   иначе — отправить отчёт на хост о неисправности собственного коммутатора, выйти из цикла (break) и остановиться (завершить работу алгоритма).
3. Когда $FS = 0$ и в течение следующего интервала времени HSR поступает новый запрос от хоста о предоставлении отчёта, то отправить отчёт на хост об исправности собственного коммутатора и остановиться (завершить работу алгоритма).

Корректность и анализ решения

Утверждение 1. *Предположим, что в системе все агенты (хост и серверы) начинают работать одновременно (синхронно) и в системе есть хотя бы два исправных коммутатора. Тогда мультиагентный алгоритм, основанный на синхронизации, гарантирует свойства прогресса и безопасности.*

Идея доказательства.
Мы считаем, что хост и все серверы являются BDI-агентами. У каждого агента есть желание узнать, правильно ли работает его коммутатор или вышел из строя. Пусть m — количество неисправных коммутаторов.

Прежде всего отметим, что временные интервалы играют роль барьеров синхронизации, т.е. первый шаг хоста завершается одновременно с первым шагом каждого сервера, и все итерации цикла 2 хоста и цикла 2 каждого сервера также завершаются одновременно.

После начала работы каждый сервер i ($1 \le i \le n$) подсчитывает в своей приватной переменной FS количество m_i других серверов, от которых не пришло подтверждение; с этого момента сервер i считает, что m_i — это количество сломанных коммутаторов. Заметим, что для любого $1 \le i \le n$, если коммутатор сервера i исправен, то $m_i = m \le (n - 2)$, а если неисправен, то $m_i = (n - 1)$. (Именно здесь используется то, что как минимум два коммутатора исправны: иначе у всех серверов значение приватной переменной FS будет $(n - 1)$.)

Поэтому хост и все серверы (как с исправными, так и с неисправными коммутаторами) синхронно исполняют первые m итераций своих циклов 2. Так как во время этих первых m итераций хост не получал ни одного отчёта, то он продолжает итерации этого цикла и отправляет ещё один запрос всем серверам. Но у серверов с исправными коммутаторами в это время значение приватной переменной FS равно 0, следовательно они уже вышли из своего цикла 2 и в ответ на очередной запрос хоста отправляют отчёт об исправности своих коммутаторов (шаг 3). Хост после получения отчётов от (как минимум 2) серверов с исправными коммутаторами выходит из своего цикла 2 и не посылает запросов серверам на следующем интервале времени HSR. В свою очередь, серверы с неисправными коммутаторами во время этого временного интервала все ещё выполняют свой

цикл 2, значения их приватных переменных FS все ещё больше 0, и так как они не получают запрос от хоста, то отправляют отчёт о неисправности своих коммутаторов.

Оценим временную сложность алгоритма, основанного на синхронизации. Запросы отчётов будут отправлены хостом m раз; поскольку время между последовательными запросами составляет HSR, тогда цикл 2 хоста требует время $m \times HSR$. Принимая во внимание шаги 1 и 3 хоста, общее время равно $(m+2) \times HSR + SRT$. Таким образом, сложность линейно зависит от общего числа неисправных коммутаторов.

Отметим, что основная проблема (которую мы пока не знаем как решить) алгоритма, основанного на синхронизации, — отказоустойчивость: если какой-либо из серверов не будет исполнять предписанный алгоритм, результат работы будет неверным.

Отметим также, что синхронизация является существенным элементом нашего решения как и в задаче о чумазых ребятишках: в задаче о чумазых ребятишках роль синхронизации выполняют запросы отца к детям. К сожалению, нам не известны какие-либо решения задачи без синхронизации, сохраняющие автономность и приватность агентов.

2. Задача о ресурсном центре

Постановка задачи

Ресурсный центр состоит из

- "облака" из $m > 0$ разных неделимых единичных ресурсов для аренды,
- "облака" из $n > 0$ агентов, которым нужны единичные ресурсы в аренду, и
- монитора.

Монитор ресурсного центра

- обеспечивает агентов данными, которые агенты могут использовать для разрешения конфликтов (например, мгновенно регистрирует участников и маркирует их отметками времени по прибытии в центр, т. е. работает как электронная очередь);
- мгновенно назначает (сдаёт в аренду) доступный ресурс агенту, если он в данный момент является *единственным* агентом, запрашивающим этот ресурс, но не назначает ресурс никому, если в данный момент два или более агента запрашивают этот ресурс;
- в случае конкуренции за ресурс регистрация новых участников прекращается до тех пор, пока ресурс не будет сдан в аренду;
- ведёт реестр ресурсов доступных для аренды (в частности, освобождённых после аренды).

Агенты прибывают в ресурсный центр в произвольном асинхронном порядке (и никогда одновременно). Каждый агент намерен арендовать на время желаемый и доступный ресурс, а затем когда-нибудь возвратить его в центр. У каждого агента есть собственный индивидуальный приватный (не подлежащий публичному объявлению) список желаемых ресурсов. Агенты общаются друг с другом через надёжную P2P-сеть с мгновенным соединением.

Индивидуальные списки желаемых ресурсов разных агентов могут пересекаться. Каждый список отсортирован в соответствии с индивидуальными приоритетами агента. Любой элемент в списке может удовлетворить текущие потребности агента, т. о. агенту нужны не все элементы, а только один. Список желаний может измениться, если агент в данный момент не зарегистрирован монитором ресурсного центра.

Наш центр ресурсов, как облако, абстрагируется от моделей обслуживания, но в значительной степени обладает описанными в разделе облачными характеристиками.

- Самообслуживание по запросу: агенты могут пользоваться ресурсами, самостоятельно разрешая возникающие конфликты.
- Широкий доступ к сети: в текущей постановке задачи эта характеристика несущественна.
- Пул ресурсов: агенты выбирают из множества доступных ресурсов, перераспределяя выбор в случае необходимости.

- Масштабируемость: алгоритм функционирования агентов в центре не зависит от их количества.
- Измеряемый сервис: в текущей постановке задачи эта характеристика несущественна.

Задача

Разработать основанный на знаниях мультиагентный алгоритм, который гарантирует, что каждый агент, прибывший в центр ресурсов, рано или поздно получит какой-либо ресурс из своего списка желаний.

Назовём задачу и мультиагентный протокол её решения (алгоритм, представленный в следующем подразделе) *гонкой за ресурсы*.

Пример конфигурации задачи (т.е. мгновенный снимок в некоторый момент времени) приведён на рисунке 2. Поясним списки желаний агентов в этом примере:

- “книга на 2 часа < теннис на 1 час” в списке желаний первого агента означает, что он предпочитает арендовать книгу на 2 часа, чем играть в теннис 1 час.
- “книга на 1 час < футбол на 1 час *или* теннис на 1 час” в списке желаний второго агента означает, что он предпочитает (а) взять напрокат книгу на 1 час, чем взять футбольный мяч на 1 час, и (б) взять книгу на 1 час, чем взять теннисный набор на 1 час, а также (в) аренда футбольного мяча на 1 час и аренда теннисного набора на 1 час имеют одинаковый приоритет.

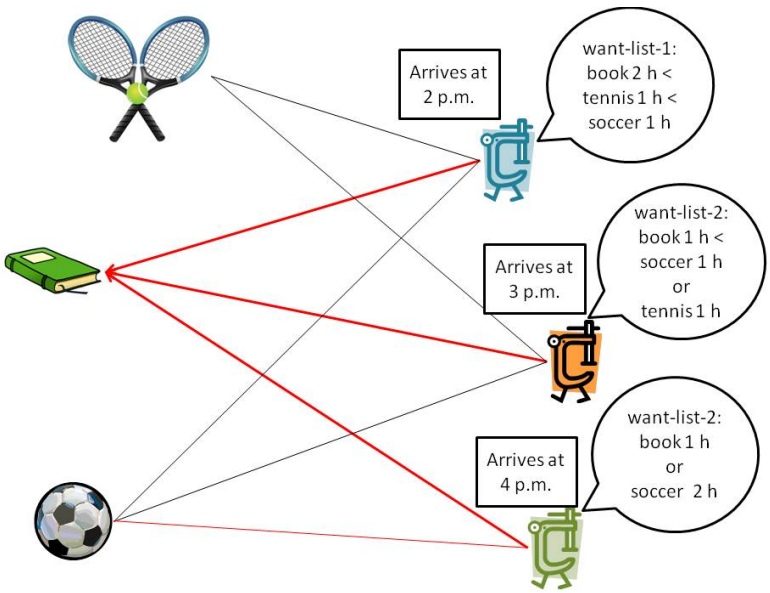


Fig. 2. Examples of agent requests

Рис. 2. Примеры запросов агентов

Нотация используемая в описании алгоритма

Алгоритм представляет собой пересмотренный мультиагентный алгоритм, основанный на знаниях из [10]. В обновлённом алгоритме мы различаем два случая: либо все агенты *мгновенные*, либо все агенты *реального времени*. Мгновенным агентам не нужно время для выполнения действий, в то время как каждое действие агентов реального времени выполняется с задержкой, и длительность этой задержки точно не известна. Мы обсудим разницу между мгновенными и агентами реального времени ниже.

Мы предполагаем, что каждый агент в любое время может запросить монитор о составе множества $POOL$ агентов, которые в настоящее время зарегистрированы. В каждый момент агент $i \in POOL$ устанавливает в качестве текущего намерения некоторый элемент ресурсов из своего списка же-

ланий $Desires_i \subseteq Resources$; в самом начале намерением является первый элемент этого списка. Конфликт (конкуренцию или гонку) между агентами, т.е. ситуацию, когда два агента выбирают один и тот же ресурс, можно выявить с помощью предиката $CONF : Agents \times Resources \times Agents \times Resources \longrightarrow Boolean$, так что $CONF(i, u_i, j, u_j) = true$ тогда и только тогда, когда $u_i = u_j$ ($i \neq j$, $i, j \in POOL$). Очевидно, что для всех агентов $i \neq j$, для всех элементов ресурсов u_i и u_j отношение конфликта симметрично: $CONF(i, u_i, j, u_j) \iff CONF(j, u_j, i, u_i)$. Мнения каждого агента представлены следующими приватными целочисленными счётчиками: счётчик $NumC$ для общего числа конфликтов используется как мгновенными агентами, так и агентами реального времени, а счётчик $CFree$ нужен только последним для подсчёта агентов без конфликтов.

Поясним подробнее разницу между мгновенными агентами и агентами реального времени: мгновенный агент захватывает свой ресурс сразу, как только $NumC = 0$, в то время как агент реального времени, которому требуется время, чтобы арендовать ресурс, хочет убедиться, что за это время никакой другой агент не планирует пользоваться этим же ресурсом. $NumC$ представляет собой верхнюю оценку агентом количества партнёров-агентов, с которыми у него может быть конфликт, и, соответственно, $CFree$ — это его текущий счётчик количества партнёров, которые в течение определённого времени подтверждали, что по их мнению, у них нет конфликтов ни с кем. Формально:

- Агент реального времени считает (имеет мнение), что он не конфликтует ни с одним другим агентом, если его счётчик $NumC = 0$,
- Агент реального времени считает (имеет мнение), что в системе нет конфликтов, как только $NumC = 0$ и $CFree = 2(n - 1)$, т.е. он дважды проверяет, что все остальные агенты полагают, что у них тоже нет конфликтов.

Если два агента конкурируют, то они решают конфликт с помощью функции $reSOL : Resources \times \mathbb{N} \times \mathbb{N} \times ResourceList \longrightarrow Resources \cup \{null\}$. Конфликтующие агенты вычисляют значение этой функции, используя данные о времени прибытия друг друга в центр ресурсов и приоритеты своих желаний: в конкуренции за ресурс выигрывает тот агент, который раньше прибыл в центр, и его намерение остаётся прежним, а проигравший агент выбирает в качестве намерения следующее по приоритету желание из своего списка. Отметим, что у проигравшего агента нет шансов когда-либо выиграть упущенный ресурс, т.к. если выигравший агент и уступит его впоследствии, то только тому, кто прибыл ещё раньше него. Поэтому проигранный ресурс удаляется из списка желаний проигравшего. В случае, если список желаний исчерпан, то значением функции $reSOL$ является $null$ и агент уходит из центра ресурсов. Формально, для всех агентов i и j ($i \neq j$), для времени прибытия t_i и t_j ($t_i \neq t_j$), для любых ресурсов u_i и u_j, и для списка желаний агента U_i:

- если $CONF(i, u_i, j, u_j)$, то
  - если $t_i < t_j$, то $reSOL(u_i, t_i, t_j, U_i) = u_i$;
  - если $t_i > t_j$, то $reSOL(u_i, t_i, t_j, U_i) = car(U_i)$;
- если $reSOL(u_i, t_i, t_j, U_i) = null$, то агент i выбывает из множества $POOL$.

Здесь и далее функция $car(List)$ возвращает первый элемент списка $List$ и удаляет его из этого списка.

В псевдокоде алгоритма мы используем следующие приватные переменные:

- `Me` — целочисленный идентификатор агента;
- `Desires` — целочисленный упорядоченный список желаний агента;
- `cur_des` — переменная для текущего списка желаний агента;
- `cur_un` и `par_un` — целочисленные переменные текущего намерения агента и намерения партнёра, с которым агент в настоящее время ведёт переговоры;
- `time` и `par_time` — целочисленное время регистрации агента и его партнёра в центре ресурсов;
- `par_bel` — булева переменная для мнения текущего партнёра о том, что у него нет конфликтов;

- contacts — переменная для (неупорядоченного) множества агентов, с которыми должны проводиться переговоры.

Описание алгоритма и псевдокод

Функционирование агента состоит из серии раундов связи. В каждом раунде агент должен пообщаться со всеми другими агентами. В каждом стандартном раунде, когда агент связывается с другим агентом, он проверяет, конфликтуют ли они за элемент ресурса. Всякий раз, когда агент не конфликтует с текущим партнёром, он уменьшает значение своей приватной переменной $NumC$, в противном случае он повторно инициализирует значение $NumC$ числом $(n-1)$, и кроме того, если агент проигрывает конкуренцию за ресурс, то его намерением становится очередное желание, а сам список его желаний убывает. Если же список желаний исчерпан, то агент завершает свою работу, покидая центр ресурсов и уменьшая при этом множество $POOL$.

Если стандартный раунд заканчивается с $NumC = 0$, т. е. агент полагает, что у него нет конфликтов, то он начинает раунды двойной проверки. В этих раундах он должен убедиться, что другие агенты тоже считают, что они бесконфликтны. Всякий раз при двойной проверке, когда партнёр уведомляет, что он полагает себя свободным от конфликтов, агент увеличивает значение своей приватной переменной $CFree$. Но если партнёр сообщает, что он не может считать, что у него нет конфликтов, то агент возвращается к стандартным раундам, повторно инициализируя переменную $NumC$ числом $(n-1)$ и переменную $CFree$ числом 0. Эти раунды называются двойной проверкой, потому что агент останавливается и берёт, наконец, в аренду последний выбранный ресурс, когда $CFree = 2(n-1)$, т.е. когда каждый из остальных агентов дважды подтвердит своё мнение о том, что у него нет конфликтов.

Мы приводим алгоритм для агентов реального времени. Алгоритм для мгновенных агентов получается игнорированием счётчика $CFree$.

```
const Me : integer in [1..n];
const time : integer;
const Desires : list of [1..m];
var NumC : integer in [1..(n-1)];
var CFree : integer in [1..(n-1)];
var contacts : set of [1..n];
var partner : integer in [1..n];
var cur_un, par_un : integer in [1..m];
var cur_des : list of [1..m];
var par_time : integer;
var par_bel : boolean;
begin
1: NumC := (n - 1); CFree := 0;
2: cur_des := Desires; cur_un := car(cur_des);
3: repeat
4:    if NumC > 0 then NumC := (n-1);
5:    contacts := POOL\{Me};
6:    repeat
7:      partner := any in the contacts ready to communicate;
8:      start communication with the partner:
9:      { send (<cur_un>; time; <(NC=0)?>) to partner
10:       receive (<par_un>; par_time; <par_bel>) from partner }
11:       if CONF(Me, cur_un, partner, par_un)
12:       then { cur_un := reSOL(cur_un, time, par_time, cur_des);
```

```
13:                if cur_un = null then goto leave;
14:                NumC := (n-1); CFree:= 0 }
15:       else if NumC > 0
16:            then { NumC := (NumC-1); CFree := 0}
17:            else if par_bel
18:                 then CF := CF+1
19:                 else { NumC := (n-1); CFree := 0 }
20:      close communication session with partner; }
21:      contacts := contacts\partner;
22:    until ( contacts != empty )
23: until ( NumC = 0 & CFree = 2(n-1) )
24: leave: POOL := POOL\{Me};
end.
```

Корректность и анализ решения

Утверждение 2. *Предположим, что в системе гарантирована справедливость коммуникаций между агентами и завершение работы. Тогда в момент завершения работы каждый зарегистрированный агент будет знать, что на ресурс, на который он претендует, не претендует никакой другой агент.*

Вполне упорядоченное множество — это частичный порядок $(D, \leq)$ без бесконечных строго убывающих последовательностей. Мы говорим, что распределённая система является вполне упорядоченной, если существует вполне упорядоченное множество $(D, \leq)$ и отображение F из конфигураций системы в D такое, что для всех агентов $i \neq j$, для всех элементов ресурса u_i и u_j, $CONF(i, u_i, j, u_i)$ означает, что выполнение $reSOL(u_i, t_i, t_j, U_i)$ уменьшает значение F.

Утверждение 3. *Предположим, что в системе гарантирована справедливость коммуникаций между агентами и система является вполне упорядоченной. Тогда система рано или поздно обязательно завершает свою работу.*

Наша мультиагентная система, реализующая вышеприведённый алгоритм, является вполне упорядоченной. В качестве функции F можно взять общее количество текущих желаний агентов. Действительно, в случае конкуренции за ресурс, проигравший агент уменьшает количество своих желаний вплоть до *null*.

Первая проблема с основанным на знаниях мультиагентным протоколом гонки за ресурсы — это отказоустойчивость: если какой-либо из агентов откажется от предписанного поведения, результат будет неверным. Вторая и основная проблема — это сложность этого алгоритма: наилучшая известная нам верхняя граница количества раундов равна $n!$ и в каждом раунде содержится $O(n^2)$ сеансов взаимодействия между агентами.

Заключение

Фактически, рассмотренные в настоящей статье мультиагентные алгоритмы, основанные на знаниях — это алгоритмы достижения своего рода консенсуса и, возможно, общего знания [7] при сохранении приватности: мнение (убеждение) каждого агента становится знанием (имеет доказательство) и каждый агент знает, что все другие агенты знают, что если какой-либо агент совершает действие, то этот совершающий действие агент имеет на это право. Для нас представляет интерес задача о достижимости не только консенсуса, но общего знания об этом консенсусе, но эта задача требует дополнительного исследования.

Надо сказать, что нам известно очень мало литературы по мультиагентным алгоритмам, основанных на знаниях. Фактически, мы можем указать только на фундаментальную монографию

[7]. Правда, есть довольно-таки обширная литература по различным вариантам динамической эпистемической логики для мультиагентных систем (например, [11, 12]), но эта литература посвящена классическим логическим вопросам — семантика и аксиоматизируемость, выразительная сила и разрешимость — но не вопросам конструирования мультиагентных алгоритмов, основанных на знаниях.

Из нашей статьи можно сделать вывод, что мультиагентные алгоритмы зачастую неэффективны и немасштабируемы: согласованность (достижение консенсуса, например), доступность (системы для новых агентов, желающих принять участие в ее работе) и высокая производительность (например, время, необходимое для достижения консенсуса) не достижимы одновременно. Например, в алгоритме гонки за ресурсы именно получение итоговых знаний с соблюдением приватности приводит к снижению производительности системы. Но мы хотели бы закончить статью на более оптимистической ноте: необходимы дополнительные исследования (включая моделирование) мультиагентного алгоритма гонки за ресурсы, и исследования достижения частичного консенсуса (с определённой вероятностью).

References

[1] M. Takada, *Distributed Systems: for Fun and Profit*. 2013. [Online]. Available: http://book.mixu.net/distsys/.

[2] A. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms*. Prentice-Hall, 2006.

[3] M. Wooldridge, *An Introduction to Multiagent Systems*. John Willey&Sons, 2002.

[4] C. Chappell, *Stanford Encyclopedia of Philosophy*. 2019, ch. Plato on Knowledge in the Theaetetus. [Online]. Available: http://plato.stanford.edu/entries/plato-theaetetus/.

[5] J. Ichikawa and M. Steup, *Stanford Encyclopedia of Philosophy*. 2017, ch. The Analysis of Knowledge. [Online]. Available: http://plato.stanford.edu/entries/knowledge-analysis/.

[6] P. Dütting and A. Geiger, *Algorithmic Mechanism Design*. Seminar Report, University of Karlsruhe, Fakultät für Informatik, 2007. [Online]. Available: https://webspace.science.uu.nl/~leeuw112/msagi/mech_design.pdf.

[7] R. Fagin, J. Halpern, Y. Moses, and M. Vardi, *Reasoning about Knowledge*. MIT Press, 1995.

[8] P. Mell and T. Grance, *The NIST Definition of Cloud Computing*. NIST Special Publication 800-145, 2011. [Online]. Available: http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf.

[9] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer, 2012.

[10] A. Satekbayeva and N. Shilov, “Some Results on Multiagent Algorithms in Social Computing/Software Context”, *Information*, vol. 17, no. 1, pp. 229–240, 2014.

[11] J. van Benthem, *Logical Dynamics of Information and Interaction*. Cambridge University Press, 2011.

[12] N. Alechina and B. Logan, “State of the Art in Logics for Verification of Resource-Bounded Multi-Agent Systems”, in *Fields of Logic and Computation III — Essays Dedicated to Yuri Gurevich on the Occasion of His 80th Birthday*, ser. LNCS, vol. 12180, Springer, 2020, pp. 9–29.

MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 27, NO. 4, 2020
journal homepage: www.mais-journal.ru

THEORY OF COMPUTING

InnoChain: a Distributed Ledger for Industry with Formal Verification on all Implementation Levels

V. A. Kukharenko[1], K. V. Ziborov[2], R. F. Sadykov[2,3], A. V. Naumchev[3], R. M. Rezin[3], L. A. Merkin-Janson[3]

DOI: 10.18255/1818-1015-2020-4-454-471

[1]Moscow Institute of Physics and Technology, 9 Institutskiy per., Dolgoprudny, Moscow Region 141701, Russia.
[2]Lomonosov Moscow State University, 1 Leninskie Gory, Moscow 119991, Russia.
[3]Innopolis University, 1 Universitetskaya, Innopolis 420500, Russia.

MSC2020: 93A30, 68Q60
Research article
Full text in Russian

Received November 18, 2020
After revision December 2, 2020
Accepted December 16, 2020

The extent of formal verification methods applied to industrial projects has always been limited. The proliferation of distributed ledger systems (DLS), also known as blockchain, is rapidly changing the situation. Since the main area of DLSs' application is the automation of financial transactions, the properties of predictability and reliability are critical for implementing such systems. The actual behavior of the DLS is determined by the chosen consensus protocol, which properties require strict specification and formal verification. Formal specification and verification of the consensus protocol is necessary but not sufficient. It is required to ensure that the software implementation of the DLS nodes complies with this protocol. The verified software implementation of the protocol must run on a fairly reliable operating system. The so-called "smart contracts", which are an important part of the applied implementations of specific business processes based on DLSs, must be verifiable as well. In this paper, we describe an ongoing industrial project that will result in a DLS verified at least at the four technological levels described above. We then share our experience with the formal specification and verification of HotStuff, a leader-based fault-tolerant protocol that ensures reaching distributed consensus in the presence of Byzantine processes.

Keywords: Byzantine fault tolerance; distributed consensus; blockchain; TLA+; verification; model checking

INFORMATION ABOUT THE AUTHORS

Vladimir Aleksandrovich Kukharenko correspondence author	orcid.org/0000-0001-7377-7548. E-mail: vladimir.a.kukharenko@gmail.com Bachelor student.
Kirill Viktorovich Ziborov correspondence author	orcid.org/0000-0002-5676-9105. E-mail: an49x00@gmail.com Bachelor student.
Rafael Faritovich Sadykov correspondence author	orcid.org/0000-0002-2792-2465. E-mail: sadykovrf@gmail.com Ph.D. student.
Alexandr Vladimirovich Naumchev	orcid.org/0000-0002-7683-0751. E-mail: a.naumchev@innopolis.ru Assistant professor, Ph.D.
Ruslan Maratovich Rezin	orcid.org/0000-0002-0604-2645. E-mail: r.rezin@innopolis.ru MSc.
Leonid Albertovich Merkin-Janson	orcid.org/0000-0002-8427-2200. E-mail: l.merkin@innopolis.ru Professor at Innopolis University, Doctor of Mathematics, Delft University of Technology, Dr.

Funding: Ministry of Digital Development, Communications and Mass Media of the Russian Federation and Russian Venture Company (Agreement No. 004/20 dd. 20.03.2020, IGK 0000000007119P190002).

For citation: V. A. Kukharenko, K. V. Ziborov, R. F. Sadykov, A. V. Naumchev, R. M. Rezin, and L. A. Merkin-Janson, "InnoChain: a Distributed Ledger for Industry with Formal Verification on all Implementation Levels", *Modeling and analysis of information systems*, vol. 27, no. 4, pp. 454-471, 2020.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 27, № 4, 2020
сайт журнала: www.mais-journal.ru

THEORY OF COMPUTING

InnoChain: распределенный реестр для индустриального применения с формальной верификацией на всех уровнях реализации

В. А. Кухаренко[1], К. В. Зиборов[2], Р. Ф. Садыков[2,3], А. В. Наумчев[3], Р. М. Резин[3], Л. А. Меркин[3]

DOI: 10.18255/1818-1015-2020-4-454-471

[1]Московский физико-технический институт, Институтский пер., д. 9, г. Долгопрудный, Московская обл., 141701 Россия.
[2]Московский государственный университет им. М.В. Ломоносова, Ленинские горы, д. 1, г. Москва, 119991 Россия.
[3]Университет Иннополис, Университетская, д. 1, г. Иннополис , 420500 Россия.

УДК 519.7
Научная статья
Полный текст на русском языке

Получена 18 ноября 2020 г.
После доработки 2 декабря 2020 г.
Принята к публикации 16 декабря 2020 г.

Степень применения методов формальной верификации в индустриальных проектах всегда была ограничена. Распространение систем распределенного реестра (СРР), известных также как блокчейн, быстро меняет ситуацию. Поскольку основной областью применения СРР является автоматизация финансовых транзакций, свойства предсказуемости и надежности являются критическими при реализации таких систем. Реальное поведение СРР определяется выбранным протоколом консенсуса, свойства которого нуждаются в строгой спецификации и формальной верификации. Формальная спецификация и верификация протокола консенсуса необходима, но недостаточна. Необходимо удостовериться, что программная реализация узлов СРР соответствует данному протоколу. Верифицированная программная реализация протокола должна запускаться на достаточно надежной операционной системе. Так называемые “умные контракт”, которые являются важной частью прикладных реализаций конкретных бизнес-процессов на основе СРР, также должны быть верифицируемы.В данной работе мы описываем реализующийся в настоящее время индустриальный проект, результатом которого станет СРР, верифицированная по меньшей мере на четырех описанных выше технологических уровнях. Мы также описываем наш опыт формальной спецификации и верификации протокола HotStuff – отказоустойчивого протокола для гарантированного достижения консенсуса в присутствии византийских процессов и лидера.

Ключевые слова: устойчивость к византийским условиям; распределенный консенсус; блокчейн; TLA+; верификация; проверка моделей

ИНФОРМАЦИЯ ОБ АВТОРАХ

Владимир Александрович Кухаренко автор для корреспонденции	orcid.org/0000-0001-7377-7548. E-mail: vladimir.a.kukharenko@gmail.com студент бакалавриата.
Кирилл Викторович Зиборов автор для корреспонденции	orcid.org/0000-0002-5676-9105. E-mail: an49x00@gmail.com студент бакалавриата.
Рафаэль Фаритович Садыков автор для корреспонденции	orcid.org/0000-0002-2792-2465. E-mail: sadykovrf@gmail.com аспирант.
Александр Владимирович Наумчев	orcid.org/0000-0002-7683-0751. E-mail: a.naumchev@innopolis.ru доцент, кандидат наук.
Руслан Маратович Резин	orcid.org/0000-0002-0604-2645. E-mail: r.rezin@innopolis.ru магистр.
Леонид Альбертович Меркин	orcid.org/0000-0002-8427-2200. E-mail: l.merkin@innopolis.ru профессор.

Финансирование: Министерство цифрового развития, связи и массовых коммуникаций РФ и АО "Российская венчурная компания"(договор №004/20 от 20.03.2020, ИГК 0000000007119P190002).

Для цитирования: V. A. Kukharenko, K. V. Ziborov, R. F. Sadykov, A. V. Naumchev, R. M. Rezin, and L. A. Merkin-Janson, “InnoChain: a Distributed Ledger for Industry with Formal Verification on all Implementation Levels”, *Modeling and analysis of information systems*, vol. 27, no. 4, pp. 454-471, 2020.

Введение

Целью данной работы является публикация опыта применения формальной спецификации и верификации в условиях реализации крупного индустриального проекта (далее – *Проекта*). Наблюдения, приведенные в статье, могут помочь другим исследователям в области формальных методов в выборе правильного подхода к спецификации и верификации разрабатываемых систем. Мы начинаем изложение результатов с сравнительного анализа протоколов консенсуса для систем распределенного реестра (СРР) с целью выбора такого из них, который удовлетворял бы требованиям задач, которые стоят перед нами в рамках Проекта. Основной задачей в данном случае является задача формализации и автоматизации процесса оказания услуг по заправке воздушных судов (ВС) с целью контроля своевременной оплаты и минимизации рисков, связанных с отказом одной из сторон от исполнения своих обязательств.

В обобщенном виде задачу заправки ВС можно рассматривать как регуляцию отношений между двумя видами участников: потребители и поставщики услуг. Также, без ограничения общности, можно утверждать, что каждый участник отношений управляет узлом СРР и участвует в изменениях состояния СРР. Под состоянием СРР можно понимать некоторый набор переменных, характеризующий процессы реального мира. Например, это может быть объем средств (баланс) участников или этап, на котором находится заправка некоторого ВС. Состояние СРР хранится каждым узлом и изменяется специальными командами – *транзакциями* – которые были инициированы узлами сети. Состояние СРР не может изменяться произвольным образом, что контролируется протоколом консенсуса. Не нарушая общности, можно сказать, что *протокол консенсуса* – это набор правил обмена сообщениями специального вида между узлами СРР с целью согласования списка транзакций для исполнения и перехода в новое состояние. Тем не менее, описание СРР зависит от специфики управляемого процесса. Чтобы не приходилось разрабатывать отдельную систему под каждую задачу, вводится понятие *смарт-контракта*. Под *смарт-контрактом* понимается некоторая программа, написанная на специальном языке программирования, и описывающая, каким образом изменяется состояние отдельно взятого процесса реального мира. Код смарт-контракта также хранится в качестве состояния СРР. Например, целесообразно иметь разные смарт-контракты для управления услугами по заправке ВС и хранению топлива. Таким образом, смарт-контракты позволяют упростить разработку высоконадежных систем для потенциально не связанных между собой задач реального мира, используя, тем не менее, общую инфраструктуру сети и общую реализацию базовых протоколов взаимодействия между узлами сети. Исходя из этой модели, отдельные транзакции содержат вызовы определенных функций смарт-контрактов для перехода в новое состояние СРР из текущего. Практически во всех существующих системах СРР код смарт-контракта не может быть изменен после его развертывания. Это гарантирует каждому участнику, что ни одна из сторон не может отказаться от обязательств, даже если является разработчиком смарт-контракта.

Важно отметить, что каждый участник отношений действует, в первую очередь, исходя из максимизации собственной выгоды – настолько, насколько это позволяет протокол консенсуса. Класс протоколов консенсуса, которые управляют отношениями подобного рода, принято называть устойчивым к «византийским» условиям (Byzantine Fault Tolerant, BFT). Другим классом протоколов, которые используются на практике, являются протоколы, устойчивые к выходу из строя определенного набора узлов (Crash Fault Tolerant, CFT). В качестве примера CFT протокола можно привести Raft [1]. CFT протоколы являются более производительными по сравнению с BFT протоколами, но не позволяют справиться со злонамеренным поведением узлов, и поэтому неприменимы в рамках рассматриваемой задачи.

Современные СРР также подразделяются на открытые и закрытые. Открытые СРР не накладывают ограничения на возможность подключения произвольного узла к сети, в то время как в закрытых СРР такие ограничения присутствуют. Как правило, в закрытых СРР узлы либо фиксируются при

развертывании сети, либо поддерживаются специальные транзакции, позволяющие существующим узлам добавлять новые. Как можно догадаться, открытые СРР значительно сложнее закрытых из-за более общей задачи, которая ими решается. Например, в открытых СРР возникает проблема, связанная с неконтролируемым порождением «бесполезных» транзакций узлами, которые сложно идентифицировать в реальном мире и невозможно отключить от СРР. Данная атака может в конечном счете привести к отказу в обслуживании для правильных узлов. Напротив, в закрытых СРР правильные узлы могут исключить узел из сети в случае вскрытия факта злонамеренного поведения. Более того, на участника закрытых СРР могут накладываться контрактные обязательства в реальном мире, прежде чем другой участник инициирует транзакцию по добавлению нового узла в сеть. Таким образом, основным требованием к протоколу консенсуса в закрытых СРР является обнаружение злонамеренного поведения узла с сохранением корректного состояния системы. Стоит заметить, однако, что скорость работы в закрытых СРР сильно зависит от числа узлов (в лучшем случае зависимость линейная), но гарантируется финальность принятого состояния – принятые и исполненные в рамках протокола консенсуса транзакции, которые привели систему в данное состояние, не могут быть отменены ни при каких обстоятельствах. В открытых СРР нет подобной зависимости от числа узлов, но в каждый момент времени финальность состояния можно вычислить лишь с некоторой вероятностью. Другими словами, принятая и исполненная в рамках протокола консенсуса транзакция может быть отменена с некоторой вероятностью, которая уменьшается с ростом количества транзакций принятых после нее. Примерами открытых СРР являются Bitcoin [2], Ethereum [3] и EOS [4], закрытых – Hyperledger Iroha [5] и Fabric [6]. Важно отметить, что современные протоколы консенсуса для открытых СРР содержат часть, реализующую протокол консенсуса для фиксированного количества участников, который также применим в закрытых СРР. Таким образом, на базе закрытой СРР возможна разработка открытой СРР.

Возвращаясь к задаче заправки ВС, кратко сформулированной в начале раздела, стоит заметить, что участники заправочной сети – это компании, сертифицированные на оказание услуг, связанных с топливом, а также владельцы ВС. Между ними можно распределить некоторое ограниченное множество узлов для доступа к заправочной сети, поскольку такие компании могут быть не заинтересованы в издержках на содержание выделенного узла. С другой стороны, количество узлов важно с точки зрения безопасности сети. Например, часть протокола DPOS (Delegated Proof of Stake) консенсуса СРР EOS [4], реализующая протокол BFT-консенсуса для фиксированного количества участников, требует порядка 21 узла, допуская, таким образом, что не более 7 из них могут управляться злоумышленниками. Также важно заметить, что возникает необходимость выделения некоторого привилегированного множество узлов, которые будут управлять процессом добавления или удаления новых узлов, что автоматически делает СРР закрытой. Можно заключить, таким образом, что достаточно выбрать подходящий протокол консенсуса для закрытой СРР. Это не означает, однако, что закрытая СРР не может эволюционировать в открытую: как было замечено ранее, новое состояние открытой СРР может определяться фиксированным числом участников, но этот набор должен периодически переизбираться из общего числа участников некоторой дополнительной логикой на основе прошлых состояний СРР.

В данной статье мы: рассматриваем наиболее популярные протоколы BFT-консенсуса и взаимосвязи между ними (раздел 1); приводим обоснование выбора протокола консенсуса HotStuff, а также технологий TLA+/TLC для его верификации (раздел 2); анализируем две независимые попытки спецификации и верификации протокола, одна из которых завершилась успешно, а другая натолкнулась на некоторые проблемы, причины которых мы разбираем (раздел 3). Анализ обеих попыток позволит будущим исследователям избежать аналогичных проблем, а также повторно использовать практики, которые привели к положительным результатам.

Итоговая TLA+ спецификация протокола HotStuff, формально верифицированная с помощью инструмента TLC на предмет соответствия типовым требованиям безопасности (safety), будет служить формальным техническим заданием для программной реализации узла распределенного реестра InnoChain. Для формальной верификации самой программной реализации мы планируем использовать инструментарий HOL4 [7]. Для этой цели TLA+ спецификация протокола HotStuff, вместе с программной реализацией узла, будет транслирована в набор теорем HOL4. Процесс верификации будет состоять в доказательстве полученных теорем с использованием операционной семантики целевого языка программирования, выраженной в виде аксиом HOL4. Программную реализацию узла планируется собирать с помощью формально верифицированного компилятора с последующим запуском на операционной системе, управляемой формально верифицированным микроядром seL4 [8].

1. Устойчивость к византийским условиям

Под *протоколом консенсуса* будем понимать распределенный алгоритм синхронизации состояния СРР между узлами. В данной работе будем рассматривать протоколы BFT-консенсуса, которые предполагают наличие не более чем f узлов (из $n \geq 3f + 1$ [9]) с произвольным злонамеренным поведением. Узлы, которые следуют протоколу консенсуса, будем называть *правильными*. Каждую попытку перехода из одного состояния СРР в другое будем называть *раундом*. Как уже было сказано в предыдущем разделе, состояние СРР изменяется в результате исполнения транзакций, отправленных пользователями. Таким образом, в рамках раунда протокола консенсуса узлам необходимо договориться о едином списке транзакций для выполнения.

Раунды протокола консенсуса для удобства описания подразделяют на фазы, в каждой из которых узлы обмениваются сообщениями с некоторой целью. Далее скажем, что протокол консенсуса является *частично синхронным*, если действует предположение, что время доставки сообщения не превышает заранее установленного значения Δ по истечении некоторого времени стабилизации сети (Global Stabilization Time, GST). В противном случае, будем называть протокол консенсуса *асинхронным*. В данной работе все рассматриваемые протоколы, за исключением HoneyBadgerBFT, являются частично синхронными.

Сложность протоколов консенсуса привязана к количеству операций создания или проверки электронно-цифровой подписи, используемой при обмене сообщениями, так как данная операция зачастую занимает даже больше времени, чем передача данных по сети.

Наконец, корректность протокола консенсуса определяется двумя классами свойств: *безопасности* (*safety*) и *живости* (*liveness*). Свойства безопасности гарантируют, что для любого момента времени t найдется такой момент времени $t' \geq t$, что все правильные узлы будут иметь одно и тоже состояние, которое будет являться результатом исполнения списка транзакций, согласованных в рамках протокола консенсуса. Свойства живости гарантируют, что для любого момента времени t найдется такой момент времени $t' \geq t$, что состояние сети изменится при наличии корректных транзакций от пользователя.

Дальнейший разбор наиболее популярных протоколов BFT-консенсуса строится на основе эволюции одного из первых реализованных на практике протоколов – PBFT (Practical Byzantine Fault Tolerant). Таким образом, демонстрируются решения задач, которые вставали перед научным сообществом после применения очередных улучшений в рамках актуального на тот момент протокола консенсуса. Также мы рассматриваем наиболее актуальный асинхронный протокол консенсуса – HoneyBadgerBFT – и проводим его сравнение с частично синхронными протоколами.

1.1. PBFT

Practical Byzantine Fault Tolerance (PBFT) [10] относится к классу частично синхронных BFT протоколов консенсуса. Данный протокол определяет поведение трех сущностей: пользователь, узел и лидер. Допустим, имеется перенумерованное множество узлов $N = \{1, ..., n\}$, где n ($n \geq 3f + 1$) – количество узлов сети, а f – максимальное количество злонамеренных узлов. Тогда лидер для раунда r определяется по правилу: $l = r \pmod n$. Под пользователем понимается произвольное устройство, которое может инициировать транзакции в СРР, используя интерфейс узла, но при этом может и не участвовать в протоколе консенсуса.

Предлагается следующий алгоритм действия пользователя:

- Формируется подписанная транзакция $\langle REQUEST, o, t, c \rangle_{\sigma_c}$, где o – данные операции, t – метка времени, c – идентификатор пользователя, σ_c – подпись пользователя.
- На основе отслеживаемого номера раунда вычисляется узел, который в данный момент является лидером. Далее, именно этому узлу направляется транзакция.
- Ожидается $f+1$ сообщений от узлов вида $\langle REPLY, r, t, c, i, rs \rangle_{\sigma_i}$, где r – номер текущего раунда, rs – результат исполнения транзакции.
- Если не удалось получить $f + 1$ результатов за разумное время, то рассылаем запрос первого шага всем узлам.

Раунд протокола консенсуса состоит из следующих фаз: *предподготовка* (*pre-prepare*), *подготовка* (*prepare*), *фиксация* (*commit*).

Без ограничения общности считаем, что в рамках раунда узлы формируют одну транзакцию пользователя для исполнения. Успешный раунд протокола консенсуса выглядит следующим образом:

- Лидер формирует и отправляет остальным узлам сообщение вида: $\langle \langle PRE\_PREPARE, r, s, d \rangle_{\sigma_l}, m \rangle$, где r – номер текущего раунда, s – порядковый номер, присвоенный транзакции пользователя, m – транзакция пользователя, d – хэш-значение транзакции.
- Остальные узлы, получив сообщение *PRE_PREPARE*, проверяют его корректность. В случае, если узел не принимал для текущих значений r и n сообщение с другим d, то он переходит в фазу *prepare* и рассылает всем узлам сообщение: $\langle PREPARE, r, s, d, i \rangle_{\sigma_i}$
- Получив сообщение *PREPARE*, узел проверяет, соответствуют ли значения r, s и d сообщению *PRE_PREPARE*. После получения $2f + 1$ корректных *PREPARE* сообщений от разных узлов, лидер отправляет сообщение $\langle COMMIT, r, s, d, i \rangle_{\sigma_i}$
- После получения $2f + 1$ корректных $COMMIT$ сообщений от разных узлов – при условии исполнения транзакций с меньшим порядковым номером – команда, заключенная в транзакции m, исполняется, и состояние СРР на узле i изменяется.
- После исполнения транзакции узел i отправляет пользователю результат в *REPLY* сообщении, как было отмечено ранее.

Наряду со сменой лидера, которая будет описана ниже, протокол гарантирует следующее важное свойство: транзакция m с порядковым номером s, исполненная одним узлом, обязательно будет исполнена с тем же порядковым номером $f + 1$ корректными узлами – возможно, после нескольких смен лидера.

В каждой фазе узел запускает таймер с интервалом Δ, отмеченном в определении частичной синхронности. Данный интервал линейно увеличивается в случае срабатывания таймера и уменьшается, если фаза заканчивается раньше. В случае срабатывания таймера, узлом инициируется процесс смены лидера.

Также, в качестве оптимизации, каждые k раундов каждым узлом запускается процесс сохранения стабильного состояния, которое используется для снижения объема хранимой информации и для синхронизации «отставших» узлов. Порядковый номер транзакции, соответствующий послед-

нему стабильному состоянию, рассылается остальным узлам в сообщении $\langle CHECKPOINT, s, d, i \rangle_{\sigma_i}$ и запоминается локально после получения $2f + 1$ корректного сообщения.

Процесс смены лидера инициируется узлом путем рассылки сообщения вида $\langle VIEW\_CHANGE, r + 1, s, C, P, i \rangle_{\sigma_i}$, где s – порядковый номер последнего стабильного состояния (совпадает с порядковым номером соответствующей транзакции), C – множество из $2f + 1$ подписей, доказывающих корректность соответствующего s состояния, P – множество, состоящее из множеств P_m, соответствующих транзакции m, для которой узлом получено $2f + 1$ *PREPARED* сообщение, и имеющей превышающий s порядковый номер.

После получения лидером $2f + 1$ корректных сообщений $VIEW\_CHANGE$, локальное стабильное состояние обновляется на новое с максимальным порядковым номером. Далее, лидер формирует и рассылает сообщение $\langle NEW\_VIEW, r + 1, V, O \rangle_{\sigma_l}$, где: V – множество, содержащее $2f + 1$ полученных лидером сообщений $VIEW\_CHANGE$; O – множество, содержащее *PRE_PREPARE* сообщения для каждой транзакции из P с порядковым номером, превышающим соответствующий текущему стабильному состоянию номер. Если множество транзакций, удовлетворяющих этому условию, пусто, то в качестве хэш-значения используется специальное значение d_{null}, сообщающее, что лидер сменился, но нет команд для исполнения. Каждый узел, в свою очередь, проверяет корректность построения множества O и других данных при получении NEW_VIEW и запускает описанный раннее процесс из двух фаз для каждого *PRE_PREPARE* сообщения.

Нетрудно видеть, что при благоприятном исходе раунда алгоритм достижения консенсуса имеет сложность $O(n^2)$ в силу того, что на каждом из n узлов каждая фаза требует проверки $O(n)$ подписей. В случае смены лидера, необходимо повторить фазы для порядка n транзакций (так как число f злонамеренных узлов линейно зависит от общего числа узлов), что приводит к сложности порядка $O(n^3)$.

1.2. HoneyBadgerBFT

BFT-протокол консенсуса HoneyBadgerBFT [11] относится к классу *асинхронных*, что означает отсутствие каких-либо предположений о задержках СРР по сравнению с протоколом, рассмотренном в предыдущем разделе. Главная идея протокола заключается в использовании распределенного алгоритма надежной доставки. Таким образом, узлам не нужно использовать таймер для смены состояния по истечении некоторого промежутка времени, так как сообщение, отправленное в сеть, рано или поздно будет доставлено.

Допустим, имеется пронумерованное множество узлов $N = \{1, ..., n\}$, где n ($n \geq 3f + 1$) – количество узлов сети, f – максимальное количество злонамеренных узлов. Перед началом работы сети генерируются необходимые для работы порогового шифрования [12] ключи: открытый ключ PK и n секретных SK_i ключей для каждого узла. Пороговое шифрование, как будет показано далее, используется для предотвращения цензуры транзакций. Предполагается, что на каждом узле имеется неограниченный буфер, который накапливает транзакции пользователей.

Абстрактное описание протокола консенсуса выглядит следующим образом:

- При наступлении раунда r, узел i случайным образом выбирает $\lfloor b/n \rfloor$ транзакций из первых b транзакций буфера. Обозначим эти транзакции *proposed*.
- Выбранный набор транзакций шифруется публичным ключом пороговой подписи: v_i = TPKE.Enc(PK, proposed).
- v_i передается на вход протоколу консенсуса для раунда r ($ACS[r]$), задача которого заключается в согласовании общего подмножества из предложенных узлами шифрованных наборов транзакций – $\{v_j\}_{j \in S}$, $S \subset N$.
- После получения $\{v_j\}$, инициируется процесс расшифровки $ACS[r]$ для раунда r.
- Для каждого $j \in S$ вычисляется e_j = TPKE.DecShare(SK_i, v_j).

- Рассылается сообщение вида $\texttt{DEC}(r, j, i, e_j)$.
- При получении $f + 1$ сообщений $\texttt{DEC}(r, j, k, e_{k,j})$, расшифровывается набор транзакций $y_j = \texttt{TPKE.Dec}(\texttt{PK}, (k, e_{k,j}))$.
- Общий набор транзакций для раунда r вычисляется как: $\texttt{block}_r = \texttt{sorted}(\bigcup_{j \in S} y_j)$.
- Обработанные транзакции удаляются из буфера.

Протокол консенсуса для получения общего подмножества транзакций для узла i использует протокол надежной передачи данных (RBC) [13] двоичного консенсуса (BA) [14] и имеет следующий вид:

- При получении узлом значения v_i, оно рассылается с помощью RBC.
- При получении значения v_j, оно передается протоколу бинарного согласования BA для определения включения данного набора транзакций в финальное множество.
- Как только $N - f$ результатов согласования получено для разных v_j, остальные результаты получают 0 в качестве значения.
- Завершается доставка всех v_j от RBC, и в качестве результата возвращается множество $\{v_j\}$ с результатом BA, равным 1.

Автор оригинальной работы установил, что при $b = \Omega(\lambda n^2 log(n))$ (λ is a security paramete) протокол консенсуса в худшем случае имеет сложность $O(b)$. Также можно заметить, что протокол наиболее эффективен в условиях избытка транзакций, так как в этом случае высока вероятность выбора каждым узлом разных наборов транзакций, что увеличивает пропускную способность сети.

1.3. SBFT

Протокол консенсуса Scalable Byzantine Fault Tolerance (SBFT) [15] разрабатывался с целью оптимизации PBFT. По этой причине в данном разделе мы не останавливаемся детально на алгоритме, а рассмотрим лишь предложенные авторами протокола изменения относительно PBFT.

Во-первых, стоит заметить, что в каждом раунде протокола PBFT каждый узел проверяет подпись сообщений от других узлов, что имеет сложность $O(n^2)$, где n – число узлов в сети. По этой причине авторы SBFT предложили ввести дополнительные 2 роли узлов в каждой фазе помимо лидера: C-коллектор и E-коллектор. Задачей C-коллектора является накопление сообщений определенного назначения от других узлов, пока их количество не достигнет необходимого порога, а также рассылка одного результирующего сообщения узлам, которое содержит единственную подпись, согласованную с подписями из полученных сообщений. Таким образом, C-коллектор и остальные узлы в совокупности проверяют только $O(n)$ сообщений в случае успешности раунда. Объединение нескольких подписей в одну достигается за счет использования так называемых пороговых подписей [16]. Главным недостатком данного подхода является то, что перед началом работы сети узлам необходимо совместно сгенерировать открытый ключ, который зависит от совокупности секретных ключей каждого узла. По аналогии с C-коллектором, задачей E-коллектора является сбор сообщений от узлов с результатом исполнения транзакции, объединение подписей и отправка одного сообщения пользователю. Таким образом, пользователю достаточно получить единственное корректное сообщения от E-коллектора для того, чтобы удостовериться, что транзакция успешно исполнена в CPP, вместо ожидания $f + 1$ сообщения от узлов, как в PBFT.

Второй важной оптимизацией является добавление «оптимистичного пути» раунда. Состоит она в том, что если C-коллектор до срабатывания таймера получает *PREPARED* сообщения от всех n узлов, то он сразу формирует $FULL\_COMMIT\_PROOF$ сообщение и рассылает его узлам. В случае корректности сообщения, узел сразу добавляет транзакцию в очередь на исполнение, минуя дополнительную фазу *precommit* протокола PBFT. В случае срабатывания таймера быстрого пути, протокол переходит в C двумя фазами (как PBFT, но с учетом оптимизации подписей), при этом C-коллектор становится лидером.

Далее, авторы SBFT предлагают использовать не один коллектор, а c C-коллекторов, где c намного меньше n. Это увеличивает вероятность прохождения раунда по быстрому пути даже в случае недоступности некоторых коллекторов. В результате получается, что число узлов в сети $n = 3f + c + 1$, где f – число злонамеренных узлов, а c – число узлов, которые могут быть недоступны, но не могут совершать других зловредных действий, противоречащих протоколу консенсуса.

С учетом оптимизации подписей можно заключить, что SBFT имеет сложность $O(n)$ в случае позитивного исхода раунда и $O(n^2)$ – в случае смены лидера.

1.4. Tendermint

Протокол консенсуса Tendermint [17] строится на тех же фазах, что и PBFT, но при этом в каждом раунде протокола консенсуса узлы договариваются не об одной транзакции, а о блоке, состоящем из набора транзакций. Авторы Tendermint предлагают оптимизацию, которая помогает избежать сложностей с повторением фаз протокола консенсуса для каждого неуспешного блока после смены лидера. Это достигается за счет специально разработанного протокола рассылки сообщений (Tendermint Gossip protocol) и машины состояний, которая хранит только последний блок для повторения протокола консенсуса в новом раунде в случае смены лидера. В отличие от SBFT, Tendermint позволяет оптимизировать второе из двух узких мест протокола PBFT: отправка сообщения от каждого узла всем остальным и повторение фазы протокола консенсуса для каждой неуспешной транзакции при получении необходимого количества *PREPARE* сообщений от узлов CPP.

Протокол рассылки сообщений Tendermint использует предположение о частичной синхронности сети и гарантирует, что если правильный узел отправил сообщение в момент времени t, то остальные правильные узлы получат его не позднее, чем через $max\{t, \mathtt{GST}\} + \Delta$, где GST – некоторый момент времени, когда сеть стабилизируется, а Δ – гарантированный интервал доставки сообщения в стабильной сети.

Авторы Tendermint предлагают вместо названий фаз PBFT *pre-prepare*, *prepare*, *commit* названия *proposal*, *prevote*, *precommit* для лучшего понимания задач каждой фазы; между фазами, тем не менее, можно провести взаимно однозначное соответствие. Также машина состояний Tendermint подразумевает хранение следующих значений для оптимизации процесса смены лидера: *validValue*, *validRound*, *lockedValue*, *lockedRound*. Значения *validValue* и *validRound* хранят значение последнего блока транзакций, для которого было получено $2f + 1$ сообщений *prevote*, а также номер соответствующего раунда. Эти значения сбрасываются после получения $2f + 1$ PRE_COMMIT сообщений, то есть после согласования блока в рамках протокола консенсуса, но до этого момента правильный лидер рассылает эти значения в своем $proposal$ сообщении. Значения *lockedValue* и *lockedRound* обновляются исключительно в фазе *precommit*, перед отправкой сообщения PRE_COMMIT, и необходимы для отслеживания того факта, что не будет отправлено $PREVOTE$ сообщение на *PROPOSAL*, блок транзакций которого отличен от *lockedValue*.

Таким образом, протокол рассылки сообщений Tendermint, в совокупности с машиной состояний, гарантирует тот факт, что блок транзакций *lockedValue* рано или поздно будет принят сетью, так как после конечного количества раундов найдется правильный лидер, который разошлет *PROPOSAL* с *validValue* и *validRound*, согласованный с *lockedValue* и *lockedRound*. Данное сообщение получат остальные правильные узлы ввиду гарантий протокола передачи данных.

В результате, сложность протокола консенсуса Tendermint оценивается как $O(n^2)$ в общем случае, где $n \geq 3f+1$ – количество узлов сети, f – максимальное число злонамеренных узлов. Стоит заметить, однако, что эту сложность можно улучшить до $O(n)$, если использовать подход с пороговыми подписями SBFT.

1.5. HotStuff

У протокола консенсуса Tendermint отсутствует свойство «оптимистичного отклика» (Optimistic responsiveness) ввиду синхронной задержки, вызванной необходимостью гарантировать хоть какой-то отклик: после достижения GST, выбранному правильному лидеру достаточно получить $2f + 1$ сообщений о смене лидера, содержащих *validValue* для формирования *PROPOSAL* сообщения с блоком транзакций, который рано или поздно будет принят сетью. Следующий сценарий показывает отсутствие описанной проблемы при отсутствии синхронной задержки между раундами. Допустим, один из узлов получит $2f + 1$ *prevote* сообщений и обновит как *validValue*, так и *lockedValue*, и станет заблокированным, в то время как остальные узлы сменят лидера. Новый лидер не узнает актуального *validValue* и предложит другой блок транзакций, который отвергнет заблокированный узел. Более того, в этом новом раунде ситуация может повториться, и заблокируется еще один узел.

Описанная проблема возникает из-за того, что переменные *validValue* и *lockedValue* обновляются в одной фазе раунда. Авторы протокола HotStuff [18] предложили добавить еще одну фазу для решения этой проблемы и обновлять эти переменные в разных фазах. Таким образом, если узел блокируется (устанавливается переменная *lockedValue*), то это также означает, что не менее $2f + 1$ узлов установили переменную `validValue`, и корректный лидер включит эти транзакции в предложенные для нового раунда. Таким образом, в HotStuff узлы голосуют в фазах *prepare*, *pre-commit*, *commit* и применяют результат в фазе *decide*. Также вместо переменных *validValue* и *lockedValue* используются *prepareQC* и *lockedQC*. Помимо этого, HotStuff предполагает использование упомянутой не раз пороговой криптографии, чем и гарантирует линейную сложность работы. Сбором подписей и генерацией единой подписи занимается лидер раунда.

Успешный раунд протокола консенсуса HotStuff описывается следующим образом:

- Лидер формирует и отправляет остальным узлам сообщение вида $\langle PREPARE, curProposal, highQC \rangle_{\sigma_l}$, где $curProposal$ – блок транзакций, предлагаемый для согласования в текущем раунде, $highQC$ – доказательство того, что блок транзакций соответствует самому актуальному *prepareQC* в сети.
- Остальные узлы, получив сообщение *PREPARE*, проверяют его корректность. Если переменная узла *lockedValue* не установлена или соответствует *curProposal* и *highQC*, то узел отправляет сообщение $\langle VOTE\_PREPARE, curProposal \rangle_{\sigma_i}$.
- Получив $2f + 1$ сообщений $VOTE\_PREPARE$, лидер формирует пороговую подпись, обновляет *prepareQC* на основе *curProposal* и голосов узлов, отправляет сообщение $\langle PRE\_COMMIT, prepareQC \rangle_{\sigma_i}$.
- Остальные узлы, получив сообщение PRE_COMMIT, проверяют его корректность и подпись. В случае успеха, узел отправляет сообщение $\langle VOTE\_PRE\_COMMIT, prepareQC \rangle_{\sigma_i}$ и обновляет *prepareQC*.
- Получив $2f + 1$ сообщений $VOTE\_PRE\_COMMIT$, лидер формирует пороговую подпись, устанавливает *lockedQC* равной *prepareQC* и отправляет сообщение $\langle COMMIT, prepareQC \rangle_{\sigma_i}$.
- Остальные узлы, получив сообщение $COMMIT$, проверяют его корректность и подпись. В случае успеха, узел отправляет сообщение $\langle VOTE\_COMMIT, prepareQC \rangle_{\sigma_i}$ и обновляет *lockedQC*.
- Получив $2f + 1$ сообщений $COMMIT$, лидер формирует пороговую подпись и отправляет сообщение $\langle DECIDE, lockedQC \rangle_{\sigma_l}$. Кроме того, соответствующий *lockedQC* блок попадает в очередь на исполнение.
- Остальные узлы, получив сообщение *DECIDE*, проверяют его корректность и подпись. В случае успеха, узел добавляет соответствующий блок в очередь на исполнение.

Далее, авторы HotStuff заметили, что получившие *PREPARE* сообщения узлы могут оптимистично предположить, что предложенный блок будет успешно принят, и параллельно инициировать следующий раунд. Таким образом, блоки, ожидающие принятия, становятся зависимы друг от друга и могут быть представлены в виде списка. Каждый последующий блок содержит подписи узлов для предыдущего блока. Таким образом, если после данного блока в список добавились еще три блока, то можно считать этот блок принятым, так как это событие эквивалентно прохождению трех фаз протокола HotStuff. Данный алгоритм получил название Chained HotStuff.

Table 1. Comparison of partially synchronous BFT-consensus protocols

Таблица 1. Сравнение частично синхронных протоколов BFT-консенсуса

Протокол	Правильный лидер	Смена лидера	Оптимист. отклик
PBFT [10]	$O(n^2)$	$O(n^3)$	Есть
SBFT [15]	$O(n)$	$O(n^2)$	Есть
Tendermint [17]	$O(n^2)$	$O(n^2)$	Нет
HotStuff [18]	$O(n)$	$O(n)$	Есть

В таблице 1 приведено заключительное сравнение наиболее популярных частично синхронных протоколов BFT-консенсуса, используемых на практике. Алгоритмическая сложность измеряется в количестве проверок электронно-цифровой подписи за раунд в зависимости от числа узлов в сети $n \geq 3f + 1$, где f – максимально возможное количество злонамеренных узлов. Как было показано в предыдущих разделах, первопроходец PBFT имеет довольно высокую алгоритмическую сложность и сложную логику смены лидера. Авторы SBFT предложили оптимизацию с помощью пороговых подписей, а также оптимизацию *оптимистичным путем* для стабильных сетей, но при этом сложность логики смены лидера осталась той же. Далее, авторы Tendermint предложили существенное упрощение логики смены лидера, но при этом жертвуют свойством *оптимистичного отклика* и вносят зависимость от протокола передачи данных (Tendermint gossip) для корректной работы. Наконец, протокол HotStuff предлагает решение проблем протокола Tendermint введением дополнительной фазы, но с сохранением линейной сложности.

Отдельно стоит отметить, что асинхронные протоколы на данный момент имеют более высокую алгоритмическую сложность ($\Omega(\lambda n^2 log(n))$ - HoneyBadgerBFT) и сложнее в реализации.

2. Верификация Распределенного Консенсуса

Проведенное в разделе 1 сравнение протоколов консенсуса говорит о том, что HotStuff является лучшим выбором в качестве протокола консенсуса для реализации закрытой системы распределенного реестра с точки зрения сложности работы и наличия свойства оптимистичного отклика. Стоит заметить, однако, что ввиду необходимости использования криптографического стандарта ГОСТ 34.10 и отсутствия в нем решения для пороговых подписей, реализация HotStuff планируется без них. С одной стороны, это увеличит алгоритмическую сложность до $O(n^2)$, с другой – упростит формальную верификацию реализации протокола. Далее заметим, что все рассмотренные протоколы BFT-консенсуса с линейной сложностью предполагают использование пороговых подписей, но либо не имеют свойства оптимистичного отклика (как Tendermint), либо имеют более высокую сложность для фазы смены лидера (PBFT, SBFT) в сравнении с Hotstuff.

Помимо сложности работы протокола консенсуса и времени его реализации, необходимо оценить его формальную верифицируемость: насколько сложно доказать, что программный код алгоритма имеет свойства, требуемые его формальным описанием – спецификацией. В качестве примера можно рассмотреть не раз упомянутое свойство безопасности. Тогда одной из задач формальной верификации протокола консенсуса является математическое доказательство того, что распреде-

ленная система никогда не попадет в такую ситуацию, когда два разных правильных узла сети будут иметь противоречащие друг другу состояния (например, один узел будет сообщать, что заправка ВС закончилась успешно, а другой – что не успешно). На сегодняшний день существует два подхода к формальной верификации протоколов консенсуса: верификация методом проверки моделей (model checking [19] и верификация дедуктивными методами [7, 20, 21]).

Метод проверки моделей применяется не к исходному коду протокола, который используется для компиляции и исполнения, а к абстрактной модели протокола в виде структуры Крипке. Каждому состоянию автомата соответствует набор свойств протокола, которые верны в данном состоянии. Доказательство того, что построенная модель имеет заданное свойство, осуществляется путем представления этого свойства в виде формулы в одной из темпоральных логик – например, LTL (логика линейного времени) или CTL (логика ветвящегося времени). Для каждой из логик существует отдельный алгоритм, который проверяет свойство для заданной структуры Крипке. Результатом работы данного алгоритма является либо сообщение о том, что свойство выполняется, либо возвращение последовательности переходов (контрпример) в автомате, которая демонстрирует нарушение свойства. Преимуществом подхода проверки моделей является то, что структура Крипке представляет собой более формальную запись протокола консенсуса, нежели некоторый псевдокод, который используется в научных статьях и спецификациях для объяснения идеи работы протокола, что позволяет находить недостатки либо в момент формирования модели, либо при исследовании контрпримера после завершения процесса проверки модели. Одним из недостатков данного подхода является то, что зачастую модели протоколов консенсуса в первоначальной записи имеют огромное число состояний, что не позволяет завершить алгоритм верификации за приемлемое время, в результате чего возникает задача упрощения исходной модели. Другим недостатком метода является то, что проверке подвергается не исходный код протокола, а его абстрактная модель, в результате чего возникает задача доказательства соответствия проверенной модели ее программной реализации.

Напротив, дедуктивные методы верификации чаще всего применяются к программной реализации, но требуют разработки операционной семантики языка реализации в выбранной системе верификации. Операционная семантика необходима для получения информации о том, каким образом та или иная синтаксическая конструкция языка программирования изменяет состояние программы. Дедуктивный подход к верификации использует идею традиционного формирования доказательства математическими методами, но с возможностью автоматизации шаблонных техник – тактик (например, метод математической индукции), которые являются составной частью *доказателей* (proof assistant). Таким образом, формальная верификация дедуктивным методом подразумевает написание программы для протокола консенсуса на некотором языке программирования и перенос этого кода в доказатель, в котором уже описана операционная семантика для данного языка, с последующей попыткой автоматического доказательства с помощью встроенных в доказатель тактик. Главным недостатком данного подхода в сравнении с проверкой моделей является то, что процесс не подразумевает автоматического нахождения контрпримера – то есть либо удается доказать нужное свойство для программы, либо не получается; при этом неудачная попытка не означает, что такого доказательства не существует.

На момент написания данной работы нам удалось обнаружить результаты верификации протоколов Tendermint и PBFT. В работе по верификации Tendermint авторы используют метод проверки моделей для доказательства того, что построенная модель протокола в системе TLC имеет свойство безопасности. Напротив, для верификации PBFT применялся метод дедуктивной верификации в системе Coq. Более того, был разработан специальный фреймворк, Velisarius [22], который можно применять для верификации произвольного BFT протокола консенсуса. Velisarius представляет собой набор теорий для доказателя Coq [21], в которые встроены основные абстракции протоколов

консенсуса: например, теория *EventOrdering* для описания событий, происходящих на разных узлах сети, или теория *Process*, в которой определены машины состояний узлов. С помощью Velisarius было доказано свойство безопасности PBFT, и из полученных спецификаций сгенерирован исполняемый код протокола на языке OCaml.

Несмотря на имеющиеся результаты формальной верификации Tendermint и PBFT, мы приняли решение использовать Hotstuff в качестве основного протокола системы распределенного реестра, реализуемой в рамках Проекта, ввиду наличия свойства оптимистичного отклика и упрощенной реализации фазы смены лидера. Стоит заметить, что схожесть Tendermint и Hotstuff позволяет применить идеи, использованные при проверке модели протокола Tendermint, для верификации модели протокола Hotstuff. Помимо применения проверки моделей, необходимо также использовать дедуктивные методы для верификации исходного кода протокола – например, используя фреймворк Velisarius.

3. Спецификация и Верификация Протокола HotStuff с Помощью TLA+/TLC

Так как протоколы HotStuff и Tendermint очень похожи, то первая версия модели была реализована на основе работы Игоря Коннова [23]. Ввиду того, что она фокусировалась на свойстве безопасности, то с этого и началась работа. Все последующие рассуждения, как и проверка модели в целом, основываются на том, что мы доверяем представлению реальной модели поведения в TLA+. Мы детально описываем сделанные нами допущения и модификации, а также обосновываем наличие таких допущений. Мы не включаем код наших TLA+ спецификации в статью для экономии пространства. Обе спецификации находятся в открытом доступе на GitHub [24].

3.1. Попытка 1

В модели, описываемой в этом разделе [25], была предпринята попытка добавления древовидной структуры команд в явном виде. Согласно статье, описывающей протокол HotStuff [18], каждая команда является полем некоторой структуры данных – вершины. Вершина так же содержит множество хэш-значений всех предыдущих команд, таким образом определяя отношение частичного порядка на множестве вершин. Будем говорить, что вершина B *актуальнее* вершины A, если множество хэш-значений из A является подмножеством хэш-значений из B. При построении описываемой модели криптография HotStuff была опущена (в пороговых подписях опущено поле sig) для достижения разумного времени проверки модели. С той же целью хэш-значения предшествующих команд были заменены на множество самих команд.

При таком подходе к построению вершины она будет содержать множество всех команд, прошедших фазу *PREPARE*, что сильно скажется на числе состояний модели и, следовательно, увеличит время проверки модели протокола. Однако, согласно первоначальному замыслу, данный подход может упростить спецификацию свойств протокола при ещё одном допущении: команды добавляются в множество вместе с их порядковым номером. Нужный номер команде назначает лидер в фазе *PREPARE*, а влияние византийского лидера будет обнаружено корректными узлами с помощью предикатов *ExtendsFrom* и *SafeNode*. Таким образом, при выбранном подходе вершина состоит из родительской ссылки – множества пар команд и соответствующих им порядковых номеров – и выбранной в текущем раунде команды. Данная оптимизация заметно сокращает множество возможных значений вершины, а значит и множество возможных состояний модели при верификации. Также стоит заметить, что оптимизация не мешает перенести свойство частичного порядка определенное ранее: вершина B является *актуальнее* вершины A, если упорядоченная по номеру последовательность команд из A является префиксом упорядоченной по номеру последовательности команд из B.

3.1.1. Предикаты SafeNode и ExtendsFrom

Узел в фазе *PREPARE* ожидает сообщения типа *PREPARE* от лидера, а затем проверяет полученную вершину с помощью предикатов *ExtendsFrom* и *SafeNode*.

Двуместный предикат *ExtendsFrom* в описываемой модели выполняет проверку следующего свойства вершины.

- Новая вершина актуальнее вершины из предыдущего раунда;
- Если к множеству пар команд вершины из предыдущего раунда добавить ее выбранную команду с соответствующим номером, то полученное множество совпадет с множеством пар команд новой вершины;

В случае, если указанные свойства выполнены, *ExtendsFrom* обращается в *TRUE*.

Предикат *SafeNode* реализован аналогично псевдокоду HotSuff [18]: он получает на вход вершину, проверяет ее актуальность и соответствие раунда вершины текущему раунду узла.

3.1.2. Спецификация свойств

Для спецификации свойств протокола, по аналогии с Tendermint, была введена переменная *decision*. Для каждого процесса *decision* хранит множество пар команд, которые будут исполнены машиной состояний узла, и множество соответствующих им порядковых номеров. Изменение переменной *decision* происходит только в фазе *DECIDE* узла после получения от лидера сигнала на исполнение команд (сообщения типа *DECIDE* с *CommitQC*). Это изменение заключается в добавлении в *decision* пары, состоящей из принятой на исполнение команды и соответствующего ей номера.

Описанная модель позволяет легко специфицировать свойство корректности (agreement), заключающееся в том, что все корректно работающие узлы получают одинаковое значение после исполнения принятой команды. Другими словами у любых двух корректных узлов их множества *decision* пусты (состояние не изменилось), либо между ними можно построить биекцию.

3.1.3. Итоги

В результате запуска проверки описываемой модели в TLC была выявлена ошибка, связанная с неправильной реализацией отправки всевозможных сообщений с новой пороговой подписью византийскими узлами. Кроме того, стало очевидно, что такой подход повлечёт за собой генерацию большого количества состояний, а проверка будет занимать длительное время. По этой причине необходимо исследовать варианты оптимизации полученной модели HotStuff.

3.2. Попытка 2

Первая проблема, с которой пришлось столкнуться при разработке альтернативной спецификации [26] – использование пороговых подписей в HotStuff. Она привела к необходимости различать подписанные и неподписанные сообщения, а также наложила ряд ограничений на византийские узлы – пришлось сделать допущение о невозможности подделать поле с пороговой подписью. Это не позволяет добавлять все сообщения византийских узлов в начале работы алгоритма. Решением данной проблемы является возможность добавлять эти сообщения по мере продвижения алгоритма, то есть был добавлен переход в машине состояний, при котором, если это возможно, все византийские узлы отправляют всевозможные сообщения с новой пороговой подписью.

Вторая проблема – наличие древовидной структуры предложенных команд (используется в предикате *ExtendsFrom*), которая формируется при обработке новых команд лидером. Попытка добавления этой структуры в явном виде описана в предыдущем подразделе. Такой подход многократно увеличивал число состояний, а также усложнял описание работы византийских узлов. Поэтому эта проблема была решена позже, и временно проверялась лишь корректность команды.

3.2.1. Проблемы с оптимизацией

После решения первой проблемы, написанная спецификация почти полностью соответствовала модели поведения. Были добавлены служебные состояния и функции, но сути это не меняло. Появилась другая проблема – проверка этой модели в TLC с византийскими узлами занимала слишком много времени (даже при $N = 4$). Это был ожидаемый результат, так как по сравнению с Tendermint увеличилось число фаз, а также усложнилась подделка сообщений византийскими узлами. Все это создавало слишком много новых состояний.

В качестве решения предлагается упрощение модели и использование следующего допущения (на основании оригинального алгоритма [18]):

- Когда ожидаются $N - F$ голосов от узлов, нам важен лишь факт получения сообщения, а не личность отправителя – поэтому был введен специальный массив-счетчик (*msgsCount*). Кроме того, это сильно упрощает подделку пороговых подписей для византийских узлов, так как они формируются аналогично. Важно отметить, что мы не допускаем копий и везде учитываем лишь первую отправку сообщения.
- В некоторых фазах – $PRECOMMIT$, $COMMIT$ и $DECIDE$ – лидер не делает ничего полезного, кроме создания пороговых подписей для доказательства получения сообщений от узлов, поэтому в этих фазах было необходимо отказаться от позиции лидера. Теперь все узлы ожидают нужного количества сообщений от других узлов, а не пороговой подписи от лидера (и реализуется это через *msgsCount*).
- После предыдущих упрощений понятно, что поле фазы (в оригинальной работе $type$) в пороговой подписи становится не используемым и по этой причине можно от него избавиться.
- Также видно, что в фазе *PREPARE* лидером никак не используется поле *node*. Это поле также было ликвидировано.

Кроме того, был немного переработан способ хранения отправленных сообщений для фазы *PREPARE*. Получились массивы, которые в полях отправленных сообщений хранили множество прикрепленных пороговых подписей (функция: отправитель -> раунд -> вершина листа с командой -> множество полей $justify$).

В результате время работы TLC значительно сократилось, и стал виден прогресс исследования состояний модели. Хотя в определении *ExtendsFrom* все еще оставалась заглушка в виде проверки корректности отправленной команды, удалось запустить модель на различных значениях и найти ошибку в TLA+ спецификации. Также удалось проверить корректность на 4 узлах (за сутки работы на сервере, что является хорошим результатом для продолжительности верификации подобных моделей) и получить ошибки при различных N и большом количестве византийских узлов (больших F).

3.2.2. Добавление древовидной структуры команд

Для целей оптимизации количества состояний модели было использовано предположение, что ветвление больше двух делать нецелесообразно ввиду того, что если удается получить ошибку при большем ветвлении, то ошибка должна обнаружиться и без него. Также было использовано предположение что количество ветвлений должно быть конечным.

Для реализации этого замысла двоичное дерево задается на этапе создания модели. В результате можно не создавать новые команды динамически, что позволяет избежать новых состояний, а также упрощает написание предиката $ExtendsFrom$. Также была исключена проверка корректности предложенных команд, так как она слишком сильно зависит от конкретной реализации протокола.

Теперь все команды, предлагаемые для согласования в рамках протокола консенсуса, берутся из двоичного дерева. В такой ситуации, когда все команды корректные, ошибка может произойти лишь из-за принятия неверной ветви команд одной или несколькими вершинами. Из-за этого

изменились проверяемые инварианты. Свойство $Validity$ теперь говорит, что все принятые узлом вершины продолжают друг друга. Свойство “Agreement” теперь говорит, что все узлы приняли вершины из одной ветви. Теперь спецификация учитывает все необходимые нам детали протокола.

Древовидная структура добавила много новых состояний, из-за чего время работы (при $N = 4$, на сервере) увеличилось приблизительно до трех недель. Но это разумное время, которое позволяет проверить необходимые случаи при небольших значениях N.

3.2.3. Возможное развитие модели

Нами была рассмотрена возможность спецификации и верификации свойства живости. Для проверки этого свойства необходима модель с конечным числом состояний, что привело к двум вариантам:

1. Обобщить протокол, чтобы сократить роль раундов – пока не совсем ясно, как это делать; кроме того, это будет серьезным отступлением от оригинальной модели поведения.
2. Ограничить число раундов, как и в случае свойства безопасности. Это понизит качество проверяемого условия, но позволит проверить при конкретных значениях. Для этого варианта нужно не так много изменений в спецификации для свойства безопасности, что делает его предпочтительным.

Таким образом, для проверки свойства живости предлагается изменить инвариант в спецификации. Предполагается, что он должен гласить следующее: если все узлы оказались в одном раунде, лидер корректный, а таймер NEW_VIEW не успевает сработать (можем такое предполагать, так как считаем, что GST наступило, а также прошло необходимое время для увеличения таймера), то решение будет принято.

4. Заключение

В данной статье мы:

- Привели краткое описание индустриального проекта по построению распределенного реестра, обязательной частью которого является формальная верификация всех основных компонентов разрабатываемой системы.
- Проанализировали популярные протоколы распределенного консенсуса, применимые к нашей задаче.
- Провели обзор применяемых на практике методов формальной верификации таких протоколов.
- Объяснили выбор протокола HotStuff и метода проверки моделей для его верификации.
- Описали и проанализировали две независимые попытки спецификации HotStuff в TLA+ с последующей проверкой выполнения требуемых свойств в TLC (инструментарий TLA+/TLC является стандартом де-факто в проверке моделей распределенных протоколов).

Мы надеемся, что проведенный нами анализ сэкономит время и силы других исследователей, занимающихся реализацией похожих проектов.

References

[1] M. Fazlali, S. M. Eftekhar, M. M. Dehshibi, H. T. Malazi, and M. Nosrati, «Raft Consensus Algorithm: an Effective Substitute for Paxos in High Throughput P2P-based Systems», *CoRR*, vol. abs/1911.01231, 2019.

[2] S. Nakamoto, «Bitcoin: A peer-to-peer electronic cash system», Manubot, Tech. Rep., 2019.

[3] G. Wood *et al.*, «Ethereum: A secure decentralised generalised transaction ledger», *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[4] E. Elrom, «EOS.IO Wallets and Smart Contracts», in *The Blockchain Developer*, Springer, 2019, pp. 213–256.

[5] F. Muratov, A. Lebedev, N. Iushkevich, B. Nasrulin, and M. Takemiya, «YAC: BFT Consensus Algorithm for Blockchain», *CoRR*, vol. abs/1809.00554, 2018.

[6] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, *et al.*, «Hyperledger fabric: a distributed operating system for permissioned blockchains», in *EuroSys*, ACM, 2018, 30:1–30:15.

[7] T. Gauthier, C. Kaliszyk, and J. Urban, «TacticToe: Learning to Reason with HOL4 Tactics», in *LPAR*, ser. EPiC Series in Computing, vol. 46, EasyChair, 2017, pp. 125–143.

[8] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, «seL4: formal verification of an OS kernel», in *SOSP*, ACM, 2009, pp. 207–220.

[9] L. Lamport, R. Shostak, and M. Pease, «The Byzantine Generals Problem», *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, 1982.

[10] M. Castro and B. Liskov, «Practical Byzantine Fault Tolerance», in *OSDI*, USENIX Association, 1999, pp. 173–186.

[11] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, «The honey badger of BFT protocols», in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 31–42.

[12] J. Baek and Y. Zheng, «Simple and efficient threshold cryptosystem from the Gap Diffie-Hellman group», in *GLOBECOM*, IEEE, 2003, pp. 1491–1495.

[13] M. Ben-Or, B. Kelmer, and T. Rabin, «Asynchronous secure computations with optimal resilience», in *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, ACM, 1994, pp. 183–192.

[14] A. Mostefaoui, M. Hamouna, and M. Raynal, «Signature-Free Asynchronous Byzantine Consensus with $t < n/3$ and $O(n^2)$ Messages», in *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, ACM, 2014, pp. 2–9.

[15] G. Golan-Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. K. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu, «SBFT: a Scalable Decentralized Trust Infrastructure for Blockchains», *CoRR*, vol. abs/1804.01626, 2018.

[16] D. Boneh, B. Lynn, and H. Shacham, «Short signatures from the Weil pairing», *Journal of cryptology*, vol. 17, no. 4, pp. 297–319, 2004.

[17] E. Buchman, J. Kwon, and Z. Milosevic, «The latest gossip on BFT consensus», *CoRR*, vol. abs/1807.04938, 2018.

[18] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, «Hotstuff: Bft consensus with linearity and responsiveness», in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019, pp. 347–356.

[19] Y. G. Karpov, *MODEL CHECKING. Verification of parallel and distributed software systems*, 2010.

[20] L. C. Paulson, *Isabelle: A generic theorem prover*. Springer Science & Business Media, 1994, vol. 828.

[21] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, H. Herbelin, *et al.*, «The Coq proof assistant reference manual», *INRIA, version*, vol. 6, no. 11, 1999.

[22] V. Rahli, I. Vukotic, M. Völp, and P. Esteves-Verissimo, «Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq», in *ESOP*, ser. Lecture Notes in Computer Science, vol. 10801, Springer, 2018, pp. 619–650.

[23] I. Konnov, *Model Checking Tendermint*, 2020. [Online]. Available: https://github.com/informalsystems/verification/tree/igor/fork/spec/fork-cases.

[24] V. Kukharenko, K. Ziborov, and R. Rezin, *HotStuff TLA+ Specifications*. [Online]. Available: https://github.com/RZRussel/hotstuff-model-checking.

[25] K. Ziborov, *HotStuff TLA+ Specifications*, 2020. [Online]. Available: https://github.com/RZRussel/hotstuff-model-checking/blob/master/HotStuffAlpha.tla.

[26] V. Kukharenko, *HotStuff TLA+ Specifications*, 2020. [Online]. Available: https://github.com/RZRussel/hotstuff-model-checking/blob/master/ApalacheHotStuffCutted.tla.

MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 27, NO. 4, 2020
journal homepage: www.mais-journal.ru

THEORY OF COMPUTING

Architecture of the Formally-Verified Distributed Ledger System InnoChain

L. A. Merkin-Janson[1], R. M. Rezin[1], N. K. Vasilyev[1,2]

DOI: 10.18255/1818-1015-2020-4-472-487

[1]Innopolis University, 1 Universitetskaya, Innopolis, 420500, Russia.
[2]National Research University Higher School of Economics, 20 Myasnitskaya St., Moscow 101000, Russia.

MSC2020: 93A30, 68Q60
Research article
Full text in Russian

Received November 17, 2020
After revision December 3, 2020
Accepted December 16, 2020

In this paper we consider the software architecture of InnoChain, a distributed ledger system (DLS) with 5 levels of formal verification, including a formally-verified underlying operating system (OS). The objective of this architecture is to achieve a higher level of DLS dependability compared to more traditional software architectures and quality assurance (QA) methods. The architecture of InnoChain includes (1) a programming language for smart contracts which is a domain-specific language with formal semantics embedded into CakeML, which is a functional language of the ML family; this allows us to carry out formal verification of smart contracts' correctness properties using higher-order logic systems, such as HOL4; (2) trusted compilation of smart contracts into the machine code using the verified compiler available for CakeML, rather than relying on a virtual machine for execution of smart contracts; (3) using CakeML for implementation of InnoChain node functionality which allows for formal verification of code correctness and trusted compilation into the machine code; (4) formal verification of the consensus protocol used InnoChain, namely HotStuff BFT; (5) using seL4, a formally-verified microkernel, as the underlying OS for InnoChain instead of more traditional general-purpose OSes such as Linux. The proposed verified architecture will allow InnoChain to be used in mission-critical applications, such as the decentralized Aircraft Fuelling Control System which is currently under development for JSC Aeroflot, the Russian national air carrier.

Keywords: distributed ledger systems; formal verification; HOL4; CakeML; seL4

INFORMATION ABOUT THE AUTHORS

Leonid Al'bertovich Merkin-Janson correspondence author	orcid.org/0000-0002-8427-2200. E-mail: l.merkin@innopolis.ru Professor, Doctor of Mathematics, Delft University of Technology, Dr.
Ruslan Maratovich Rezin	orcid.org/0000-0002-0604-2645. E-mail: r.rezin@innopolis.ru Head expert in formal verification methods at Innopolis University, Master degree in Computer Science, MSc.
Nikolay Konstantinovich Vasilyev	orcid.org/0000-0003-3112-5482. E-mail: n.vasilev@innopolis.ru Lead Developer.

Funding: Ministry of Digital Development, Communications and Mass Media of the Russian Federation and Russian Venture Company (Agreement No. 004/20 dd. 20.03.2020, IGK 0000000007119P190002).

For citation: L. A. Merkin-Janson, R. M. Rezin, and N. K. Vasilyev, "Architecture of the Formally-Verified Distributed Ledger System InnoChain", *Modeling and analysis of information systems*, vol. 27, no. 4, pp. 472-487, 2020.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 27, № 4, 2020
сайт журнала: www.mais-journal.ru

THEORY OF COMPUTING

Архитектура формально-верифицированной системы распределенного реестра InnoChain

Л. А. Меркин[1], Р. М. Резин[1], Н. К. Васильев[1,2]

DOI: 10.18255/1818-1015-2020-4-472-487

[1]Университет Иннополис, Университетская, д.1, г. Иннополис, 420500 Россия.
[2]Национальный исследовательский университет «Высшая школа экономики», ул. Мясницкая, д. 20, г. Москва, 101000 Россия.

УДК 519.7
Научная статья
Полный текст на русском языке

Получена 17 ноября 2020 г.
После доработки 3 декабря 2020 г.
Принята к публикации 16 декабря 2020 г.

В настоящей работе рассматривается архитектура системы распределенного реестра (CPP) InnoChain. Основной целью этой архитектуры является реализуемость 5-ти уровней формальной верификации программного обеспечения (ПО) системы InnoChain, включая операционное окружение. Методы формальной верификации являются основными методами обеспечения качества ПО с критическими требованиями по надежности, но до сих пор они не находили широкого применения в CPP. Архитектура InnoChain включает (1) предметно-ориентированный язык смарт-контрактов с формальной семантикой, встроенный в функциональный язык CakeML (диалект языка ML), что позволяет осуществлять формальную верификацию свойств корректности смарт-контрактов в системах логики высших порядков (например, HOL4); (2) верифицированную трансляцию смарт-контрактов в машинный код с использованием компилятора CakeML вместо использования виртуальных машин для исполнения смарт-контрактов; (3) реализацию функционала узла CPP также на CakeML с формальной верификацией свойств корректности и с верифицированной трансляцией исходного кода узла в машинный код; (4) формальную верификацию протокола консенсуса CPP (HotStuff BFT); (5) использование формально-верифицированного микроядра seL4 в качестве операционного окружения CPP вместо операционных систем общего назначения. Предлагаемая архитектура открывает возможности для использования CPP InnoChain в критических по надежности приложениях, в частности, в системе управления заправкой воздушных судов ПАО Аэрофлот.

Ключевые слова: системы распределенного реестра; формальная верификация; HOL4; CakeML; seL4

ИНФОРМАЦИЯ ОБ АВТОРАХ

Леонид Альбертович Меркин автор для корреспонденции	orcid.org/0000-0002-8427-2200. E-mail: l.merkin@innopolis.ru Профессор, научный руководитель Лидирующего исследовательского центра в области формально-верифицированных систем распределенного реестра.
Руслан Маратович Резин	orcid.org/0000-0002-0604-2645. E-mail: r.rezin@innopolis.ru Главный эксперт формальных методов верификации, степень магистра по направлению “Информатика и вычислительная техника”.
Николай Константинович Васильев	orcid.org/0000-0003-3112-5482. E-mail: n.vasilev@innopolis.ru Ведущий разработчик.

Финансирование: Министерство цифрового развития, связи и массовых коммуникаций РФ и АО "Российская венчурная компания"(договор №004/20 от 20.03.2020, ИГК 0000000007119P190002).

Для цитирования: L. A. Merkin-Janson, R. M. Rezin, and N. K. Vasilyev, “Architecture of the Formally-Verified Distributed Ledger System InnoChain”, *Modeling and analysis of information systems*, vol. 27, no. 4, pp. 472-487, 2020.

Введение

На сегодняшний день системы распределенного реестра (СРР) находят свое применение в различных направлениях информационных технологий и в различных секторах экономики, начиная с традиционного для СРР финансового сектора и заканчивая специфическими применениями в области робототехники, транспорта, связи и многих других. В Российской Федерации на государственном уровне СРР идентифицированы как одна из приоритетных «сквозных» информационных технологий, то есть таких, которые применимы практически во всех отраслях национальной экономики и в сфере государственного управления.

СРР представляют собой распределенные системы, функционирующие в реальном масштабе времени. К надежности СРР (включая корректность, безопасность, отказоустойчивость и иные качественные параметры, ассоциируемые с общим понятие надежности) предъявляются особые требования. Это связано с тем, что свойства СРР, с одной стороны, по определению обеспечивают доступ к данным для всех участников системы без ограничений, а также согласованность и целостность данных. С другой стороны, любой дефект в исходном коде СРР может обернуться серьезной уязвимостью для системы в целом, ввиду того, что нет единой точки доступа для устранения проблемы.

«Традиционные» методы обеспечения качества программного обеспечения (например, ручное и автоматическое тестирование, аудит программного кода, открытый доступ к коду для сообщества, проектирование по контракту) недостаточно эффективны для СРР. Например:

- Тестирование в принципе не может использоваться как основной метод обеспечения надежности систем реального времени, включая СРР, так как никакой объем тест-кейсов не обеспечит достаточного покрытия, ввиду того, что время является одним из входных параметров системы. Более того, использование тестирования в качестве основного метода обеспечения надежности систем реального времени скорее вредно, чем полезно, так как ведет к возникновению «ложного чувства безопасности» [1].
- Аудит кода и публикация открытого программного кода являются субъективными методами обеспечения качества ПО, надежность которых трудно оценить количественно. Кроме того, в случае осуществления аудита кода путем привлечения сторонних коммерческих организаций существует теоретическая возможность возникновения конфликта интересов, влияющих на исход аудита.
- Методология проектирования по контракту в целом позволяет улучшить раннее обнаружение и изоляцию ошибок ПО. Однако в случае СРР, множественные узлы системы часто обрабатывают одни и те же блоки данных. В случае возникновения ошибки, она потенциально может затронуть все узлы СРР сразу.

На протяжении последних 30 лет, основной методологией обеспечения качества систем реального времени с критическими требованиями по надежности являются различные формальные методы, например, методы формальной спецификации, верифицированной детализации и автоматической генерации программного кода [2].

Однако в области СРР формальные методы пока остаются менее популярными, чем «традиционные» методы, перечисленные выше. Одним из примеров использования формальных методов в СРР является система Tezos с предметно-специфическим языком смарт-контрактов Archetype [3], допускающим ручную и автоматическую верификацию свойств смарт-контрактов.

Тем не менее уязвимости в СРР могут возникать не только на уровне программной логики смарт-контрактов, но и на других архитектурных уровнях системы. Например, широко известны различные уязвимости в системе исполнения смарт-контрактов СРР Ethereum, проистекающие, в частности, из нарушений целостности стека виртуальной машины при некорректных сценариях вызовов рекурсивных функций [4]. Кроме того, уязвимости могут возникать в других функцио-

нальных компонентах, относящихся к узлу СРР, в протоколе консенсуса СРР, а также ввиду ошибок в коде окружения в котором функционирует и с которым взаимодействует узел сети СРР, например, в операционной системе (ОС).

Таким образом, для СРР, предназначенных для использования в критических по надежности индустриальных приложениях, необходимо систематическое применение методов формальной верификации не только на уровне логики смарт-контрактов, но и на всех вышеперечисленных архитектурных уровнях системы. Данный принцип был положен в основу архитектуры СРР InnoChain, которая в настоящий момент находится в стадии реализации. Одной из областей индустриального применения системы СРР InnoChain является распределенная система управления заправкой воздушных судов ПАО «Аэрофлот».

Настоящая статья представляет общую архитектуру СРР InnoChain и методы формальной верификации, применяемые на каждом из 5-ти уровней. Статья организована следующим образом. В разделе 1 приводятся общие сведения об уровнях архитектурной организации и формальной верификации СРР InnoChain. В разделе 2 рассматривается функционирование СРР InnoChain как распределенной системы реального времени и выбор протокола консенсуса. В разделе 3 рассматриваются вопросы выбора языка программирования для реализации СРР InnoChain, в частности, для реализации предметно-ориентированного языка смарт-контрактов. В разделе 4 рассматриваются вопросы выбора высоконадежного операционного окружения для СРР InnoChain. Раздел 5 представляет выводы и будущие задачи.

1. Обзор архитектуры СРР InnoChain и уровней формальной верификации

1.1. Уровень языка смарт-контрактов

С точки зрения пользователя, СРР InnoChain представляет собой распределенную децентрализованную базу данных со встроенным языком программирования — предметно-ориентированным языком смарт-контрактов, которые выполняются на всех узлах СРР и изменяют (дополняют) ее состояние.

Смарт-контракты разрабатываются на предметно-ориентированном языке, встроенном в язык CakeML [5]. В текущей версии СРР InnoChan, находящейся в процессе разработки, в качестве языка смарт-контрактов выступает сам CakeML. В перспективе предполагается возможность трансляции других языков программирования в абстрактное синтаксическое дерево, реализованное в терминах конструкций языка CakeML.

Для языка смарт-контрактов требуется наличие формальной семантики, позволяющей формулировать и доказывать свойства функциональной корректности смарт-контрактов. В частности, формальная операционная семантика существует для языка CakeML.

1.2. Уровень исполнения смарт-контрактов

В большинстве современных СРР, смарт-контракты транслируются в байт-код, исполняемый на виртуальной машине, входящей в состав каждого узла СРР. Однако реализация такой виртуальной машины для InnoChain потребовала бы, в частности, разработки формальной семантики байт-кода и формально-верифицированного компилятора из языка смарт-контрактов в байт-код.

Вместо этого, в InnoChain используется прямая трансляция кода смарт-контрактов (реализованных на языке CakeML или на языке, встроенном в CakeML) в машинный код процессора x86_64 (возможна также трансляция в машинный код ARM) с использованием верифицированного компилятора CakeML. Это гарантирует функциональную корректность генерируемого кода.

1.3. Уровень функционала узла

Узел СРР InnoChain осуществляет такие функции, как сетевые взаимодействия, хранение текущего состояния распределенного реестра и исполнение смарт-контрактов. Функционал узла также

реализуется на CakeML, что делает возможным формальную верификацию корректности алгоритмов узла и генерацию машинного кода узла с помощью формально-верифицированного компилятора CakeML.

1.4. Уровень протокола консенсуса СРР

Целостность состояния данных в СРР обеспечивается с помощью протокола консенсуса, исполняемого узлами системы. В качестве протокола консенсуса в InnoChain выбран HotStuff BFT [6], в отношении которого осуществлена формальная верификация свойства *Safety* (см. Раздел 2).

1.5. Уровень операционной системы

Для высоконадежных приложений СРР желательно использование операционного окружения, которое также является формально-верифицированным. С этой целью, наряду с ОС Linux, в СРР InnoChain может использоваться формально-верифицированное микроядро seL4 [7]. Более подробно этот вопрос рассмотрен в Разделе 4.

2. Функционирование системы распределенного реестра и выбор протокола консенсуса

Под системой распределенного реестра (СРР) обычно понимают множество вычислительных станций (далее узлов), объединенных в общую сеть с целью изменения единого состояния на основе команд от пользователей системы — транзакций. Структуры данных, описывающие состояние и транзакции СРР, варьируются в зависимости от целей, для которых используется система.

В случае СРР InnoChain, данные структуры определяются как изображено на Рис. 1. В структуре данных аккаунта хранится: идентификатор (`accountId`), множество открытых ключей (`publicKeys`, для отправки транзакций от имени аккаунта ее необходимо подписать каждым соответствующим секретным ключом), тип аккаунта (`type`), количество транзакций аккаунта (`nonce`), код смарт-контракта (`code`), хранилище дополнительных данных аккаунта (`storage`). В структуре данных транзакции хранится: время создания (`timestamp`), идентификатор аккаунта, от имени которого отправлена транзакция (`from`), идентификатор аккаунта для развертывания смарт-контракта (`to`), порядковый номер транзакции (`nonce`, который должен соответствовать полю nonce аккаунта отправителя), начальное состояние смарт-контракта (`data`).

Как можно заметить, *состояние* СРР определяется как частичная функция s, ставящая в соответствие идентификатору в формате GUID некоторое значение структуры данных *Account*:

$$s : GUID \longrightarrow Account \tag{1}$$

где $GUID = \{0,1\}^{128}$

Функция s является частичной ввиду того, что не для каждого идентификатора могут существовать данные аккаунта.

Состояние СРР изменяется при исполнении транзакций, которые для оптимизации скорости их обработки группируются в `блоки`. Корректность изменения состояния СРР контролируется `протоколом консенсуса` — набором правил обмена сообщениями, которым должен следовать узел для поддержания правильного состояния системы. На основе проведенного анализа, для реализации в InnoChain была выбрана интерпретация протокола консенсуса HotStuff BFT [6] (далее протокол консенсуса) от проекта Libra [8].

Будем называть узлы, которые следуют протоколу консенсуса, `правильными`, а которые отклоняются от него — `неисправными`. Предполагается, что $N \geq 3f + 1$ узлов, где f - максимальное число неисправных узлов. Узлы, объединенные в общую сеть, согласуют между собой изменение состояния распределенного реестра на основе блоков транзакций. Также предполагается, что верно

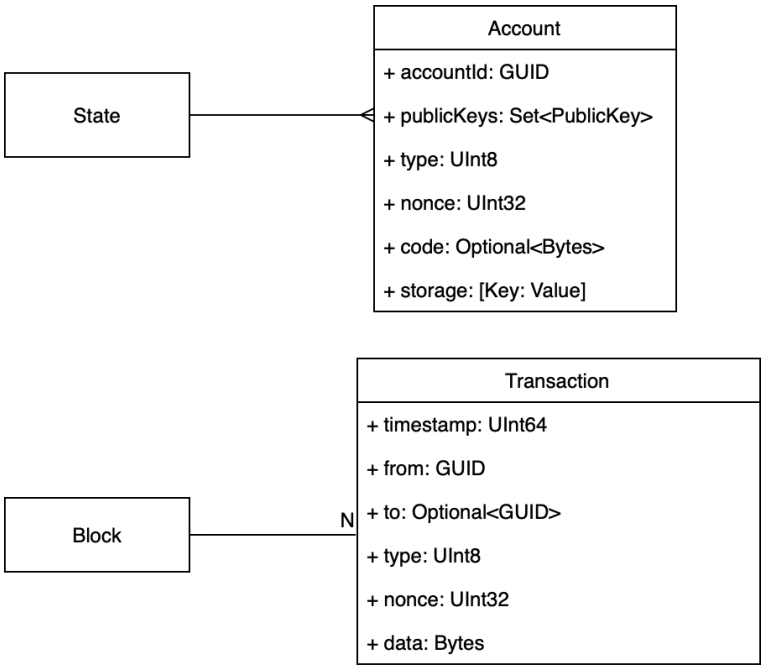


Fig. 1. Principal data structures of the InnoChain distributed ledger system

Рис. 1. Основные структуры данных CPP InnoChain

условие частичной синхронности: время доставки сообщения от одного узла другому не превосходит Δ после некоторого момента времени стабилизации сети (*GST, Global Stabilization Time*). Наконец, считаем, что вычислительной мощности узла недостаточно для того, чтобы «взломать» электронно-цифровую подпись или найти коллизию хэш-функции. Если описанные предположения верны, то должно гарантироваться, что протокол консенсуса удовлетворяет свойствам *Safety* и *Liveness*. *Safety* говорит о том, что для двух различных правильных узлов существует момент времени $t \geq GST$, когда последовательности принятых ими блоков совпадут, а до этого момента одна последовательность является префиксом другой. Свойство *Liveness* означает, что если правильный узел получил транзакцию, которая не противоречит состоянию сети, то обязательно узлами будет принят новый блок, который изменит состояние сети.

Обозначим через S множество всевозможных состояний CPP. Тогда под исполнением транзакции можно понимать функцию E, которая преобразует транзакцию и предыдущее состояние в новое состояние CPP:

$$E : Transaction \times S \longrightarrow S \tag{2}$$

Под смарт-контрактом обычно понимают программу, написанную на специальном языке программирования для децентрализованного исполнения узлами CPP. Сторонние разработчики имеют возможность развернуть новые смарт-контракты в уже запущенной сети CPP. При этом машинный код, полученный после компиляции программы, и начальное состояние смарт-контракта сохраняются в структуре данных соответствующего аккаунта CPP. Данный машинный код загружается в память и выполняется узлом при исполнении специальных транзакций, содержащих сигнатуру функции смарт-контракта и параметры для ее вызова.

Важным процессом при обработке вызовов смарт-контракта является загрузка и исполнение кода смарт-контракта. Ввиду того, что узлы могут работать в разном окружении (например, использовать разные операционные системы), создание отдельного процесса смарт-контракта и передача ему управления для выполнения функции автоматически устанавливает зависимость корректно-

сти работы программы от ее окружения. По этой причине наиболее популярным подходом для исполнения смарт-контрактов является их интерпретация специально разработанной виртуальной машиной. Более того, существуют CPP, для которых была верифицирована функциональная корректность виртуальной машины. Тем не менее, данный подход не гарантирует корректность взаимодействия между узлом CPP и операционной системой, например, для передачи данных по сети. В разделе 4 рассматривается возможность использования формально верифицированной операционной системы, что делает возможным использование первого подхода для исполнения смарт-контрактов, а также увеличивает надежность взаимодействия между процессом узла и операционной системой.

Однако помимо использования надежной операционной системы, необходимо также обеспечить функциональную корректность работы узла и смарт-контрактов. Для достижения данной цели необходимо выбрать набор инструментов формальной верификации и подходящий для этого язык программирования. Как будет показано в разделе 3, наиболее подходящим на момент написания работы является язык программирования CakeML [9] в связке с программным инструментом дедуктивных методов верификации HOL4 [10].

На Рис. 2 изображена верхнеуровневая схема формальной верификации CPP. Описание протокола консенсуса HotStuff BFT, являющееся отдельной частью общей спецификации логики работы узла, преобразуется в модель для верификации свойств безопасности методом Model Checking [11]. В то же время алгоритмы функционирования узла разрабатываются на языке CakeML. С одной стороны, это позволяет получить гарантии формально верифицированной компиляции, а с другой стороны — иметь возможность формальной верификации свойств исходного кода в HOL4, используя метод характеристических формул [12].

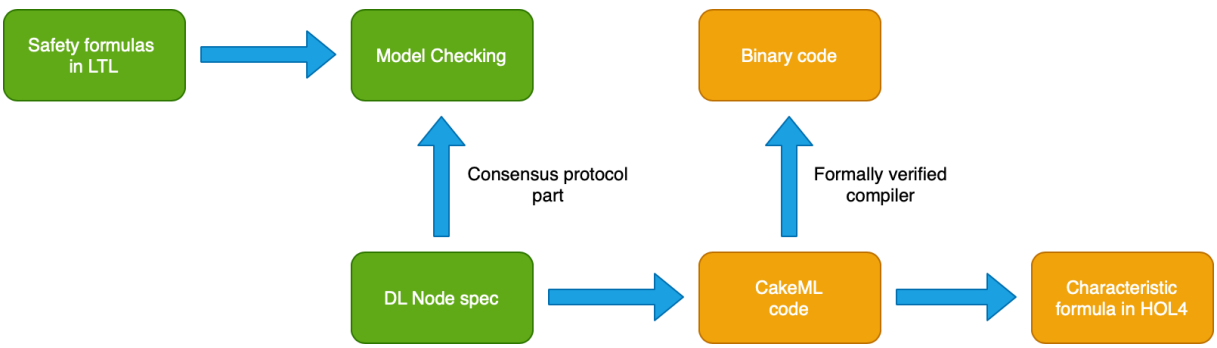


Fig. 2. High-level view of the formal verification scheme for the InnoChain distributed ledger system

Рис. 2. Верхнеуровневое представление схемы формальной верификации системы распределенного реестра

3. Выбор языка программирования

Необходимым условием для возможности формальной верификации программ на некотором языке программирования является наличие формальной семантики и программных инструментов для автоматизации проведения доказательств функциональной корректности. В рамках данной работы необходимо было выбрать язык, который одновременно подходил бы для разработки логики узла CPP и также мог бы служить в качестве языка смарт-контрактов. Таким образом, было исследовано четыре направления, изображенных на Рис. 4.

Три из четырех направлений связаны с языком CakeML. CakeML — это язык программирования, близкий к Standard ML. Формальная семантика CakeML [9] выражена в терминах логики высшего порядка (higher-order logic, HOL) и реализована в программном обеспечении для автоматизации процесса доказательств дедуктивным методом — HOL4 [10]. Для CakeML научным сообществом

разработан набор инструментов и теорий, реализованных в терминах HOL4. Основным инструментом является транслятор термов [13] из системы HOL4 в абстрактные синтаксические деревья CakeML. Поскольку семантика CakeML описана в HOL4, то появляется возможность записывать любые программы на CakeML в HOL4 — о таких программах говорят, что они «глубоко вложены» в HOL. Таким образом, определения, функции и типы данных в логике HOL можно передать на вход транслятору — и получить их CakeML-аналоги, глубоко вложенные в HOL. Важно отметить, что вместе с кодом на CakeML транслятор генерирует сертификат — доказательство того, что функции, записанные в HOL, работают так же как и полученный код на CakeML.

Обычные функции в HOL4 описывают «чистые» математические вычисления, то есть такие, в которых нет ввода-вывода и взаимодействия с внешним миром. В рамках проекта CakeML была разработана do-нотация для HOL4, которая позволяет описывать программы, в которых присутствует ввод-вывод, непосредственно в HOL4. Также создан монадический транслятор [14] — аналог обычного транслятора, но позволяющий транслировать функции HOL4, использующие do-нотацию. Таким образом, можно прямо в HOL4 записать программу, которая получает ввод с клавиатуры, обрабатывает введенные данные и выводит результаты на экран, а затем воспользоваться монадическим транслятором, чтобы автоматически получить аналогичную программу на CakeML (глубоко вложенную в HOL).

Другим важным подходом для анализа программ на CakeML является метод характеристических формул [12]. Данный инструмент позволяет формально верифицировать свойства корректности функций, изначально реализованных на CakeML, а не на HOL4. Свойства корректности предлагается задавать в виде формул сепарационной логики [15].

Авторами CakeML также разработан формально верифицированный компилятор для данного языка. Идея реализации этого компилятора, изображенная на Рис. 3, состоит в том, чтобы сначала записать компилятор в HOL4, а затем использовать транслятор для получения эквивалентного кода на CakeML. Затем этот код на CakeML подается на вход компилятору, реализованному на HOL4, и тем самым на выходе получается машинный код компилятора с гарантией функциональной корректности ввиду доказательств в HOL4.

Fig. 3. Sequence of transformations for generation of a formally-verified machine code using the CakeML compiler and HOL4 proofs.

Рис. 3. Последовательность преобразований для получения формально верифицированного машинного кода компилятора языка CakeML на основе доказательств в HOL4

Параллельно с HOL4 была рассмотрена популярная в научном сообществе система Isabell/HOL [16]. Идея была все та же: записать программу в Isabelle/HOL, а затем использовать формально верифицированный транслятор [17] для получения эквивалентного кода на CakeML. Однако данный подход оказался еще не готов к практическому применению: получаемое после трансляции абстрактное синтаксическое дерево программы приходится дорабатывать вручную для успешной компиляции компилятором CakeML. В частности, код «с эффектами» (например, с вводом-выводом) не поддерживается этим транслятором. Но более существенной проблемой является то, что структуры данных Isabelle (в частности, списки) не транслируются в структуры данных CakeML.

Наконец, последней опцией была возможность использовать систему Coq. Для данной системы научным сообществом был разработан фреймворк Verified Software Toolchain (VST) [18]. Он

представляет собой набор инструментов и библиотек для Coq, созданный с целью верификации программ, написанных на ограниченном подмножестве языка C (Clight). Проект VST тесно связан с другим проектом верификации на Coq — формально верифицированным компилятором CompCert языка C [19]. Несмотря на то, что большинство исследований, связанных с VST, ориентированы на язык C, изначально этот фреймворк разрабатывался как универсальный инструмент, позволяющий гарантировать, что утверждения статического анализатора относительно программы на некотором языке сохраняются для машинного кода, который исполняется в рамках некоторой операционной системы.

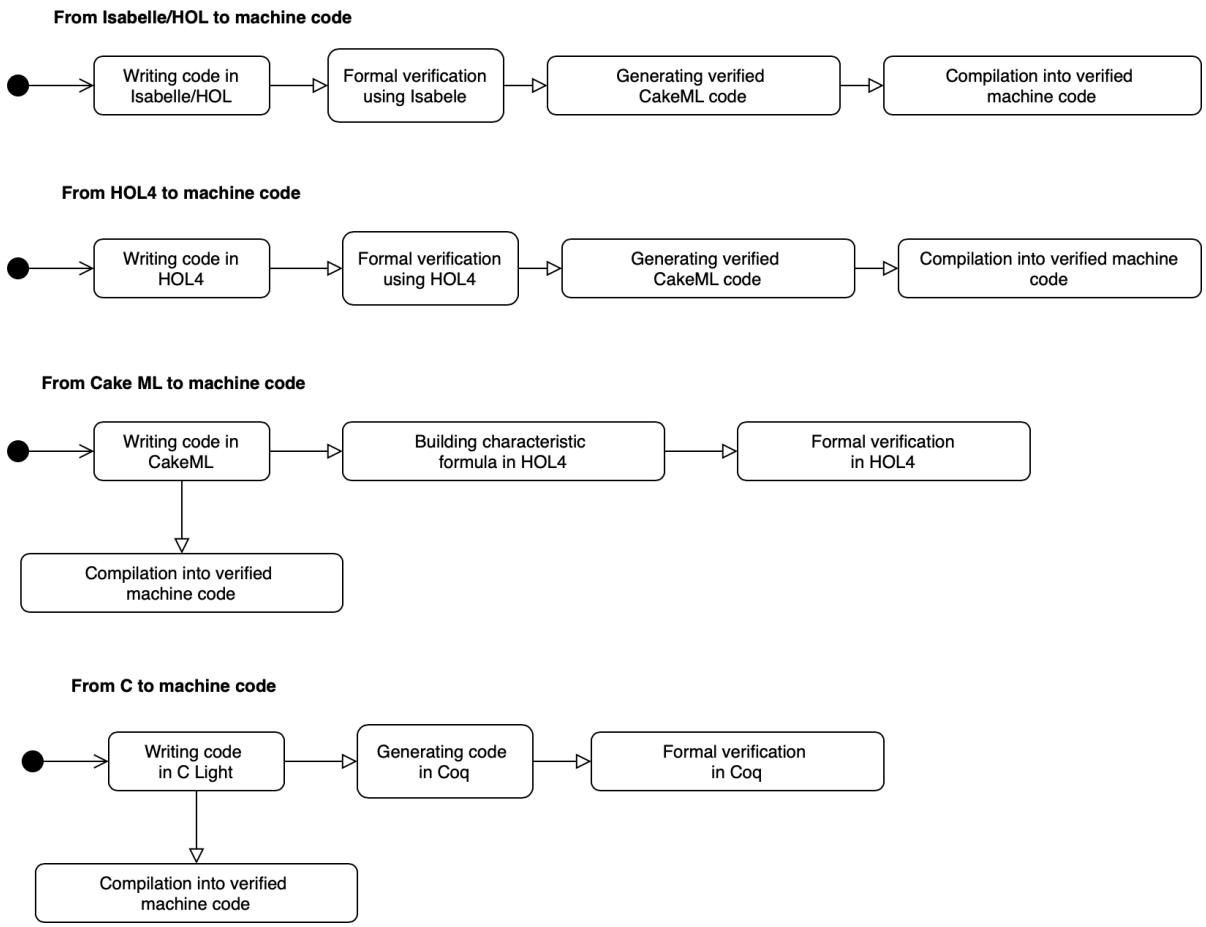


Fig. 4. The technologies and tools considered as the basis for implementation of the formally-verified distributed ledger system InnoChain

Рис. 4. Рассмотренные варианты технологий и инструментов для разработки формально верифицированной СРР

В виду совместимости с формально верифицированной операционной системой и функциональной парадигмой, которая на практике дает больше гарантий корректности кода, в качестве языка для разработки узла СРР, а также в качестве базового языка смарт-контрактов был выбран CakeML, а не Clight. В качестве системы для проведения формальных доказательств корректности программ на CakeML была выбрана система HOL4 ввиду того, что транслятор из Isabelle в CakeML еще не готов к практическому применению. Более того, для упрощения разработки смарт-контрактов было принято решение реализовать язык смарт-контрактов как предметно-ориентированный язык (DSL), специфичный для каждой отдельной области применения. В частности, свой язык смарт-контрактов разрабатывается для задачи автоматизации управления заправкой воздушных судов в

ПАО «Аэрофлот». Далее, для каждого DSL необходимо разработать транслятор для преобразования смарт-контрактов в эквивалентную программу на языке CakeML. Этот транслятор можно реализовать непосредственно в HOL4 и там же верифицировать его функциональную корректность. Эта идея уже была успешно применена разработчиками CakeML для реализации формально верифицированного компилятора для него. Таким образом, формально верифицированный транслятор из DSL в CakeML позволит использовать существующие инструменты для анализа корректности смарт-контрактов на DSL.

Тем не менее, удобная для разработчиков смарт-контрактов реализация DSL требует глубокого анализа соответствующей отрасли применения. В связи с этим, было решено реализовать первый смарт-контракт для конкретной задачи управления заправкой воздушных судов на самом CakeML, а затем, проанализировав это решение, прийти к понимаю конструкций, которые необходимо включить в DSL. В конечном итоге, разработчики смарт-контрактов получат возможность писать код как на DSL, так и напрямую на CakeML.

4. Выбор операционной системы

ОС играет важную роль в работе узла, предоставляя возможность взаимодействовать с другими узлами через сетевой интерфейс и сохранять данные, используя файловую систему. К сожалению, существующие операционные системы, получившие широкое распространение, разрабатывались без использования формальных методов верификации и имеют большую кодовую базу. Наиболее популярная на сегодняшний день операционная система Linux содержит порядка двадцати миллионов строк исходного кода. Несмотря на то, что за последние годы были сделаны попытки формально верифицировать отдельные части ядра данной операционной системы [20], бóльшая часть кода остается без формализации. Данный факт, помимо рисков возникновения дефектов при взаимодействии между узлом CPP и операционной системой, также затрудняет процесс формальной верификации исходного кода узла, ввиду необходимости построения формальной модели объекта (ОС), с которым осуществляется взаимодействие.

С другой стороны, микроядро seL4 [7] имеет кодовую базу порядка десяти тысяч строк, и его функциональная корректность была формально верифицирована для наиболее популярных на сегодняшний день архитектур процессоров и платформ. Данный факт позволяет сформировать гипотезу о возможности функционирования узла CPP с формально верифицированным исходным кодом под управлением операционной системы, код которой также формализован и формально верифицирован. Тем не менее, следует учитывать тот факт, что на данный момент микроядро seL4 еще не получило широкого распространения. По этой причине может потребоваться разработка нового или доработка существующего функционала seL4, необходимого для корректной работы узла CPP, включая сетевое взаимодействие и взаимодействие с базами данных.

В данной работе была исследована возможность использования seL4 в качестве операционной системы узла CPP. Кроме того, был разработан инструментарий для упрощения развертывания приложений для seL4 на реальном оборудовании, который может быть полезен в других исследованиях.

В отличие от полноценных ОС, таких как Linux, seL4 — это операционная система на базе микроядра. Это означает, что в режиме «ядра» исполняется урезанный функционал, а все остальные функции исполняются в «пользовательском» режиме. В случае с seL4, на уровне ядра реализована следующая функциональность: взаимодействие между процессами (IPC), управление потоками и виртуализация памяти. Преимуществом данного подхода является тот факт, что объем доверенного кода, который служит основой для работы приложений, значительно ниже чем у «традиционных» операционных систем и, таким образом, содержит потенциально меньше ошибок. Кроме того, меньший объем кодовой базы проще формализовать и для него формально верифицировать выполнение критических свойств системы, что и было сделано авторами seL4.

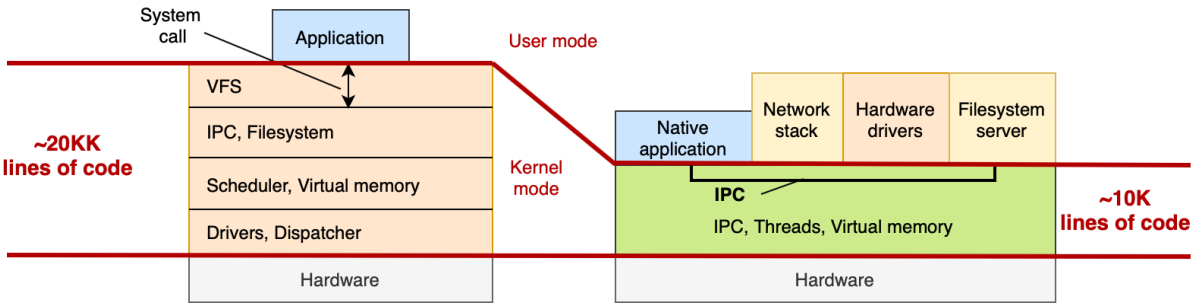


Fig. 5. Comparison of the seL4 microkernel with "traditional" monolithic kernel OSes.

Рис. 5. Сравнение seL4 с «традиционными» ОС

Как можно заметить из Рис. 5, seL4 содержит доверенную кодовую базу, которая исполняется в режиме «ядра» и наиболее критична с точки зрения безопасности. Вместе с тем, seL4 не содержит сервисов для доступа к оборудованию, а вместо этого предлагается реализовать эти сервисы как приложения для работы в «пользовательском» режиме. Существенным недостатком данного подхода является то, что стандартные компоненты для работы с оборудованием (файловая система, работа с сетью) необходимо либо разрабатывать «с нуля», либо портировать их из других ОС, но в любом случае это требует существенных трудозатрат. Тем не менее, в данном направлении сообществом seL4 уже проделана большая работа. Например, была адаптирована библиотека `picotcp` [21] для использования приложениями сетевого стека TCP/IP. Но все же в случае с узлом CPP остается открытым вопрос об интеграции приложений на базе seL4 с базами данных.

Взаимодействие между приложениями осуществляется одним из следующих способов: защищенные процедурные вызовы (RPC), общая память (Shared Memory) или сигналы (Notifications). По историческим соображениям данные способы взаимодействия объединяются под общим названием IPC (Inter-Process Communication). При взаимодействии между приложениями, seL4 выступает в качестве медиатора, отвечая за контроль доступа и корректность передачи параметров. Таким образом, работу системы под управлением seL4 можно рассматривать как работу отдельного приложения в «песочнице», при этом взаимодействие с остальными приложениями (включая драйверы оборудования) осуществляется через описанные механизмы IPC микроядра.

Формальное доказательство корректности seL4 выполнено его авторами с использованием системы верификации Isabelle/HOL [16] и изначально насчитывало 200 000 строк кода. Для формализации подмножества языка C, на котором написано микроядро, в Isabelle был записан синтаксический анализатор, который трансформирует исходный код в семантически эквивалентную запись на языке математической логики HOL. После формализации и преобразования требований к seL4 в формулы спецификации, формальная верификация проводилась с использованием инструментов Isabelle. Однако доказательство корректности кода на C еще не означает, что компилятор не привнесет ошибок в машинный код, так как для компиляции seL4 использовался стандартный компилятор gcc, а не формально-верифицированный компилятор CompCert.

Для доказательства корректности машинного кода в Isabelle были записаны трансляторы для преобразования исходного кода на языке C и машинного кода в графовое представление (control flow) с сохранением семантики. Таким образом, для доказательства корректности работы микроядра остается доказать семантическую эквивалентность полученных графовых моделей. Для этой цели использовался набор SMT решателей [22]. На Рис. 7 приведены основные этапы доказательства эквивалентности машинного кода исходному коду. На Рис. 6 приведены основные этапы проведения общего доказательства корректности seL4, а также свойства корректности, формализованные в спецификации:

- конфиденциальность: приложение не сможет прочитать данные или извлечь информацию другим образом, если у него нет доступа на чтение;
- целостность: приложение не сможет изменить данные, если у него нет доступа для этой операции;
- доступность: одно приложение не может помешать другому приложению использовать ресурс, если у второго есть на это доступ.

Конечно, формальное доказательство требует некоторых предположений о контексте, в котором оно проводится, а также о контексте, в котором работает микроядро. Данные предположения должны быть явно закреплены. Важно отметить, что такой подход может помочь выявить проблемы корректности системы на ранней стадии, ввиду четкого понимания ограничений, исходящих от закрепленных предположений о работе системы.

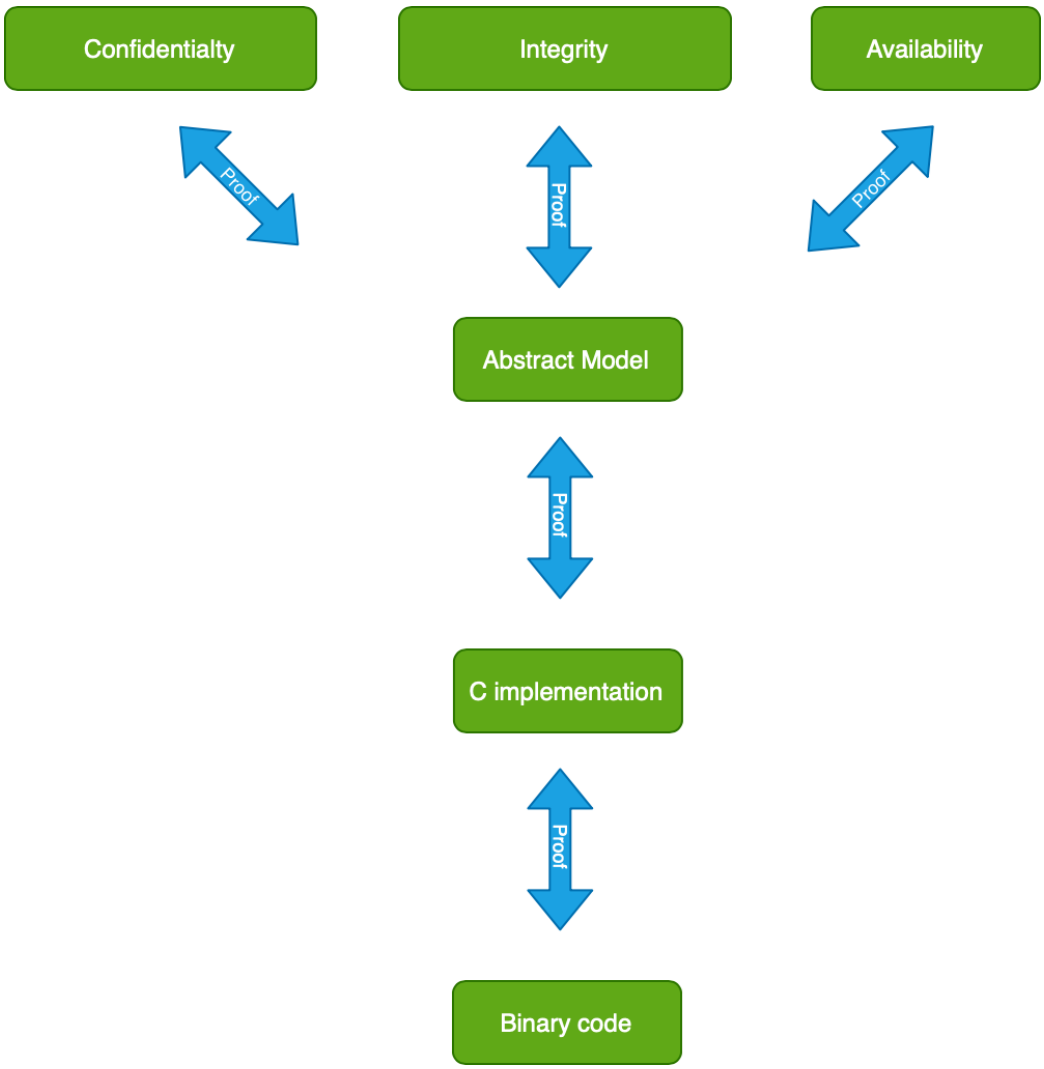


Fig. 6. Main stages of seL4 correctness verification

Рис. 6. Основные этапы проведения доказательства корректности seL4

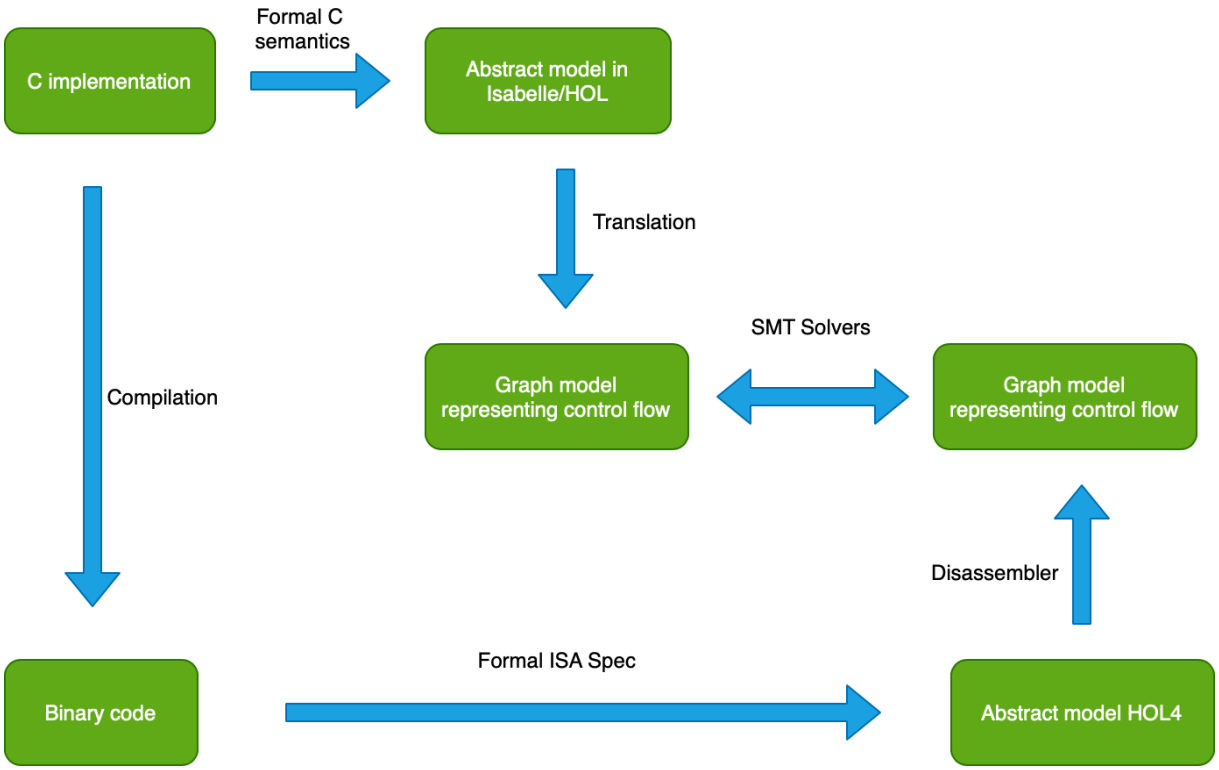


Fig. 7. Main stages of proving the functional equivalence of seL4 machine code and its source C code

Рис. 7. Этапы проведения доказательства эквивалентности машинного кода seL4 исходному коду на языке C

Формальная верификация seL4 его авторами была проведена с учетом следующих предположений:

- оборудование функционирует корректно;
- спецификация удовлетворяет ожиданиям — это самое сложное, так как не всегда легко удостовериться в том, что математическая формула действительно соответствует тому, что написано в требованиях к системе;
- сама система верификации функционирует корректно;
- код загрузки seL4 корректен: 1200 строк кода отвечающего за старт/загрузку seL4 не верифицировано;
- виртуальная память: механизм разделения памяти между ядром и приложениями, а также между приложениями, верифицирован в предположении о стандартном поведении виртуальной памяти при исполнении кода ядра;
- ассемблер: около 340 строк кода для доступа к оборудованию на ассемблере не верифицировано;
- прямой доступ к памяти (DMA): при формальной верификации было сделано предположение о том, что только процессор и блок управления памятью (MMU) имеют доступ к памяти; таким образом, DMA либо должен отсутствовать вовсе, либо нужно сделать предположение о корректной работе драйвера для этого контроллера;
- модель оборудования учитывает все каналы передачи информации для доказательства конфиденциальности;
- в микроядре имеется статический набор процессов, не изменяющийся после развертывания и запуска системы.

Однако стоит заметить, что машинный код может отличаться в зависимости от архитектуры процессора, поэтому необходимо провести формальную верификацию для каждой интересующей конфигурации в отдельности. Кроме того, seL4 может функционировать и в режиме гипервизора, поэтому желательно осуществить верификацию также и этого режима. В настоящее время, полный процесс формальной верификации, описанный выше, был проведен только для архитектуры ARMv7 (Sabre Light), а для x86-64 формально верифицирована функциональная корректность.

На момент написания работы существуют два способа развертывания приложений на базе seL4. Первый способ заключается в использовании очень гибкого фреймворка Genode [23]. Он позволяет запустить образ микроядра, а затем динамически компилировать и исполнять приложения, аналогично тому, как это делается в «традиционных» ОС. Тем не менее, ввиду того, что Genode поддерживает и другие микроядра, только общая для всех ядер функциональность доступна пользователям этого фреймворка. Например, при развертывании приложений seL4 на основе Genode не будет доступен режим гипервизора. Более того, использование Genode исключает гарантии формальной верификации seL4, так как эти гарантии основаны на предположении о статическом наборе процессов в системе.

Второй способ заключается в сборке и запуске статической конечной системы на базе seL4 с использованием фреймворка CAmkES [24]. Этот фреймворк позволяет описать компоненты системы и взаимодействия между ними на специальном языке CAmkES ADL (Architecture Description Language). Базовые элементы описания следующие:

- **Компоненты** — приложения, в общем случае, запускаемые в виде отдельных процессов (но есть возможность объединить несколько компонентов в одно приложение) и взаимодействующие между собой через механизмы IPC.
- **Интерфейсы** — набор сигнатур функций для возможности указания того, как один компонент может обратиться к другому; на данный момент поддерживаются следующие типы интерфейсов: функция (RPC), общая память (Shared Memory), сигналы (Notifications).
- **Коннекторы**, которые связывают взаимодействующие компоненты друг с другом, например, импортируемый интерфейс одного компонента с экспортируемым интерфейсом другого.

В рамках процесса приложения создаются отдельные потоки для основного сценария исполнения и для каждого обработчика взаимодействия с другими компонентами. Для синхронизации потоков поддерживаются стандартные примитивы: семафоры и мьютексы.

5. Заключение и дальнейшие задачи

В данной работе предложена 5-уровневая архитектура CPP InnoChain, допускающая применение методов формальной верификации на каждом из уровней, с целью достижения более высокого уровня надежности CPP, чем было бы возможно при использовании «традиционных» методов обеспечения качества ПО, ранее применявшихся к CPP.

В настоящее время CPP InnoChain находится в стадии реализации. В частности, разработан набор инструментов для автоматического развертывания приложений для seL4 на реальном оборудовании [25].

Для завершения реализации необходимо будет решить следующие дополнительные задачи.

Несмотря на то что формальная верификация seL4 гарантирует свойство функциональной корректности, для микроядра не реализована библиотека для интеграции с базами данных. Одним из возможных путей решения этой проблемы предлагается использовать seL4 с включенным режимом гипервизора для взаимодействия с приложением базы данных, запущенном на виртуальной машине под управлением ОС Linux. Для реализации сетевых взаимодействий между узлами CPP предлагается использовать адаптированную под seL4 библиотеку `picotcp`[21].

Более существенной проблемой является требование по исполнению машинного кода смарт-контракта при обработке транзакций. Ввиду ограничений CAmkES ADL, связанных с формальной

верификацией контроля доступа приложений, на базе seL4 разрешается развертывать исключительно приложения со статической конфигурацией, что означает невозможность порождения дополнительных процессов после запуска приложения. В частности, это означает невозможность динамического развертывания новых смарт-контрактов в системе на основе seL4. В качестве начального решения в данном направлении предлагается линковать смарт-контракты на CakeML с основным приложением узла на этапе развертывания. В качестве альтернативного решения предполагается отдельно исследовать фреймворк Genode, который позволяет динамически запускать новые приложения для seL4, ценой отказа от некоторых гарантий формальной верификации микроядра.

References

[1] A. Burns and A. Wellings, *Concurrent and Real-Time Programming in Ada 2005*. Cambridge University Press, 2007.

[2] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages*. Addison–Wesley, 2009.

[3] *Smart Contracts Under Control*, 2020. [Online]. Available: https://archetype-lang.org.

[4] *More Ethereum Attacks: Race-To-Empty is the Real Deal*, 2016. [Online]. Available: https://vessenes.com/more-ethereum-attacks-race-to-empty-is-the-real-deal.

[5] *CakeML: A verified implementation of ML*, 2020. [Online]. Available: https://cakeml.org.

[6] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, «HotStuff: BFT Consensus with Linearity and Responsiveness», in *PODC*, ACM, 2019, pp. 347–356.

[7] G. Heiser, *The seL4 Microkernel: An Introduction*, 2020. [Online]. Available: https://sel4.systems/About/seL4-whitepaper.pdf.

[8] *Libra by Facebook*. [Online]. Available: https://github.com/libra/libra.

[9] S. Owens, M. O. Myreen, R. Kumar, and Y. K. Tan, «Functional Big-step Semantics», in *Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016*, P. Thiemann, Ed., ser. Lecture Notes in Computer Science, vol. 9632, Springer, Apr. 2016, pp. 589–615. DOI: 10.1007/978-3-662-49498-1"_23.

[10] T. Gauthier, C. Kaliszyk, and J. Urban, «TacticToe: Learning to Reason with HOL4 Tactics», in *LPAR*, ser. EPiC Series in Computing, vol. 46, EasyChair, 2017, pp. 125–143.

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

[12] J. A. Pohjola, H. Rostedt, and M. O. Myreen, «Characteristic Formulae for Liveness Properties of Non-terminating CakeML Programs», in *Interactive Theorem Proving (ITP)*, To appear, LIPICS, 2019. [Online]. Available: https://cakeml.org/itp19.pdf.

[13] M. O. Myreen and S. Owens, «Proof-producing Translation of Higher-order logic into Pure and Stateful ML», *Journal of Functional Programming*, vol. 24, no. 2-3, pp. 284–315, 2014. DOI: 10.1017/S0956796813000282.

[14] O. Abrahamsson, S. Ho, H. Kanabar, R. Kumar, M. O. Myreen, M. Norrish, and Y. K. Tan, «Proof-Producing Synthesis of CakeML from Monadic HOL Functions», Springer, 2020. [Online]. Available: https://rdcu.be/b4FrU.

[15] P. O'Hearn, «Separation Logic», *Communications of the ACM*, vol. 62, no. 2, pp. 86–95, 2019. DOI: 10.1145/3211968.

[16] L. C. Paulson, *Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow)*, ser. Lecture Notes in Computer Science. Springer, 1994, vol. 828.

[17] L. Hupel and T. Nipkow, «A Verified Compiler from Isabelle/HOL to CakeML», in *European Symposium on Programming (ESOP)*, ser. Lecture Notes in Computer Science, vol. 10801, Springer, 2018, pp. 999–1026. DOI: 10.1007/978-3-319-89884-1_35. [Online]. Available: https://lars.hupel.info/pub/isabelle-cakeml.pdf.

[18] A. W. Appel, «Verified Software Toolchain», in *European Symposium on Programming*, Springer, 2011, pp. 1–17.

[19] *CompCert project*. [Online]. Available: https://compcert.org.

[20] *Verification center of the operating system Linux*. [Online]. Available: http://linuxtesting.org/publications.

[21] *Implementation of the network layer for seL4*. [Online]. Available: https://github.com/SEL4PROJ/picotcp.

[22] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, «Handbook of Satisfiability», in, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, vol. 185, ch. Satisfiability Modulo Theories, pp. 825–885.

[23] *seL4 deployment using Genode*. [Online]. Available: https://genode.org/documentation/articles/sel4_parts_1.

[24] *Introduction to CAmkES*. [Online]. Available: https://docs.sel4.systems/Tutorials/hello-camkes-0.html.

[25] R. Rezin, *seL4 deploy scripts*, 2020. [Online]. Available: https://github.com/RZRussel/seL4-deploy-public.git.

MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 27, NO. 4, 2020
journal homepage: www.mais-journal.ru

THEORY OF COMPUTING

The "One-fifth Rule" with Rollbacks for Self-Adjustment of the Population Size in the $(1 + (\lambda, \lambda))$ Genetic Algorithm

A. O. Bassin[1], M. V. Buzdalov[1], A. A. Shalyto[1]

DOI: 10.18255/1818-1015-2020-4-488-508

[1]ITMO University, 49 Kronverkskiy ave., Saint Petersburg 197101, Russia.

MSC2020: 60G40, 90C56
Research article
Full text in Russian

Received October 22, 2020
After revision November 18, 2020
Accepted December 16, 2020

Self-adjustment of parameters can significantly improve the performance of evolutionary algorithms. A notable example is the $(1 + (\lambda, \lambda))$ genetic algorithm, where adaptation of the population size helps to achieve the linear running time on the OneMax problem. However, on problems which interfere with the assumptions behind the self-adjustment procedure, its usage can lead to the performance degradation. In particular, this is the case with the "one-fifth rule" on problems with weak fitness-distance correlation.

We propose a modification of the "one-fifth rule" in order to have less negative impact on the performance in the cases where the original rule is destructive. Our modification, while still yielding a provable linear runtime on OneMax, shows better results on linear function with random weights, as well as on random satisfiable MAX-3SAT problems.

Keywords: parameter adaptation; $(1 + (\lambda, \lambda))$ GA; linear functions; MAX-3SAT

INFORMATION ABOUT THE AUTHORS

Anton Olegovich Bassin correspondence author	orcid.org/0000-0002-6697-6714. E-mail: anton.bassin@gmail.com PhD student.
Maxim Viktorovich Buzdalov correspondence author	orcid.org/0000-0002-7120-8824. E-mail: mbuzdalov@gmail.com Researcher, PhD.
Anatoly Abramovich Shalyto	orcid.org/0000-0002-2723-2077. E-mail: shalyto@mail.ifmo.ru Chief researcher, Doctor.

Funding: This research was supported by the Russian Scientific Foundation, agreement No.17-71-20178.

For citation: A. O. Bassin, M. V. Buzdalov, and A. A. Shalyto, "The "One-fifth Rule" with Rollbacks for Self-Adjustment of the Population Size in the $(1 + (\lambda, \lambda))$ Genetic Algorithm", *Modeling and analysis of information systems*, vol. 27, no. 4, pp. 488-508, 2020.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 27, № 4, 2020
сайт журнала: www.mais-journal.ru

THEORY OF COMPUTING

Правило «одной пятой» с возвратами для настройки размера популяции в генетическом алгоритме $(1 + (\lambda, \lambda))$

А. О. Басин[1], М. В. Буздалов[1], А. А. Шалыто[1]

DOI: 10.18255/1818-1015-2020-4-488-508

[1]Университет ИТМО, Кронверкский пр., д. 49, Санкт-Петербург, 197101 Россия.

УДК 004.023:004.85
Научная статья
Полный текст на русском языке

Получена 22 октября 2020 г.
После доработки 18 ноября 2020 г.
Принята к публикации 16 декабря 2020 г.

Известно, что настройка параметров может существенно улучшить время работы эволюционных алгоритмов. Ярким примером этого является генетический алгоритм $(1 + (\lambda, \lambda))$, где адаптация размера популяции в процессе работы помогает достичь линейного времени работы на задаче OneMax. Однако если свойства решаемой задачи вступают в конфликт с принципами работы используемого метода настройки параметров, производительность эволюционного алгоритма может существенно ухудшаться. Так, например, происходит при использовании правила «одной пятой» в упомянутом алгоритме при решении задач со слабой корреляцией между приспособленностью и расстоянием до оптимума.

В данной работе предлагается модификация правила «одной пятой», существенно снижающая отрицательные эффекты от его использования при их наличии. Показывается, что данная модификация также достигает линейного времени работы на задаче OneMax, при этом ее использование приводит к улучшению производительности на линейных псевдобулевых функциях со случайными весами, а также на некотором классе задач MAX-3SAT.

Ключевые слова: настройка параметров; $(1 + (\lambda, \lambda))$-ГА; линейные функции; MAX-3SAT

ИНФОРМАЦИЯ ОБ АВТОРАХ

Антон Олегович Басин
автор для корреспонденции
orcid.org/0000-0002-6697-6714. E-mail: anton.bassin@gmail.com
Аспирант.

Максим Викторович Буздалов
автор для корреспонденции
orcid.org/0000-0002-7120-8824. E-mail: mbuzdalov@gmail.com
Научный сотрудник, кандидат технических наук.

Анатолий Абрамович Шалыто
orcid.org/0000-0002-2723-2077. E-mail: shalyto@mail.ifmo.ru
Главный научный сотрудник, доктор технических наук.

Финансирование: Исследование поддержано Российским научным фондом, соглашение №17-71-20178.

Для цитирования: A. O. Bassin, M. V. Buzdalov, and A. A. Shalyto, “The “One-fifth Rule” with Rollbacks for Self-Adjustment of the Population Size in the $(1 + (\lambda, \lambda))$ Genetic Algorithm”, *Modeling and analysis of information systems*, vol. 27, no. 4, pp. 488-508, 2020.

Введение

Эволюционные алгоритмы — это обширный подкласс стохастических методов оптимизации, как правило, взаимодействующих с оптимизируемой функцией как с «черным ящиком». Под эволюционными алгоритмами в узком смысле чаще всего подразумевают генетические алгоритмы [1], эволюционные стратегии [2, 3], эволюционное программирование [4] и генетическое программирование [5]. В широком смысле данный термин также включает в себя методы роевого интеллекта [6—8], методы на основе физических моделей [9, 10], а также методы, разработанные без привлечения каких-либо аналогий с природными процессами или явлениями [11—13]. Данную совокупность методов также называют метаэвристиками [14] или рандомизированными оптимизационными эвристиками. Эволюционные алгоритмы являются разновидностью так называемых мягких вычислений, а в более широком смысле — частью предметной области, все чаще называемой «вычислительный интеллект» (англ. computational intelligence).

Эволюционные алгоритмы применяются для решения множества практических задач, таких как составление расписаний в промышленности [15, 16], оптимизация разводки печатных плат [17—20], построение систем управления [21—27], сегментирование изображений [28], а также приближенного решения других задач комбинаторной [29—32] и вещественной оптимизации [33—35].

Как правило, в работах, посвященных эволюционным алгоритмам, используется набор терминов, заимствованных из теории эволюции. Так, пробные решения называются *особями*, а их составные части — *генами*. Операторы, вносящие небольшие изменения в особи, называются *операторами мутации*, а операторы, создающие новые особи на основе двух и более исходных (*родительских*) особей, называются *операторами скрещивания*, *рекомбинации* или *кроссинговера*. Функция оценки решения, которую требуется оптимизировать, называется *функцией приспособленности*. Многие эволюционные алгоритмы оперируют наборами решений ограниченного размера, которые в этом случае называются *популяциями*, а различные функции, выбирающие особи из популяции или строящие новую популяцию, называются *операторами отбора*.

В настоящей работе особое внимание уделяется настройке параметров в эволюционных алгоритмах. Необходимо отметить, что во многих ранних работах по эволюционным алгоритмам считалось, что их эффективность не зависит значительным образом от конкретных значений параметров (таких как, например, вероятность мутации каждого гена). Данная парадигма мышления объясняется тем, что даже при очевидно неправильных значениях параметров результаты работы таких алгоритмов могут быть вполне удовлетворительными [36, стр. 22]. Следующим этапом были попытки поиска универсальных оптимальных значений параметров [37], далее была осознана необходимость ставить значения параметров в зависимость от размера задачи [38]. В настоящее время общепризнано, что настройка параметров играет одну из ключевых ролей в повышении производительности эволюционных алгоритмов [39]. Исследованиями в этом направлении занимается множество исследовательских групп по всему миру [40—52].

Эволюционные алгоритмы являются прежде всего алгоритмами, предназначенными для применения на практике. Однако для объяснения их принципов работы было развито несколько ветвей теоретических исследований, из которых стоит особо отметить математический анализ времени работы эволюционных алгоритмов, вплотную связанный с именем И. Вегенера [53]. С обширным перечнем известных результатов и их обсуждением можно ознакомиться в обзорах [54, 55]. Среди российских ученых стоит отметить В. Редько [56, 57], Е. Семенкина [58, 59], С. Родзина [60, 61], А. Еремеева [62—65].

Польза анализа времени работы эволюционных алгоритмов не ограничивается констатацией фактов об оценках времени нахождения оптимума той или иной задачи тем или иным алгоритмом. С помощью сравнения фактических оценок времени работы и оценок вычислительной сложности задач оптимизации для алгоритмов различных классов можно ставить вопросы о причинах их

недостаточной эффективности, формулировать гипотезы о наиболее перспективных направлениях исследований и, наконец, разрабатывать новые, более эффективные эволюционные алгоритмы.

Ярким примером эволюционного алгоритма, разработанного исходя из теоретических соображений, является генетический алгоритм $(1 + (\lambda, \lambda))$, предложенный в работе [66] (также см. конференционную версию [67]). Данный алгоритм обладает рядом уникальных свойств. Во-первых, на модельной задаче OneMax, наиболее часто рассматриваемой в теории эволюционных вычислений, данному алгоритму достаточно линейного, относительно размера задачи n, числа запросов к функции приспособленности для нахождения оптимума, в то время как все другие ранее известные эволюционные алгоритмы требуют как минимум $\Omega(n \log n)$ запросов. Во-вторых, известно, что для достижения этого успеха в данном алгоритме требуется выполнять настройку параметров в процессе работы — оптимальный статический выбор параметров дает следующую оценку необходимого числа запросов [68]:

$$\Theta\left(n\sqrt{\frac{\log n \log\log\log n}{\log\log n}}\right),$$

в то время как линейное число запросов достигается при использовании правила «одной пятой» [69] или же стохастического выбора параметра из распределения со степенным законом [70].

Помимо задачи OneMax, данный алгоритм рассматривался также и на других задачах. В работе [66] он, в частности, исследовался экспериментальным способом на линейных псевдобулевых функциях со случайными весами из диапазона $[1; 2]$, а также на функциях из семейства Royal Road. В работе [71] было показано, что данный алгоритм хорошо справляется с некоторым подклассом задач MAX-3SAT (состоящих в нахождении подстановки булевых переменных, максимизирующей число дизъюнктов в заданной КНФ-формуле), а в последовавшей за этим работе [72] это утверждение было доказано (хотя и для несколько иного подкласса этих задач). Однако применение данного алгоритма на практике все еще нельзя назвать широким: среди инженерных применений можно отметить лишь работу [73], посвященную генерации конструкции башен, также можно отметить работу [74] из области поисковой инженерии программного обеспечения. Одним из факторов можно назвать то, что правило «одной пятой», используемое для настройки параметров с целью достижения максимальной производительности, способно существенно ухудшать поведение алгоритма на задачах с малой корреляцией между приспособленностью и расстоянием Хэмминга до оптимума. Отчасти это было показано уже в работе [72], в которой для надлежащей эффективности потребовалось дополнительно ограничить сверху значение параметра. Польза подобного ограничения была также показана в работе [75] для другого класса задач.

В данной работе, являющейся расширенной версией работы [76], мы проводим углубленное изучение этой проблемы. После введения определений, обозначений, а также описания рассматриваемых в работе алгоритмов и модельных задач (раздел 1), в разделе 2 выполняется анализ зависимости эффективности генетического алгоритма $(1 + (\lambda, \lambda))$ от значения параметра λ при различных значениях расстояния Хэмминга до оптимума, из которого делаются выводы о возможных причинах неэффективного поведения данного алгоритма и их связи с настройкой параметров. В разделе 3 излагается модификация правила «одной пятой», используемого в рассматриваемом алгоритме для настройки параметров. Экспериментальные исследования, результаты которых изложены в разделе 4, показывают, что предложенная модификация действительно приводит к повышению эффективности генетического алгоритма $(1 + (\lambda, \lambda))$ на задачах, отличных от OneMax. В то же время в разделе 5 показано, что на задаче OneMax предложенная модификация сохраняет асимптотическую эффективность, и среднее число запросов к функции приспособленности, необходимое для нахождения ее оптимума, остается линейным. В разделе 6 дополнительно обсуждается то, насколько эффективность предложенной модификации близка к теоретически достижимым значениям, и потенциал дальнейшего улучшения данного алгоритма. Итоги работы подводятся в заключении.

1. Определения и обозначения

Будем обозначать множество целых чисел от 1 до n как $[1..n]$. Логарифм по неопределенному основанию, который может встречаться в асимптотических обозначениях, обозначается как log, в то время как натуральный логарифм обозначается как ln. Значение x, округленное вниз, записывается как $\lfloor x \rfloor$. Если x — логическое значение, то $[x]$ — единица, если x истинно, иначе ноль. В псевдокодах алгоритмов операция UniformRandom обозначает случайную выборку элемента из множества, причем каждый элемент выбирается с равной вероятностью.

1.1. Исследуемые модельные задачи

Ранее упомянутая во введении задача OneMax является классической задачей, изучаемой во многих теоретических и экспериментальных работах из области эволюционных вычислений. Экземплярами данной задачи являются функции $\mathrm{OneMax}_{n,z}$, $z \in \{0,1\}^n$, определенные на битовых строках размера n следующим образом:

$$\mathrm{OneMax}_{n,z} : \{0,1\}^n \longrightarrow \mathbb{R}; x \mapsto |\{i \in [1..n] \mid x_i = z_i\}|.$$

Таким образом, функция $\mathrm{OneMax}_{n,z}$ возвращает число битовых позиций, на которых битовые строки x и z совпадают. Множество всех функций $\mathrm{OneMax}_{n,z}$ для фиксированного значения n составляет задачу OneMax размера n.

Функции, составляющие OneMax, могут быть интерпретированы как полиномиальные псевдобулевы функции с максимальной степенью монома, равной единице, или *линейные* псевдобулевы функции. Линейные псевдобулевы функции могут быть представлены в следующем общем виде:

$$\mathrm{Linear}_{n,z,w} : \{0,1\}^n \longrightarrow \mathbb{R}; x \mapsto \sum_{i \in [1..n]} w_i \cdot [x_i = z_i],$$

где w — вектор положительных весов, ассоциированных с битовыми позициями. В данной работе рассмотрены подклассы линейных псевдобулевых функций следующего вида: элементы вектора w являются целочисленными и выбираются равновероятно из множества $[1..W]$ для некоторого значения W. Будем обозначать задачу, состоящую из всех таких функций, как LinInt_W. Отметим, что $\mathrm{OneMax} = \mathrm{LinInt}_1$.

Также в работе будет рассмотрен специальный класс задач выполнимости булевой формулы [77], являющийся подмножеством класса MAX-3SAT. Класс MAX-3SAT состоит из псевдобулевых задач оптимизации, в которых, путем подбора подстановки переменных, требуется максимизировать число выполненных дизъюнктов булевой формулы, записанной в конъюнктивной нормальной форме с тремя литералами в каждом дизъюнкте (т.е. 3-КНФ). Пусть число переменных равно n, а число дизъюнктов равно m. В общем виде данные задачи являются NP-трудными, однако существуют относительно простые классы таких задач [78]. Одним из таких классов являются случайно сгенерированные выполнимые формулы в так называемой модели *planted solution*. Суть данной модели состоит в следующем: вначале генерируется некоторая подстановка x^*, которая будет заведомо являться решением. Далее, каждый из m дизъюнктов генерируется путем равновероятной и независимой выборки каждого из литералов из множества всех литералов с учетом того, что дизъюнкт должен удовлетворяться подстановкой x^*. *Плотностью дизъюнктов* называется значение m/n, равное среднему числу дизъюнктов, приходящееся на одну переменную. Выбор данной модели обусловлен тем, что она ранее изучалась в теории эволюционных вычислений [72, 79, 80], и, в частности, ее свойства достаточно сильно похожи на свойства задачи OneMax в том случае, когда формула является *плотной*, то есть $m \geq n^2$. При рассматриваемой в данной работе комбинации параметров, $m = 4n \ln n$, которая также изучалась в указанных работах, данное сходство также выражено, но не столь сильно, поэтому задача является более сложной, чем OneMax.

1.2. Эволюционные методы локального спуска

В силу относительной простоты связанного с ними анализа, в теоретических исследованиях эволюционных алгоритмов нередко фигурируют алгоритм рандомизированного локального поиска (англ. randomized local search, далее RLS, рисунок 1) и так называемый эволюционный алгоритм (1+1) (далее (1 + 1)-ЭА, рисунок 2). В обоих алгоритмах между итерациями переходит только одна особь, на каждой итерации с помощью оператора мутации порождается одна дочерняя особь, и если ее приспособленность не хуже, чем у родительской, она заменяет родительскую. В алгоритме RLS оператор мутации локальный, в нем инвертируется один случайно выбранный бит, в (1 + 1)-ЭА оператор мутации глобальный — каждый бит инвертируется независимо с вероятностью $1/n$.

Известно, что оба данных алгоритма оптимизируют линейные псевдобулевы функции, включая OneMax, в среднем за $\Theta(n \log n)$ запросов к функции приспособленности. Для рассматриваемого в данной работе класса задач MAX-3SAT для (1 + 1)-ЭА также была доказана оценка $\Theta(n \log n)$ для числа запросов в среднем [80]. Отметим, что для RLS эта оценка также выполняется с поправкой на то, что каждый запуск завершается успешно лишь с вероятностью $1 - e^{-\Omega(n)}$, то есть вероятность неуспеха экспоненциально убывает с ростом числа переменных n.

Require: n: the problem size, $f : \{0,1\}^n \to \mathbb{R}$: the function to maximize
1: $x \leftarrow$ UniformRandom($\{0,1\}^n$)
2: **for** $t \leftarrow 1, 2, 3, \ldots$ **do**
3: $\quad i \leftarrow$ UniformRandom($[1..n]$)
4: $\quad y \leftarrow x$ with i-th bit flipped
5: $\quad$ **if** $f(y) \geq f(x)$ **then**
6: $\quad\quad x \leftarrow y$
7: $\quad$ **end if**
8: **end for**

Fig. 1. Randomized local search

Рис. 1. Рандомизированный случайный поиск

Require: n: the problem size, $f : \{0,1\}^n \to \mathbb{R}$: the function to maximize
1: $x \leftarrow$ UniformRandom($\{0,1\}^n$)
2: **for** $t \leftarrow 1, 2, 3, \ldots$ **do**
3: $\quad y \leftarrow x$
4: $\quad$ **for** $i \in [1..n]$ **do**
5: $\quad\quad$ **with the probability of** $1/n$ **do**
6: $\quad\quad\quad$ flip i-th bit in y
7: $\quad\quad$ **done**
8: $\quad$ **end for**
9: $\quad$ **if** $f(y) \geq f(x)$ **then**
10: $\quad\quad x \leftarrow y$
11: $\quad$ **end if**
12: **end for**

Fig. 2. The (1 + 1) evolutionary algorithm

Рис. 2. Эволюционный алгоритм (1 + 1)

1.3. Генетический алгоритм $(1 + (\lambda, \lambda))$

Генетический алгоритм $(1+(\lambda, \lambda))$ (для краткости будем обозначать его как $(1+(\lambda, \lambda))$-ГА) был впервые предложен в работе [67], журнальная версия которой опубликована двумя годами позже [66].

Его основное отличие от существующих эволюционных алгоритмов, предназначенных для оптимизации псевдобулевых функций, можно кратко сформулировать следующим образом. Во-первых, в операторе мутации используется более высокая, чем обычно, вероятность инвертирования битов, что увеличивает скорость исследования пространства поиска. Во-вторых, в алгоритме используется оператор скрещивания, настроенный таким образом, чтобы нивелировать отрицательные эффекты от такого оператора мутации. Более подробно этот алгоритм будет описан далее.

В канонической версии данного алгоритма имеется, по сути, один параметр λ, а все остальные параметры выводятся из него и из размера задачи n. В работах, в которых данный алгоритм был впервые предложен [66, 67], основной объем теоретического анализа был проделан для фиксированного значения параметра λ. В частности, было доказано, что при некотором выборе параметра λ в зависимости от n матожидание времени работы алгоритма, требуемое для решения задачи OneMax, составляет $O(n\sqrt{\log n})$, что асимптотически меньше, чем у любого другого известного на тот момент эволюционного алгоритма. Данная оценка была несколько улучшена в последующих работах [68, 81], в них показана не только верхняя оценка, но и асимптотически равная ей нижняя.

В то же время, в работах [66, 67] было показано и то, что если выбирать значение λ в зависимости не только от размера задачи n, но и от значения приспособленности лучшей особи (то есть, *динамически*, с изменением данного параметра в процессе работы), то матожидание времени работы такого алгоритма на задаче OneMax составляет $O(n)$. Таким образом, показано, что динамический выбор параметра лучше, чем наилучший возможный статический выбор. Также с помощью экспериментов было показано, что известное из области вещественнозначных эволюционных алгоритмов так называемое правило «одной пятой», примененное для управления параметром λ, способно поддерживать значение этого параметра в непосредственной близости от оптимального значения, не используя знание об оптимальной для данной задачи зависимости. Данная особенность, а вместе с ней и оценка $O(n)$ на матожидание времени работы на OneMax, была впоследствии доказана теоретически [68, 69]. Линейная оценка для матожидания времени работы была также доказана для описанного выше класса задач MAX-3SAT [72], однако для того, чтобы она выполнялась, параметр λ необходимо дополнительно ограничивать некоторым достаточно медленно растущим значением (в работе [72] использовалось значение $\overline{\lambda} = 2\ln(n+1)$).

В данной работе исследуется адаптивная версия $(1 + (\lambda, \lambda))$-ГА, использующая ограничение на значение λ (рисунок 3). Опишем ее подробнее. В данном алгоритме используются следующие операторы, заданные на битовых строках:

- оператор мутации ℓ бит: оператор $\mathrm{Mutate}(x, \ell)$ создает на основе битовой строки $x \in \{0,1\}^n$ новую битовую строку y путем инвертирования ровно ℓ бит, таким образом, что любое множество бит размером ℓ выбирается равновероятно;
- смещенный оператор скрещивания: оператор $\mathrm{Crossover}(x, x', c)$, параметризованный *смещением* $c \in [0,1]$ создает новую битовую строку y на основе двух данных битовых строк x и x' путем независимого выбора для каждой позиции $i \in [1..n]$ значения из второго аргумента ($y_i = x'_i$) с вероятностью c и значения из первого аргумента ($y_i = x_i$) в противном случае.

Алгоритм начинает работу со случайной инициализации начальной популяции, состоящей из одного элемента x. На каждой итерации выполняются следующие действия.

- На *фазе мутации*, на основе родительской особи x генерируется λ потомков путем применения оператора мутации, для которого параметр ℓ выбирается одинаковым для всех потомков и выбирается из биномиального распределения $\mathcal{B}(n, \lambda/n)$. В данной работе, следуя недавно предложенным практико-ориентированным рекомендациям [82], которые уже привели к ряду нетривиальных результатов [83], значение ℓ выбирается из условного распределения, которое может быть описано как $\ell \sim \mathcal{B}(n, \lambda/n) \mid \ell > 0$, то есть всякий раз, когда ℓ выбирается равным нулю, выборка повторяется.

- Фаза мутации оканчивается *промежуточным отбором*, на котором из λ потомков остается потомок, имеющий максимальное значение приспособленности. Данный потомок обозначается как x'. Если потомков с максимальной приспособленностью несколько, один из них выбирается равновероятно.
- На *фазе скрещивания*, на основе родительской особи x и лучшего из потомков из предыдущей фазы x' генерируется λ новых потомков путем применения оператора скрещивания. Смещение в операторе скрещивания выбирается равным $c = 1/\lambda$. Данный выбор был обоснован (по крайней мере, для решения задач, подобных OneMax) в работах [66, 84]. Используемая в данной работе практико-ориентированная модификация состоит в том, что при попытке выбора нуля бит из особи x' получившаяся особь, идентичная родительской, игнорируется и строится заново, а если получающаяся особь идентична особи x', то ее приспособленность повторно не вычисляется.
- После этого осуществляется *элитарный отбор* в следующее поколение: лучший из потомков, сгенерированных на фазе скрещивания, замещает собой родительскую особь x, если его приспособленность не хуже, чем у x.

Require: n: the problem size, $f : \{0,1\}^n \to \mathbb{R}$: the function to maximize

$F \leftarrow$ a constant $\in (1; 2)$ ▷ The adaptation speed
$U \leftarrow 5$ ▷ The value of 5 from the 1/5-th rule
$\lambda \leftarrow 1$ ▷ The initial value of λ
$x \leftarrow \text{UniformRandom}(\{0,1\}^n)$
for $t \leftarrow 1, 2, 3, \ldots$ **do**
  $p \leftarrow \lambda/n$, $c \leftarrow 1/\lambda$, $\lambda' \leftarrow \lfloor \lambda \rfloor$, $\ell \sim \mathcal{B}(n, p)$
  for $i \in [1..\lambda']$ **do** ▷ Phase 1: Mutation
    $x^{(i)} \leftarrow \text{Mutate}(x, \ell)$
  end for
  $x' \leftarrow \text{UniformRandom}(\{x^{(j)} \mid f(x^{(j)}) = \max\{f(x^{(i)})\}\})$
  for $i \in [1..\lambda']$ **do** ▷ Phase 2: Crossover
    $y^{(i)} \leftarrow \text{Crossover}(x, x', c)$
  end for
  $y \leftarrow \text{UniformRandom}(\{y^{(j)} \mid f(y^{(j)}) = \max\{f(y^{(i)})\}\})$
  if $f(y) > f(x)$ **then** ▷ Selection and adaptation
    $x \leftarrow y$, $\lambda \leftarrow \max\{\lambda/F, 1\}$
  else if $f(y) = f(x)$ **then**
    $x \leftarrow y$, $\lambda \leftarrow \min\{\lambda F^{1/(U-1)}, \overline{\lambda}\}$
  else
    $\lambda \leftarrow \min\{\lambda F^{1/(U-1)}, \overline{\lambda}\}$
  end if
end for

Fig. 3. The $(1 + (\lambda, \lambda))$ genetic algorithm with the adaptive choice of $\lambda \le \overline{\lambda}$

Рис. 3. Генетический алгоритм $(1 + (\lambda, \lambda))$ с адаптивным выбором параметра $\lambda \le \overline{\lambda}$

Правило «одной пятой», используемое для адаптивной настройки параметра λ, устроено следующим образом. Если лучшая приспособленность на текущей итерации увеличилась, то значение λ уменьшается в F раз. Если же улучшения не было, то значение λ увеличивается в $F^{1/(U-1)}$ раз, где U — константа (как правило, $U = 5$). В результате, если средняя вероятность улучшения составляет $1/U$, то значение λ поддерживается на постоянном уровне. В дополнение к этому, значение λ ограничено снизу единицей, а сверху — значением $\overline{\lambda}$, которое, как правило, зависит от n и не превосходит n.

2. Исследование ландшафта параметров

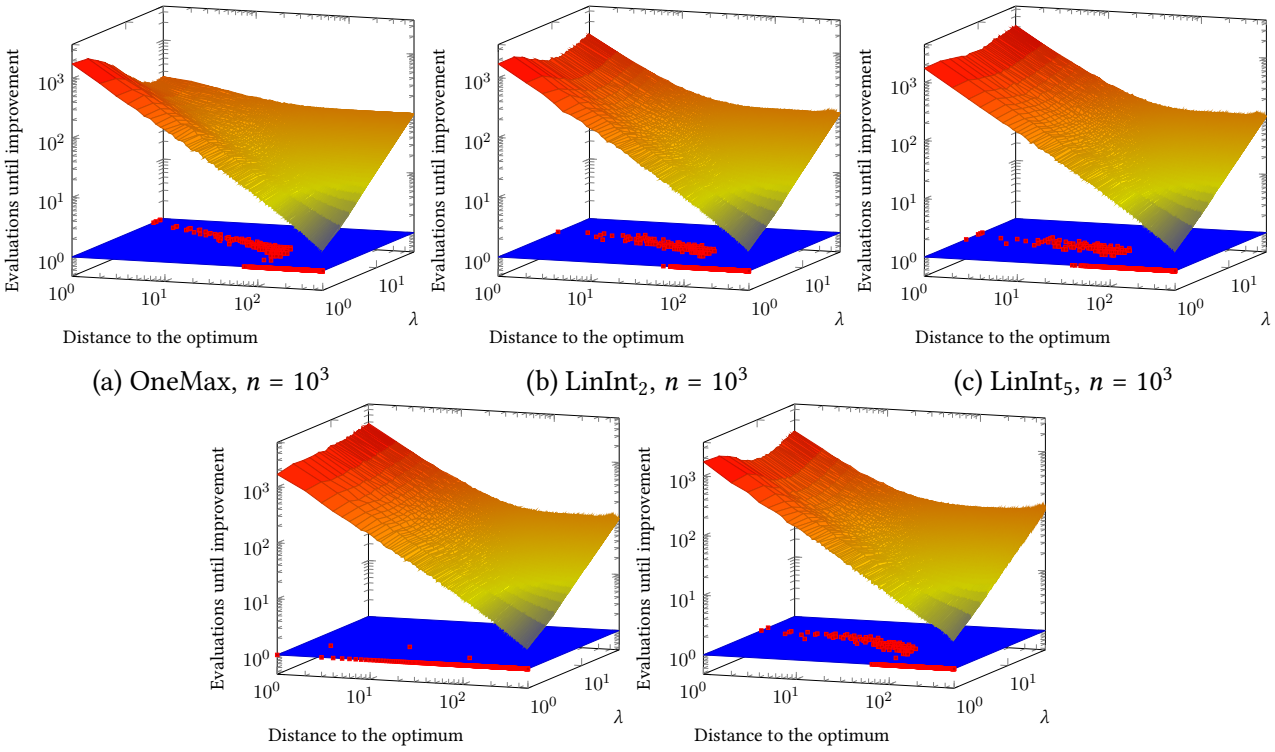
С целью анализа того, какую эффективность имеют различные значения параметра λ в зависимости от расстояния Хэмминга до оптимума, были проведены предварительные экспериментальные исследования, результаты которых представлены на рисунке 4. Данные исследования были проведены для всех рассматриваемых в данной работе задач: OneMax, LinInt_2, LinInt_5, LinInt_n и MAX-3SAT. Для всех задач использовался единый размер задачи $n = 1000$. Для каждой задачи и для всех целочисленных значений $\lambda_0 \in [1..50]$ было выполнено 10^3 запусков генетического алгоритма $(1 + (\lambda, \lambda))$ с фиксированным значением $\lambda = \lambda_0$. В процессе данных запусков для всех расстояний Хэмминга до оптимума $1 \le d \le 500$ и всех исследованных значений λ_0 было вычислено, сколько вычислений функции приспособленности $E(d, \lambda_0)$ было выполнено, когда расстояние между родительской особью и оптимумом было равно d при $\lambda = \lambda_0$, а также число $I(d, \lambda_0)$, описывающее, сколько раз алгоритм покидал данное состояние, переходя в состояние с меньшим расстоянием Хэмминга до оптимума.

На рисунке 4 верхние поверхности сложной формы отражают полученные в результате описанных экспериментов аппроксимации *матожидания числа вычислений функции приспособленности, необходимых для улучшения*, вычисленные как $E(d, \lambda_0)/I(d, \lambda_0)$. Нижние поверхности синего цвета отображают лучшие (т.е. минимальные) среди полученных таким образом значений: для каждого значения d сначала находится такое λ^*, что $E(d, \lambda^*)/I(d, \lambda^*)$ минимально, затем определяются такие достаточно хорошие λ_+, что $E(d, \lambda^+)/I(d, \lambda^+) \le 1.02 \cdot E(d, \lambda^*)/I(d, \lambda^*)$, то есть эффективность которых находится в пределах 2% от оптимального выбора. Такие значения λ+ на указанных рисунках отображаются в виде красных кубиков на синей поверхности.

Внешний вид рисунка 4a, иллюстрирующего результаты на задаче OneMax, соответствует известным фактам относительно поведения $(1 + (\lambda, \lambda))$-ГА на заданной задаче. За исключением некоторых отклонений при больших расстояниях Хэмминга до оптимума, вызванных дискретными значениями параметра λ и использованием практико-ориентированных модификаций, значения λ, близкие к оптимальным, находятся в окрестности прямой в логарифмических осях, проведенной от точки $d = 500, \lambda = 1$ до точки $d = 1, \lambda \approx 30 \ldots 40$. Данные наблюдения соответствуют известному факту, что значения $\lambda \approx \sqrt{n/(n - f(x))}$ оптимальны для задачи OneMax.

Результаты для задач LinInt_2 (рисунок 4b) и LinInt_5 (рисунок 4c) уже показывают несколько иную картину. Пока расстояние Хэмминга до оптимума превышает примерно 50, оптимальные значения ведут себя близко к тому, как это было на OneMax. Однако это поведение изменяется по мере приближения к оптимуму. Визуально общая тенденция поведения все еще напоминает выстраивание вдоль некоторой прямой в логарифмических осях, но наклон этой прямой уже существенно меньше: на единичном расстоянии до оптимума лучшее значение λ для задачи LinInt_2 лишь немного превышает 10, в то время как для задачи LinInt_5 лучшее значение уже меньше десяти. Соответствующие оценки матожиданий числа вычислений функции приспособленности, необходимые для улучшения (то есть для достижения оптимума), примерно равны 10^3 в случае LinInt_2 и достигают $2 \cdot 10^3$ в случае LinInt_5, в то время как для OneMax эта оценка была около 200. Данные наблюдения подсказывают, что уже для линейных функций улучшение наступает недостаточно быстро для того, чтобы правило «одной пятой» поддерживало λ в окрестности оптимальных значений.

В более абстрактных терминах, у задач LinInt_2 и LinInt_5 наблюдается недостаточно хорошая корреляция между функцией приспособленности и расстоянием Хэмминга до оптимума, в то время как у задачи OneMax она по определению идеальна (т.е. равна минус единице). Что же касается задачи LinInt_n, то у нее эта корреляция уже весьма мало отличается от нуля, и, судя по рисунку 4d, сама структура $(1 + (\lambda, \lambda))$-ГА уже недостаточно хороша, чтобы решать эту задачу эффективно: оптимальные значения λ сконцентрированы около единицы, что практически превращает данный алгоритм в алгоритм локального поиска, подобный (1 + 1)-ЭА.



(a) OneMax, $n = 10^3$ (b) LinInt_2, $n = 10^3$ (c) LinInt_5, $n = 10^3$

(d) LinInt_n, $n = 10^3$ (e) MAX-3SAT, $n = 10^3$

Fig. 4. Parameter landscapes for different problems. Red cubes indicate values of $\lambda(d)$ within 2% of the best

Рис. 4. Ландшафты параметров для различных задач. Красные кубики соответствуют значениям $\lambda(d)$ в пределах 2% от лучших

На задаче MAX-3SAT с логарифмической плотностью дизъюнктов поведение $(1 + (\lambda, \lambda))$-ГА представляется более интересным. Из рисунка 4e можно сделать вывод, что поведение данного алгоритма, исходя из величин λ, находится где-то между поведением на задаче OneMax и на LinInt_2. Кроме того, линия, вокруг которой сконцентрированы оптимальные значения λ, представляется скорее как кривая, выпуклая вверх. Исходя из данных результатов, положительный эффект от ограничения значений λ значением $2\ln(n + 1)$, показанный в работе [72], представляется вполне естественным.

3. Предлагаемая модификация

В данной работе предлагается модификация адаптивной настройки параметров по правилу «одной пятой» в $(1 + (\lambda, \lambda))$-ГА (рисунок 5). Основная идея состоит в том, чтобы препятствовать непрерывному росту λ при длительных сериях неудачных итераций, в то же время позволяя повышать λ до произвольно больших значений, если это действительно необходимо. Если итерация была успешной (приспособленность лучшего из потомков превзошла приспособленность родителя), обновленное значение λ также сохраняется в переменной λ_0. Каждая же непрерывная последовательность неудачных итераций разбивается на *отрезки* линейно увеличивающейся длины: каждый следующий отрезок на единицу длиннее предыдущего. В рамках каждого отрезка значение λ растет экспоненциально, как в исходном алгоритме, однако в начале каждого такого отрезка значение λ сбрасывается в значение λ_0. Начальный размер отрезка равен константному значению 10, чтобы соответствовать исходному $(1 + (\lambda, \lambda))$-ГА в случае, когда адаптация работает эффективно. Описанная стратегия ограничивает скорость, с которой растет максимальное значение λ, в то же время сохраняется возможность выполнить итерацию с относительно небольшим значением λ, что может быть полезно в тех ситуациях, когда малые λ выгоднее.

Require: n: the problem size, $f : \{0,1\}^n \to \mathbb{R}$: the function to maximize
1: $F \leftarrow$ a constant $\in (1;2)$ ▷ The adaptation speed
2: $U \leftarrow 5$ ▷ The value of 5 from the 1/5-th rule
3: $\lambda \leftarrow 1$ ▷ The initial value of λ
4: $B \leftarrow 0$ ▷ The number of consecutive bad iterations
5: $\Delta \leftarrow 10$ ▷ The limit of growth for λ
6: $\lambda_0 \leftarrow 1$ ▷ The base value of λ
7: $x \leftarrow \text{UniformRandom}(\{0,1\}^n)$
8: **for** $t \leftarrow 1, 2, 3, \ldots$ **do**
9: $p \leftarrow \lambda/n$, $c \leftarrow 1/\lambda$, $\lambda' \leftarrow \lfloor \lambda \rfloor$, $\ell \sim \mathcal{B}(n, p)$
10: **for** $i \in [1..\lambda']$ **do** ▷ Phase 1: Mutation
11: $x^{(i)} \leftarrow \text{Mutate}(x, \ell)$
12: **end for**
13: $x' \leftarrow \text{UniformRandom}(\{x^{(j)} \mid f(x^{(j)}) = \max\{f(x^{(i)})\}\})$
14: **for** $i \in [1..\lambda']$ **do** ▷ Phase 2: Crossover
15: $y^{(i)} \leftarrow \text{Crossover}(x, x', c)$
16: **end for**
17: $y \leftarrow \text{UniformRandom}(\{y^{(j)} \mid f(y^{(j)}) = \max\{f(y^{(i)})\}\})$
18: **if** $f(y) > f(x)$ **then** ▷ Selection and adaptation
19: $x \leftarrow y$, $\lambda \leftarrow \max\{\lambda/F, 1\}$, $\lambda_0 \leftarrow \lambda$, $B \leftarrow 0$, $\Delta \leftarrow 10$
20: **else**
21: **if** $f(y) = f(x)$ **then**
22: $x \leftarrow y$
23: **end if**
24: $B \leftarrow B + 1$
25: **if** $B = \Delta$ **then**
26: $B \leftarrow 0$, $\Delta \leftarrow \Delta + 1$
27: **end if**
28: $\lambda \leftarrow \min\{\lambda_0 F^{B/(U-1)}, \overline{\lambda}\}$
29: **end if**
30: **end for**

Fig. 5. The $(1 + (\lambda, \lambda))$ genetic algorithm with the modified adaptive choice of $\lambda \le \overline{\lambda}$

Рис. 5. Генетический алгоритм $(1 + (\lambda, \lambda))$ с модифицированным адаптивным выбором параметра $\lambda \le \overline{\lambda}$

Данная модификация также оставляет возможность ограничения значения λ сверху некоторым заданным значением $\overline{\lambda}$. В рамках экспериментального исследования будут рассмотрены два варианта ограничения: $\overline{\lambda} = n$ и $\overline{\lambda} = 2\ln(n+1)$, что позволяет оценить влияние предложенной стратегии в совокупности с влиянием ограничения наибольшего значения λ, и при независимом влиянии новой стратегии.

Рисунок 6 представляет поведение значений размера популяции λ на сериях итераций генетического алгоритма, не находящих лучшее решение. Эта модификация приблизительно возводит в квадрат число итераций, необходимых $(1 + (\lambda, \lambda))$-ГА для достижения некоторого достаточно большого значения λ. Как следствие, в ситуациях, когда λ превышает оптимальное значение для текущего расстояния до оптимума, предлагаемый метод адаптации, в отличие от исходной стратегии настройки параметров, все еще способен регулярно выполнять итерации с использованием значения λ, эффективного в данный момент времени, даже если улучшение приспособленности еще не наступило.

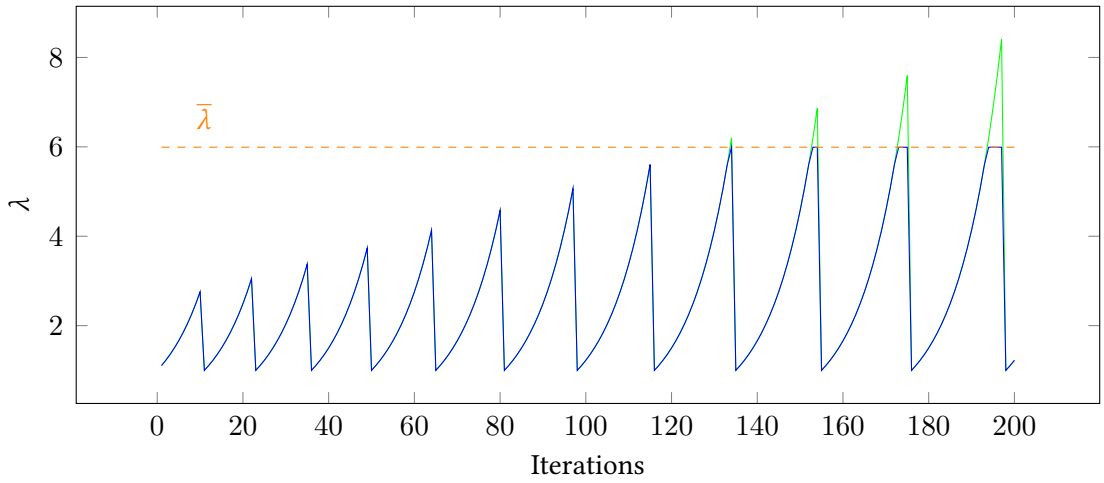


Fig. 6. The behavior of λ in the proposed modification in the case of no improvements

Рис. 6. График поведения λ в предлагаемом методе адаптации при отсутствии улучшений

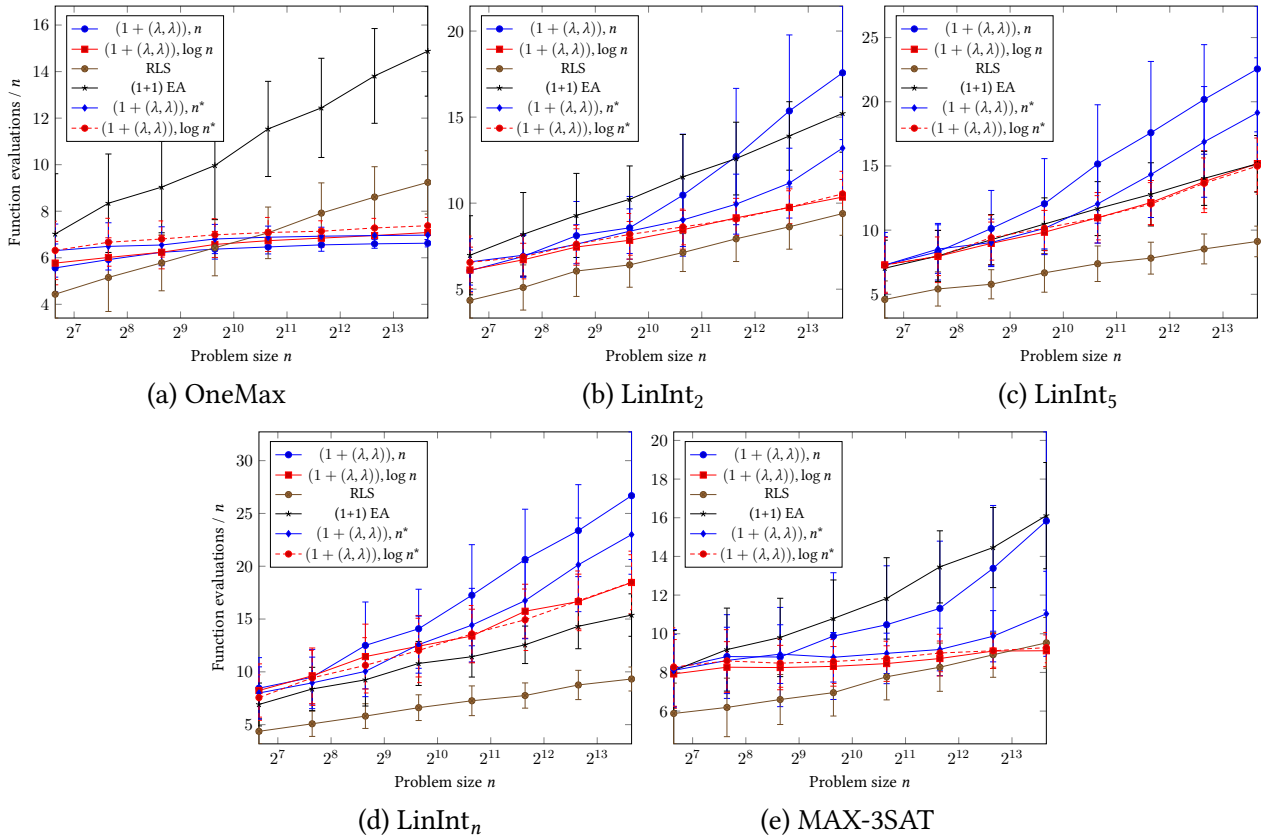
4. Экспериментальные результаты

В данном разделе описывается постановка и результаты экспериментального исследования, нацеленного на сравнение исходной версии $(1 + (\lambda, \lambda))$-ГА и предложенной в данной работе модификации. Каждый из алгоритмов рассматривался в двух вариантах: ограничение сверху на размер популяции составляло $\overline{\lambda} = n$ и $\overline{\lambda} = 2\ln(n + 1)$. Для краткости первый вариант будем называть *неограниченным*, а второй — *с логарифмическим ограничением*. Используется скорость адаптации $F = 1.5$. Для полноты картины в сравнении также участвовали алгоритмы $(1 + 1)$ ЭА и RLS, причем в первом из них, с целью приведения всех алгоритмов в одинаковые условия, стандартная битовая мутация использовалась также в практико-ориентированной модификации: выборка числа инвертируемых бит из биномиального распределения осуществлялась, пока это число не станет ненулевым.

В качестве задач оптимизации использовались те же пять задач, что и в разделе об исследовании ландшафта параметров: OneMax, LinInt_2, LinInt_5, LinInt_n, MAX-3SAT. Рассматривались размеры задач из множества $n \in \{100, 200, 400, 800, 1600, 3200, 6400, 12800\}$. Было выполнено 100 независимых запусков каждого алгоритма на каждой задаче для каждого из указанных размеров n.

Результаты представлены на рисунке 7. Алгоритмы с предложенной модификацией обозначены звездочкой. При решении задачи OneMax все вариации $(1 + (\lambda, \lambda))$-ГА ожидаемо показали хороший результат (рисунок 7a). В частности, варианты с логарифмическим ограничением немного уступают неограниченным вариантам, что также было ожидаемо, поскольку на последних итерациях варианты с логарифмическим ограничением используют субоптимальные значения λ. Предлагаемый алгоритм немного уступает исходному при сравнении соответствующих вариантов, но различие остается небольшим. Предполагается, что с увеличением размера задачи разница стремится приблизительно к 10%. Как будет показано в следующем разделе, асимптотическое поведение модифицированного $(1 + (\lambda, \lambda))$-ГА остается линейным на задаче OneMax.

На задаче LinInt_2 (рисунок 7b) обе версии $(1 + (\lambda, \lambda))$-ГА с логарифмическим ограничением попрежнему показывают достаточно высокую производительность, хотя асимптотика их времени работы, по-видимому, уже отличается от линейной. На рассмотренных размерах задачи их производительность лучше, чем у $(1 + 1)$-ЭА, и немного хуже, чем у RLS, однако тенденции, выраженные наклоном графиков, показывают, что при достаточно больших размерах задач данные алгоритмы превзойдут по производительности и RLS. Неограниченные варианты алгоритма показывают заметно худшие результаты. Исходный неограниченный вариант $(1 + (\lambda, \lambda))$-ГА проявляет тенденцию



(a) OneMax (b) LinInt_2 (c) LinInt_5

(d) LinInt_n (e) MAX-3SAT

Fig. 7. Runtime plots of the algorithms on the benchmark problems. Modified algorithms are marked with a star

Рис. 7. Графики времени работы алгоритмов на модельных задачах. Модифицированные алгоритмы обозначены звездочкой

к замедлению уже при $n = 1600$ и становится хуже, чем (1+1)-ЭА, при $n = 6400$. В свою очередь, модифицированный неограниченный вариант $(1 + (\lambda, \lambda))$-ГА только начинает замедляться при $n = 6400$, но все еще решает задачу быстрее, чем (1 + 1) ЭА, поэтому можно утверждать, что, по крайней мере, константный множитель при асимптотическом выражении для времени выполнения у модифицированной версии лучше, чем у исходной версии $(1 + (\lambda, \lambda))$-ГА.

Аналогичная ситуация наблюдается на рисунке 7c при решении задачи LinInt_5 и на рисунке 7d для LinInt_n. На этих задачах неограниченные варианты $(1 + (\lambda, \lambda))$-ГА демонстрируют худшее время работы по сравнению с (1 + 1)-ЭА, начиная с некоторого значения размера задачи. При этом стоит отметить, что предложенная в данной работе модификация стратегии настройки параметров всегда показывает лучший результат, чем исходная стратегия, в случае неограниченных вариантов, и не отличается от исходной стратегии в случае вариантов с логарифмическим ограничением. Асимптотическое поведение всех рассмотренных алгоритмов, судя по форме графиков, одинаково и составляет $\Theta(n \log n)$, таким образом, влияние модификаций и ограничений на этих задачах ограничивается вкладом в константный множитель при асимптотике и слагаемых меньших порядков.

Поведение алгоритмов на задаче MAX-3SAT (рисунок 7e) выглядит немного иначе, поскольку график неограниченного исходного варианта $(1 + (\lambda, \lambda))$-ГА растет быстрее (возможно, даже асимптотически) по сравнению с неограниченным модифицированным вариантом. Дальнейшее исследование этого поведения представляет интерес, однако вычисление функции приспособленности, в том числе ее пересчитывание при изменении небольшого числа бит, является для MAX-3SAT более трудоемкой задачей. Также из графиков можно сделать нетривиальный вывод, что данная задача для рассматриваемых вариантов $(1 + (\lambda, \lambda))$-ГА является более простой, чем даже LinInt_2.

5. Анализ времени работы предложенной модификации на задаче OneMax

Данный раздел посвящен доказательству того, что $(1 + (\lambda, \lambda))$-ГА с использованием предложенного в настоящей работе метода адаптации параметров сохраняет линейное время работы на задаче OneMax. Приведенные доказательства опираются на теоремы и утверждения из работы [69], посвященной аналогичным доказательствам для исходной версии алгоритма. Отметим, что ряд предположений, на которые опираются теоремы в указанной работе, остаются в силе и в случае модифицированной версии: так, например, используются те же значения параметров, задающих силу мутации ($p = \lambda/n$) и смещенность скрещивания ($c = 1/\lambda$), также коэффициент скорости адаптации F находится в тех же пределах $(1; 2)$.

Теорема 1. *Число запросов к функции приспособленности, выполненных $(1 + (\lambda, \lambda))$-ГА с предложенным в настоящей работе методом настройки размера популяции λ, значением $F \in (1; 2)$ и зависимостями значений параметров от λ в виде $p = \lambda/n$ и $c = 1/\lambda$, составляет $O(n)$ при оптимизации функции* OneMax.

Доказательство. Принцип выполнения доказательства остается неизменным по сравнению с теоремой 5 из [69]. В частности, основным лейтмотивом является доказательство того, что размер популяции λ не отклоняется значительно от оптимального выбора $\lambda^* = \lceil \sqrt{n/n - f(x)} \rceil$, для которого линейное время работы доказано в теореме 2 из [69].

Пусть $x \in \{0, 1\}^n$, $\lambda \geq C_0 \lceil \sqrt{n/n - f(x)} \rceil$ и $q = q(\lambda)$ — вероятность того, что итерация ГА, начинающаяся с особью x, будет успешной. По лемме 6 из [69] следует, что $q > 1/5$ для всех $C_0 > C$. Такая достаточно высокая вероятность дает возможность предположить, что на одной из следующих итераций значение λ уменьшится и устремится к λ^* для случаев, когда λ намного больше λ^*.

Аналогично тому, как это было сделано в [69], в данном доказательстве процесс оптимизации делится на *фазы* двух типов. Первый тип называется *короткой* фазой — в ней неравенство $\lambda \leq C_0\lambda^*$ остается верным на протяжении всех итераций ГА, составляющих эту фазу. Пусть $\tilde{\lambda}$ — начальное значение λ, а t — число итераций в фазе. На каждой итерации ГА производится 2λ вычислений функции приспособленности, следовательно, общее число вычислений функции приспособленности для модифицированного алгоритма на этой фазе выглядит следующим образом.
Если $t \leq \tilde{d}$:

$$\sum_{i=0}^{t-1} 2\tilde{\lambda}F^{i/4} = 2\tilde{\lambda}\frac{F^{t/4} - 1}{F^{1/4} - 1} = O(\lambda^*), \tag{1}$$

иначе:

$$\sum_{d=\tilde{d}}^{k-1}\sum_{i=0}^{d-1} 2\tilde{\lambda}F^{i/4} + \sum_{j=0}^{c-1} 2\tilde{\lambda}F^{j/4} \leq \sum_{i=0}^{t-1} 2\tilde{\lambda}F^{i/4} = 2\tilde{\lambda}\frac{F^{t/4} - 1}{F^{1/4} - 1} = O(\lambda^*) \tag{2}$$

$$\sum_{d=\tilde{d}}^{k-1}\sum_{i=0}^{d-1} 2\tilde{\lambda}F^{i/4} + \sum_{j=0}^{c-1} 2\tilde{\lambda}F^{j/4} = 2\tilde{\lambda}\frac{\sum_{d=\tilde{d}}^{k-1}(F^{d/4} - 1) + F^{c/4} - 1}{F^{1/4} - 1} \geq O(C_0\lambda^*) = O(\lambda^*), \tag{3}$$

где $(k - \tilde{d})$ — число сбросов λ до последнего успешного значения, c — число итераций после последнего сброса λ, $\sum_{i=0}^{k}(\tilde{d} + k) + c = t$ и $\tilde{d} = 10$.

Вторым типом стадии оптимизации является *длительная* фаза. В длительной фазе неравенство $\lambda \leq C_0\lambda^*$ выполняется хотя бы на одной итерации. В свою очередь, каждая *длительная* фаза разделяется на *открывающую* подфазу, которая заканчивается вместе с последней итерацией, удовлетворяющей $\lambda < C_0\lambda^*$, и на *главную* подфазу, которая начинается с размера популяции $C_0\lambda^* \leq \lambda < C_0\lambda^* F^{1/4}$.

Для *открывающей* подфазы число вычислений функции приспособленности можно рассчитать аналогично неравенствам (1), (2) и (3): с поправкой на то, что число итераций равно $t = m$, где

$m = \max\{k \mid \lambda F^{k/4} < C_0\lambda^*\}$, общее число запросов к функции приспособленности составляет $O(\lambda^*)$. Таким же образом можно оценить затраты на *главной* подфазе с учетом того, что начальное значение λ не превосходит $C_0\lambda^* F^{1/4}$.

Рассмотрим случай, который отличается от исходной стратегии настройки параметров по правилу «одной пятой», а именно при $t > \tilde{d}$:

$$C_0\lambda^* F^{1/4}\left(\sum_{d=\tilde{d}}^{k-1}\sum_{i=0}^{d} F^{i/4} + \sum_{j=0}^{c} F^{j/4}\right) \le C_0\lambda^* F^{1/4}\sum_{i=1}^{t} F^{i/4} \le D' C_0\lambda^* F^{(t+1)/4}, \tag{4}$$

для $D' \ge 1/(F^{1/4} - 1)$, где $(k - \tilde{d})$ — число сбросов λ до последнего успешного значения, c — число итераций после последнего сброса λ, $\sum_{i=0}^{k}(\tilde{d} + k) + c = t$ и $\tilde{d} = 10$.

Неравенство (4) является эквивалентом утверждения 2.1 из [69]: $E[T \mid I = t] \le D\lambda^* F^{t/4}$ для достаточно большой константы D, где T — число вычислений функции приспособленности на протяжении *длительной* фазы, а I — число итераций в *главной* подфазе.

Для рассматриваемой модификации остается в силе и утверждение 2.2 из [69], которое определяет вероятность необходимости t итераций: $\Pr[I = t] = e^{-ct}, c > 0$. Данное утверждение следует из того, что, при отсутствии успешных итераций, λ со временем достигает произвольно больших значений, и уменьшение размера популяции возможно в ситуации, когда λ превышает $C_0\lambda^*$.

Общее число вычислений функции приспособленности в ходе *длительной* фазы составляет:

$$\sum_{t=1}^{\infty} E[T \mid I = t]\Pr[I = t] \le D\lambda^* \sum_{t=1}^{\infty} F^{t/4} e^{-ct},$$

что при $F^{1/4} < e^c$ равно $O(\lambda^*)$. □

6. Аппроксимация производительности при оптимальном выборе параметров

Несмотря на то, что предложенная в настоящей работе модифицированная версия правила «одной пятой» позволяет добиться прироста производительности $(1 + (\lambda, \lambda))$-ГА на задачах, более сложных, чем OneMax, вопрос о теоретическом пределе производительности этого алгоритма при оптимальной адаптации параметров, остается нерешенным (в отличие от задачи OneMax, для которой ответ уже известен [66]). С целью исследования этого вопроса было проведено повторное использование экспериментальных результатов из раздела 2: для каждого значения расстояния Хэмминга до оптимума были выбраны значения λ, соответствующие минимальному времени ожидания улучшения.

Далее, для всех алгоритмов, исследованных в разделе 4, а также для $(1 + (\lambda, \lambda))$-ГА, который выбирает значения λ из таблицы, рассчитанной по указанному выше алгоритму для решаемой задачи, на основе расстояния Хэмминга текущего лучшего решения до оптимума, были проведены дополнительные экспериментальные исследования при фиксированном размере задачи n для всех ранее рассмотренных задач: OneMax, LinInt_2, LinInt_5, LinInt_n, MAX-3SAT. Последний алгоритм назовем *псевдооптимальным* $(1 + (\lambda, \lambda))$-ГА.

Результаты представлены в таблице 1, где алгоритмы с предложенной в работе модификацией правила «одной пятой» отмечены звездочками, а псевдооптимальный $(1+(\lambda, \lambda))$-ГА помечен словом «pseudo». Представленные результаты показывают, что все алгоритмы (за исключением RLS на задачах, отличных от OneMax) уступают по производительности псевдооптимальному $(1+(\lambda, \lambda))$-ГА на всех задачах, включая задачу LinInt_n, для которой структура $(1 + (\lambda, \lambda))$-ГА уже, по-видимому, не является перспективной. Таким образом, приведенные наблюдения демонстрируют значительный потенциал методов адаптации параметров и мотивируют их дальнейшее улучшение в применении к $(1 + (\lambda, \lambda))$-ГА.

Table 1. Comparison of existing algorithms and the pseudo-optimal $(1 + (\lambda, \lambda))$ GA, problem size $n = 1000$

Problem	Algorithm	Evaluations
OneMax	RLS	6600 ± 1200
	(1+1) EA	10900 ± 2400
	$(1 + (\lambda, \lambda))$, n	6400 ± 400
	$(1 + (\lambda, \lambda))$, $\log n$	6600 ± 600
	$(1 + (\lambda, \lambda))$, n *	9300 ± 2200
	$(1 + (\lambda, \lambda))$, $\log n$ *	9700 ± 2600
	$(1 + (\lambda, \lambda))$, pseudo	5700 ± 400
LinInt_2	RLS	6700 ± 1100
	(1+1) EA	10900 ± 2000
	$(1 + (\lambda, \lambda))$, n	9200 ± 2300
	$(1 + (\lambda, \lambda))$, $\log n$	8100 ± 1100
	$(1 + (\lambda, \lambda))$, n *	9900 ± 2300
	$(1 + (\lambda, \lambda))$, $\log n$ *	9900 ± 1900
	$(1 + (\lambda, \lambda))$, pseudo	7500 ± 1200
LinInt_5	RLS	6900 ± 1300
	(1+1) EA	10700 ± 2300
	$(1 + (\lambda, \lambda))$, n	12600 ± 3300
	$(1 + (\lambda, \lambda))$, $\log n$	10400 ± 1700
LinInt_{1000}	RLS	6800 ± 1400
	(1+1) EA	11200 ± 2400
	$(1 + (\lambda, \lambda))$, n	15400 ± 4300
	$(1 + (\lambda, \lambda))$, $\log n$	12800 ± 2700
	$(1 + (\lambda, \lambda))$, n *	12400 ± 2600
	$(1 + (\lambda, \lambda))$, $\log n$ *	12300 ± 2400
	$(1 + (\lambda, \lambda))$, pseudo	11000 ± 2300
MAX-3SAT	RLS	7500 ± 1400
	(1+1) EA	11100 ± 2100
	$(1 + (\lambda, \lambda))$, n	9800 ± 3100
	$(1 + (\lambda, \lambda))$, $\log n$	8400 ± 1000
	$(1 + (\lambda, \lambda))$, n *	10300 ± 2000
	$(1 + (\lambda, \lambda))$, $\log n$ *	10600 ± 2100
	$(1 + (\lambda, \lambda))$, pseudo	7800 ± 900
LinInt_5	$(1 + (\lambda, \lambda))$, n *	11100 ± 2100
	$(1 + (\lambda, \lambda))$, $\log n$ *	11000 ± 2000
	$(1 + (\lambda, \lambda))$, pseudo	9500 ± 1600

Таблица 1. Сравнение существующих алгоритмов и псевдооптимального $(1 + (\lambda, \lambda))$-ГА, размерность задач $n = 1000$

7. Заключение

В работе предложена модификация правила «одной пятой», которая используется для настройки параметра λ в генетическом алгоритме $(1 + (\lambda, \lambda))$. Предлагаемый метод направлен на снижение нежелательных эффектов, приводящих к снижению производительности при решении как задач со сниженной корреляцией между приспособленностью и расстоянием до оптимума (линейные функции со случайными целочисленными весами, ограниченными константой), так и задач со слабой или нулевой корреляцией между приспособленностью и расстоянием до оптимума (линейные функции с линейными случайными весами), а также и других задач оптимизации (случайные экземпляры задачи MAX-3SAT в модели *planted solution* и логарифмической плотностью дизъюнктов). Основная идея метода состоит в том, чтобы замедлить рост λ при длительных сериях неудачных итераций. В терминах числа итераций генетического алгоритма $(1 + (\lambda, \lambda))$, скорость роста λ, измеренная по пикам, становится приблизительно равной квадратным корням от исходной скорости.

Несмотря на то, что предложенный метод определенно не является окончательным решением указанной проблемы, в работе показано стабильное улучшение производительности модифицированной версии по сравнению с исходным $(1 + (\lambda, \lambda))$-ГА на всех проблемных функциях. На OneMax предложенная стратегия работает приблизительно на 10% хуже, однако ее время выполнения остается линейным не только на практике, но и в теории, что было доказано по аналогии с соответствующим доказательством для исходной версии.

Также была исследована теоретически возможная производительность $(1 + (\lambda, \lambda))$-ГА в случае, когда выбор λ близок к оптимально возможному. Для этого для исследованных задач были предподсчитаны оптимальные значения λ в зависимости от расстояния Хэмминга текущего лучшего решения до оптимума, которые затем были использованы в экспериментах. Несмотря на то, что полученный вариант алгоритма не является практически значимым, данная часть исследования демонстрирует, что теоретически возможности структуры $(1+(\lambda, \lambda))$-ГА для решения сложных задач шире, чем представлялось ранее.

References

[1] J. H. Holland, *Adaptation in Natural and Artificial Systems*. University of Michigan, 1975, 211 pp.

[2] I. Rechenberg, *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Stuttgart: Fromman-Holzboorg Verlag, 1973.

[3] H.-P. Schwefel, «Binäre Optimierung durch somatische Mutation», Technical University of Berlin and Medical University of Hannover, Tech. Rep., May 1975.

[4] L. J. Fogel, «Autonomous automata», *Industrial Research*, vol. 4, pp. 14–19, 1962.

[5] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*. Cambridge, MA, USA: MIT Press, 1992.

[6] R. C. Eberhart and J. Kennedy, «A new optimizer using particle swarm theory», in *Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, 1995, pp. 39–43.

[7] M. Dorigo and L. M. Gambardella, «Ant colony system: A cooperative learning approach to the traveling salesman problem», *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 53–66, 1997.

[8] R. Poli, J. Kennedy, and T. Blackwell, «Particle swarm optimization, An overview», *Swarm Intelligence*, vol. 1, pp. 33–57, 2007.

[9] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, «Optimization by simulated annealing», *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[10] J. M. Bishop, «Stochastic searching networks», in *Proceedings of 1st IEEE Conference on Artificial Neural Networks*, 1989, pp. 329–331.

[11] R. Storn and K. Price, «Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces», *Journal of Global Optimization*, vol. 11, no. 4, pp. 341–359, 1997.

[12] M. Pelikan, D. E. Goldberg, and E. Cantu-Paz, «Linkage problem, distribution estimation, and bayesian networks», *Evolutionary Computation*, vol. 8, no. 3, pp. 311–340, 2000.

[13] B. Doerr and M. S. Krejca, «Significance-based estimation-of-distribution algorithms», in *Proceedings of Genetic and Evolutionary Computation Conference*, 2018, pp. 1483–1490.

[14] S. Luke, *Essentials of Metaheuristics*. Lulu, 2009, 253 pp.

[15] A. Orlov, V. V. Kureichik, and A. Glushchenko, «Hybrid genetic algorithm for cutting stock and packaging problems», in *Proceedings of East-West Design & Test Symposium*, IEEE, 2016.

[16] L. A. Gladkov, N. V. Gladkova, and S. A. Gromov, «Hybrid models of solving optimization tasks on the basis of integrating evolutionary design and multiagent technologies», in *Proceedings of Computer Science Online Conference*, ser. Advances in Intelligent Systems and Computing 985, 2019, pp. 381–391.

[17] E. V. Kuliev, A. N. Dukkardt, V. V. Kureychik, and A. A. Legebokov, «Neighborhood research approach in swarm intelligence for solving the optimization problems», in *Proceedings of East-West Design & Test Symposium*, IEEE, 2014.

[18] E. V. Kuliev, V. V. Kureichik, and I. O. Kursitys, «Decision making in VLSI components placement problem based on grey wolf optimization», in *Proceedings of East-West Design & Test Symposium*, IEEE, 2019.

[19] V. V. Kureichik and D. V. Zaruba, «Combined approach to place electronic computing equipment circuit elements», in *Proceedings of East-West Design & Test Symposium*, IEEE, 2015.

[20] L. A. Gladkov, N. V. Gladkova, and S. Leiba, «Electronic computing equipment schemes elements placement based on hybrid intelligence approach», in *Proceedings of Computer Science Online Conference*, ser. Advances in Intelligent Systems and Computing 348, 2015, pp. 35–44.

[21] M. Semenkina, «Parallel version of self-configuring genetic algorithm application in spacecraft control system design», in *Proceedings of Genetic and Evolutionary Computation Conference Companion*, 2013, pp. 1751–1752.

[22] D. Chivilikhin, V. Ulyantsev, and A. Shalyto, «Modified ant colony algorithm for constructing finite state machines from execution scenarios and temporal formulas», *Automation and Remote Control*, vol. 77, no. 3, pp. 473–484, 2016.

[23] C. M. Fonseca and P. J. Fleming, «Nonlinear system identification with multiobjective genetic algorithm», in *Proceedings of the World Congress of the International Federation of Automatic Control*, 1996, pp. 187–192.

[24] I. Buzhinsky, V. Ulyantsev, D. Chivilikhin, and A. Shalyto, «Inducing finite state machines from training samples using ant colony optimization», *Journal of Computer and Systems Sciences International*, vol. 53, no. 2, pp. 256–266, 2014.

[25] D. Chivilikhin, V. Ulyantsev, and A. Shalyto, «Extended finite-state machine inference with parallel ant colony based algorithms», in *International Student Workshop on Bioinspired Optimization Methods and Their Applications*, 2014, pp. 117–126.

[26] D. Chivilikhin, V. Ulyantsev, and A. Shalyto, «Combining exact and metaheuristic techniques for learning extended finite state machines from test scenarios and temporal properties», in *Proceedings of International Conference on Machine Learning and Applications*, 2014, pp. 350–355.

[27] I. Buzhinsky, V. Ulyantsev, F. Tsarev, and A. Shalyto, «Search-based construction of finite-state machines with real-valued actions: New representation model», in *Proceedings of Genetic and Evolutionary Computation Conference Companion*, 2013, pp. 199–200.

[28] S. El-Khatib, Y. Skobtsov, and S. Rodzin, «Improved particle swarm medical image segmentation algorithm for decision making», in *Intelligent Distributed Computing XIII*, ser. Studies in Computational Intelligence 868, 2019, pp. 437–442.

[29] A. V. Eremeev and Y. V. Kovalenko, «Genetic algorithm with optimal recombination for the asymmetric travelling salesman problem», in *LSSC 2017: Large-Scale Scientific Computing*, ser. Lecture Notes in Computer Science 10665, 2017, pp. 341–349.

[30] A. V. Eremeev and Y. V. Kovalenko, «A memetic algorithm with optimal recombination for the asymmetric travelling salesman problem», *Memetic Computing*, vol. 12, no. 1, pp. 23–36, 2020.

[31] D. S. Sanches, D. Whitley, and R. Tinós, «Improving an exact solver for the traveling salesman problem using partition crossover», in *Proceedings of Genetic and Evolutionary Computation Conference*, 2017, pp. 337–344.

[32] L. D. Whitley, F. Chicano, and B. W. Goldman, «Gray box optimization for Mk landscapes (NK landscapes and MAX-kSAT)», *Evolutionary Computation*, vol. 24, no. 3, pp. 491–519, 2016.

[33] A. Glotić and A. Zamuda, «Short-term combined economic and emission hydrothermal optimization by surrogate differential evolution», *Applied Energy*, vol. 141, pp. 42–56, 2015.

[34] T. Liao, P. J. Egbelu, and P. Chang, «Two hybrid differential evolution algorithms for optimal inbound and outbound truck sequencing in cross docking operations», *Applied Soft Computing*, vol. 12, no. 11, pp. 3683–3697, 2012.

[35] V. Feoktistov, S. Pietravalle, and N. Heslot, «Optimal experimental design of field trials using differential evolution», in *Proceedings of Congress on Evolutionary Computation*, 2017, pp. 1690–1696.

[36] T. Bäck, D. B. Fogel, and Z. Michalewicz, Eds., *Evolutionary Computation 1: Basic Algorithms and Operators*. Institute of Physics Publishing, 2000, 339 pp.

[37] J. J. Grefenstette, «Optimization of control parameters for genetic algorithms», *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 16, pp. 122–128, 1986.

[38] H. Mühlenbein, «How genetic algorithms really work: Mutation and hillclimbing», in *Parallel Problem Solving from Nature – PPSN II*, Elsevier, 1992, pp. 15–26.

[39] Á. E. Eiben, R. Hinterding, and Z. Michalewicz, «Parameter control in evolutionary algorithms», *IEEE Transactions on Evolutionary Computation*, vol. 3, no. 2, pp. 124–141, 1999.

[40] V. Stanovov, S. Akhmedova, E. Semenkin, and M. Semenkina, «Generalized Lehmer mean for success history based adaptive differential evolution», in *Proceedings of International Joint Conference on Computational Intelligence*, 2019, pp. 93–100.

[41] V. Stanovov, S. Akhmedova, and E. Semenkin, «LSHADE algorithm with rank-based selective pressure strategy for solving CEC 2017 benchmark problems», in *Proceedings of Congress on Evolutionary Computation*, IEEE, 2018.

[42] M. Semenkina and E. Semenkin, «Memetic self-configuring genetic programming for solving machine learning problems», in *Proceedings of International Congress on Advanced Applied Informatics*, 2015, pp. 599–604.

[43] M. Semenkina, S. Akhmedova, C. Brester, and E. Semenkin, «Choice of spacecraft control contour variant with self-configuring stochastic algorithms of multi-criteria optimization», in *Proceedings of International Conference on Informatics in Control, Automation and Robotics*, 2016, pp. 281–286.

[44] V. Stanovov, E. Semenkin, and O. Semenkina, «Self-configuring hybrid evolutionary algorithm for fuzzy imbalanced classification with adaptive instance selection», *Journal of Artificial Intelligence and Soft Computing Research*, vol. 6, no. 3, pp. 173–188, 2016.

[45] N. Hansen and A. Ostermeier, «Completely derandomized self-adaptation in evolution strategies», *Evolutionary Computation*, vol. 9, pp. 159–195, 2001.

[46] R. Tanabe and A. Fukunaga, «Success-history based parameter adaptation for differential evolution», in *Proceedings of IEEE Congress on Evolutionary Computation*, 2013, pp. 71–78.

[47] R. Senkerik, M. Pluhacek, T. Kadavy, and A. Zamuda, «Distance based parameter adaptation for success-history based differential evolution», *Swarm and Evolutionary Computation*, vol. 50, 2019. DOI: 10.1016/j.swevo.2018.10.013.

[48] N. Dang and C. Doerr, «Hyper-parameter tuning for the $(1 + (\lambda, \lambda))$ GA», in *Proceedings of Genetic and Evolutionary Computation Conference*, 2019, pp. 889–897.

[49] E. Ridge and D. Kudenko, «Tuning an algorithm using design of experiments», in *Experimental Methods for the Analysis of Optimization Algorithms*, T. Bartz-Beielstein, M. Chiarandini, L. Paquete, and M. Preuss, Eds. Springer, 2010, pp. 265–286.

[50] M. López-Ibáñez, J. Dubois-Lacoste, L. P. Cáceres, T. Stützle, and M. Birattari, «The irace package: Iterated racing for automatic algorithm configuration», *Operations Research Perspectives*, vol. 3, pp. 43–58, 2016.

[51] F. Hutter, H. H. Hoos, and K. Leyton-Brown, «Sequential model-based optimization for general algorithm configuration», in *Proceedings of Learning and Intelligent Optimization*, Springer, 2011, pp. 507–523.

[52] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, «ParamILS: An automatic algorithm configuration framework», *Journal of Artificial Intelligence Research*, vol. 36, pp. 267–306, 2009.

[53] I. Wegener, «Methods for the analysis of evolutionary algorithms on pseudo-Boolean functions», in *Evolutionary Optimization*, ser. International Series in Operations Research & Management Science 48, 2003, pp. 349–369.

[54] A. Auger and B. Doerr, *Theory of Randomized Search Heuristics: Foundations and Recent Developments*. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 2011.

[55] B. Doerr and F. Neumann, Eds., *Theory of Evolutionary Computation—Recent Developments in Discrete Optimization*. Springer, 2020.

[56] V. G. Red'ko and Y. Tsoy, «Estimation of the evolution speed for the quasispecies model: Arbitrary alphabet case», in *Artificial Intelligence and Soft Computing – ICAISC 2006*, ser. Lecture Notes in Computer Science 4029, 2006, pp. 460–469.

[57] V. G. Red'ko, O. P. Mosalov, and D. V. Prokhorov, «Investigation of evolving populations of adaptive agents», in *Artificial Neural Networks: Biological Inspirations – ICANN 2005*, ser. Lecture Notes in Computer Science 3696, 2005, pp. 337–342.

[58] A. Antamoshkin, E. Antamoshkin, and E. Semenkin, «Local search efficiency when optimizing unimodal pseudoboolean functions», *Informatica*, vol. 9, no. 3, 1998.

[59] A. N. Antamoshkin, V. N. Saraev, and E. Semenkin, «Optimization of unimodal monotone pseudoboolean functions», *Kybernetika*, vol. 26, no. 5, pp. 432–442, 1990.

[60] S. Rodzin and L. Rodzina, «Theory of bionic optimization and its application to evolutionary synthesis of digital devices», in *Proceedings of East-West Design & Test Symposium*, IEEE, 2014.

[61] S. El-Khatib, Y. Skobtsov, S. Rodzin, and S. Potryasaev, «Theoretical and experimental evaluation of PSO-K-Means algorithm for MRI images segmentation using drift theorem», in *Proceedings of Computer Science Online Conference*, ser. Advances in Intelligent Systems and Computing 985, 2019, pp. 316–323.

[62] P. A. Borisovsky and A. V. Eremeev, «Comparing evolutionary algorithms to the (1+1)-EA», *Theoretical Computer Science*, vol. 403, no. 1, pp. 33–41, 2008. DOI: 10.1016/j.tcs.2008.03.008.

[63] D. Corus, D.-C. Dang, A. V. Eremeev, and P. K. Lehre, «Level-based analysis of genetic algorithms and other search processes», *IEEE Transactions on Evolutionary Computation*, vol. 22, no. 5, pp. 707–719, 2018.

[64] A. V. Eremeev, «On non-elitist evolutionary algorithms optimizing fitness functions with a plateau», in *Mathematical Optimization Theory and Operations Research*, ser. Lecture Notes in Computer Science 12095, 2020, pp. 329–342.

[65] A. V. Eremeev, «On proportions of fit individuals in population of mutation-based evolutionary algorithm with tournament selection», *Evolutionary Computation*, vol. 26, no. 2, pp. 269–297, 2018.

[66] B. Doerr, C. Doerr, and F. Ebel, «From black-box complexity to designing new genetic algorithms», *Theoretical Computer Science*, vol. 567, pp. 87–104, 2015.

[67] B. Doerr, C. Doerr, and F. Ebel, «Lessons from the black-box: Fast crossover-based genetic algorithms», in *Proceedings of Genetic and Evolutionary Computation Conference*, 2013, pp. 781–788.

[68] B. Doerr and C. Doerr, «Optimal static and self-adjusting parameter choices for the $(1 + (\lambda, \lambda))$ genetic algorithm», *Algorithmica*, vol. 80, no. 5, pp. 1658–1709, 2018.

[69] B. Doerr and C. Doerr, «Optimal parameter choices through self-adjustment: Applying the 1/5-th rule in discrete settings», in *Proceedings of Genetic and Evolutionary Computation Conference*, 2015, pp. 1335–1342.

[70] D. Antipov, M. Buzdalov, and B. Doerr, «Fast mutation in crossover-based algorithms», in *Proceedings of Genetic and Evolutionary Computation Conference*, ACM, 2020, pp. 1268–1276.

[71] B. Goldman and W. Punch, «Parameter-less population pyramid», in *Proceedings of Genetic and Evolutionary Computation Conference*, 2014, pp. 785–792.

[72] M. Buzdalov and B. Doerr, «Runtime analysis of the $(1 + (\lambda, \lambda))$ genetic algorithm on random satisfiable 3-CNF formulas», in *Proceedings of Genetic and Evolutionary Computation Conference*, 2017, pp. 1343–1350.

[73] A. Gandomi and B. Goldman, «Parameter-less population pyramid for large-scale tower optimization», *Expert Systems with Applications*, vol. 96, pp. 175–184, 2018.

[74] V. Mironovich and M. Buzdalov, «Hard test generation for maximum flow algorithms with the fast crossover-based evolutionary algorithm», in *Proceedings of Genetic and Evolutionary Computation Conference Companion*, 2015, pp. 1229–1232.

[75] M. A. Hevia Fajardo and D. Sudholt, «On the choice of the parameter control mechanism in the $(1 + (\lambda, \lambda))$ genetic algorithm», in *Proceedings of Genetic and Evolutionary Computation Conference*, 2020, pp. 832–840.

[76] A. Bassin and M. Buzdalov, «The 1/5-th rule with rollbacks: On self-adjustment of the population size in the $(1 + (\lambda, \lambda))$ GA», in *Proceedings of Genetic and Evolutionary Computation Conference Companion*, 2019, pp. 277–278. [Online]. Available: https://arxiv.org/abs/1904.07284.

[77] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979, 338 pp.

[78] D. Mitchell, B. Selman, and H. Levesque, «Hard and easy distributions of SAT problems», in *Proceedings of AAAI Conference on Artificial Intelligence*, 1992, pp. 459–465.

[79] A. M. Sutton and F. Neumann, «Runtime analysis of evolutionary algorithms on randomly constructed high-density satisfiable 3-CNF formulas», in *Proceedings of Parallel Problem Solving from Nature*, ser. Lecture Notes in Computer Science 8672, 2014, pp. 942–951.

[80] B. Doerr, F. Neumann, and A. M. Sutton, «Improved runtime bounds for the (1+1) EA on random 3-CNF formulas based on fitness-distance correlation», in *Proceedings of Genetic and Evolutionary Computation Conference*, 2015, pp. 1415–1422.

[81] B. Doerr and C. Doerr, «A tight runtime analysis of the $(1 + (\lambda, \lambda))$ genetic algorithm on OneMax», in *Proceedings of Genetic and Evolutionary Computation Conference*, 2015, pp. 1423–1430.

[82] E. Carvalho Pinto and C. Doerr. (2018). Towards a more practice-aware runtime analysis of evolutionary algorithms, [Online]. Available: https://arxiv.org/abs/1812.00493.

[83] E. C. Pinto and C. Doerr, «A simple proof for the usefulness of crossover in black-box optimization», in *Parallel Problem Solving from Nature – PPSN XV, Vol. 2*, ser. Lecture Notes in Computer Science 11102, 2018, pp. 29–41.

[84] B. Doerr, «Optimal parameter settings for the $(1 + (\lambda, \lambda))$ genetic algorithm», in *Proceedings of Genetic and Evolutionary Computation Conference*, Full version available at http://arxiv.org/abs/1604.01088, 2016, pp. 1107–1114.

MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 27, NO. 4, 2020
journal homepage: www.mais-journal.ru

ERRATUM

Corrigendum to: V. A. Sokolov, "On the Existence Problem of Finite Bases of Identities in the Algebras of Recursive Functions", Modeling and analysis of information systems, vol. 27, no. 3, pp. 304-315, 2020. DOI: https://doi.org/10.18255/1818-1015-2020-3-304-315

V. A. Sokolov[1]

DOI: 10.18255/1818-1015-2020-4-510-511

[1]P. G. Demidov Yaroslavl State University, 14 Sovetskaya, Yaroslavl 150003, Russia.

MSC2020: 03D20
Erratum
Full text in Russian

Received December 11, 2020
After revision December 11, 2020
Accepted December 11, 2020

The author regrets that in the original list the references [3] and [4] are in the wrong places and they should be rearranged. In addition, [3] has the wrong article title. The corrected reference list is shown below.
The author would like to apologize for an inconvenience caused.

References

[1] A. I. Mal'tsev, "Constructive algebras I", Russian Mathematical Surveys, vol. 16, no. 3, pp. 77–129, 1961.
[2] A. I. Mal'tsev, Algoritmy i rekursivnye funktsii. Moscow: Nauka, 1965, In Russian.
[3] R. M. Robinson, "Primitive recursive functions", Bulletin of the American Mathematical Society, vol. 53, no. 10, pp. 925–942, 1947.
[4] J. Robinson, "General recursive functions", Proceedings of the American Mathematical Society, vol. 1, no. 6, pp. 703–718, 1950.
[5] V. A. Sokolov, "Ob odnom klasse tozhdestv v algebre Robinsona", in 14-ya Vsesoyuznaya algebraicheskaya konferentsiya: tezisy dokladov, In Russian, vol. 2, Novosibirsk, 1977, pp. 123–124.
[6] P. M. Cohn, Universal Algebra. New York, Evanston, and London: Harper & Row, 1965.
[7] A. Robinson, "Equational logic for partial functions under Kleene equality: a complete and an incomplete set of rules", The Journal of Symbolic Logic, vol. 54, no. 2, pp. 354–362, 1989.

Keywords: algebra; recursive function; identity; basis; superposition; iteration; function inversion.

INFORMATION ABOUT THE AUTHORS

Valery Anatolyevich Sokolov
correspondence author
orcid.org/0000-0003-1427-4937. E-mail: valery-sokolov@yandex.ru
Doctor of Science, Professor.

For citation: V. A. Sokolov, "Corrigendum to: V. A. Sokolov, "On the Existence Problem of Finite Bases of Identities in the Algebras of Recursive Functions", Modeling and analysis of information systems, vol. 27, no. 3, pp. 304-315, 2020. DOI: https://doi.org/10.18255/1818-1015-2020-3-304-315", *Modeling and analysis of information systems*, vol. 27, no. 4, pp. 510-511, 2020.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 27, № 4, 2020
сайт журнала: www.mais-journal.ru

ERRATUM

Исправление к статье: В. А. Соколов, “О проблеме существования конечных базисов тождеств в алгебрах рекурсивных функций”, Моделирование и анализ информационных систем, Том. 27, № 3, с. 304–315, 2020. DOI: https://doi.org/10.18255/1818-1015-2020-3-304-315

В. А. Соколов[1]

DOI: 10.18255/1818-1015-2020-4-510-511

[1]Ярославский государственный университет им. П. Г. Демидова, ул. Советская, 14, г. Ярославль, 150003 Россия.

УДК 512.57
Исправление
Полный текст на русском языке

Получена 11 декабря 2020 г.
После доработки 11 декабря 2020 г.
Принята к публикации 11 декабря 2020 г.

Автор сожалеет, что в исходном списке ссылки [3] и [4] находятся в неправильных местах и их следует переставить. Кроме того, [3] содержит ошибку в названии статьи. Исправленный список цитируемых источников приведен ниже. Автор приносит извинения за причиненные неудобства.

References

[1] A. I. Mal’tsev, “Constructive algebras I”, Russian Mathematical Surveys, vol. 16, no. 3, pp. 77–129, 1961.
[2] A. I. Mal’tsev, Algoritmy i rekursivnye funktsii. Moscow: Nauka, 1965, In Russian.
[3] R. M. Robinson, “Primitive recursive functions”, Bulletin of the American Mathematical Society, vol. 53, no. 10, pp. 925–942, 1947.
[4] J. Robinson, “General recursive functions”, Proceedings of the American Mathematical Society, vol. 1, no. 6, pp. 703–718, 1950.
[5] V. A. Sokolov, “Ob odnom klasse tozhdestv v algebre Robinsona”, in 14-ya Vsesoyuznaya algebraicheskaya konferentsiya: tezisy dokladov, In Russian, vol. 2, Novosibirsk, 1977, pp. 123–124.
[6] P. M. Cohn, Universal Algebra. New York, Evanston, and London: Harper & Row, 1965.
[7] A. Robinson, “Equational logic for partial functions under Kleene equality: a complete and an incomplete set of rules”, The Journal of Symbolic Logic, vol. 54, no. 2, pp. 354–362, 1989.

Ключевые слова: алгебра; рекурсивная функция; тождество; базис; суперпозиция; итерация; обращение функции.

ИНФОРМАЦИЯ ОБ АВТОРАХ

Валерий Анатольевич Соколов
автор для корреспонденции
orcid.org/0000-0003-1427-4937. E-mail: valery-sokolov@yandex.ru
доктор физ.-мат. наук, профессор.

Для цитирования: V. A. Sokolov, “Corrigendum to: V. A. Sokolov, “On the Existence Problem of Finite Bases of Identities in the Algebras of Recursive Functions”, Modeling and analysis of information systems, vol. 27, no. 3, pp. 304-315, 2020. DOI: https://doi.org/10.18255/1818-1015-2020-3-304-315”, *Modeling and analysis of information systems*, vol. 27, no. 4, pp. 510-511, 2020.