ISSN 1818-1015
eISSN 2313-5417

2024 Volume 31 No. 1

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

2024 Том 31 № 1

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 № ФС77-66186 от 20.06.2016 выдано Федеральной службой по надзору в сфере связи, информационных технологий и массовых коммуникаций. Подписной индекс в каталоге «Урал-Пресс» — 31907. Технический редактор, компьютерная вёрстка — К. В. Лагутина. Дата выхода в свет 31.03.2024. Формат 200×265 мм. Объем 114 с. Тираж 28 экз. Свободная цена. Заказ 24035. Издатель и его адрес: Ярославский государственный университет им. П. Г. Демидова; ул. Советская, 14, Ярославль, 150003, Россия. Типография и ее адрес: ООО «Филигрань»; ул. Свободы, 91, Ярославль, 150049, Россия.
Содержание предназначено для детей старше 12 лет.

Contents

Содержание

MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 31, NO. 1, 2024
journal homepage: www.mais-journal.ru

THEORY OF SOFTWARE

Requirement Patterns in Deductive Verification of poST Programs

I. M. Chernenko[1], I. S. Anureev[1], N. O. Garanina[1]

DOI: 10.18255/1818-1015-2024-1-6-31

[1]Institute of Automation and Electrometry SB RAS, Novosibirsk, Russia

MSC2020: 68N30
Research article
Full text in Russian

Received January 12, 2024
Revised February 1, 2024
Accepted February 7, 2024

Process-oriented programming is one of the approaches to developing control software. A process-oriented program is defined as a sequence of processes. Each process is represented by a set of named states containing program code that define the logic of the process's behavior. Program execution is sequential execution of each of these processes in their current states at every iteration of the control cycle. Processes can interact through changing each other's states and shared variables.

The paper expands a method for classifying temporal requirements for process-oriented programs in order to simplify and automate the deductive verification of such programs. The method consists of the following steps. At the first step, the requirements are formalized in a specialized language DV-TRL, a variant of typed first-order predicate logic with a set of interpreted types and predicate and functional symbols, that reflect specific concepts of control systems in a process-oriented paradigm. At the second step, the formalized requirements are divided into classes, each of which is defined by a pattern — a parametric formula of the DV-TRL language. The correctness conditions generated for process-oriented programs regarding requirements satisfying the same pattern have the same proof scheme. At the third step, appropriate proof schemes are developed. In our paper, we first give a brief introduction to the poST language, a process-oriented extension to the ST language of the IEC 61131-3 standard. Next, the DV-TRL language is defined. We also provide a collection of natural language requirements for several control systems. Then we define patterns that fully cover all the requirements of this collection. For each of these patterns we give an example of a formalized requirement from the collection and describe a scheme for proving the correctness conditions for this pattern. Statistics on the distribution of requirements from the collection across patterns reveals the most popular patterns. We also analyzed related works.

Keywords: deductive verification; temporal requirements; requirement patterns; control software; process-oriented programming

INFORMATION ABOUT THE AUTHORS

Chernenko, Ivan M.	ORCID iD: 0000-0001-7675-8449. E-mail: cheriv98@mail.ru Graduate student
Anureev, Igor S. (corresponding author)	ORCID iD: 0000-0001-9574-128X. E-mail: anureev@gmail.com Senior researcher, PhD
Garanina, Natalia O.	ORCID iD: 0000-0001-9734-3808. E-mail: garanina@iis.nsk.su Senior researcher, PhD

Funding: State task IAaE SB RAS, project No. 122031600173-8.

For citation: I. M. Chernenko, I. S. Anureev, and N. O. Garanina, "Requirement patterns in deductive verification of poST Programs", *Modeling and Analysis of Information Systems*, vol. 31, no. 1, pp. 6–31, 2024. DOI: 10.18255/1818-1015-2024-1-6-31.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 31, № 1, 2024
сайт журнала: www.mais-journal.ru

THEORY OF SOFTWARE

Шаблоны требований в дедуктивной верификации poST-программ

И. М. Черненко[1], И. С. Ануреев[1], Н. О. Гаранина[1]

DOI: 10.18255/1818-1015-2024-1-6-31

[1]Институт автоматики и электрометрии СО РАН, Новосибирск, Россия

УДК 004.415.52
Научная статья
Полный текст на русском языке

Получена 12 января 2024 г.
После доработки 1 февраля 2024 г.
Принята к публикации 7 февраля 2024 г.

Процесс-ориентированное программирование — один из подходов к разработке управляющего программного обеспечения. Процесс-ориентированная программа определяется как последовательность процессов. Каждый процесс представляется набором именованных состояний, содержащих программный код, которые задают логику поведения процесса. Выполнение программы заключается в последовательном исполнении этих процессов в их текущих состояниях на каждой итерации цикла управления. Процессы могут взаимодействовать через изменение состояний друг друга и через разделяемые переменные.

Статья является развитием метода классификации темпоральных требований к процесс-ориентированным программам с целью упростить и автоматизировать дедуктивную верификацию таких программ. Метод состоит из следующих шагов. На первом шаге требования формализуются на специализированном языке DV-TRL, варианте типизированной логики предикатов первого порядка с набором интерпретированных типов и предикатных и функциональных символов, позволяющем отражать специфические понятия систем управления в процесс-ориентированной парадигме. На втором шаге формализованные требования разбиваются на классы, каждый из которых определяется шаблоном — параметрической формулой языка DV-TRL, причем условия корректности, порождаемые для процесс-ориентированных программ относительно требований, удовлетворяющих одному шаблону, имеют одну и ту же схему доказательства. На третьем шаге разрабатываются соответствующие схемы доказательства. В статье мы сначала даём краткое введение в язык poST, процесс-ориентированное расширение языка ST стандарта МЭК 61131-3. Далее определяется язык DV-TRL. Мы также приводим коллекцию требований на естественном языке для нескольких систем управления. Затем мы определяем шаблоны, позволяющие полностью покрыть все требования этой коллекции и для каждого из шаблонов приводим пример формализованного требования из коллекции и описываем схему доказательства условий корректности для этого шаблона. Статистика распределения требований из коллекции по шаблонам выявляет наиболее востребованные шаблоны. Мы также провели анализ связанных работ.

Ключевые слова: дедуктивная верификация; темпоральные требования; шаблоны требований; управляющее программное обеспечение; процесс-ориентированное программирование

ИНФОРМАЦИЯ ОБ АВТОРАХ

Черненко, Иван Михайлович	ORCID iD: 0000-0001-7675-8449. E-mail: cheriv98@mail.ru Аспирант
Ануреев, Игорь Сергеевич (автор для корреспонденции)	ORCID iD: 0000-0001-9574-128X. E-mail: anureev@gmail.com Старший научный сотрудник, к.ф.-м.н.
Гаранина, Наталья Олеговна	ORCID iD: 0000-0001-9734-3808. E-mail: garanina@iis.nsk.su Старший научный сотрудник, к.ф.-м.н.

Финансирование: Госзадание ИАиЭ СО РАН, проект № 122031600173-8.

Для цитирования: I. M. Chernenko, I. S. Anureev, and N. O. Garanina, “Requirement patterns in deductive verification of poST Programs”, *Modeling and Analysis of Information Systems*, vol. 31, no. 1, pp. 6–31, 2024. DOI: 10.18255/1818-1015-2024-1-6-31.

Введение

Формальная верификация играет важную роль в разработке критического с точки зрения безопасности программного обеспечения, в частности, управляющего программного обеспечения. Дедуктивная верификация — один из методов формальной верификации, в котором конструкции верифицируемой программы (включая саму программу) и требования к ней формализуются в виде логических формул, а соответствие программы требованиям проверяется с помощью логического вывода.

Процесс-ориентированное программирование [1] — один из подходов к разработке управляющего программного обеспечения. Процесс-ориентированная программа определяется как последовательность процессов. Каждый процесс представляется набором именованных состояний, содержащих программный код, которые задают логику поведения процесса. Выполнение программы заключается в последовательном (в порядке вхождения в программу) исполнении этих процессов в их текущих состояниях на каждой итерации цикла управления. Процессы могут взаимодействовать через изменение состояний друг друга и через разделяемые переменные.

Распространенным классом требований к управляющему программному обеспечению являются темпоральные требования, которые описывают свойства поведения систем во времени. Разработанный ранее подход к дедуктивной верификации [2] процесс-ориентированных программ на языке Reflex [3], использующий для доказательства условий корректности систему Coq [4], позволяет работать с темпоральными требованиями. В этом подходе требования задаются как инварианты цикла управления — формулы над состояниями программы, которые должны быть истинны между итерациями цикла управления. При обычном определении состояния как отображения программных переменных и метапеременных, специфицирующих характеристики исполнения программы (процесс, исполняемый в данный момент, состояние этого процесса, исполняемое в данный момент, счетчики времени процессов и т. п.) в их значения, такие инварианты позволяют задавать только простые свойства безопасности. Чтобы иметь возможность описывать более сложные темпоральные требования, состояния были переопределены в работе [5] таким образом, чтобы хранить всю историю изменений значений переменных и метапеременных программы.

В этой работе эти требования были формализованы на языке DV-TRL, варианте типизированной логики предикатов первого порядка с набором интерпретированных типов и предикатных и функциональных символов. Особенность этого языка состоит в том, что интерпретация его символов и типов позволяет отражать специфические понятия систем управления в процесс-ориентированной парадигме. В рамках подхода был формализован набор требований для коллекции управляющих программ на языке Reflex и было обнаружено, что существенная часть требований принадлежит небольшому количеству классов. Были разработаны четыре шаблона требований, описывающие выявленные классы с помощью формул типизированной логики предикатов первого порядка.

В работе [6] подход был адаптирован к программам на языке poST [7] — процесс-ориентированном расширении языка ST стандарта МЭК 61131-3 [8]. Были разработаны аксиоматическая семантика этого языка и генератор условий корректности, основанный на ней, который порождал условия корректности в формате системы Isabelle/HOL [9]. В работе [10] этот подход был применен к коллекции управляющих программ на языке poST, для которых был формализован набор требований, при этом были перенесены и скорректированы шаблоны требований и схемы доказательства для условий корректности, порождаемых для этих шаблонов, с учетом перехода с Reflex и Coq на poST и Isabelle/HOL, и добавлен новый шаблон. В результате существенная часть требований из этого набора была покрыта предложенными шаблонами.

Эта статья расширяет исследование [10] за счет добавления новых шаблонов требований, позволяющих полностью покрыть требования к коллекции, и описания примеров требований и схем до-

казательства для каждого из шаблонов. Она имеет следующую структуру. В разделе 1 дается краткое введение в язык poST. В разделе 2 определяется язык DV-TRL. В разделе 3 приведена коллекция требований для различных систем управления. В разделе 4 определены шаблоны, выделяющие классы требований, для которых порождаемые условия корректности имеют общие схемы доказательства, а также описаны эти схемы доказательства. В разделе 5 приведена статистика по распределению требований из коллекции по шаблонам. Раздел 6 содержит анализ связанных работ. В заключении подводятся итоги и рассматриваются планы дальнейших исследований.

1. Введение в язык poST

В этом разделе дается краткое введение в процесс-ориентированный язык poST на примере программы *Controller* управления сушилкой для рук на процесс-ориентированном расширении poST языка ST (Листинг 1). Цель этого раздела — ввести терминологию, связанную с процесс-ориентированными программами в целом и с программами на языке poST в частности (входные и выходные переменные, таймеры, период активации, типы данных и т. д.), которая используется в следующих разделах.

Листинг 1: Программа управления сушилкой для рук. Hand dryer control program

```
PROGRAM Controller
  VAR_INPUT
    hands : BOOL;
  END_VAR

  VAR_OUTPUT
    dryer : BOOL;
  END_VAR

  PROCESS Ctrl
    STATE waiting
      IF hands  THEN
        dryer := TRUE;
        SET NEXT;
      ELSE
        dryer := FALSE;
      END_IF
    END_STATE
    STATE drying
      IF hands  THEN
        RESET TIMER;
      END_IF
      TIMEOUT T#1s THEN
        SET STATE waiting;
      END_TIMEOUT
    END_STATE
  END_PROCESS
END_PROGRAM

CONFIGURATION Conf
  RESOURCE Res1 ON TestCPU
    TASK T1 (INTERVAL := T#100ms, PRIORITY := 1);
    PROGRAM controller WITH T1 : Controller;
  END_RESOURCE
END_CONFIGURATION
```

В этой программе объявлены две логические переменные: входная (VAR_INPUT) переменная *hands*, показывающая наличие рук под сушилкой, и выходная (VAR_OUTPUT) переменная *dryer*, определяющая, включен ли тепловентилятор. Через входные переменные программа получает сигналы из окружения (в частности, от объекта управления). Через выходные переменные программа посылает управляющие сигналы окружению.

Программа состоит из одного процесса *Ctrl*, который может находиться в двух состояниях: *waiting* и *drying*. При запуске программы процесс запускается в своем первом (в текстовом порядке) состоянии. В данном случае это состояние *waiting*. В случае нескольких процессов в момент запуска программы запускается только первый процесс. Остальные находятся в специальном состоянии останова *STOP*. Существует два специальных состояния процесса: *STOP* и *ERROR* (состояние ошибки).

В состоянии *waiting* проверяется наличие рук. Если руки есть, тепловентилятор включается и процесс *Ctrl* переходит в состояние *drying*. Для перехода процесса в это состояние используется

оператор перехода к следующему (в текстовом порядке) состоянию SET NEXT. Если руки отсутствуют, тепловентилятор выключается.

В состоянии *drying* используется оператор таймаута TIMEOUT *t* THEN *s* END_TIMEOUT, чтобы выключить сушилку через одну секунду. Время задается константой *T#1s* типа *Time* языка ST. С каждым процессом связан (локальный) таймер, который отсчитывает время, которое процесс находился в текущем состоянии. Оператор таймаута срабатывает, когда время таймера достигает *t*. В этом случае, выполняется оператор *s*, являющийся обработчиком события таймаута. При срабатывании таймаута процесс *Ctrl* переходит в состояние *waiting*. Для этого в качестве обработчика используется оператор SET STATE. Таймер процесса сбрасывается при переходе процесса в другое состояние, или может быть сброшен явно оператором RESET TIMER. Так в состоянии *drying* таймер сбрасывается, если появляются руки, т. е. значение переменной *hands* меняется с *FALSE* на *TRUE*.

Также имеется (глобальный) таймер программы, который начинает отсчет в момент запуска программы.

Программа на языке poST исполняется циклически, выполняя за один шаг одну итерацию цикла управления. На каждой итерации происходит получение значений входных переменных из окружения, последовательное (в текстовом порядке) выполнение действий всех процессов программы в их текущих состояниях и передача значений выходных переменных окружению в качестве управляющих сигналов. Связь входных и выходных переменных с окружением задается конфигурацией (CONFIGURATION). Конфигурация определяет ресурсы (RESOURCE), связанные с конкретными устройствами (*TestCPU*), а также наборы задач (TASK) для них. Задача связана с выполняющей ее программой и имеет период активации (INTERVAL) и приоритет (PRIORITY) выполнения среди других задач. Период активации задачи задает время, в течение которого должна завершиться любая итерация выполнения программы, связанной с этой задачей. Для программы управления сушилкой для рук определена задача *T1* с периодом активации 100 миллисекунд.

Язык poST наследует систему типов языка ST, но для целей данной статьи нам достаточно знать, что эта система включает булевский тип *BOOL* с константами *TRUE* и *FALSE*, целочисленный тип *INT*, вещественный тип *REAL*, а также тип (статических) массивов с элементами любого из этих трех типов. Другие типы при дедуктивной верификации мы сводим к вышеупомянутым типам и типам языка описания требований DV-TRL (раздел 2), абстрагируясь от их особенностей. Например, тип *TIME* сводится к типу натуральных чисел *nat*.

2. Язык темпоральных требований DV-TRL

Язык темпоральных требований DV-TRL, введенный в [5], является вариантом типизированной логики предикатов первого порядка с набором интерпретированных типов и предикатных и функциональных символов, предназначенным для описания требований к процесс-ориентированным программам.

Он включает простые типы *bool*, *int*, *real*, *nat*, *variable*, *process* и *pstate*, а также тип данных *ustate* (update state), хранящий историю всех изменений значений переменных (как обычных программных переменных, так и метапеременных, специфицирующих состояние poST-программы).

Типы *bool*, *int*, *real* соответствуют типам *BOOL*, *INT* и *REAL* языка poST. Заметим, что мы различаем логические константы *true* и *false* логики предикатов и логические константы *TRUE* и *FALSE* типа *bool*.

Тип *nat* описывает множество натуральных чисел (с нулем).

Типы *variable*, *process* и *pstate* используются для кодирования имен переменных, процессов и состояний процессов в процесс-ориентированной программе, соответственно.

Тип данных *ustate* определяется следующим набором конструкторов (значений):

- *emptyState* : *ustate* соответствует начальному состоянию poST-программы, т. е. моменту ее запуска;

- *toEnv* : *ustate* → *ustate* соответствует завершению очередной итерации цикла управления, т. е. моменту передачи значений выходных переменных poST-программы окружению (в частности, объекту управления). В этот момент таймер глобального времени увеличивается на период активации — константу, задаваемую в конфигурации poST-программы;
- *setVarBool* : *ustate* × *variable* × *bool* → *ustate* соответствует присваиванию значения переменной типа *bool*;
- *setVarInt* : *ustate* × *variable* × *int* → *ustate* соответствует присваиванию значения переменной типа *int*;
- *setVarReal* : *ustate* × *variable* × *real* → *ustate* соответствует присваиванию значения переменной типа *real*;
- *setVarArrayBool* : *ustate* × *variable* × *int* × *bool* → *ustate* соответствует присваиванию значения типа *bool* элементу массива. Третий аргумент конструктора задает индекс элемента, которому присваивается значение;
- *setVarArrayInt* : *ustate* × *variable* × *int* × *int* → *ustate* соответствует присваиванию значения типа *int* элементу массива;
- *setVarArrayReal* : *ustate* × *variable* × *int* × *bool* → *ustate* соответствует присваиванию значения типа *real* элементу массива;
- *setPstate* : *ustate* × *process* × *pstate* → *ustate* соответствует изменению состояния процесса в состоянии управляющей программы. Третий аргумент конструктора задает новое состояние процесса;
- *reset* : *ustate* × *process* → *ustate* соответствует сбросу таймера процесса (присваиванию ему значения 0).

Таким образом, для каждого типа изменений в программе (присваивание значения программной переменной, изменение состояния процесса, изменение таймера процесса и т. п.) определен соответствующий конструктор.

Такое определение состояния изменений делает его похожим на матрешку. Каждое состояние изменений содержит в себе предыдущее состояние изменений и специфицирует через имя конструктора последнее изменение, которое привело к текущему состоянию. Самая маленькая неделимая матрешка соответствует состоянию изменений, порождаемому конструктором *emptyState*, т. е. моменту запуска программы. Например, состояние изменений программы управления сушилкой для рук (раздел 1):

setVarBool(*setVarBool*(
setPstate(*setVarBool*(*setVarBool*(*emptyState*(), *hands*, *FALSE*), *dryer*, *FALSE*), *Ctrl*, *waiting*),
hands, *TRUE*), *dryer*, *TRUE*),

хранит историю о том, что программа запустилась, логическим переменным *hands* и *dryer* были присвоены значения по умолчанию (после запуска программы рук нет и сушилка выключена), запустился процесс *Ctrl* в состоянии ожидания рук *waiting*, появились руки и включилась сушилка.

Заметим, что хотя язык DV-TRL не включает явно тип массивов языка poST, но массивы моделируются в нем конструкторами *setVarArrayBool*, *setVarArrayInt* и *setVarArrayReal* значений типа *ustate*.

Также следует отметить особенность моделирования локальных таймеров процессов и глобального таймера программы в формулах на этом языке. Время на нем отсчитывается в количестве итераций цикла управления, выполненных программой. Поэтому, при переходе от требований на естественном языке к их формализации на DV-TRL каждое вхождение реального времени t заменяется на $\lceil t/i \rceil$, где i — период активации программы, $\lceil . \rceil$ — операция округления к большему целому числу. Например, *T#1s* из программы управления сушилкой для рук (раздел 1) заменится на 10, так как период активации этой программы равен 100 миллисекундам.

Помимо стандартных операций для типов *bool*, *int* и *real* poST-программы, язык DV-TRL включает следующие интерпретированные предикатные и функциональные символы для работы с состояниями изменений:

- $getVarBool : ustate \times variable \to bool$ возвращает значение переменной типа *bool*;
- $getVarInt : ustate \times variable \to int$ возвращает значение переменной типа *int*;
- $getVarReal : ustate \times variable \to real$ возвращает значение переменной типа *real*;
- $getVarArrayBool : ustate \times variable \times int \to bool$ возвращает значение элемента массива типа *bool*. Третий аргумент функции задает индекс элемента, для которого возвращается значение;
- $getVarArrayInt : ustate \times variable \times int \to int$ возвращает значение элемента массива типа *int*;
- $getVarArrayReal : ustate \times variable \times int \to real$ возвращает значение элемента массива типа *real*;
- $getPstate : ustate \times process \to pstate$ возвращает текущее состояние процесса;
- $substate : ustate \times ustate \to bool$ является истинным тогда и только тогда, когда раскрывая второе состояние изменений как матрешку можно добраться до первого состояния изменений, т. е. $substate(s_1, s_2) = true$ в том случае, если $s_1 = s_2$, или существуют состояние s_3, конструктор c и значения v_1, ..., v_n такие, что $s_2 = c(s_3, v_1, \ldots, v_n)$, и $substate(s_1, s_3) = true$. В этом случае говорят, что s_1 является подсостоянием s_2;
- $toEnvNum : ustate \times ustate \to nat$ возвращает количество применений конструктора *toEnv*, необходимых для получения второго состояния изменений из первого, т. е. количество моментов передачи значений выходных переменных poST-программы окружению, которые случились между первым и вторым состоянием. Между применениями конструктора *toEnv* могут применяться другие конструкторы. Если первое состояние не является подсостоянием второго, то эта функция возвращает 0;
- $toEnvP : ustate \to bool$ проверяет, является ли состояние изменений результатом применения конструктора *toEnv*, т. е. находится ли программа в моменте передачи значений выходных переменных poST-программы окружению.
- $ltime : ustate \times process \to nat$ возвращает значение таймера процесса.

Заметим, что терм $toEnvNum(emptyState(), s)$ определяет время глобального таймера программы для ее текущего состояния s.

Эти функции и предикаты позволяют естественным образом описывать требования к процесс-ориентированным программам. Чтобы использовать эти функции и предикаты в доказательствах условий корректности, для них была разработана *общая теория состояний изменений* на языке Isabelle/HOL.

3. Коллекция требований

В этом разделе представлена коллекция требований к управляющим программам на языке poST для 5 устройств (турникета, светофора на пешеходном переходе, вращающихся дверей, холодильника и термопота). Сами программы можно найти на GitHub[1].

3.1. Турникет

Рассмотрим в качестве управляемого устройства турникет. Он оснащен монетоприемником, створками, управляемыми сигналом (*open*), светодиодом (*enter*), указывающим на возможность прохода, и двумя датчиками для обнаружения присутствия пользователя (*PdOut*) и контроля открытия створок (*opened*). Створки остаются закрытыми до тех пор, пока от монетоприемника не поступит сигнал оплаты (*paid*), после чего они открываются. Если пользователь не пройдет в течение 10 секунд после открытия турникета, он автоматически закроется. После успешной оплаты монетоприемник блокируется. Для простоты проверка успешной оплаты (*paid*) выполняется за пределами

[1] https://github.com/ivchernenko/extended_requirements_classification

управляющей программы, т. е. окружением. Монетоприемник разблокируется (*reset*) после закрытия турникета.

Мы формулируем следующие 8 требований к программе управления турникетом:

1. Турникет должен оставаться открытым не более чем 10,2 с.
2. Если турникет был закрыт и оплата не выполнена, то он не откроется, пока не будет выполнена оплата.
3. После получения оплаты турникет должен быть открыт не позднее, чем через 0,2 с.
4. После прохода пользователя турникет должен быть закрыт не позднее, чем через 1,2 с.
5. Турникет должен быть открыт не менее 1 с.
6. Светодиод должен включаться не позднее, чем через 0,2 с после открытия турникета и гореть до его закрытия.
7. После закрытия турникета не позднее, чем через 0,2 с разблокируется монетоприемник.
8. Если турникет только что открылся, он будет открыт в течение 10 с или пока не пройдет пользователь.
9. Если турникет только что открылся, он будет открыт в течение 9,9 секунд и остается открытым через 9,9 секунд, если за это время не пройдет пользователь.

3.2. Светофор на пешеходном переходе

Рассмотрим в качестве управляемого устройства пешеходный светофор с кнопкой, установленный на переходе. Его состояние по умолчанию — «красный». Когда у перехода появляется пешеход, он нажимает на кнопку. Если при этом зеленый сигнал светофора горел более 10 с назад, то зеленый сигнал включится через 5 с после нажатия кнопки. Если зеленый горел время t назад, которое меньше 10 с, то зеленый включится через $15 - t$ с после нажатия кнопки. Если включился зеленый, он будет гореть в течение 30 с, затем включается красный. Имеется один входной сигнал — «кнопка нажата» и один сигнал управления — «горит зеленый» Программа принимает входной сигнал и в зависимости от него управляет сигналом светофора.

Мы сформулировали следующие требования к программе управления светофором:

1. Если горел красный свет и кнопку нажали, то не позднее чем через *Tw* с загорится зеленый свет.
2. Если только что загорелся зеленый, то зеленый будет гореть не менее *Tg_min* с.
3. Если только что загорелся зеленый, то за время не более чем *Tg_max* с он переключится на красный.
4. Если загорелся красный, то красный будет гореть не менее чем *Tr* с.
5. Если только что включился красный, то он будет гореть, пока не нажмут на кнопку.

Здесь *Tr* — минимальное время горения красного света, *Tw* — максимальное время, в течение которого пешеход ожидает включения зеленого сигнала светофора после нажатия на кнопку в момент, когда горит красный, равное 15 с, *Tg_min* — минимальная продолжительность зеленого сигнала светофора, равная 30 с, *Tg_max* — максимальная продолжительность зеленого сигнала светофора, на 0,2 с большая, чем *Tg_min*. Значение *Tr* на 0,2 с меньше, чем *Tw*.

3.3. Вращающаяся дверь

Рассмотрим в качестве управляемого устройства вращающуюся дверь. Она состоит из трех- или четырехсекционной двери, вращающейся вокруг вертикальной оси, привода (*rotation*) и тормоза (*brake*) для остановки двери. При отсутствии пользователей дверь неподвижна, а при приближении пользователя начинает вращение. Вращение продолжается, пока пользователь находится внутри пространства вращения. Приближение пользователя и его присутствие внутри пространства вращения регистрирует датчик движения (*user*). Если пользователь покидает пространство вращения, то после определенного времени вращение останавливается. Датчик давления регистри-

рует давление на секционные перегородки. Вращение приостанавливается на небольшое время, когда на перегородки оказывается давление.

Мы сформулировали следующие требования к программе управления вращающейся дверью:

1. При входе пользователя дверь начинает вращаться не позднее, чем через 0,2 с, если на перегородки не оказывается давление.
2. Вращение продолжается, пока пользователь находится внутри пространства вращения, если на перегородки не оказывается давление.
3. Если пользователь покинул пространство вращения, то не позднее, чем через 1 с вращение остановится, если за это время пользователи не появятся вновь.
4. Если на секционные перегородки оказывается давление, то не позднее, чем через 0,3 с вращение приостанавливается не менее, чем на 1 с.
5. Если на секционные перегородки перестали оказывать давление, то не позднее, чем через 1,2 с вращение возобновится.
6. Привод и тормоз не могут работать одновременно.

3.4. Холодильник

Рассмотрим в качестве управляемого устройства холодильник. Он состоит из холодильной и морозильной камер и имеет два компрессора. Температура в холодильной камере регистрируется датчиками *fridgeTempGreaterMin* и *fridgeTempGreaterMax*, показывающими, выходит ли температура холодильника за рамки минимального и максимального значений, соответственно. Для контроля температуры в морозильной камере устройство имеет датчики *freezerTempGreaterMin* и *freezerTempGreaterMax*. Холодильник поддерживает температуру в диапазоне между минимальным и максимальным значениями. При превышении температуры в холодильной камере включается компрессор (*fridgeCompressor*), который выключается, когда температура достигает минимального значения. Для морозильной камеры используется компрессор *freezerCompressor*. При открытии двери холодильной камеры включается освещение (*lighting*), которое выключается при ее закрытии. Если дверь холодильной камеры открыта более 30 с, подается звуковой сигнал (*doorSignal*).

Для данной системы мы сформулировали следующие требования:

1. При открытии двери холодильника не позднее чем через 0,2 с включается освещение.
2. При закрытии двери холодильника не позднее чем через 0,2 с выключается освещение.
3. Если дверь холодильника открыта, то не позднее чем через 30 с подается сигнал, если за это время пользователь не закроет дверь.
4. Звуковой сигнал не подается произвольно. Сигнал подается, только если дверь открыта в течение не менее чем 30 с.
5. Если дверь закрыта и температура в холодильнике превышает максимальную, то не позднее чем через 0,2 с включается компрессор.

3.5. Термопот

Рассмотрим в качестве управляемого устройства термопот — устройство, объединяюшее функции чайника и термоса. Термопот поддерживает три температурных режима, подогревает воду до температуры, соответствующей выбранному температурному режиму (*selectedTemp*), и поддерживает данную температуру. Устройство содержит корпус с герметичной колбой, которая позволяет длительное время поддерживать требуемую температуру, крышку и нагревательный элемент (*heater*). На крышке расположена панель управления с тремя кнопками (*button1*, *button2*, *button3*), позволяющими выбрать требуемый температурный режим. Для включения кипячения используется кнопка кипячения (*boilingButton*). Во время кипячения крышка блокируется (*lid*). После кипячения термопот переходит в режим поддержания температуры. В режиме поддержания температуры нагревательный элемент включается, когда температура воды понижается более, чем на 5 градусов

ниже заданной. Индикаторы *boilingMode* и *maintainingMode* показывают, находится ли термопот в режиме кипячения и поддержания температуры, соответственно.

Мы формулируем следующие требования к программе управления термопотом:

1. Пока термопот находится в режиме кипячения и требуемая температура не достигнута, крышка заблокирована.
2. В режиме поддержания при достижении заданной температуры если кнопка кипячения не нажата, то нагревательный элемент отключается не позднее, чем через 0,2 с.
3. В режиме поддержания температуры если кнопка кипячения не нажата, то нагревательный элемент включается не позднее, чем через 0,2 с после обнаружения понижения температуры воды более, чем на 5 градусов ниже заданной.
4. При нажатии одной из кнопок выбора температурного режима выбирается соответствующая температура.
5. Если термопот не находится ни в режиме поддержания температуры, ни в режиме кипячения, и кнопка кипячения не нажата, то нагрев не включится.

Заметим, что в этом разделе не стоит задача составить полный список требований для рассмотренных устройств, так как полнота в этом случае является эмпирической характеристикой, однако мы рассматриваем образцы требований, часто встречающиеся на практике, в том числе и в более сложных устройствах. Кроме того, целью статьи является не охват всех требований, а рассмотрение образцов требований, для которых порождаемые условия корректности имеют схемы доказательства, что позволит в дальнейшем автоматизировать дедуктивную верификацию таких требований.

4. Шаблоны требований

Для доказательства условий корректности, порождаемых из poST-программ для требований, формализованных на языке DV-TRL, мы используем систему интерактивного доказательства теорем Isabelle/HOL. Было замечено, что разные классы требований на языке DV-TRL требуют применения разных схем доказательств. В [5] было предложено задавать эти классы требований с помощью параметрических формул языка DV-TRL, называемых шаблонами требований, а в [10] выделены 5 таких шаблонов, примененных к коллекции требований раздела 3. В этом разделе мы описываем схемы доказательств для этих пяти и четырех новых шаблонов требований, а также примеры требований из коллекции раздела 3, удовлетворяющих данным шаблонам.

4.1. Шаблон *P1*

Шаблон $P1(s, t, A_1, A_2, A_3)$ определяет класс требований, которые утверждают, что не позднее, чем через время t после события A_1 должно произойти событие A_2, причем от момента наступления события A_1 и до наступления события A_2 должно выполняться (инвариантное) свойство A_3. В этом и следующих шаблонах события и свойства идентифицируются соответствующими формулами A_1, A_2 и т. д. Заметим, что события A_i обычно представляются булевскими комбинациями равенств и неравенств над значениями программных переменных. Шаблон *P1* имеет следующий вид:

$$
\begin{aligned}
&\mathit{toEnvP}(s) \wedge \\
&\forall s_1 (\mathit{substate}(s_1, s) \wedge \mathit{toEnvP}(s_1) \wedge \mathit{toEnvNum}(s_1, s) \geqslant t \wedge A_1(s_1) \longrightarrow \\
&\quad \exists s_3 (\mathit{toEnvP}(s_3) \wedge \mathit{substate}(s_1, s_3) \wedge \mathit{substate}(s_3, s) \wedge \mathit{toEnvNum}(s_1, s_3) \leqslant t \wedge A_2(s_3)) \wedge \\
&\quad \forall s_2 (\mathit{toEnvP}(s_2) \wedge \mathit{substate}(s_1, s_2) \wedge \mathit{substate}(s_2, s_3) \wedge s_2 \neq s_3 \longrightarrow A_3(s_2))).
\end{aligned}
$$

Примером, удовлетворяющим этому шаблону, является первое требование к программе управления турникетом: *«Турникет должен оставаться открытым не более чем 10,2 с»* (см. раздел 3.1).

Для этого требования имеем:

$$t \equiv 101;$$
$$A_1 \equiv getVarBool(s_1, open) = TRUE;$$
$$A_2 \equiv getVarBool(s_3, open) = FALSE;$$
$$A_3 \equiv getVarBool(s_2, open) = TRUE.$$

При формализации требований явно описываются входные и выходные переменные программы. Это может привести к сдвигу по времени в формальном описании требования по сравнению с его описанием на естественном языке в силу следующих причин. Во-первых, необходимо учитывать, что события могут произойти не сразу после подачи управляющих сигналов, связанных с выходными переменными, в силу инертности физических процессов. Во-вторых, так как формальные требования задаются как инварианты цикла управления, выполняющиеся при каждом завершении итерации цикла управления, значения входных переменных проверяется в момент завершения итерации, тогда как в требованиях на естественном языке они проверяются в момент начала итерации. Следовательно, время t в формальном требовании определяется как $t' + \Delta t$, где t' — время в требовании на естественном языке. Напомним также, что при переходе от требований на естественном языке к их формализации на DV-TRL каждое вхождение реального времени t' заменяется на $\lceil t'/i \rceil$, где i — период активации программы, $\lceil . \rceil$ — операция округления к большему целому числу. В данном случае, турникет закрывается в течение 100 мс после прекращения подачи сигнала *open*. Поэтому $\Delta t = -100$ мс и $t = 10.1$ с. = 10 100 мс. Требование проверяется для программы с $i = 100$ мс. Тогда $t = \lceil 10100/100 \rceil = 101$.

На языке системы Isabelle/HOL схема доказательства условий корректности, порождаемых для требований этого класса, имеет следующий вид:

Листинг 2: Схема доказательства для шаблона 1. Proof scheme for the pattern 1

```
theorem req_proof: "VC inv s0 input_values"
apply(unfold VC_def inv_def req_def)
apply(rule impI)
apply(subgoal_tac "extraInv (s s0 input_values)")
 apply(rule conjI)
  apply(rule conjI)
   apply simp
  apply(erule conjE)
  apply(unfold extraInv_def)[1]
subgoal premises vc_prems
  apply(insert vc_prems(1))
  apply((erule conjE)+)
  subgoal premises ei
    apply(rule L1[OF ei(n)])
    using vc_prems(3) by simp
  done
using extraInv_proof by(auto simp add: VC_def)
```

Здесь *VC* обозначает доказываемое условие корректности. Условия корректности параметризованы инвариантом цикла управления *inv*, состоянием изменений в начале итерации цикла *s0* и значениями входных переменных программы *input_values*, получаемых из окружения. Требование, для которого доказывается условие корректности, является частью инварианта цикла управления. Однако для доказательства условий корректности необходимы вспомогательные утверждения,

в частности, информация о связи используемых в требованиях значений входных и выходных переменных программы с состояниями процессов, значениями локальных переменных и локальных таймеров, которые обычно не используются в требованиях, так как являются деталями реализации. Поэтому инвариант цикла управления является конъюнкцией формального описания требования *req* и дополнительного инварианта *extraInv*, содержащего такую информацию.

В доказательстве сначала раскрываются определения условия корректности, инварианта и требования. Затем доказательство сводится к доказательству требования в предположении, что дополнительный инвариант истинен, и доказательству дополнительного инварианта. Далее расщепляется конъюнкция в заключении условия корректности. Часть инварианта цикла, соответствующая требованию, доказывается следующим образом. Первый конъюнкт в требовании $toEnvP(s)$ доказывается автоматически с помощью *simp*. Далее расщепляется конъюнкция в посылке условия корректности и раскрывается определение дополнительного инварианта. Затем расщепляется конъюнкция в дополнительном инварианте. В результате получаем последовательность конъюнктов *ei*. Для завершения доказательства используются лемма *L1*, утверждение $ei(n)$ (n-й элемент последовательности *ei*) из дополнительного инварианта и утверждение $vc\_prems(3)$ из посылки условия корректности *vc_prems*. Часть инварианта цикла, соответствующая дополнительному инварианту, доказывается с помощью отдельной леммы *extraInv_proof*[2]. Эта лемма доказывает условие корректности, используя в качестве инварианта цикла управления только дополнительный инвариант. Ее вид и схема доказательства порождаются отдельно для каждого условия корректности.

Лемма *L1* зависит от параметров s, t, A_1, A_2, A_3 шаблона и от параметров-функций P и t_1 дополнительного инварианта, где $t_1(s')$ — это время, до истечения которого случилось A_2 и которое произошло до перехода в состояние s', и P — это свойство, которое выполняется в состоянии s'. Она имеет вид:

$$
\begin{aligned}
&(\forall s_4(toEnvP(s_4) \land substate(s_4, s) \land P(s_4) \longrightarrow \\
&\quad \forall s_1(toEnvP(s_1) \land substate(s_1, s_4) \land A_1(s_1) \longrightarrow \\
&\qquad \exists s_3(toEnvP(s_3) \land substate(s_1, s_3) \land substate(s_3, s_4) \land toEnvNum(s_1, s_3) \leqslant t \land A_2(s_3) \land \\
&\qquad\quad \forall s_2(toEnvP(s_2) \land substate(s_1, s_2) \land substate(s_2, s_3) \land s_2 \neq s_3 \longrightarrow A_3(s_2))) \lor \\
&\qquad toEnvNum(s_1, s_4) < t_1(s_4) \land (\forall s2.toEnvP(s_2) \land substate(s_1, s_2) \land substate(s_2, s_4) \longrightarrow A_3(s_2))))) \longrightarrow \\
&toEnvP(s) \land P(s) \land t_1(s) \leqslant t \longrightarrow \\
&(\forall s_1(substate(s_1, s) \land toEnvP(s_1) \land toEnvNum(s_1, s) \geqslant t \land A_1(s_1) \longrightarrow \\
&\quad \exists s_3(toEnvP(s_3) \land substate(s_1, s_3) \land substate(s_3, s) \land toEnvNum(s_1, s_3) \leqslant t \land A_2(s_3) \land \\
&\qquad \forall s_2(toEnvP(s_2) \land substate(s_1, s_2) \land substate(s_2, s_3) \land s_2 \neq s_3 \longrightarrow A_3(s_2))))).
\end{aligned}
$$

4.2. Шаблон *P2*

Шаблон $P2(s, A_1, A_2)$ определяет класс требований, которые утверждают, что если после двух последовательных итераций цикла управления в момент завершения второй итерации произошло событие A_1, связывающее значения переменных в моментах завершения этих двух итераций, то в этот же момент (момент завершения второй итерации) должно произойти событие A_2. Так как значения входных переменных на выходе из итерации цикла управления совпадают со значениями, поступающими на контроллер из окружения на этой итерации (значения входных переменных не меняются контроллером), то, как правило, A_1 определяет связь значений входных и выходных переменных на итерации цикла управления. Шаблон *P2* имеет следующий вид:

$$
\begin{aligned}
&toEnvP(s) \land \\
&\forall s_1 \forall s_2(substate(s_1, s_2) \land substate(s_2, s) \land toEnvP(s_1) \land \\
&\quad toEnvP(s_2) \land toEnvNum(s_1, s_2) = 1 \land A_1(s_1, s_2) \longrightarrow A_2(s_2)).
\end{aligned}
$$

[2] https://github.com/ivchernenko/extended_requirements_classification

Примером, удовлетворяющим этому шаблону, является первое требование к программе управления термопотом: *«Пока термопот находится в режиме кипячения и требуемая температура не достигнута, крышка заблокирована»* (см. раздел 3.5). Для этого требования имеем:

$$A_1 \equiv getVarBool(s_1, boilingMode) = TRUE \wedge getVarInt(s_2, temperature) < BOILING\_POINT;$$
$$A_2 \equiv getVarBool(s_2, lid) = LOCKED.$$

В некоторых случаях условия корректности для требований второго класса могут быть доказаны автоматически методом *auto*. Если доказательство не требует дополнительного инварианта, но не может быть выполнено автоматически с помощью *auto*, то применяется следующая схема доказательства (Листинг 3):

Листинг 3: Схема доказательства для шаблона 2. Proof scheme for the pattern 2

```
theorem req_proof: "VC inv env s0 input_values"
 apply(unfold VC_def inv_def Req_def)
 apply(rule impI)
 apply(rule conjI)
  apply(rule conjI)
   apply simp
  apply((rule allI)+)
  apply(rule impI)
  apply(simp split: if_splits)
 using substate_toEnvNum_id apply blast
 apply auto[1]
using extraInv_proof by (auto simp add: VC_def)
```

В доказательстве сначала раскрываются определения условия корректности, инварианта и требования. Определение дополнительного инварианта не раскрывается, так как он не используется для доказательства того, что требование выполняется в состоянии s. Далее расщепляется конъюнкция в заключении условия корректности. Первый конъюнкт $toEnvP(s)$ доказывается автоматически с помощью метода *simp*. Затем применяются правила для кванторов всеобщности и импликации и выполняется разбор случаев $s_2 = s$ и $s_2 \neq s$ с упрощением полученных подцелей с помощью *simp split: if_splits*. Случай $s_2 = s$ доказывается методом *blast* с помощью леммы *substate_toEnvNum_id* из общей теории состояний изменений. Случай $s_2 \neq s$ доказывается методом *auto*. Затем применяется лемма *extraInv_proof* (см. 4.1).

Если для доказательства необходим дополнительный инвариант, применяется другая схема доказательства (Листинг 4):

Листинг 4: Схема доказательства для шаблона 2 с использованием дополнительного инварианта. Proof scheme for the pattern 2 using additional invariant

```
 theorem req_proof: "VC inv6 env s0 input_values"
 apply(unfold VC_def loopinv_def Req_def)
   apply(rule impI)
  apply(rule conjI)
   apply(rule conjI)
    apply simp
   apply(erule conjE)
   apply(erule conjE)
   apply(rotate_tac)
```

```
  apply(erule conjE)
 apply(unfold extraInv_def)[1]
  subgoal premises vc_prems
  apply(rule allI)+
  apply(rule impI)
   apply(simp split: if_splits)
   using vc_prems(2)[simplified] vc_prems(4)
  sledgehammer
  using vc_prems(3) by auto
using extraInv_proof by (auto simp add: VC_def)
```

В данной схеме *sledgehammer* обозначает скрипт доказательства, полученный применением команды *sledgehammer*. Сначала раскрывается определение условия корректности методом *unfold* и выполняется его упрощение методом *simp*. Затем раскрываются определения инварианта и требования и расщепляется конъюнкция в посылке условия корректности. Далее расщепляется конъюнкция в заключении условия корректности. Первый конъюнкт $toEnvP(s)$ доказывается автоматически с помощью метода *simp*. После этого необходимо доказать две подцели, первая соответствует требованию, а вторая — дополнительному инварианту. В первой подцели раскрывается определение дополнительного инварианта. Затем команда *subgoal* задает имя посылкам условия корректности. Далее применяются правила для квантора всеобщности и импликации и выполняется разбор случаев с упрощение полученных подцелей с помощью команды *simp split: if_splits*. Первая подцель соответствует случаю $s_2 = s$ и доказывается с использованием посылки условия корректности, соответствующей условиям в программе ($prems(1)$) и дополнительного инварианта ($prems(3)$), а также *sledgehammer*. Доказательство для случая $s_2 \neq s$ следует из того, что требование $prems(2)$ было истинно перед выполнением текущей итерации цикла управления.

4.3. Шаблон *P3*

Шаблон $P3(s, A_1, A_2)$ определяет класс требований, которые утверждают, что события A_1 и A_2 происходят в один и тот же момент времени, соответствующий завершению итерации цикла управления (моменту передачи значений выходных переменных poST-программы окружению). Он имеет следующий вид:

$$toEnvP(s) \wedge \forall s_1(substate(s_1, s) \wedge toEnvP(s_1) \wedge A_1(s_1) \longrightarrow A_2(s_1)).$$

Заметим, что хотя может показаться, что этот шаблон описывает нетемпоральные требования вида $A_1(s_1) \longrightarrow A_2(s_1)$, на самом деле это не так, так как импликация выполняется только для определенных моментов времени, а именно для моментов передачи значений выходных переменных poST-программы окружению. Внутри цикла управления импликация может быть ложной.

Примером, удовлетворяющим данному шаблону, является шестое требование к программе управления вращающейся дверью: *«Привод и тормоз не могут работать одновременно»* (см. раздел 3.3). Для этого требования имеем:

$$A_1 \equiv getVarBool(s_1, brake) = TRUE;$$
$$A_2 \equiv getVarBool(s_2, rotation) = FALSE.$$

В некоторых случаях условия корректности для требований этого класса могут быть доказаны автоматически с помощью метода *auto*. В остальных случаях для доказательства необходимо использовать дополнительный инвариант и доказательство выполняется в соответствии со схемой, приведенной на листинге 4 для требований второго класса.

4.4. Шаблон *P4*

Шаблон $P4(s, t, A_1, A_2, A_3)$ определяет класс требований, которые утверждают, что если после двух последовательных итераций цикла управления в момент завершения второй итерации произошло событие A_1, связывающее значения переменных в моментах завершения этих двух итераций, то от этого момента (момента завершения второй итерации), не позднее, чем через время t должно произойти событие A_2, причем от момента наступления события A_1 и до наступления события A_2 должно выполняться (инвариантное) свойство A_3. Он имеет следующий вид:

$$
\begin{aligned}
&toEnvP(s) \wedge \\
&\forall s_1 \forall s_2 (substate(s_1, s_2) \wedge substate(s_2, s) \wedge toEnvP(s_1) \wedge toEnvP(s_2) \wedge toEnvNum(s_1, s_2) = 1 \wedge \\
&\quad toEnvNum(s_2, s) \geqslant t \wedge A_1(s_1, s_2) \longrightarrow \\
&\qquad \exists s_4 (toEnvP(s_4) \wedge substate(s_2, s_4) \wedge substate(s_4, s) \wedge toEnvNum(s_2, s_4) \leqslant t \wedge A_2(s_4)) \wedge \\
&\qquad \forall s_3 (toEnvP(s_3) \wedge substate(s_2, s_3) \wedge substate(s_3, s_4) \wedge s_3 \neq s_4 \longrightarrow A_3(s_3))),
\end{aligned}
$$

Примером, удовлетворяющим этому шаблону, является первое требование к программе управления светофором: *«Если горел красный свет и кнопку нажали, то не позднее, чем через Tw с загорится зеленый свет»* (см. раздел 3.2). Для этого требования имеем:

$$
\begin{aligned}
&t \equiv Tr; \\
&A_1 \equiv getVarBool(s_1, trafficLight) = RED \wedge getVarBool(s_1, requestButton) = NOT\_PRESSED \wedge \\
&\quad getVarBool(s_2, requestButton) = PRESSED; \\
&A_2 \equiv getVarBool(s_4, trafficLight) = GREEN; \\
&A_3 \equiv getVarBool(s_3, trafficLight) = RED.
\end{aligned}
$$

Доказательство условий корректности для требований, соответствующих четвертому классу выполняется аналогично доказательствам для требований первого класса, но лемма *L1* имеет другой вид:

$$
\begin{aligned}
&(\forall s_5 (toEnvP(s_5) \wedge substate(s_5, s) \wedge P(s_5) \longrightarrow \\
&\quad \forall s_1 \forall s_2 (toEnvP(s_1) \wedge toEnvP(s_2) \wedge substate(s_1, s_2) \wedge substate(s_2, s_5) \wedge toEnvNum(s_1, s_2) = 1 \wedge A_1(s_1, s_2) \longrightarrow \\
&\qquad \exists s_4 (toEnvP(s_4) \wedge substate(s_2, s_4) \wedge substate(s_4, s_5) \wedge toEnvNum(s_2, s_4) \leqslant t \wedge A_2(s_4) \wedge \\
&\qquad\quad \forall s_3 (toEnvP(s_3) \wedge substate(s_2, s_3) \wedge substate(s_3, s_4) \wedge s_3 \neq s_4 \longrightarrow A_3(s_3))) \vee \\
&\qquad toEnvNum(s_2, s_5) < t_1(s_5) \wedge \forall s_3 (toEnvP(s_3) \wedge substate(s_2, s_3) \wedge substate(s_3, s_5) \longrightarrow A_3(s_3))))) \longrightarrow \\
&(toEnvP(s) \wedge P(s) \wedge t_1(s) \leqslant t) \longrightarrow \\
&\forall s_1 \forall s_2 (substate(s_1, s_2) \wedge substate(s_2, s) \wedge toEnvP(s_1) \wedge toEnvP(s_2) \wedge toEnvNum(s_1, s_2) = 1 \wedge \\
&\quad toEnvNum(s_2, s) \geqslant t \wedge A_1(s_1, s_2) \longrightarrow \\
&\qquad \exists s_4 (toEnvP(s_4) \wedge substate(s_2, s_4) \wedge substate(s_4, s) \wedge toEnvNum(s_2, s_4) \leqslant t \wedge A_2(s_4) \wedge \\
&\qquad\quad \forall s_3 (toEnvP(s_3) \wedge substate(s_2, s_3) \wedge substate(s_3, s_4) \wedge s_3 \neq s_4 \longrightarrow A_3(s_3)))).
\end{aligned}
$$

4.5. Шаблон *P5*

Шаблон $P5(s, t, A_1, A_2)$ определяет класс требований, которые утверждают, что если после двух последовательных итераций цикла управления в момент завершения второй итерации произошло событие A_1, связывающее значения переменных в моментах завершения этих двух итераций, то от этого момента (момента завершения второй итерации) событие A_2 должно выполняться, по крайней мере, до того, как будет достигнуто время t. Он имеет следующий вид:

$$
\begin{aligned}
&toEnvP(s) \wedge \\
&\forall s_1 \forall s_2 \forall s_3 (substate(s_1, s_2) \wedge substate(s_2, s_3) \wedge substate(s_3, s) \wedge toEnvP(s_1) \wedge toEnvP(s_2) \wedge \\
&\quad toEnvP(s_3) \wedge toEnvNum(s_1, s_2) = 1 \wedge toEnvNum(s_2, s_3) < t \wedge A_1(s_1, s_2) \longrightarrow A_2(s_3)).
\end{aligned}
$$

Примером, удовлетворяющим этому шаблону, является четвертое требование к программе управления холодильником: *«Звуковой сигнал не подается произвольно. Сигнал подается только если дверь открыта в течение не менее, чем 30 с»* (см. раздел 3.4). Для этого требования имеем:

$$t \equiv OPEN\_DOOR\_TIME\_LIMIT;$$
$$A_1 \equiv getVarBool(s_1, fridgeDoor) = CLOSED' \wedge getVarBool(s_2, fridgeDoor) = OPEN;$$
$$A_2 \equiv getVarBool(s_3, doorSignal) = FALSE.$$

Заметим, что в этом случае, прежде чем описывать требование формально, мы переформулируем его следующим образом: *«Звуковой сигнал не подается произвольно. Сигнал не подается, если дверь открыта менее, чем 30 с»*.

В некоторых случаях доказательство условий корректности для требований этого класса может быть выполнено автоматически с помощью метода *auto*. Рассмотрим схему доказательства для случая, когда доказательство может быть выполнено без использования дополнительного инварианта, но не может быть выполнено методом *auto* (Листинг 5):

Листинг 5: Схема доказательства для шаблона 5. Proof scheme for the pattern 5

```
theorem req_proof: "VC inv env s0 input_values"
 apply(unfold VC_def)
 apply simp
 apply(unfold inv_def req_def)
 apply(rule impI)
 apply(rule conjI)
 apply(rule conjI)
  apply simp
 subgoal premises vc_prems
  apply((rule allI)+)
  apply(rule impI)
  apply(simp split: if_splits del: One_nat_def)
  subgoal for s1 s2
   apply(rule cut_rl[of "toEnvNum s2 s0 < t"])
   using vc_prems substate_refl apply blast
   by simp
  using vc_prems by blast
using extraInv_proof by (auto simp add: VC_def)
```

Начало доказательства выполняется аналогично доказательству для класса 2 (Листинг 3). Здесь команда *apply(simp split: if_splits)* доказывает случай 1. Поэтому после ее применения необходимо доказать только случаи 2 и 3. Для доказательства случая 2 задается промежуточное утверждение $toEnvNum(s_2, s_0) < t$. С помощью этого утверждения доказательство выполняется методом *blast*. Лемма *substate_refl* из теории состояний изменений используется для вывода утверждения $substate(s_0, s_0)$, необходимого для доказательства данного случая. Промежуточное утверждение доказано методом *simp*. Случай 3 доказывается методом *blast*.

4.6. Шаблон *P6*

Шаблон $P6(s, A_1, A_2, A_3)$ определяет класс требований, которые утверждают, что если после двух последовательных итераций цикла управления в момент завершения второй итерации произошло событие A_1, связывающее значения переменных в моментах завершения этих двух итераций, то от этого момента (момента завершения второй итерации) условие A_2 будет выполняться, пока

не произойдет событие A_3 (или всегда, если A_3 никогда не произойдет). Он имеет следующий вид:

$$\begin{aligned}
&toEnvP(s) \wedge \\
&\forall s_1 \forall s_2 \forall s_3 (substate(s_1, s_2) \wedge substate(s_2, s_3) \wedge substate(s_3, s) \wedge toEnvP(s_1) \wedge \\
&\quad toEnvP(s_2) \wedge toEnvP(s_3) \wedge toEnvNum(s_1, s_2) = 1 \wedge A_1(s_1, s_2) \wedge \\
&\quad \forall s_4 (toEnvP(s_4) \wedge substate(s_2, s_4) \wedge substate(s_4, s_3) \longrightarrow \neg A_3(s_4)) \longrightarrow \\
&\qquad A_2(s_3)).
\end{aligned}$$

Примером, удовлетворяющим этому шаблону, является пятое требование к программе управления светофором: *«Если только что включился красный, то он будет гореть пока не нажмут на кнопку»* (см. раздел 3.2). Для этого требования имеем:

$$\begin{aligned}
&A_1 \equiv getVarBool(s_1, trafficLight) \neq RED \wedge getVarBool(s_2, trafficLight) = RED; \\
&A_2 \equiv getVarBool(s_3, trafficLight) = RED; \\
&A_3 \equiv getVarBool(s_4, requestButton) = NOT_PRESSED.
\end{aligned}$$

Рассмотрим схему доказательства условий корректности для требований шестого класса для случая, когда не требуется использование дополнительного инварианта.

Листинг 6: Схема доказательства для шаблона 6. Proof scheme for the pattern 6

```
theorem req_proof: "VC inv s0 input_values"
apply(unfold VC_def inv_def req_def)
apply(rule impI)
apply(rule conjI)
apply(rule conjI)
 apply simp
apply(rule allI)+
subgoal  for s1 s2 s3
  apply(cases "s3 = s s0 input_values")
   apply(erule conjE)+
   apply(erule allE[of _ s1])
   apply(erule allE[of _ s2])
  apply(erule allE[of _ s0])
  apply(rule impI)
  apply(erule impE)
  using substate_refl apply (simp split: if_splits)
  using substate_toEnvNum_id apply blast
   apply simp
   apply(erule conjE)+
   apply(erule allE[of _ s1])
   apply(erule allE[of _ s2])
   apply(erule allE[of _ s3])
  by simp
  using extraInv_proof by(auto simp add: VC_def)
```

В доказательстве сначала раскрываются определения условия корректности, инварианта и требования. Затем расщепляется конъюнкция в заключении условия корректности. Сначала доказывается требование. Первый конъюнкт $toEnvP(s)$ требования доказывается методом *simp*. В доказательстве второго конъюнкта выполняется разбор случаев $s_3 = s$ и $s_3 \neq s$. Для доказательства случая $s_3 = s$ связанные квантором всеобщности переменные s_1, s_2 и s_3 в формуле, утверждающей, что требование

выполняется в состоянии s_0, сопоставляются с s_1, s_2 и s_0, соответственно. Далее доказывается посылка импликации в этой формуле и ее заключение используется для доказательства истинности требования в состоянии изменений s. Получаем две подцели. Первая подцель упрощается с помощью *simp* с использованием правила расщепления *if*-выражений *if_splits*, и далее применяется метод *blast* для логики первого порядка. Вторая подцель доказывается с помощью метода *simp*. Для доказательства случая $s_3 \neq s$ связанные квантором всеобщности переменные s_1, s_2 и s_3 в формуле, утверждающей, что требование выполняется в состоянии s_0, сопоставляются с s_1, s_2 и s_3, соответственно, и далее доказательство выполняется методом *simp*. Затем применяется лемма *extraInv_proof* для доказательства дополнительного инварианта.

4.7. Шаблон *P7*

Шаблон $P7(s, t, A_1, A_2, A_3)$ определяет требования, утверждающие, что если после двух последовательных итераций цикла управления в момент завершения второй итерации произошло событие A_1, связывающее значения переменных в моментах завершения этих двух итераций, то от этого момента (момента завершения второй итерации) условие A_2 должно выполняться как минимум в течение времени t или, пока не произойдет событие A_3. Формула, описывающая событие A_1 связывает значения переменных в двух состояниях изменений, время между которыми составляет одну итерацию цикла управления. Этот шаблон имеет следующий вид:

$$
\begin{aligned}
&toEnvP(s) \wedge \\
&\forall s_1 \forall s_2 (substate(s_1, s_2) \wedge substate(s_2, s) \wedge toEnvP(s_1) \wedge toEnvP(s_2) \wedge toEnvNum(s_1, s_2) = 1 \wedge A_1(s_1, s_2) \longrightarrow \\
&\quad \exists s_4 (toEnvP(s_4) \wedge substate(s_2, s_4) \wedge substate(s_4, s) \wedge toEnvNum(s_1, s_2) \leqslant t \wedge A_3(s_4)) \wedge \\
&\qquad \forall s_3 (toEnvP(s_3) \wedge substate(s_2, s_3) \wedge substate(s_3, s_4) \wedge s_3 \neq s_4 \longrightarrow A_2(s_3))) \vee \\
&\quad \forall s_3 (toEnvP(s_3) \wedge substate(s_2, s_3) \wedge substate(s_3, s) \wedge toEnvNum(s_2, s_3) \leqslant t \longrightarrow A_2(s_3))).
\end{aligned}
$$

Примером, удовлетворящим данному шаблону, является девятое требование к программе управления турникетом, которое может быть сформулировано следующим образом: *«Если турникет только что открылся, он будет открыт в течение 9,9 секунд и остается открытым через 9,9 секунд, если за это время не пройдет пользователь»*:

$$
\begin{aligned}
&t \equiv 100; \\
&A_1 \equiv getVarBool(s_1, open) = FALSE \wedge getVarBool(s_2, open) = TRUE; \\
&A_2 \equiv getVarBool(s_3, open) = TRUE; \\
&A_3 \equiv getVarBool(s_4, PdOut) = TRUE.
\end{aligned}
$$

Доказательство условий корректности для требований этого класса выполняется следующим образом (Листинг 7):

Листинг 7: Схема доказательства для шаблона 7. Proof scheme for the pattern 7

```
theorem "VC inv env s0 input_values"
apply(unfold VC_def inv_def req_def)
apply rule
apply(rule context_conjI)
using extraInv_proof apply(simp add: VC_def)
apply rule
 apply simp
subgoal premises prems
  apply(insert prems(2))
  apply(unfold extraInv_def)
```

```
  apply(erule conjE)+
  subgoal premises ei
   apply(rule L2)
   using prems(1) ei(n) by simp
  done
done
```

В доказательстве сначала раскрываются определения условия корректности, инварианта и требования. Далее применяется правило *context_conjI* чтобы свести доказательство конъюнкции дополнительного инварианта и требования к доказательству дополнительного инварианта и доказательству требования в предположении, что дополнительный инвариант истинен. Дополнительный инвариант доказывается с помощью леммы *extraInv_proof*. Требование доказывается следующим образом. Первый конъюнкт *toEnvPs* доказывается методом *simp*. Для доказательства второго конъюнкта расщепляется конъюнкция в посылке условия корректности и в дополнительном инварианте применяется лемма *L2* и доказательство завершается методом *simp* с использованием посылок условия корректности и утверждения $ei(n)$ из дополнительного инварианта.

Лемма *L2* имеет вид:

$$
\begin{aligned}
&P7inv(A_1, A_2, A_3, t, t_1, s) \longrightarrow\\
&\forall s_1 \forall s_2 (toEnvP(s_1) \wedge toEnvP(s_2) \wedge substate(s_1, s_2) \wedge substate(s_2, s) \wedge toEnvNum(s_1, s_2) = 1 \wedge A_1(s_1, s_2) \longrightarrow\\
&\quad \exists s4 (toEnvP(s_4) \wedge substate(s_2, s_4) \wedge substate(s_4, s) \wedge toEnvNum(s_2, s_4) \leqslant t \wedge A_3(s_4) \wedge\\
&\qquad \forall s_3 (toEnvP(s_3) \wedge substate(s_2, s_3) \wedge substate(s_3, s_4) \wedge s_3 \neq s_4 \longrightarrow A_2(s_3))) \vee\\
&\quad \forall s_3 (toEnvP(s_3) \wedge substate(s_2, s_3) \wedge substate(s_3, s) \wedge toEnvNum(s_2, s_3) \leqslant t \longrightarrow A_2(s_3))).
\end{aligned}
$$

Здесь *P7inv* является шаблоном части дополнительного инварианта, порождаемого для требований, удовлетворяющих шаблону *P8*. Он имеет следующий вид:

$$
\begin{aligned}
&P7inv(A_1, A_2, A_3, t, t_1, s) \equiv\\
&\forall s_1 \forall s_2 (toEnvP(s_1) \wedge toEnvP(s_2) \wedge substate(s_1, s_2) \wedge substate(s_2, s) \wedge toEnvNum(s_1, s_2) = 1 \wedge A_1(s_1, s_2) \longrightarrow\\
&\quad \exists s_4 (toEnvP(s_4) \wedge substate(s_2, s_4) \wedge substate(s_4, s) \wedge toEnvNum(s_2, s_4) \leqslant t \wedge A_3(s_4) \wedge\\
&\qquad \forall s_3 (toEnvP(s_3) \wedge substate(s_2, s_3) \wedge substate(s_3, s_4) \wedge s_3 \neq s_4 \longrightarrow A_2(s_3))) \vee\\
&\quad toEnvNum(s_2, s) \geqslant t_1 \wedge\\
&\qquad \forall s_3 (toEnvP(s_3) \wedge substate(s_2, s_3) \wedge substate(s_3, s) \wedge toEnvNum(s_2, s_3) \leqslant t \longrightarrow A_2(s_3))).
\end{aligned}
$$

4.8. Шаблон *P8*

Шаблон $P8(s, t, A_1, A_2, A_3)$ так же, как и шаблон *P7* определяет требования, утверждающие, что если после двух последовательных итераций цикла управления в момент завершения второй итерации произошло событие A_1, связывающее значения переменных в моментах завершения этих двух итераций, то от этого момента (момента завершения второй итерации) условие A_2 должно выполняться как минимум в течение времени t или, пока не произойдет событие A_3. Но этот шаблон отличается от шаблона *P7* тем, что не требуется чтобы условие A_2 выполнялось в момент времени t после события A_1, если до этого момента не произошло событие A_3. Формула, описывающая событие A_1 связывает значения переменных в двух состояниях изменений, время между которыми составляет одну итерацию цикла управления. Этот шаблон имеет следующий вид:

$$
\begin{aligned}
&toEnvP(s) \wedge\\
&\forall s_1 \forall s_2 (substate(s_1, s_2) \wedge substate(s_2, s) \wedge toEnvP(s_1) \wedge toEnvP(s_2) \wedge toEnvNum(s_1, s_2) = 1 \wedge A_1(s_1, s_2) \longrightarrow\\
&\quad \exists s_4 (toEnvP(s_4) \wedge substate(s_2, s_4) \wedge substate(s_4, s) \wedge toEnvNum(s_1, s_2) \leqslant t \wedge A_3(s_4)) \wedge\\
&\qquad \forall s_3 (toEnvP(s_3) \wedge substate(s_2, s_3) \wedge substate(s_3, s_4) \wedge s_3 \neq s_4 \longrightarrow A_2(s_3))) \vee\\
&\quad \forall s_3 (toEnvP(s_3) \wedge substate(s_2, s_3) \wedge substate(s_3, s) \wedge toEnvNum(s_2, s_3) < t \longrightarrow A_2(s_3))).
\end{aligned}
$$

Примером, удовлетворяющим этому шаблону, является восьмое требование к программе управления турникетом: *«Если турникет только что открылся, он будет открыт в течение 10 секунд или пока не пройдет пользователь»* (см. раздел 3.1). Для этого требования имеем:

$$t \equiv 101;$$
$$A_1 \equiv getVarBool(s_1, open) = FALSE \wedge getVarBool(s_2, open) = TRUE;$$
$$A_2 \equiv getVarBool(s_3, open) = TRUE;$$
$$A_3 \equiv getVarBool(s_4, PdOut) = TRUE.$$

При $t = 0$ формула, определяющая данный шаблон, всегда истинна, так как истинен второй дизъюнкт в заключении импликации. Следовательно, в требованиях значение параметра t всегда больше нуля. Мы не определяем схему доказательства условий корректности непосредственно для данного шаблона. Для выполнения доказательства требования, соответствующие шаблону 8, сводятся к требованиям, соответствующим шаблону 7. В качестве значения параметра t в шаблоне 8 берется $t_0 - 1$, где t_0 — значение параметра t в шаблоне 7. Корректность этого сведения следует из следующей теоремы:

Теорема 1. *Требование* $\mathrm{P8}(s, t_0, A_1, A_2, A_3)$ *при* $t_0 > 0$ *истинно тогда и только тогда, когда истинно требование* $\mathrm{P7}(s, t_0 - 1, A_1, A_2, A_3)$.

Доказательство. Формулы $P8(s, t_0, A_1, A_2, A_3)$ и $P7(s, t_0 - 1, A_1, A_2, A_3)$ имеют вид $\forall s_1 \forall s_2 (substate(s_1, s_2) \wedge substate(s_2, s) \wedge toEnvP(s_1) \wedge toEnvP(s_2) \wedge toEnvNum(s_1, s_2) = 1 \wedge A_1(s_1, s_2) \longrightarrow Q)$. Пусть Q_1 — заключение импликации в формуле $P8(s, t_0, A_1, A_2, A_3)$, Q_2 — заключение импликации в формуле $P7(s, t_0 - 1, A_1, A_2, A_3)$.

Докажем эквивалентность формул Q_1 и Q_2.

1) Пусть выполняется Q_2. Если выполняется первый дизъюнкт в Q_2, то существует состояние изменений s_4, такое, что $toEnvNum(s_2, s_4) \leqslant t_0 - 1$, в s_4 выполняется A_3, а между состояниями s_2 и s_4, включая s_2, но не включая s_4 выполняется A_2. Но так как $toEnvNum(s_2, s_4) \leqslant t_0$, то выполняется первый дизъюнкт в формуле Q_1:

$$\exists s_4 (toEnvP(s_4) \wedge substate(s_2, s_4) \wedge substate(s_4, s) \wedge toEnvNum(s_1, s_2) \leqslant t_0 \wedge A_3(s_4)) \wedge \forall s_3 (toEnvP(s_3) \wedge substate(s_2, s_3) \wedge substate(s_3, s_4) \wedge s_3 \neq s_4 \longrightarrow A_2(s_3))).$$

Второй дизъюнкт в Q_2:

$$\forall s_3 (toEnvP(s_3) \wedge substate(s_2, s_3) \wedge substate(s_3, s) \wedge toEnvNum(s_2, s_3) \leqslant t_0 - 1 \longrightarrow A_2(s_3))$$

эквивалентен второму дизъюнкту в Q_1:

$$\forall s_3 (toEnvP(s_3) \wedge substate(s_2, s_3) \wedge substate(s_3, s) \wedge toEnvNum(s_2, s_3) < t_0 \longrightarrow A_2(s_3)).$$

Таким образом, из истинности Q_2 следует истинность Q_1.

2) Пусть выполняется Q_1. Если выполняется первый дизъюнкт в Q_2, то существует состояние изменений s_4, такое, что $toEnvNum(s_2, s_4) \leqslant t_0 - 1$, в s_4 выполняется A_3, а между состояниями s_2 и s_4, включая s_2, но не включая s_4 выполняется A_2. Если при этом выполняется $toEnvNum(s_2, s_4) \leqslant t_0 - 1$, выполняется первый дизъюнкт в Q_2:

$$\exists s_4 (toEnvP(s_4) \wedge substate(s_2, s_4) \wedge substate(s_4, s) \wedge toEnvNum(s_1, s_2) \leqslant t_0 - 1 \wedge A_3(s_4)) \wedge \forall s_3 (toEnvP(s_3) \wedge substate(s_2, s_3) \wedge substate(s_3, s_4) \wedge s_3 \neq s_4 \longrightarrow A_2(s_3))).$$

Если $toEnvNum(s_2, s_4) = t_0$ и не существует другого состояния s_4', удовлетворяющего указанным условиям, то любого состояния s_3 между s_2 и s, такого, что $toEnvNum(s_2, s_3) \leqslant t_0 - 1$, выполняется $substate(s_3, s_4)$ и $s_3 \neq s_4$. Действительно, если эти условия не выполняются, то выполняется

$substate(s_4, s_3)$, но это невозможно, так как $toEnvNum(s_2, s_3) < toEnvNum(s_2, s_4)$. Но тогда выполняется второй дизъюнкт в Q_2:

$$\forall s_3(toEnvP(s_3) \wedge substate(s_2, s_3) \wedge substate(s_3, s) \wedge toEnvNum(s_2, s_3) \leqslant t_0 - 1 \longrightarrow A_2(s_3)).$$

Если выполняется второй дизъюнкт в Q_1, то выполняется второй дизъюнкт в Q_2 в силу их эквивалентности.

□

4.9. Шаблон *P9*

Шаблон $P9(s, t_1, t_2, A_1, A_2, A_3)$ определяет класс требований, которые утверждают, что если после двух последовательных итераций цикла управления в момент завершения второй итерации произошло событие A_1, связывающее значения переменных в моментах завершения этих двух итераций, то от этого момента (момента завершения второй итерации), не более, чем через время t_1 должно произойти событие A_2, и после этого свойство A_3 должно выполняться по крайней мере до того, как будет достигнуто время t_2. Этот шаблон имеет следующий вид:

$$\begin{aligned}
&toEnvP(s)\wedge\\
&\forall s_1\forall s_2(substate(s_1, s_2) \wedge substate(s_2, s) \wedge toEnvP(s_1) \wedge toEnvP(s_2) \wedge toEnvNum(s_1, s_2) = 1\wedge\\
&\quad toEnvNum(s_2, s) \geqslant t_1 \wedge A_1(s_1, s_2) \longrightarrow\\
&\qquad \exists s_4(toEnvP(s_4) \wedge substate(s_2, s_4) \wedge substate(s_4, s) \wedge toEnvNum(s_2, s_4) \leqslant t_1 \wedge A_2(s_4))\wedge\\
&\qquad \forall s_5(toEnvP(s_5) \wedge substate(s_4, s_5) \wedge substate(s_5, s) \wedge toEnvNum(s_4, s_5) < t_2 \longrightarrow A_3(s_5)\wedge\\
&\qquad \forall s_3(toEnvP(s_3) \wedge substate(s_2, s_3) \wedge substate(s_3, s_4) \wedge s_3 \neq s_4 \longrightarrow \neg A_2(s_3))).
\end{aligned}$$

Примером, удовлетворяющим этому шаблону, является четвертое требование к программе управления вращающейся дверью: *«Если на секционные перегородки оказывается давление, то не более, чем через 0,3 с вращение приостанавливается не менее, чем на 1 с»* (см. раздел 3.3). Для этого требования имеем:

$$\begin{aligned}
&t_1 \equiv 1;\\
&t_2 \equiv 11;\\
&A_1 \equiv getVarBool(s_1, rotation) = TRUE \wedge getVarBool(s_2, pressure) = TRUE;\\
&A_2 \equiv getVarBool(s_4, brake) = TRUE;\\
&A_3 \equiv getVarBool(s_5, brake) = TRUE.
\end{aligned}$$

Доказательство условий корректности для требований, соответствующих девятому классу выполняется аналогично доказательствам для требований седьмого класса , но вместо леммы $L2$ применяется лемма $L3$, которая имеет следующий вид:

$$\begin{aligned}
&P9inv(s, t_1, t_{11}, t_2, t_{21}, A_1, A_2, A_3) \longrightarrow\\
&\forall s_1\forall s_2(substate(s_1, s_2) \wedge substate(s_2, s) \wedge toEnvP(s_1) \wedge toEnvP(s_2) \wedge toEnvNum(s_1, s_2) = 1\wedge\\
&\quad toEnvNum(s_2, s) \geqslant t_1 \wedge A_1(s_1, s_2) \longrightarrow\\
&\qquad \exists s_4(toEnvP(s_4) \wedge substate(s_2, s_4) \wedge substate(s_4, s) \wedge toEnvNum(s_2, s_4) \leqslant t_1 \wedge A_2(s_4))\wedge\\
&\qquad \forall s_5(toEnvP(s_5) \wedge substate(s_4, s_5) \wedge substate(s_5, s) \wedge toEnvNum(s_4, s_5) < t_2 \longrightarrow A_3(s_5)\wedge\\
&\qquad \forall s_3(toEnvP(s_3) \wedge substate(s_2, s_3) \wedge substate(s_3, s_4) \wedge s_3 \neq s_4 \longrightarrow \neg A_2(s_3))).
\end{aligned}$$

Здесь *P9inv* является шаблоном части дополнительного инварианта, порождаемого для требований, удовлетворяющих шаблону *P9*. Он имеет следующий вид:

$$
\begin{aligned}
&P9inv(s, t_1, t_{11}, t_2, t_{21}, A_1, A_2, A_3) \equiv \\
&\forall s_1 \forall s_2 (substate(s_1, s_2) \wedge substate(s_2, s) \wedge toEnvP(s_1) \wedge toEnvP(s_2) \wedge toEnvNum(s_1, s_2) = 1 \wedge A_1(s_1, s_2) \longrightarrow \\
&\quad (\exists s_4 (toEnvP(s_4) \wedge substate(s_2, s_4) \wedge substate(s_4, s) \wedge toEnvNum(s_2, s_4) \leqslant t_1 \wedge \\
&\quad toEnvNum(s_4, s) \geqslant t_{21} \wedge A_2(s_4)) \wedge \\
&\quad \forall s_5 (toEnvP(s_5) \wedge substate(s_4, s_5) \wedge substate(s_5, s) \wedge toEnvNum(s_4, s_5) < t_2 \longrightarrow A_3(s_5) \wedge \\
&\quad \forall s_3 (toEnvP(s_3) \wedge substate(s_2, s_3) \wedge substate(s_3, s_4) \wedge s_3 \neq s_4 \longrightarrow \neg A_2(s_3)))) \vee \\
&\quad toEnvNum(s_2, s) < t_{11} \wedge \\
&\quad \forall s_3 (toEnvP(s_3) \wedge substate(s_2, s_3) \wedge substate(s_3, s) \longrightarrow \neg A_2(s_3))).
\end{aligned}
$$

5. Распределение требований

В этом разделе приведены результаты распределения требований из коллекции, представленной в разделе 3 по классам в соответствие с шаблонами, определенными в разделе 4. Результаты классификации требований коллекции представлены в таблице 1. Каждый столбец таблицы соответствует номеру требования. На круговой диаграмме 1 показано общее распределение требований коллекции по шаблонам.

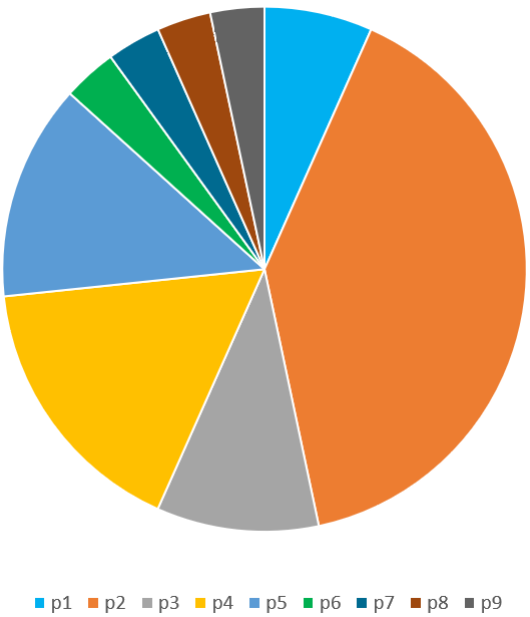


Fig. 1. Requirements distribution diagram

Рис. 1. Диаграмма распределения требований

Table 1. Requirements classification

Таблица 1. Классификация требований

Пример	1	2	3	4	5	6	7	8	9
Турникет	P1	P2	P2	P4	P5	P3	P2	P8	P7
Светофор	P4	P5	P4	P5	P6				
Вращающаяся дверь	P2	P2	P4	P9	P4	P3			
Холодильник	P2	P2	P1	P5	P2				
Термопот	P2	P2	P2	P3	P2				

6. Обзор литературы

6.1. Дедуктивная верификация темпоральных требований

В дедуктивной верификации требования описываются утверждениями в определенных точках программы, связывающими текущие значения переменных в этих точках [11]. Поэтому, обычно дедуктивная верификация используется для проверки нетемпоральных требований [12, 13]. Тем не менее, есть ряд работ, в которых дедуктивная верификация применяется для проверки темпоральных требований. Рассмотрим наиболее близкие из них.

В [14] представлена дедуктивная верификация управляющих программ на графическом языке LD релейно-контактных схем из стандарта МЭК 61131-3 [8] с темпоральными требованиями, заданных с помощью временных диаграмм. Код на языке LD и временная диаграмма транслировались в программу на входном языке WhyML системы дедуктивной верификации Why3 с последующей верификацией этой программы. События и стабильные состояния временной диаграммы представлялись в результирующей программе циклами, тело которых соответствует итерации цикла управления, а условие — стабильному состоянию временной диаграммы. По сравнению с этой работой мы верифицируем программы на более концептуально выразительном языке poST (по сравнению с LD) и требования к этим программам также имеют более общий вид, выраженный формулами логики предикатов первого порядка (по сравнению с временными диаграммами).

В STeP [15] используются правила верификации для сведения доказательства корректности программы на языке SPL относительно темпоральных требований к набору условий корректности, являющихся формулами логики первого порядка. В отличие от STeP, мы не разрабатываем специальные правила верификации для темпоральных требований, а используем обычные правила аксиоматической семантики, похожие на правила логики Хоара. Это достигается за счет использования состояний изменений вместо обычных состояний.

В [16] предложен дедуктивный метод верификации свойств реального времени для встраиваемых программ на Ассемблере. Метод строит временную вычислительную модель по программе на языке Ассемблер, приписывая каждому состоянию этой модели временную метку, определяемую в соответствие с нормативным временем вычисления ассемблерных инструкций, а затем проверяет относительно этой модели свойства на языке RTLTL (Real-Time Linear Temporal Logic), порождая условия корректности на языке логики предикатов первого порядка. В нашем случае используется более богатый язык программ, а требования на специализированном варианте типизированной логики предикатов первого порядка задавать проще, чем на RTLTL.

6.2. Языки и шаблоны описания темпоральных требований

Спецификация темпоральных требований играет важную роль в разработке критического управляющего программного обеспечения. Как правило, темпоральные свойства программ и систем формулируются в терминах модальных логик. Самыми распространёнными являются темпоральные логики LTL (Linear Temporal Logic) и CTL (Computational Tree Logic), которые не используют явные значения времени в спецификациях требований [17] (глава 1). Варианты этих логик MTL (Metric Temporal Logic) и TCTL (Timed CTL) позволяют задавать временные рамки выполнения свойств, однако верификация таких свойств становится значительно более сложной [17] (глава 29). Ранее нами была предложена темпоральная логика cycle-LTL, учитывающая циклическую природу функционирования систем управления [18]. Хотя по выразительной силе логика LTL эквивалентна логике предикатов первого порядке [17] (глава 2), её прямое использование, также как и cycle-LTL, в нашем подходе дедуктивной верификации процесс-ориентированных программ [2] оказывается затруднительным в силу отсутствия в них явного подсчёта времени и сохранения истории изменений переменных системы, которая соответствует путям в моделях Крипке, на которых обычно определяется истинность формул LTL. Поэтому для спецификации процесс-ориентированных систем

управления мы разработали язык DV-TRL, учитывающий такие их особенности как время функционирования цикла управления и таймеры процессов, а также позволяющий использовать историю изменений значений переменных.

Для совершенствования методов спецификации программ и систем широко проводится работа по систематической классификации формул темпоральных логик. Самая ранняя классификация [19] является синтаксической и включает наиболее общие свойства программных систем (живость, безопасность, справедливость, тупик). Классическая система шаблонов [20] включает наиболее популярные качественные требования для параллельных систем. Каждый шаблон описывается на естественном языке вместе с его формализацией в темпоральных логиках CTL и LTL [17], количественными регулярными выражениями и графическим представлением с помощью GIL. В [21—23] эти шаблоны распространены на случай вероятностных систем и систем реального времени соответственно. Некоторые составные шаблоны событий предложены в [24, 25]. В [26] авторы представляют шаблоны количественных характеристик возникновения событий, а также шаблон данных [27]. Все упомянутые подходы работают только с шаблонами, семантика которых выражается в логике линейного времени LTL, её вероятностных расширениях или расширениях реального времени. Однако [28] показывает необходимость в некоторых случаях использовать логику ветвящегося времени CTL с аналогичными расширениями. Работа [29] объединяет представления классических шаблонов с вероятностными шаблонами и шаблонами реального времени и задаёт их описание на ограниченном английском языке. Недавно мы разработали классификацию требований к процесс-ориентированным программам [30], основанную на LTL-семантике шаблона EDTL [31]. Подводя итог, можно сказать, что современные подходы обычно фокусируются на шаблонах спецификаций, имеющих семантику в виде темпоральных логик, в то время как в данной работе шаблоны требований и соотвествующая им классификация построены на логике предикатов первого порядка, и принимают во внимание принципы функционирования процесс-ориентированных систем. Наша классификация, хоть и не является полной, включает представительные классы требований, встречающихся в реальных программах.

Заключение

В статье предложены новые шаблоны описания классов темпоральных требований на языке DV-TRL, а также представлены схемы доказательства условий корректности, порождаемых для требований, удовлетворяющих ранее разработанным и новым шаблонам, в системе Isabelle/HOL. Описана коллекция требований и для каждого шаблона рассмотрены примеры требований из этой коллекции. Приведена статистика по распределению требований коллекции по классам, определяемых шаблонами. Результаты статьи являются важным вкладом в разработанный ранее дедуктивный подход к верификации процесс-ориентированных программ, так как в этом подходе требования являются инвариантами цикла управления и, таким образом, мы фактически определяем шаблоны инвариантов цикла управления.

Планируется расширить коллекцию требований применительно к новым программ и новым устройствам и разработать новые шаблоны требований и схемы доказательства для них. На основе предложенных схем доказательства для шаблонов требований мы также планируем разработать средства автоматизации доказательства условий корректности.

References

[1] V. E. Zyubin, “Hyper-automaton: A model of control algorithms”, in *2007 Siberian Conference on Control and Communications*, 2007, pp. 51–57. DOI: 10.1109/SIBCON.2007.371297.

[2] I. Anureev, N. Garanina, T. Liakh, A. Rozov, V. Zyubin, and S. Gorlatch, “Two-step deductive verification of control software using Reflex”, in *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, Springer, 2019, pp. 50–63. DOI: 10.1007/978-3-030-37487-7_5.

[3] V. E. Zyubin, T. V. Liakh, and A. S. Rozov, “Reflex language: A practical notation for cyber-physical systems”, *System Informatics*, no. 12, pp. 85–104, 2018.

[4] C. Paulin-Mohring, “Introduction to the Coq proof-assistant for practical software verification”, in *LASER Summer School on Software Engineering*, Springer, 2011, pp. 45–95. DOI: 10.1007/978-3-642-35746-6_3.

[5] I. Chernenko, I. S. Anureev, N. O. Garanina, and S. M. Staroletov, “A temporal requirements language for deductive verification of process-oriented programs”, in *IEEE 23rd International Conference of Young Professionals in Electron Devices and Materials (EDM)*, 2022, pp. 657–662. DOI: 10.1109/EDM55285.2022.9855145.

[6] I. M. Chernenko and I. S. Anureev, “Development of verification condition generator for process-oriented programs in poST language”, in *IEEE 24th International Conference of Young Professionals in Electron Devices and Materials (EDM)*, 2023, pp. 1760–1765. DOI: 10.1109/EDM58354.2023.10225217.

[7] V. E. Zyubin, A. S. Rozov, I. S. Anureev, N. O. Garanina, and V. Vyatkin, “poST: A process-oriented extension of the IEC 61131-3 structured text language”, *IEEE Access*, vol. 10, 2022. DOI: 10.1109/ACCESS.2022.3157601.

[8] IEC, *IEC 61131-3: 2013 programmable controllers-Part 3: Programming languages*, https://webstore.iec.ch/publication/4552, International Standard, 2013.

[9] L. C. Paulson, T. Nipkow, and M. Wenzel, “From LCF to Isabelle/HOL”, *Formal Aspects of Computing*, vol. 31, pp. 675–698, 2019.

[10] I. Chernenko, “Requirements patterns in deductive verification of process-oriented programs and examples of their use”, *System Informatics*, no. 22, 2023. DOI: 10.31144/si.2307-6410.2023.n22.p11-20.

[11] J.-C. Filliâtre, “Deductive software verification”, *International Journal on Software Tools for Technology Transfer*, vol. 13, pp. 397–403, 2011. DOI: 10.1007/s10009-011-0211-0.

[12] D. Gurov, P. Herber, and I. Schaefer, “Automated verification of embedded control software”, in *International Symposium on Leveraging Applications of Formal Methods*, Springer, 2020, pp. 235–239. DOI: 10.1007/978-3-030-61467-6_15.

[13] D. Gurov, C. Lidström, M. Nyberg, and J. Westman, “Deductive functional verification of safety-critical embedded C-code: An experience report”, in *Critical Systems: Formal Methods and Automated Verification*, 2017, pp. 3–18.

[14] C. B. Lourenço, D. Cousineau, F. Faissole, C. Marché, D. Mentré, and H. Inoue, “Automated verification of temporal properties of Ladder programs”, in *Formal Methods for Industrial Critical Systems*, 2021, pp. 21–38.

[15] Z. Manna *et al.*, “An update on STeP: Deductive-algorithmic verification of reactive systems”, in *Tool Support for System Specification, Development and Verification*, 1999, pp. 174–188. DOI: 10.1007/978-3-7091-6355-9_13.

[16] S. Yamane, “Deductive verification method of real-time safety properties for embedded assembly programs”, *Electronics*, vol. 8, no. 10, p. 1163, 2019.

[17] E. M. Clarke, T. A. Henzinger, H. Veith, R. Bloem, *et al.*, *Handbook of model checking*. Springer, 2018, 1212 pp.

[18] N. O. Garanina *et al.*, “A temporal logic for programmable logic controllers”, *Automatic Control and Computer Sciences*, vol. 55, no. 7, pp. 763–775, 2021. DOI: 10.3103/S0146411621070038.

[19] Z. Manna and A. Pnueli, *The temporal logic of reactive and concurrent systems: Specification*. Springer New York, NY, 1992, 427 pp. DOI: 10.1007/978-1-4612-0931-7.

[20] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification", in *Proceedings of the 21st International Conference on Software Engineering*, Association for Computing Machinery, 1999, pp. 411–420. DOI: 10.1145/302405.302672.

[21] L. Grunske, "Specification patterns for probabilistic quality properties", in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 31–40. DOI: 10.1145/1368088.1368094.

[22] S. Konrad and B. H. C. Cheng, "Real-time specification patterns", in *Proceedings of the 27th International Conference on Software Engineering*, Association for Computing Machinery, 2005, pp. 372–381. DOI: 10.1145/1062455.1062526.

[23] A. Mekki, M. Ghazel, and A. Toguyéni, "Assisting temporal requirement specification", *Computer Technology and Application*, vol. 3, no. 1, pp. 47–55, 2012.

[24] O. Mondragon, A. Q. Gates, and S. Roach, "Prospec: Support for elicitation and formal specification of software properties", *Electronic Notes in Theoretical Computer Science*, vol. 89, no. 2, pp. 67–88, 2003. DOI: 10.1016/S1571-0661(04)81043-0.

[25] S. Salamah, A. Gates, and V. Kreinovich, "Validated templates for specification of complex LTL formulas", *Journal of Systems and Software*, vol. 85, no. 8, pp. 1915–1929, 2012. DOI: 10.1016/j.jss.2012.02.041.

[26] D. Bianculli, C. Ghezzi, C. Pautasso, and P. Senti, "Specification patterns from research to industry: A case study in service-based applications", in *34th International Conference on Software Engineering (ICSE)*, 2012, pp. 968–976. DOI: 10.1109/ICSE.2012.6227125.

[27] S. Halle, R. Villemaire, and O. Cherkaoui, "Specifying and validating data-aware temporal web service properties", *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 669–683, 2009. DOI: 10.1109/TSE.2009.29.

[28] A. Post, I. Menzel, and A. Podelski, "Applying restricted english grammar on automotive requirements—Does it work? A case study", in *Requirements Engineering: Foundation for Software Quality*, Springer Berlin Heidelberg, 2011, pp. 166–180.

[29] M. Autili, L. Grunske, M. Lumpe, P. Pelliccione, and A. Tang, "Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar", *IEEE Transactions on Software Engineering*, vol. 41, no. 7, pp. 620–638, 2015. DOI: 10.1109/TSE.2015.2398877.

[30] A. N. Getmanova, N. O. Garanina, S. M. Staroletov, V. E. Zyubin, and I. S. Anureev, "Semantic classification of event driven temporal logic requirements", in *IEEE 23rd International Conference of Young Professionals in Electron Devices and Materials (EDM)*, 2022, pp. 663–668. DOI: 10.1109/EDM55285.2022.9855053.

[31] V. Zyubin, I. Anureev, N. Garanina, S. Staroletov, A. Rozov, and T. Liakh, "Event-driven temporal logic pattern for control software requirements specification", in *International Conference on Fundamentals of Software Engineering*, Springer, 2021, pp. 92–107.

MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 31, NO. 1, 2024
journal homepage: www.mais-journal.ru

THEORY OF SOFTWARE

Model Checking Programs in Process-Oriented IEC 61131-3 Structured Text

N. O. Garanina[1], S. M. Staroletov[1], V. E. Zyubin[1], I. S. Anureev[1]

DOI: 10.18255/1818-1015-2024-1-32-53

[1]Institute of Automation and Electrometry SB RAS, Novosibirsk, Russia

MSC2020: 68N30
Research article
Full text in Russian

Received January 17, 2024
Revised February 6, 2024
Accepted February 14, 2024

The process-oriented programming is a paradigm based on the process concept where each process is a concurrent finite state machine inside. The paradigm is intended for PLC (programmable logic controllers) developers to write Industry 4.0-enabled software. The poST language is a promising process-oriented extension of the IEC 61131-3 Structured Text (ST) language designed to provide a conceptual consistency of the PLC source code with technological description of the process under control. This language combines the advantages of FSM-based programming with the standard syntax of the ST language. We propose transformational semantics of poST providing rules for translation of poST language statements to Promela — the input language of the SPIN model checker. Following these semantic rules, our Xtext-based translator outputs a Promela model for the poST program. Our contribution is the poST transformational semantics and the method for automatic generation of the Promela code from poST control programs. The resulting Promela program is ready to be verified with SPIN model checker against linear temporal logic requirements to the source poST program.

In the paper we provide an overview of related work, as well as a brief description of the poST and Promela languages. Further, the Promela poST translation rules cover control flow statements, process creation and state management constructs, and timeout management. Then we define service processes for modeling the external environment and managing high-level LTL specifications. Then we present the main ideas of implementing the translator poST to Promela. We also illustrate our approach using the example of a system for managing electricity consumption and production, including renewable sources.

Keywords: control software; model checking; process-oriented programming; LTL; SPIN; Structured Text

INFORMATION ABOUT THE AUTHORS

Garanina, Natalia O. (corresponding author)	ORCID iD: 0000-0001-9734-3808. E-mail: garanina@iis.nsk.su Senior researcher, PhD
Staroletov, Sergey M.	ORCID iD: 0000-0001-5183-9736. E-mail: serg_soft@mail.ru Senior researcher, PhD
Zyubin, Vladimir E.	ORCID iD: 0000-0002-8198-3197. E-mail: zyubin@iae.nsk.su Head of Laboratory, Dr. Sc.
Anureev, Igor S.	ORCID iD: 0000-0001-9574-128X. E-mail: anureev@iis.nsk.su Senior researcher, PhD

Funding: State task IAaE SB RAS, project No. 122031600173-8, state task No. AAAA-A19-119120290056-0.

For citation: N. O. Garanina, S. M. Staroletov, V. E. Zyubin, and I. S. Anureev, "Model checking programs in process-oriented IEC 61131-3 Structured Text", *Modeling and Analysis of Information Systems*, vol. 31, no. 1, pp. 32–53, 2024. DOI: 10.18255/1818-1015-2024-1-32-53.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 31, № 1, 2024
сайт журнала: www.mais-journal.ru

THEORY OF SOFTWARE

Верификация моделей программ на процесс-ориентированном расширении языка Structured Text стандарта IEC 61131-3

Н. О. Гаранина[1], С. М. Старолетов[1], В. Е. Зюбин[1], И. С. Ануреев[1]

DOI: 10.18255/1818-1015-2024-1-32-53

[1]Институт автоматики и электрометрии СО РАН, Новосибирск, Россия

УДК 004.822+681.51
Научная статья
Полный текст на русском языке

Получена 17 января 2024 г.
После доработки 6 февраля 2024 г.
Принята к публикации 14 февраля 2024 г.

Процессно-ориентированное программирование — это парадигма, основанная на концепции процесса. Каждый процесс представляет собой конечный автомат. Эта парадигма предназначена для разработчиков ПЛК (программируемых логических контроллеров) для написания программного обеспечения с поддержкой Индустрии 4.0. Язык poST является многообещающим процессно-ориентированным расширением языка структурированного текста (ST) МЭК 61131-3, предназначенным для обеспечения концептуальной согласованности исходного кода ПЛК с технологическим описанием управляемого процесса. Этот язык сочетает в себе преимущества программирования на основе конечных автоматов со стандартным синтаксисом языка ST. Мы предлагаем трансформационную семантику poST, заданную правилами перевода операторов языка poST в Promela — входной язык средства проверки моделей SPIN. Следуя этим правилам, наш транслятор, основанный на технологии Xtext, строит модель Promela для программы poST.

Основной вклад нашей статьи — это трансформационная семантика poST и метод автоматической генерации кода Promela из программ управления PoST. Полученная модель Promela готова к проверке с помощью системы верификации моделей SPIN на соответствие требованиям к исходной программе poST, выраженных в терминах линейной темпоральной логики LTL. В статье мы приводим обзор связанных работ, а также краткое описание языков poST и Promela. Представленные далее правила трансляции poST в Promela покрывают операторы потока управления, конструкции создания процессов и управления их состояниями, а также операторы для таймаутов. Отдельно определены сервисные процессы для моделирования внешней среды и задания высокоуровневых LTL спецификаций. Затем мы останавливаемся на основных идеях реализации транслятора poST в Promela, и далее иллюстрируем наш подход на примере системы управления потреблением и производством электроэнергии, в том числе из возобновляемых источников.

Ключевые слова: управляющее программное обеспечение; проверка моделей; процесс-ориентированное программирование; LTL; SPIN; Structured Text

ИНФОРМАЦИЯ ОБ АВТОРАХ

Гаранина, Наталья Олеговна (автор для корреспонденции)	ORCID iD: 0000-0001-9734-3808. E-mail: garanina@iis.nsk.su Старший научный сотрудник, к.ф.-м.н.
Старолетов, Сергей Михайлович	ORCID iD: 0000-0001-5183-9736. E-mail: serg_soft@mail.ru Старший научный сотрудник, к.ф.-м.н.
Зюбин, Владимир Евгеньевич	ORCID iD: 0000-0002-8198-3197. E-mail: zyubin@iae.nsk.su Заведующий лаборатории, д.т.н.
Ануреев, Игорь Сергеевич	ORCID iD: 0000-0001-9574-128X. E-mail: anureev@iis.nsk.su Старший научный сотрудник, к.ф.-м.н.

Финансирование: Госзадание ИАиЭ СО РАН, проект № 122031600173-8, госзадание № АААА-А19-119120290056-0.

Для цитирования: N. O. Garanina, S. M. Staroletov, V. E. Zyubin, and I. S. Anureev, “Model checking programs in process-oriented IEC 61131-3 Structured Text”, *Modeling and Analysis of Information Systems*, vol. 31, no. 1, pp. 32–53, 2024. DOI: 10.18255/1818-1015-2024-1-32-53.

Введение

При значительном прогрессе в формальных методах доказательства моделей программ остается вопрос их применимости к реальному программному обеспечению. Можно констатировать, что применение таких методов к произвольным программам не представляется целесообразным из-за трудоемкости перевода языковых конструкций с неоднозначной семантикой, большого пространства состояний, неопределенности требований или сложности их задания.

С другой стороны, существует такой класс, как программы для ПЛК (Программируемые Логические Контроллеры), предназначенные для реализации алгоритмов управления. Цена ошибки в управляющем ПО высока, что указывает на то, что управляющие программы нуждаются в формализации и последующем доказательстве их корректности относительно требований, которые также нужно формализовать. Программы на языках для ПЛК лаконичны, сами языки содержат небольшое количество конструкций, а требования выражаются с помощью небольшого количества ключевых переменных. Эти особенности языков ПЛК делают применение формальных методов для них более перспективным.

В настоящее время для программируемых промышленных контроллеров используются языки стандарта МЭК 61131-3 [1]. Они различаются представлением программы: в виде текста, лестничных схем, функциональных блок-схем и списков команд. В частности, широкое распространение имеет язык структурированного текста (Structured Text, ST) [2]. Этот язык похож на Паскаль и имеет простой синтаксис с дополнительными конструкциями (например, таймаутами и интервалами), специфичными для программ ПЛК. Однако большинство программ управления цикличны и зависят от состояния, что приводит к большому количеству операторов переключения в тексте программы.

Чтобы избежать громоздкости текста программ для ПЛК в работе [3] мы предложили использовать понятие процесса как основной логической сущности программы для предоставления разработчикам удобных средств работы с состояниями и переходами. Мы называем программирование по этой методологии процесс-ориентированным, а язык, который включает средства работы с процессами и их состояниями, — процесс-ориентированным языком программирования. Разработанный нами язык poST (process-oriented Structured Text) [4] является одним из процесс-ориентированных языков. Он расширяет стандартизированный язык ST, для которого имеются среды разработки, моделирования и создания встроенного ПО реальных контроллеров.

Использование формальных методов верификации программ ПЛК оправдано прежде всего тем, что такие программы могут работать с дорогостоящим оборудованием предприятий и некорректное поведение программы может привести как к финансовым потерям, так и к серьезным последствиям для окружающей среды и персонала. Поэтому необходимо предусмотреть как можно большее разнообразие средств верификации таких программ. Поэтому в нашей работе мы предлагаем трансляцию из языка poST во входной язык инструмента верификации SPIN [5]. Эту трансляцию можно также назвать транспиляцией, поскольку она использует только исходные коды программ, и в настоящее время это удобный способ переключения между различными языками [6].

Инструмент SPIN использует один из формальных методов — проверку моделей [7]. Обзор практики применения проверки моделей представлен в статье Оватмана и др. [8]. Метод проверки моделей, применяемый в SPIN, состоит в следующем: на вход системе проверки подаётся формальное описание конечной системы переходов и требование корректности этой системы, представленное в виде темпоральной формулы; по системе переходов и требованию строится автомат Бюхи (композиция системы и отрицания требования), язык которого методом DFS или BFS проверяется на непустоту. Если находится допустимый путь в автомате-композиции — это означает, что язык композиции системы и отрицания требования не пуст, и найденный путь (трасса) служит контрпримером, т. е. историей изменения значений переменных системы, которая приводит

к нарушению требования. Начиная с первых работ Кларка [9], в качестве источника системы переходов предлагалось использовать формализованный язык, из которого система переходов получается единственным способом. Инструмент SPIN отвечает всем нашим требованиям: (1) имеет формальный входной язык для описания моделей; (2) требования выражаются в виде формул линейной темпоральной логики LTL [7] над ключевыми переменными программы; (3) инструмент одобрен сообществом, используется в большом количестве реальных проектов и реализует эффективные алгоритмы проверки моделей с большим пространством состояний.

Входным языком SPIN является Promela (Protocol Meta-Language), который наследует семантику алгебры процессов CSP (Communicating Sequential Processes) [10], расширяя её некоторыми акторными возможностями. Он описывает параллельные и распределенные системы как взаимодействующие процессы. В силу близости этой парадигмы к процесс-ориентированному подходу можно было бы описывать процесс-ориентированные системы управления сразу на Promela, однако в этом языке нет средств работы с состояниями процессов и таймаутами. Хотя состояния, процессы и переключение между ними могут быть представлены с помощью условных конструкций, дополнительных переменных и передачи активирующих сообщений, это делает код более громоздким и запутанным. Кроме того, возникает необходимость разработки для Promela компилятора или транслятора в ST.

В этой статье мы описываем правила трансформации кода из poST в Promela, а также вопросы реализации транспайлера poST2Promela в соответствии с этими правилами. Реализация представляет собой решение, основанное на инструменте синтаксического анализа Eclipse Xtext на Java и Xtend. Мы добавили обзор связанных работ, пример трансляции и детали реализации к предварительной версии работы, опубликованной в [11].

Оставшаяся часть статьи имеет следующую структуру. В разделе 1 мы обсуждаем современное состояние в области создания моделей Promela для верификации различных свойств программных систем. В разделе 2 мы рассматриваем особенности языков poST и Promela. Раздел 3 определяет правила трансформационной семантики языка poST. Вопросы реализации обсуждаются в разделе 4. Процесс трансформации продемонстрирован на примере солнечной электростанции в разделе 5, а выводы приведены в Заключении.

1. Обзор связанных работ

Тема трансляции разнообразных программных моделей и программ во входные языки верификаторов моделей, в частности в Promela, довольно популярна и в разное время создавались трансляторы программ на различных языках для их последующей верификации. Рассмотрим некоторые из них.

В статье [12] предлагается трансформация предметно-ориентированного языка управления роботами в Promela. Поскольку конструкции языка относительно просты и язык не допускает даже рекурсии, перевод на Promela относительно прост: 1) получение AST (абстрактного синтаксического дерева) из программы управления роботом, 2) получение CFG (графа потока управления) из AST, 3) трансляция CFG в полученный код. Перед этим выполняется процесс абстрактной интерпретации [13], чтобы избавиться от символьных переменных и оптимизировать CFG. Авторы также используют специальный процесс PLC для координации между роботами, а требования задаются матрицами возможных запросов от пользователя. Все необходимые конструкции выражаются в Promela. Процесс трансляции не формализован, и при этом авторы отмечают, что абстрактная интерпретация и оптимизация могут привести к неправильному генерированию программ, если в исходной программе присутствуют некоторые типы goto-переходов.

В статье [14] Леу и Хольцманн представили концепцию визуального языка v-Promela, который генерирует код Promela из моделей, созданных пользователями в графической среде Hybrid FSM. Вводится понятие капсулы как логического состояния с возможностью описания связей, сообщений между капсулами, при этом поддерживается декомпозиция. В этом случае состояния преобразуются в метки в коде Promela с генерацией встроенных функций для поддержки семантики входа и выхода из состояния. Для обеспечения корректности переходов используются конструкции `d_step` и `atomic`, чтобы сгенерированный код согласно модели работал как положено во всех допустимых ситуациях чередования. При этом семантика языка не формализована. Похожий подход можно наблюдать в статье Бенерекетти и др. [15], где код Promela генерируется из расширенного представления автомата, в данном случае из динамических конечных автоматов, путем определения семантики операторов (частей) конечного автомата и последующего описания алгоритма генерации кода в виде текста алгоритма, который представляет собой лишь полуформальное описание.

Лион и др. [16] предложили подход с преобразованием кода из языка описания протоколов Reo в код Promela для проверки LTL-требований к протоколам. Поскольку в Reo есть концепция портов, связь через них транслируется по каналам, встроенным в язык Promela, и для каждого порта используется два канала: один для данных, другой для синхронизации. Поскольку протокол Reo описывается как агент, выполняющий последовательность защищенных условий и действий (их также можно получить из графических схем), авторы формально определяют семантику преобразования спецификации таких условий и действий в код, где каждый агент — это процесс Promela.

Кларл [17] предлагает язык Helena — предметно-ориентированную спецификацию агентов и их отношений (например, для описания протоколов p2p) в редакторе на основе Eclipse. Предлагаются алгоритмы перевода в Promela для дальнейшей проверки свойств, выраженных в LTL\X (LTL без оператора neXt). Сам перевод описан неформально и для одного из операторов языка дан шаблон Xtend, поскольку для преобразования используется структура Xtext. По используемым инструментам трансляции наша работа близка к этой.

В работах [18, 19] Дилли и Ланге предлагают подход Gomela проверки программ на подмножестве языка Go путем генерации соответствующего кода на Promela. Проверяется корректность функций передачи сообщений. Преобразование описывается в функциональном стиле с использованием рекурсивной функции TransStmts, которая переводит все операторы MiniGo в конструкции Promela. В результате по прилагаемым бенчмаркам можно проверить корректность простых алгоритмов, а не готовых программ.

Работу Бринксмы, Мадера и Фенкера [20] можно считать новаторской работой по верификации программ ПЛК. Здесь впервые была введена формальная концепция цикла управления ПЛК. На примере бетонного завода строится модель процесса управления в Promela и требования к ней выражаются в LTL. Однако модель системы не генерируется автоматически, а создается вручную.

Что касается верификации программ на языках семейства 61131, то работа [21] предлагает получить промежуточный код Promela для блоков FBD (Function Block Diagram) с целью их дальнейшей проверки. Поскольку код предполагается генерировать по исходному представлению графа, сначала выполняется топологическая сортировка графа. Применимость метода продемонстрирована на одном примере [22].

Формальное моделирование и верификация программ ПЛК на основе схем МЭК 61499 представлены в статьях Лях и др. [23], а также Шатрова и Вяткина [24]. Они используют системы верификации NuSMV и SPIN, и отмечают более короткое время проверки и большую пригодность для циклических моделей как преимущества системы SPIN.

Недавняя статья Эбненасира [25] направлена на отказоустойчивый анализ программ ПЛК, выраженных в виде схем лестничной логики. Представлены не до конца уточненные способы транс-

ляции языковых конструкций (импульс, инструкции установки/сброса, времени) в конструкции Promela. В качестве источника требований предлагается использовать временные диаграммы, по которым автор генерирует простые последовательные отношения с помощью логики LTL. Автор не приводит примеры использования этого подхода для спецификации и верификации требований к реальным программам.

Подводя итог вышесказанному, большинство этих статей в основном являются работами proof-of-the-concept или диссертациями, и их применимость в индустрии неизвестна. Как правило, в рассматриваемых статьях недостаточно проработаны сложные вопросы, связанные с особенностью семантики программ ПЛК, а требования к примерам программ не выражены в терминах циклов управления ПЛК [26]. В нашем подходе мы используем предметно-ориентированный язык, удобный для выражения процессов и их состояний, создаем проекции Promela для всех синтаксических конструкций этого языка и детально описываем условия возможности трансляции poST-программ для ПЛК в язык Promela.

2. Языки poST и Promela

Язык poST представляет собой процесс-ориентированное расширение языка ST МЭК 61131-3, которое обеспечивает концептуальную согласованность исходного кода ПЛК с технологическим описанием управляемого процесса. Язык сочетает в себе преимущества программирования на основе конечных систем переходов с традиционным синтаксисом языка ST, что облегчает его внедрение. В основе его семантики лежит понятие гиперпроцесса [3].

Программа poST, отражая цикл управления ПЛК, включает в себя процессы, активность которых организована в цикл в порядке их появления в программном коде. Эта схема предполагает модели ПЛК, которые абстрагируются от времени цикла управления [26], при этом само время цикла управления задано с помощью параметра INTERVAL. Каждый процесс определяется упорядоченным набором своих состояний. Когда в цикле управления очередь активации переходит к процессу, он выполняет конечную последовательность действий, связанную с его текущим состоянием. Для описания этих действий процесса мы используем стандартные конструкции ST (объявления переменных, операторы потока управления и т. д.) и специфические процесс-ориентированные функции: операторы управления состояниями процесса (START/STOP PROCESS, SET NEXT и другие) и операторы таймаута. Взаимодействие процессов в poST происходит через общие переменные. Семантика языка poST предполагает автоматическую реализацию низкоуровневых конструкций для отображения сигналов ввода-вывода на программные переменные, состояний процесса, операторов таймаута и цикла управления, имеющего определенную длительность, заданную в INTERVAL. Грамматика языка poST в формате Xtext доступна в репозитории[1].

Язык Promela[2] используется для описания параллельных процессов, основанных на формализме CSP. Программа на Promela состоит из параллельных процессов, исполнение которых реализует семантику чередования. Чередование может быть ограничено операторами `atomic` и `d_step`, которые не позволяют прерывать последовательность действий процесса. В Promela переменные могут принимать значения следующих конечных типов: логические, перечислимые, целочисленные значения (в диапазоне от $-(2^{31}-1)$ до $2^{31}-1$). Процессы Promela взаимодействуют через общие переменные, а также через каналы сообщений, реализующие синхронные и асинхронные способы связи. Язык Promela включает блокирующие операторы потока управления `if` и `do`, в то время как аналогичные операторы потока управления poST являются неблокирующими. Программа на Promela может быть проверена на соответствие требованиям, сформулированным средствами

[1] https://github.com/SergeyStaroletov/poST_to_Promela_compiler_dev/blob/main/poST_grammar.xtext
[2] http://spinroot.com/spin/Man/grammar.html

темпоральной логики LTL, с помощью инструмента проверки моделей SPIN [5]. Для проверки этот инструмент использует композицию автоматов Бюхи для программы и формулы, поэтому в Promela возможны только конечные типы переменных модели. Следовательно, типы poST, представляющие действительные числа, не могут быть напрямую преобразованы в типы Promela, и в настоящей работе мы ограничиваемся программами poST, которые не содержат переменные действительных типов и библиотечные функции, возвращающие действительные значения. Ещё одна особенность poST, требующая специального внимания при трансляции — это работа с таймаутами, поскольку в Promela нет переменных типа времени.

Язык poST определяет алгоритмы управления, которые взаимодействуют с некоторой средой. Мы предполагаем, что исходный код poST кроме управляющей программы содержит и программу управляемого объекта для целей анализа поведения системы управления, которую они составляют. Транслируя код обеих программ, мы строим модель Promela, соответствующую порядку активации процессов, структуре состояний процессов, управлению таймаутами и типам переменных с точностью до абстрагирования от типов poST, представляющих действительные числа.

Сформулируем следующие принципы преобразования poST-программ в Promela-модели:

1. **Переменные**. Каждая переменная poST-программы имеет взаимно-однозначное соответствие некоторой переменной Promela-модели. Для каждого процесса p определена локальная переменная его состояния $state_p$. Она принимает значения перечислимого типа $State_p$, включающего специальные состояния *Stop* и *Error*. Глобальная переменная $time_p$ объявляется для каждого процесса p, в poST-коде которого есть операторы таймаута.
2. **Процессы**. Процесс poST p отображается в Promela-процесс p, бесконечно выполняющийся в цикле `do-od`. Пребывание процесса p в состоянии $s \in State_p$ задается с помощью оператора $(state_p == s)$, за которым следует тело состояния s. Состояние процесса p изменяется путем присвоения переменной $state_p$ значения из $State_p$. Тело Promela-процесса p состоит из условий пребывания в состояниях и их тел. Это тело заключено в оператор `atomic`, чтобы гарантировать последовательное выполнение действий согласно семантике poST.
3. **Таймауты**. В случае, если poST-процесс p выполняет действия по таймауту T, в соответствующем Promela-процессе запуск (сброс) связанного таймера $time_p$ транслируется в присваивание $time_p = 0$, и выполнение этих действий ограничено защитным условием $(time_p == T_p)$. Значения таймера увеличиваются на единицу в каждом цикле, пока не достигнут значения T_p или не произойдёт сброс таймера. При вычислении значения T_p используется значение соответствующего таймаута и значение INTERVAL.
4. **Синхронизация**. Существует несколько способов моделирования последовательного выполнения процессов poST в Promela. Один из них основан на передаче активирующего сообщения от процесса к процессу по каналу *cur* единичной емкости. Это сообщение содержит уникальный идентификатор процесса. После завершения действий в текущем состоянии функции процесс p передает ход процессу q, отправляя сообщение q в канал *cur*. Процесс q начинает выполняться после получения этого сообщения. Благодаря блокирующей семантике Promela-выражений последовательное выполнение процессов обеспечивается тем, что процесс может запуститься только после получения сообщения со своим именем.
5. **Моделирование входных данных**. Входные данные для алгоритмов управления поступают из трех источников: данные внешней среды, которые могут не зависеть от работы алгоритма управления, данные управляемого объекта, которые зависят от выходов алгоритма управления, а также выходные данные процессов, обеспечивающие их внутреннее взаимодействие. Сервисный Promela-процесс `Gremlin` моделирует данные неуправляемой среды. Этот процесс присваивает случайные значения соответствующим входным переменным. Мо-

делирование данных объекта управления и выходных данных других процессов задается процессами Promela, соответствующими их poST-коду, при этом сервисный Promela-процесс `OutInput` корректно сопоставляет имена входных и выходных данных процессов.

6. **Структура выполнения итоговой Promela-модели**. Согласно семантике poST, передача активирующего сообщения в результирующей Promela-модели организована следующим образом. Сначала входные данные обновляются процессом `Gremlin`, затем процессом `OutInput`. Затем очередь переходит к процессам контроллера, которые образуют последовательность выполнения посредством передачи активирующих сообщений. Последний процесс контроллера снова передает ход процессу `Gremlin`.

В следующем разделе правила трансляции конструкций языка poST в конструкции языка Promela описаны более подробно.

3. Правила трансляции языка poST в язык Promela

3.1. Именование, типы, объявления и операции

В языке poST есть глобальное пространство имен, пространство имен программ и процессов, тогда как в Promela есть только глобальное пространство имен и пространства имен процессов. Чтобы избежать конфликтов имен, мы формируем полные имена объектов в модели Promela программы poST, учитывая (1) имя объекта; (2) вид объекта (процесс, переменная и т. д.); и (3) имя процесса или программы, в которой находится объект. Чтобы улучшить читаемость программы, мы используем режим именования по умолчанию, который учитывает только вид объекта и может добавить порядковый номер, если в глобальном пространстве имен Promela есть несколько одинаковых имен.

Типы переменных в большинстве случаев транслируются тривиально. Здесь мы рассматриваем poST-программы без переменных действительных типов, поскольку в Promela нет типов, представляющих действительные числа, а абстракция типов данных выходит за рамки данной статьи. Трансляция типа TIME обсуждается ниже при описании трансляции выражения TIMEOUT.

В соответствии с политикой именования и трансляции типов, приведенными выше, все переименованные переменные poST объявляются в модели Promela как глобальные переменные, а константы poST тривиально транслируются с помощью директивы Promela `#define`.

Promela включает в себя те же арифметические и логические операции, что и poST, за исключением возведения в степень, которое можно смоделировать операциями побитового сдвига Promela.

3.2. Операторы потока управления

В таблице 1 мы приводим метод трансляции трех видов операторов потока управления poST в код Promela. Пусть *code′* — образ Promela для poST *code*, созданный нашим алгоритмом трансляции. В силу блокирующей семантики Promela и неблокирующей семантики poST для операторов потока управления, необходимо использовать ветку `else` и оператор `skip` при трансляции poST оператора IF, поскольку Promela-оператор `if` блокирует исполнение процесса, если условие *cond* ложно. По семантике Promela, ветвь `else` выбирается, когда другие условия в операторе `if` невыполнимы. Оператор `skip` в этой ветке не выполняет никаких действий. poST оператор CASE оценивает выражение *expr* (тип INT) и выполняет ветвь, соответствующую списку значений, содержащему результат вычисления. Списки значений не пересекаются. Promela-оператор `do` также является блокирующим, поэтому при трансляции poST оператора DO мы добавляем ветвь `else` с оператором `break` для моделирования завершения цикла в Promela.

3.3. Специфические операторы процесс-ориентированных программ

Система управления в poST задается набором poST-программ. Следовательно, нам необходимо перевести несколько poST-программ в единую Promela-модель. Программа на poST состоит из про-

Table 1. Control-Flow statements **Таблица 1.** Операторы потока управления

poST	Promela	poST	Promela

IF *cond* THEN
  body
END_IF
	if
	:: *cond′* -> { *body′* }
	:: else -> skip;
	fi;
		CASE *expr* OF
		  $list_1$: $body_1$
		...
		  $list_n$: $body_n$
		ELSE
		  body
		END_CASE
			int v = *expr′*;
			if
			:: v == l_{11}||...||v == l_{1m_1} ->
			  { $body'_1$ }
			...
			:: v == l_{n1}||...||v == l_{nm_n} ->
			  { $body'_n$ }
			:: else ->
			  { *body′* }
			fi;

WHILE *cond* DO *body* END_WHILE	do :: *cond′* -> { *body′* } :: else -> break; od;		

цессов, которые циклически активируются один за другим. Несколько poST-программ запускаются в едином цикле управления в порядке их появления в коде. В Promela мы моделируем этот циклический запуск, последовательно передавая *активирующее сообщение* от процесса к процессу через Promela-канал в том порядке, в котором poST-процессы появляются в коде исходных программ. Promela-каналы поддерживают блокировку чтения и записи. Мы используем канал cur емкостью 1 для передачи псевдонимов процессов в активирующих сообщениях. В результирующей Promela-программе каждый процесс заблокирован до тех пор, пока не прочтет свой псевдоним из этого канала. После выполнения своего тела процесс передает в канал псевдоним следующего процесса для его активации. Promela-процесс для последнего poST-процесса передаёт ход в сервисный процесс Gremlin, описанный в разделе 3.4, который является первым процессом результирующей модели Promela. Следуя семантике последовательной активации poST-программ мы используем Promela-оператор atomic для тела каждого транслируемого poST-процесса. Этот оператор уменьшает степень чередования исполнения в Promela-программе, что значительно упрощает верификацию. В левом блоке таблицы 2 представлена трансляция верхней структуры процессов poST-программ.

Тело poST-процесса состоит из *состояний*. На каждой итерации цикла управления poST-процесс выполняет код, соответствующий некоторым его состояниям, за исключением состояний STOP и ERROR, когда он ничего не делает. Для каждого транслируемого процесса n с Promela именем n' мы используем специальный счетчик состояний c_n', чтобы сохранить имя его текущего состояния. В начале программы poST её первый объявленный процесс находится в своем первом состоянии, а все остальные процессы находятся в состоянии STOP. В правом блоке таблицы 2 приводится трансляция отдельного poST-процесса в язык Promela.

poST-процессы могут проверять статус активности других процессов с помощью операторов ACTIVE и INACTIVE. Также каждый процесс может перевести себя или другой процесс в другое состояние с помощью операторов RESTART, STOP, START PROCESS, STOP PROCESS и других. Отметим, что имеет место инкапсуляция процессов в том смысле, что другой процесс можно только остановить или запустить, а доступа к отдельным именованным состояниям посторонние процессы не имеют. Перечисленные операторы poST тривиально транслируются в Promela, если целевое состояние не включает оператор TIMEOUT. См. примеры в левом блоке таблицы 3.

Последним оператором состояния poST-процесса может быть оператор TIMEOUT. Инструкции тела этого оператора выполняются по истечении указанного времени с момента перехода процесса в это состояние. Чтобы смоделировать такое поведение в Promela, мы вводим переменную счетчика времени t_n' — по одной на каждый процесс n, содержащий состояния с TIMEOUT. Если процесс

Table 2. Process-oriented features

Таблица 2. Процессно-ориентированные структуры

poST	Promela	poST	Promela

PROGRAM $prog_1$
PROCESS n_{11}
$body_{11}$
END_PROCESS
...
PROCESS n_{n1}
$body_{n1}$
END_PROCESS
END_PROGRAM
...
PROGRAM $prog_m$
PROCESS n_{1m}
$body_{1m}$
END_PROCESS
...
PROCESS n_{nm}
$body_{nm}$
END_PROCESS
END_PROGRAM
	mtype : P =
	{p_n'_{11}, ... p_n'_{nm}}
	chan cur=[1] of {mtype:P}
	active proctype n'_{11}() {
	do :: cur ? p_n'_{11} ->
	atomic {
	$body'_{11}$;
	cur ! p_n'_{21};}
	od;
	}
	active proctype n'_{12}() {
	do :: cur ? p_n'_{21} ->
	atomic {
	$body'_{12}$;
	cur ! p_n'_{31};}
	od;
	}
	...
	active proctype n'_{nm}() {
	do :: cur ? p_n'_{nm} ->
	atomic {
	$body'_{nm}$;
	cur ! Gremlin;}
	od;
	}
		PROCESS n
		STATE s_1
		$body_1$
		END_STATE
		...
		STATE s_m
		$body_m$
		END_STATE
		END_PROCESS
			mtype:S_n' = s_s'_1,...
			s_s'_n,s_$Stop'$,s_$Error'$}
			mtype:S_n' c_n';
			active proctype n'() {
			c_n' = s_$Stop'$;
			do :: cur ? p_n' ->
			atomic {
			if
			:: c_n'==s_s'_1 ->
			{ $body'_1$ }
			...
			:: c_n'==s_s'_n ->
			{ $body'_n$ }
			:: else -> skip;
			fi;
			cur ! nxt_p;}
			od;
			}

находится в состоянии с таймаутом, то в каждом цикле управления его счетчик времени увеличивается на единицу. Согласно семантике poST, этот счетчик времени обнуляется, когда (1) процесс сбрасывает таймер (RESET); (2) процесс переходит в другое состояние; и (3) происходит таймаут. Следуя семантике poST, мы используем отношение > в Promela-операторе `if` для таймаута, поскольку выполнение блока таймаута начинается в следующем цикле после истечения времени таймаута. Мы приводим типичные конструкции модели для таймаутов в правом блоке таблицы 3.

Чтобы уменьшить размер Promela-модели, мы проводим в ней следующую оптимизацию счетчиков времени. Во-первых, мы используем значение INTERVAL, которое задаёт физическое время исполнения цикла управления в poST, чтобы уменьшить все значения таймаута до ближайшего кратного этому интервалу. Во-вторых, мы делим все значения таймаутов poST-программы на их наибольший общий делитель. Кроме того, для счетчиков времени выбираем минимально достаточный размер *nb* беззнакового типа. Например, мы добавляем один счетчик времени размером 4 бита, если результирующий Promela-процесс имеет два состояния с таймаутами, отсчитывающими 5 (101b) и 9 (1001b) итераций цикла управления.

Table 3. State and Timeout Statements **Таблица 3.** Операторы состояний и таймаутов

poST	Promela	poST	Promela

IF (PROCESS n INACTIVE)
THEN $body$
END_IF
	if
	:: c_n' == s_$Stop'$ ||
	c_n' == s_$Error'$ ->{ $body'$ }
	:: else -> skip;
	fi;
		PROCESS n_1
		STATE s_1
		$body_1$
		START PROCESS n_2
		END_STATE
		...
		END_PROCESS
			active proctype n_1'(){
			...
			:: c_n_1' == s_s_1' -> {
			$body_1'$
			c_n_2' = s_2';
			t_n_2' = 0; }
			...
			}
			unsigned t_n_2' : nb

PROCESS n
STATE s_1
$body_1$
SET NEXT
END_STATE
STATE s_2
$body_2$
END_STATE
...
END_PROCESS
	active proctype n'() {
	do :: cur ? p_n' ->atomic {
	if
	:: c_n' == s_s_1' ->{ $body_1'$
	c_n' = s_s_2'; }
	:: c_n' == s_s_2' ->{ $body_2'$ }
	...
	:: else -> skip;
	fi;
	cur ! nxt_p;}
	od;
	}
		PROCESS n_2
		STATE s_2
		$body_2$
		TIMEOUT T#$tout$
		THEN $body_t$
		END_TIMEOUT
		END_STATE
		...
		END_PROCESS
			active proctype n_2'(){
			...
			:: c_n_2' == s_s_2' -> {
			if
			:: t_n_2' > $tout'$ ->$body_t'$
			:: else -> t_n_2'++;
			fi;}
			...
			}

3.4. Сервисные процессы

Мы вводим три специальных процесса Promela: процесс `BOC` для маркера начала цикла управления, процесс `Gremlin` для моделирования неопределенного поведения среды и процесс `OutInput` для корректного взаимодействия программ poST.

Требования к функционированию реактивных систем, определенные для poST-программ, часто естественным образом выражаются в терминах взаимосвязи между входными и выходными данными программы. Соответственно, проверка таких высокоуровневых требований происходит *в промежутке* между циклами управления системы, а программные процессы рассматриваются как черные ящики. Для обеспечения такой проверки мы вводим сервисный Promela-процесс `BOC`, который фиксирует начало цикла управления для проверки требований. Используя значение «импульсной» переменной `cycle_u` этого процесса, мы формулируем аналоги модальностей LTL — *циклические темпоральные операторы* — для высокоуровневых требований в Promela, как показано в таблице 4. Эти темпоральные операторы использует логика cycle-LTL, разработанная нами в [27].

Разберём, как используется импульсный сигнал `cycle_u` на примере оператора `G_cltl`. В [27] доказано, что $\mathbf{G}^{\mathbf{i}}\varphi \equiv \mathbf{G}(\mathit{Input} \rightarrow \varphi)$, где $\mathbf{G}^{\mathbf{i}}$ — циклический глобальный оператор, $\mathbf{G}$ — стандартный глобальный LTL-оператор и Input — булева переменная, истинная только в начале цикла управления. В Promela-коде таблицы 4 *Input* представлен импульсным сигналом `cycle_u`. Таким образом макрос `c_imp(expr)` кодирует импликацию $(\mathit{Input} \rightarrow \varphi)$, где φ соответствует `(expr)`. Поэтому при подстановке `c_imp(expr)` в `G_cltl(expr)` получаем: `G_cltl(expr) [] c_imp(expr) = [](cycle_u ->`

Table 4. High-level LTL modalities and service process BOC

Таблица 4. Высокоуровневые модальности LTL и сервисный процесс BOC

LTL модальности	Процесс BOC

`#define c_imp(expr) (cycle_u -> (expr))`
`#define c_and(expr) (cycle_u && (expr))`
`#define G_cltl(expr) [] c_imp(expr)`
`#define F_cltl(expr) <> c_and(expr)`
`#define U_cltl(expr1, expr2)`
`  c_imp(expr1) U c_and(expr2)`
`#define W_cltl(expr1, expr2)`
`  c_imp(expr1) W c_and(expr2)`
`#define V_cltl(expr1, expr2)`
`  c_and(expr1) V c_imp(expr2)`
`#define next_cltl(expr) (cycle_u ->`
`  (cycle_u U (!cycle_u W c_and(expr))))`
	`bool cycle_u;`
	`active proctype BOC () {`
	`  do :: current ? p_BOC ->`
	`  cycle_u = true;`
	`  atomic {`
	`  cycle_u = false;`
	`  current ! p_`$name_{11}$`; }`
	`  od;`
	`}`

`expr)`, что и даёт нужное выражение циклического темпорального оператора через обычный темпоральный LTL-оператор. Для остальных операторов рассуждения аналогичны.

Для взаимодействия процессов с окружением в poST-программах объявляются переменные INPUT, OUTPUT и IN_OUT. Процессы в poST могут включать переменные INPUT, не связанные с переменными OUTPUT и IN_OUT других процессов. Такие переменные реализуют входы от внешней среды, поведение которой считается неопределенным. Мы моделируем неопределенное поведение среды с помощью процесса `Gremlin`, представленного в левом блоке таблицы 5 сверху. При этом в силу ресурсоемкости верификации моделей в SPIN, рассматриваются только переменные «небольших» типов BOOL, USINT и SINT. Более подробно идея спецификации неопределенного поведения среды изложена в разделе 4.3.

Программы poST взаимодействуют через переменные с одинаковыми именами. Политика именования при трансляции такова, что эти переменные имеют разные имена в результирующей Promela-модели. Поэтому необходимо явно обновлять входы одной программы выходами другой программы между итерациями цикла управления с помощью Promela-процесса `OutInput`, который приведён в левом блоке Таблицы 5 снизу.

3.5. Общая модель Promela для программ poST

В общем случае, наш алгоритм трансляции принимает на вход несколько poST-программ, описывающих систему управления в одном файле. Эта система управления может включать в себя алгоритм управления и его окружение: управляемые и неуправляемые объекты. В правом блоке таблицы 5 представлена результирующая Promela-модель, включающая три сервисных процесса и процессы, соответствующие исходным poST-процессам. Деятельность этих процессов образует цикл управления посредством передачи между ними активирующих сообщений, начиная с недетерминированного процесса `Gremlin`, моделирующего неконтролируемый объект с неопределенным поведением, и заканчивая результатом трансляции последнего poST-процесса, который снова передает активирующее сообщение процессу `Gremlin`. Внутри цикла активность процессов упорядочена, как описано в таблицах 2 и 5.

Table 5. Service processes and the Promela model for a poST program

Таблица 5. Сервисные процессы и модель Promela для программы poST

poST	Promela	poST	Promela

PROCESS n
VAR_INPUT
v_b : BOOL;
v_u : USINT;
v_s : SINT;
...
END_VAR
...
END_PROCESS
	active proctype Gremlin(){
	do :: cur ? p_Gremlin ->
	atomic {
	if
	:: v'_b = true;
	:: v'_b = false;
	fi;
	select (v'_u: 0..255);
	select (v'_s: -128..127);
	cur ! p_OutInput;}
	od;
	}
		PROGRAM $prog_1$
		Var_Decl_1
		PROCESS n_{11}
		...
		PROCESS n_{n1}
		END_PROGRAM
		...
		PROGRAM $prog_m$
		Var_Decl_m
		PROCESS n_{1m}
		...
		PROCESS n_{nm}
		END_PROGRAM
			$Var\_Decl'_1$
			...
			$Var\_Decl'_m$
			$Service\_Decl$
			init{
			cur ! p_Gremlin;}
			active proctype
			Gremlin(){...}
			active proctype
			OutInput(){...}
			active proctype
			BOC(){...}
			active proctype
			$name'_{11}$(){...}
			...
			active proctype
			$name'_{nm}$(){
			do :: cur ? p_n'_{nm} ->
			atomic {
			...
			cur ! p_Gremlin;
			}
			od;
			}

PROGRAM n_1
VAR_OUTPUT
var : $type$;
END_VAR
...
END_PROGRAM
...
PROGRAM n_2
VAR_INPUT
var : $type$;
END_VAR
...
END_PROGRAM
	active proctype OutInput(){
	do :: cur ? p_OutInput ->
	atomic {
	$var'(n_2)$ = $var'(n_1)$;
	...
	cur ! p_BOC;}
	od;
	}		

4. Реализация транслятора poST в Promela

4.1. Общий подход к реализации

Для реализации нашего транслятора мы следуем ранее заданным архитектурным подходам к проектированию трансляторов языка poST. В частности, мы используем фреймворк Xtext[3] [28] в качестве базового инструмента для разработки парсеров. При этом код описанной трансформационной семантики реализован на языке Xtend[4], который в дальнейшем преобразуется в Java с помощью Eclipse IDE для DSL разработчиков[5]. Мы используем этот язык прежде всего потому, что на нем удобно описывать шаблоны генерации языковых конструкций, параметризованных с использованием подготовленного контекста в объектах. Что касается модели ввода, мы используем представление программы poST в виде EMF объектов [29], которые естественным образом обрабатываются в Xtend. Парсинг внутри Xtend проектов осуществляется с использованием сред-

[3] http://eclipse.org/Xtext
[4] https://www.eclipse.org/xtend
[5] https://www.eclipse.org/downloads/packages/release/2022-12/r/eclipse-ide-java-and-dsl-developers

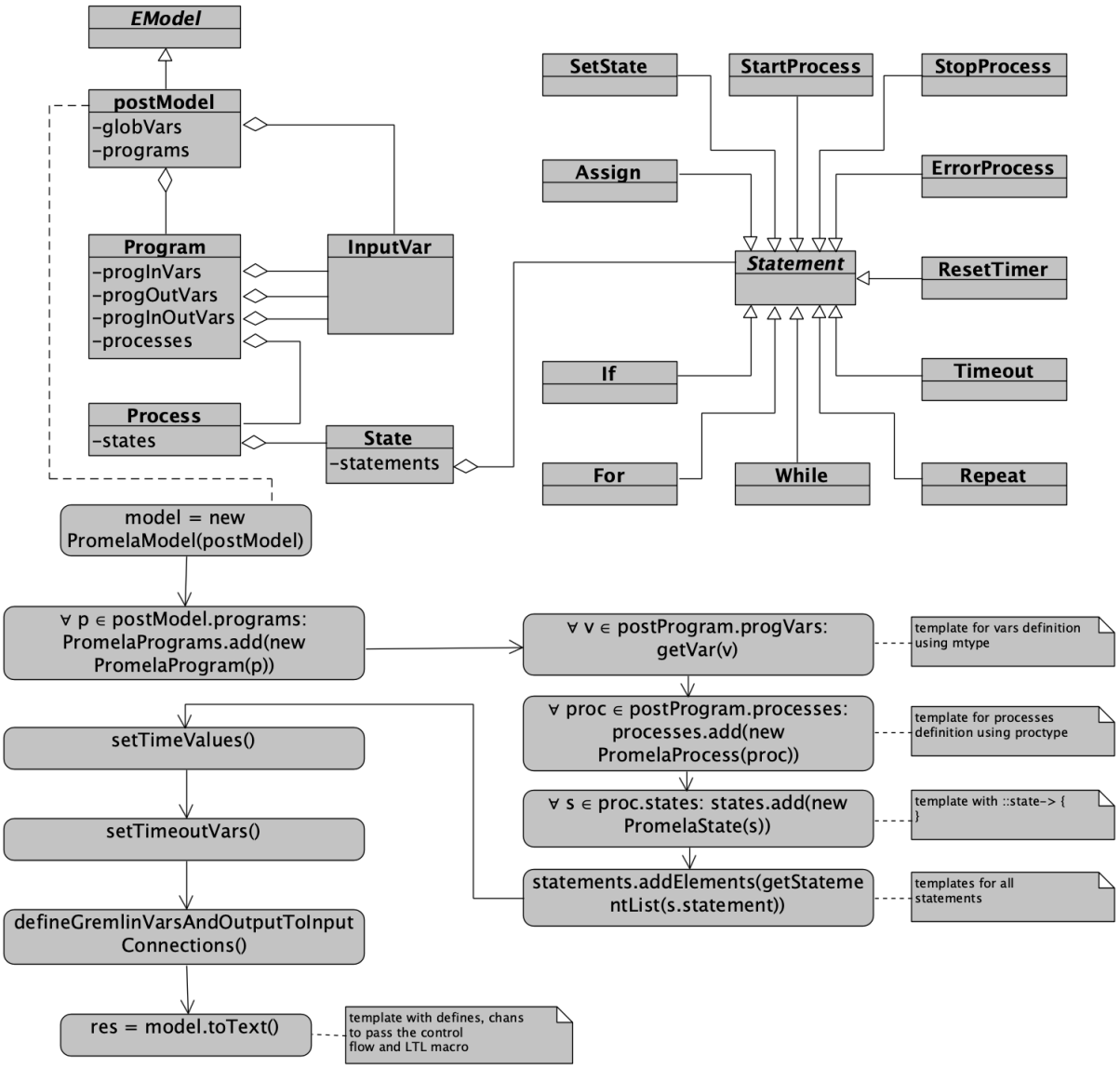


Fig. 1. Implementation of the translation of poST statements to Promela

Рис. 1. Реализация трансляции выражений poST в язык Promela

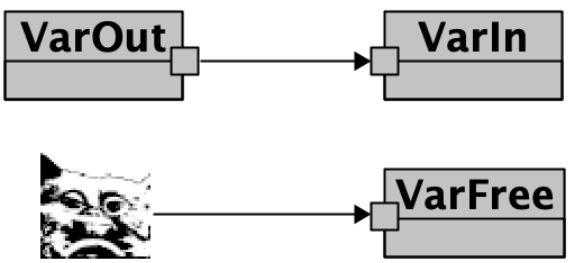


Fig. 2. A normal and a gremlin-based variable binding

Рис. 2. Связывание переменных стандартное и с гремлином

ства ANTLR [30], но мы работаем с объектами языка и их связями на высоком уровне. Обсуждаемый транслятор представляет собой JVM-приложение, которое запускается из командной строки.

4.2. Детали реализации

Рассмотрим, каким именно образом мы получаем код на языке Promela из poST-программы. В верхней части рис. 1 представлены EMF объекты программы на poST. В соответствии с процесс-ориентированной структурой модель, полученная из исходного poST-кода, определяет несколько программ с процессами (например, Plant и Controller). Процессы этих программ взаимодействуют через значения входных и выходных переменных, которыми они обмениваются по завершении цикла управления. Сам процесс является конечной системой переходов, при этом каждое состояние представлено атомарной последовательностью действий, которая определяется операторами языка poST (на рисунке показаны основные из них). Задача транслятора — корректно перевести операторы, описания процессов, переменных и т. п. языка poST в код на языке Promela, согласно семантике, определенной таблицах раздела 3.

Транслятор (в нижней части рис. 1) итеративно перебирает структуры данных процессов, переменных, состояний и операторов исходного кода poST для создания соответствующих языковых конструкций в Promela. Данные о текущем процессе, состоянии и т. п. сохраняются в текущем контексте. Он используется в шаблонах, которые описываются в виде конструкций языка Xtend, где удобно работать со строками. Приведём пример такого шаблона для процесса на poST:

```
>>’
<<IF !vars.isEmpty()>>
<<vars.toText()>>
<<ENDIF>>

active proctype <<Context.getName(name)>>()
{
  do ::
  <<Context.getName(__currentProc)>> ?
  <<Context.getName(nameMType)>> ->
    atomic {
    if
      <<states.toText()>>
      :: else -> skip;
    fi;
     <<Context.getName(__currentProc)>> !
     <<Context.getName(nextMType)>>;
    }
  od;
}
>>’
```

Во французских угловых кавычках здесь находится код в Xtext, который выполняется и подставляет результат в шаблон. Сначала генерируются переменные процесса. Затем объявляется процесс с именем, порожденным согласно вышеопределенной политике именования. Для порождённого таким образом процесса генерируется (1) получение активирующего сообщения по ранее сгенерированному каналу, (2) тело процесса и (3) передача хода следующему процессу, информация о котором предварительно была получена из poST-программы.

Отметим на рис. 1 следующие функции трансляции реализующие работу с таймаутами, которые позволяют уменьшить размер результирующей модели (см. конец раздела 3.3). Первый способ редуцирования размера модели состоит в сокращении количества итераций цикла при работе с таймаутами: функция *setTimeValues* корректирует значения таймаутов процессов, разделив их на НОД (аналогичная техника применяется в [31]). Второй способ редуцирования уменьшает размер необходимой памяти для хранения пространства состояний во время проверки модели: функция *setTimeoutVars* находит в выражениях все заданные значения таймаута в стиле ST[6] и вычисляет количество битов для минимизации их представления, используя беззнаковый тип данных Promela[7].

Функция трансляции *defineGremlinVarsAndOutputToInputConnections()* вставляет служебные процессы обработки переменных из таб. 5 в список процессов. Процесс работы с переменными активируется после очередного цикла управления. Этот процесс присваивает значения выходных переменных значениям тех входных переменных, для которых определена такая привязка. Переменные, которые не имеют программных связей с выходными переменными, считаются переменными, значения которых задаёт окружение программируемой системы управления, поведение которого недетерминировано. Это окружение моделируется с помощью подхода *«гремлин»*.

4.3. Недетерминизм на основе подхода «гремлин»

Алгоритм управления обычно взаимодействует с внешним устройством, которое может иметь сложную логику изменения переменных в результате действия физических законов. Однако не всегда необходимо знать и моделировать эту логику, чтобы обеспечить верификацию заданных требований при активации определенных структур в управляющем коде. Для моделирования взаимодействия алгоритма управления и такой внешней среды, мы используем *гремлина*, который представляет собой нерассуждающую силу, способную непредсказуемым образом изменять значения входных переменных перед выполнением следующего цикла управления (подобно гремлинам из знаменитого фильма Джо Данте и Стивена Спилберга, которые буянили в магазине, хватая и бросая всё, что попадалось на глаза). Насколько нам известно, такой гремлин-подход впервые был использован при тестировании пользовательского интерфейса программ Palm OS [32]: здесь гремлины активируются в произвольных местах экрана, обеспечивая рандомизированное тестирование и огромное количество вариантов переходов между экранными формами приложения.

Если входная переменная в программе poST не связана с какой-либо другой переменной (см. рис. 2), считается, что она может принимать произвольное значение из диапазона типа переменной. При моделировании это означает, что переменной будет присвоено случайное значение, а при верификации — что переменной будут присвоены все возможные значения. Такой подход позволяет получить представление обо всех возможных случаях входных данных для работы системы управления. Развитием этого подхода является создание ограниченных гремлинов посредством установки условий на свободные переменные.

5. Пример трансляции

Рассмотрим задачу моделирования потребления и производства электроэнергии, в том числе из возобновляемых источников. Компонентами системы являются:

- **солнце (sun)**: днем может светить, а может и не светить из-за появления облаков; ночью не светит;
- **солнечная панель (solar Panel)**: производит одну единицу энергии в цикл, если светит солнце;

[6] См., например, https://infosys.beckhoff.com/english.php?content=../content/1033/tc3_plc_intro/2528280971.html.
[7] http://spinroot.com/spin/Man/datatypes.html

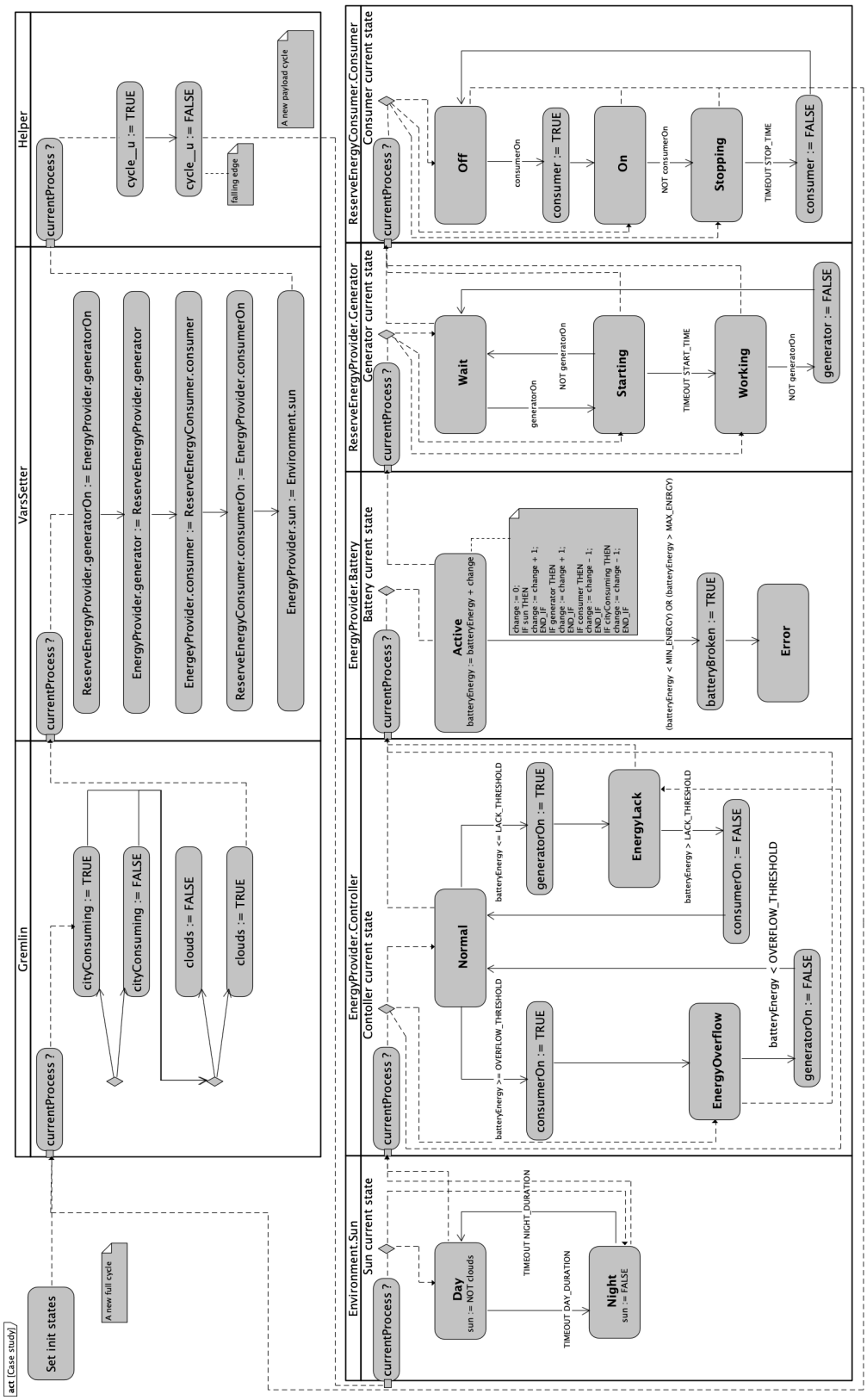


Fig. 3. UML modeling of the case study

Рис. 3. UML-модель иллюстративного примера

- **батарея (battery)**: накапливает определенное количество единиц энергии; если единиц энергии мало или много, то батарея выходит из строя (случай ошибки);
- **потребитель (consumer)**: может потреблять или не потреблять одну единицу энергии в цикл;
- **нагрузка (load)**: можно включить для рассеивания лишней энергии, если светит солнце и город не потребляет энергию;
- **генератор (generator)**: может включаться при недостатке электроэнергии, когда не светит солнце и потребитель потребляет энергию;
- **контроллер (controller)**: включает и выключает генератор и нагрузку, чтобы поддерживать заряд батареи в допустимых пределах.

UML-моделирование работы системы в процесс-ориентированной парадигме показано на рис. 3. В данном случае используется диаграмма деятельности, предназначенная для отображения нескольких процессов с их системами переходов [33]. Однако здесь нет процессов, исполняющихся параллельно: видно, что в самом начале каждый процесс ждёт активирующего сообщения в свой канал («ожидание очереди»), чтобы продолжить работу в своём текущем состоянии. Такое исполнение в системе процессов аналогично циклическому планированию с состояниями и заданной последовательностью процессов. Пунктирными линиями мы отметили необходимый поток управления средой исполнения (в соответствии с заданной семантикой), а простыми линиями отмечаем переходы по логике системы, описанной в poST-программе.

Работа системы является циклической. Получив ход, процесс предпринимает свои действия исходя из текущего состояния и передает ход следующему. При этом вначале действуют три сгенерированных служебных процесса: (1) для установки значений свободных переменных (здесь: гремлин для переменных `cityConsuming` и `clouds`); (2) для установки значения входных переменных в соответствии с заданными выходными переменными (здесь: согласование значений пяти переменных, которые задаются одними процессами и должны быть доступны другим после завершения итерации цикла управления); и (3) установка импульса начала цикла управления, который используется в спецификациях требований.

Рассмотрим наш метод трансляции на одном из процессов примера, а именно на процессе `Consumer`. Пусть его исходный код в poST выглядит следующим образом:

```
PROCESS Consumer
  VAR CONSTANT
    STOP_TIME : TIME := T#1h;
  END_VAR
  STATE Off
    IF consumerOn THEN
      consumer := TRUE;
      SET STATE On;
    END_IF
  END_STATE
  STATE On
    IF NOT consumerOn THEN
      SET STATE Stopping;
    END_IF
  END_STATE
  STATE Stopping
    TIMEOUT STOP_TIME THEN
      consumer := FALSE;
```

```
      SET STATE Off;
    END_TIMEOUT
  END_STATE
END_PROCESS
```

Результат трансляции этого процесса в код на языке Promela показан ниже:

```
#define STOP_TIME 3600000 //1h
active proctype Consumer() {
do :: __currentProcess ? Consumer__p ->
 atomic {
 if :: Off__s == Consumer__cs ->
        if :: consumerOn__0 ->
                consumer__1 = true;
                Consumer__cs = On__s;
           :: else -> skip;
        fi;
    :: On__s == Consumer__cs ->
        if :: !consumerOn__0 ->
                Consumer__t = 1;
                Consumer__cs = Stopping__s;
           :: else -> skip;
        fi;
    :: Stopping__s == Consumer__cs ->
        if :: Consumer__t > STOP_TIME ->
                consumer__1 = false;
                Consumer__cs = Off__s;
           :: else ->
                Consumer__t = Consumer__t + 1;
        fi;
    :: else -> skip;
  fi;
  __currentProcess ! Gremlin__sp;
 }
od;}
```

В результирующем процессе объявлен бесконечный цикл, в начале которого процесс ждёт активирующее сообщение в канале передачи активирующих сообщений. Сгенерированные имена состояний процесса `Off__s`, `On__s` и `Stopping__s` являются ключами при выборе текущего состояния процесса `Consumer__cs`. `Consumer__1` — это сгенерированное Promela-имя переменной `Consumer` в poST с добавленным номером, поскольку в других программах в модели тоже есть такая переменная. Эта переменная будет также участвовать в обмене значениями в начале цикла управления в сгенерированном процессе `OutInput`, как описано в разделе 3.4. Наконец, генерируется переменная таймера `Consumer__t` для подсчета времени, в течение которого процесс находится в этом состоянии, чтобы реализовать переход в результате таймаута. В конце тела процесса ход передается процессу следующему по схеме.

В терминах логики линейного времени LTL с использованием циклического темпорального оператора `G_cltl` (раздел 3.4) мы сформулировали следующие требования:

1. `G_cltl (!batteryBroken)` — означает, что батарея никогда не сломается.
2. `G_cltl (!(generatorOn && ConsumerOn))` — означает, что потребитель и генератор никогда не работают одновременно.

Исходная poST-программа и ее представление в Promela доступны в нашем репозитории[8].

Заключение

В статье мы рассмотрели правила трансляции процесс-ориентированного языка poST в язык формальной верификации Promela. Таким образом, наша трансляция является транспиляцией, которая оправдана для языков программирования ПЛК с небольшим числом операторов, поскольку все конструкции входного языка могут быть преобразованы в конструкции языка, для которых уже хорошо разработаны методы формальной верификации (в нашем случае мы используем метод проверки моделей и инструмент верификации SPIN). Автоматическая транспиляция из процесс-ориентированного языка poST позволяет писать верифицируемые программы для ПЛК, не кодируя на Promela повторяющиеся языковые конструкции, связанные с семантикой переключения процессов, переходом по состояниям, обменом значениями переменных и генерацией импульса для верификации свойств для циклов управления. Проект находится в публичном доступе в репозитории[9]. Среди недостатков текущего подхода к трансляции можно отметить неполную поддержку типов данных. В частности, не поддерживаются переменные, принимающие действительные значения. Наш подход можно развить, реализуя библиотеки арифметики как с фиксированной, так и с плавающей запятой, однако это повлечет за собой очень большое количество состояний модели и верификацию результирующих программ затруднительно будет осуществить без подходящих методов абстрагирования. Также мы не реализовывали библиотечные функции poST. Однако разработка транспилятора poST2Promela является значительным шагом в построении инструментальной цепочки создания верифицируемых процесс-ориентированных программ.

References

[1] IEC, *IEC 61131-3: 2013 Programmable controllers-Part 3: Programming languages*, https://webstore.iec.ch/publication/4552, International Standard, 2013.

[2] T. M. Antonsen, *PLC Controls with Structured Text (ST), V3: IEC 61131-3 and best practice ST programming*. Books on Demand, 2020, 208 pp.

[3] V. E. Zyubin, “Hyper-automaton: A model of control algorithms”, in *2007 Siberian Conference on Control and Communications*, 2007, pp. 51–57. DOI: 10.1109/SIBCON.2007.371297.

[4] V. E. Zyubin, A. S. Rozov, I. S. Anureev, N. O. Garanina, and V. Vyatkin, “poST: A process-oriented extension of the IEC 61131-3 structured text language”, *IEEE Access*, vol. 10, pp. 35 238–35 250, 2022. DOI: 10.1109/ACCESS.2022.3157601.

[5] G. J. Holzmann, *The SPIN Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003, 608 pp.

[6] L. Schneider and D. Schultes, “Evaluating Swift-to-Kotlin and Kotlin-to-Swift transpilers”, in *Proceedings of the 9th IEEE/ACM International Conference on Mobile Software Engineering and Systems*, 2022, pp. 102–106.

[7] E. M. Clarke, T. A. Henzinger, H. Veith, R. Bloem, *et al.*, *Handbook of model checking*. Springer, 2018, 1212 pp.

[8] https://github.com/SergeyStaroletov/poST_to_Promela_compiler_dev/tree/main/samples
[9] https://github.com/SergeyStaroletov/poST_to_Promela_compiler_dev

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

[9] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 8, no. 2, pp. 244–263, 1986.

[10] C. A. R. Hoare, "Communicating sequential processes", *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[11] N. O. Garanina, S. M. Staroletov, V. E. Zyubin, and I. S. Anureev, "Model checking programs in process-oriented IEC 61131-3 structured text", *System Informatics*, vol. 22, pp. 21–30, 2023. DOI: 10.31144/si.2307-6410.2023.n22.p21-30.

[12] M. Weißmann, S. Bedenk, C. Buckl, and A. Knoll, "Model checking industrial robot systems", in *Model Checking Software. SPIN 2011*, Springer, 2011, pp. 161–176.

[13] P. Cousot, "Abstract interpretation", *ACM Computing Surveys*, vol. 28, no. 2, pp. 324–328, 1996. DOI: 10.1145/234528.234740.

[14] S. Leue and G. Holzmann, "v-Promela: A visual, object-oriented language for SPIN", in *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, IEEE, 1999, pp. 14–23.

[15] M. Benerecetti *et al.*, "From dynamic state machines to Promela", in *International Symposium on Model Checking Software*, Springer, 2019, pp. 56–73.

[16] B. Lion, S. Chouali, and F. Arbab, "Compiling protocols to Promela and verifying their LTL properties", in *Proceedings of 21st MODELS Workshops*, 2018, pp. 31–39.

[17] A. Klarl, "From Helena ensemble specifications to Promela verification models", in *Model Checking Software. SPIN 2015*, Springer, 2015, pp. 39–45.

[18] N. Dilley and J. Lange, "Bounded verification of message-passing concurrency in Go using Promela and SPIN", in *Proceedings of the 12th International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software*, EPTCS, 2010, pp. 34–45. DOI: 10.4204/EPTCS.314.4.

[19] N. Dilley and J. Lange, "Automated verification of Go programs via bounded model checking", in *36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2021, pp. 1016–1027.

[20] E. Brinksma, A. Mader, and A. Fehnker, "Verification and optimization of a PLC control schedule", *International Journal on Software Tools for Technology Transfer*, vol. 4, no. 1, pp. 21–33, 2002. DOI: 10.1007/s10009-002-0079-0.

[21] M. Xia, M. Sun, G. Luo, and X. Zhao, "Design and implementation of automatic verification for PLC systems", in *IEEE 12th International Conference on Cognitive Informatics and Cognitive Computing*, IEEE, 2013, pp. 374–379.

[22] P. Liu, G. Luo, M. Xia, and M. He, "Automatic verification of event-driven control programs: A case study", in *The Fourth International Workshop on Advanced Computational Intelligence*, IEEE, 2011, pp. 249–256.

[23] T. Liakh, R. Sorokin, D. Akifev, S. Patil, and V. Vyatkin, "Formal model of IEC 61499 execution trace in FBME IDE", in *IEEE 20th International Conference on Industrial Informatics (INDIN)*, IEEE, 2022, pp. 588–593.

[24] V. Shatrov and V. Vyatkin, "Promela formal modelling and verification of IEC 61499 systems with comparison to SMV", in *IEEE 19th International Conference on Industrial Informatics (INDIN)*, IEEE, 2021, pp. 1–6.

[25] A. Ebnenasir, "Formalizing ladder logic programs and timing charts for fault impact analysis and verification of fault tolerance", Michigan Technological University, Tech. Rep. CS-TR-23-01, Jan. 2023.

[26] A. Mader, "A classification of PLC models and applications", *Discrete Event Systems: Analysis and Control*, vol. 569, pp. 239–246, 2000.

[27] N. O. Garanina *et al.*, "A temporal logic for programmable logic controllers", *Automatic Control and Computer Sciences*, vol. 55, no. 7, pp. 763–775, 2021. DOI: 10.3103/S0146411621070038.

[28] M. Eysholdt and H. Behrens, "Xtext: Implement your language faster than the quick and dirty way", in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, 2010, pp. 307–309.

[29] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse modeling framework*. Pearson Education, 2008, 744 pp.

[30] "ANTLR: A predicated-LL(k) parser generator", *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[31] D. Lepri, E. Ábrahám, and P. C. Ölveczky, "Sound and complete timed CTL model checking of timed Kripke structures and real-time rewrite theories", *Science of Computer Programming*, vol. 99, pp. 128–192, 2015.

[32] N. Rhodes and J. McKeehan, *Palm OS programming: the developer's guide*. O'Reilly Media, Inc., 2002, 681 pp.

[33] R. Miles and K. Hamilton, *Learning UML 2.0: a pragmatic introduction to UML*. O'Reilly Media, Inc., 2006, 290 pp.

MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 31, NO. 1, 2024
journal homepage: www.mais-journal.ru

THEORY OF COMPUTING

On the Application of the Calculus of Positively Constructed Formulas for the Study of Controlled Discrete-Event Systems

A. V. Davydov[1], A. A. Larionov[1], N. V. Nagul[1]

DOI: 10.18255/1818-1015-2024-1-54-77

[1]Matrosov Institute for System Dynamics and Control Theory SB RAS, Irkutsk, Russia

MSC2020: 68V15, 93C65
Research article
Full text in Russian

Received February 2, 2024
Revised February 8, 2024
Accepted February 14, 2024

The article is devoted to the development of an approach to solving the main problems of the theory of supervisory control of logical discrete-event systems (DES), based on their representation in the form of positively constructed formulas (PCF). We consider logical DESs in automata form, understood as generators of some regular languages. The PCF language is a complete first-order language, the formulas of which have a regular structure of alternating type quantifiers and do not contain a negation operator in the syntax. It was previously proven that any formula of the classical first-order predicate calculus can be represented as a PCF. PCFs have a visual tree representation and a natural question-and-answer procedure for searching for an inference using a single inference rule. It is shown how the PCF calculus, developed in the 1990s to solve some problems of control of dynamic systems, makes it possible to solve basic problems of the theory of supervisory control, such as checking the criteria for the existence of supervisory control, automatically modifying restrictions on the behavior of the controlled system, and implementing a supervisor. Due to some features of the PCF calculus, it is possible to use a non-monotonic inference. It is demonstrated how the presented PCF-based method allows for additional event processing during inference. The Bootfrost software system, or the so-called prover, designed to refute the obtained PCFs is also presented, and the features of its implementation are briefly described. As an illustrative example, we consider the problem of controlling an autonomous mobile robot.

Keywords: positively constructed formula; automated theorem proving; prover; discrete event system; supervisory control

INFORMATION ABOUT THE AUTHORS

Davydov, Artem V.	ORCID iD: 0000-0002-8703-3096. E-mail: artem@icc.ru Research fellow
Larionov, Aleksandr A.	ORCID iD: 0000-0001-7116-9338. E-mail: bootfrost@zoho.com Programmer
Nagul, Nadezhda V. (corresponding author)	ORCID iD: 0000-0003-1439-3274. E-mail: sapling@icc.ru Senior researcher, PhD

Funding: Ministry of Science and Higher Education of the Russian Federation (121032400051-9).

For citation: A. V. Davydov, A. A. Larionov, and N. V. Nagul, "On the application of the calculus of positively constructed formulas for the study of controlled discrete-event systems", *Modeling and Analysis of Information Systems*, vol. 31, no. 1, pp. 54–77, 2024. DOI: 10.18255/1818-1015-2024-1-54-77.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 31, № 1, 2024
сайт журнала: www.mais-journal.ru

THEORY OF COMPUTING

О применении исчисления позитивно-образованных формул для исследования управляемых дискретно-событийных систем

А. В. Давыдов[1], А. А. Ларионов[1], Н. В. Нагул[1]

DOI: 10.18255/1818-1015-2024-1-54-77

[1]Институт динамики систем и теории управления им. В.М. Матросова СО РАН, Иркутск, Россия

УДК 004.832.3
Научная статья
Полный текст на русском языке

Получена 2 февраля 2024 г.
После доработки 8 февраля 2024 г.
Принята к публикации 14 февраля 2024 г.

Статья посвящена разработке подхода к решению основных задач теории супервизорного управления логическими дискретно-событийными системами (ДСС), основанного на представлении их в виде позитивно-образованных формул (ПОФ). Рассматриваются логические ДСС в автоматной форме, понимаемые как генераторы некоторых регулярных языков. Язык ПОФ представляет собой полный язык первого порядка, формулы которого имеют регулярную структуру из чередующихся типовых кванторов и не содержат в синтаксисе оператора отрицания. Ранее было доказано, что любая формула классического исчисления предикатов первого порядка может быть представлена в виде ПОФ. ПОФ имеют наглядное древовидное представление и естественную вопросно-ответную процедуру поиска вывода с помощью единственного правила вывода. Показано, как разработанное в 1990-х годах для решения некоторых задач управления динамическими системами исчисление ПОФ позволяет решать базовые задачи теории супервизорного управления, такие как проверка критериев существования супервизорного управления, автоматическая модификация ограничений на поведение управляемой системы и реализация супервизора. Благодаря некоторым особенностям исчисления ПОФ существует возможность применения немонотонного вывода. Продемонстрировано, как представленный метод на основе ПОФ позволяет выполнять дополнительную обработку событий во время логического вывода. Также представлена программная система Bootfrost, или так называемый прувер, разработанный для опровержения полученных ПОФ, кратко описываются особенности его реализации. В качестве иллюстративного примера рассматривается задача управления автономным мобильным роботом.

Ключевые слова: позитивно-образованная формула; автоматическое доказательство теорем; прувер; дискретно-событийная система; супервизорное управление

ИНФОРМАЦИЯ ОБ АВТОРАХ

Давыдов, Артем Васильевич	ORCID iD: 0000-0002-8703-3096. E-mail: artem@icc.ru Научный сотрудник
Ларионов, Александр Александрович	ORCID iD: 0000-0001-7116-9338. E-mail: bootfrost@zoho.com Программист
Нагул, Надежда Владимировна (автор для корреспонденции)	ORCID iD: 0000-0003-1439-3274. E-mail: sapling@icc.ru Старший научный сотрудник, к.ф.-м.н.

Финансирование: Министерство науки и высшего образования РФ (121032400051-9).

Для цитирования: A. V. Davydov, A. A. Larionov, and N. V. Nagul, “On the application of the calculus of positively constructed formulas for the study of controlled discrete-event systems”, *Modeling and Analysis of Information Systems*, vol. 31, no. 1, pp. 54–77, 2024. DOI: 10.18255/1818-1015-2024-1-54-77.

Введение

Класс дискретно-событийных систем (ДСС) — это широкий класс моделей, как правило, используемых для представления сложных технических и технологических систем, таких как производственные и сборочные процессы, коммуникационные протоколы, транспортные системы, системы управления базами данных, в которых смена дискретных состояний системы происходит вследствие возникновения некоторых дискретных событий [1—3]. Для описания таких систем используются различные формализмы, которые могут включать или не включать явное указание времени, например, конечные автоматы, сети Петри, диоидные алгебры, темпоральные логики и др. Если важна только последовательность событий, а не время их возникновения и длительность, говорят о логических ДСС. Два формализма наиболее часто применяются для исследования логических ДСС: конечные автоматы, являющиеся самыми наглядными моделями, подверженными, однако, проклятию размерности, и сети Петри, обладающие компактностью представления и достаточной выразительностью.

В случае, когда некоторые события системы могут быть запрещены, возникает возможность ограничить поведение системы в пределах, заданных определенной спецификацией. Для исследования и построения управляемых логических ДСС была разработана теория супервизорного управления (ТСУ) [4], современное состояние которой представлено, например, в [3, 5, 6]. Соответствующее внешнее средство управления называется супервизором, а система, следуя теории автоматического управления, — объектом управления. Например, логические ДСС и ТСУ в настоящее время широко используются в робототехнике. Публикации в этой области касаются управления одиночными роботами [7, 8], группами роботов [9—11], формациями роботов [12] и их роями [13, 14]. Несмотря на интенсивные исследования, проводимые с 1980-х годов, многие задачи ТСУ ожидают новых подходов к их решению, что связано со сложностью реализации известных алгоритмов исследования и построения ДСС, особенно частично наблюдаемых, децентрализованных или распределенных.

В представлении ДСС с помощью конечного автомата переходы из состояния в состояние помечены буквами некоторого конечного алфавита и соответствуют событиям, происходящим в системе. Последовательности таких переходов образуют слова регулярного языка, описывающего поведение системы с высокоуровневой, или символьной, точки зрения. Свойства системы, такие как живучесть, безопасность, неблокирование и др., могут быть описаны как утверждения над этими формальными выражениями, а для проверки истинности таких утверждений оказывается возможным использовать автоматическое доказательство теорем (АДТ). Разработанное для проверки формальных математических доказательств и реализованное с помощью специальных компьютерных программ, называемых пруверами, АДТ в настоящее время применяется в широком ряде областей, включая анализ программ, верификацию систем и др. Последние примеры включают доказательство 400-летней гипотезы Кеплера об упаковке сфер [15] и корректность ядра операционной системы seL4 [16], не говоря уже о более раннем доказательстве теоремы о четырёх красках в теории графов [17] и верификацию компилятора языка C CompCert [18]. В [19] АДТ применяется для проверки закодированных специальным образом условий верификации (verification conditions) с помощью системы ACL2 ([20]). Важной областью современного применения АДТ является робототехника, где с его помощью осуществляется планирование [21] и принятие решений [22]. Например, в [23] для планирования и управления роем роботов используется язык PDDL, основанный на классическом АДТ в стиле STRIPS. В [24] доказательство теорем предлагается использовать для проверки структуры моделирования контроллеров автономных роботов в сочетании с автоматической генерацией кода на C++.

Как известно, использование АДТ сопряжено с рядом известных трудностей, таких как критическое увеличение размера базы фактов, с которыми работает прувер, или следование по ложному

пути при поиске вывода. Для преодоления этих трудностей в 1990-е годы академиком С. Н. Васильевым и А. К. Жерловым было разработано исчисление позитивно-образованных формул (ПОФ) и основанный на нем метод АДТ [25, 26]. ПОФ — это формулы первого порядка, имеющие регулярную структуру и не содержащие символа отрицания. Метод АДТ на основе ПОФ представляет собой мощный инструмент качественного анализа систем, прежде всего, динамических, а приложения включают ориентацию телескопа, управление группой лифтов [26], обеспечение достижимости множества целей [27], преследование подвижных целей [28]. Для автоматизации логического вывода в языке ПОФ разработан прувер Bootfrost[1]. Наиболее важными особенностями исчисления ПОФ и его программной реализации являются следующие:

- крупноблочные структуры данных для представления формул и правил вывода;
- отсутствие необходимости удаления кванторов существования с помощью процедуры сколемизации, что снижает сложность поиска вывода;
- совместимость с эвристиками, специфичными для приложения, и общими эвристиками управления поиском логического вывода;
- наглядность логического вывода, что помогает найти ошибки формализации;
- возможность модификации семантики для поддержки неклассических логик.

Помимо других преимуществ, прувер Bootfrost сочетает в себе возможности полностью автоматического вывода с возможностью реализации разработанных пользователем стратегий вывода.

Принимая во внимание известные примеры применения АДТ в исчислении ПОФ к управлению динамическими системами, нами предложен новый способ исследования и проектирования логических ДСС, основанный на ПОФ. Предлагаемый метод может быть использован на различных этапах построения системы управления, включая этап проверки спецификации на управляемость и наблюдаемость, в случае частичного наблюдения событий в системе, модификацию спецификации для обеспечения управляемости, если свойство управляемости не выполнено, реализацию управления на верхнем символьном уровне. Представление ДСС с помощью ПОФ позволяет использовать в процессе построения логического вывода информацию, поступающую из окружающей среды, а также данные о функционировании самой системы. Эта функция может существенно помочь в задачах синтеза и реализации супервизоров для программируемых логических контроллеров (ПЛК) [29], где используются расширенные автоматы с конечным числом состояний (РКА). Переходы в РКА могут содержать условия (например, логические условия) на переменные и действия (например, обновление) на переменные. В ПОФ условия переходов могут быть реализованы с помощью вопросов обработки событий, служащих специальными логическими правилами. Ответы на них запускают подвыводы, в которых события, поступающие из окружающей среды, используются в вычислениях или иной обработке данных.

Отметим близкое к представляемому подходу исследование [30, 31], где для верификации ПЛК показана эффективность применения ограниченных хорновских дизьюнктов (constrained Horn clauses). Требование безопасности программы для ПЛК формулируется как свойство автоматов потоков управления (control flow automaton). С помощью известного SMT-солвера Z3 [32] оказывается возможным доказать безопасность формализованной программы для ПЛК, причем предварительно применяется процедура построения абстракции для уменьшения сложности вывода. Заметим, что исчисление ПОФ предлагается использовать для решения задач, предшествующих реализации желаемого поведения управляемой системы с помощью ПЛК, т. е. задач более высокого уровня абстракции. Однако, реализация построенного управления несомненно является важной задачей, решению которой посвящено немало работ в области ДСС. Так, в [33] представлен метод, который позволяет проектировщику эффективным образом внедрять результаты, полученные с помощью

[1] https://github.com/snigavik/bootfrost

ТСУ, для координации работы подсистем в программе для ПЛК, реализующей конкретную архитектуру супервизорного управления.

Еще одним направлением теоретического и практического развития ПОФ-исчисления является исследование темпоральной логики и применение ПОФ-исчисления для автоматизации поиска темпоральных выводов, а также использование темпоральной логики для исследования и управления ДСС. Так, в рамках ТСУ логика линейного времени (LTL) использовалась для задания и проверки свойств безопасности и живучести системы (например, [34, 35]). Формулы LTL позволяют формализовать такие утверждения как «ничего плохого никогда не произойдет» и «что-то хорошее, например, выполнение задач, будет происходить регулярно», поэтому они охватывают полезный набор спецификаций, связанных с регулярным достижением заданных целей без ущерба для безопасности системы [36]. В последующем, логика деревьев вычислений (CTL) и эпистемическая темпоральная логика начали применяться для работы с более сложными свойствами ДСС, например, свойством устойчивости, которое требует, чтобы система в конечном итоге достигла набора состояний, в которых выполняется какое-то утверждение, и оставалась там навсегда [37—39].

Близким к нашему исследованию является подход к тестированию диагностируемости ДСС на основе ее логического представления, предложенного в [40]. В [40] для изучения свойств диагностируемости ДСС используются конъюнктивные нормальные формы (КНФ). Переходы автоматов описываются как множество дизъюнктов, и затем хорошо известный метод резолюций применяется для проверки того, можно ли обнаружить события сбоя среди конечного числа наблюдаемых событий. Учитывая, что КНФ является менее выразительным, по сравнению с ПОФ, средством представления автоматов, лежащих в основе ДСС, мы оставляем проблему диагностируемости ДСС и сравнение с подходом на основе КНФ для будущих исследований.

В рамках ТСУ разработано несколько инструментов для анализа и проектирования управляемых ДСС. Среди них ТСТ [41], DESUMA/UMDES[2] [42], Supremica[3] и другие. В этом ряду следует отметить систему Supremica, имеющую удобный графический интерфейс. Supremica поддерживает РКА, в которых условия и действия переходов оперируют переменными, изменяющимися в конечных целочисленных диапазонах или имеющими перечисляемый тип [43]. Хотя программная система для решения задач ТСУ на основе ПОФ находится в стадии развития, наш подход предполагает, что как условия переходов, так и действия после них могут быть выражены в форме логических утверждений любого рода. При этом формат выходных данных прувера Bootfrost может быть организован таким образом, чтобы обеспечить интерпретацию полученных результатов в системе Supremica.

В отличие от крупномасштабных промышленных примеров в [44—46], применение исчисления ПОФ для управления ДСС предполагает использование логических инструментов для представления и обработки знаний. Конструктивная семантика ПОФ-исчисления позволяет извлекать знания (например, планы действий для автономных роботов) из построенных выводов, а немонотонность вывода и возможный учет времени могут применяться для планирования действий в динамично меняющихся предметных областях.

Цель данной статьи — дать представление о ПОФ-исчислении и его применимости для решения основных задач ТСУ, таких как проверка управляемости заданной спецификации, построение управляемого подъязыка заданного языка спецификации и реализация супервизорного управления. В статье описываются основные элементы предлагаемого подхода.

[2] https://gitlab.eecs.umich.edu/wikis/desuma
[3] https://supremica.org

1. Предварительные сведения

1.1. Исчисление позитивно-образованных формул

Рассмотрим язык логики первого порядка, состоящий из первопорядковых логических формул (ПЛФ), построенных из атомарных формул с помощью операторов $\&$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$, кванторных символов $\forall$ и $\exists$ и констант *true* и *false*. Понятия термин, атом, литер определяются обычным образом. Далее неатомарные формулы и подформулы будут обозначаться прописными каллиграфическими буквами ($\mathcal{F}, \mathcal{P}, \mathcal{Q}$ и т. д.), возможно, с индексами. Множества формул будут обозначаться прописными греческими буквами (Φ, Ψ и т. д.), возможно, с индексами.

Пусть $X = \{x_1, \ldots, x_k\}$ — множество переменных, $A = \{A_1, \ldots, A_m\}$ — множество атомарных формул, называемое *конъюнктом*, а $\Phi = \{\mathcal{F}_1, \ldots, \mathcal{F}_n\}$ — множество ПЛФ. Формулы $\forall x_1 \ldots \forall x_k (A_1 \& \ldots \& A_m) \rightarrow (\mathcal{F}_1 \vee \ldots \vee \mathcal{F}_n)$ и $\exists x_1 \ldots \exists x_k (A_1 \& \ldots \& A_m) \& (\mathcal{F}_1 \& \ldots \& \mathcal{F}_n)$ обозначаются как $\forall x_1, \ldots, x_k A_1, \ldots, A_m\{\mathcal{F}_1, \ldots, \mathcal{F}_n\}$ и $\exists x_1, \ldots, x_k\, A_1, \ldots, A_m\{\mathcal{F}_1, \ldots, \mathcal{F}_n\}$, соответственно. Их можно сокращенно обозначать $\forall_X A\ \Phi$ и $\exists_X A\ \Phi$, имея в виду, что $\forall$-квантор соответствует $\rightarrow \Phi^{\vee}$, где $\Phi^{\vee}$ означает дизъюнкцию всех формул из Φ, а $\exists$-квантор соответствует $\& \Phi^{\&}$, где $\Phi^{\&}$ означает конъюнкцию всех формул из Φ. Любое из множеств X, A, Φ может быть пустым, и в этом случае при формализации формул их можно опустить. Таким образом, если $Q \in \{\forall, \exists\}$, то $Q_X A\ \varnothing \equiv Q_X A$, $Q_X \varnothing\ \Phi \equiv Q_X\ \Phi$ и $Q_{\varnothing} A\ \Phi \equiv QA\ \Phi$. Поскольку пустая дизъюнкция тождественна *false*, а пустая конъюнкция тождественна *true*, то корректны следующие эквивалентности: $\forall_X A\ \varnothing \equiv \forall_X A \rightarrow false \equiv \forall_X A$ и $\exists_X A\ \varnothing \equiv \exists_X A \& true \equiv \exists_X A$ и $\forall \varnothing\ \Phi \equiv true \rightarrow \Phi \equiv \forall\ \Phi$ и $\exists \varnothing\ \Phi \equiv true \& \Phi \equiv \exists\ \Phi$. Кванторы без переменных называются *фиктивными* и не опускаются, так как позволяют корректно интерпретировать множество ПЛФ Φ и делают структуру формулы регулярной. Если $\mathcal{P} = \forall A\ \Phi$, то формула $\mathcal{P}$ интерпретируется как $A \rightarrow \Phi^{\vee}$. А если $\mathcal{P} = \exists A\ \Phi$, то $\mathcal{P}$ интерпретируем как $A \& \Phi^{\&}$.

Фразы $\forall_X A\ \Phi$ и $\exists_X A\ \Phi$ обычно обозначают такие высказывания на естественном языке, как, например, «для всех X, удовлетворяющих свойству A, существует Φ», «для всех целых x, y, z и $n > 2$ существует $x^n + y^n \neq z^n$», и т. д. Первоначально конструкции $\forall_{XA}$ и $\exists_{XA}$ назывались *ти́повыми кванторами* и были введены Н. Бурбаки [47] как часть нотации для формализации математики.

Определение 1 (Позитивно-образованные формулы (ПОФ))**.** Синтаксис языка ПОФ в нотации Бэкуса-Наура можно представить следующим образом:

<pcf–formula>::= ∀ {<e–formulas>}
<a–formulas> ::= "" | <a–formula> | <a–formula>, <a–formulas>
<e–formulas> ::= "" | <e–formula> | <e–formula>, <e–formulas>
<e–formula> ::= ∃ <vars> <conjunct> {<a–formulas>}
<a–formula> ::= ∀ <vars> <conjunct> {<e–formulas>}
<vars> ::= "" | <symbol> | <symbol> , <vars>
<conjunct> ::= "" | <atom> | <atom> , <conjunct>
<atom> :: <symbol> (<terms>)
<terms> ::= "" | <term> | <term> , <terms>
<term> :: <symbol> | <symbol> (<terms>)
<symbol> ::= [a–zA–Z0–9_]+

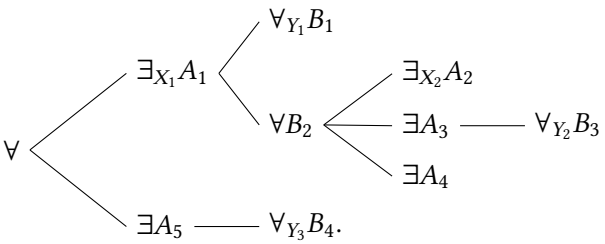
Термин «позитивные» обусловлен тем, что согласно определению, ПОФ не содержат оператор отрицания ($\neg$). Это справедливо и для семантики языка: отрицание «скрыто» в импликациях, которые подразумеваются после кванторов всеобщности. Отметим, что любая ПЛФ может быть представлена в виде ПОФ, поскольку ПОФ-язык — это специальный способ записи классических ПЛФ, так же как, например, конъюнктивные нормальные формы, дизъюнктивные нормальные формы и т. д. Алгоритм преобразования ПЛФ в ПОФ и обратно представлен в [48].

Если ПОФ имеет вид $\forall\ \Phi$, то такая ПОФ называется *канонической ПОФ* (это самый верхний нетерминальный символ в определении, `pcf-formula`). Очевидно, что любая ПОФ может быть приведена к канонической форме. Если $\mathcal{P}$ — неканоническая $\exists$-ПОФ, то $\mathcal{P} \equiv \forall\ \mathcal{P}$, так как $\forall\ \mathcal{P} \equiv true \rightarrow \mathcal{P} \equiv \mathcal{P}$. Если $\mathcal{P}$ — неканоническая $\forall$-ПОФ, то $\mathcal{P} \equiv \forall\ \{\exists\ \mathcal{P}\}$, так как $\forall\ \{\exists\ \mathcal{P}\} \equiv true \rightarrow (true \& \mathcal{P}) \equiv \mathcal{P}$.

Для удобства чтения мы представляем ПОФ в виде деревьев, узлами которых являются типовые кванторы, и используем привычные для них наименования: *узел, корень, лист, ветвь.* Например, ПОФ:

$$\forall\ \{\exists_{X_1} A_1\ \{\forall_{Y_1} B_1,\ \forall B_2\ \{\exists_{X_2} A_2,\ \exists A_3\ \{\forall_{Y_2} B_3\},\ \exists A_4\}\},\ \exists A_5\ \{\forall_{Y_3} B_4\}\}$$

представляется в виде дерева следующим образом:



Если дана ПОФ $\mathcal{P} = \forall\{\mathcal{F}_1, \ldots, \mathcal{F}_n\}$ и $\mathcal{F}_i = \exists_{X_i} B_i\{\mathcal{Q}_{i1}, \ldots, \mathcal{Q}_{im}\}$, $i = \overline{1,n}$, то $\mathcal{F}_i$ будут называться *базовыми подформулами* $\mathcal{P}$, B_i называются *базами фактов* или просто *базами*, $\mathcal{Q}_{ij}$ называются *подформулами-вопросами*, а корни подформул-вопросов называются *вопросами* к базе B_i, $i = \overline{1,n}$. Вопрос вида $\forall_X A$ (без дочерних элементов) называется *целевым вопросом.* Будем считать, что внутри базовых подформул любая переменная не может быть одновременно свободной и связанной, более того, она не может быть связана разными кванторами одновременно.

В процессе рассуждений часто доказывают утверждение $\mathcal{F}$, опровергая его отрицание. Мы намерены действовать аналогичным образом. Метод применяется только к каноническим ПОФ.

Определение 2 (Ответ)**.** Рассмотрим некоторую базовую подформулу $\exists_X A\ \Psi$ ПОФ. Вопрос подформулы $\mathcal{Q} = \forall_Y B\ \Phi$, $Q \in \Psi$ имеет ответ θ тогда и только тогда, когда θ является подстановкой $Y \rightarrow H^\infty \cup X$ и $B\theta \subseteq A$, где H^∞ — эрбрановский универсум в основе которого лежат символы констант и функций, встречающиеся в соответствующей базовой подформуле.

Определение 3 (Merge)**.** Пусть $\mathcal{P}_1 = \exists_X A\ \Psi$ и $\mathcal{P}_2 = \exists_Y B\ \Phi$, тогда $merge(\mathcal{P}_1, \mathcal{P}_2) = \exists_{X \cup Y} A \cup B\ \Psi \cup \Phi$.

Определение 4 (Split)**.** Рассмотрим некоторую базовую подформулу $\mathcal{B} = \exists_X A\ \Psi$. Пусть подформула-вопрос $\mathcal{Q} \in \Psi$ имеет вид $\forall_Y D\ \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, где $\mathcal{P}_i = \exists_{Z_i} C_i\ \Gamma_i, i = \overline{1,n}$, тогда $split(B, Q) = \{merge(B, \mathcal{P}_1'), \ldots,\ merge(B, \mathcal{P}_n')\}$, где $'$ — оператор переименования переменных. Будем говорить, что $\mathcal{B}$ расщепляется с помощью $\mathcal{Q}$, а $split(\mathcal{B}, \mathcal{Q})$ — результат расщепления $\mathcal{B}$. Очевидно, что $split(\mathcal{B}, \forall_Y D) = \varnothing$.

Определение 5 (Правило вывода ω)**.** Рассмотрим некоторую каноническую ПОФ $\mathcal{F} = \forall\ \Phi$. Если существует базовая подформула $\mathcal{B} = \exists_X A\ \Psi$, $\mathcal{B} \in \Phi$ и существует подформула-вопрос $\mathcal{Q} \in \Psi$, и вопрос $\mathcal{Q}$ имеет ответ θ на $\mathcal{B}$, тогда $\omega(\mathcal{F}) = \forall\ \Phi \setminus \{\mathcal{B}\} \cup split(\mathcal{B}, \mathcal{Q}\theta)$.

Обратим внимание, что если после применения правила ω множество Φ становится пустым, а ПОФ — лишь квантором $\forall$, то можно сделать вывод, что отрицание исходной формулы невыполнимо, следовательно, сама формула общезначима.

Любая конечная последовательность ПОФ $\mathcal{F}, \omega\mathcal{F}, \omega^2\mathcal{F}, \ldots, \omega^n\mathcal{F}$, где $\omega^s\mathcal{F} = \omega(\omega^{s-1}\mathcal{F})$, $\omega^1 = \omega$, $\omega^n\mathcal{F} = \forall$, называется *выводом* $\mathcal{F}$ в ПОФ-исчислении (с аксиомой $\forall$).

Предположим, что стандартная стратегия поиска выводов проверяет вопросы в последовательном порядке, повторяя проверку только тогда, когда список вопросов заканчивается, и не использует повторное применение ω к вопросу с той же подстановкой θ (вопросно-ответный метод автоматического поиска вывода). Подробные сведения о ПОФ-исчислении изложены в [26], [49].

Пример 1. Покажем вывод ПОФ $\mathcal{F}_1$.

$$\mathcal{F}_1 = \forall - \exists S(e) \begin{cases} \forall x\ S(x) \text{ —— } \exists A(x) \\ \forall x, y\ C(x,y) \\ \forall x\ A(x) \begin{cases} \exists y\ C(y, f(x)) \\ \exists \text{ —— } \forall x\ S(x), A(x) \text{ — } \exists C(x, f(x)). \end{cases} \end{cases}$$

На первом шаге можно найти только одну ответную подстановку: $\{x \to e\}$ для первого вопроса (обозначим вопросы $Q_i, i = \overline{1,3}$, нумеруя по возрастанию сверху вниз). После применения правила вывода ω с этим ответом к единственной базе $\mathcal{F}_1$ формула трансформируется и принимает следующий вид:

$$\mathcal{F}_2 = \forall - \exists S(e), A(e) \begin{cases} \forall x\ S(x) \text{ —— } \exists A(x) \\ \forall x, y\ C(x,y) \\ \forall x\ A(x) \begin{cases} \exists y\ C(y, f(x)) \\ \exists \text{ —— } \forall x\ S(x), A(x) \text{ — } \exists C(x, f(x)). \end{cases} \end{cases}$$

На втором шаге также доступен только один ответ на вопрос Q_3 с подстановкой $\{x \to e\}$. После применения ω формула расщепляется, поскольку после Q_3 имеется дизъюнктивное ветвление. Формула примет следующий вид:

$$\mathcal{F}_3 = \forall \begin{cases} \mathcal{F}_3^1 \\ \mathcal{F}_3^2, \end{cases}$$

где $\mathcal{F}_3^1$ имеет вид:

$$\mathcal{F}_3^1 \text{ —— } \exists y_1\ S(e), A(e), C(y_1, f(e)) \begin{cases} \forall x\ S(x) \text{ —— } \exists A(x) \\ \forall x, y\ C(x,y) \\ \forall x\ A(x) \begin{cases} \exists y\ C(y, f(x)) \\ \exists \text{ —— } \forall x\ S(x), A(x) \text{ — } \exists C(x, f(x)), \end{cases} \end{cases}$$

а $\mathcal{F}_3^2$ есть:

$$\mathcal{F}_3^2 \text{ —— } \exists S(e), A(e) \begin{cases} \forall x\ S(x) \text{ —— } \exists A(x) \\ \forall x, y\ C(x,y) \\ \forall x\ A(x) \begin{cases} \exists y\ C(y, f(x)) \\ \exists \text{ —— } \forall x\ S(x), A(x) \text{ — } \exists C(x, f(x)) \end{cases} \\ \forall x\ S(x), A(x) \text{ - } \exists C(x, f(x)). \end{cases}$$

На третьем шаге первая база может быть опровергнута ответом на целевой вопрос Q_2 с подстановкой $\{x \to y_1, y \to f(e)\}$. Затем опровергнутая база, как и вся базовая подформула $\mathcal{F}_3^1$, должны быть удалены из списка базовых подформул формулы $\mathcal{F}_3$.

На четвертом шаге можно найти ответ на четвертый, добавленный вопрос базовой подформулы $\mathcal{F}_3^2$. После ответа на него с подстановкой $\{x \to e, y \to e\}$ к базе $\mathcal{F}_3^2$ добавится атом $C(e, f(e))$.

На пятом шаге используется вновь добавленный атом для ответа на целевой вопрос Q_2 с подстановкой $\{x \to e; y \to f(e)\}$, что опровергает базу $\mathcal{F}_3^2$ и завершает вывод, так как все базы были опровергнуты.

1.2. Логическая ДСС как генератор формального языка

Рассмотрим логическую дискретно-событийную систему (ДСС), описывающую функционирование некоторой реальной системы на абстрактно-символическом уровне последовательностями событий, происходящих в неопределенные моменты времени. Такая ДСС может быть представлена [4] конечным автоматом $\boldsymbol{G} = (\boldsymbol{Q}, \Sigma, \boldsymbol{\delta}, \boldsymbol{q_0}, \boldsymbol{Q_m})$, играющим роль генератора регулярного языка. Здесь $\boldsymbol{Q}$ — множество состояний $\boldsymbol{q}$; Σ — множество событий; $\boldsymbol{\delta}$: $\Sigma \times \boldsymbol{Q} \to \boldsymbol{Q}$ — функция перехода; $\boldsymbol{q_0} \in \boldsymbol{Q}$ — начальное состояние; $\boldsymbol{Q_m} \subset \boldsymbol{Q}$ — множество выделенных состояний. $\boldsymbol{G}$ также называется *объектом управления* в теории автоматического управления. Пусть Σ^* обозначает замыкание Клини, $\boldsymbol{\varepsilon}$ — пустая строка. $\boldsymbol{\delta}$ легко распространяется на строки из Σ^*. Язык, генерируемый $\boldsymbol{G}$ — это $\boldsymbol{L(G)} = \{\boldsymbol{s} : \boldsymbol{s} \in \Sigma^*$ и $\boldsymbol{\delta(s, q_0)}$ определен$\}$, а язык, маркируемый $\boldsymbol{G}$, это – $\boldsymbol{L_m(G)} = \{\boldsymbol{s} : \boldsymbol{s} \in \boldsymbol{L(G)}$ и $\boldsymbol{\delta(s, q_0)} \in \boldsymbol{Q_m}\}$. Маркировка необходима для того, чтобы выделить строки, которые в системе считаются «завершенными», например, окончание процесса сборки на производственной линии или завершение последовательности действий, формирующих миссию мобильного робота. При этом элементы множества $\boldsymbol{Q_m}$ не рассматриваются как терминальные состояния. В связи с этим к генераторам также применяется термин «машина конечных состояний». Однако, начиная с основополагающей работы [4], термин «автомат» стал стандартным термином, используемым в ТСУ. Обозначим через $\overline{\boldsymbol{K}}$ множество всех префиксов слов из $\boldsymbol{K}$. Язык $\boldsymbol{K}$ называется префиксно-замкнутым, если $\overline{\boldsymbol{K}} = \boldsymbol{K}$. Для любого генератора $\boldsymbol{G}$ справедливо $\boldsymbol{L(G)} = \overline{\boldsymbol{L(G)}}$.

ТСУ предполагает, что возникновение некоторых событий в системе может быть запрещено внешним средством управления, называемым супервизором. Пусть $\Sigma = \Sigma_c \cup \Sigma_{uc}$, где Σ_c — множество управляемых событий, $\Sigma_{uc} = \Sigma \backslash \Sigma_c$, события в множестве Σ_{uc} являются неуправляемыми, $\Sigma_c \cap \Sigma_{uc} = \varnothing$. Чтобы система не вышла за рамки поведения, определенного неким регулярным языком, например, $\boldsymbol{K}$, супервизор запрещает возникновение событий из $\boldsymbol{\Sigma_c}$. $\boldsymbol{K}$ называется спецификацией поведения системы. Обозначим через $\boldsymbol{L(J/G)}$ язык, генерируемый ДСС $\boldsymbol{G}$ под управлением супервизора $\boldsymbol{J}$. Пусть $\boldsymbol{L_m(J/G)}$ обозначает язык, маркированный супервизором: $\boldsymbol{L_m(J/G)} = \boldsymbol{L(J/G)} \cap \boldsymbol{L_m(G)}$. Основная задача супервизорного управления состоит в построении такого супервизора $\boldsymbol{J}$, что $\boldsymbol{L(J/G)} = \overline{\boldsymbol{K}}$ и $\boldsymbol{L_m(J/G)} = \boldsymbol{K}$. Второе равенство важно для задач, связанных с обеспечением неблокирования системы.

Критерием существования супервизора, обеспечивающего решение основной задачи супервизорного управления, является управляемость языка спецификации $\boldsymbol{K}$. Язык $\boldsymbol{K}$ называется *управляемым* (относительно $\boldsymbol{L(G)}$ и Σ_{uc}), если $\overline{\boldsymbol{K}}\boldsymbol{\Sigma_{uc}} \cap \boldsymbol{L(G)} \subseteq \overline{\boldsymbol{K}}$. Здесь $\overline{\boldsymbol{K}}\boldsymbol{\Sigma_{uc}}$ — сокращенное выражение, обозначающее конкатенацию всех строк из $\overline{\boldsymbol{K}}$ с любым из символов из множества $\boldsymbol{\Sigma_{uc}}$. Проверка управляемости является первым этапом процедуры построения супервизорного управления по заданной спецификации. В случае, когда управляемость не имеет места, можно построить гарантированно управляемый подъязык спецификации, обеспечивающий наименьшее ограничение возможностей функционирования системы. Доказано, что наибольший управляемый подъязык $\boldsymbol{K}^{\uparrow C}$ языка $\boldsymbol{K}$ всегда существует. Если построенный подъязык оказывается удовлетворительным ограничением на систему, то в качестве новой спецификации выбирается $\boldsymbol{K}^{\uparrow C}$, и для его обеспечения строится соответствующее супервизорное управление. Ниже будет показано, как логический

вывод ПОФ применяется для анализа управляемости, построения минимально ограничивающих спецификаций и реализации супервизорного управления.

2. Супервизорное управление с помощью ПОФ

2.1. Представление ДСС в виде ПОФ

Для представления некоторой ДСС G с помощью ПОФ-исчисления мы используем следующие предикаты. Пусть предикат $L(s, q)$ обозначает, что «последовательность событий s приводит систему в состояние q». Аналогично, предикат $L^m(s, q)$ обозначает факт «последовательность событий s приводит систему в маркированное состояние q». Первые аргументы этих предикатов будут накапливать строки языков, генерируемого и маркированного автоматом G. Пусть предикат $\delta(q_1, \sigma, q_2)$ интерпретируется как переход из состояния q_1 в состояние q_2 по событию σ. Аналогично, предикат $\delta^m(q_1, \sigma, q_2)$ соответствует переходу в маркированное состояние q_2. Как обычно, функциональный символ «$\cdot$» обозначает конкатенацию строк, а символ «ε» соответствует пустой строке.

$$\exists\, B_G \left\langle \begin{array}{ll} \forall s, q, \sigma, q'\ L(\sigma, s), \delta(q, \sigma, q') & \text{——} \ \exists L(s \cdot \sigma, q') \\ \forall s, q, \sigma, q'\ L(\sigma, s), \delta_m(q, \sigma, q') & \text{——} \ \exists L^m(s \cdot \sigma, q') \end{array} \right.$$

Fig. 1. PCF $\mathcal{F}_G$ generating $L(G)$ and $L_m(G)$ for DES G

Рис. 1. Общий вид ПОФ $\mathcal{F}_G$, строящей $L(G)$ и $L_m(G)$ для ДСС G

Если дан генератор G, процесс построения языков $L(G)$ и $L_m(G)$ можно реализовать с помощью ПОФ $\mathcal{F}_G$ на рис. 1. Ее базой является множество $B_G = \{L(\varepsilon, S_0),\ L_m(\varepsilon, S_0),\ \delta(S_1^i, \sigma^i, S_2^i),\ \delta^m(S_1^j, \sigma^j, S_2^j)\}$, $i, j \in \{1, \ldots, n\}$, где n — количество событий G. База содержит атомы, описывающие переходы между состояниями G, включая маркированные, и начальные атомы $L(\varepsilon, S_0)$, $L_m(\varepsilon, S_0)$, которые будут использоваться для построения языков G. В зависимости от выбранного на текущем шаге вывода вопроса использование правила вывода ω ПОФ-исчисления влечет за собой генерацию нового факта $L(s, q)$ или $L^m(s, q)$ и добавление его в базу, причем аргументом s является слово языка $L(G)$ или $L_m(G)$ соответственно. Для различных задач ТСУ управляемые и неуправляемые события будут представлены атомами, которые являются предикатами $\Sigma_c(\_)$ и $\Sigma_{uc}(\_)$ соответственно. Вся доступная информация о системе может храниться в базе в виде фактов для дальнейшего использования. Расширенной мы называем базу, содержащую информацию, которая не является необходимой для вывода текущей формулы.

Пример 2. Рассмотрим ДСС G на рис. 2. Пусть $\Sigma_{uc} = \{a_1, a_2, a_3, a_4\}$. Для этого автомата общая форма ПОФ $\mathcal{F}_G$ на рис. 1 имеет расширенную базу, представленную множеством $B_G = \{Q_0(A), Q(A), Q(B), Q(C), Q(D), Q(E), \Sigma_{uc}(a_i), L(\varepsilon, A), L^m(\varepsilon, A), \delta(A, a_1, B), \delta(B, p_3, A), \ldots, \delta^m(C, p_2, A), \delta^m(B, p_3, A), \ldots\}$, $i = \overline{1, 4}$.

Следует заметить, что в ПОФ $\mathcal{F}_G$ отсутствует целевой вопрос. Отсутствие целевого вопроса в ПОФ означает ее выполнимость, т. е. такая ПОФ никогда не может быть опровергнута. В этом случае вывод завершается только из-за исчерпания подстановок, позволяющих применить правило вывода ω. В зависимости от рассматриваемой задачи ТСУ нас может интересовать построение вывода формулы без цели ее опровержения. При выводе $\mathcal{F}_G$, если это позволяет атом $\delta(q_1^i, \sigma^i, q_2^i)$, с помощью функционального символа $\cdot$ создается новый атом, который добавляется к базе на каждом этапе вывода, поэтому невозможно исчерпать все возможные подстановки. Таким образом, бесконечность вывода позволяет реализовать ДСС как генератор регулярного языка, включающего слова любой длины. Если атомы $\delta(q_1^i, \sigma^i, q_2^i)$ не допускают никаких подстановок, это означает, что все слова $L(G)$, $L_m(G)$ уже построены.

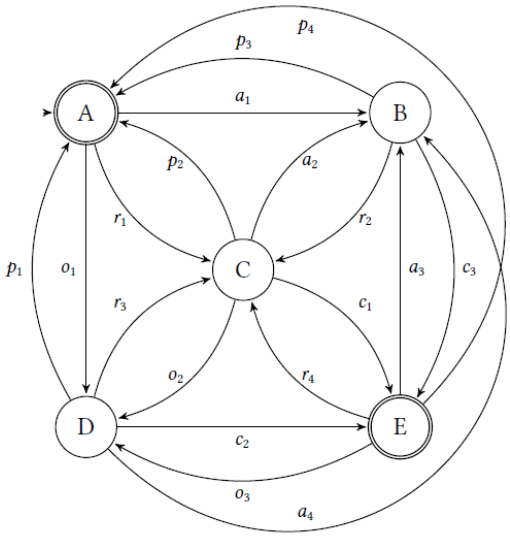


Fig. 2. Generator G **Рис. 2.** Генератор G

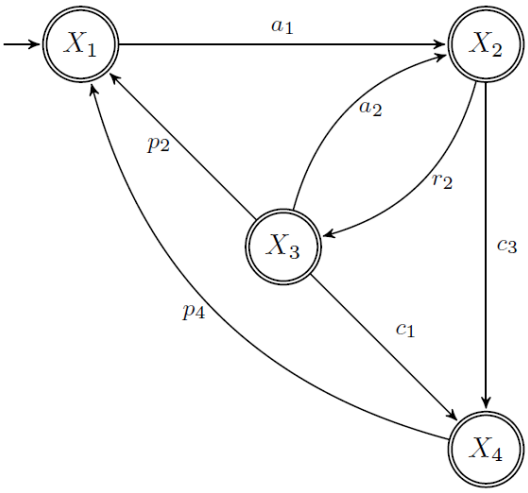


Fig. 3. Automaton H recognizing specification language K **Рис. 3.** Автомат H, распознающий язык спецификации K

Далее в данной статье для простоты изложения не рассматриваются маркированный язык и связанные с ним задачи, такие как проверка блокирования или построение неблокирующего супервизорного управления.

2.2. Анализ языка спецификации на ДСС

Пусть спецификация языка K на ДСС G распознается автоматом H. Как говорилось ранее, критерием существования супервизорного управления является управляемость K. Согласно парадигме АДТ, для проверки некоторого свойства ищется опровержение отрицания этого свойства. Таким образом, будем доказывать отсутствие неуправляемости спецификации K, т. е. что возникновение неуправляемого события не приводит к строке, которая принадлежит языку $L(G)$, но не является префиксом слова из K. Для этого воспользуемся операцией произведения автоматов, соответствующих системе и спецификации $G \times H$, поскольку такой автомат генерирует только строки, общие для $L(G)$ и K, $L(G \times H) = L(G) \cap K$.

Рассмотрим ПОФ $\mathcal{F}_{Ctl\_ch}$, реализующую проверку управляемости спецификации K:

$$\mathcal{F}_{Ctl\_ch} = \exists B_{Ctl\_ch} \begin{cases} R_1 \\ R_2 \\ \dots \\ R_6, \end{cases}$$

$$R_1 : \ \forall q_1^1, q_2^1, q_1^3, q_2^3, \sigma \ Q_0^1(q_1^1), Q_0^3(q_1^3), \delta_1(q_1^1, \sigma, q_2^1), \delta_3(q_1^3, \sigma, q_2^3)^- \exists \ N(q_1^1, q_1^3),$$
$$R_2 : \ \forall q_1^1, q_2^1, q^3, \sigma \ N(q_1^1, q^3), \delta_1(q_1^1, \sigma, q_2^1), E_{uc}(\sigma)^- \exists \ Chk(q^3, \sigma, 0),$$
$$R_3 : \ \forall q_1^3, q_2^3, \sigma \ Chk^*(q_1^3, \sigma, 0), \delta_3(q_1^3, \sigma, q_2^3)^- \exists \ Chk(q_1^3, \sigma, 1),$$
$$R_4 : \ \forall q^3, \sigma \ Chk(q^3, \sigma, 0) - \exists \ UC(q^3, \sigma),$$
$$R_5 : \forall p, \sigma \ UC(p, \sigma),$$
$$R_6 : \ \forall q_1^1, q_2^1, q_1^3, q_2^3, \sigma \ N(q_1^1, q_1^3), \delta_1(q_1^1, \sigma, q_2^1), \delta_3(q_1^3, \sigma, q_2^3)^- \exists \ N(q_2^1, q_2^3).$$

Предполагается, что база B_{Ctl_ch} ПОФ $\mathcal{F}_{Ctl\_ch}$ содержит предикатные описания переходов $\boldsymbol{G}$, $\boldsymbol{H}$ и $\boldsymbol{G} \times \boldsymbol{H}$. Заметим, что синхронную композицию автоматов и их произведение можно построить и с использованием подхода на основе ПОФ. В правилах $R_1 - R_6$ используются вспомогательные предикаты. Предикат $N(\_, \_)$ будет хранить пары состояний, которые одновременно достигаются из состояний $\boldsymbol{G}$ и $\boldsymbol{G}\times\boldsymbol{H}$ по одному и тому же событию. Такие состояния называются *соседними* состояниями. Остальные новые предикаты будут объясняться по мере их появления. Правило R_1 добавляет в базу начальные состояния $\boldsymbol{G}$ и $\boldsymbol{G} \times \boldsymbol{H}$, которые являются стартовыми для процедуры проверки нарушения управляемости. Правило R_2 обнаруживает в $\boldsymbol{G}$ переход по неуправляемому событию $\boldsymbol{\sigma}$. Если ответ на этот вопрос успешен, то в базу добавляется атом $Chk(q^3, \sigma, 0)$ для проверки состояния q^3 автомата $\boldsymbol{G} \times \boldsymbol{H}$ на существование в нем, а значит и в $\boldsymbol{H}$, перехода по событию $\boldsymbol{\sigma}$. Наличие такого перехода означает, что условие управляемости не нарушается, и флаг 0 в $Chk(q^3, \sigma, 0)$ меняется на 1. Затем вопрос R_6 используется для добавления следующей пары проверяемых состояний. Если флаг в $Chk(q^3, \sigma, 0)$ остаётся 0, то имеет место нарушение управляемости, и вопросом R_4 в базу добавляется атом $UC(p, \sigma)$, который позволяет ответить на целевой вопрос R_5. Если вывод завершился исчерпанием вариантов поиска замен и ответ на целевой вопрос не найден, то язык спецификации является управляемым. Мы формализовали задачу таким образом, что если спецификация неуправляема, то успешное завершение вывода обеспечивает набор переменных, нарушающих управляемость спецификации, в рамках ответа $UC(p, \sigma)$ на вопрос R_5. Поскольку рассматриваются только конечные автоматы, пространство поиска вывода в этой формализации является конечным. Следовательно, вывод заканчивается как в случае управляемости спецификации, так и в случае ее неуправляемости.

В случае нарушения управляемости спецификации нас интересует наибольший управляемый подъязык $\boldsymbol{K}^{\uparrow C}$, который можно выбрать в качестве альтернативной спецификации. Предложена ПОФ $\mathcal{F}_{Sub}$, обеспечивающая построение $\boldsymbol{K}^{\uparrow C}$ в процессе проверки управляемости спецификации $\boldsymbol{K}$, таким образом позволяя объединить анализ спецификаций и построение супервизора:

$$\mathcal{F}_{Sub} = \exists B_{Sub} \left\langle \begin{array}{l} R_1 \\ R_2 \\ \dots \\ R_7, \end{array} \right.$$

$$R_4 : \ \forall q^3, \sigma \ Chk(q^3, \sigma, 0) - \exists \ Del(q^3),$$
$$R_5 : \ \forall q_1^1, q_2^1, q_1^3, q_2^3, \sigma \ N(q_1^1, q_1^3), \delta_1(q_1^1, \sigma, q_2^1), \delta_3(q_1^3, \sigma, q_2^3)^- \exists \ N(q_2^1, q_2^3),$$
$$R_6 : \ \forall q_{11}, q_{12}, q_{21}, q_{22}, \sigma \ Del(q_{11} \cdot q_{12}), \delta_2^*(q_{12}, \sigma, q_{22}), \delta_3^*(q_{11} \cdot q_{12}, \sigma, q_{21} \cdot q_{22})^- \exists \ Deleted(q_{12}, \sigma, q_{22}),$$
$$R_7 : \ \forall q_{11}, q_{12}, q_{21}, q_{22}, \sigma \ Del(q_{21} \cdot q_{22}), \delta_2^*(q_{12}, \sigma, q_{22}), \delta_3^*(q_{11} \cdot q_{12}, \sigma, q_{21} \cdot q_{22})^- \exists \ Deleted(q_{12}, \sigma, q_{22}).$$

База B_{Sub} совпадает с базой B_{Ctl_ch}, и теми же остаются правила R_1–R_3. Вопрос R_5 используется для проверки следующего состояния $\boldsymbol{G}$. Если флаг в $Chk(q^3, \sigma, 0)$ остаётся 0, то вспомогательный предикат $Del()$ в вопросе R_4 запускает процесс удаления переходов, нарушающих управляемость спецификации, с помощью вопросов R_6 и R_7. Затем процесс повторяется до тех пор, пока поиск вывода не остановится, исчерпав все возможные подстановки. В результате вывода база ПОФ будет содержать атомы, описывающие генератор $\boldsymbol{H'}$ наибольшего управляемого подъязыка $\boldsymbol{K}^{\uparrow C}$ спецификации $\boldsymbol{K}$.

Одной из существенных особенностей исчисления ПОФ, которая будет использоваться для анализа ДСС, является тот факт, что мы можем построить *немонотонный* вывод, слегка изменив определение правила вывода. Для этого введем оператор *, которым могут быть помечены атомы в вопросах. Теперь, если на вопрос с атомами, отмеченными оператором *, есть ответ, то после применения правила вывода атомы в базе, участвовавшие в поиске совпадений с отмеченными

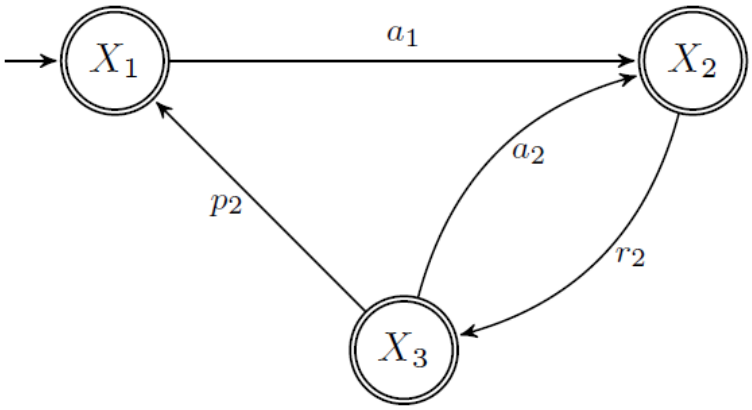


Fig. 4. Automaton H' recognizing $K^{\uparrow C}$ **Рис. 4.** Автомат H', распознающий $K^{\uparrow C}$

атомами, должны быть удалены из базы. В целом оператор * влияет на свойство полноты ПОФ-исчисления, но для рассматриваемых в статье задач благодаря правильной формализации вывод с использованием * всегда корректен.

Чтобы построить $K^{\uparrow C}$, мы используем следующую стратегию. Вывод строится путем проверки применимости правила вывода к вопросам в порядке от R_1 до R_7. Если при применении правил $R_1 - R_5$ был найден ответ на вопрос R_4, к базе добавляется атом $Del()$, обозначающий неуправляемость K. Затем применяем правила R_6, R_7, игнорируя правила $R_1 - R_5$, пока не будут исчерпаны возможные замены для применения правила вывода к R_6 и R_7. Далее процесс повторяется до тех пор, пока вывод не остановится, исчерпав все возможные замены. Например, если для данной спецификации управляемый подъязык равен только пустой строке, то при выводе все переходы в H будут удалены. Рассмотрим эту стратегию на примере.

Пример 3. Для ДСС G на рисунке 2 рассмотрим язык спецификации K, определенный автоматом H на рисунке 3. Можно заметить, что неуправляемое событие a_3 делает $K = L(H)$ неуправляемым относительно G и Σ_{uc}. Для проверки управляемости K логическими методами воспользуемся ПОФ $\mathcal{F}_{Sub}$. Для этого примера база $B_{Sub} = \{Q_0^1(A), Q_0^2(X_1), Q_0^3(A \cdot X_1), \Sigma_{uc}(a_i), \delta_1(A, a_1, B), \ldots, \delta_2(X_1, a_1, X_2), \ldots, \delta_3(A \cdot X_1, a_1, B \cdot X_2), \ldots\}$ включает группу атомов $\delta_1(\_, \_, \_)$, соответствующую переходам автомата G, группу атомов $\delta_2(\_, \_, \_)$, соответствующую переходам автомата H, и $\delta_3(\_, \_, \_)$, соответствующую переходам автомата-произведения $G \times H$. В таблице 1 показаны шаги вывода, строящего автомат H'. Таблица иллюстрирует минимальный вывод, построенный прувером Bootfrost, разработанным для опровержения ПОФ. Вывод был построен за 17 шагов. В таблице не показаны шаги, которые добавляют уже отмеченные для проверки пары состояний, поглощаются текущей базой, и те, которые приводят к успешной проверке управляемости. Заметим, что в этом примере вывод прекращается из-за исчерпания всех возможных подстановок. После окончания вывода переходы H' можно извлечь из базы $\mathcal{F}_{Sub}$ как атомы $\delta_2(\_, \_, \_)$. Получившийся автомат показан на рисунке 4.

2.3. Реализация супервизорного управления

Если H — автомат, распознающий управляемый язык спецификации K, то H можно использовать в качестве супервизора J для G и справедливо равенство $L(J/G) = J||G$ [5]. Таким образом, операция построения синхронной композиции автоматов может быть использована для реализации супервизорного управления. Поскольку множества событий H и G совпадают, вместо $H||G$ можно использовать операцию произведения $H \times G$. Заметим, что если спецификация K не является управляемой, но $K^{\uparrow C} \neq \emptyset$, то язык $K^{\uparrow C}$ может рассматриваться как новая, наименее ограничительная спецификация, а автомат H', распознающий $K^{\uparrow C}$, может использоваться в качестве супервизора J.

Table 1. Constructive inference providing H' as a recognizer of $K^{\uparrow C}$

Таблица 1. Конструктивный вывод, строящий H' как распознаватель $K^{\uparrow C}$

Шаг	Использованные атомы	Ответ	Добавленные атомы
1	$Q_0^1(A), Q_0^3(A \cdot X_1)$, $\delta_1(A, a_1, B)$, $\delta_3(A \cdot X_1, a1, B \cdot X_2)$	$\{q_1^1 \to A, q_2^1 \to B, q_1^3 \to A \cdot X_1$, $q_2^3 \to B \cdot X_2, \sigma \to a_1\}$	$N(A, A \cdot X_1)$
На первом этапе поиска вывода был найден ответ на вопрос R_1, что означает, что пара состояний A и $A \cdot X_1$ в автоматах G и $G \times H$ подлежит проверке на нарушение управляемости.			
2	$N(A, A \cdot X_1), \delta_1(A, a_1, B)$, $\Sigma_{uc}(a_1)$	$\{q_1^1 \to A, q_1^2 \to B, q^3 \to A \cdot X_1, \sigma \to a_1\}$	$Chk(A \cdot X_1, a_1, 0)$
Ранее добавленный атом $N(A, A \cdot X_1)$ помогает найти ответ на вопрос R_2. Ответ означает, что в автомате G найдена неконтролируемый переход по событию a_1, поэтому необходима проверка соответствующего перехода в автомате $G \times H$.			
3	$Chk(A \cdot X_1, a_1, 0)(deleted)$, $\delta_3(A \cdot X_1, a_1, B \cdot X_2)$	$\{q_1^3 \to A \cdot X_1, q_2^3 \to OA \cdot X_2, \sigma \to a_1\}$	$Chk(A \cdot X_1, a_1, 1)$
На вопрос R_3 получен ответ, значит, проверка, назначенная на предыдущем шаге, пройдена, т. е. нарушение управляемости не обнаружено. Заметим, что поиск вывода дает несколько возможных ответов на вопросы R_2 и R_3. Мы опускаем дополнительные варианты, чтобы показать минимальный вывод.			
4	$N(A, A \cdot X_1), \delta_1(A, a_1, B)$, $\delta_3(A \cdot X_1, a_1, B \cdot X_2)$	$\{q_1^1 \to A, q_2^1 \to B, q_1^3 \to A \cdot X_1, q_2^3 \to$ $B \cdot X_2, \sigma \to a_1\}$	$N(B, B \cdot X_2)$
Ответ на вопрос R_5 получен. Ответ аналогичен первому шагу данного вывода и добавляет следующую пару проверяемых состояний.			
5	$N(B, B \cdot X_2), \delta_1(B, r_2, C)$, $\delta_3(B \cdot X_2, r_2, B \cdot X_2)$	$\{q_1^1 \to B, q_2^1 \to C, q_1^3 \to B \cdot X_2$, $q_2^3 \to C \cdot X_3, \sigma \to r_2\}$	$N(C, C \cdot X_3)$
Ответ на вопрос R_5 получен. В базу для последующей проверки на нарушение управляемости добавляется еще одна пара состояний.			
6	$N(B, B \cdot X_2), \delta_1(B, c_3, E)$, $\delta_3(B \cdot X_2, c_3, E \cdot X_4)$	$\{q_1^1 \to B, q_2^1 \to E, q_1^3 \to B \cdot X_2$, $q_2^3 \to E \cdot X_4, \sigma \to c_3\}$	$N(E, E \cdot X_4)$
На вопрос R_5 дан ответ. В базу добавлена пара состояний $N(E, E \cdot X_4)$. Состояние E в автомате G имеет исходящий переход, помеченный неуправляемым событием a_3.			
7	$N(E, E \cdot X_4), \delta_1(E, a_3, B), \Sigma_{uc}(a_3)$	$\{q_1^1 \to E, q_2^1 \to B, q^3 \to E \cdot X_4, \sigma \to a_3\}$	$Chk(E \cdot X_4, a_3, 0)$
Получен ответ на вопрос R_2. Результат аналогичен второму шагу этого умозаключения, но событие, которое будет проверяться следующим, — a_3 в состоянии E.			
8	$Chk(E \cdot X_4, a_3, 0)$	$\{q^3 \to E \cdot E, \sigma \to a_3\}$	$Del(E \cdot X_4)$
Ответ на вопрос R_4 означает, что проверка, назначенная на предыдущем шаге, провалена. Таким образом, спецификация, соответствующая автомату H, является неуправляемой. Правила R_6, R_7 удаления переходов, связанных с состоянием E, будут срабатывать благодаря добавляемым атомам $Del(E \cdot X_4)$.			
9	$Del(E \cdot X_4), \delta_2(X_2, c_3, X_4)$, $\delta_3(B \cdot X_2, c_3, E \cdot X_4)$	$\{q_{11} \to B, q_{12} \to X_2, q_{21} \to E$, $q_{22} \to X_4, \sigma \to c_3\}$	$Deleted(X_2, c_1, X_4)$
Ответ на вопрос R_6 получен. Переход $\delta_2(X_2, c_3, X_4)$ автомата H удален.			
10	$Del(E \cdot X_4), \delta_2(X_4, p_4, X_1)$, $\delta_3(E \cdot X_3, p_4, A \cdot X_1)$	$\{q_{11} \to E, q_{12} \to X_3, q_{21} \to A$, $q_{22} \to X_1, \sigma \to p_4\}$	$Deleted(X_4, p_4, X_1)$
Ответ на вопрос R_6 получен. Переход $\delta_2(E, p_4, A)$ автомата H удален.			
11	$Del(E \cdot X_4), \delta_2(X_3, c_1, X_4)$, $\delta_3(C \cdot X_3, c_1, E \cdot X_4)$	$\{q_{11} \to C, q_{12} \to X_3, q_{21} \to E$, $q_{22} \to X_4, \sigma \to c_1\}$	$Deleted(C, c_1, E)$
Ответ на вопрос R_7 получен. Переход $\delta_2(X_3, c_1, X_4)$ автомата H удален.			

В предлагаемом подходе ПОФ $\mathcal{F}_{SupC}$ на рисунке 5 генерирует язык $L(J/G)$. Здесь множество $B_{J/G}$ включает в себя базы B_G и B_H, а также содержит атом $L^{J/G}(\varepsilon, S_0^G)$ в качестве начального атома для конструкции языка $L(J/G)$. Также в базе находится атом $E(\sigma^{\#})$, где $\sigma^{\#}$ — это результат вызова функции $get\_random\_event()$, которая предоставляет случайное событие, которое может произойти

$$\exists\ B_{J/G} \cup \{E(\sigma^{\#})\} \text{ — } \begin{array}{r} \forall\ s, s', q, \sigma, q'\ L^{J/G}(s,q), \\ L^J(s',q), E^*(\sigma), \\ \delta_J(q,\sigma,q') \end{array} \text{ — } \begin{array}{r} \exists L^J(s\cdot\sigma, q') \\ E(\sigma^{\#}) \end{array} \text{ — } \begin{array}{r} \forall\ L^{J/G}(s',q), \\ \delta_G(q,\sigma,q') \end{array} \text{ — } \exists L^{J/G}(s'\cdot\sigma, q')$$

Fig. 5. PCF $\mathcal{F}_{SupC}$ generating $\boldsymbol{L(J/G)}$

Рис. 5. Общая форма ПОФ $\mathcal{F}_{SupC}$, генерирующей $\boldsymbol{L(J/G)}$

$$\exists\ B_{J/G} \cup \{E(\sigma^{\#}), L^J(\varepsilon\cdot a_1, B)\} \left\langle \begin{array}{l} \forall\ s, s', q, \sigma, q'\ L^{J/G}(s,q), \\ L^J(s',q), E^*(\sigma), \\ \delta_J(q,\sigma,q') \end{array} \right. \text{ — } \begin{array}{r} \exists L^J(s\cdot\sigma, q') \\ E(\sigma^{\#}) \end{array} \text{ — } \begin{array}{r} \forall\ L^{J/G}(s',q), \\ \delta_G(q,\sigma,q') \end{array} \text{ — } \exists L^{J/G}(s'\cdot\sigma, q')$$
$$\left. \begin{array}{l} \\ \forall\ L^{J/G}(\varepsilon, A), \delta_G(A, a_1, B) \end{array} \right. \text{ — } \exists L^{J/G}(\varepsilon\cdot a_1, B)$$

Fig. 6. Generation of $\boldsymbol{L(J/G)}$, step 1, event a_1 occurred

Рис. 6. Генерация $\boldsymbol{L(J/G)}$, шаг 1, произошло событие a_1

$$\exists\ B_{J/G} \cup \{E(\sigma^{\#}), L^J(\varepsilon\cdot a_1, B), L^{J/G}(\varepsilon\cdot a_1, B)\} \text{ — } \begin{array}{l} \forall\ s, s', q, \sigma, q'\ L^{J/G}(s,q), \\ L^J(s',q), E^*(\sigma), \\ \delta_J(q,\sigma,q') \end{array} \text{ — } \begin{array}{r} \exists L^J(s\cdot\sigma, q') \\ E(\sigma^{\#}) \end{array} \text{ — } \begin{array}{r} \forall\ L^{J/G}(s',q), \\ \delta_G(q,\sigma,q') \end{array} \text{ — } \exists L^{J/G}(s'\cdot\sigma, q')$$

Fig. 7. Generation of $\boldsymbol{L(J/G)}$, step 2

Рис. 7. Генерация $\boldsymbol{L(J/G)}$, шаг 2

$$\exists\ B_{J/G} \cup \{E(\sigma^{\#}), L^J(\varepsilon\cdot a_1, B), L^{J/G}(\varepsilon\cdot a_1, B), L^J(\varepsilon\cdot a_1\cdot r_2, C)\} \left\langle \begin{array}{l} \forall\ s, s', q, \sigma, q'\ L^{J/G}(s,q), \\ L^J(s',q), E^*(\sigma), \\ \delta_J(q,\sigma,q') \end{array} \right. \text{ — } \begin{array}{r} \exists L^J(s\cdot\sigma, q') \\ E(\sigma^{\#}) \end{array} \text{ — } \begin{array}{r} \forall\ L^{J/G}(s',q), \\ \delta_G(q,\sigma,q') \end{array} \text{ — } \exists L^{J/G}(s'\cdot\sigma, q')$$
$$\left. \begin{array}{l} \\ \forall\ L^{J/G}(\varepsilon\cdot a_1, B), \delta_G(B, r_2, C) \end{array} \right. \text{ — } \exists L^{J/G}(\varepsilon\cdot a_1\cdot r_2, C)$$

Fig. 8. Generation of $\boldsymbol{L(J/G)}$, step 3, event r_2 occurred

Рис. 8. Генерация $\boldsymbol{L(J/G)}$, шаг 3, произошло событие r_2

в текущем состоянии. $\delta_J(\_,\_,\_)$ соответствует переходам $\boldsymbol{H}$, играющего роль супервизора для $\boldsymbol{G}$. Поскольку $\boldsymbol{H}$ является подавтоматом $\boldsymbol{G}$, предполагается, что состояния $\boldsymbol{H}$ переименованы соответственно состояниям $\boldsymbol{G}$.

Пример 4. Рисунки 6-8 и таблица 2 представляет первые шаги вывода ПОФ $\mathcal{F}_{SupC}$ для ДСС из примера 2 с языком спецификации $\boldsymbol{K}^{\uparrow C}$, распознаваемым автоматом на рисунке 4. Текущее состояние системы определяется вторым аргументом последнего добавленного атома $L^{J/G}(\_,\_)$. Первый столбец таблицы 2 — это номер вопроса, на который ищется ответ. Следующие пять столбцов показывают части найденной подстановки, например, в первой строке представлена подстановка: $\{\sigma \to a_1, q \to A, q1 \to B, s \to \varepsilon, s' \to \varepsilon\}$. Колонка **Добавления** содержит информацию о том, что было добавлено в базу на этом этапе, атомы и/или вопросы. Колонка **Удаления** содержит информацию об удалениях, произошедших в формуле на этом этапе.

Заметим, что из формальных определений ПОФ-исчисления следует, что если у вопроса нет кванторных переменных и типовое условие содержит только константы, такой вопрос можно удалять после ответа, т. к. на него можно ответить только один раз с пустой подстановкой. Такие вопросы называются *тривиальными*. На первом шаге вывода (рисунок 6) в формулу добавляется второй вопрос, являющийся тривиальным. После ответа на него на втором шаге он удаляется из ПОФ. Аналогично происходит на третьем и последующих шагах.

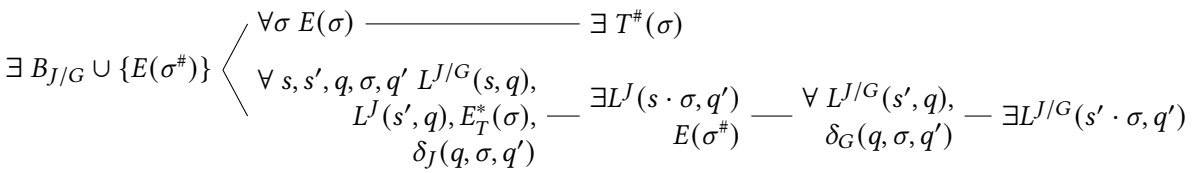


Fig. 9. PCF $\mathcal{F}_{Knl}$ realizing supervisory control using additional knowledge

Рис. 9. ПОФ $\mathcal{F}_{Knl}$, реализующая супервизорное управление с использованием дополнительных знаний

Table 2. Comments to the inference of $\mathcal{F}_{SupC}$ in Example 3

Таблица 2. Пояснения к выводу ПОФ $\mathcal{F}_{SupC}$ в примере 3

$Q\#$	σ	q	q'	s	s'	Добавления	Удаления
1	a_1	A	B	ε	ε	$L^J(\varepsilon\cdot a_1, B)$ $\forall\, L^{J/G}(\varepsilon, A), \delta_G(A, a_1, B)-$ $\exists L^{J/G}(\varepsilon\cdot a_1, B)$	$E(a_1)$
2	—	—	—	—	—	$L^{J/G}(\varepsilon\cdot a_1, B)$	вопрос 2 удален
1	r_2	B	C	a_1	a_1	$L^J(\varepsilon\cdot a_1\cdot r_2, C)$ $\forall\, L^{J/G}(\varepsilon\cdot a_1, B), \delta_G(B, r_2, C)-$ $\exists L^{J/G}(\varepsilon\cdot a_1\cdot r_2, C)$	$E(r_2)$
2	—	—	—	—	—	$L^{J/G}(\varepsilon\cdot a_1\cdot r_2, C)$	вопрос 2 удален

Из сравнения первых аргументов атомов $L^{J/G}(\_,\_)$ и $L^J(\_,\_)$ очевидно, что $\boldsymbol{L(J/G) = K^{\uparrow C}}$. Таким образом, реализовано требуемое управление, при этом не использовалось построение структуры $\boldsymbol{H'||G}$.

2.4. Обработка событий

Представление ДСС с помощью ПОФ позволяет использовать в процессе построения логического вывода информацию, поступающую из окружающей среды, а также данные о функционировании самой системы. Это можно реализовать с помощью специальных логических правил, представленных в ПОФ в виде вопросов обработки событий. Ответ на них запускает дополнительные выводы, в которых события, поступающие из окружающей среды, служат параметрами, используемыми в расчетах или других процедурах. На рисунке 9 показана ПОФ, генерирующая язык ДСС, ограниченной спецификацией, с возможностью обработки событий. Здесь $E(\sigma)$ — это атом, который определяет все события ДСС и каждый такт заменяется на реальное событие, произошедшее в системе.

В этой ПОФ оператор # — это символ, обозначающий вызов внешнего мира. Первый вопрос $\mathcal{F}_{Knl}$ служит для внешней обработки события, произошедшего в системе. Событие входит в вычислимую конструкцию $T^{\#}$ в качестве аргумента σ. $T^{\#}$ может иметь любую разумную сложность, например, это может быть некоторая логическая формула, включая ПОФ (как на рисунке 10), или любой черный ящик, выполняющий вычисления или другие процедуры обработки знаний. Атом $E^*_T(\sigma)$ в этом вопросе является результатом обработки события σ с помощью конструкции $T^{\#}$. Здесь мы используем оператор $*$ для удаления использованных атомов.

Конъюнкт *Trigger* в первом вопросе ПОФ на рисунке 10 может задавать некоторое условие. Ответ на такой заранее подготовленный вопрос вызывает процесс обработки $\sigma^{\#}$, результатом которого

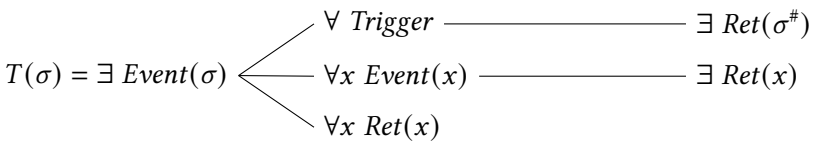


Fig. 10. PCF processing predicate T **Рис. 10.** ПОФ, обрабатывающая предикат T

будет добавление в базу атома $Ret(\sigma)$ с конкретизированным σ. Это может быть, например, «подмена» события, принудительная смена состояния или другая реконфигурация системы. Второй вопрос отрабатывает поведение системы по умолчанию, добавляя атом $Ret(\sigma)$ с тем же событием, которое поступило в подвывод, чтобы ответить на третий, целевой, вопрос и вернуть управление поиском вывода вызывающему потоку. Стратегия вывода должна быть настроена таким образом, что если ответ на первый, условный, вопрос, осуществился, то второй вопрос необходимо убрать из очереди, чтобы не добавить лишнее в данном случае необработанное событие.

3. Программная система Bootfrost, реализующая метод опровержения ПОФ

Прувер Bootfrost для исчисления ПОФ написан на языке программирования Rust. Исходный код проекта, документацию и примеры можно найти на странице GitHub[4]. Благодаря развитой системе типов Rust и модели владения гарантируется безопасность памяти и потокобезопасность. Кроме того, в Rust отсутствует среда исполнения и сборщик мусора. Эти особенности обеспечивают реализацию безопасных и эффективных систем.

Разработанный прувер основан на системе транзакций, которая позволяет регистрировать и откатывать любые изменения, происходящие в процессе поиска логического вывода. Также данная версия прувера специализирована для защищённых ПОФ. Хотя это и ограничивает класс решаемых задач, тем не менее, в прикладных задачах и задачах, связанных с динамическими системами, обычно используются только такие формулы. Защищёнными ПОФ называются такие формулы, в ти́повых условиях которых все переменные, управляемые квантором, встречаются в конъюнкте. Например, $\forall x, yA(x), B(y)$ — это защищённая подформула, поскольку переменные x, y встречаются в конъюнкте $A(x), B(y)$, при этом сами переменные также называются защищёнными, а $\forall x, yA(x), B(x)$ — это незащищённая подформула, поскольку переменная y не встречается в конъюнкте $A(x), B(x)$ и также называется незащищённой.

Стратегии, используемые в Bootfrost, делятся на три основные группы: стратегии выбора вопроса, стратегии выбора ответа и вычислимые термы и команды.

3.1. Стратегии выбора вопроса

На каждом шаге вывода прувер должен выбрать вопрос, на который он будет искать ответ. В Bootfrost для выбора вопроса используется специальная процедура. Она оценивает каждый вопрос по нескольким критериям. Эти критерии могут быть настроены пользователем, но по умолчанию используется стратегия, при которой вопрос оценивается следующим образом:

1. Является ли вопрос целевым? Целевые вопросы получают наивысшую оценку, поскольку ответ на них завершает вывод.
2. Сколько шагов вывода прошло с момента последнего ответа на вопрос? Наиболее давние вопросы получают наивысшую оценку. Такой подход обеспечивает высокий уровень разнообразия, ограничивая использование одного и того же вопроса несколько раз подряд.
3. Сколько раз на этот вопрос уже отвечали? Вопросы с наименьшим значением получают наивысшую оценку.

[4] https://github.com/snigavik/bootfrost

4. Каков уровень ветвления вопроса? Вопросы с наименьшим значением оцениваются выше.

Эта общая стратегия сортирует вопросы в порядке убывания баллов от самого высокого до самого низкого. Затем специальный метод пытается ответить на вопросы из списка, используя стратегию выбора ответа (см. следующий подраздел). Если ответ найден, то формула преобразуется по правилу вывода ω, и осуществляется переход к следующему шагу.

3.2. Стратегии выбора ответа

В прувере Bootfrost существует две стратегии выбора ответа: «Первый подходящий ответ» и «Лучший ответ».

Стратегия «Первый подходящий ответ» выбирает первый найденный ответ, удовлетворяющий заданному критерию. Такой подход позволяет эффективно использовать ресурсы памяти и процессора. Критерии могут быть настроены, по умолчанию используется тривиальная функция, возвращающая *true*, что приводит к выбору первого найденного ответа.

Стратегия «Лучший ответ» выбирает лучший ответ из множества всех возможных ответов. То есть эта стратегия находит все возможные ответы, а затем выбирает лучший из них в соответствии со специальной функцией выбора, которая может быть настроена пользователем. По умолчанию, функция просто выбирает ответ с наименьшим суммарным весом всех задействованных термов. Вес терма — это количество узлов в дереве, представляющем терм, например, терм $A(e)$ имеет вес 2, а терм $A(e, f(e))$ имеет вес 4.

Другой вспомогательной процедурой, используемой при поиске ответа, является выбор начальной точки. Существует два варианта того, с чего начинать поиск: «от последнего» и «с нуля». Метод «от последнего» означает, что следующий поиск ответа на вопрос будет начинаться в той точке, где прувер остановился во время предыдущей процедуры поиска. Метод «с нуля» означает, что процедура поиска ответа начинается с нуля, т. е. без учета предыдущих результатов поиска. Такой подход применим для немонотонных выводов и в аналогичных ситуациях, когда предыдущая история вывода может меняться с течением времени и становиться неактуальной. Например, метод «с нуля» используется для построения подъязыка $\boldsymbol{K}^{\uparrow C}$ спецификации. В ПОФ $\mathcal{F}_{Sub}$, обеспечивающей построение подъязыка, используется специальный оператор *. Если вопрос с атомом, помеченным оператором *, имеет ответ, то после применения правила вывода из базы должны быть удалены те атомы, которые участвовали в поиске подстановки с помеченным атомом. Оператор * также может быть смоделирован командой *remove-fact* (см. ниже).

3.3. Вычислимые термы и команды

Вычислимые термы (ВТермы) — это термы, обрабатываемые прувером не как синтаксические структуры, а как функции, которые должны быть вычислены. На данный момент в прувере реализованы такие ВТермы, как арифметические операции, операции сравнения, операции со списком (*in*, *notin*, *first*, *last*, конкатенация, *length*), решения (*solve*). Основная схема вычисления термов такова: специальная процедура извлекает из среды термы по их ID; затем эти термы исполняются; затем для результирующего терма, используя структуру идеального разделения (perfect sharing structure), вычисляется ID и этот ID возвращается в качестве результата.

Команды — это вычислимые термы, которые выполняются сразу после применения правила вывода ω. Например, *remove-fact* является командой. ВТермы и команды относятся к категории стратегий, поскольку их основное назначение — модификация процесса поиска логических выводов и расширение логических возможностей ПОФ. С помощью команд можно моделировать немонотонный вывод.

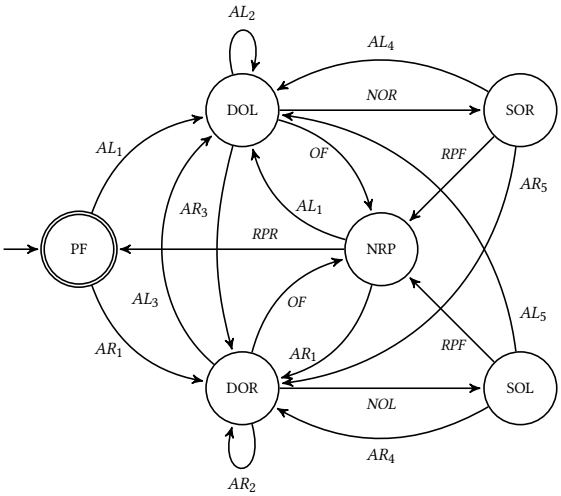


Fig. 11. Generator $\mathcal{G}$

Рис. 11. Генератор $\mathcal{G}$

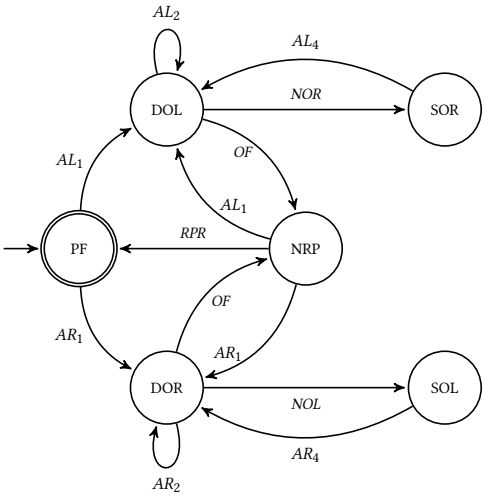


Fig. 12. Recogniser $\mathcal{H}$ of the language K

Рис. 12. Распознаватель $\mathcal{H}$ языка K

3.4. Оценка сложности

Скажем несколько слов о сложности предложенных алгоритмов на примере процедуры построения наибольшего управляемого подъязыка при проверке управляемости языка спецификации. Ответы на вопрос R_2 зависят от количества $N(\_, \_)$, добавляемых в базу. Они добавляются один раз при ответе на R_1 и столько раз, сколько переходов в произведении при ответе на R_5. Таким образом, получается не более $D+1$ копий $N(\_, \_)$. В наихудшем случае для каждого неуправляемого события будет найден ответ на R_2 с $N(\_, \_)$, так что мы получим $(D+1) * U$ шагов вывода. Каждый успешный ответ на вопрос R_2 приводит к возможности ответить на вопрос R_3 и затем R_4, т. е. всего получается $3(D+1) * U$ шагов. Если все состояния помечены на удаление, то на вопросы R_6 и R_7 можно ответить столько раз, сколько переходов в произведении. Итого, получаем $3(D+1) * U + 2D$ шагов. Таким образом, ограничив выше D по mn и U по $|\Sigma|$, найденная верхняя граница совпадает с известной оценкой $O(mn|\Sigma|)$ [5], где m и n — количество состояний автоматов $\boldsymbol{G}$ и $\boldsymbol{H}$, соответственно.

4. Иллюстративный пример

В качестве примера рассмотрим ДСС, описывающую задачу планирования пути автономного необитаемого подводного аппарата (АНПА) в неизвестной среде [50]. Предположим, что АНПА должен следовать по заданному эталонному пути, покидая его во избежание столкновения с встреченными препятствиями и возвращаясь к нему после выполнения маневров уклонения (рисунок 11). ДСС имеет следующие состояния, соответствующие режимам работы аппарата: *PF* (следование по опорному пути), *DOL* (обход обнаруженного препятствия с левой стороны), *DOR* (обход обнаруженного препятствия с правой стороны), *NRP* (навигация по опорному пути), *SOL* (поиск препятствия слева, *SOR* (поиск препятствия справа). Состояние *PF* маркировано, чтобы показать, что АНПА должен всегда возвращаться к следованию по опорному пути. События ДСС: *NOR* — «препятствий справа нет», *OF* — «обнаруженное препятствие находится далеко», *RPR* — «аппарат достиг эталонного пути» и наборы событий AL_i, AR_i, $i = \overline{1,5}$, обозначающие переключение в режимы обхода препятствий из других режимов.

Пусть спецификацию задает автомат, изображенный на рисунке 12. Стратегию поведения АНПА, предусмотренную языком спецификации $\boldsymbol{K}$, можно выразить так: стараться не попасть в место, откуда невозможно выбраться, используя стандартное препятствие. алгоритмы уклонения; не менять однажды выбранное направление обхода препятствия, пока АНПА не вернется к исходному

Table 3. DES events **Таблица 3.** События ДСС

Имя	Условие возникновения	Описание
eOLN	$R^L_{\max} < R_{\min}$	Препятствие, обнаруженное слева, находится близко
eOLNF	$R_{\min} \leqslant R^L_{\max} < R_{\min} + \Delta R$	Препятствие, обнаруженное слева, находится недалеко
eORN	$R^R_{\max} < R_{\min}$	Препятствие, обнаруженное справа, находится близко
eORNF	$R_{\min} \leqslant R^R_{\max} < R_{\min} + \Delta R$	Препятствие, обнаруженное справа, находится недалеко
eNOR	$R^R_{\max} = \infty$	Справа нет препятствий
eON	$\rho_{\min} < \rho_n$	Обнаруженное препятствие находится близко

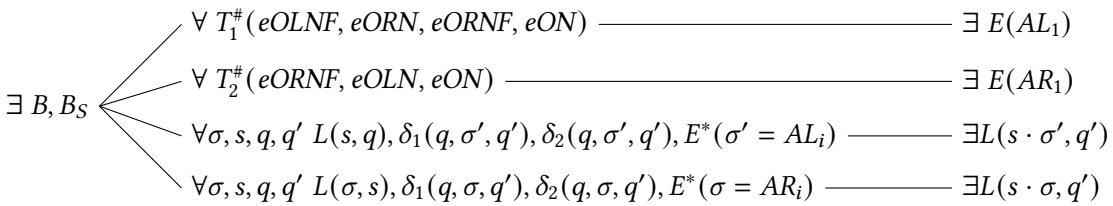


Fig. 13. PCF realizing supervisory control with composite events

Рис. 13. ПОФ, реализующая супервизорное управление с учетом композитных событий

пути (правило левой или правой руки); перестраивать путь только если это жизненно необходимо. События *RPR*, *NOR*, *NOD*, *OF* неуправляемы. Прувер Bootfrost проверяет управляемость $\boldsymbol{K}$ за 22 шага. Заметим, что любое изменение спецификации требует повторной ее проверки на управляемость, а в случае отсутствия таковой — выявления причин нарушения управляемости и построение наибольшего допустимого подъязыка спецификации, что может быть произведено с помощью представленных выше ПОФ.

После того как доказано, что спецификация управляема, можно реализовать супервизор как распознаватель языка $\boldsymbol{K}$. Его управляющее воздействие затем реализуется путем параллельной композиции автоматов, соответствующих объекту управления и супервизору. На рисунке 13 показана ПОФ, представляющая реализацию спецификации $\boldsymbol{K}$ на рисунке 12 для ДСС на рисунке 11. Здесь база представляет собой множество $\{I_1(PF), I_2(PF), \Sigma_{uc}(OF), \Sigma_{uc}(NOR), \Sigma_{uc}(NOL)\ , \Sigma_{uc}(RPR), \delta_1(PF, AL_1, DOL), \delta_1(PF, AR_1, DOR), \ldots, \delta_2(PF, AL_1, DOL)), \delta_2(PF, AR_1, DOR), \ldots\}$, состоящее из атомов объекта (множество B) и супервизора (множество B_S), и мы опускаем некоторые очевидные атомы.

Некоторые из событий AL_i, AR_i являются композиционными, т. е. определяются несколькими атомарными событиями, определяемыми условиями окружающей среды. Для учета этой информации используются предикаты T_i. Например, AL_1 как переход в режим обхода препятствия слева является результатом события *eOLNF* с одним из событий *eORN*, *eORNF* или *eON*, где *eOLNF*, *eORN*, *eORNF*, *eON* описаны в таблице 3.

Таким образом, события AL_i, AR_i могут произойти, т. е. появиться в следующих вопросах ПОФ, только в том случае, если выполняются условия их срабатывания. Оператор *, использованный с предикатом E в последних двух вопросах, показывает, что произошедшее событие должно быть удалено из базы, чтобы исключить его повторное использование машиной поиска вывода.

На рисунке 14 представлена схема реализации супервизорного управления в составе иерархической системы управления группой мобильных роботов. Система управления разрабатывается на базе роботизированного стенда, состоящего из роботов Lego Mindstorms EV3, перемещающихся по специальному полю, и набора камер, считывающих положение роботов и других объектов на поле. Программное обеспечение распространяется по беспроводной сети между роботами и сервером.

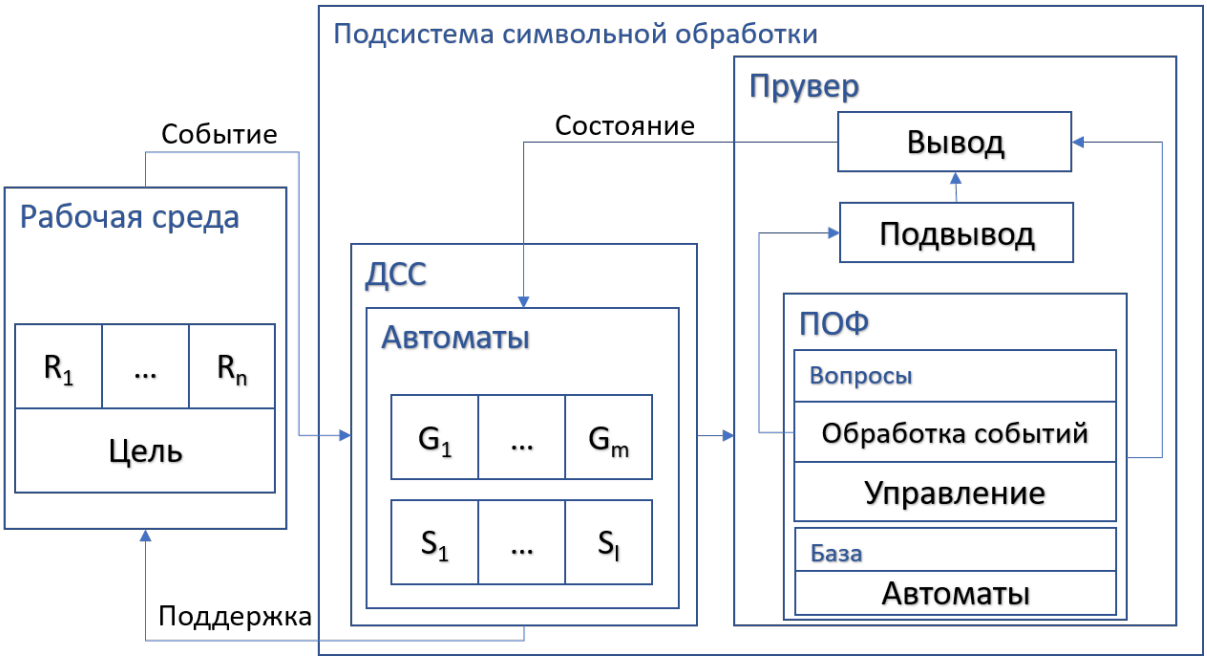


Fig. 14. Implementation of supervisory control in control system

Рис. 14. Реализация супервизорного управления как часть системы управления

На рисунке 14 реальная рабочая среда роботов представлена блоком «Рабочая среда». События, происходящие в рабочей среде, распознаются, маркируются заранее заданными символами и подаются в подсистему символьной обработки данных.

Заключение

В статье был представлен разработанный подход к решению основных задач теории супервизорного управления логическими ДСС с помощью представления таких систем в виде ПОФ и применения метода опровержения полученных ПОФ. Логический вывод в исчислении ПОФ осуществляется программной системой Bootfrost. Областью применения разработанного подхода являются системы управления техническими комплексами, в первую очередь робототехническими. Дальнейшие исследования будут направлены на решение задач управления децентрализованными и распределенными системами, в том числе частично наблюдаемыми ДСС, а также на реализацию описанного подхода в современных робототехнических системах.

References

[1] S. Lafortune, “Discrete event systems: Modeling, observation, and control”, *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 2, pp. 141–159, 2019. DOI: 10.1146/annurev-control-053018-023659.

[2] C. Seatzu, M. Silva, and J. H. Van Schuppen, Eds., *Control of discrete-event systems*. Springer London, 2013, 480 pp. DOI: 10.1007/978-1-4471-4276-8.

[3] W. M. Wonham and K. Cai, *Supervisory Control of Discrete-Event Systems*. Springer International Publishing, 2019, 487 pp.

[4] P. J. Ramadge and W. M. Wonham, “Supervisory control of a class of discrete event processes”, *SIAM Journal on Control and Optimization*, vol. 25, no. 1, pp. 206–230, 1987. DOI: 10.1137/0325013.

[5] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*. Springer Cham, 2021, 804 pp. DOI: 10.1007/978-3-030-72274-6.

[6] W. M. Wonham, K. Cai, and K. Rudie, "Supervisory control of discrete-event systems: A brief history", *Annual Reviews in Control*, vol. 45, pp. 250–256, 2018. DOI: 10.1016/j.arcontrol.2018.03.002.

[7] A. Jayasiri, G. Mann, and R. Gosine, "Behavior coordination of mobile robotics using supervisory control of fuzzy discrete event systems", *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 41, no. 5, pp. 1224–1238, 2011.

[8] C. R. Torrico, A. B. Leal, and A. T. Watanabe, "Modeling and supervisory control of mobile robots: A case of a sumo robot", *IFAC-Papers OnLine*, vol. 49, no. 32, pp. 240–245, 2016.

[9] X. Dai, L. Jiang, and Y. Zhao, "Cooperative exploration based on supervisory control of multi-robot systems", *Applied Intelligence*, vol. 45, no. 1, pp. 18–29, 2016.

[10] A. Tsalatsanis, A. Yalcin, and K. P. Valavanis, "Dynamic task allocation in cooperative robot teams", *Robotica*, vol. 30, no. 5, pp. 721–730, 2012. DOI: 10.1017/S0263574711000920.

[11] R. C. Hill and S. Lafortune, "Scaling the formal synthesis of supervisory control software for multiple robot systems", in *2017 American Control Conference (ACC)*, 2017, pp. 3840–3847. DOI: 10.23919/ACC.2017.7963543.

[12] G. W. Gamage, G. K. I. Mann, and R. G. Gosine, "Discrete event systems based formation control framework to coordinate multiple nonholonomic mobile robots", in *Proceedings of the 2009 IEEE RSJ International Conference on Intelligent Robots and Systems*, 2009, pp. 4831–4836.

[13] Y. K. Lopes, S. M. Trenkwalder, A. B. Leal, T. J. Dodd, and R. Groß, "Supervisory control theory applied to swarm robotics", *Swarm Intelligence*, vol. 10, no. 1, pp. 65–97, 2016.

[14] F. J. Mendiburu, M. R. Morais, and A. M. Lima, "Behavior coordination in multi-robot systems", in *2016 IEEE International Conference on Automatica (ICA-ACCA)*, IEEE, 2016, pp. 1–7.

[15] T. Hales *et al.*, "A formal proof of the Kepler conjecture", in *Forum of mathematics, Pi*, Cambridge University Press, vol. 5, 2017, e2. DOI: doi:10.1017/fmp.2017.1.

[16] G. Klein *et al.*, "Sel4: Formal verification of an os kernel", in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 207–220.

[17] G. Gonthier *et al.*, "Formal proof–the four-color theorem", *Notices of the AMS*, vol. 55, no. 11, pp. 1382–1393, 2008.

[18] X. Leroy, "Formal verification of a realistic compiler", *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[19] D. A. Kondratyev and A. V. Promsky, "The complex approach of the C-lightVer system to the automated error localization in C-programs", *Automatic Control and Computer Sciences*, vol. 54, no. 7, pp. 728–739, 2020. DOI: 10.3103/S0146411620070093.

[20] J. S. Moore, "Milestones from the pure lisp theorem prover to ACL2", *Formal Aspects of Computing*, vol. 31, no. 6, pp. 699–732, 2019. DOI: 10.1007/s00165-019-00490-3.

[21] E. Karpas and D. Magazzeni, "Automated planning for robotics", *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 3, pp. 417–439, 2020.

[22] Z. Zombori, J. Urban, and C. E. Brown, "Prolog technology reinforcement learning prover", in *International Joint Conference on Automated Reasoning*, Springer, 2020, pp. 489–507.

[23] M. Schader and S. Luke, "Planner-guided robot swarms", in *International Conference on Practical Applications of Agents and Multi-Agent Systems*, Springer, 2020, pp. 224–237.

[24] W. Li, A. Miyazawa, P. Ribeiro, A. Cavalcanti, J. Woodcock, and J. Timmis, "From formalised state machines to implementations of robotic controllers", in *Distributed Autonomous Robotic Systems: the 13th International Symposium*, Springer, 2018, pp. 517–529.

[25] S. N. Vassilyev, "Machine synthesis of mathematical theorems", *The Journal of Logic Programming*, vol. 9, no. 2-3, pp. 235–266, 1990. DOI: 10.1016/0743-1066(90)90042-4.

[26] A. K. Zherlov, S. N. Vassilyev, E. A. Fedosov, and B. E. Fedunov, *Intelligent control of dynamic systems*. Fizmatlit, 2000, 351 pp., In Russian.

[27] S. Vassilyev and G. Ponomarev, "Automation methods for logical derivation and their application in the control of dynamic and intelligent systems", *Proceedings of the Steklov Institute of Mathematics*, vol. 276, pp. 161–179, 2012.

[28] S. Vassilyev and A. Galyaev, "Logical-optimization approach to pursuit problems for a group of targets", *Doklady Mathematics*, vol. 95, pp. 299–304, 3 2017.

[29] F. F. Reijnen, T. R. Erens, J. M. van de Mortel-Fronczak, and J. E. Rooda, "Supervisory controller synthesis and implementation for safety PLCs", *Discrete Event Dynamic Systems*, vol. 32, no. 1, pp. 115–141, 2022.

[30] D. Bohlender and S. Kowalewski, "Compositional verification of PLC software using horn clauses and mode abstraction", *IFAC-PapersOnLine*, vol. 51, pp. 428–433, Jan. 2018. DOI: 10.1016/j.ifacol.2018.06.336.

[31] D. Bohlender and S. Kowalewski, "Leveraging horn clause solving for compositional verification of PLC software", *Discrete Event Dynamic Systems*, vol. 30, no. 1, pp. 1–24, 2020. DOI: 10.1007/s10626-019-00296-8.

[32] N. Bjørner and L. Nachmanson, "Navigating the universe of Z3 theory solvers", in *Formal Methods: Foundations and Applications*, Springer International Publishing, 2020, pp. 8–24. DOI: https://doi.org/10.1007/978-3-030-63882-5_2.

[33] A. D. Vieira, E. A. P. Santos, M. H. de Queiroz, A. B. Leal, A. D. de Paula Neto, and J. E. R. Cury, "A method for PLC implementation of supervisory control of discrete event systems", *IEEE Transactions on Control Systems Technology*, vol. 25, no. 1, pp. 175–191, 2017. DOI: 10.1109/TCST.2016.2544702.

[34] J. Thistle and W. Wonham, "Control problems in a temporal logic framework", *International Journal of Control*, vol. 44, no. 4, pp. 943–976, 1986. DOI: 10.1080/00207178608933645.

[35] B. C. Rawlings, S. Lafortune, and B. E. Ydstie, "Supervisory control of labeled transition systems subject to multiple reachability requirements via symbolic model checking", *IEEE Transactions on Control Systems Technology*, vol. 28, no. 2, pp. 644–652, 2020. DOI: 10.1109/TCST.2018.2877621.

[36] K. T. Seow, "Supervisory control of fair discrete-event systems: A canonical temporal logic foundation", *IEEE Transactions on Automatic Control*, vol. 66, no. 11, pp. 5269–5282, 2021. DOI: 10.1109/TAC.2020.3037156.

[37] S. Jiang and R. Kumar, "Supervisory control of discrete event systems with CTL* temporal logic specifications", *SIAM Journal on Control and Optimization*, vol. 44, no. 6, pp. 2079–2103, 2006. DOI: 10.1137/S0363012902409982.

[38] G. Aucher, "Supervisory control theory in epistemic temporal logic", 2014, pp. 333–340.

[39] K. Ritsuka and K. Rudie, "Do what you know: Coupling knowledge with action in discrete-event systems", *Discrete Event Dynamic Systems*, vol. 33, pp. 257–277, 2023. DOI: 10.1007/s10626-023-00381-z.

[40] X. Geng, D. Ouyang, and C. Han, "Verifying diagnosability of discrete event system with logical formula", *Chinese Journal of Electronics*, vol. 29, pp. 304–311, 2020. DOI: 10.1049/cje.2020.01.008.

[41] L. Feng and W. M. Wonham, "TCT: A computation tool for supervisory control synthesis", in *Proceedings of the 8th International Workshop on Discrete Event Systems*, IEEE, 2006, pp. 388–389.

[42] L. Ricker, S. Lafortune, and S. Genc, "DESUMA: A tool integrating GIDDES and UMDES", in *2006 8th International Workshop on Discrete Event Systems*, 2006, pp. 392–393. DOI: 10.1109/WODES.2006.382402.

[43] R. Malik, K. Åkesson, H. Flordal, and M. Fabian, "Supremica — an efficient tool for large-scale discrete event systems", *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 5794–5799, 2017. DOI: 10.1016/j.ifacol.2017.08.427.

[44] K. Åkesson, H. Flordal, and M. Fabian, "Exploiting modularity for synthesis and verification of supervisors", *IFAC Proceedings Volumes*, vol. 35, no. 1, pp. 175–180, 2002.

[45] B. Brandin, R. Malik, and P. Malik, "Incremental verification and synthesis of discrete-event systems guided by counter examples", *IEEE Transactions on Control Systems Technology*, vol. 12, no. 3, pp. 387–401, 2004.

[46] S. Mohajerani, R. Malik, and M. Fabian, "A framework for compositional synthesis of modular nonblocking supervisors", *IEEE Transactions on Automatic Control*, vol. 59, no. 1, pp. 150–162, 2013.

[47] N. Bourbaki, *Theory of Sets*. Hermann, 1968, 414 pp.

[48] A. Larionov, A. Davydov, and E. Cherkashin, "The method for translating first-order logic formulas into positively constructed formulas", *Software & Systems*, vol. 32, no. 4, pp. 556–564, 2019, In Russian. DOI: 10.15827/0236-235X.128.556-564.

[49] A. Davydov, A. Larionov, and E. Cherkashin, "On the calculus of positively constructed formulas for automated theorem proving", *Automatic Control and Computer Sciences*, vol. 45, no. 7, pp. 402–407, 2011. DOI: 10.3103/s0146411611070054.

[50] S. Ulyanov, I. Bychkov, and N. Maksimkin, "Event-based path-planning and path-following in unknown environments for underactuated autonomous underwater vehicles", *Applied Sciences*, vol. 10, no. 21, p. 7894, 2020. DOI: 10.3390/app10217894.

MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 31, NO. 1, 2024
journal homepage: www.mais-journal.ru

THEORY OF DATA

Minimal Coverage of Generalized Typed Inclusion Dependencies in Databases

S. V. Zykin[1]

DOI: 10.18255/1818-1015-2024-1-78-89

[1]Sobolev institute of mathematics SB RAS, Novosibirsk, Russia

MSC2020: 68P15
Research article
Full text in Russian

Received February 12, 2024
Revised February 26, 2024
Accepted February 28, 2024

The paper discusses the theory and algorithms necessary to construct a minimal covering of generalized typed inclusion dependencies. Traditionally, the construction of minimal covering is used for all types of dependencies in order to obtain a non-redundant and consistent database design. Generalized inclusion dependencies correspond to referential integrity constraints, when several main and several external relations are involved in one constraint, which corresponds to an ultragraph edge. In previous work, based on a study of the properties of dependencies, a system of axioms was presented with proof of consistency and completeness. In this work, the closures were studied for generalized typed inclusion dependencies. An algorithm for constructing closures has been developed and its correctness has been proven. The results obtained are further used to develop an algorithm for constructing a minimal covering. At the end of the article, examples are presented that demonstrate the operation of the algorithms.

Keywords: database; generalized inclusion dependencies; minimal covering

INFORMATION ABOUT THE AUTHORS

Zykin, Sergey V. (corresponding author) | ORCID iD: 0000-0002-0576-2149. E-mail: szykin@mail.ru
Leading researcher, Dr. Sc.

Funding: State task IM SB RAS, project No. FWNF-2022-0016.

For citation: S. V. Zykin, "Minimal coverage of generalized typed inclusion dependencies in databases", *Modeling and Analysis of Information Systems*, vol. 31, no. 1, pp. 78–89, 2024. DOI: 10.18255/1818-1015-2024-1-78-89.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 31, № 1, 2024
сайт журнала: www.mais-journal.ru

THEORY OF DATA

Минимальное покрытие обобщенных типизированных зависимостей включения в базах данных

С. В. Зыкин[1]

DOI: 10.18255/1818-1015-2024-1-78-89

[1]Институт математики им. С.Л. Соболева СО РАН, Новосибирск, Россия

УДК 004.652.4
Научная статья
Полный текст на русском языке

Получена 12 февраля 2024 г.
После доработки 26 февраля 2024 г.
Принята к публикации 28 февраля 2024 г.

В статье рассматривается теория и алгоритмы, необходимые для построения минимального покрытия обобщенных типизированных зависимостей включения. Традиционно аппарат построения минимальных покрытий используется для всех видов зависимостей с целью получения не избыточного и непротиворечивого проекта базы данных. Обобщенные зависимости включения соответствуют ссылочным ограничениям целостности, когда в одном ограничении участвуют несколько главных и несколько внешних отношений, что соответствует ребру ультраграфа. В предыдущей работе на основе исследования свойств зависимостей представлена система аксиом с доказательством непротиворечивости и полноты. В данной работе проведены исследования замыканий для обобщенных типизированных зависимостей включения. Разработан алгоритм построения замыканий, доказана его корректность. Полученные результаты далее используются для разработки алгоритма построения минимального покрытия. В конце статьи представлены примеры, которые демонстрируют работу алгоритмов.

Ключевые слова: база данных; обобщенные зависимости включения; минимальное покрытие

ИНФОРМАЦИЯ ОБ АВТОРАХ

Зыкин, Сергей Владимирович (автор для корреспонденции)	ORCID iD: 0000-0002-0576-2149. E-mail: szykin@mail.ru Вед. науч. сотр., доктор техн. наук

Финансирование: Госзадание ИМ СО РАН, проект № FWNF-2022-0016.

Для цитирования: S. V. Zykin, “Minimal coverage of generalized typed inclusion dependencies in databases”, *Modeling and Analysis of Information Systems*, vol. 31, no. 1, pp. 78–89, 2024. DOI: 10.18255/1818-1015-2024-1-78-89.

Введение

Теоретической основой ссылочных ограничений целостности в базах данных являются зависимости включения. Они позволяют получить согласованное и не избыточное множество связей на схеме базы данных (БД) за счет построения минимальных покрытий.

Развитию теории зависимостей включения в настоящее время уделяется внимание с точки зрения взаимодействия с другими видами зависимостей в БД. С другой стороны, вопросы исследования новых видов ссылочных ограничений, которые предлагает практика, не потеряли своей актуальности.

Введем следующие обозначения: пусть $U = \{A_1, A_2, \ldots, A_n\}$ — множество атрибутов, R_i — отношения (таблицы) в БД, которые определены на множестве U, $[R_i]$ — схема отношения R_i (атрибуты отношения R_i), $[R_i] \subseteq U, 1 \leq i \leq k, R = (R_1, R_2, \ldots, R_k)$ — БД, $S = \{[R_1], [R_2], \ldots, [R_k]\}$ — схема БД [1, 2], для которой не заданы ссылочные ограничения целостности. Пусть $R_i[X]$ — проекция отношения R_i по атрибутам X и $t_i[X]$ кортеж проекции $R_i[X]$.

Впервые определение зависимости включения было представлено в работе [3].

Определение 1. Пусть R_i и R_j — отношения БД (возможно совпадающие), для которых заданы два множества атрибутов $V \subseteq [R_i]$ и $W \subseteq [R_j]$, с одинаковой мощностью. Тогда соотношение $R_i[V] \subseteq R_j[W]$ называется зависимостью включения.

Замечание. Если имеет место равенство $V = W$, то зависимость в определении 1 является типизированной (typed), в противном случае — нетипизированной. Для типизированных зависимостей атрибут $A_i \in V$ соответствует атрибуту $A_i \in W$ независимо от места расположения во множествах V и W. Для нетипизированных зависимостей первый атрибут по порядку множества V соответствует первому атрибуту множества W, второй – второму и т. д. В этом заключается принципиальное различие этих двух видов зависимостей. Следует заметить, что в языке SQL (Structured Query Language) атрибут идентифицируется своим именем, а не местом расположения в записи, что соответствует типизированным зависимостям.

Проблема построения проекта БД исследуется достаточно давно. Общий подход заключается в использовании зависимостей данных, которые подвергаются проверке на избыточность, противоречивость и полноту. После аксиоматизации появляется возможность использования зависимостей для построения минимальных покрытий, которые затем используются при формировании схемы БД. Основной проблемой при этом становится выводимость. Препятствием является взаимодействие зависимостей включения с функциональными зависимостями.

Неоднократно исследователями осуществлялись попытки решить проблему выводимости совместно для функциональных зависимостей и зависимостей включения. Однако, отсутствие полной системы аксиом не позволяет решить данную проблему [3—5], за исключением некоторых частных случаев, где полная аксиоматизация [6, 7] была получена. В работе [8] сформулированы условия существования нормальной формы зависимостей включения (IDNF), которая основана на нормальной форме Бойса-Кодда (BKNF), дополненная условием ацикличности.

В работе [9] рассматривается задача совместной аксиоматизации функциональных зависимостей, зависимостей включения и частного случая многозначных зависимостей, у которых компоненты базиса названы независимыми атомами. В результате исследований было получено, что конечные и неограниченные задачи выводимости для комбинированного класса атомов, унарных функциональных зависимостей и унарных зависимостей включения аксиоматизируемы и разрешимы за полиномиальное время.

Большое количество исследований было посвящено разработке алгоритмов определения обычных и условных зависимостей включения в существующих БД [10—16]. Для этого используются различные источники информации, например, свойства схемы БД, трудоемкие алгоритмы анализа

сохраненных данных в БД, обработка текущих запросов к БД. Такие подходы требуют дополнительной проверки результатов, поскольку полная автоматизация нереализуема из-за специфики бизнес-правил в приложениях. Польза таких исследований в том, что частично автоматизируется трудоемкий процесс поиска неизвестных зависимостей включения.

Практически значимые исследования посвящены неопределенным значениям в БД. Выводимость при наличии неопределенных значений исследуется в работах [17—20]. Взаимодействие функциональных зависимостей и нетипизированных зависимостей включения стало препятствием для получения полной и непротиворечивой системы аксиом (решение есть только для унарных зависимостей). Однако, удаление семантических несоответствий, прежде всего синонимии, на этапе проектирования БД позволяет использовать типизированные зависимости включения [20], которые не взаимодействуют с функциональными зависимостями.

В данной работе рассматривается проблема выводимости для обобщенных типизированных зависимостей включения, для которых в работе [21] получена полная и непротиворечивая система аксиом.

1. Предварительные сведения

В работе [21] рассмотрен практический пример схемы БД, в котором одной связью объединяются несколько отношений. Эта связь содержит несколько главных отношений, первичные ключи которых совпадают, и несколько внешних отношений, в которых первичные ключи главных отношений становятся внешними ключами. При этом внешние ключи могут принимать только те значения, которые уже есть хотя бы в одном главном отношении. В примере также показано, что практический смысл имеет ситуация, когда атрибуты внешних ключей принимают неопределенные значения (*Null*), если они не являются компонентами первичного ключа во внешнем отношении. Заметим, что с точки зрения теории ссылочная целостность не обязательно должна опираться на первичные ключи, это может быть любой другой набор атрибутов главного отношения, допускающий наличие неопределенных значений.

Чтобы учитывать неопределенные значения, в работе [20] было предложено условие соответствия, которое устанавливает частичный порядок для кортежей.

Определение 2. Кортеж $t_i[X]$ *соответствует* кортежу $t_j[X]$ для множества атрибутов X ($t_j[X] \preccurlyeq t_i[X]$):

- если $t_i[A_l] \neq Null$, тогда $t_j[A_l] = t_i[A_l]$ либо $t_j[A_l] = Null$;
- если $t_i[A_l] = Null$, тогда $t_j[A_l] = Null$,

для каждого атрибута $A_l \in X$.

Соответствие кортежей в определении 2 удовлетворяет условию транзитивности: если $t_j[X] \preccurlyeq t_i[X]$ и $t_i[X] \preccurlyeq t_m[X]$, тогда $t_j[X] \preccurlyeq t_m[X]$. На основе определения 2 в работе [20] вводится определение типизированной зависимости включения.

Определение 3. Типизированная зависимость включения $\sigma = R_j[X] \subseteq R_i[X]$ между главным отношением $R_i[X]$ и внешним отношением $R_j[X]$ по атрибутам X выполнена, если для любого кортежа $t_j[X] \in R_j[X]$ имеется соответствующий кортеж $t_i[X]$ в отношении $R_i[X]$.

Из определения 3 следует, что кортеж $t_j[X] \in R_j[X]$ может иметь несколько соответствующих кортежей в отношении $R_i[X]$. Когда происходит замена неопределенных значений в кортеже $t_j[X]$, то в $R_i[X]$ должен присутствовать, по крайней мере, один соответствующий кортеж, что ограничивает варианты замены.

В работе [21] введено обобщение типизированных зависимостей включения: $W[X] \subseteq V$, где W — множество внешних отношений зависимости, V — множество главных отношений зависимости.

Оба множества отношений зависимости определены на наборе атрибутов X, поэтому достаточно указать X только слева.

Определение 4. Обобщенная типизированная зависимость включения $\sigma = W[X] \subsetneq V$ между главными отношениями V и внешними отношениями W по атрибутам X выполнена, если для каждого отношения $R_i \in W$ и каждого кортежа $t_i \in R_i$ существует отношение $R_j \in V$ и существует соответствующий кортеж $t_j \in R_j$, то есть выполнено условие: $t_i[X] \preccurlyeq t_j[X]$.

Приведенное условие в определении 4 соответствует ссылочной целостности в БД.

На основании исследования свойств зависимостей в [21] была получена следующая система аксиом.

Аксиома 1 (рефлексия). Если $W \subseteq V$, тогда $W[X] \subsetneq V$.

Аксиома 2 (проекция). Если $W[X] \subsetneq V$ и $Y \subseteq X$, тогда $W[Y] \subsetneq V$.

Аксиома 3 (объединение). Если $W[X] \subsetneq V$ и $S[X] \subsetneq V$, тогда $\{W \cup S\}[X] \subsetneq V$.

Аксиома 4 (транзитивность). Если $W[X] \subsetneq S$ и $S[X] \subsetneq V$, тогда $W[X] \subsetneq V$.

Здесь V, W, S — множества отношений БД, X, Y — множества атрибутов.

Для представленной системы аксиом были доказаны непротиворечивость (все выводимые за счет аксиом зависимости выполнимы) и полнота (все выполнимые зависимости выводимы).

С использованием аксиом 1–4 получены следующие правила, которые далее используются при построении замыканий.

Правило 1 (декомпозиция). Если $W[X] \subsetneq V$ и $S \subseteq W$, тогда $S[X] \subsetneq V$.

Действительно, применив сначала аксиому 1, а затем аксиому 4 получим результат.

Правило 2 (пополнение). Если $W[X] \subsetneq V$ и $V \subseteq S$, тогда $W[X] \subsetneq S$.

Заметим, правило 2 позволяет дополнять в правую часть зависимости произвольное количество отношений. В [21] это правило использовано при доказательстве полноты системы аксиом 1–4 (теорема 2).

В работе [21] доказаны следующие две теоремы.

Теорема 1 (надежность). *Система аксиом 1–4 надежна.*

Теорема 2 (полнота). *Система аксиом 1–4 полна.*

При построении замыканий важными являются свойства, которые не выполнены для обобщенных зависимостей включения.

Утверждение 1. Объединение двух (и более) правых частей зависимостей, даже если они имеют непустое пересечение, но не совпадают друг с другом, не является эквивалентным преобразованием:

$$W[X] \subsetneq S \text{ и } W[X] \subsetneq V \text{ не эквивалентно } W[X] \subsetneq \{V \cup V\}.$$

Действительно, пусть $S \not\subseteq V$ и $V \not\subseteq S$. Допустим, что $R_i \in S$ и $R_i \notin V$. Тогда в зависимости $W[X] \subsetneq \{V \cup V\}$ отсутствует необходимость принадлежности кортежей W одному из отношений V, что противоречит зависимости $W[X] \subsetneq V$. Аналогично доказывается следующее утверждение.

Утверждение 2. Из зависимости $W[X] \subsetneq S$ не следует $W[X] \subsetneq V$, если $V \subset S$.

Из этого можно сделать вывод, что правая часть обобщенной зависимости включения не допускает удаления отношений.

2. Замыкание множества зависимостей включения

Для поиска избыточных зависимостей традиционно используется построение замыканий [1, 20]. Для обнаружения зависимостей $\sigma = W[X] \subsetneq V$, которые являются логическим следствием множества обобщенных зависимостей включения Σ, используется выводимость $\Sigma \vdash \sigma$ за счет системы аксиом 1–4. Поскольку система аксиом является полной и непротиворечивой, то гарантируется выполнение выводимой зависимости σ, если БД удовлетворяет множеству зависимостей Σ. В этом случае зависимость σ можно удалить.

Рассмотрим определение и структуру замыкания множества типизированных обобщенных зависимостей включения.

Определение 5. Замыканием $V^+[X]$ множества отношений V по атрибутам X на зависимостях Σ будем называть множества отношений $W_1, W_2, \ldots, W_k$ таких, что зависимости $W_i[X] \subsetneq V$, $i = \overline{1,k}$, выводимы из Σ с использованием аксиом 1–4, другими словами $\Sigma \vdash W_i[X] \subsetneq V$, $i = \overline{1,k}$.

Оперирование множеством множеств отношений в замыкании затруднит определение выводимых зависимостей, поскольку остается открытым вопрос о выводимости за счет объединения подмножеств множеств отношений, уже находящихся в замыкании. Следующая лемма дает ответ на этот вопрос и существенно упрощает процесс построения замыкания.

Лемма 1. *Множества отношений $W_1, W_2, \ldots, W_k$, выводимые по атрибутам X из одного и того же множества отношений V, в замыкании $V^+[X]$ объединяются.*

Доказательство. Допустим, множества W_1 и W_2 принадлежат замыканию $V^+[X]$ и $W = \{W_1 \cup W_2\}$. По аксиоме объединения 3 выводима зависимость $W[X] \subsetneq V$. По аксиоме транзитивности 4 из V выводимо любое множество отношений S, для которого имеет место зависимость $S[X] \subsetneq \{W_1 \cup W_2\}$. По правилу 2 (пополнение) из множества $\{W_1 \cup W_2\}$ выводимо все, что выводимо из множеств W_1 и W_2 по отдельности. Следовательно, множества W_1 и W_2 в замыкании $V^+[X]$ можно заменить одним множеством W. Аналогичным способом к W присоединяются остальные множества $W_i[X]$, $i = \overline{3,k}$. □

Следствием леммы 1 является возможность представления замыкания $V^+[X]$ в виде одного множества отношений БД, что существенно упрощает его формирование и дальнейшее использование. Для проверки выводимости обобщенных зависимостей включения будем использовать аппарат построения замыканий [1, 20], адаптировав его для обобщенных типизированных зависимостей включения.

Построим функцию, реализующую алгоритм формирования замыкания. В качестве текущего замыкания и результата функции будем использовать список отношений $V^*[X]$. Входными параметрами функции являются: Σ^* — текущее множество зависимостей, на котором строится замыкание; V — исходное множество (список) отношений; X — множество атрибутов, по которому строится замыкание.

function CLOS_M(Σ^*,V,X)
CLOS_M $\leftarrow \emptyset$
$V^*[X] \leftarrow \emptyset$
for each R_i **from** V
  if $X \subseteq [R_i]$ **then** $V^*[X] \leftarrow V^*[X] \cup R_i$
end for
if $V^*[X] = \emptyset$ **then exit function**
insert $\leftarrow$ TRUE
while *insert*

insert ← FALSE
for each $W_l[Y] \subsetneq V_l$ **from** Σ^*
if $V_l \subseteq V^*[X]$ **and** $W_l \notin V^*[X]$ **and** $X \subseteq Y$ **then**
$V^*[X] \leftarrow V^*[X] \cup W_l$
insert ← TRUE
end if
end for
end while
CLOS_M ← $V^*[X]$
end function

Получим оценку вычислительной сложности алгоритма, реализованного функцией *CLOS_M*. Основной цикл *while* по своей организации является бесконечным. Однако, для выполнения очередной итерации необходимо дополнение к замыканию не менее одного множества отношений W_l. Следовательно, верхней границей количества итераций в алгоритме является произведение nk, где n — мощность множества зависимостей Σ^*, k — количество отношений в БД (при добавлении отношений к замыканию на каждой итерации).

Теорема 3 (замыкание). *Функция* CLOS_M *корректно формирует замыкание* $V^+[X]$.

Доказательство. Рассмотрим произвольное множество отношений V и произвольное множество атрибутов X. В соответствии с леммой 1 замыкание представляет собой одно множество отношений. Компонентами этого множества являются отдельные отношения R_j. В теореме надо показать: 1) если $R_j \in V^*[X]$, тогда $R_j \in V^+[X]$, и 2) если $R_j \in V^+[X]$, тогда $R_j \in V^*[X]$. Другими словами $V^+[X]$ равно $V^*[X]$.

1. Покажем, что каждое отношение R_j, принадлежащее $V^*[X]$, также принадлежит $V^+[X]$. Замыкание $V^*[X]$ формируется в трех операторах:

a. $V^*[X] \leftarrow \emptyset$ ($V^*[X]$ равно $\emptyset$). Начальное множество $V^*[X]$ не содержит ни одного отношения, что может стать результатом построения замыкания, если в X есть атрибуты, которые отсутствуют в каждом отношении множества V. По аксиомам **A1-A4** в этом случае также ничего не выводимо: $V^+[X]$ равно $\emptyset$.
b. Далее в цикле к замыканию $V^*[X]$ присоединяются отношения R_i такие, что $R_i \in V$ и $X \subseteq [R_i]$. Замыкание $V^+[X]$ также будет содержать R_i, поскольку зависимость $R_i[X] \subsetneq V$ выводима по аксиоме рефлексии 1.
c. Для текущей зависимости $W_l[X] \subsetneq V_l$ к замыканию $V^*[X]$ присоединяется новое множество W_l: $V^*[X]$ равно $V^*[X] \cup W_l$. При этом должны быть выполнены условия: $V_l \subseteq V^*[X]$, $W_l \notin V^*[X]$ и $X \subseteq Y$. Пусть $R_i \in W_l \setminus V^*[X]$ одно из отношений, которое будет дополнено к $V^*[X]$. Используя индукцию, покажем, что $R_i \in V^+[X]$. В качестве базиса используем вариант (b). Предположим, что все отношения в $V^*[X]$ до рассмотрения зависимости $W_l[X] \subsetneq V_l$ принадлежат $V^+[X]$. Поскольку зависимость $W_l[X] \subsetneq V_l$ выводима по аксиоме проекции 2, зависимость $R_i[X] \subsetneq V_l$ выводима по правилу 1 (декомпозиция) и зависимость $V_l[X] \subsetneq V$ выводима в силу леммы 1, то по аксиоме транзитивности 4 имеем $R_i \in V^+[X]$. Следовательно, $V^*[X] \subseteq V^+[X]$.

2. Допустим, что отношение R_i принадлежит замыканию $V^+[X]$. В этом случае существует последовательность вывода, состоящая из k строк, в которой последней строке соответствует зависимость $W_k[X] \subsetneq V$ и $R_i \in W_k$. Пусть k равно 1. Единственной аксиомой, которая может быть использована при изначально пустом множестве $V^+[X]$, является аксиома рефлексии 1, $V^+[X] = W_1$, $W_1 \subseteq V$ и отношения множества W_1 определены на атрибутах X. В алгоритме отношения множества W_1 будут присоединены к $V^*[X]$ в первом цикле *for*. Допустим, что выводимые зависимости $W_j[X] \subsetneq V$, $j = \overline{1, k-1}$, удовлетворяют условию $W_j \subseteq V^*[X]$. На шаге k множество W_k было присоединено

к замыканию $V^+[X]$. Присоединение W_k не могло быть сделано за счет аксиомы рефлексии 1, так как множество W_k уже содержалось бы в $V^*[X]$. Аналогичная ситуация с аксиомой объединения 3: если W_k получено объединением двух множеств, то оба эти множества уже содержатся в $V^*[X]$, поскольку цепочка их вывода не превышает $k-1$ шагов.

Использование аксиомы проекции 3 подразумевает наличие зависимости $W_k[Y] \subsetneq V_l$, для которой $X \subseteq Y$ и $V_l \subseteq V^+[X]$. Сформулированные условия согласуются с условиями оператора *IF* во втором операторе цикла *for*. Следовательно, по алгоритму множество W_k будет присоединено к $V^*[X]$.

Пусть на шаге k при построении $V^+[X]$ была использована аксиома транзитивности 4 для зависимостей $W_k[X] \subsetneq S$ и $S[X] \subsetneq V_l$. Множества отношений V_l и S принадлежат $V^+[X]$, следовательно, они принадлежат $V^*[X]$, поскольку цепочка их вывода не превышает $k-1$ шагов. Перечисленные условия не противоречат условиям оператора *IF* во втором операторе цикла *for*. Следовательно, по алгоритму множество W_k будет присоединено к $V^*[X]$, что доказывает соотношение $V^+[X] \subseteq V^*[X]$. □

3. Построение минимального покрытия множества зависимостей включения

В настоящее время отсутствуют СУБД, в которых была бы реализована поддержка обобщенных зависимостей включения. Однако, с учетом реализации обычных зависимостей включения (по одному отношению справа и слева в зависимости) можно предположить, что реализация ссылочной целостности, соответствующей обобщенным зависимостям включения, потребует дополнительной памяти для индексных файлов и дополнительного времени для обслуживания этих файлов. Следовательно, актуальной задачей является минимизация множества Σ, причем, избыточными могут быть сами зависимости и компоненты этих зависимостей.

Допустимые состояния БД останутся прежними после удаления выводимых зависимостей, поскольку их выполнимость будет гарантирована оставшимися зависимостями.

Определение 6. Два множества зависимостей Σ и Σ^* будем считать эквивалентными, если любая зависимость $W[X] \subsetneq V$, выводимая в Σ, также выводима в Σ^*, и любая зависимость $W^*[X] \subsetneq V^*$, выводимая в Σ^*, также выводима в Σ.

Не сложно заметить, что из определения 6 следует очевидное утверждение, которое избавляет от необходимости проверки выводимости любой зависимости.

Утверждение 3. Два множества зависимостей Σ и Σ^* являются эквивалентными, если любая зависимость $W[X] \subsetneq V$, принадлежащая Σ, выводима в Σ^*, и любая зависимость $W^*[X] \subsetneq V^*$, принадлежащая Σ^*, выводима в Σ^*.

Таким образом, для проверки эквивалентности двух множеств зависимостей не требуется проверять выводимость всех возможных зависимостей. Достаточно проверить выводимость собственных зависимостей каждого из множеств в другом множестве.

Пусть задано исходное множество обобщенных типизированных зависимостей включения Σ. На его основе построим множество Σ^*. Для каждой зависимости $W[X] \subsetneq V \in \Sigma$ во множество Σ^* дополним зависимости $R_i[X] \subsetneq V$, $i = \overline{1,m}$, где $W = \cup_{i=1}^{m} R_i$. Из способа построения Σ^* можно сделать следующий вывод.

Утверждение 4. Два множества зависимостей Σ и Σ^* эквивалентны.

Действительно, выводимость зависимостей Σ^* из множества Σ следует по правилу 1 (декомпозиция), а по аксиоме 3 следует выводимость зависимостей Σ из множества Σ^*.

Для построения минимального множества зависимостей будем использовать множество Σ^*, поскольку несколько отношений в левой части зависимостей могут помешать друг другу при проверке

выводимости. Например, зависимость $R_i[X] \subsetneq V$ выводима, зависимость $R_j[X] \subsetneq V$ не выводима, тогда зависимость $\{R_i, R_j\}[X] \subsetneq V$ не выводима, что препятствует удалению $R_i[X] \subsetneq V$.

Рассмотрим процедуру построения минимального покрытия Σ^*:

procedure MIN_CLOS(Σ^*)
modification ← TRUE
while *modification*
  modification ← FALSE
  for each $R_i[X] \subsetneq V_i$ **from** Σ^*
    if $R_i \in$ CLOS_M($\Sigma^* - \{R_i[X] \subsetneq V_i\}, V_i, X$) **then**
      $\Sigma^* \leftarrow \Sigma^* - \{R_i[X] \subsetneq V_i\}$
      modification ← TRUE
    end if
  end for
  for each $R_i[X] \subsetneq V_i$ bf from Σ^*
    if $|V_i| > 1$ **then**
      for each R_j **from** V_i
        $V_j \leftarrow V_i - R_j$
        $\Sigma^- \leftarrow \Sigma^* - \{R_i[X] \subsetneq V_i\} \cup \{R_i[X] \subsetneq V_j\}$
        if $R_i \in$ CLOS_M(Σ^*, V_j, X) **then**
          $\Sigma^* \leftarrow \Sigma^-$
          modification ← TRUE
        end if
      end for
    end if
  end for
end while
end procedure

С учетом количества итераций в алгоритме *CLOS_M* максимальное количество итераций в алгоритме *MIN_CLOS* будет равно $n^2k + n^2k^2$, где n — мощность множества зависимостей Σ^*, k — количество отношений в БД.

Замечание. Можно предположить, что удаление атрибутов, на которых определены отношения в зависимостях, будет способствовать минимизации множества зависимостей. Однако, всякий раз мы будем иметь большее количество зависимостей, которые все выводимы из исходного множества зависимостей и, следовательно, будут удалены в очередном цикле *while* алгоритма *CLOS_M*.

Необходимость повторного поиска избыточных зависимостей в цикле *while* при анализе Σ^* обусловлена тем, что после удаления избыточных отношений в правых частях зависимостей, возможно, что некоторые из зависимостей окажутся избыточными. Для демонстрации этого свойства рассмотрим пример (аналогичный пример для функциональных зависимостей представил в сети Интернет А. Ю. Филиппович).

Пример 1. Пусть исходное множество зависимостей Σ^* задано для трех отношений: R_1, R_2 и R_3, каждое из которых определено на атрибутах X:

$$\Sigma^* = \{R_3[X] \subsetneq \{R_1, R_2\}, R_2[X] \subsetneq R_1, R_2[X] \subsetneq R_3\}. \tag{1}$$

Проверим на избыточность зависимость $R_3[X] \subsetneq \{R_1, R_2\}$: строим замыкание $\{R_1, R_2\}^+ = R_1, R_2$ на оставшемся множестве зависимостей. В результате имеем, что отношение R_3 не принадлежит замыканию, следовательно, зависимость $R_3[X] \subsetneq \{R_1, R_2\}$ не лишняя.

Проверка на избыточность зависимостей $R_2[X] \subsetneq R_1$ и $R_2[X] \subsetneq R_3$ показала, что эти зависимости не являются избыточными: отношение в левой части не принадлежит замыканию правой части на оставшемся множестве зависимостей. На следующем этапе сначала удалим отношение R_1 из правой части зависимости $R_3[X] \subsetneq \{R_1, R_2\}$. Полученное множество не эквивалентно исходному, поэтому возвращаем R_1 на место. После удаления R_2 получается следующее множество зависимостей:

$$\Sigma^- = \{R_3[X] \subsetneq R_1, R_2[X] \subsetneq R_1, R_2[X] \subsetneq R_3\}.$$

Не сложно показать, что множества Σ^* и Σ^- эквивалентны. Действительно, вторая и третья зависимости в них совпадают. Замыкание правой части зависимости $R_3[X] \subsetneq \{R_1, R_2\}$ на множестве Σ^- содержит R_3 по правилу 2 (пополнение), и замыкание правой части зависимости $R_3[X] \subsetneq R_1$ на множестве Σ^* содержит R_3 по алгоритму построения замыкания. Следовательно, множества Σ^* и Σ^- эквивалентны.

Анализируя множество Σ^- не трудно заметить, что зависимость $R_2[X] \subsetneq R_1$ является избыточной (транзитивной). Эта зависимость будут удалена при *повторном* выполнении цикла *while* в алгоритме *MIN_CLOS*. После удаления избыточной зависимости получим следующее множество:

$$\Sigma^- = \{R_3[X] \subsetneq R_1, R_2[X] \subsetneq R_3\}. \tag{2}$$

На последующих итерациях алгоритма удалить ничего не получится, следовательно, минимальное покрытие Σ^* равно $\{R_3[X] \subsetneq R_1, R_2[X] \subsetneq R_3\}$. Рассмотренный пример показал необходимость повторной проверки на избыточность множества зависимостей после удаления избыточных отношений в правых частях зависимостей.

Дополнительно проверим условия ссылочной целостности полученного результата в соответствии с определением 4. Для исходного множества (1) выполнены следующие условия:

- если кортеж t принадлежит отношению R_1, то ограничения отсутствуют;
- если $t[X] \in R_2[X]$, то $t[X] \in R_1[X]$ и $t[X] \in R_3[X]$;
- если $t[X] \in R_3[X]$, то $t[X] \in R_1[X]$ или $t[X] \in R_2[X]$, в случае если $t[X] \in R_2[X]$, то $t[X] \in R_1[X]$; перепишем последнее условие:
- если $t[X] \in R_3[X]$, то $t[X] \in R_1[X]$, условие $t[X] \in R_3[X]$ может быть не выполнено.

Для результирующего множества (2) выполнены следующие условия:

- если кортеж $t \in R_1$, то ограничения отсутствуют;
- если $t[X] \in R_2[X]$, то $t[X] \in R_1[X]$ и $t[X] \in R_3[X]$;
- если $t[X] \in R_3[X]$, то $t[X] \in R_1[X]$.

Проверка показала, что ссылочные ограничения целостности остались прежними после выполненных преобразований.

После рассмотрения алгоритма *MIN_CLOS* закономерным является вопрос о корректности его работы. Это означает, что выводимая зависимость будет удалена в алгоритме и удаленная зависимость является выводимой.

Теорема 4 (Минимальное покрытие)**.** *Процедура* MIN_CLOS *корректно формирует минимальное покрытие множества обобщенных типизированных зависимостей включения.*

Доказательство. Достаточно просто обосновывается удаление выводимых зависимостей. В соответствии с теоремой 3 зависимость является выводимой на проверочном множестве зависимостей, если ее левая часть принадлежит замыканию правой части зависимости. В алгоритме *MIN_CLOS* циклически просматриваются все возможные варианты выводимости, и выход из бесконечного цикла *while* осуществляется только при отсутствии выводимых зависимостей и избыточных отношений в правых частях зависимостей. Следовательно, результирующее множество Σ^* не содержит выводимых зависимостей.

Также не сложно обосновать, что удаленные зависимости являются выводимыми. Зависимости в первом цикле *for* удаляются, если левая часть зависимости принадлежит замыканию правой части без учета самой зависимости. По теореме 3 такая зависимость является выводимой. Во втором цикле *for* удаляются отношения в правой части зависимости σ^* и проверяется выводимость вновь полученной зависимости σ^- на исходном множестве Σ^*. Если выводимость получена, то в соответствии с определением 6 новое множество зависимостей Σ^- эквивалентно Σ^* (выводимость σ^* на Σ^- гарантируется правилом 2 (пополнение)). Следовательно, исходное множество Σ^* заменяем на множество Σ^-, в котором количество отношений в правой части σ^- меньше. □

Замечание. Различный порядок удаления зависимостей и отношений в алгоритме *MIN_CLOS* может привести к различным минимальным покрытиям, однако, все они эквивалентны друг другу. Если дополнить критерий минимальности для выбора одного из всех покрытий (как это сделано в работе [2] для функциональных зависимостей), то соответствующий алгоритм будет NP-трудным и не будет гарантировать единственность решения.

Рассмотрим пример, в котором есть два минимальных покрытия исходного множества зависимостей.

Пример 2. Исходное множество зависимостей задано для трех отношений, определенных на атрибутах X:

$$\Sigma^* = \{R_3[X] \subsetneq \{R_1, R_2\}, R_2[X] \subsetneq R_1, R_1[X] \subsetneq R_2\}. \tag{3}$$

Различный порядок удаления отношений в первом правиле порождает два минимальных покрытия:

$$\Sigma_1^* = \{R_3[X] \subsetneq R_1, R_2[X] \subsetneq R_1, R_1[X] \subsetneq R_2\} \tag{4}$$

и

$$\Sigma_2^* = \{R_3[X] \subsetneq R_2, R_2[X] \subsetneq R_1, R_1[X] \subsetneq R_2\}. \tag{5}$$

Ссылочные ограничения целостности для множеств зависимостей (3)–(5) совпадают:

- если кортеж $t[X] \in R_1$, то $t[X] \in R_2[X]$;
- если $t[X] \in R_2[X]$, то $t[X] \in R_1[X]$;
- если $t[X] \in R_3[X]$, то $t[X] \in R_1[X]$ и $t[X] \in R_2[X]$.

Проверка показала, что ссылочные ограничения целостности являются эквивалентными для различных минимальных покрытий. Поскольку условие эквивалентности, кроме всего прочего, обладает свойством транзитивности, то все минимальные покрытия будут эквивалентны друг другу (все вместе они эквивалентны исходному множеству зависимостей). Следовательно, всем им соответствует одно и то же множество выполнимых (и выводимых) зависимостей и одни те же ограничения на допустимые состояния БД.

Заключение

В работе рассмотрена проблема выводимости и построения минимального покрытия для обобщенных зависимостей включения, что позволяет избавиться от избыточных затрат по памяти и по времени при их практическом применении. Как отмечалось ранее, в настоящее время пока нет СУБД, реализующих ссылочные ограничения целостности, соответствующие обобщенным типизированным зависимостям включения. Однако, большинство СУБД имеют в своем составе аппарат триггеров, с использованием которого возможна реализация таких ограничений для конкретной БД.

References

[1] J. Ullman, *Principles of Database Systems*. Stanford University: Computer Science Press, 1980, 484 pp.

[2] D. Maier, *The Theory of Relational Databases*. Rockville: Computer Science Press, 1983, 637 pp.

[3] M. Casanova, R. Fagin, and C. Papadimitriou, "Inclusion dependencies and their interaction with functional dependencies", *Journal of Computer and System Sciences*, vol. 28, no. 1, pp. 29–59, 1984.

[4] A. K. Chandra and M. Y. Vardi, "The implication problem for functional and inclusion dependencies is undecidable", *SIAM Journal on Computing*, vol. 14, no. 3, pp. 671–677, 1985. DOI: 10.1137/0214049.

[5] R. Fagin and M. Y. Vardi, "Armstrong databases for functional and inclusion dependencies", *Information Processing Letters*, vol. 16, no. 1, pp. 13–19, 1983. DOI: 10.1016/0020-0190(83)90005-4.

[6] P. M. Kanellakis, R. Cosmadakis, and M. Y. Vardi, "Unary inclusion dependencies have polynomial time inference problems", in *Proceedings of the fifteenth annual ACM symposium on Theory of computing (STOC '83)*, 1983, pp. 264–277. DOI: 10.1145/800061.808756.

[7] S. S. Cosmadakis, P. C. Kanellakis, and M. Y. Vardi, "Polynomial-time implication problems for unary inclusion dependencies", *ACM*, vol. 37, no. 1, pp. 15–46, 1990. DOI: 10.1145/78935.78937.

[8] M. Levene and V. M. W., "Justification for inclusion dependency normal form", *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 2, pp. 281–291, 2000. DOI: 10.1109/69.842267.

[9] M. Hannula and S. Link, "On the interaction of functional and inclusion dependencies with independence atoms", in *Proceedings of the International Conference on Database Systems for Advanced Applications*, 2018, pp. 353–369. DOI: 10.1007/978-3-319-91458-9_21.

[10] J. Biskup and P. Dublish, "Objects in relational database schemes with functional, inclusion and exclusion dependencies", in *3rd Symposium on Mathematical Fundamentals of Database and Knowledge Base Systems*, 1991, pp. 276–290. DOI: 10.1007/3-540-54009-1_20.

[11] F. De Marchi, S. Lopes, and J.-M. Petit, "Efficient algorithms for mining inclusion dependencies", in *Advances in Database Technology — EDBT 2002*, C. S. Jensen *et al.*, Eds., Springer Berlin Heidelberg, 2002, pp. 464–476. DOI: 10.1007/3-540-45876-X_30.

[12] J. Bauckmann, Z. Abedjan, H. Müller, and F. Naumann, "Discovering conditional inclusion dependencies", in *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, 2012, pp. 2094–2098. DOI: 10.1145/2396761.2398580.

[13] M. T. Gomez-Lopez, R. M. Gasca, and J. M. Perez-Alvarez, "Compliance validation and diagnosis of business data constraints in business processes", *Information Systems*, vol. 48, pp. 26–43, 2015.

[14] S. Ma, W. Fan, and L. Bravo, "Extending inclusion dependencies with conditions", *Theoretical Computer Science*, vol. 515, pp. 64–95, 2014. DOI: 10.1016/j.tcs.2013.11.002.

[15] F. Tschirschnitz, T. Papenbrock, and F. Naumann, "Detecting inclusion dependencies on very many tables", *ACM Transactions on Database Systems*, vol. 42, no. 3, pp. 1–29, 2017. DOI: 10.1145/3105959.

[16] Y. Kaminsky, E. Pena, and F. Naumann, "Discovering similarity inclusion dependencies", *Proceedings of the ACM on Management of Data*, vol. 1, no. 1, pp. 1–24, 2023. DOI: 10.1145/3588929.

[17] M. Levene and G. Loizou, "Null inclusion dependencies in relational databases", *Information and Computation*, vol. 136, no. 2, pp. 67–108, 1997. DOI: 10.1006/inco.1997.2631.

[18] M. Levene and G. Loizou, "The additivity problem for data dependencies in incomplete relational databases", in *Semantics in Databases*, vol. 1358, Springer, 1998, pp. 136–169. DOI: 10.1007/BFb0035008.

[19] H. Köhler and S. Link, "Inclusion dependencies reloaded", in *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, 2015, pp. 1361–1370.

[20] V. S. Zykin and S. V. Zykin, "Analysis of typed inclusion dependences with null values", *Automatic Control and Computer Sciences*, vol. 52, no. 7, pp. 638–646, 2018. DOI: 10.3103/S0146411618070258.

[21] S. V. Zykin, "Generalization of typed include dependencies with null values in databases", *Modeling and Analysis of Information Systems*, vol. 30, no. 3, pp. 192–201, 2023, in Russian.

MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 31, NO. 1, 2024
journal homepage: www.mais-journal.ru

ARTIFICIAL INTELLIGENCE

Application of Deep Neural Networks for Automatic Irony Detection in Russian Texts

M. A. Kosterin[1], I. V. Paramonov[1]

DOI: 10.18255/1818-1015-2024-1-90-101

[1]P.G. Demidov Yaroslavl State University, Yaroslavl, Russia

MSC2020: 68T50
Research article
Full text in Russian

Received February 15, 2024
Revised February 23, 2024
Accepted February 28, 2024

The paper examines automatic methods for classifying Russian-language sentences into two classes: ironic and non-ironic. The discussed methods can be divided into three categories: classifiers based on language model embeddings, classifiers using sentiment information, and classifiers with embeddings trained to detect irony. The components of classifiers are neural networks such as BERT, RoBERTa, BiLSTM, CNN, as well as an attention mechanism and fully connected layers. The irony detection experiments were carried out using two corpora of Russian sentences: the first corpus is composed of journalistic texts from the OpenCorpora open corpus, the second corpus is an extension of the first one and is supplemented with ironic sentences from the Wiktionary resource.

The best results were demonstrated by a group of classifiers based on embeddings of language models with the maximum F-measure of 0.84, achieved by a combination of RoBERTa, BiLSTM, an attention mechanism and a pair of fully connected layers in experiments on the extended corpus. In general, using the extended corpus produced results that were 2–5 % higher than those of the basic corpus. The achieved results are the best for the problem under consideration in the case of the Russian language and are comparable to the best one for English.

Keywords: irony detection; sarcasm detection; neural network-based classifier; deep learning; natural language processing; BERT

INFORMATION ABOUT THE AUTHORS

Kosterin, Maksim A.	ORCID iD: 0000-0002-8298-3156. E-mail: makcost@gmail.com Post-graduate student
Paramonov, Ilya V. (corresponding author)	ORCID iD: 0000-0003-3984-8423. E-mail: ilya.paramonov@fruct.org PhD, associate professor

Funding: Russian Science Foundation (Project no. 23-21-00495).

For citation: M. A. Kosterin and I. V. Paramonov, “Application of deep neural networks for automatic irony detection in Russian texts”, *Modeling and Analysis of Information Systems*, vol. 31, no. 1, pp. 90–101, 2024. DOI: 10.18255/1818-1015-2024-1-90-101.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 31, № 1, 2024
сайт журнала: www.mais-journal.ru

Применение глубоких нейронных сетей для автоматического определения иронии в русскоязычных текстах

М. А. Костерин[1], И. В. Парамонов[1]

DOI: 10.18255/1818-1015-2024-1-90-101

[1]Ярославский государственный университет им. П.Г. Демидова, Ярославль, Россия

УДК 004.912
Научная статья
Полный текст на русском языке

Получена 15 февраля 2024 г.
После доработки 23 февраля 2024 г.
Принята к публикации 28 февраля 2024 г.

В работе исследуются автоматические методы классификации русскоязычных предложений на два класса: содержащие и не содержащие ироничный посыл. Рассматриваемые методы могут быть разделены на три категории: классификаторы на основе эмбеддингов языковых моделей, классификаторы с использованием информации о тональности и классификаторы с обучением эмбеддингов обнаружению иронии. Составными элементами классификаторов являются нейронные сети, такие как BERT, RoBERTa, BiLSTM, CNN, а также механизм внимания и полносвязные слои. Эксперименты по обнаружению иронии проводились с использованием двух корпусов русскоязычных предложений: первый корпус составлен из публицистических текстов из открытого корпуса OpenCorpora, второй корпус является расширением первого и дополнен ироничными предложениями с ресурса Wiktionary.

Лучшие результаты продемонстрировала группа классификаторов на основе чистых эмбеддингов языковых моделей с максимальным значением F-меры 0.84, достигнутым связкой из RoBERTa, BiLSTM, механизма внимания и пары полносвязных слоев в ходе экспериментов на расширенном корпусе. В целом использование расширенного корпуса давало результаты на 2–5 % выше результатов на базовом корпусе. Достигнутые результаты являются лучшими для рассматриваемой задачи в случае русского языка и сравнимы с лучшими для английского.

Ключевые слова: обнаружение иронии; обнаружение сарказма; нейросетевой классификатор; глубокое обучение; обработка естественного языка; BERT

ИНФОРМАЦИЯ ОБ АВТОРАХ

Костерин, Максим Алексеевич	ORCID iD: 0000-0002-8298-3156. E-mail: makcost@gmail.com Аспирант
Парамонов, Илья Вячеславович (автор для корреспонденции)	ORCID iD: 0000-0003-3984-8423. E-mail: ilya.paramonov@fruct.org Канд. физ.-мат. наук, доцент

Финансирование: Российский научный фонд (проект № 23-21-00495).

Для цитирования: M. A. Kosterin and I. V. Paramonov, “Application of deep neural networks for automatic irony detection in Russian texts”, *Modeling and Analysis of Information Systems*, vol. 31, no. 1, pp. 90–101, 2024. DOI: 10.18255/1818-1015-2024-1-90-101.

Введение

Автоматическое обнаружение иронии и сарказма — одна из задач обработки естественного языка, направленная на выявление в отдельных предложениях или текстах содержания, имеющего иронический или саркастический посыл. Для русского языка эта задача на настоящий момент остается малоисследованной. Вызвано это в первую очередь недостатком корпусов текстов на русском языке, размеченных по наличию и отсутствию иронии и сарказма. Тем не менее эта задача является актуальной в такой области, как анализ общественного мнения, также её решение может быть полезно для других задач автоматической обработки текста, например, классификации по тональности.

Данная статья развивает рассмотренные в работе [1] автоматизированные методы классификации русскоязычных предложений на два класса: содержащие и не содержащие ироничный посыл. Сарказм в обеих работах предполагается частным случаем иронии. В исходной работе исследовались только базовые нейросетевые модели, такие как BERT и BiLSTM, а также алгоритмы классического машинного обучения: логистическая регрессия, методы опорных векторов и случайного леса. Целью текущей работы является изучение возможностей более сложных моделей и методов, использующих связки классификаторов и различные подходы к классификации в рамках одной сложной архитектуры, для повышения качества результатов.

В качестве основных подходов в текущей работе используются методы с использованием эмбеддингов чистых языковых моделей (Language Modeling, LM), а также эмбеддингов языковых моделей, адаптированных под предметную область задач классификации по тональности (Sentiment Analysis, SA) и обнаружения иронии (Irony Detection, ID). Здесь LM-эмбеддингами называются эмбеддинги предпоследнего внутреннего слоя BERT, который обучался задаче маскированного моделирования языка (MLM), SA-эмбеддингами — эмбеддинги предпоследнего внутреннего слоя BERT, обученного классифицировать тональность, а под ID-эмбеддингами понимаются эмбеддинги предпоследнего внутреннего слоя BERT, который дополнительно к предобучению MLM обучался обнаружению иронии. В качестве структурных элементов классификаторов используются нейронные сети BERT, RoBERTa, BiLSTM, CNN и механизм внимания (Attention).

Следует отметить, что в отличие от подавляющего большинства исследований, встречающихся в литературе, в данной работе для экспериментов используются не короткие тексты из социальных сетей Twitter, Reddit и им подобным, а тексты из новостных и аналитических статей и записи блогов. Это обеспечивает дополнительную новизну исследованию, а также приводит к некоторому усложнению задачи, поскольку в подобных текстах используется значительно более широкий спектр средств выражения иронии.

Статья построена следующим образом. В разделе 1 приведён обзор работ по тематике автоматического определения иронии и сарказма в текстах. Детальное описание используемых в проведённом исследовании методов и моделей приводится в разделе 2. Раздел 3 описывает корпуса, используемые для проведения экспериментов. В разделе 4 производится оценка предложенных методов и моделей на двух корпусах русскоязычных предложений. В заключении подведены итоги работы.

1. Обзор связанных работ

Наиболее распространенным подходом к обнаружению иронии является применение векторов характеристик для кодирования текстовых последовательностей с последующей бинарной классификацией этих векторов. Векторы характеристик обычно строятся с использованием эмбеддингов, генерируемых такими инструментами, как Word2Vec, GloVe, BERT и т. д.

Использование базовых подходов для обнаружения сарказма в англоязычном корпусе, собранном из текстов социальной сети Reddit, демонстрируется в работе [2]. В ней приводится сравнение

классификаторов, построенных с использованием методов машинного обучения и нейронных сетей BERT и BiLSTM, при анализе комментариев пользователей. Лучшим среди методов машинного обучения оказался метод гребневой регрессии, позволяющий получить F-меру 0.68. В целом же наилучший результат демонстрирует нейронная сеть BERT, достигшая F-меры 0.72. Схожее сравнение базовых подходов для русского языка из предшествующей работы [1] показало близкие результаты: BERT также занял первое место с F-мерой 0.76, а среди методом машинного обучения лучше всего себя показала логистическая регрессия с F-мерой 0.68.

Более совершенный подход с применение эмбеддингов RoBERTa в связке с BiLSTM, аналог которого применяется и в текущей работе, для англоязычных текстов рассматривается в работе [3]. В ней предобученные эмбеддинги RoBERTa являются входными данными для слоя BiLSTM, а также конкатенируются с его выходными данными в один вектор, используемый в классификаторе. Предложенный подход показывает F-меру 0.80, 0.78 или 0.90 для корпусов текстов Irony/SemVal-2018-Task 3.A [4], Reddit SARC2.0 politics [5] и Riloff sarcastic dataset [6] соотвественно.

В работе [7] авторами исследуется применение подхода трансферного обучения для решения задачи обнаружения иронии. В ней используются две нейронный сети BiLSTM: одна из них сперва обучается задаче классификации тональности на корпусе англоязычных твитов; далее веса первой BiLSTM замораживаются и эти же твиты подаются на вход уже обеим BiLSTM; наконец, выходные данные обеих BiLSTM объединяются в один вектор и передаются в последовательно расположенные слой внимания и полносвязный слой, выполняющий функцию классификатора для задачи обнаружения иронии. В данной структуре первая BiLSTM, обученная классифицировать тональность, предназначена для выделения тональных характеристик, которые дополняют семантические характеристики, извлекаемые второй BiLSTM. С предложенным подходом в работе достигается F-мера 0.68 на корпусе Irony/SemVal-2018-Task 3.A [4] и 0.78 на корпусе Riloff sarcastic dataset [6]

В работе [8] предлагается метод обнаружения сарказма в текстовых онлайн-дискуссиях с использованием контекста — характеристик пользователя, оставившего комментарий, включающие его стилометрические характеристики, личностные характеристики и характеристики сообщений форума, в рамках которого ведется дискуссия. Моделирование контекста производится путем использования CNN для извлечения высокоуровневых характеристик из истории сообщений пользователя и форума. Позже эти характеристики добавляются к содержательным характеристикам классифицируемого сообщения, также извлеченным с помощью CNN, для обучения модели решению задачи обнаружению сарказма. В качестве корпуса данных в этой работе применяется Reddit SARC2.0 [5]. Предложенная модель без использования личностных характеристик показывает F-меру 0.66, а при их использовании этот показатель повышается до 0.77.

Проблема создания корпуса русскоязычных текстов для обнаружения сарказма освещается в работе [9]. Согласно полученным в ней данным, хэштеги, являющиеся маркерами сарказма или иронии, не используются в достаточной степени русскоязычной аудиторией Twitter. Отмечается, что хэштег #ирония в основном употребляется в качестве описания изображений или шуток вместе с хэштегами #шутка и #смех. В тоже время хэштег #сарказм действительно применяется для обозначения сарказма, однако его релевантное употребление недостаточно распространено, чтобы использовать его для составления корпуса.

В работе [10] задача обнаружения сарказма в текстах на русском языке решается посредством решения задачи классификации по тональности. Авторами используется корпус предложений Twitter, который сперва разделяется на три класса тональности (положительный, отрицательный, нейтральный) нейронной сетью RuBERT, обученной решению задачи классификации тональности. Далее на размеченном корпусе обучается модель для обнаружения сарказма следующим образом: с использованием словаря тональности слов подсчитывается количество положительных и отрицательных слов в предложении; если предложение содержит слова с отрицательной тональностью

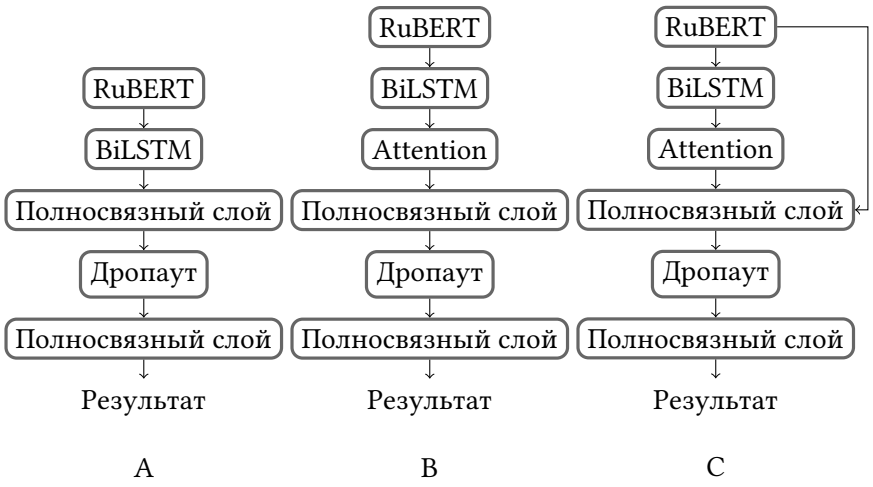


Fig. 1. RuBERT (LM) — BiLSTM model

Рис. 1. Модель RuBERT (LM) — BiLSTM

и при этом количество слов с положительной тональностью превышает количество слов с отрицательной, но класс тональности данного предложения является отрицательным, то в таком предложении отмечается наличие сарказма. Наличие сарказма также отмечается и в противоположном случае, когда количество отрицательно окрашенных слов превышает количество положительных, а класс тональности предложения является положительным. Таким образом задача обнаружения сарказма в данной работе решается без выполнения разметки корпуса по наличию или отсутствию сарказма, а использует только информацию о тональности. F-мера предложенного подхода составляет 0.69.

В приведенных работах описывается только часть подходов, используемых для решения задачи обнаружения иронии и сарказма, а более подробный обзор области приведен в работе [11]. Рассмотренные же в данном разделе работы выделяются в том плане, что используемые ими подходы, модели или данные представляют интерес в рамках исследования данной работы. В целом можно отметить, что задача обнаружения иронии является менее исследованной, чем многие другие задачи обработки естественных языков, например, классификации тональности, вопросно-ответные системы или машинный перевод. Особенно это актуально для русского языка, где количество исследований по данной теме минимально, а проведение экспериментов сопровождается определенными трудностями в плане сбора экспериментальных данных.

2. Используемые методы и модели

2.1. Классификаторы на основе эмбеддингов

В данную группу входят классификаторы, которые оперируют эмбеддингами токенов, генерируемыми моделью RuBERT [12]. Для каждого подаваемого на вход текстового токена RuBERT производит соответствующий ему числовой вектор. В дальнейшем обучение классификатора происходит с использованием последовательностей этих векторов.

Наиболее простым классификатором из данной группы является связка из RuBERT с BiLSTM, дополненная парой полносвязных слоев (рис. 1A). В данной связке RuBERT отвечает за создание контекстуализированных эмбеддингов, BiLSTM — за моделирование контекста входной последовательности, первый полносвязный слой с функцией активации ReLU — за классификацию, а второй полносвязный слой с функцией активации softmax выполняет роль выходного слоя, преобразуя вектор результатов классификатора в значения предсказания. Два полносвязных слоя соединены слоем дропаута для предотвращения переобучения.

Следующая модель в группе дополнена слоем внимания, расположенным между BiLSTM и первым полносвязным слоем (рис. 1B). Задачей слоя внимания является назначение весов токенам входной последовательности в зависимости от их важности при определении того, является ли предложение ироничным или нет. Конструкции, которые являются наиболее явным признаком наличия иронии, будут иметь большие веса и наоборот.

Следующая модель также расширяет предыдущую, в этот раз добавляя значения эмбеддингов токенов к выходным значениям слоя внимания (рис. 1C). Идея данного подхода заключается в том, чтобы вместе с моделью входной последовательности в слой классификатора передать и изначальные сведения об контексте употребления токенов, тем самым увеличив количество доступной информации о токене. Эмбеддинги токенов и вектор внимания каждого предложения конкатенируются поэлементно между собой и передаются на вход первому полносвязному слою. Для этой модели также была проверена ее вариация, в которой вместо RuBERT для формирования эмбеддингов использовалась модель RoBERTa [13].

Следующие три модели из этой группы помимо эмбеддингов учитывают также и характеристики предложения, потенциально указывающие на наличие иронии в нем (рис. 2A). В первой модели в качестве характеристик выступают только определенные знаки препинания («?», «!» и «. . . »), кавычки и круглые скобки как части смайлов. Во второй модели к признакам добавляется информация о присутствующих в предложении частях речи и биграммах, а в третьей — информация о наличии в предложении последовательностей токенов, входящих в список ироничных выражений, который был собран с использованием русскоязычной версии Интернет-ресурса Wiktionary[1]. В качестве ироничных выражений собирались фразы, принадлежащие к категории «Ироничные выражения/ru». В результате получился список из 1144 крылатых выражений, жаргонизмов, фразеологизмов и прочих фраз и слов, употребляемых с целью высмеять или оскорбить. Информация о характеристиках предложений преобразовывается в числовой вектор, который конкатенируется с вектором внимания, после чего результат подается на вход первому полносвязному слою. Таким образом классифицируемый вектор содержит как контекстуализированную модель предложения, так и информацию о некоторых характеристиках текста, потенциально сигнализирующих о наличии иронии.

Последний классификатор из данной группы (рис. 2B) использует два последовательно идущих слоя CNN с размером ядра 3 для выделения последовательностей высокоуровневых представлений групп эмбеддингов входной последовательности. За слоями CNN также следуют слой BiLSTM, слой внимания и два полносвязных слоя. Такая структура нейронной сети позволяет учитывать локальные характеристики фраз за счет CNN, а также семантику всего предложения за счет BiLSTM [14].

2.2. Классификаторы с использованием информации о тональности

Второй подход к обнаружению ироничных предложений использует информацию о тональности предложения в качестве вспомогательных данных. Ирония как литературный прием выражается в сокрытии или противопоставлении истинного смысла предложения его буквальному смыслу. Наиболее часто это проявляется тем, что вкладываемый в предложение эмоциональный окрас противоположен буквальному окрасу, т. е. текст, нацеленный на то, чтобы принизить или оскорбить, может выражаться фразами, имеющими вне контекста положительную тональность. Обнаружение такого несоответствия между буквальным и истинной тональностью предложения будет выступать в качестве показателя наличия ироничного посыла приложения.

С этой целью в корпус предложений, размеченных на предмет наличия иронии, дополнительно была добавлена разметка всех предложений по классам положительной, отрицательной или ней-

[1] https://ru.wiktionary.org

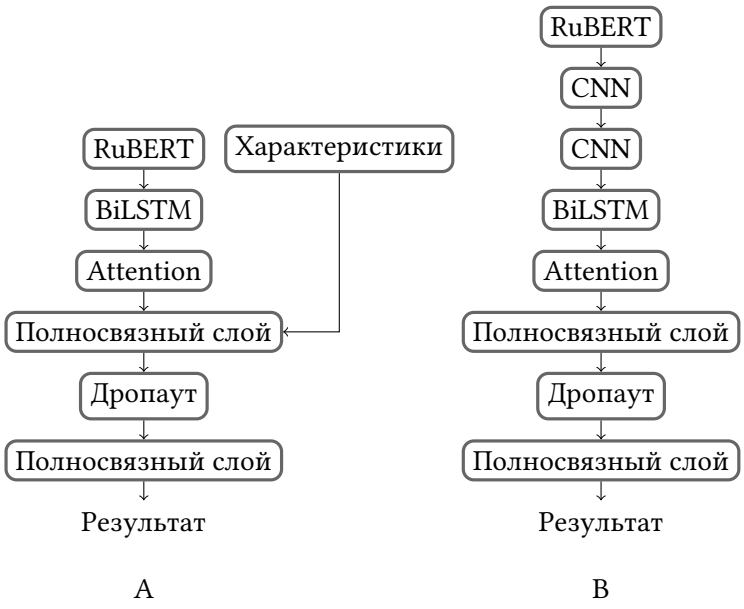


Fig. 2. A. RuBERT (LM) — BiLSTM — Att. + Fts. model. B. RuBERT (LM) — CNN — CNN — BiLSTM — Att. model

Рис. 2. A. Модель RuBERT (LM) — BiLSTM — Att. + Fts. B. Модель RuBERT (LM) — CNN — CNN — BiLSTM — Att.

тральной тональности. Разметка тональности подается на вход классификатору параллельно с разметкой по наличию или отсутствию иронии и используется на этапе обучения моделей.

Первая модель из данной группы представлена на рис. 3. Она имеет те же ключевые компоненты, что и ранее рассмотренные модели, но выполняет обработку двух типов входных данных параллельно. Как и ранее, с помощью RuBERT выполняется преобразование предложений в эмбеддинги, далее каждому эмбеддингу предложения сопоставляется соответствующая метка класса тональности и наличия иронии. Эти данные подаются на вход слоям BiLSTM: в один из них передаются эмбеддинги и метки наличия иронии, а во второй — те же эмбеддинги и метки тональности. Следом за каждой BiLSTM идет слой внимания; результаты, полученные на входе этого слоя конкатенируются и передаются в общую последовательность из двух полносвязных слоев. Эта цепочка слоев используется для обучения модели обнаружению иронии. Параллельно с этим из BiLSTM, в которую были переданы метки тональности, идет еще одна отдельная цепочка из полносвязных слоев. Ее результат используется только для обучения этой части модели классификации по тональности. На этапе оценки и разметки результаты классификации тональности игнорируются, так они не важны для данного исследования. В результате получается, что одна из BiLSTM обучается исключительно обнаружению иронии, в то время как вторая параллельно обучается классификации тональности, при этом передавая свои выходы в общую цепочку. Таким образом общая цепочка слоев работает как с информацией о наличии иронии, так и информацией о тональности предложений.

Вторая модель данной группы использует информацию о тональности предложений для дообучения модели RuBERT с целью получения эмбеддингов, изначально содержащих информацию об эмоциональном контексте употребляемых токенов (рис. 4A). Дообучение происходит с использованием обучающей выборки корпуса предложений текущего этапа кросс-валидации. После дообучения RuBERT также используется для создания контекстуализированных эмбеддингов, но их веса будут отличаться от весов эмбеддингов стандартного RuBERT в связи с проведенным дообучением. Полученные эмбеддинги вместе с информацией о наличии иронии передаются в модель из BiLSTM, слоя внимания и двух полносвязных слоев для обучения модели обнаружению иронии.

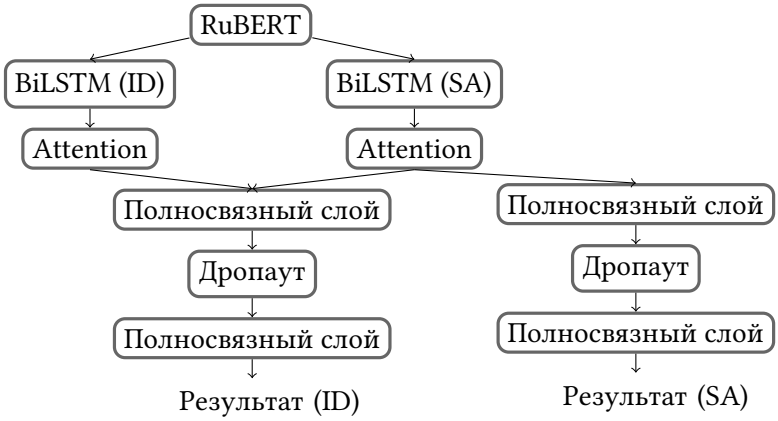


Fig. 3. RuBERT (LM) — BiLSTM (ID) — Att. + BiLSTM (SA) — Att. model

Рис. 3. Модель RuBERT (LM) — BiLSTM (ID) — Att. + BiLSTM (SA) — Att.

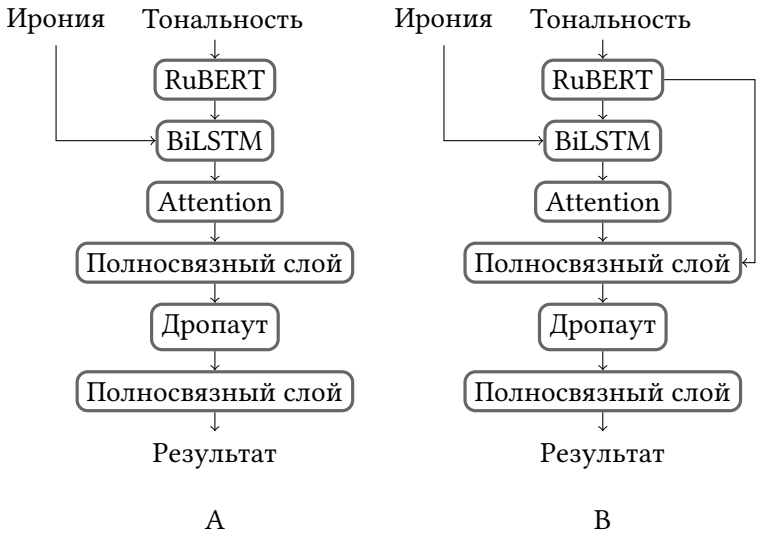


Fig. 4. RuBERT (SA) — BiLSTM — Att. model

Рис. 4. Модель RuBERT (SA) — BiLSTM — Att.

Таким образом, данная модель учитывает информацию о тональности предложения сразу, начиная с этапа создания эмбеддингов.

Следующая модель группы расширяет предыдущую, добавляя значения эмбеддингов токенов к значениям на выходе слоя внимания (рис. 4B). Идея данного подхода аналогична той, что описывалась для модели, приведенной на рис. 1C.

Еще одна модель данной группы использует ранее рассмотренную связку из эмбеддингов RuBERT, BiLSTM, слоя внимания и двух полносвязных слоев с тем различием, что вместо стандартной модели RuBERT для языкового моделирования применяется обученная на корпусе RuSentiment [15] модель RuBERT для классификации тональности. Использование такой модели также позволяет получать контекстуализированные эмбеддинги, учитывающие тональность.

2.3. Классификаторы с обучением эмбеддингов обнаружению иронии

К последней группе принадлежат классификаторы, в который RuBERT обучается задаче обнаружения иронии на корпусе ироничных предложений. Так как RuBERT изначально обучался задаче языкового моделирования на гораздо большем корпусе текстов из русскоязычной версии Интернет-ресурса Wikipedia и российских новостных изданий, то его дополнительное обучение задаче

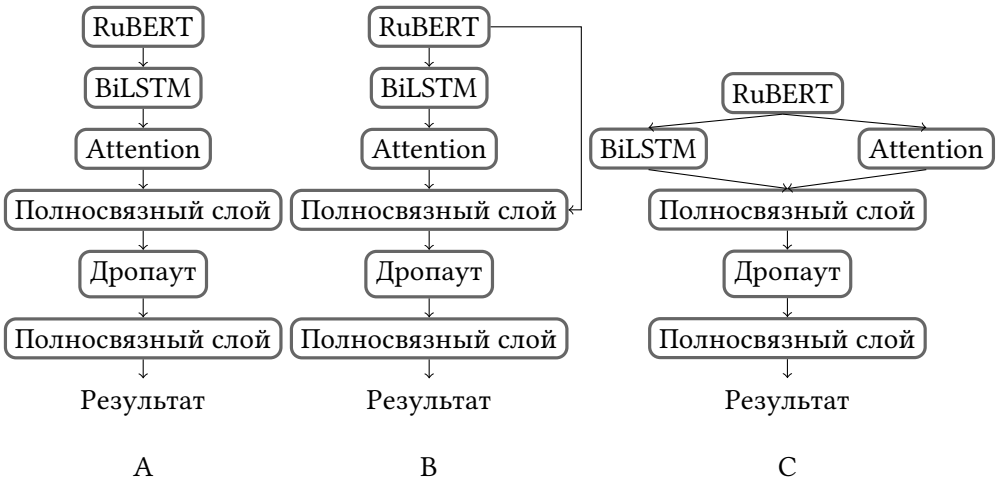


Fig. 5. RuBERT (ID)-based models featuring domain adaptation

Рис. 5. Группа моделей с адаптацией к предметной области RuBERT (ID)

обнаружения иронии на меньшем корпусе здесь является попыткой адаптации модели к заданной предметной области. В приведенных в данной группе классификаторах обучение RuBERT происходит совместно с обучением остальных слоев: каждое предложение проходит через цепочку из слоев RuBERT, BiLSTM и полносвязных слоев, в результате чего изменяются их веса, в том числе веса двенадцати внутренних слоев RuBERT.

Все модели данной группы приведены на рис. 5. Их структура аналогична ранее описываемым классификаторам с той лишь разницей, что веса внутренних слоев RuBERT не являются замороженными.

3. Корпуса текстов, используемые в экспериментах

Для проведения экспериментов в настоящей работе используется корпус русскоязычных текстов из новостных, аналитических статей и рецензий, отобранных в рамках исследования [1]. Корпус собирался из текстов, представленных на Интернет-ресурсе OpenCorpora[2]. Отобранные предложения проходили первоначальный этап разметки группой волонтеров, в ходе которого был составлен список предложений, в которых по оценке волонтеров выражается иронический или саркастический посыл. Далее выбранные волонтерами предложения прошли этап валидации экспертом, целью которого было уменьшение доли ошибок при разметке. Отобранные экспертом ироничные предложения были дополнены равным количеством предложений, не содержащих иронии. Так был составлен базовый корпус для экспериментов, содержащий 964 предложения каждого класса.

Для оценки влияния объема корпуса на качество классификации также рассматривался расширенный корпус, в который были добавлены предложения, приведенные в качестве примеров ироничного употребления слов из категории «Ироничные выражения/ru» ресурса Wiktionary. Для достижения баланса классов также было добавлено такое же количество предложений без иронии из корпуса OpenCorpora. Расширенный корпус содержит 1965 предложений каждого класса.

4. Результаты экспериментов

Результаты проведённых экспериментов приведены в таблице 1. Так как данная работа является продолжением работы [1], то полученные показатели будут сравниваться с показателями предыдущей работы. В качестве исходных показателей для сравнения берутся достигнутые в ней на базовом

[2] http://opencorpora.org

Table 1. Results of the experiments **Таблица 1.** Результаты экспериментов

Модель	Базовый корпус			Расшир. корпус		
	P	*R*	*F*	*P*	*R*	*F*
BERT	0.73	0.79	0.76	—	—	
BiLSTM	0.74	0.72	0.73	—	—	
BERT (LM) — BiLSTM	0.75	0.76	0.76	0.81	0.80	0.81
BERT (LM) — BiLSTM — Att.	0.85	0.75	0.80	0.82	0.82	0.82
BERT (LM) — BiLSTM — Att. + BERT (LM)	0.79	0.75	0.77	0.84	0.81	0.82
RoBERTa (LM) — BiLSTM — Att. + RoBERTa (LM)	—	—	—	0.84	0.84	0.84
BERT (SA) — BiLSTM — Att.	0.86	0.68	0.75	—	—	—
BERT (SA) — BiLSTM — Att. + BERT (SA)	0.87	0.72	0.79	—	—	—
BERT (fine-tuned, SA) — BiLSTM — Att. + BERT (fine-tuned, SA)	0.78	0.69	0.73	0.79	0.78	0.78
BERT (LM) — BiLSTM — Att. + BiLSTM — Att.	0.79	0.74	0.76	—	—	—
BERT (LM) — CNN — CNN — BiLSTM — Att.	0.82	0.74	0.78	0.82	0.82	0.82
BERT (ID) — BiLSTM — Att. + BERT (ID)	0.77	0.79	0.78	0.83	0.84	0.83
BERT (ID) — BiLSTM — Att.	0.73	0.79	0.76	0.74	0.84	0.78
BERT (ID) — Att. + BiLSTM	0.71	0.79	0.74	—	—	—
BERT (LM) — BiLSTM — Att. + Fts. (Punct.)	0.84	0.75	0.79	—	—	—
BERT (LM) — BiLSTM — Att. + Fts. (Punct., POS, Bigrams)	0.85	0.74	0.79	0.80	0.82	0.81
BERT (LM) — BiLSTM — Att. + Fts. (Punct., Irony Dictionary)	0.83	0.75	0.79	—	—	—

Обозначения: LM (Language Modelling) — моделирование языка; SA (Sentiment Analysis) — анализ тональности; ID (Irony Detection) — обнаружение иронии; Att. (Attention) — слой внимания; Fts. (Features) — использование дополнительных признаков: Punct. — пунктуации, POS — POS-тегов, Irony Dictionary — тонального словаря; *P* (Precision) — точность; *R* (Recall) — полнота; *F* (F-measure) — F-мера.

корпусе значения F-меры 0.76, полученное при дообучении модели RuBERT, и 0.73, продемонстрированное моделью BiLSTM с эмбеддингами spaCy[3].

Самая простая связка из RuBERT и BiLSTM на базовом корпусе показала результат, равный результату, полученному при использовании только модели RuBERT (F-мера — 0.76). Использование расширенного корпуса с этой связкой привело к одному из самых больших приростов F-меры — на 5 %. Обратная ситуация произошла со связкой, добавляющей слой внимания. На базовом корпусе она достигла F-меры 0.80, что стало лучшим результатом среди всех исследуемых моделей на базовом корпусе, но показала меньший прирост при использовании расширенного корпуса по сравнению с некоторыми другими классификаторами — 2 %. Дальнейшее усовершенствованием модели за счет добавления значений эмбеддингов к выходам слоя внимания не дало какого-либо прироста эффективности классификации ни на базовом корпусе (0.77), ни на расширенном (0.82).

Дальнейшего улучшения показателей классификации удалось добиться заменой языковой модели RuBERT на модель RuRoBERTa, которая обладает большим количеством внутренних слоев, параметров и размером эмбеддингов по сравнению с RuBERT. На расширенном корпусе данная модель достигла самого высокого результата среди всех проведенных экспериментов — 0.84. На базовом корпусе эксперимент с ней не проводился.

Среди моделей, использующих информацию о тональности, только одна показала результат близкий к удовлетворительному — связка из RuBERT, BiLSTM и слоя внимания с добавлением эмбеддингов. На базовом корпусе данная модель достигла F-меры 0.79, что на 1 % ниже лучшего результата на данном корпусе. На расширенном корпусе модель не оценивалась, так как дополнившие базовый корпус предложения не были размечены по классам тональности и ресурсов

[3] https://spacy.io/models/ru#ru_core_news_lg

на выполнение такой разметки у авторов работы не было. В целом низкие показатели классификаторов, использующих информацию о классах тональности предложений корпуса, объясняются значительным дисбалансом классов тональности. Более половины текстов имели нейтральную тональность, меньшая часть — отрицательную, и лишь очень малая часть предложений корпуса имела положительную тональность, что объясняется природой литературных оборотов иронии и сарказма.

Эксперименты с моделями, в которых RuBERT адаптировался под задачу обнаружения иронии, дали смешанные результаты. На базовом корпусе все они показали относительно низкие показатели в диапазоне 0.74–0.78, но на расширенном корпусе наиболее сложная модель данной группы, состоящая из RuBERT, BiLSTM и слоя внимания с добавлением эмбеддингов, дала второй по величине результат среди всех экспериментов — 0.83. Такой показатель может говорить о важности большого объема корпуса данных и наличия слоя внимания для адаптации модели BERT к предметной области.

Добавление в данные модели дополнительных характеристик (пунктуации, POS-тегов и т. д.) не оказало положительного влияния на показатели моделей: все классификаторы данной группы имеют почти одинаковые результаты с небольшими вариациями точности и полноты. Это может свидетельствовать о недостаточной существенности этих характеристик для решения задачи по сравнению с эмбеддингами.

Заключение

В данной работе описаны результаты исследования по разработке комплексных классификаторов для решения задачи обнаружения иронии на корпусах русскоязычных предложений. Был разработан и протестирован ряд моделей на двух корпусах текстов: базовом, используемом в предыдущей работе авторов, и расширенном. Результаты, продемонстрированные большинством разработанных классификаторов, превзошли результаты предыдущих экспериментов на базовом корпусе. Сравнение показателей моделей BiLSTM с BERT (LM) — BiLSTM и BERT (LM) — BiLSTM — Att. + BERT (LM) с RoBERTa (LM) — BiLSTM — Att. + RoBERTa (LM) явно показало, что увеличение размерности эмбеддингов, количества внутренних параметров в языковой модели и размера корпуса обучающих данных при переходе от эмбеддингов spaCy к BERT и от BERT к RoBERTa соответственно дает ощутимый прирост качества классификации модели без внесения каких-либо дополнительных модификаций в структуру. Таким образом, при наличии достаточных вычислительных мощностей можно добиться удовлетворительного качества классификации с относительно простым классификатором, если использовать большую языковую модель. Тем не менее, внедрение в модели средств выделения дополнительной информации из эмбеддингов все же показывает себя как наиболее эффективная доработка как в плане простоты реализации, так и в плане качества метрик.

Добавление в модель информации о тональности классифицируемых предложений не оказало желаемого эффекта на результаты экспериментов — все три модели данной группы показали ухудшение в метриках по сравнению с LM-моделями, что может быть вызвано дисбалансом классов тональности в размеченном корпусе. В целом составление корпуса, который был бы одновременно сбалансирован по классам тональности и классам наличия или отсутствия иронии, и обладал бы достаточным количеством предложений, является сложной задачей.

Добавление дополнительных характеристик, таких как наличие знаков пунктуации и POS-тегов, в модель также не привело к повышению метрик. В дальнейших экспериментах планируется использовать значительно более сложные характеристики (например, стилометрические), а также рассмотреть возможность учета языковых средств проявления иронии и сарказма, известных в классической лингвистике.

Для моделей, представляющих наибольший интерес, были проведены повторные эксперименты на расширенном корпусе. Во всех случаях F-мера увеличилась на 2–5 % при увеличении объема

обучающих данных, что характерно для нейронных сетей. Поскольку повышение качества классификации с объемом обучающей выборки все еще сильно сказывается на результатах классификации, составление подходящего корпуса продолжает оставаться одной из приоритетных задач.

Полученные в данной работе показатели являются лучшими на момент проведения исследования для задачи обнаружения иронии в русскоязычных текстах, а результаты наилучшего предложенного метода — RoBERTa (LM) — BiLSTM — Att. + RoBERTa (LM) — сравнимы со многими высокими показателями исследований для английского языка.

References

[1] M. Kosterin, I. Paramonov, and N. Lagutina, "Automatic irony and sarcasm detection in Russian sentences: Baseline methods", in *33rd Conference of Open Innovations Association FRUCT*, IEEE, 2023, pp. 148–154. DOI: 10.23919/FRUCT58615.2023.10142992.

[2] D. Šandor and M. B. Babac, "Sarcasm detection in online comments using machine learning", *Information Discovery and Delivery*, 2023. DOI: 10.1108/IDD-01-2023-0002.

[3] R. A. Potamias, G. Siolas, and A.-G. Stafylopatis, "A transformer-based approach to irony and sarcasm detection", *Neural Computing and Applications*, vol. 32, pp. 17 309–17 320, 2020. DOI: 10.1007/s00521-020-05102-3.

[4] C. Van Hee, E. Lefever, and V. Hoste, "Semeval-2018 task 3: Irony detection in English tweets", in *Proceedings of The 12th International Workshop on Semantic Evaluation*, 2018, pp. 39–50. DOI: 10.18653/v1/S18-1005.

[5] M. Khodak, N. Saunshi, and K. Vodrahalli, *A large self-annotated corpus for sarcasm*, 2017. arXiv: 1704.05579 [cs.CL].

[6] E. Riloff, A. Qadir, P. Surve, L. De Silva, N. Gilbert, and R. Huang, "Sarcasm as contrast between a positive sentiment and negative situation", in *Proceedings of the 2013 conference on empirical methods in natural language processing*, 2013, pp. 704–714.

[7] S. Zhang, X. Zhang, J. Chan, and P. Rosso, "Irony detection via sentiment-based transfer learning", *Information Processing & Management*, vol. 56, no. 5, pp. 1633–1644, 2019. DOI: 10.1016/j.ipm.2019.04.006.

[8] D. Hazarika, S. Poria, S. Gorantla, E. Cambria, R. Zimmermann, and R. Mihalcea, *Cascade: Contextual sarcasm detection in online discussion forums*, 2018. arXiv: 1805.06413 [cs.CL].

[9] T. Zefirova and N. Loukachevitch, "Irony and sarcasm expression in Twitter", *EPiC Series in Language and Linguistics*, vol. 4, pp. 45–49, 2019. DOI: 10.29007/tpzw.

[10] A. Gurin and T. Zhukov, "Avtomaticheskoe opredelenie sarkazma v tekstakh na russkom yazyke", *Tsyfrovaya ekonomika*, vol. 1(22), pp. 44–53, 2023, in Russian.

[11] A. D. Yacoub, S. Slim, and A. Aboutabl, "A survey of sentiment analysis and sarcasm detection: Challenges, techniques, and trends", *International journal of electrical and computer engineering systems*, vol. 15, no. 1, pp. 69–78, 2024. DOI: 10.32985/ijeces.15.1.7.

[12] Y. Kuratov and M. Arkhipov, *Adaptation of deep bidirectional multilingual transformers for Russian language*, 2019. arXiv: 1905.07213 [cs.CL].

[13] D. Zmitrovich *et al.*, *A family of pretrained transformer language models for Russian*, 2023. arXiv: 2309.10931 [cs.CL].

[14] C. Zhou, C. Sun, Z. Liu, and F. Lau, *A C-LSTM neural network for text classification*, 2015. arXiv: 1511.08630 [cs.CL].

[15] A. Rogers, A. Romanov, A. Rumshisky, S. Volkova, M. Gronas, and A. Gribov, "RuSentiment: An enriched sentiment analysis dataset for social media in Russian", in *Proceedings of the 27th international conference on computational linguistics*, 2018, pp. 755–763.

MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 31, NO. 1, 2024
journal homepage: www.mais-journal.ru

DISCRETE MATHEMATICS IN RELATION TO COMPUTER SCIENCE

NP-completeness of the Eulerian Walk Problem for a Multiple Graph

A. V. Smirnov[1]

DOI: 10.18255/1818-1015-2024-1-102-114

[1]P.G. Demidov Yaroslavl State University, Yaroslavl, Russia

MSC2020: 05C45, 05C65
Research article
Full text in Russian

Received January 22, 2024
Revised January 29, 2024
Accepted January 31, 2024

In this paper, we study undirected multiple graphs of any natural multiplicity $k > 1$. There are edges of three types: ordinary edges, multiple edges and multi-edges. Each edge of the last two types is a union of k linked edges, which connect 2 or $(k + 1)$ vertices, correspondingly. The linked edges should be used simultaneously. If a vertex is incident to a multiple edge, it can be also incident to other multiple edges and it can be the common end of k linked edges of some multi-edge. If a vertex is the common end of some multi-edge, it cannot be the common end of another multi-edge.

We study the problem of finding the Eulerian walk (the cycle or the trail) in a multiple graph, which generalizes the classical problem for an ordinary graph. We prove that the recognition variant of the multiple eulerian walk problem is NP-complete. For this purpose we first prove NP-completeness of the auxiliary problem of recognising the covering trails with given endpoints in an ordinary graph.

Keywords: multiple graph; multiple path; divisible graph; covering trails; edge-disjoint paths; eulerian trail; eulerian cycle; NP-completeness

INFORMATION ABOUT THE AUTHORS

Smirnov, Alexander V. (corresponding author)	ORCID iD: 0000-0002-0980-2507. E-mail: alexander_sm@mail.ru PhD, Associate Professor

Funding: Yaroslavl State University (project VIP-016).

For citation: A. V. Smirnov, “NP-completeness of the Eulerian walk problem for a multiple graph”, *Modeling and Analysis of Information Systems*, vol. 31, no. 1, pp. 102–114, 2024. DOI: 10.18255/1818-1015-2024-1-102-114.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 31, № 1, 2024
сайт журнала: www.mais-journal.ru

DISCRETE MATHEMATICS IN RELATION TO COMPUTER SCIENCE

NP-полнота задачи об эйлеровом маршруте в кратном графе

А. В. Смирнов[1]

DOI: 10.18255/1818-1015-2024-1-102-114

[1]Ярославский государственный университет им. П.Г. Демидова, Ярославль, Россия

УДК 519.17+519.161
Научная статья
Полный текст на русском языке

Получена 22 января 2024 г.
После доработки 29 января 2024 г.
Принята к публикации 31 января 2024 г.

В статье рассматриваются неориентированные кратные графы произвольной натуральной кратности $k > 1$. Кратный граф содержит ребра трех типов: обычные, кратные и мультиребра. Ребра последних двух типов представляют собой объединение k связанных ребер, которые соединяют 2 или $(k+1)$ вершину соответственно. Связанные ребра могут использоваться только согласованно. Если вершина инцидентна кратному ребру, то она может быть инцидентна другим кратным ребрам, а также она может быть общим концом k связанных ребер мультиребра. Если вершина является общим концом мультиребра, то она не может быть общим концом никакого другого мультиребра.

Рассматривается задача об эйлеровом маршруте (цикле или цепи) в кратном графе, которая обобщает классическую задачу для обычного графа. Доказывается, что задача о кратном эйлеровом маршруте в варианте распознавания является NP-полной. Для этого предварительно обосновывается NP-полнота вспомогательной задачи о покрывающих цепях с заданными концами в обычном графе.

Ключевые слова: кратный граф; кратный путь; делимый граф; покрывающие цепи; пути, не пересекающиеся по ребрам; эйлерова цепь; эйлеров цикл; NP-полнота

ИНФОРМАЦИЯ ОБ АВТОРАХ

Смирнов, Александр Валерьевич (автор для корреспонденции) | ORCID iD: 0000-0002-0980-2507. E-mail: alexander_sm@mail.ru
Канд. физ.-мат. наук, доцент

Финансирование: ЯрГУ (проект VIP-016).

Для цитирования: A. V. Smirnov, “NP-completeness of the Eulerian walk problem for a multiple graph”, *Modeling and Analysis of Information Systems*, vol. 31, no. 1, pp. 102–114, 2024. DOI: 10.18255/1818-1015-2024-1-102-114.

Введение

В данной статье мы рассмотрим задачу об *эйлеровом маршруте* (цикле или цепи) в кратном графе. Кратные графы содержат три типа ребер (обычные, кратные и мультиребра) и являются обобщением обычных графов — по сути, обычный граф имеет кратность $k = 1$. Определения кратного графа кратности $k > 1$ и делимого кратного графа были сформулированы в статье [1]. Там же были введены понятия кратного пути и кратного цикла.

Отметим, что частным случаем кратного графа является кратная сеть (см. [2, 3]). Задача о наибольшем потоке в кратной сети обобщает классическую задачу (см. [4]) и имеет ряд приложений в сфере экономики, управления, финансов. В частности, кратные сети и потоки используются для поиска решения NP-трудной задачи целочисленного сбалансирования трех- и четырехмерной матрицы (см., например, [5, 6]).

В статье [7] была поставлена задача об эйлеровом маршруте (цепи или цикле) в неориентированном кратном графе. Главное отличие от эйлерова маршрута в обычном графе — необходимость согласования кратного эйлерова маршрута на связанных ребрах, что означает, что в таком маршруте связанные ребра каждого кратного и мультиребра могут проходиться только одновременно и только в одинаковом направлении. Рассмотрены необходимые условия существования эйлерова маршрута в кратном графе, которые, однако, не будут достаточными из-за того, что при их выполнении не всегда возможно обеспечить согласованность всех связанных ребер в маршруте. Предложен экспоненциальный алгоритм решения задачи о кратном эйлеровом маршруте, выделен один полиномиальный подкласс задачи.

Также в работе [7] была выдвинута гипотеза об NP-трудности задачи об эйлеровом маршруте в кратном графе (что также отличает задачу от классической, для которой существуют быстрые полиномиальные алгоритмы, например, алгоритм Хиргольцера, см. [8]). В настоящей статье мы покажем, что соответствующая задача распознавания является NP-полной, откуда и будет следовать истинность указанной гипотезы.

Стоит отметить, что, хотя задача об эйлеровом маршруте в классической постановке для обычного графа полиномиальна, при переходе к обобщению обычного графа либо к варианту постановки с введением дополнительных ограничений задача зачастую становится NP-трудной.

Так происходит, например, в случае одного из наиболее популярных обобщений графов — гиперграфов. Напомним, что гиперграф содержит гиперребра, соединяющие k вершин (см. [9]). Понятие гиперребра, таким образом, в какой-то мере родственно понятию мультиребра в кратном графе, исследуемом в данной статье. Однако есть и существенное отличие двух концепций: все вершины гиперребра «равноправны», в то время как мультиребро определяется как множество из k связанных ребер, соединяющих одну вершину — общий конец с k отдельными вершинами. В работе [10] показано, что задача распознавания эйлерова маршрута в гиперграфе NP-полна даже в случае k-однородного гиперграфа ($k > 2$), в котором все ребра соединяют ровно k вершин.

В статье [11] для темпоральных графов рассмотрены различные варианты задачи об эйлеровом маршруте, среди них выделены полиномиальные и NP-полные. В темпоральном графе каждое ребро e доступно лишь в моменты времени, определяемые функцией $\lambda(e) \in 2^\tau$ ($\tau \geqslant 2$ — это общее время жизни графа), соответственно, любой маршрут должен учитывать это обстоятельство. Отметим, что NP-полной будет задача в постановке, наиболее близкой к классической, когда требуется найти цепь, проходящую каждое ребро темпорального графа ровно один раз. Это справедливо в том числе для $\tau = 2$.

В работе [12] рассмотрена классическая задача об эйлеровом цикле с дополнительным ограничением: граф изображен на плоскости и для каждой вершины все ребра, инцидентные ей, пронумерованы по часовой стрелке; при этом любые два ребра, идущие подряд в эйлеровом цикле, должны

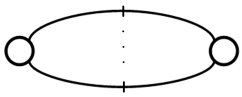


Fig. 1. Multiple edge

Рис. 1. Кратное ребро

иметь соседние номера относительно общей вершины. В указанной статье доказано, что задача распознавания такого эйлерова цикла будет NP-полной даже в случае планарного графа.

Еще один пример классической задачи с дополнительным ограничением — это распознавание эйлерова цикла, в котором каждый подцикл имеет длину не меньше k. В статье [13] обоснована NP-полнота сужения этой задачи, откуда следует NP-полнота общей задачи.

1. Постановка задачи об эйлеровом маршруте в кратном графе

Напомним несколько понятий, связанных с кратными графами и путями. Поскольку данная статья продолжает исследование, описанное в работе [7], здесь будут сформулированы только самые важные определения. Поясняющие примеры и ряд связанных определений были подробно рассмотрены в статьях [1, 7].

Определение 1. *Кратный граф* G произвольной натуральной кратности $k > 1$ — это граф, вершины которого могут соединяться ребрами одного из 3 видов:

1. *Обычное ребро* e^o; множество обычных ребер обозначим через E^o.
2. *Кратное ребро* e^k между двумя вершинами, которое состоит из k одинаковых связанных ребер; связанные ребра кратного ребра могут использоваться только согласованно; множество кратных ребер обозначим через E^k.
3. *Связанное ребро* e между двумя вершинами, имеющее один общий конец с другим $(k-1)$ ребром (у любых двух из k связанных ребер только один конец является общим); множество связанных общей вершиной ребер будем называть *мультиребром* e^m; связанные ребра мультиребра могут использоваться только согласованно; множество мультиребер обозначим через E^m.

Если вершина инцидентна какому-либо кратному ребру, то она может быть инцидентна другим кратным ребрам, а также она может быть общим концом какого-либо мультиребра.

Если вершина является общим концом какого-либо мультиребра, то она не может быть общим концом никакого другого мультиребра.

Если вершина является отдельным концом мультиребра или инцидентна обычному ребру, то она не может быть общим концом мультиребра и не может быть инцидентна кратному ребру.

Множества вершин и ребер графа G обозначим через V и E соответственно. Заметим, что $E = E^o \cup E^k \cup E^m$. Обычное или кратное ребро, соединяющее две вершины x и y, обозначается стандартным образом: $\{x, y\}$. Мультиребро, соединяющее общую вершину x с k отдельными вершинами $y_1, \ldots, y_k$, обозначается так: $e^m_x = \{x, \{y_1, \ldots, y_k\}\}$.

Рис. 1 и 2 иллюстрируют определение 1. В левой части рис. 1 кратное ребро представлено в виде объединения k одинаковых ребер между двумя вершинами, что показано штрихами. Равенство (или согласованность) связанных ребер предполагает, что все характеристики этих ребер (например, длина) одинаковы, и эти ребра могут использоваться только одновременно. Так, если осуществляется проход в определенном направлении по одному из связанных ребер, то одновременно с этим все остальные ребра проходятся в том же самом направлении. Кратное ребро может включаться в какие-либо новые структуры только целиком. В дальнейшем мы будем обозначать кратные ребра жирными линиями, как в правой части рис. 1.

В левой части рис. 2 мультиребро $\{x_0, \{x_1, \ldots, x_k\}\}$ представлено в виде объединения k одинаковых ребер, связывающих общую вершину x_0 с k разными вершинами $x_1, \ldots, x_k$. Как и на рис. 1, равен-

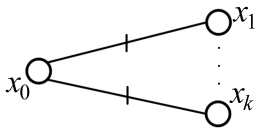


Fig. 2. Multi-edge

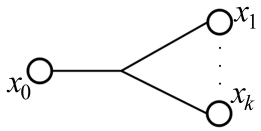


Рис. 2. Мультиребро

ство ребер показано штрихами. Согласованность связанных ребер имеет тот же смысл, что и для кратных ребер. В дальнейшем мультиребра мы будем изображать при помощи расщепляющихся на k частей линий, как в правой части рис. 2.

В данной статье рассматриваются только неориентированные кратные графы.

Определение 2. *Обычной вершиной* назовем вершину, которая инцидентна обычному ребру или является отдельным концом мультиребра.

Кратной вершиной назовем вершину, которая инцидентна кратному ребру или является общим концом мультиребра.

Из определения 1 следует, что множества обычных и кратных вершин не пересекаются. При этом кратная вершина может быть соединена с обычными только посредством мультиребра.

Определение 3. *Делимым кратным графом* назовем такой граф, в котором между двумя концами одного мультиребра не существует пути, проходящего только по обычным ребрам.

При удалении всех мультиребер делимый граф распадется на n компонент связности (связность здесь понимается в том же смысле, что и для обычных графов), каждая из которых содержит только кратные ребра либо только обычные ребра. При этом связанные ребра каждого мультиребра можно пронумеровать от 1 до k таким образом, что каждой компоненте связности, содержащей только обычные ребра, будут инцидентны связанные ребра мультиребер с одинаковыми номерами.

Определение 4. *Частью* G_i $(i \in \overline{1,k})$ делимого графа $G(V,E)$ назовем подграф, содержащий связанные ребра с номером i всех кратных и мультиребер, а также компоненты связности, состоящие из обычных ребер и инцидентные i-ым связанным ребрам всех мультиребер.

Определение 5. $S(x,y) = \cup_{i=1}^{k} S^i(x,y)$ является *кратным путем из вершины* x *в вершину* y в графе $G(V,E)$, если выполнены следующие условия:

1. $S^i(x,y) = \left(\{x, v_1^i\}, \{v_1^i, v_2^i\}, \ldots, \{v_{l_i-1}^i, v_{l_i}^i\}, \{v_{l_i}^i, y\}\right)$, где $l_i \geqslant 0$, — последовательность ребер, представляющая собой обычный (некратный) путь из x в y, где каждое ребро $\{a,b\}$ является либо обычным ребром в графе $G(V,E)$, либо i-ым связанным ребром кратного или мультиребра. Значения l_i и l_j $(i \neq j)$ не согласовываются и могут быть как равными, так и различными. Если в путь $S(x,y)$ не входит ни одного кратного или мультребра, то $S^2(x,y) = S^3(x,y) = \ldots = S^k(x,y) = \varnothing$.
2. Любая обычная вершина может встретиться в $S^i(x,y)$ несколько раз, то есть $S^i(x,y)$ может содержать циклы.
3. Никакая кратная вершина не может встретиться в $S^i(x,y)$ дважды.
4. Любое обычное ребро может встречаться в $S^i(x,y)$ несколько раз, причем направления, в которых оно проходится в разных вхождениях, могут не совпадать.
5. Обычное ребро, входящее в $S^i(x,y)$, может также входить в любой $S^j(x,y)$, $j \neq i$.
6. Все пути $S^i(x,y)$ согласованы (одинаковы) на общей части. Это условие означает, что если связанное ребро какого-то кратного или мультиребра входит в некоторый путь $S^i(x,y)$, то остальные связанные ребра должны входить во все $S^j(x,y)$, $j \neq i$ (по одному связанному

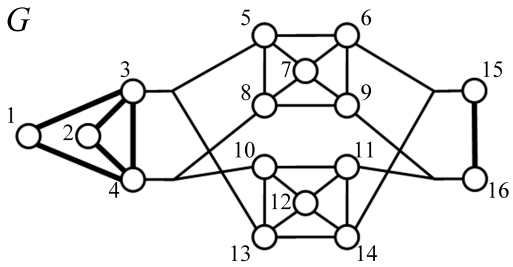


Fig. 3. Divisible graph of multiplicity 2

Рис. 3. Делимый граф кратности 2

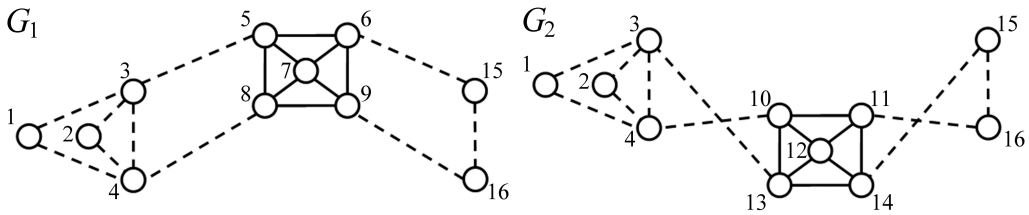


Fig. 4. Partition of a divisible graph

Рис. 4. Части делимого графа

ребру в каждый $S^j(x,y)$). При этом порядок вхождения всех кратных и мультиребер во все $S^i(x,y)$ одинаков.

Фактически это значит, что если e_1 и e_2 — это два ребра пути $S(x,y)$, каждое из которых либо кратное, либо мультиребро, и в проекции $S^i(x,y)$ связанное ребро из e_1 проходится раньше связанного ребра из e_2, то во всех остальных проекциях $S^j(x,y)$ связанные ребра из e_2 могут проходиться только после связанных ребер из e_1.

7. Если $S(x,y)$ содержит мультиребро $\{x_0,\{x_1,\ldots,x_k\}\}$, проходимое в направлении от общего конца, то он не может содержать никакого другого мультиребра $\{y_0,\{x_1,\ldots,x_k\}\}$, проходимого в том же направлении. Аналогичное условие должно выполняться и в случае движения к общему концу.

Определение 6. Кратный путь $S(x,y)$ является *кратным циклом*, если $x=y$ и $S(x,y)\neq\varnothing$.

Пример 1. Рассмотрим представленный на рис. 3 кратный граф G кратности 2 со следующими множествами обычных, кратных и мультиребер (вершины будем обозначать их номерами):

$$E^k=\big\{\{1,3\},\ \{1,4\},\ \{2,3\},\ \{2,4\},\ \{3,4\},\ \{15,16\}\big\};$$

$$E^m=\big\{\{3,\{5,13\}\},\ \{4,\{8,10\}\},\ \{15,\{6,14\}\},\ \{16,\{9,11\}\}\big\};$$

$$E^o=\big\{\{5,6\},\ \{5,7\},\ \{5,8\},\ \{6,7\},\ \{6,9\},\ \{7,8\},\ \{7,9\},\ \{8,9\},\ \{10,11\},$$
$$\{10,12\},\ \{10,13\},\ \{11,12\},\ \{11,14\},\ \{12,13\},\ \{12,14\},\ \{13,14\}\big\}.$$

Граф G является делимым. Части G_1 и G_2 этого графа показаны на рис. 4, связанные ребра всех кратных и мультиребер изображены пунктирными линиями.

Заметим, что граф перестанет быть делимым, если добавить в него обычное ребро между любой парой вершин из множеств $\{5,6,7,8,9\}$ и $\{10,11,12,13,14\}$.

Построим теперь в кратном графе $G(V,E)$ один из многочисленных возможных кратных путей $S(1,2)$ из вершины 1 в вершину 2. Он состоит из двух обычных путей:

$$S^1(1,2)=\big(\underline{\{1,3\}},\ \underline{\{3,5\}},\ \underline{\underline{\{5,6\}}},\ \{6,9\},\underline{\{9,16\}},\underline{\{16,15\}},\ \underline{\{15,6\}},\ \underline{\underline{\{6,5\}}},\ \{5,7\},\ \{7,8\},\ \underline{\{8,4\}},\ \underline{\{4,2\}}\big);$$

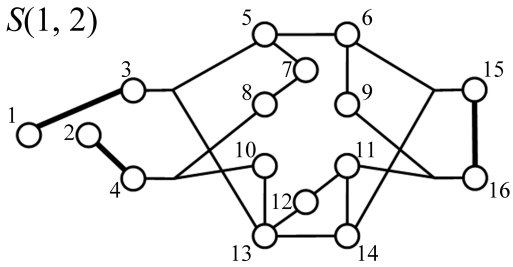


Fig. 5. Multiple path in the divisible graph

Рис. 5. Кратный путь в делимом графе

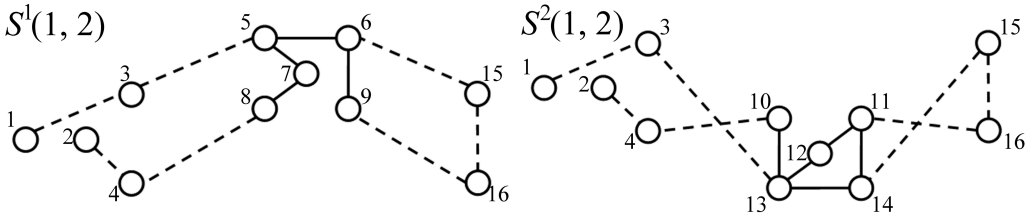


Fig. 6. Partition of the multiple path

Рис. 6. Части кратного пути

$$S^2(1,2) = \big(\underline{\{1,3\}},\ \underline{\{3,13\}},\ \{13,14\},\ \underline{\underline{\{14,11\}}},\ \underline{\{11,16\}},$$
$$\underline{\{16,15\}},\ \underline{\{15,14\}},\ \underline{\underline{\{14,11\}}},\ \{11,12\},\ \{12,13\},\ \{13,10\},\ \underline{\{10,4\}},\ \underline{\{4,2\}}\big).$$

Связанные ребра кратных и мультиребер отмечены подчеркиванием. Двойным подчеркиванием в пути $S^1(1,2)$ отмечено обычное ребро $\{5,6\}$, которое в этом пути проходится дважды, но в противоположных направлениях. Соответственно, в пути $S^2(1,2)$ двойным подчеркиванием отмечено обычное ребро $\{14,11\}$, которое также проходится дважды, но в одном и том же направлении. Таким образом, в пути $S^1(1,2)$ содержится обычный цикл $(6,9,16,15,6)$, а в пути $S^2(1,2)$ содержатся обычные циклы $(14,11,16,15,14)$ и $(13,14,11,16,15,14,11,12,13)$, а вершины 5, 6, 11, 13, 14 проходятся дважды. Однако кратный путь $S(1,2)$ не содержит в себе кратных циклов, как и должно быть (ни одна кратная вершина не проходится дважды). Полученный кратный путь $S(1,2)$ показан на рис. 5. Части $S^1(1,2)$ и $S^2(1,2)$ этого пути представлены на рис. 6.

Отметим, что при замене в кратном пути $S(1,2)$ ребра $\{4,2\}$ на ребро $\{4,1\}$ мы получим кратный цикл.

Определение 7. *Эйлеровым маршрутом* μ в кратном графе $G(V,E)$ назовем такой обход графа $G(V,E)$, в котором каждое ребро из E встречается ровно один раз, а связанные ребра каждого кратного и мультиребра из $E^k \cup E^m$ используются только согласованно (одновременно).

Как и в определении 5, для эйлерова маршрута выполнено: $\mu = \cup_{i=1}^{k} \mu^i$, то есть каждый эйлеров маршрут μ в кратном графе представляет собой объединение k обычных маршрутов μ^i $(i \in \overline{1,k})$, в каждом из которых присутствует ровно одно связанное ребро каждого кратного и мультиребра, причем порядок обхода связанных ребер одинаков во всех μ^i.

Определение 8. Замкнутый эйлеров маршрут в кратном графе (начальная вершина равна конечной) называется *эйлеровым циклом.*

Незамкнутый эйлеров маршрут в кратном графе называется *эйлеровой цепью.*

Кратный граф назовем *эйлеровым*, если в нем существует эйлеров маршрут (цикл или цепь).

Если сравнить определение кратного пути и эйлеровой цепи в кратном графе, можно заметить, что эйлерова цепь — это, по сути, кратный путь, в котором допускается использование нескольких мультиребер с одинаковыми множествами конечных вершин, проходимых в одном направлении,

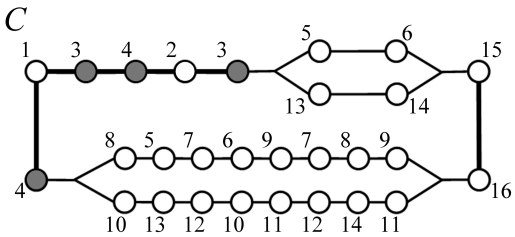


Fig. 7. Eulerian cycle in the multiple graph

Рис. 7. Эйлеров цикл в кратном графе

а также повторное использование кратных вершин, однако не допускается повтор встречавшихся ранее обычных ребер. Точно те же допущения относятся к эйлерову циклу в кратном графе в сравнении с кратным циклом.

Пример 2. Снова рассмотрим кратный граф из примера 1 (рис. 3). Нетрудно убедиться, что в этом графе существует эйлеров цикл C, представленный на рис. 7.

Следует отметить, что в эйлеровом цикле C две кратные вершины 3 и 4 проходятся дважды (на рисунке отмечены серым), поэтому цикл C является эйлеровым маршрутом, но не является кратным циклом согласно определению 6.

Поставим следующую задачу распознавания кратного эйлерова маршрута.

Задача 1 (распознавание эйлерова маршрута в кратном графе)**.**
Дано: связный кратный граф $G(V, E)$.
Вопрос: существует ли в кратном графе $G(V, E)$ эйлеров маршрут (цикл или цепь)?

В дальнейшем мы будем обозначать задачу распознавания 1 как MEW (multiple eulerian walk).

2. Покрывающие цепи с заданными концами в обычном графе

В данном разделе мы докажем NP-полноту задачи о покрывающих цепях с заданными концами в обычном графе (в постановке задачи распознавания). Соответствующая оптимизационная задача решается при построении эйлерова маршрута в кратном графе (см. [7]).

Рассмотрим сначала классическую задачу о путях, не пересекающихся по ребрам (см. [14]).

Задача 2 (пути, не пересекающиеся по ребрам)**.**
Дано: обычный связный граф $G(V, E)$, в котором выделено p начальных вершин $s_1, \ldots, s_p$ и p конечных вершин $t_1, \ldots, t_p$.
Вопрос: существуют ли в графе $G(V, E)$ p путей (цепей) $\mu_1(s_1, t_1), \ldots, \mu_p(s_p, t_p)$, попарно не пересекающихся по ребрам?

Будем использовать для задачи 2 стандартное обозначение EDP (edge-disjoint paths). В работе [14] доказано, что задача EDP является NP-полной, если параметр p не фиксирован (является частью входных данных задачи).

Часто наряду с графом $G(V, E)$ рассматривают также граф $H(V, F)$, где $F = \{\{s_1, t_1\}, \ldots, \{s_p, t_p\}\}$. Тогда для графа $G \cup H$ с множеством вершин V и множеством ребер $E \cup F$ можно рассмотреть следующую задачу распознавания, которая будет эквивалентна задаче EDP: *существует ли в графе $G \cup H$ множество из $|F|$ не пересекающихся по ребрам циклов, каждый из которых содержит ровно одно ребро из F?*

Пользуясь свойствами графа $G \cup H$, можно выделить различные подклассы исходной задачи EDP. В частности, нас будет интересовать следующая задача.

Задача 3.
Дано: обычный связный граф $G(V, E)$, в котором выделено p начальных вершин $s_1, \ldots, s_p$ и p конечных вершин $t_1, \ldots, t_p$. При этом соответствующий граф $G \cup H$ является эйлеровым.
Вопрос: существуют ли в графе $G(V, E)$ p путей (цепей) $\mu_1(s_1, t_1), \ldots, \mu_p(s_p, t_p)$, попарно не пересекающихся по ребрам?

Задачу 3 обозначим через EDPE (E — eulerian). NP-полнота задачи EDPE обосновывается в статье [15].

Теперь сформулируем задачу распознавания покрывающих цепей с заданными концами в обычном графе.

Определение 9. *Покрывающими цепями* в обычном графе $G(V, E)$ называется множество цепей, не пересекающихся по ребрам и не содержащих повторяющихся ребер; при этом множество покрывающих цепей содержит все ребра графа.

Задача 4 (покрывающие цепи с заданными концами в обычном графе)**.**
Дано: обычный связный граф $G(V, E)$, в котором выделено p начальных вершин $s_1, \ldots, s_p$ и p конечных вершин $t_1, \ldots, t_p$.
Вопрос: существуют ли в графе $G(V, E)$ p покрывающих цепей $\mu_1(s_1, t_1), \ldots, \mu_p(s_p, t_p)$?

Задачу 4 обозначим через CT (covering trails). По аналогии с задачами EDP и EDPE, рассмотрим следующее сужение CTE задачи CT (задача 5).

Задача 5.
Дано: обычный связный граф $G(V, E)$, в котором выделено p начальных вершин $s_1, \ldots, s_p$ и p конечных вершин $t_1, \ldots, t_p$. При этом соответствующий граф $G \cup H$ является эйлеровым.
Вопрос: существуют ли в графе $G(V, E)$ p покрывающих цепей $\mu_1(s_1, t_1), \ldots, \mu_p(s_p, t_p)$?

Теорема 1. *Задача CTE является NP-полной.*

Доказательство. Очевидно, что CTE $\in$ NP, так как для проверки соответствующих цепей требуется $O(|E|)$ шагов. Для этого нужно убедиться, что каждое ребро графа принадлежит ровно одной из цепей $\mu_1(s_1, t_1), \ldots, \mu_p(s_p, t_p)$ (это можно сделать, извлекая по очереди ребра цепей и помечая их в исходном графе; в итоге в графе $G(V, E)$ не должно быть непомеченных ребер или ребер, помеченных несколько раз).

Для доказательства CTE $\in$ NPC покажем, что задача EDPE $\in$ NPC полиномиально сводится к задаче CTE. В качестве функции сведения f возьмем тождественную функцию, которая каждой индивидуальной задаче $I \in$ EDPE сопоставляет индивидуальную задачу $f(I) \in$ CTE с тем же графом $G(V, E)$, с тем же значением параметра p и с теми же выделенными начальными и конечными вершинами.

Если в задаче I ответом является «нет», это означает, что не существует p путей от начальных вершин к конечным, не пересекающихся по ребрам. Однако покрывающие цепи с заданными концами — это частный случай таких путей. Следовательно, в задаче $f(I)$ ответом тоже будет «нет».

Пусть теперь в задаче I ответом является «да». Покажем, что «да» будет ответом и в задаче $f(I)$.

Найдем попарно не пересекающиеся по ребрам цепи $\mu_1(s_1, t_1), \ldots, \mu_p(s_p, t_p)$. Если $\cup_{i=1}^{p} \mu_i = G(V, E)$, то эти цепи являются покрывающими, что дает ответ «да» в задаче $f(I)$. В противном случае удаление всех цепей μ_i ($i \in \overline{1, p}$) из графа G приведет к тому, что граф распадется на r компонент связности $C_1, \ldots, C_r$ (удаляя цепи, мы удаляем ребра; вершина удаляется только в том случае, если не осталось ни одного инцидентного ей ребра). Для каждой компоненты C_j ($j \in \overline{1, r}$) выполнено:

1. Каждая вершина $v \in C_j$ имеет четную степень. Это обусловлено тем, что граф $G \cup H$ эйлеров (все вершины имеют четную степень), а удаление всех цепей μ_i из графа G эквивалентно удалению циклов из графа $G \cup H$, что соответствует уменьшению степеней всех затронутых вершин в графе $G \cup H$ на четную величину. После удаления в графе H не останется ребер, а значит, степени всех вершин в графах G и $G \cup H$ будут одинаковы. Как следствие, в компоненте C_j существует эйлеров цикл.
2. Компонента C_j имеет хотя бы одну общую вершину с какой-то цепью μ_i, что обусловлено связностью исходного графа G.

Найдем в каждой компоненте C_j ($j \in \overline{1,r}$) эйлеров цикл h_j. Далее найдем в цикле h_j вершину $v \in \mu_i$ и перестроим цепь μ_i, встроив в нее цикл h_j, начиная с вершины v.

В результате выполнения описанной процедуры цепи μ_i ($i \in \overline{1,p}$) будут содержать все ребра графа G, то есть они будут покрывающими для него. Это означает, что в задаче $f(I)$ ответом будет «да».

Следовательно, мы получили полиномиальное сведение NP-полной задачи EDPE к задаче CTE, а значит, CTE ∈ NPC. □

Следствие 1. *Задача CT является NP-полной.*

Следствие 2. *Соответствующая оптимизационная задача поиска p покрывающих цепей с заданными концами в обычном графе является NP-трудной.*

3. Доказательство NP-полноты задачи об эйлеровом маршруте в кратном графе

В данном разделе обозначения обычных графов мы будем снабжать индексом *ord*, чтобы отличать их от кратных графов. Тот же индекс будем использовать и для множеств вершин и ребер обычных графов.

Теорема 2. *Задача MEW является NP-полной.*

Доказательство. Очевидно, что MEW ∈ NP, так как для проверки эйлеровости маршрута требуется $O(|E|)$ шагов. Для этого нужно убедиться, что каждое ребро кратного графа входит в маршрут ровно один раз (это можно сделать, извлекая по очереди ребра маршрута и помечая их в исходном графе; в итоге в графе $G(V, E)$ не должно быть непомеченных ребер или ребер, помеченных несколько раз).

Будем выполнять полиномиальное сведение задачи CTE к задаче MEW.

Пусть произвольная индивидуальная задача $I \in$ CTE для произвольного p определяется обычным графом $G_{ord}(V_{ord}, E_{ord})$, в котором выделено p начальных вершин $s_1, \ldots, s_p$ и p конечных вершин $t_1, \ldots, t_p$. При этом по условию задачи соответствующий граф $G_{ord} \cup H_{ord}$ обязательно является эйлеровым.

В качестве функции полиномиального сведения f будем использовать алгоритм, который по входным данным индивидуальной задачи $I \in$ CTE строит кратный граф $G(V, E)$, представляющий собой входные данные индивидуальной задачи $f(I) \in$ MEW. Шаги алгоритма:

1. Установим $k = 2$ (кратность графа $G(V, E)$).
2. Определим множество вершин $V = V_{ord} \cup \{x_1, \ldots, x_{3p}\}$, где вершины $x_1, \ldots, x_{2p}$ будут кратными, а все остальные — обычными.
3. Определим множество обычных ребер $E^o = E_{ord}$.
4. Определим множество кратных ребер

$$E^k = \big\{\{x_2, x_3\},\ \{x_4, x_5\},\ \ldots,\ \{x_{2p-2}, x_{2p-1}\},\ \{x_{2p}, x_1\}\big\}.$$

5. В множество мультиребер поместим $2p$ мультиребер, полученных по правилу:

$$e^m_{x_{2i-1}} = \{x_{2i-1}, \{x_{2p+i}, s_i\}\}, \quad e^m_{x_{2i}} = \{x_{2i}, \{x_{2p+i}, t_i\}\} \quad (i \in \overline{1,p}).$$

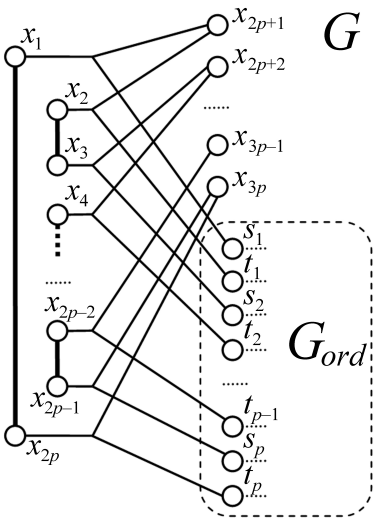


Fig. 8. Multiple graph $G(V, E)$

Рис. 8. Кратный граф $G(V, E)$

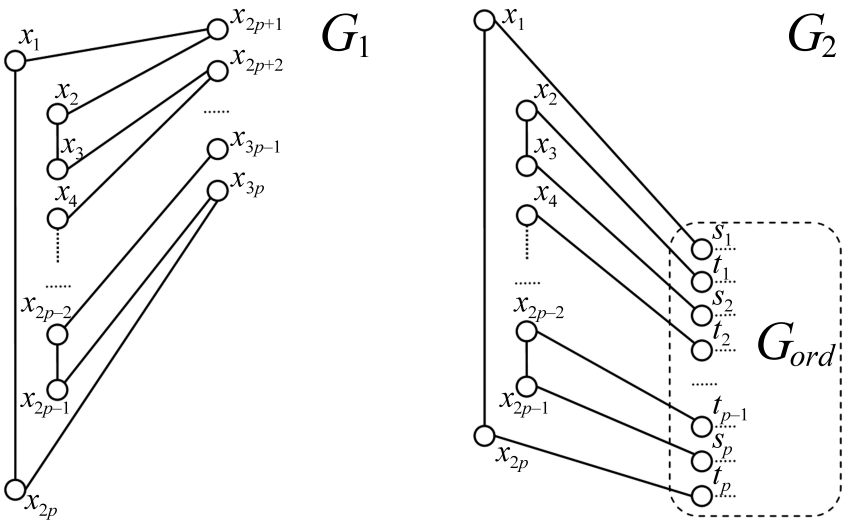


Fig. 9. Partition of the divisible graph $G(V, E)$

Рис. 9. Части делимого графа $G(V, E)$

Полиномиальность описанной процедуры очевидна: все шаги выполняются однократно, каждый шаг при этом содержит $O(p)$ операций добавления вершин или ребер. Полученный в результате кратный граф $G(V, E)$ представлен на рис. 8. Кратные ребра показаны жирными линиями, мультиребра — расщепляющимися линиями. Обычные ребра на рисунке не показаны, так как их расположение зависит от структуры конкретного графа $G_{ord}(V_{ord}, E_{ord})$.

Нетрудно убедиться, что граф G является делимым, его части G_1 и G_2 показаны на рис. 9.

Проанализируем структуру графа $G(V, E)$. Во-первых, заметим, что граф является связным в кратном смысле (каждая часть G_1 и G_2 представляет собой связный обычный граф, при этом между каждой парой кратных вершин существует кратный путь; подробно условия связности кратного графа рассматриваются в [1]).

Во-вторых, оценим степени всех вершин. Степени всех кратных вершин x_i ($i \in \overline{1, 2p}$) представляют собой чётное число, умноженное на кратность графа: $\deg x_i = 4 = k \cdot 2$ (каждое связанное ребро учитывается в значении степени). Степени всех обычных вершин x_i ($i \in \overline{2p+1, 3p}$) чётны: $\deg x_i = 2$. Чётность $\deg v$ для всех $v \in V_{ord}$ следует из того, что граф $G_{ord} \cup H_{ord}$ — эйлеров, а в кратном

графе $G(V, E)$ каждому ребру $\{s_j, t_j\}$ графа H_{ord} соответствует пара связанных ребер, одно из которых инцидентно s_j, а другое инцидентно t_j ($j \in \overline{1, p}$).

Таким образом, для кратного графа $G(V, E)$ выполняется необходимое условие существования эйлерова цикла (см. [7]).

Поскольку часть G_1 кратного графа $G(V, E)$ представляет собой простой цикл, то кратный эйлеров цикл h, если он существует, имеет вид $h = h^1 \cup h^2$, где

$$h^1 = \left(x_1,\ x_{2p+1},\ x_2,\ x_3,\ x_{2p+2},\ x_4,\ \ldots,\ x_{2p-1},\ x_{3p},\ x_{2p},\ x_1\right),$$

$$h^2 = \left(x_1,\ s_1,\ \mu_1(s_1, t_1),\ t_1,\ x_2,\ x_3,\ s_2,\ \mu_2(s_2, t_2),\ t_2,\ x_4,\ \ldots,\ x_{2p-1},\ s_p,\ \mu_p(s_p, t_p),\ t_p,\ x_{2p},\ x_1\right),$$

при этом цепи $\mu_1(s_1, t_1)$, …, $\mu_p(s_p, t_p)$ должны содержать все обычные ребра из $E^o = E_{ord}$ ровно по одному разу. Это возможно тогда и только тогда, когда данные цепи являются покрывающими цепями с заданными концами в графе $G_{ord}(V_{ord}, E_{ord})$.

Следовательно, задачи I и $f(I)$ дают либо одновременно ответ «да», либо одновременно ответ «нет». Значит, мы построили полиномиальное сведение задачи CTE к сужению задачи MEW. По теореме 1 задача CTE является NP-полной, поэтому NP-полной будет и рассмотренное сужение задачи MEW. А отсюда следует, что MEW ∈ NPC. □

Заметим, что в доказательстве теоремы 2 мы строили кратный граф минимально возможной кратности $k = 2$. Аналогичные рассуждения можно было провести и для любой другой кратности $k > 2$. В этом случае в граф $G(V, E)$ достаточно было бы добавить дополнительные обычные вершины $y_3, \ldots, y_k$, а в каждое мультиребро — $(k-2)$ связанных ребра, соединяющих общую вершину с каждой из $y_3, \ldots, y_k$.

Также заметим, что рассуждения в теореме 2 проведены для ситуации, когда в кратном графе может существовать эйлеров цикл. Если же удалить из множества кратных ребер ребро $\{x_{2p}, x_1\}$, те же самые рассуждения будут соответствовать ситуации возможного существования эйлеровой цепи.

Следствие 3. *Задача MEW для делимого кратного графа является NP-полной.*

Справедливость данного утверждения следует из того, что алгоритм из доказательства теоремы 2 всегда будет получать именно делимый кратный граф.

Следствие 4. *Задача поиска эйлерова маршрута (цикла или цепи) является NP-трудной как для произвольного, так и для делимого кратного графа.*

References

[1] A. V. Smirnov, “The shortest path problem for a multiple graph”, *Automatic Control and Computer Sciences*, vol. 52, no. 7, pp. 625–633, 2018. DOI: 10.3103/S0146411618070234.

[2] V. S. Rublev and A. V. Smirnov, “Flows in multiple networks”, *Yaroslavsky Pedagogichesky Vestnik*, vol. 3, no. 2, pp. 60–68, 2011, in Russian.

[3] A. V. Smirnov, “The problem of finding the maximum multiple flow in the divisible network and its special cases”, *Automatic Control and Computer Sciences*, vol. 50, no. 7, pp. 527–535, 2016. DOI: 10.3103/S0146411616070191.

[4] L. R. Ford and D. R. Fulkerson, *Flows in Networks*. Princeton University Press, 1962, 216 pp.

[5] V. S. Roublev and A. V. Smirnov, “The problem of integer-valued balancing of a three-dimensional matrix and algorithms of its solution”, *Modeling and Analysis of Information Systems*, vol. 17, no. 2, pp. 72–98, 2010, in Russian.

[6] A. V. Smirnov, “Network model for the problem of integer balancing of a four-dimensional matrix”, *Automatic Control and Computer Sciences*, vol. 51, no. 7, pp. 558–566, 2017. DOI: 10.3103/S0146411617070185.

[7] A. V. Smirnov, “The algorithms for the Eulerian cycle and Eulerian trail problems for a multiple graph”, *Modeling and Analysis of Information Systems*, vol. 30, no. 3, pp. 264–282, 2023, in Russian. DOI: 10.18255/1818-1015-2023-3-264-282.

[8] C. Hierholzer, “Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren”, *Mathematische Annalen*, vol. 6, no. 1, pp. 30–32, 1873, in German. DOI: 10.1007/BF01442866.

[9] C. Berge, *Graphs and Hypergraphs*. North-Holland Publishing Company, 1973, 528 pp.

[10] Z. Lonc and P. Naroski, “On tours that contain all edges of a hypergraph”, *The Electronic Journal of Combinatorics*, vol. 17, R144, 2010. DOI: 10.37236/416.

[11] A. Marino and A. Silva, “Eulerian walks in temporal graphs”, *Algoritmica*, vol. 85, no. 3, pp. 805–830, 2023. DOI: 10.1007/s00453-022-01021-y.

[12] S. W. Bent and U. Manber, “On non-intersecting Eulerian circuits”, *Discrete Applied Mathematics*, vol. 18, no. 1, pp. 87–94, 1987. DOI: 10.1016/0166-218X(87)90045-X.

[13] S. Jimbo, “The NP-completeness of Eulerian recurrent length for 4-regular Eulerian graphs”, in *Proceedings of the 2014 4th International Conference on Artificial Intelligence with Applications in Engineering and Technology*, 2014, pp. 155–159. DOI: 10.1109/ICAIET.2014.34.

[14] R. M. Karp, “On the computational complexity of combinatorial problems”, *Networks*, vol. 5, no. 1, pp. 45–68, 1975. DOI: 10.1002/net.1975.5.1.45.

[15] M. Middendorf and F. Pfeiffer, “On the complexity of the disjoint paths problem”, *Combinatorica*, vol. 13, pp. 97–107, 1993. DOI: 10.1007/BF01202792.