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.

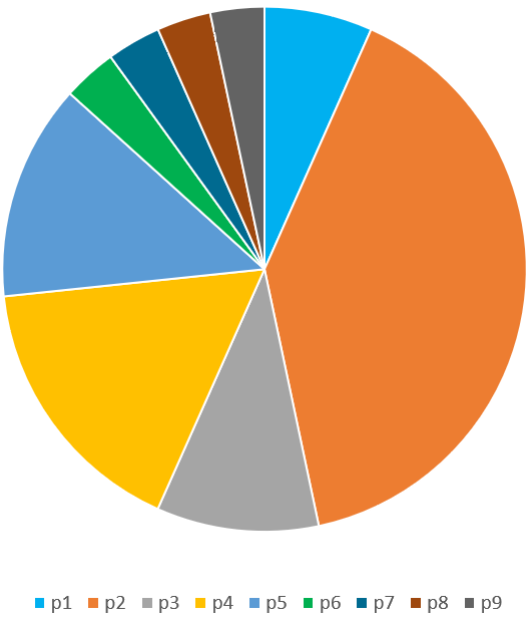

Fig. 1. Requirements distribution diagram

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.