MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 33, NO. 2, 2026
journal homepage: www.mais-journal.ru

THEORY OF SOFTWARE

Automation of Proving Verification Conditions in Deductive Verification System for poST Programs

I. M. Chernenko[1], I. S. Anureev[1]

DOI: 10.18255/1818-1015-2026-2-150-175

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

MSC2020: 68Q60
Research article
Full text in Russian

Received May 4, 2026
Revised May 20, 2026
Accepted May 26, 2026

Process-oriented programming is an approach to developing control software where a program is structured as a set of processes. PoST is a process-oriented extension of ST language from the IEC 61131-3 standard. Since control systems often have high reliability requirements, formal verification of their software plays an important role. One formal verification method is deductive verification, which involves building a formal specification, generating verification conditions, and proving them. We use the Isabelle/HOL theorem prover for the proof step. Only the generation of verification conditions is fully automated. Deductive verification itself is labor-intensive, so automating it as much as possible is desirable. Control software involves temporal requirements, which in deductive verification of process-oriented programs are expressed as control loop invariants. However, these requirements are insufficient invariants, making it necessary to introduce extra invariants that carry auxiliary information about the program. An earlier approach to proving verification conditions used patterns for both the requirements and the extra invariants, with pattern-specific lemmas satisfying predefined schemas used in the proofs. This paper looks at automating the proof of both the verification conditions and the lemmas used in those proofs. We describe the previously proposed approach to automating deductive verification and give an introduction to Isabelle/HOL. Revised schemas for patterns and lemmas are presented, along with an algorithm for generating lemma proofs. We discuss the implementation of this algorithm and of the previously developed algorithm for generating verification condition proofs. The proposed approach is demonstrated with an example. A review of related work is provided.

Keywords: deductive verification; temporal requirements; requirement pattern; loop invariant; control software; process-oriented programming

INFORMATION ABOUT THE AUTHORS

Chernenko, Ivan M. (corresponding author)	ORCID iD: 0000-0001-7675-8449. E-mail: chernenkoim@iae.nsk.su Postgraduate Student
Anureev, Igor S.	ORCID iD: 0000-0001-9574-128X. E-mail: anureev@gmail.com PhD, Senior Researcher

Funding: State task IAaE SB RAS, project No. 125022803031-1.

For citation: I. M. Chernenko and I. S. Anureev, “Automation of proving verification conditions in deductive verification system for poST programs”, *Modeling and Analysis of Information Systems*, vol. 33, no. 2, pp. 150–175, 2026.
DOI: 10.18255/1818-1015-2026-2-150-175.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 33, № 2, 2026
сайт журнала: www.mais-journal.ru

THEORY OF SOFTWARE

Автоматизация доказательства условий корректности в системе дедуктивной верификации poST-программ

И. М. Черненко[1], И. С. Ануреев[1]

DOI: 10.18255/1818-1015-2026-2-150-175

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

УДК 004.415.52
Научная статья
Полный текст на русском языке

Получена 4 мая 2026 г.
После доработки 20 мая 2026 г.
Принята к публикации 26 мая 2026 г.

Процесс-ориентированное программирование — это подход к разработке управляющего программного обеспечения, при котором программа представляется в виде набора взаимодействующих процессов. PoST представляет собой процесс-ориентированное расширение языка ST из стандарта IEC 61131-3. Поскольку к управляющим системам часто предъявляются высокие требования надёжности, важную роль играет формальная верификация используемого в них программного обеспечения. Один из методов формальной верификации — дедуктивная верификация, которая включает построение формальной спецификации, генерацию условий корректности и их доказательство. Для доказательства мы применяем систему доказательства теорем Isabelle/HOL. При этом полностью автоматизирована только генерация условий корректности. Дедуктивная верификация сама по себе трудоёмка, поэтому её желательно автоматизировать в максимально возможной степени. Для управляющего программного обеспечения характерно наличие темпоральных требований, которые при дедуктивной верификации процесс-ориентированных программ задаются в виде инвариантов цикла управления. Однако этих требований часто оказывается недостаточно как инвариантов, что требует введения дополнительных инвариантов, содержащих вспомогательную информацию о программе. Ранее был предложен подход к доказательству условий корректности, в котором требования и дополнительные инварианты задаются с помощью шаблонов, а для доказательства условий корректности используются связанные с шаблонами леммы, удовлетворяющие предопределённым схемам. В данной статье рассматривается автоматизация доказательства как условий корректности, так и самих лемм, используемых для их доказательства. Описан ранее предложенный подход к автоматизации дедуктивной верификации и даётся введение в Isabelle/HOL. Представлены скорректированные схемы шаблонов и лемм, а также алгоритм порождения доказательств для лемм. Рассмотрена реализация этого алгоритма и ранее разработанного алгоритма генерации доказательств условий корректности. Предложенный подход демонстрируется на примере. Также приводится обзор связанных работ.

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

ИНФОРМАЦИЯ ОБ АВТОРАХ

Черненко, Иван Михайлович (автор для корреспонденции) | ORCID iD: 0000-0001-7675-8449. E-mail: chernenkoim@iae.nsk.su
Аспирант

Ануреев, Игорь Сергеевич | ORCID iD: 0000-0001-9574-128X. E-mail: anureev@gmail.com
Канд. физ.-мат. наук, старший научный сотрудник

Финансирование: Госзадание ИАиЭ СО РАН, проект № 125022803031-1.

Для цитирования: I. M. Chernenko and I. S. Anureev, “Automation of proving verification conditions in deductive verification system for poST programs”, *Modeling and Analysis of Information Systems*, vol. 33, no. 2, pp. 150–175, 2026.
DOI: 10.18255/1818-1015-2026-2-150-175.

Введение

Для разработки управляющего программного обеспечения систем промышленной автоматизации на основе программируемых логических контроллеров широко применяются языки из стандарта IEC 61131-3 [1], в частности, язык Structured Text (ST). Но сложность управляющего программного обеспечения и требования к надёжности систем управления постоянно возрастают, и языки из стандарта IEC 61131-3 не всегда удовлетворяют современным требованиям к разработке такого программного обеспечения [2]. Это привлекает интерес исследователей к созданию других подходов к разработке управляющего программного обеспечения.

Перспективным подходом является процесс-ориентированное программирование [3]. Процесс-ориентированная программа определяется как упорядоченный набор взаимодействующих процессов. Процесс представляет собой конечный автомат с набором состояний, содержащих программный код. Каждый процесс имеет таймер, позволяющий контролировать время нахождения процесса в текущем состоянии. Взаимодействие с окружением в процесс-ориентированной программе реализуется с помощью входных и выходных переменных. Исполнение программы имеет циклическую основу: на каждой итерации цикла управления считываются значения входных переменных, все процессы выполняются последовательно в их текущих состояниях, и выдаются управляющие сигналы, определяемые выходными переменными. Язык poST [4] является процесс-ориентированным расширением языка Structured Text (ST) из стандарта IEC 61131-3 [1].

Так как управляющее программное обеспечение часто используется в системах управления с высокими требованиями к безопасности, требуется его верификация. Тестирование и динамическая верификация не обеспечивают достаточный уровень корректности, так как проверяют не все пути исполнения программы. Следовательно, требуется формальная верификация, основными методами которой являются проверка моделей и дедуктивная верификация. В настоящее время для верификации управляющего программного обеспечения чаще используется проверка моделей [5]. Дедуктивная верификация как правило применяется для проверки требований, не связанных с поведением системы в течение времени [6]. Основным недостатком метода проверки моделей является проблема взрыва числа состояний [7].

Дедуктивная верификация предоставляет наиболее сильные гарантии корректности программы. В этом методе, как и в методе проверки моделей, проверяются все варианты поведения программы, но дедуктивная верификация не сталкивается с проблемой взрыва числа состояний. Процесс дедуктивной верификации включает следующие этапы: 1) формализацию требований в виде логических формул; 2) построение инвариантов циклов — логических формул, истинных при входе в цикл и после каждой итерации цикла; 3) порождение условий корректности — логических формул, истинность которых гарантирует корректность программы; 4) доказательство условий корректности. Полностью автоматизирован может быть только этап порождения условий корректности. Остальные этапы часто требуется выполнять вручную. Однако эти этапы являются трудоёмкими, и их целесообразно автоматизировать в максимально возможной степени.

В работе [8] был предложен метод дедуктивной верификации процесс-ориентированных программ. В этом методе требования к программам задаются в виде инвариантов цикла управления, используя типизированную логику предикатов первого порядка. Однако SMT-решатель Z3 [9], применявшийся в этой работе, позволил доказать условия корректности только для простых требований и не справился с проверкой более сложных темпоральных требований. Это требует использования систем интерактивного доказательства теорем и делает актуальной задачу разработки стратегий доказательства условий корректности в них. В данной работе используется система Isabelle/HOL [10]. Кроме того, требования обычно описывают поведение программы в виде реакций программы на внешние события, то есть выражают отношения между значениями входных и выходных переменных в различные моменты времени. При этом не учитывается внутренняя

структура программы. Но для доказательства условий корректности необходимо использовать инвариантные свойства программы, связывающие значения входных и выходных переменных со значениями локальных переменных, состояниями процессов, и значениями таймеров. Поэтому к инварианту цикла управления, представляющему требование, добавляется дополнительный инвариант, содержащий инвариантные свойства программы. Задача автоматического синтеза дополнительных инвариантов также является актуальной.

В работе [11] был предложен подход к автоматизации дедуктивной верификации poST-программ, основанный на шаблонах. В этом подходе требования формализуются с помощью шаблонов, что позволяет порождать доказательство условий корректности на основании того, какому шаблону соответствует проверяемое требование. Определяется набор предопределенных базовых шаблонов. Шаблоны требований строятся из базовых шаблонов с помощью предопределенных операций: конъюнкции, дизъюнкции и композиции шаблонов. Используется два типа дополнительных инвариантов: зависящие от требований и не зависящие от требований. Дополнительные инварианты, зависящие от требований, также задаются с помощью шаблонов. Для доказательства условий корректности используются леммы. С каждой парой, состоящей из шаблона требований и соответствующего шаблона дополнительных инвариантов, связан набор лемм. Похожие леммы для различных шаблонов описываются схемами лемм, что позволяет использовать одни и те же методы доказательства для требований, соответствующих различным шаблонам. В этой работе были предложены алгоритмы построения производных шаблонов дополнительных инвариантов по шаблонам требований и построения лемм для шаблонов, а также скрипты доказательства условий корректности. В работе [12] был представлен генератор теории Isabelle/HOL с шаблонами и леммами, основанный на этих алгоритмах. В работе [11] также были предложены методы доказательства лемм в системе Isabelle/HOL, однако некоторые леммы не были доказаны с помощью этих методов.

В данной работе мы предлагаем алгоритм доказательства лемм, расширение генератора теории Isabelle/HOL с шаблонами и леммами функцией генерации доказательств лемм, а также реализацию генератора скриптов доказательства условий корректности.

Статья организована следующим образом. В разделе 1 описан подход к автоматизации дедуктивной верификации процесс-ориентированных программ и дано введение в систему Isabelle/HOL. В разделе 2 рассмотрен алгоритм порождения скриптов доказательства лемм. В разделе 3 приведена реализация этого алгоритма. В разделе 4 рассмотрена реализация генератора скриптов доказательства условий корректности. В разделе 5 на примере рассматривается дедуктивная верификация с помощью разработанных инструментов. В разделе 6 проведен обзор связанных работ. В заключении подводятся итоги работы и обсуждаются планы.

1. Предварительные понятия

1.1. Дедуктивная верификация процесс-ориентированных программ

Для задания темпоральных требований мы используем язык, предложенный в работе [13], который основан на типизированной логике первого порядка. В языке используется тип данных *состояние изменений*, значения которого хранят все изменения значений переменных в программе. Следующие обозначения используются при задании требований:

- $s, s_1, \ldots, s_n, r, r_1, \ldots, r_n$ — состояния;
- $A_1, \ldots, A_n, B_1, \ldots, B_n, C_1, \ldots, C_n, D_1, \ldots, D_n$ — произвольные логические формулы;
- $p(s)$ возвращает предыдущее *внешнее* состояние. *Внешнее* состояние — это состояние в точке передачи значений переменных от контроллера к объекту управления в цикле управления. В противном случае, состояние является *внутренним*;
- $s \leqslant r$ возвращает *true*, если $s = r$, или $s \leqslant p(r)$;
- $s_1 \leqslant \ldots \leqslant s_n$ — сокращение для $s_1 \leqslant s_2 \wedge \ldots \wedge s_{n-1} \leqslant s_n$;

- $s < r$ возвращает *true*, если $s \leqslant r$ и $s \neq r$;
- $e(s)$ возвращает *true*, если состояние s является внешним;
- $n(s_1, s_2)$ возвращает количество внешних состояний между состояниями s_1 и s_2, включая s_2, но не включая s_1;
- $s[x]$ — значение программной переменной x в состоянии s;
- $getPstate(s, p)$ — состояние процесса p в состоянии изменений s;
- $ltime(s, p)$ — значение таймера процесса p в состоянии s;
- $consecutive(s_1, s_2)$ возвращает *true*, если $e(s_1) \wedge e(s_2) \wedge s_1 \leqslant s_2 \wedge n(s_1, s_2) = 1$.

В предложенном подходе требования формализуются с помощью шаблонов, что позволяет порождать доказательство условий корректности на основании того, какому шаблону соответствует проверяемое требование. Определяется набор предопределенных базовых шаблонов. Базовые шаблоны делятся на шаблоны будущего и шаблоны прошлого, используемые для задания утверждений о будущем и прошлом соответственно. Шаблоны требований строятся из базовых шаблонов с помощью предопределенных операций: конъюнкции, дизъюнкции и композиции шаблонов. Так как для задания требований используются истории значений переменных, для задания требований используются только шаблоны прошлого. Шаблоны будущего могут использоваться только для определения других шаблонов и не используются непосредственно для задания требований. Все шаблоны, определяемые пользователем, являются шаблонами прошлого. Дополнительные инварианты, зависящие от требований, также задаются с помощью шаблонов. Для доказательства условий корректности используются леммы. С каждой парой, состоящей из шаблона требований и соответствующего шаблона дополнительных инвариантов, связан набор лемм. Похожие леммы для различных шаблонов описываются схемами лемм, что позволяет использовать одни и те же методы доказательства для требований, соответствующих различным шаблонам.

Для каждой пары шаблонов будущего определяются 3 леммы в соответствии со следующими схемами:

1. LS_1. Леммы, соответствующие этой схеме, используются для доказательства того, что выполняется экземпляр шаблона с конечным состоянием (т. е. состоянием, в котором выполняется инвариант цикла управления) s', соответствующим концу итерации цикла, если он выполняется с конечным состоянием s, соответствующим началу итерации.
2. LS_2. Леммы, соответствующие этой схеме, используются для доказательства экземпляров шаблонов будущего, когда текущее состояние совпадает с конечным.
3. LS_3. Леммы, соответствующие этой схеме, используются для доказательства того, что выполняется экземпляр шаблона требований, если выполняется соответствующий экземпляр шаблона дополнительных инвариантов.

Для каждой пары шаблонов прошлого определяются 6 лемм, соответствующие следующим схемам:

1. LS_4. Леммы, соответствующие этой схеме, используются для доказательства того, что выполняется экземпляр шаблона требований прошлого с конечным состоянием s', соответствующим концу итерации цикла, если он выполняется с конечным состоянием s, соответствующим началу итерации.
2. LS_5. Леммы, соответствующие этой схеме, используются для доказательства того, что выполняется экземпляр шаблона требований прошлого с определёнными значениями fm-параметров, если выполняется экземпляр того же шаблона с другими значениями fm-параметров.
3. LS_6. Леммы, соответствующие этой схеме, используются для доказательства того, что экземпляр шаблона дополнительных инвариантов прошлого выполняется в конце итерации, если он выполняется в начале итерации.

4. LS_7. Леммы, соответствующие этой схеме, используются для доказательства того, что в конце итерации выполняется экземпляр шаблона требований прошлого, если в начале итерации выполняется соответствующий экземпляр шаблона дополнительных инвариантов.
5. LS_8. Леммы, соответствующие этой схеме, используются для доказательства того, что дополнительный инвариант выполняется в конце итерации, если он выполняется в начале итерации.
6. LS_9. Леммы, соответствующие этой схеме, используются для доказательства того, что выполняется требование, если выполняется соответствующий дополнительный инвариант.

Формулы, определяющие схемы лемм, можно найти в [11]. Леммы, соответствующие схемам LS_4, LS_5, LS_6 и LS_7, используются в случаях, когда экземпляр шаблона прошлого вложен в другой экземпляр шаблона. Леммы, соответствующие схемам LS_8 и LS_9, используются в случаях, когда экземпляр шаблона прошлого не вложен в другие экземпляры шаблонов. Для шаблонов, определяемых пользователем, порождаются только леммы, соответствующие схемам LS_8 и LS_9. Поэтому эти шаблоны не могут быть вложенными в другие шаблоны.

1.2. Isabelle/HOL

В этом разделе дается краткое введение в Isabelle/HOL.

Isabelle — универсальный инструмент для интерактивного доказательства теорем. Его основой служит металогика [14], которая представляет собой интуиционистский фрагмент логики высшего порядка и включает только импликацию $\Longrightarrow$, квантор всеобщности $\bigwedge$ и равенство $\equiv$. Металогика используется для определения новых объектных логик. Наиболее распространенной является версия Isabelle/HOL — специализация Isabelle для логики высшего порядка. Isabelle/HOL предоставляет функциональный язык программирования, с помощью которого можно задавать типы данных и функции, а также позволяет доказывать леммы и теоремы. В HOL реализованы основные конструкции функционального программирования, такие как if-, case- и let-выражения.

Аналогом модулей в языках программирования в Isabelle являются теории. Они представляют собой именованные наборы определений типов данных, констант, функций, а также теорем с их доказательствами. Каждая теория с именем T определяется в файле T.thy и имеет следующий синтаксис:

```
theory T
imports T1 ... Tn
begin
    определения, теоремы и доказательства
end
```

где $T_1, \ldots, T_n$ — имена существующих теорий, на которых основана данная теория.

Команды `lemma` и `theorem` задают название теоремы и утверждение, которое необходимо доказать, и начинают её доказательство. В начальный момент доказательства есть одна подцель, совпадающая с утверждением теоремы. Доказательство может быть записано в виде скрипта, состоящего из последовательности команд, таких как `apply` и `subgoal`. Команда apply(m) применяет выбранный метод доказательства m.

К методам, выполняющим отдельные шаги доказательства, относятся методы *rule* и *erule*, позволяющие применять правила вывода. Метод *rule* $r_1 \ldots r_n$ применяет к текущей цели доказательства некоторое правило из набора правил $\{r_1,\ldots, r_n\}$. Если правило r имеет вид $P_1 \Longrightarrow \cdots \Longrightarrow P_n \Longrightarrow Q$, то его применение к цели вида $\bigwedge x_1 \ldots x_k.A_1 \Longrightarrow \cdots \Longrightarrow A_m \Longrightarrow C$ унифицирует Q и C и заменяет текущую цель новыми подцелями $\bigwedge x_1 \ldots x_k.U(A_1) \Longrightarrow \cdots \Longrightarrow U(A_m) \Longrightarrow U(P_1),\ldots,$ $\bigwedge x_1 \ldots x_k.U(A_1) \Longrightarrow \cdots \Longrightarrow U(A_m) \Longrightarrow U(P_n)$, где U — найденный унификатор. Метод *erule* $r_1 \ldots r_n$ применяет некоторое правило из набора правил $\{r_1,\ldots, r_n\}$ следующим образом. Если правило r имеет вид $P_1 \Longrightarrow \cdots \Longrightarrow P_n \Longrightarrow Q$, то его применение к цели вида $\bigwedge x_1 \ldots x_k.A_1 \Longrightarrow \cdots \Longrightarrow A_m \Longrightarrow C$ унифицирует Q с C и P_1 с A_j для некоторого j и заменяет текущую цель новыми

подцелями $\bigwedge x_1 \ldots x_k.U(A_1) \Longrightarrow \cdots \Longrightarrow U(A_{j-1}) \Longrightarrow U(A_{j+1}) \Longrightarrow \cdots \Longrightarrow U(A_m) \Longrightarrow U(P_2), \ldots,$ $\bigwedge x_1 \ldots x_k.U(A_1) \Longrightarrow \ldots \Longrightarrow U(A_{j-1}) \Longrightarrow U(A_{j+1}) \Longrightarrow \cdots \Longrightarrow U(A_m) \Longrightarrow U(P_n)$ где U — найденный унификатор.

Isabelle/HOL предоставляет автоматические средства доказательства. Одним из них является метод *simp*, который выполняет упрощения с помощью переписывания равенств.

Доказательство теоремы завершается командой `done` или by(m) (by(m) является сокращением для последовательности команд apply(m) и `done`). После завершения доказательства теоремы все её свободные переменные заменяются неизвестными, которые в дальнейшем могут быть заменены на конкретные термы либо явно с помощью атрибута *of*, либо неявно в процессе унификации. Также имеются атрибуты теорем *OF* и *THEN*. Например, атрибут *OF* позволяет применить одну теорему к другой. Если $A \Longrightarrow B$ и A' — утверждения теорем r и r' соответственно, то выражение $r[OFr']$ обозначает применение теоремы r к теореме r', где r можно рассматривать как функцию, возвращающую теорему B для заданной теоремы A.

Проверка теорий Isabelle может быть выполнена с помощью инструмента построения образов сессий `isabelle buid`. Сессии Isabelle представляют собой коллекции связанных теорий. Каждая сессия основывается на некоторой родительской сессии, такой как объектная логика HOL, и включает импортируемые сессии, теории которых используются в текущей сессии, и теории, которые должны быть обработаны.

2. Алгоритм порождения скриптов доказательства лемм

В данном разделе описываются скорректированные схемы базовых шаблонов дополнительных инвариантов прошлого и лемм для них, а также порождение доказательств лемм.

Новая схема базовых шаблонов дополнительных инвариантов прошлого имеет вид:

$$I(p_1, \ldots, p_m, b, ep_1, \ldots, ep_k, A_1, \ldots, A_n, s),$$

где I — имя шаблона, $p_1, \ldots, p_m$ — константные параметры (далее c-параметры), b, $ep_1, \ldots, ep_k$ — функциональные параметры (далее fn-параметры), $A_1, \ldots, A_n$ — формульные параметры (далее fm-параметры), s — текущее состояние, в котором выполняется данный экземпляр шаблона. Логический fn-параметр b определяет условие, при котором задаваемое данным шаблоном свойство необходимо для доказательства требования: при ложности терма $b(s)$ экземпляр I тривиально истинен. В противном случае, когда $b(s)$ истинно, шаблон задаёт вспомогательное утверждение, необходимое для доказательства целевого требования.

Определение производного шаблона дополнительных инвариантов имеет вид:

$$I_0 \wedge I_1 \wedge \ldots \wedge I_l$$

где I_0 — экземпляр некоторого шаблона дополнительных инвариантов, а $I_1, \ldots, I_n$ — дополнительные конъюнкты. В подходе, предложенном в работе [11], дополнительные конъюнкты имели вид $b(s) \longrightarrow P$, где b — дополнительный логический fn-параметр, а P — экземпляр некоторого шаблона дополнительных инвариантов прошлого. В данной работе дополнительные конъюнкты $I_1, \ldots, I_l$ являются экземплярами шаблонов прошлого. Следовательно, если определение производного шаблона дополнительных инвариантов содержит дополнительные конъюнкты, содержащие экземпляры шаблонов дополнительных инвариантов прошлого, то эти экземпляры выполняются во всех состояниях изменений.

Описанное изменение схем шаблонов дополнительных инвариантов позволяет привести схемы лемм LS_6 и LS_7 к следующему виду:

- LS_6

$$I(p_1, \ldots, p_m, b, ep_1, ep_k, A'_1, \ldots, A'_n, s) \Longrightarrow$$
$$consecutive(s, s') \Longrightarrow$$
$$cond(p_1, \ldots, p_m, ep_1(s), \ldots, ep_k(s), ep_1(s'), \ldots, ep_k(s'), A_1(s', s'), \ldots, A_n(s', s'),$$
$$(\forall s_1.e(s_1) \wedge s_1 \leqslant s \wedge A'_1(s, s_1) \longrightarrow A'_1(s', s_1)), \ldots, (\forall s_1.e(s_1) \wedge s_1 \leqslant s \wedge A'_n(s, s_1) \longrightarrow A'_n(s', s_1))) \Longrightarrow$$
$$I(p_1, \ldots, p_m, b, ep_1, \ldots, ep_k, A'_1, \ldots, A'_n, s')$$

- LS_7

$$I(p_1, \ldots, p_m, b, ep_1, \ldots, ep_k, A'_1, \ldots, A'_n, s) \Longrightarrow$$
$$consecutive(s, s') \Longrightarrow$$
$$cond(p_1, \ldots, p_m, ep_1(s), \ldots, ep_k(s), ep_1(s'), \ldots, ep_k(s'), A'_1(s', s'), \ldots, A'_n(s', s'),$$
$$(\forall s_1.e(s_1) \wedge s_1 \leqslant s \wedge A'_1(s, s_1) \longrightarrow A'_1(s', s_1)), \ldots, (\forall s_1.e(s_1) \wedge s_1 \leqslant s \wedge A'_n(s, s_1) \longrightarrow A'_n(s', s_1))) \Longrightarrow$$
$$R(s', s', p_1, \ldots, p_m, A'_1, \ldots, A'_n, s', s')$$

Леммы, соответствующие схеме LS_6, утверждают, что экземпляр соответствующего шаблона дополнительных инвариантов выполняется в конце итерации в состоянии s', если он выполняется в начале итерации в состоянии s и выполняется некоторое условие *cond*, связывающее значения параметров шаблона. Леммы, соответствующие схеме LS_7, утверждают, что экземпляр шаблона требований прошлого выполняется в конце итерации, если в начале итерации выполняется соответствующий экземпляр шаблона дополнительных инвариантов, и выполняется дополнительное условие *cond*, связывающее значения параметров шаблонов. Другие схемы лемм и алгоритм построения лемм для производных шаблонов остаются без изменений.

Введём рекурсивные функции $proofL8(I, s, s')$ и $proofL9(I, R, s')$, которые порождают скрипты доказательства для формул, возникающих при доказательстве лемм, соответствующих схемам LS_8 и LS_9. Параметры I, R, s и s' обозначают подформулу экземпляра шаблона дополнительных инвариантов, подформулу экземпляра шаблона требований, начальное и конечное состояния изменений, то есть состояния в начале и в конце итерации.

Функция $proofInner(f, s)$ порождает скрипт доказательства для формулы f, возникшей после применения лемм и не являющейся подформулой шаблона. Опишем определение функций *proofL8*, *proofL9* и *proofInner* для различных типов формул, используя псевдокод.

Для конъюнкции функция *proofL8* определяется следующим образом:

```
1  proofL8(I'_1 ∧ I'_2, s, s') {
2    premisesName = genPremisesName();
3    return '''
4      apply(erule conjE)
5      apply(erule conjE)
6      subgoal premises «premisesName»
7      apply(rule conjI)
8      apply(insert «premisesName»(1,2,4))[1]
9        «proofL8(I'_1, s, s')»
10     apply(insert «premisesName»(1,3,5))
11       «proofL8(I'_2, s, s')»
12     done
13     '''
14     }
```

В определении данной функции *genPremiseName* представляет собой функцию, генерирующую имя, задаваемое посылкам подцели. Текст, содержащийся в тройных кавычках ''' является скриптом доказательства или его частью. Скрипт доказательства может содержать имена переменных и вызовы функций, содержащихся в кавычках «». Вместо имен переменных в скрипт подставляются их значения, а вместо вызовов функций — возвращаемые значения. Если заключение доказываемой цели является подформулой шаблона, то цель имеет три посылки: 1) формула c, имеющая вид $e(s)$ или $consecutive(s, s')$, 2) соответствующая подформула дополнительного инварианта в посылке леммы, 3) подформула формулы *cond* доказываемой леммы. Порядок посылок подцели может быть

любым. Если заключение текущей цели является конъюнкцией $I'_1 \wedge i'_2$, то посылки 2 и 3 являются конъюнкциями $I_1 \wedge I_2$ и $p_1 \wedge p_2$ для некоторых формул I_1, I_2, p_1 и p_2. Следовательно, цель имеет вид $c \Longrightarrow I_1 \wedge I_2 \Longrightarrow p_1 \wedge p_2 \Longrightarrow I'_1 \wedge I'_2$ и её доказательство сводится к доказательству подцелей $c \Longrightarrow I_1 \Longrightarrow p_1 \Longrightarrow I'_1$ и $c \Longrightarrow I_2 \Longrightarrow p_2 \Longrightarrow I'_2$. Доказательства этих подцелей порождаются функцией *proofL8*

Для дизъюнкции функция *proofL8* определяется следующим образом:

```
proofL8(I'_1 ∨ I'_2, s, s') {
  premisesName = genPremisesName();
  return '''
    apply(erule conjE)
    subgoal premises «premisesName»
    apply(insert «premisesName»(1,2)
    apply(erule disjE)
    apply(insert «premisesName»(3)
    apply(rule disjI1)
      «proof(I_1, s, s')»
    apply(insert «premisesName»(4)
    apply(rule disjI2)
      «proof(I_2, s, s')»
    done
    '''
}
```

Если заключение текущей цели является дизъюнкцией $I'_1 \vee I'_2$, то посылка 2 является дизъюнкцией $I_1 \vee I_2$, а посылка 3 — конъюнкцией $p_1 \wedge p_2$. Следовательно, цель имеет вид $c \Longrightarrow I_1 \vee I_2 \Longrightarrow p_1 \wedge p_2 \Longrightarrow I'_1 \vee I'_2$, и её доказательство сводится к доказательству подцелей $c \Longrightarrow I_1 \Longrightarrow p_1 \Longrightarrow I'_1$ и $c \Longrightarrow I_2 \Longrightarrow p_2 \Longrightarrow I'_2$. Доказательства этих подцелей порождаются функцией *proofL8*.

Функция *proofL9* для конъюнкции и дизъюнкции определяется аналогично функции *proofL8*.

Для экземпляров шаблонов дополнительных инвариантов прошлого функция *proofL8* определяется следующим образом:

```
proofL8(I, s, s') {
  return '''
    apply(erule «getLemma(I)»)
      apply simp
      «proofInner(L8premiseInstance(I, s, s'), s)»
    '''
 }
```

Пусть P — шаблон, экземпляром которого является I. Если I является экземпляром шаблона прошлого, то посылка 2 является тем же экземпляром, выполняющимся в начальном состоянии s. Доказательство выполняется следующим образом. Сначала применяется лемма, удовлетворяющая схеме LS_8, которая связана с шаблоном P. В результате применения леммы возникают две подцели. Первая подцель имеет вид $consecutive(s, s')$ и доказывается методом *simp*. Вторая подцель имеет вид

$$consecutive(s, s') \Longrightarrow p \Longrightarrow p' \tag{1}$$

В данной функции используются функции *getLemma* и *L8premiseInstance*. Функция $getLemma(I)$ возвращает имя леммы, удовлетворяющей схеме LS_8. Функция $L8premiseInstance(I, s)$ вычисляет формулу p'.

Для экземпляра шаблона требований прошлого функция *proofL9* определяется следующим образом:

```
proofL9(I_0 ∧ I_1 ∧ ... ∧ I_l, R, s') {
  extraConjs = [I_1, ..., I_i];
```

```
  premises1 = genPremisesName();
  premises2 = genPremisesName();
  script = "";
  if (l > 0) {
    script += '''
      subgoal premises «premises1»
      apply(insert «premises1»(1,2))
    '''
    if (l > 1) {
      script += "apply(simp only: conj_assoc) "
    }
    script += '''
    apply(erule conjE)
    subgoal premises «premises2»
    apply(insert «premises2»(1,2) «premises1»(3))
    '''
  }
  script += '''
    apply(erule «getLemma(R)»)
    apply simp
    «proofInner(L9PremiseInstance(I_0, R, s'), null)»
    '''
    if (l > 0) {
      script += '''
        done
        done
        '''
    }
  return script;
```

Пусть P_R — шаблон, экземпляром которого является заключение текущей цели R. Если P_R — шаблон требований прошлого, то посылка 2 имеет вид $i_0 \wedge I_1 \wedge \ldots \wedge I_l$, где I_0 — соответствующий экземпляр шаблона дополнительных инвариантов, а $I_1, \ldots, I_l$ — дополнительные конъюнкты, Дополнительные конъюнкты не используются для доказательства требования. Цель имеет вид $e(s) \Longrightarrow I_0 \wedge I_1 \wedge \ldots \wedge I_l \Longrightarrow p \Longrightarrow R$, где p — некоторая формула. Доказательство данной цели сводится к доказательству цели $e(s) \Longrightarrow I_0 \Longrightarrow p \Longrightarrow R$ путём удаления дополнительных конъюнктов $I_1, \ldots, I_l$. Далее применяется лемма, соответствующая схеме LS_9, связанная с шаблоном P_R. В результате применения леммы возникают две подцели. Первая подцель имеет вид $e(s)$ и доказывается методом *simp*. Вторая подцель имеет вид

$$e(s) \Longrightarrow p \Longrightarrow p', \tag{2}$$

где p' — некоторая формула. Доказательство этой подцели порождается функцией *proofInner*.

В данной функции используется функция *L9PremiseInstance*, вычисляющая формулу p'.

Функция $proofInner(p', s)$ порождает доказательства для подцелей, имеющих вид (1) или (2).

Для конъюнкции функция *proofInner* определяется следующим образом:

```
proofInner(p'_1 ∧ p'_2, s) {
  premisesName = genPremisesName();
  return '''
    apply(erule conjE)
    subgoal premises «premisesName»
    apply(rule conjI)
    apply(insert «premisesName»(1,2))[1]
      «proofInner(p'_1, s)»
    apply(insert «premisesName»(1,3))
      «proofInner(p'_2, s)»
    done
```

```
    ’’’
}
```

Если заключение текущей цели является конъюнкцией $p'_1 \wedge p'_2$, то посылка p также является конъюнкцией $p_1 \wedge p_2$, где p_1 и p_2 — некоторые формулы. Следовательно, цель имеет вид $c \Longrightarrow p_1 \wedge p_2 \Longrightarrow p'_1 \wedge p'_2$, где c имеет вид $consecutive(s, s')$ или $e(s)$, и её доказательство сводится к доказательству подцелей $c \Longrightarrow p_1 \Longrightarrow p'_1$ и $c \Longrightarrow p_2 \Longrightarrow p'_2$. Доказательство этих подцелей порождается функцией *proofInner*.

Для дизъюнкции функция *proofInner* определяется следующим образом:

```
proofInner(p'_1 ∨ p'_2, s) {
  return ’’’
    apply(erule disjE)
    apply(rule disjI1)
      «proofInner(p_1, s)»
    apply(rule disjI2)
      «proofInner(P_2, s)»
    ’’’
}
```

Если заключение текущей цели является имеет вид дизъюнкции $p'_1 \vee p'_2$, то посылка p также является дизъюнкцией $p_1 \vee p_2$, где p_1, p_2 — некоторые формулы. Следовательно, цель имеет вид $c \Longrightarrow p_1 \vee p_2 \Longrightarrow p'_1 \vee p'_2$, и её доказательство сводится к доказательству подцелей $c \Longrightarrow p_1 \Longrightarrow p'_1$ и $c \Longrightarrow p_2 \Longrightarrow p'_2$. Доказательство этих подцелей порождается функцией *proofInner*.

Для импликации функция *proofInner* определяется следующим образом:

```
proofInner(b ⟶ p'_1, s) {
  premiseName = genPremiseName();
  return ’’’
    apply(rule impI)
    apply(erule impE)
    apply assumption
    subgoal premises «premisesName»
    apply(insert «premisesName»(1,3))
      «proofInner(P, s)»
    done
    ’’’
}
```

Если заключение текущей цели является импликацией $b(s') \longrightarrow p'_1$, где b — fn-параметр, p'_1 — некоторая формула, то посылка p является импликацией $b(s') \longrightarrow p_1$, где p_1 — некоторая формула, то есть посылки импликаций совпадают. Следовательно, цель имеет вид $c \Longrightarrow b(s') \longrightarrow p_1 \Longrightarrow b(s') \longrightarrow p'_1$, где c имеет вид $consecutive(s, s')$, и её доказательство сводится к доказательству подцели $c \Longrightarrow p_1 \Longrightarrow p'_1$. Доказательство для этой подцели порождается функцией *proofInner*.

Для экземпляра шаблона дополнительных инвариантов будущего функция *proofInner* определяется следующим образом.

```
proofInner(P, s) {
  return ’’’
    apply(rule «getLemma(P)»)
    apply simp
      «proofInner(premiseInstance(P, s), s)»
    ’’’
}
```

Если заключение текущей подцели является экземпляром шаблона дополнительных инвариантов будущего, то доказательство выполняется следующим образом. Сначала применяется лемма,

удовлетворяющая схеме LS_2, связанная с этим шаблоном, имя которой возвращается функцией *getLemma*. В результате применения леммы возникает две подцели. Первая подцель имеет вид $e(s)$ и доказывается методом *simp*. Доказательство второй подцели порождается рекурсивным вызовом функции *proofInner*.

Доказательство для формул вида $\forall s_1.e(s_1) \land s_1 \leqslant s \land A(s_1) \longrightarrow A'(s_1)$, где A и A' являются конъюнкциями, дизъюнкциями или экземплярами шаблонов и $A \neq A'$, выполняется аналогично. Но используются следующие леммы:

- если A и A' являются конъюнкциями, применяется лемма L_{conj}

  $e(s) \Longrightarrow$
  $(\forall s_1.e(s_1) \land s_1 \leqslant s \land A(s_1) \longrightarrow A'(s_1)) \land (\forall s_1.e(s_1) \land s_1 \leqslant s \land B(s_1) \longrightarrow B'(s_1)) \Longrightarrow$
  $(\forall s_1.e(s_1) \land s_1 \leqslant s \land A(s_1) \land B(s_1) \longrightarrow A'(s_1) \land B'(s_1))$;

- если A и A' являются дизъюнкциями, то применяется лемма L_{disj}

  $e(s) \Longrightarrow$
  $(\forall s_1.e(s_1) \land s_1 \leqslant s \land A(s_1) \longrightarrow A'(s_1)) \land (\forall s_1.e(s_1) \land s_1 \leqslant s \land B(s_1) \longrightarrow B'(s_1)) \Longrightarrow$
  $(\forall s_1.e(s_1) \land s_1 \leqslant s \land (A(s_1) \lor B(s_1)) \longrightarrow (A'(s_1) \lor B'(s_1))$;

- если A и A' являются экземплярами шаблона дополнительных инвариантов будущего, применяется лемма, соответствующая схеме LS_1, связанная с этим шаблоном;
- если A является экземпляром шаблона дополнительных инвариантов будущего, а A' — экземпляром соответствующего шаблона требований, то применяется лемма, удовлетворяющая схеме LS_3, связанная с этими шаблонами;
- если A и A' являются экземплярами шаблона требований прошлого, и конечные состояния различны, то применяется лемма, удовлетворяющая схеме LS_4, связанная с этим шаблоном;
- если A и A' являются экземплярами шаблона требований прошлого, и конечные состояния равны, то применяется лемма, удовлетворяющая схеме LS_5, связанная с этим шаблоном.

Для экземпляра шаблона требований прошлого функция *proofInner* определяется следующим образом:

```
proofInner(R, s) {
  return '''
    apply(rule «getLemma(P)»[OF «getExtraInvName(R)»])
    apply simp
      «proofInner(premiseInstance(R, s), s)»
    '''
}
```

Пусть P — шаблон требований, экземпляром которого является заключение текущей цели R, L — лемма, удовлетворяющая схеме LS_7, связанная с шаблоном P, I — экземпляр шаблона дополнительных инвариантов, соответствующий R. Доказательство экземпляра шаблона требований прошлого выполняется аналогично, но в качестве правила используется правило, полученное применением леммы L к I с помощью атрибута *OF*. Имя посылки в контексте доказательства, представляющего I, возвращается функцией *getExtraInvName*.

Утверждения вида $\forall s_1.e(s_1) \land s_1 \leqslant s \land A(s_1) \longrightarrow A(s_1)$ доказываются с помощью леммы L, утверждающей, что данные формулы являются тождественно истинными. Функция *proofInner* для формул такого вида определяется следующим образом:

```
proofInner(∀s1.e(s1) ∧ s1 ⩽ s ∧ A(s1) ⟶ A(s1), s) {
  return '''
    apply(rule L)
    '''
}
```

Если заключение текущей цели является отрицанием конъюнкции или дизъюнкции, то применяется соответствующий закон де Моргана, а затем доказательство для полученной подцели порождается рекурсивным вызовом функции *proofInner*. Для отрицания конъюнкции функция *proofInner* определяется следующим образом:

```
proofInner(¬(p'₁ ∧ p'₂), s) {
  return '''
    apply(subst (1) de_Morgan_conj)
      «proofInner(¬p'₁ ∨ ¬p'₂, s)»
    '''
},
```

где *de_Morgan_conj* — лемма из стандартной библиотеки Isabelle/HOL. Для отрицания дизъюнкции функция *proofInner* определяется аналогично. Вместо леммы *de_Morgan_conj* используется лемма *de_Morgan_disj* из стандартной библиотеки Isabelle/HOL.

Для двойного отрицания функция *proofInner* определяется следующим образом:

```
proofInner(¬¬p', s) {
  return '''
    apply(rule cnf.clause2raw_not_not)
      «proofInner(p' s)»
    '''
}
```

Если заключение текущей цели является двойным отрицанием $\neg\neg p'$, то оно преобразуется в p' с помощью леммы $cnf.clause2raw\_not\_not$ из стандартной библиотеки Isabelle/HOL, имеющей вид $P \Longrightarrow \neg\neg P$, после чего доказательство полученной подцели порождается функцией *proofInner*.

В остальных случаях заключение текущей цели является атомарной формулой, отрицанием атомарной формулы или формулой вида $\forall s_1.e(s) \wedge s_1 \leqslant s \wedge A(s_1) \longrightarrow A'(s_1)$, где $A(s_1$ и $A'(s_1)$ — атомарные формулы или их отрицания и $A \neq A'$. В этом случае посылка p цели совпадает с заключением, и доказательство выполняется методом *assumption*. Функция *proofInner* для целей такого вида имеет вид:

```
proofInner(p', s) {
  return "apply assumption"
}
```

3. Реализация алгоритма порождения скриптов доказательства лемм

В данном разделе рассмотрим реализацию описанного выше алгоритма.

Ранее был разработан язык спецификации шаблонов, позволяющий определять производные шаблоны требований. Синтаксический анализатор для этого языка был создан с помощью фреймворка Xtext [15]. Ранее также был разработан генератор теории Isabelle/HOL, содержащей определения шаблонов и леммы. Генератор был реализован на языке Java. В данной работе мы дополняем этот генератор генератором скриптов доказательства порожденных лемм.

Фреймворк Xtext предоставляет Java-подобный язык Xtend, поддерживающий шаблонные выражения — строки, которые могут содержать выражения, значения которых вычисляются и подставляются в строку, а также поддерживает автоматическое добавление отступов к строкам для обеспечения форматирования выходного текста. Это делает Xtend удобным инструментом для создания генераторов кода. Мы используем язык Xtend для реализации алгоритма генерации скриптов доказательства лемм. Для реализации функций определен класс *ProofScriptGenerator* на языке Xtend.

На вход программе подаётся файл, содержащий спецификацию шаблонов на ранее разработанном языке. Для определения производных шаблонов могут использоваться как шаблоны, объявленные в том же файле, так и шаблоны, импортированные из других файлов. Для реализации

импорта шаблонов использовались средства предоставляемые Xtext. При определении производного шаблона также могут использоваться производные шаблоны, определенные в том же файле раньше (в текстовом порядке). В класс *OuterFormulaGenerator*, в котором реализовано вычисление подформул шаблонов, не вложенных в другие шаблоны, добавлен ассоциативный массив, в котором ключами являются имена шаблонов, а значениями — порождённые шаблоны со связанными леммами. Это позволяет при определении шаблона использовать шаблоны, определенные в этом файле.

Для импорта шаблонов из других файлов в языке спецификации шаблонов используется конструкция import < file > from <session>, где < file > — путь к импортируемому файлу с объявлениями шаблонов и леммами на языке спецификации шаблонов, <session> — имя сессии Isabelle, которая содержит теорию с определениями этих шаблонов и соответствующими леммами с их доказательствами. Конструкция from <sesssion> является необязательной. Если она отсутствует, то предполагается, что импортируемый файл и соответствующая теория Isabelle находятся в том же каталоге, в котором находится текущий файл. Если сессия задана, то импортируемый файл может находиться в другом каталоге, а порожденная теория Isabelle с определениями шаблонов, леммами и их доказательствами импортирует соответствующую теорию из указанной сессии.

4. Реализация алгоритма порождения скриптов доказательства условий корректности

Ранее в работе [11] были предложены схемы доказательства условий корректности. В данном разделе описана реализация генератора скриптов доказательства условий корректности, основанного на этих схемах.

Генератор скриптов доказательства условий корректности poST-программ работает в режиме командной строки. Программа имеет две опции. С помощью опции -s задаётся путь к файлу, в котором задано соответствие между требованиями и шаблонами, а также имена дополнительных инвариантов, соответствующих этим требованиям. Для порождения скриптов доказательства необходимы имена лемм, связанных с шаблонами, которым удовлетворяют требования. Поэтому этот файл импортирует файлы, содержащие объявления используемых шаблонов и связанные с ними леммы. Параметр -cei задаёт имя общего (не зависящего от требований) дополнительного инварианта. Общий дополнительный инвариант должен быть определен в теории Isabelle/HOL в текущем каталоге, имя которой определяется по определенным правилам по имени инварианта.

Для каждого файла, содержащего условия корректности программа создаёт следующие файлы:

- для каждого дополнительного инварианта, включая независящий от требований, создаётся теория Isabelle/HOL, содержащая доказательства условий корректности для этого инварианта;
- для каждого требования создаются теория Isabelle/HOL, содержащая доказательства условий корректности для соответствующего дополнительного инварианта, а также теория Isabelle/HOL, содержащая определение расширенного инварианта и доказательства условий корректности для этого расширенного инварианта.

Для порождения теорий Isabelle/HOL необходима информация об именах теорий, содержащих условия корректности, и именах условий корректности, содержащихся в них. Также необходимо определить имена параметров условий корректности, соответствующих значениям входных переменных. Эта информация может быть получена от генератора условий корректности, но для её получения потребовалось изменить формат его выходных данных.

Опишем формат выходных данных новой версии генератора условий корректности и входных данных генератора скриптов их доказательства. Генератор условий корректности создает следующие теории Isabelle/HOL:

- теорию, содержащую имена переменных, процессов и состояний процессов, а также программные константы. Имя этой теории совпадает с именем файла, содержащего программу на языке poST;
- одну или несколько теорий, содержащих порождённые условия корректности. Максимальное количество условий корректности, сохраняемых в одну теорию может быть задано опцией командной строки. Если это значение не указано, используется значение по умолчанию. Если указано значение 0 или отрицательное значение, все условия корректности сохраняются в один файл.

В стандартный поток вывода генератор условий корректности выдаёт данные в следующем формате. В первой строке содержатся разделённые пробелами имена параметров условий корректности, соответствующих входным переменным poST-программы. Каждая следующая строка имеет вид: *<VCTheory, start, end>*, где *VCTheory* — имя теории Isabelle/HOL, содержащей условия корректности с номерами i, $start \leqslant i \leqslant end$. Вывод генератора условий корректности подаётся через стандартный поток ввода на вход генератору скриптов доказательства условий корректности.

Генератор скриптов доказательства условий корректности реализован с использованием языка Xtend.

Исходный код генератора теории Isabelle/HOL с шаблонами и леммами и генератора скриптов доказательства условий корректности можно найти на GitHub[1].

5. Пример

В этом разделе рассмотрим пример определения производных шаблонов требований, порождение теории Isabelle/HOL с определениями этих шаблонов и соответствующих шаблонов дополнительных инвариантов, леммами и их доказательствами, а также задание требований и дополнительных инвариантов к poST-программе с помощью шаблонов.

5.1. Программа

В качестве примера рассмотрим программу управления турникетом.

Турникет включает в себя монетоприёмник, двери, открывающиеся по сигналу *open*, светодиод *enter*, показывающий возможность прохода, а также датчики присутствия пользователя на выходе из турникета *PdOut* и открытия дверей *opened*. Двери остаются закрытыми до получения сигнала оплаты *paid* и открываются при его получении. Если пользователь не пройдёт в течение 10 секунд, двери закрываются. Монетоприёмник блокируется после оплаты и разблокируется по сигналу *reset* после закрытия турникета.

В программе управления турникетом объявлена локальная переменная *passed* и определены четыре процесса *Init*, *Controller*, *Indicator* и *Unlocker*. Процесс *Init* имеет единственное активное состояние *init*, в котором он запускает процессы *Controller* и *Indicator* и останавливается.

Процесс *Controller* обрабатывает сигналы *PdOut* и *paid* и в зависимости от них формирует выходной сигнал *open*. Данный процесс имеет три активных состояния: *isClosed*, *isMinimallyOpened* и *isOpened*. В состоянии *isClosed*, если оплата получена, турникет открывается, значение переменной *passed* сбрасывается и процесс переходит в состояние *isMinimallyOpened*. В состоянии *isMinimallyOpened* прохождение пользователя запоминается в переменной *passed*. В этом состоянии установлен таймаут, который определяет минимальное время, в течение которого турникет должен быть открыт. По истечении этого времени, если пользователь прошёл, турникет закрывается и процесс переходит в состояние *isClosed*, в противном случае процесс переходит в состояние *isOpened*. В состоянии *isOpened*, если пользователь прошёл, а также по таймауту, турникет закрывается и процесс переходит в состояние *isClosed*. Сумма временных интервалов в операторах таймаута в состояниях

[1] https://github.com/ivchernenko/requirement-language

isMinimallyOpened и *isOpened* определяет максимальное время, в течение которого турникет может быть открыт.

Процесс *Indicator* обрабатывает сигнал *opened* и в зависимости от него формирует сигналы *enter* и *reset*. Этот процесс имеет два активных состояния: *isClosed* и *isOpened*. В состоянии *isClosed*, если турникет открыт, включается светодиод *enter* и процесс переходит в состояние *isOpened*. В состоянии *isOpened*, если турникет закрыт, выключается светодиод *enter*, подаётся сигнал разблокировки монетоприемника *reset*, запускается процесс *Unlocker* и процесс *Indicator* переходит в состояние *isClosed*.

Процесс *Unlocker* формирует сигнал *reset*. Этот процесс имеет одно активное состояние *unlock*, в котором установлен таймаут. По истечении заданного времени сигнал разблокировки монетоприемника *reset* перестаёт подаваться и процесс останавливается.

Период активации процессов равен 100 миллисекундам.

5.2. Требования на естественном языке

К программе управления турникетом предъявляются следующие требования:

1. После получения сигнала о проходе пользователя *PdOut* турникет должен быть закрыт не позднее, чем через 1 секунду.
2. Если турникет был закрыт и оплата не выполнена, то он не откроется, пока не будет выполнена оплата.
3. После появления от монетоприёмника сигнала получения оплаты *payed* немедленно должен быть сформирован сигнал открытия турникета *open*.
4. Сигнал *open* должен быть в состоянии *true* не более 10 секунд.
5. Сигнал *open* должен быть в состоянии *true* не менее 1 секунды.
6. После запрета прохода должен быть разрешена работа монетоприёмника.

5.3. Шаблоны требований

Мы определили набор типовых производных шаблонов требований на ранее разработанном языке спецификации шаблонов. Библиотека включает 15 шаблонов. Следующие три шаблона из этой библиотеки используются для формализации требований к программе управления турникетом:

1. $P1$: если произошло событие A_1, связывающее значения программных переменных на двух последовательных итерациях, то не позднее, чем через время t, должно произойти событие A_3, и после наступления события A_1, и до наступления события A_3 выполняется условие A_2.
2. $P2$: если произошло событие A_1, связывающее значения переменных на двух последовательных итерациях, то условие A_2 выполняется, пока не будет достигнуто время t.
3. $P3$: если произошло событие A_1, связывающее значения переменных на двух последовательных итерациях, то на второй итерации должно произойти событие A_2.

Требования 1 и 4 к программе управления турникетом удовлетворяют шаблону $P1$. Требование 5 удовлетворяет шаблону $P2$. Требования 2, 3 и 6 удовлетворяют шаблону $P3$.

Рассмотрим определения этих шаблонов на языке спецификации шаблонов.

```
import "Basic_Patterns.rpl";

derivedreq pattern P1(const : t simple formulas : A11, A12 formulas : A2, A3) =
PRP1(formulas : lambda r2 r1.
  PRP3(formulas: lambda r4 r3. ~ A11(r3) final: r2 current: r1) \/ ~ A12(r1)
    \/ FRP1(const : t formulas : A2, A3 final : r2 current : r1)
);

derivedreq pattern P2(const : t simple formulas : A11, A12 formulas : A2) =
PRP1(formulas : lambda r2 r1.
```

```
  PRP3(formulas: lambda r4 r3. ~ A11(r3) final: r2 current: r1) \/ ~ A12(r1)
    \/ FRP2(const : t formulas : A2 final : r2 current : r1)
);

derivedreq pattern P3(simple formulas: A11, A12 formulas: A2) =
PRP1(formulas: lambda r2 r1. PRP3(formulas: lambda r4 r3.
~A11(r3) final: r2 current: r1) \/ ~A12(r1) \/ A2(r2, r1));
```

Файл, содержащий определения шаблонов, импортирует библиотеку базовых шаблонов `Basic_Patterns.rpl`. Эта библиотека содержит объявления базовых шаблонов, определенных в теории Isabelle/HOL `Basic_Patterns`. Файл `Basic_Patterns.rpl` содержит информацию о связи базовых шаблонов требований и шаблонов дополнительных инвариантов и леммах для этих шаблонов, необходимую для порождения теории Isabelle/HOL для производных шаблонов.

Так как событие A_1 в шаблонах связывает значения переменных на двух итерациях, оно описывается двумя формулами $A11$ и $A12$. Так как к fm-параметрам $A11$ и $A12$ применяется отрицание, они объявлены как простые fm-параметры, то есть fm-параметры, значения которых не могут содержать вложенные шаблоны. Другие fm-параметры могут содержать вложенные шаблоны.

В определениях этих шаблонов используются следующие базовые шаблоны требований:

- *FRP*1 задаёт утверждения вида: «Не позднее, чем через время t после начала отсчета времени произойдет событие A_2, и после начала отсчета времени и до наступления события A_2 выполняется условие A_1»;
- *FRP*2 задаёт утверждения вида: «Условие A выполняется, пока не будет достигнуто время t»;
- *PRP*1 задаёт утверждения вида: «Условие A должно выполнять всегда вплоть до текущего состояния»;
- *PRP*3 задаёт утверждения вида: «Если предыдущее внешнее состояние существует, то в нём выполняется условие A».

Определения этих шаблонов можно найти в [11].

С помощью разработанного нами генератора теории Isabelle/HOL для производных шаблонов требований, определенных на языке спецификации шаблонов, была порождена теория Isabelle/HOL, содержащая определения этих шаблонов на языке системы Isabelle/HOL, определения соответствующих шаблонов дополнительных инвариантов, леммы для этих шаблонов и их доказательства. Рассмотрим фрагмент порожденной теории, содержащий шаблон $P1$.

```
theory Derived_Patterns imports Basic_Patterns
begin

definition P1 where "P1 t A11 A12 A2 A3 s ≡
(PRP1 (λ r2 r1 . ((PRP3 (λ r4 r3 . (¬ (A11 r3))) r2 r1) ∨ ((¬ (A12 r1)) ∨
(FRP1 t A2 A3 r2 r1)))) s)"

definition P1_part where "P1_part t A11 A12 A2 A3 s ≡
(P1_2 t A11 A12 (λ s s1 . (A2 s1)) (λ s s1 . (A3 s1)) s)"

definition P1_inv where "P1_inv t b_0 t1_0 A11 A12 A2_1 A3_1 s ≡
((PIP1 (λ r2 r1 . ((PRP3 (λ r4 r3 . (¬ (A11 r3))) r2 r1) ∨ ((¬ (A12 r1)) ∨
(FIP1 t t1_0 A2_1 A3_1 r2 r1)))) s) ∧ (PIP3 b_0 (λ r4 r3 . (¬ (A11 r3))) s))"

definition P1_inv_part where "P1_inv_part t b_0 t1_0 A11 A12 A2 A3 s ≡
(P1_inv t b_0 t1_0 A11 A12 (λ s s1 . (A2 s1)) (λ s s1 . (A3 s1)) s)"

lemma P1_L1: "
P1_inv t b_0 t1_0 A11 A12 A2_1 A3_1 s0 ⟹
consecutive s0 s ⟹
(((True ∧ (True ∧ ((always_imp s0 (A2_1 s0) (A2_1 s))
```

```
∧ ((always_imp s0 (A3_1 s0) (A3_1 s)) ∧ (((t1_0 s0) = 0) ∨ (((A3_1 s s) ∧
((t1_0 s0) <= t)) ∨ ((A2_1 s s) ∧ ((t1_0 s0) < (t1_0 s)))))))) ∧ (((b_0 s0) ∧ True)
∨ ((¬ (A12 s)) ∨ ((A3_1 s s) ∨ ((A2_1 s s) ∧ ((t1_0 s) > 0)))))) ∧ ((b_0 s) ⟶
(¬ (A11 s)))) ⟹
P1_inv t b_0 t1_0 A11 A12 A2_1 A3_1 s"

lemma P1_L2: "
P1_inv t b_0 t1_0 A11 A12 A2_1 A3_1 s0 ⟹
e s0 ⟹
(True ∧ (True ∧ ((always_imp s0 (A2_1 s0) (A2 s0))
∧ ((always_imp s0 (A3_1 s0) (A3 s0)) ∧ ((t1_0 s0) <= t))))) ⟹
P1 t A11 A12 A2 A3 s0"

lemma P1_part_L1: "
P1_inv_part t b_0 t1_0 A11 A12 A2 A3 s0 ⟹
consecutive s0 s ⟹
(((((t1_0 s0) = 0) ∨ (((A3 s) ∧ ((t1_0 s0) <= t)) ∨ ((A2 s) ∧ ((t1_0 s0) <
(t1_0 s))))) ∧ ((b_0 s0) ∨ ((¬ (A12 s)) ∨ ((A3 s) ∨ ((A2 s) ∧ ((t1_0 s) > 0)))))) ∧
((b_0 s) ⟶ (¬ (A11 s)))) ⟹
P1_inv_part t b_0 t1_0 A11 A12 A2 A3 s"
unfolding P1_2_inv_part_def P1_2_part_def
apply(simp add: P1_L1 L)
done

lemma P1_part_L2: "
P1_inv_part t b_0 t1_0 A11 A12 A2 A3 s ⟹
e s ⟹
((t1_0 s) <= t) ⟹
P1_part t A11 A12 A2 A3 s"
unfolding P1_2_inv_part_def P1_2_part_def
apply(simp add: P1_L2 L)
done

lemmas P1_used_patterns = P1_def PRP1_def PRP3_def FRP1_def

lemmas P1_inv_used_patterns = P1_inv_def PIP1_def PRP3_def FIP1_def PIP3_def

lemmas P1_inv_part_used_patterns = P1_inv_part_def P1_inv_used_patterns

lemmas P1_part_used_patterns = P1_part_def P1_used_patterns
```

Порождённая теория `Derived_Patterns` импортирует теорию `Basic_Patterns`. Для производного шаблона требований $P1$ были порождены следующие элементы:

- общий шаблон требований $P1$, который может использоваться для определения новых производных шаблонов на основе шаблона $P1$;
- общий шаблон дополнительных инвариантов $P1\_inv$, связанный с шаблоном $P1$. Этот шаблон может использоваться для определения новых шаблонов дополнительных инвариантов;
- частный шаблон требований *P1_part*.Этот шаблон определяется на основе общего шаблона $P1$ и отличается от него тем, что все его fm-параметры являются простыми, то есть их значения не могут содержать вложенные шаблоны. Этот шаблон используется для задания требований;
- частный шаблон дополнительных инвариантов $P1\_inv\_part$. Этот шаблон используется для задания дополнительных инвариантов;
- лемму $P1\_L1$, которая связана с общим шаблоном $P1\_inv$ и удовлетворяет схеме LS_8. Эта лемма используется для доказательства лемм, связанных с шаблонами, определяемыми на основе шаблона $P1\_inv$, и удовлетворяющих схеме LS_8;
- лемму $P1\_L2$, которая связана с общими шаблонами и удовлетворяет схеме LS_9;

- лемму $P1\_part\_L1$, которая связана с частным шаблоном $P1\_inv\_part$ и удовлетворяет схеме LS_8. Эта лемма используется для доказательства условий корректности;
- лемму $P1\_part\_L2$, которая связана с частными шаблонами и удовлетворяет схеме LS_9;
- списки $P1\_used\_patterns$, $P1\_inv\_used\_patterns$, $P1\_part\_used\_patterns$ и $P1\_inv\_part\_used\_patterns$ определений шаблонов, используемых соответственно в определениях шаблонов $P1$, $P1\_inv$, $P1\_part$ и $P1\_inv\_part$. Они используются при доказательстве условий корректности, соответствующих инициализации программы, так как при доказательстве этих условий корректности требуется раскрытие всех шаблонов.

Порождённые шаблоны дополнительных инвариантов имеют два fn-параметра: b_0 и $t1\_0$, определяющих соответственно условие, при котором не выполняется формула $A11$, и максимальное время ожидания события $A3$. Здесь не приводятся доказательства лемм для общих шаблонов, порождаемые разработанным генератором. Эти доказательства могут быть найдены на GitHub. Доказательства лемм для частных шаблонов выполняются путём раскрытия определений частных шаблонов и упрощения полученного утверждения с применением соответствующей леммы для общих шаблонов и леммы L из раздела 2.

5.4. Формализация требований и доказательство условий корректности

Для выполнения дедуктивной верификации poST-программы выполняются следующие действия:

1. Порождение теории Isabelle/HOL с именами переменных, процессов и состояний процессов, программными константами и условиями корректности.
2. Формализация требований.
3. Определение независимого от требований дополнительного инварианта.
4. Определение зависящих от требований дополнительных инвариантов.
5. Порождение скриптов доказательства условий корректности.
6. Проверка порожденных доказательств в Isabelle/HOL.

Опишем процесс верификации программы управления турникетом по шагам.

На первом шаге с помощью генератора условий корректности порождаются теория Isabelle/HOL `Turnstile`, содержащая имена переменных, процессов и состояний процессов, а также программные константы, и теория `Turnstile_VC_1_1009`, содержащая условия корректности. Для программы управления турникетом порождается 1009 условий корректности, и они сохраняются в одной теории Isabelle/HOL. Вывод генератора условий корректности имеет следующий вид:

```
v_PdOut'value v_paid'value v_opened'value
Turnstile_VC_1_1009 1 1009
```

В первой строке содержатся имена параметров условий корректности, соответствующих входным переменным программы *PdOut*, *paid* и *opened*. Вторая строка содержит имя порожденной теории с условиями корректности и диапазон номеров условий корректности, содержащихся в этой теории. Вывод генератора условий корректности подаётся на вход генератору скриптов доказательства условий корректности на шаге 5.

На втором шаге выполняется формализация требований. На этом шаге для каждого требования пользователь выбирает подходящий производный шаблон требований и задаёт значения его параметров. Требования задаются на языке системы Isabelle/HOL. Для программы управления турникетом требования определяются следующим образом:

```
definition R1 where "R1 s  ≡
let t=9 in let A11 = (λ s1. s1[open] = True)
in let A12 = (λ s2. s2[PdOut] = True) in let A2 = (λ s3. s3[open] = False)
let A3 = (λ s4. s4[open] = False)
P1_part t A11 A12 A2 A3 s"
```

```
definition R2 where "R2 s ≡
let A11 = (λ s1. s1[open] = False) in let A12 = (λ s2. s2[paid] = False)
in let A2 = (λ s2. s2[open] = False) in P3_part A11 A12 A2 s"

definition R3 where "R3 s ≡
let A11 = (λ s1. s1[open] = False ∧ s1[paid] = False) in let A12 =
(λ s2. s2[paid] = True) in let A2 = (λ s2. s2[open] = True) in P3_part A11 A12 A2 s"

definition R4 where "R4 s ≡
let t = 100 in let A11 = (λ s1. s1[open] = False) in let A12 = (λ s2.
s2[open] = True) in let A2 = (λ s3. s3[open] = True) in let (λ s4. s4[open] = False)
in P1_part t A11 xA12 A2 A3 s"

definition R5 where "R5 s ≡
let t = 10 in let A11 = (λ s1. s1[open] = False) in let A12 = (λ s2. s2[open] = True)
let A2 = (λ s3. s3[open] = True)
P2_part t A11 A12 A2 s"

definition R6 where "R6 s ≡
let A11 = (λ s1. s1[enter] = True) in let A12 = (λ s2. s2[enter] = False)
in let A2 = (λ s2. s2[reset] = True) in P3_part A11 A12 A2 s"
```

На третьем шаге пользователь определяет независимый от требований дополнительный инвариант в Isabelle/HOL в теории с именем CommonExtraInv. Определение дополнительного инварианта имеет вид:

```
definition commonExtraInv where "commonExtraInv s ≡ e s ∧
(getPstate s Controller ∈ {Controller'isClosed, STOP} ⟶ s[open] = False) ∧
(getPstate s Controller ∈ {Controller'isMinimallyOpened, Controller'isOpened} ⟶
s[open] = True) ∧
(getPstate s Controller = Controller'isMinimallyOpened ⟶ ltime s Controller ≤ 10) ∧
(getPstate s Controller = Controller'isOpened ⟶ ltime s Controller ≤ 90) ∧
getPstate s Controller ∈ {Controller'isClosed, Controller'isMinimallyOpened,
Controller'isOpened, STOP} ∧
(getPstate s Indicator = Indicator'isOpened ⟶ s[enter] = True) ∧
(getPstate s Indicator ≠ Indicator'isOpened ⟶ s[enter] = False) ∧
getPstate s Indicator ∈ {Indicator'isClosed, Indicator'isOpened, STOP} ∧
(getPstate s Unlocker = Unlocker'unlock ⟶ s[reset] = True) ∧
(getPstate s Unlocker = STOP ⟶ s[reset] = False) ∧
(getPstate s Unlocker = Unlocker'unlock ⟶ ltime s Unlocker ≤ 10) ∧
getPstate s Unlocker ∈ {Unlocker'unlock, STOP} ∧
(getPstate s Init = Init'init ⟶ getPstate s Controller = STOP ∧
getPstate s Indicator = STOP) ∧
(getPstate s Init = STOP ⟶ getPstate s Controller ≠ STOP ∧ getPstate s
Indicator ≠ STOP) ∧
(getPstate s Init ∈ {Init'init, STOP})
```

Данный дополнительный инвариант является конъюнкцией следующих свойств:

1. Если процесс *Controller* находится в состоянии *isClosed* или *STOP*, то турникет закрыт.
2. Если процесс *Controller* находится в состоянии *isMinimallyOpened* или *isOpened*, то турникет открыт.
3. Если процесс *Controller* находится в состоянии *isMinimallyOpened*, то значение его таймера не превышает значение, заданное в операторе таймаута в этом состоянии равное 10 итерациям цикла управления (1 секунде).

4. Если процесс *Controller* находится в состоянии *isOpened*, то значение его таймера не превосходит значение, заданное в операторе таймаута в этом состоянии, равное 90 итерациям цикла управления (9 секундам).
5. Процесс *Controller* может находиться только в одном из следующих состояний: *isClosed*, *isMinimallyOpened*, *isOpened* и *STOP*.
6. Если процесс *Indicator* находится в состоянии *isOpened*, то горит светодиод.
7. Если процесс *Indicator* не находится в состоянии *isOpened*, то светодиод не горит.
8. Процесс *Indicator* может находиться только в одном из следующих состояний: *isClosed*, *isOpened* и *STOP*.
9. Если процесс *Unlocker* находится в состоянии *unlock*, то подаётся сигнал *reset*.
10. Если процесс *Unlocker* находится в состоянии *STOP*, то сигнал *reset* не подаётся.
11. Если процесс *Unlocker* находится в состоянии *unlock*, то значение его таймера не превышает значение, указанное в операторе таймаута в этом состоянии, равное 10 итерациям цикла управления (1 секунде).
12. Процесс *Unlocker* может находиться только в состояниях *unloack* и *STOP*.
13. Если процесс *Init* находится в состоянии *init*, то процессы *Controller* и *Indicator* находятся в состоянии *STOP*.
14. Если процесс *Init* находится в состоянии *STOP*, то процессы *Controller* и *Indicator* не находятся в состоянии *STOP*.
15. Процесс *Init* может находиться только в состояниях *init* и *STOP*.

На четвёртом шаге пользователь задаёт зависящие от требований дополнительные инварианты в Isabelle/HOL. На этом шаге пользователь также описывает соответствие требований шаблонам и связи между требованиями и зависящими от требований дополнительными инвариантами. Эта информация используется генератором скриптов доказательства условий корректности. Дополнительные инварианты для требований имеют следующий вид:

```
definition Einv1 where "Einv1 s  ≡
commonExtraInv s ∧
(let t=9 in
let b_0 = (λ s. getPstate s Controller' ∈ {Controller'isClosed', STOP}) in
let t1_0 = (λ s. if getPstate s Controller = Controller'isMinimallyOpened ∧
s[passed] = True then ltime s Controller - 1 else 0) in
let A11 = (λ s1. s1[open] = True) in
let A12 = (λ s2. s2[PdOut]) in
let A2 = (λ s3. s3[open] = True) in
let A3 = (λ s4. s4[open] = False) in
P1_inv_part t b_0 t1_0 A11 A12 A2 A3 s)"

...

definition Einv6 where "Einv6 s ≡
...
```

Здесь дополнительный инвариант *Einv1* определяется для требования *R*1, *Einv2* — для требования *R*2 и т. д. Каждый дополнительный инвариант представляет собой конъюнкцию независимого от требований дополнительного инварианта и экземпляра шаблона зависящих от требований дополнительных инвариантов. Значения константных и формульных параметров в дополнительном инварианте совпадают со значениями соответствующих параметров в требовании. В дополнительном инварианте *Einv1* значение fn-параметра *b*_0 определяет, что, когда процесс *Controller* находится в состоянии *isClosed* или *STOP*, то не выполняется формула *A*11, то есть турникет закрыт, а значение fn-параметра *t*1_0 определяет, что, когда процесс *Controller* находится в состоянии *isMinimallyOpened* и значение переменной *passed* равно *True*, то есть пользователь прошёл после открытия турни-

кета, то максимальное время ожидания закрытия турникета равно $ltime(s, Controller)$, где s — текущее состояние изменений, иначе закрытие турникета не ожидается, так как турникет закрыт или пользователь не прошёл. В дополнительных инвариантах *Einv2*, *Einv3*, *Einv4* и *Einv5* значение параметра b_0 определяет, что, когда процесс *Controller* находится в состоянии *isMinimallyOpened* или *isOpend*, то не выполняется формула $A11$ так как турникет открыт. Значение параметра $t1\_0$ в дополнительном инварианте *Einv4* определяет, что, если процесс *Controller* находится в состоянии *isMinimallyOpened*, то максимальное время ожидания закрытия турникета равно $ltime(s, Control)$, если процесс *Controller* находится в состоянии *isOpened*, то максимальное время ожидания закрытия турникета равно $ltime(s, Controller) + 10$, иначе турникет закрыт. Значение fn-параметра $t1\_0$ в дополнительном инварианте *Einv5* определяет, что, если процесс *Controller* находится в состоянии *isMinimallyOpened*, то после открытия турникета прошло время, не меньшее $ltime(s, Controller) - 1$, иначе, если турникет открывался, то после его последнего открытия прошло время по крайней мере 9 итераций цикла управления (на одну итерацию меньше, чем время, указанное в операторе таймаута в состоянии *isMinimallyOpened*). Значение параметра b_0 в дополнительном инварианте *Einv6* определяет, что, если процесс *Indicator* не находится в состоянии *isOpened*, то светодиод не горит.

Затем пользователь определяет соответствие требований шаблонам и дополнительным инвариантам в файле, имя которого совпадает с именем теории Isabelle/HOL, содержащей определения требований и дополнительных инвариантов, но имеющем расширение «.rpl». Соответствие требований шаблонам и дополнительным инвариантам описывается следующим образом:

```
import "../Basic_Patterns.rpl";
import "../Derived_Patterns.rpl";

requirement R1 : P1_part with Einv1;
requirement R2: P3_part with Einv2;
requirement R3: P3_part with Einv3;
requirement R4: P1_part with Einv4;
requirement R5: P2_part with Einv5;
requirement R6 : P3_part with Einv6;
```

На пятом шаге порождаются скрипты доказательства условий корректности. Для этого запускается генератор скриптов доказательства следующей командой:

```
java -jar proofScriptGenerator.jar -s Invariants.rpl -cei commonExtraInv,
```

где proofScriptGenerator.jar — JAR-файл с генератором скриптов доказательства, Invariants.rpl — файл, в котором описано соответствие требований шаблонам и дополнительным инвариантам, опция -s используется для указания того, что необходимо породить скрипты доказательства, опция -cei используется для указания имени независимого от требований дополнительного инварианта. Генератор порождает следующие теории Isabelle/HOL:

- `CommonExtraInv_Turnstile_VC_1_1009` с доказательствами условий корректности для независимого от требований дополнительного инварианта;
- теории `Einvi_Turnstile_VC_1_1009` с доказательствами условий корректности для дополнительных инвариантов *Einvi* соответственно ($1 \leqslant i \leqslant 6$);
- теории `Ri_Turnstile_VC_1_1009` с определениями расширенных инвариантов, представляющих собой конъюнкции требований и соответствующих дополнительных инвариантов, и доказательствами условий корректности для расширенных инвариантов ($1 \leqslant i \leqslant 6$). Например, расширенный инвариант для первого требования в Isabelle/HOL имеет следующий вид:

```
definition R1_extended_inv where "R1_extended_inv s ≡ Einv1 s ∧ R1 s"
```

На шестом шаге выполняется проверка порождённых доказательств в Isabelle/HOL.

6. Обзор связанных работ

Многие инструменты дедуктивной верификации, такие как Frama-C [16] и VCC [17] для языка C, Spec# [18] для языка C#, SPARK 2014 [19] для языка Ada, верификатор Dafny-программ [20] используют инструменты дедуктивной верификации Why3 [21] и Boogie [22], предоставляющие промежуточные языки верификации. Такие инструменты имеют следующую архитектуру. Инструмент верификации программ на некотором языке программирования транслируется на промежуточный язык верификации, предоставляемый системой Why3 или Boogie, которая порождает условия корректности и использует внешние инструменты доказательства теорем. Boogie использует SMT-решатели для доказательства условий корректности. Why3 помимо SMT-решателей также использует интерактивные инструменты доказательства теорем. В данной работе мы разрабатываем инструменты дедуктивной верификации для языка poST. Мы не используем промежуточные языки верификации, а определяем аксиоматическую семантику для языка poST, основанную на состояниях изменений. Для доказательства условий корректности используется система интерактивного доказательства Isabelle/HOL. В отличие от вышеперечисленных инструментов, наша система включает развитые средства автоматизации доказательства условий корректности, в частности, генератор скриптов доказательства условий корректности.

При автоматизированном доказательстве теорем, в частности, в дедуктивной верификации, часто возникает необходимость использовать леммы. Для порождения лемм существуют различные методы [23], как нисходящие методы, основанные на обобщении текущего доказываемого утверждения, так и восходящие методы, основанные на исследовании теории (theory exploration), которые строят леммы, выражающие свойства имеющихся функций и типов данных. Методы обобщения порождают возможные леммы путём замены некоторых термов новыми переменными [24]. Восходящие методы порождают термы или равенства согласно определенным эвристикам и используют различные эвристики для оценки того, является ли найденная лемма полезной. Методы генерации лемм могут использовать проверку наличия контрпримеров для порожденных лемм для исключения ложных лемм. Примером инструмента, использующего нисходящий подход для генерации лемм, является PGT [25]. Для генерации предположений PGT выполняет обобщение текущей цели путем замены констант и повторяющихся подтермов переменными, после чего генерирует леммы путем замены подтермов в исходной и обобщенной целях термами, созданными в соответствие с набором эвристик. Примерами инструментов генерации лемм восходящим методом в системе Isabelle/HOL являются IsaScheme [26] и TBC [27]. IsaScheme порождает леммы на основе предоставляемых пользователем схем, представляющих собой формулы высшего порядка определенного вида. Порождение лемм сводится к подстановке функций из заданного пользователем набора вместо параметров схем. В TBC используются шаблоны лемм, выражающие известные алгебраические свойства, такие, как коммутативность, и свойства отношений, такие, как транзитивность. Еще одним инструментом является Hipster [28], который порождает леммы, имеющие вид равенств. Для этого из переменных и имеющихся в теории констант и функций порождаются термы до заданного размера, выполняется разбиение множества термов на классы эквивалентности путем рандомизированного тестирования, после чего леммы определяются путем приравнивания термов из одного класса эквивалентности. Леммы, которые могут быть легко доказаны, например, не требующие ииндукции, удаляются.

Рассмотренные выше работы посвящены порождению лемм, требующих доказательства по индукции. В нашей работе мы порождаем леммы, выражающие свойства темпоральных требований и используемые для доказательства условий корректности и других лемм. Использование этих лемм позволяет свести доказательство условий корректности с инвариантами, выражающие темпоральные свойства программы к бесквантворным формулам, не содержащим шаблонов и связывающим значения переменных, состояния процессов и значения таймеров в одной точке программы.

Такие формулы доказываются с помощью инструментов автоматического доказательства. Подобно [26] и [27], в нашей работе леммы порождаются в соответствии с определёнными схемами. Порождение лемм выполняется до начала доказательства утверждения, требующего использование леммы. Но в отличие от подхода, используемого в работе [27], значениями параметров схем являются не функции, а шаблоны и формулы, являющиеся посылками лемм и выражающими ограничения на значения переменных, состояния процессов и значения таймеров, при которых сохраняется дополнительный инвариант или из дополнительного инварианта следует требование. Подобно нисходящим методам, метод, используемый в данной работе, порождает леммы, применяемые для доказательства утверждений определенного вида: требований и дополнительных инвариантов, соответствующих определённому шаблону. Обобщение выполняется путем введения функциональных параметров в шаблон дополнительных инвариантов. Производный шаблон дополнительных инвариантов является обобщением соответствующего шаблона требований.

Заключение

В этой статье был разработан алгоритм порождения скриптов доказательства лемм, определяемых для шаблонов требований и дополнительных инвариантов и применяемых для доказательства условий корректности. Также были откорректированы схемы шаблонов дополнительных инвариантов и лемм. Разработанный алгоритм был реализован в генераторе теории Isabelle/HOL. Этот генератор принимает на вход спецификацию производных шаблонов требований на разработанном нами языке спецификации шаблонов и порождает теорию Isabelle/HOL, содержащую определения этих шаблонов на языке системы Isabelle/HOL, а также порождённые шаблоны дополнительных инвариантов и леммы с их доказательствами. Порождённые шаблоны могут использоваться как для задания требований и дополнительных инвариантов, так и для определения новых шаблонов на их основе. Генератор также порождает файл, содержащий объявления порожденных шаблонов на языке спецификации шаблонов. Это позволяет ссылаться на порождённые шаблоны при определении новых шаблонов, требований и дополнительных инвариантов в языке спецификации шаблонов.

Мы также разработали генератор скриптов доказательства условий корректности. Этот генератор принимает на вход файл с описанием соответствия требований шаблонам и связи требований с дополнительными инвариантами и имя независимого от требований дополнительного инварианта через параметры командной строки, а также список параметров условий корректности и информацию о теориях Isabelle/HOL, содержащих условия корректности. Выходом генератора скриптов доказательства являются теории Isabelle/HOL с доказательствами условий корректности для дополнительных инвариантов и расширенных инвариантов, представляющих собой конъюнкции требований и соответствующих дополнительных инвариантов. Необходимость предоставления информации об условиях корректности потребовала изменения формата вывода генератора условий корректности. Была создана новая версия генератора условий корректности, в которой реализован требуемый формат вывода.

Разработанные генераторы теории Isabelle/HOL с шаблонами и лемма и скриптов доказательства условий корректности позволяют автоматизировать доказательство условий корректности позволяет автоматизировать доказательство условий корректности poST-программ для широкого класса требований. Но в настоящее время для выполнения дедуктивной верификации пользователь должен определить дополнительные инварианты. Кроме того, текущая версия генератора условий корректности не поддерживает некоторые конструкции языка poST.

В дальнейшем мы планируем разработать алгоритмы порождения дополнительных инвариантов на основе шаблонов, а также разработать новую версию генератора условий корректности, поддерживающего другие конструкции языка poST.

References

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

[2] F. Basile, P. Chiacchio, and D. Gerbasio, "On the implementation of industrial automation systems based on PLC", *IEEE Transactions on Automation Science and Engineering*, vol. 10, no. 4, pp. 990–1003, 2012.

[3] V. E. Zyubin, "Hyper-automaton: A model of control algorithms", in *Proceedings of the 2007 Siberian Conference on Control and Communications*, IEEE, 2007, pp. 51–57, ISBN: 1-4244-0346-4. 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] P. Ovsiannikova, I. Buzhinsky, A. Pakonen, and V. Vyatkin, "Oeritte: User-friendly counterexample explanation for model checking", *IEEE Access*, vol. 9, pp. 61 383–61 397, 2021.

[6] D. Gurov, P. Herber, and I. Schaefer, "Automated verification of embedded control software: Track introduction", in *Proceedings of the International Symposium on Leveraging Applications of Formal Methods*, 2020, pp. 235–239.

[7] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, "Model checking and the state explosion problem", in *LASER Summer School on Software Engineering*, Springer, 2011, pp. 1–30.

[8] I. Anureev, N. Garanina, T. Liakh, A. Rozov, V. Zyubin, and S. Gorlatch, "Two-step deductive verification of control software using Reflex", in *Proceedings of the International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, 2019, pp. 50–63.

[9] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver", in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 337–340.

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

[11] I. M. Chernenko and I. S. Anureev, "Pattern-based approach to automation of deductive verification of process-oriented programs: Patterns, lemmas and algorithms", *Modeling and Analysis of Information Systems*, vol. 31, no. 4, pp. 384–425, 2024, in Russian. DOI: 10.18255/1818-1015-2024-4-384-425

[12] I. Chernenko and I. Anureev, "Generation of Isabelle/HOL theory focused on proving verification conditions of PoST programs and based on derived requirement patterns", in *Proceedings of the 26th International Conference of Young Professionals in Electron Devices and Materials (EDM)*, 2025, pp. 1480–1485.

[13] I. Chernenko, I. S. Anureev, N. O. Garanina, and S. M. Staroletov, "A temporal requirements language for deductive verification of process-oriented programs", in *Proceedings of the 23rd International Conference of Young Professionals in Electron Devices and Materials (EDM)*, 2022, pp. 657–662.

[14] L. C. Paulson, "The foundation of a generic theorem prover", *Journal of Automated Reasoning*, vol. 5, pp. 363–397, 1989.

[15] L. Bettini, *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016, ISBN: 978-1786464965.

[16] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C: A software analysis perspective", *Formal Aspects of Computing*, vol. 27, no. 3, pp. 573–609, 2015.

[17] E. Cohen et al., "VCC: A practical system for verifying concurrent C", in *Proceedings of the Theorem Proving in Higher Order Logics: 22nd International Conference*, 2009, pp. 23–42.

[18] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter, "Specification and verification: The Spec# experience", *Communications of the ACM*, vol. 54, no. 6, pp. 81–91, 2011.

[19] N. Kosmatov, C. Marché, Y. Moy, and J. Signoles, "Static versus dynamic verification in Why3, Frama-C and SPARK 2014", in *Proceedings of the International Symposium on Leveraging Applications of Formal Methods*, 2016, pp. 461–478.

[20] K. R. M. Leino and V. Wüstholz, "The dafny integrated development environment", in *Proceedings of the 1st Workshop on Formal Integrated Development Environment*, 2014, pp. 3–15. DOI: 10.4204/EPTCS.149.2

[21] J.-C. Filliâtre and A. Paskevich, "Why3 — where programs meet provers", in *European Symposium on Programming*, 2013, pp. 125–128.

[22] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs", in *Proceedings of the International Symposium on Formal Methods for Components and Objects*, Springer, 2005, pp. 364–387.

[23] M. Johansson, "Lemma discovery for induction: A survey", in *Proceedings of the Intelligent Computer Mathematics: 12th International Conference*, Springer, 2019, pp. 125–139.

[24] M. Aderhold, "Improvements in formula generalization", in *Proceedings of the International Conference on Automated Deduction*, 2007, pp. 231–246.

[25] Y. Nagashima and J. Parsert, "Goal-oriented conjecturing for Isabelle/HOL", in *Proceedings of the Intelligent Computer Mathematics: 11th International Conference*, 2018, pp. 225–231.

[26] O. Montano-Rivas, R. McCasland, L. Dixon, and A. Bundy, "Scheme-based synthesis of inductive theories", in *Proceedings of the Mexican International Conference on Artificial Intelligence*, 2010, pp. 348–361.

[27] Y. Nagashima, Z. Xu, N. Wang, D. S. Goc, and J. Bang, "Template-based conjecturing for automated induction in Isabelle/HOL", in *Proceedings of the International Conference on Fundamentals of Software Engineering*, 2023, pp. 112–125.

[28] M. Johansson, D. Rosén, N. Smallbone, and K. Claessen, "Hipster: Integrating theory exploration in a proof assistant", in *Proceedings of the International Conference on Intelligent Computer Mathematics*, 2014, pp. 108–122.