МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет информационных технологий
Кафедра компьютерных технологий

Направление подготовки 09.04.01 Информатика и вычислительная техника
Направленность (профиль): Технология разработки программных систем

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА МАГИСТРА

Черненко Ивана Михайловича

Тема работы:

РАЗРАБОТКА ГЕНЕРАТОРА УСЛОВИЙ КОРРЕКТНОСТИ POST-ПРОГРАММ И СТРАТЕГИЙ ИХ ДОКАЗАТЕЛЬСТВА В СИСТЕМЕ ISABELLE/HOL

«К защите допущена»
Заведующий кафедрой,
д.т.н., доцент Каф. КТ ФИТ НГУ
Зюбин В. Е. /…………..
(ФИО) / (подпись)
«……»………………20…г.

Руководитель ВКР
к.ф.-м.н.,
доцент каф. СИ ФИТ НГУ
Ануреев И. С. /………...
(ФИО) / (подпись)
«……»………………20…г.

Новосибирск, 2023

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)
Факультет информационных технологий

Кафедра компьютерных технологий
(название кафедры)

Направление подготовки: 09.04.01 ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА
Направленность (профиль): Технология разработки программных систем

УТВЕРЖДАЮ

Зав. кафедрой…Зюбин В. Е.
(фамилия, И., О.)

…………………
(подпись)
«03» марта 2023г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ МАГИСТРА

Студент*у* Черненко Ивану Михайловичу, группы 21224
(фамилия, имя, отчество, номер группы)

Тема Разработка генератора условий корректности poST-программ и стратегий их доказательства в системе Isabelle/HOL
(полное название темы выпускной квалификационной работы магистра)

утверждена распоряжением проректора по учебной работе от 29 октября 2021г. №0296, скорректирована распоряжением проректора по учебной работе от 3 марта 2023 г. № 0037
Срок сдачи студентом готовой работы 20 мая 2023 г.
Исходные данные (или цель работы) Разработать генератор условий корректности poST-программ и стратегии их доказательства в системе Isabelle/HOL
Структурные части работы анализ предметной области, разработка и реализация генератора условий корректности, апробация генератора условий корректности и доказательство условий корректности
Консультанты по разделам ВКР (при необходимости, с указанием разделов)
……………………………………………………………………………………
(раздел, ФИО)

Руководитель ВКР
Доцент каф. СИ ФИТ НГУ,
к.ф.-м.н.
Ануреев И. С. /…………..
(ФИ О) / (подпись)

«03» марта 2023г.

Задание принял к исполнению
Черненко И. М. /…………..
(ФИО студента) / (подпись)

«03» марта 2023г.

СОДЕРЖАНИЕ

Определения, обозначения и сокращения **4**

Введение **5**

1 Анализ предметной области 7

1.1 Процесс-ориентированное программирование и язык poST 7

1.2 Дедуктивная верификация 8

1.3 Isabelle/HOL 13

1.4 Требования к генератору условий корректности 17

2 Разработка и реализация генератора условий корректности 19

2.1 Язык аннотаций для процесс-ориентированных программ 19

2.2 Алгоритм генерации условий корректности 20

2.3 Реализация генератора условий корректности 29

3 Аппробация генератора условий корректности и доказательство условий корректности 34

3.1 Тестовый набор управляющих программ 34

3.2 Классификация требований 36

3.3 Дополнительный инвариант 39

3.4 Доказательство условий корректности 44

Заключение **53**

Список использованных источников и литературы **54**

Приложение А **57**

Приложение Б **65**

Приложение В **84**

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

ПЛК — Программируемый логический контроллер.

Стандарт IEC — Стандарт IEC 61131-3.

AST — Abstract Syntax Tree, абстрактное синтаксическое дерево.

ATP — Automated Theorem Prover, инструмент доказательства теорем в некоторой логике.

SMT — Satisfiability Modulo Theories, выполнимость с учетом теорий. SMT-решатель — инструмент, проверяющий выполнимость формул для теорий первого порядка.

ВВЕДЕНИЕ

Системы промышленной автоматизации в настоящее время обычно основаны на программируемых логических контроллерах (ПЛК). Для разработки управляющего программного обеспечения для таких систем широко используются языки семейства IEC 61131-3. Однако, так как сложность управляющих программ возрастает, языки IEC 61131-3 не всегда удовлетворяют современным требованиям. Это привело к попыткам создания новых подходов к программированию систем управления.

Перспективным подходом является процесс-ориентированное программирование. Программа в этой парадигме представляется множеством взаимодействующих процессов, которые могут запускать, останавливать и контролировать состояние друг друга. В Институте автоматики и электрометрии СО РАН создан язык процесс-ориентированного программирования poST, который является процесс-ориентированным расширением языка Structured Text (ST) из семейства IEC 61131-3.

Во многих системах управления к программному обеспечению предъявляются повышенные требования к надежности. Следовательно, такое программное обеспечение требует верификации, а именно применения формальных методов доказательства корректности программ. Одним из методов формальной верификации является дедуктивная верификация. В этом методе требования корректности записываются в виде аннотаций (выражений некоторого формального языка), которые добавляются к программам, и для полученных аннотированных программ порождаются условия корректности — логические формулы. Метод дедуктивной верификации предполагает, что если все условия корректности истинны, то программа считается корректной.

При верификации программ большого объема возникают трудности, связанные с тем, что условия корректности являются большими формулами. Поэтому процесс верификации желательно автоматизировать. Генерация условий корректности является наиболее простым этапом дедуктивной верификации и может быть автоматизирована. Генераторы условий корректности (инструменты, выполняющие автоматическую генерацию условий корректности) применяются для различных языков программирования, но для языка poST такой инструмент еще не был разработан. Следовательно, актуальной задачей является разработка генератора условий корректности для программ на языке poST.

К программам управления часто предъявляются темпоральные требования (требования, связанные с поведением системы в течение времени). Поэтому актуальной задачей является выражение таких требований в многосортной логике первого порядка, обычно используемой в дедуктивной верификации программ для задания аннотаций.

Также, актуальной задачей является разработка стратегий доказательства условий корректности в системе машинной поддержки доказательства. Это позволяет обеспечить дальнейшую автоматизацию процесса дежуктивной верификации. Одной из таких систем является Isabelle/HOL.

Целью работы является разработка генератора условий корректности poST-программ и стратегий их доказательства в системе Isabelle/HOL.

Для достижения данной цели были поставлены следующие задачи:

1) анализ языка poST и возможностей системы Isabelle/HOL в контексте дедуктивной верификации программ;
2) определение требований к генератору условий корректности poST-программ;
3) разработка алгоритма генерации условий корректности poST-программ;
4) реализация генератора условий корректности;
5) апробация разработанного генератора на тестовом наборе управляющих программ;
6) разработка стратегий доказательства условий корректности в системе Isabelle/HOL;
7) доказательство условий корректности для тестового набора аннотированных управляющих программ.

Новизна данной работы состоит в разработке генератора условий корректности для программ на языке poST и стратегий их доказательства в системе Isabelle/HOL. Разработанный инструмент генерации условий корректности и стратегии их доказательства позволяют повысить качество процесс-ориентированных программ.

Работа состоит из трех глав. В первой главе вводится понятие процесс-ориентированного программирования, определяется синтаксис и семантика процесс-ориентированного языка poST, даются основы дедуктивной верификации и рассматриваются возможности системы Isabelle/HOL. Во второй главе вводится язык аннотаций poST-программ, описывается алгоритм генерации условий корректности poST-программ и его реализация. В третьей главе представлен тестовый набор управляющих программ на языке poST и требования к ним, классы требований, задаваемые шаблонами в виде формул первого порядка и стратегии доказательства условий корректности.

1 Анализ предметной области

1.1 Процесс-ориентированное программирование и язык poST

Системы промышленной автоматизации в настоящее время обычно основаны на программируемых логических контроллерах (ПЛК). Вся логика управления в таких системах задается в программном обеспечении ПЛК. Для разработки управляющего программного обеспечения широко используются языки семейства IEC 61131-3. Однако при возрастании сложности управляющих программ возрастает, языки IEC 61131-3 перестали удовлетворять современным требованиям к разработке управляющего программного обеспечения. Это привело к попыткам создания новых подходов к программированию систем управления.

Перспективным подходом является процесс-ориентированное программирование. Данная парадигма основана на модели гиперавтомата [29], в которой программа определяется как множество взаимодействующих процессов. Каждый процесс представляется конечным автоматом с набором состояний. Состояния процессов определяются как последовательности действий, включая действия по изменению состояний других процессов и действия с таймаутами. В любой момент времени процесс находится в одном из своих состояний, называемом текущим, и выполняет действия, определенные для этого состояния. Каждый процесс может иметь два неактивных состояния: $STOP$ и $ERROR$, в которых не выполняются никакие действия. Состояние $STOP$ означает, что процесс был остановлен в результате нормального выполнения. Состояние $ERROR$ означает, что процесс перешел в это состояние в результате ошибки.

Таким образом, процесс-ориентированная программа представляется как последовательность взаимодействующих процессов.

Язык poST [30] является процесс-ориентированным расширением языка ST (Structured Text) из семейства языков IEC 61131-3, широко применяемых при разработке программного обеспечения для систем управления на основе ПЛК. PoST позволяет описывать систему управления как множество взаимодействующих процессов, сохраняя привычный синтаксис ST.

PoST предполагает циклическое исполнение программы с фиксированным периодом активации процессов. На каждой итерации цикла управления последовательно выполняются процессы в порядке их определения в тексте программы.

Арифметические и логические выражения, условный оператор, операторы выбора и циклов языка poST не отличаются от соответствующих конструкций языка ST. Далее будут рассмотрены только конструкции, специфичные для poST.

Для определения процесса используется конструкция PROCESS <имя процесса> <тело процесса> END_PROCESS Тело процесса состоит из объявлений локальных переменных и

определений его состояний. Для определения состояния используется конструкция STATE <имя состояния> <тело состояния> END_STATE

Тело состояния определяется как последовательность конструкций языка ST, расширенного операторами таймаута и управления состояниями, а также операциями проверки состояний других процессов. Конструкция SET STATE <имя состояния> переводит текущий процесс в указанное состояние. Конструкция SET NEXT переводит текущий процесс в состояние, которое определено следующим в тексте программы. Состояние процесса, определенное первым в тексте программы, является его начальным состоянием. Процессы могут запускать и останавливать другие процессы, используя операторы START PROCESS <имя процесса> и STOP PROCESS <имя процесса> соответственно. После запуска процесс находится в своем начальном состоянии. Операторы STOP и ERROR позволяют останавливать текущий процесс нормально или по ошибке соответственно. При запуске программы процесс, определенный первым в тексте программы, устанавливается в его начальное состояние, а остальные процессы — в состояние STOP.

Время нахождения процесса в текущем состоянии может контролироваться. Каждый процесс имеет таймер нахождения в текущем состоянии. Для указания действий, которые должны быть выполнены через определенное время, в языке poST используется конструкция TIMEOUT: TIMEOUT <временной интервал> THEN <последовательность действий> END_TIMEOUT, где <временной интервал> — константа или переменная типа TIME из языка ST, определяющая время, через которое должны быть выполнены действия.

Если процесс перешел в другое состояние до истечения указанного времени, то действия не выполняются. Любое изменение состояния сбрасывает таймер процесса. Таймер также может быть сброшен с помощью оператора RESET TIMER.

Процесс может проверять, является ли другой процесс активным или пассивным. Для этого используются предикаты ACTIVE/INACTIVE/STOP/ERROR в условных выражениях, например IF PROCESS < имя процесса> IN STATE INACTIVE THEN <последовательность действий> END_IF

Предикат INACTIVE проверяет, что указанный процесс находится в неактивном состоянии, то есть в состоянии STOP или ERROR. Предикат ACTIVE проверяет, что указанный процесс находится в активном состоянии, то есть в состоянии, отличном от STOP и ERROR. Предикаты STOP и ERROR проверяют, что указанный процесс находится соответственно в состоянии STOP или ERROR.

1.2 Дедуктивная верификация

Обеспечение надежности программного обеспечения имеет большое значение для систем управления, которые часто являются критическими с точки зрения безопасности. Ошибки

в таких системах могут приводить к серьезным последствиям [34]. Для обеспечения надежности управляющего программного обеспечения применяют методы верификации — установления отсутствия ошибок в программах. В отличие от тестирования, которое проверяет отдельные пути выполнения программы, формальная верификация позволяет с помощью формального математического доказательства установить соответствие программы предъявляемым к ней требованиям для всех возможных путей выполнения.

Дедуктивная верификация — один из методов формальной верификации программного обеспечения, позволяющий проверять соответствие программы требованиям, задаваемым в виде логических формул: предусловия и постусловия, являющихся утверждениями о значениях переменных до исполнения и после исполнения программы соответственно, а также инвариантов, добавляемых в программу в качестве аннотаций. Основными формулами в данном методе являются тройки Хоара вида {P}A{Q}. Тройка Хоара {P}A{Q} является истинной, если программа A является корректной относительно предусловия P и постусловия Q. Программа A называется (частично) корректной, если из того, что перед выполнением программы A выполняется предусловие P и программа завершается, то после ее завершения выполняется постусловие q. В данном методе для каждого пути в программе между соседними контрольными точками порождается соответствующее условие корректности — логическая формула. Дедуктивная верификация предполагает, что если все условия корректности инстинны, то программа считается корректной.

Дедуктивная верификация основана на аксиоматической семантике языка программирования, в которой свойства операторов определяются в виде логических формул, выражающих отношения между переменными программы. Аксиоматическая семантика задается языком утверждений, на котором специфицируются требования к программе, и аксиоматической системой для конструкций языка программирования. Аксиоматическая семантика, предложенная Хоаром [14], включала аксиомы для простых операторов (пустого оператора и оператора присваивания) и правила вывода для управляющих структур (последовательности операторов, условных операторов, операторов циклов).

Доказательство корректности программы в аксиоматической системе Хоара является трудной задачей, так как вывод в этой системе не является однозначным. Поэтому на практике применяют два основных метода для порождения условий корректности: метод слабейшего предусловия и метод сильнейшего постусловия [7]. В методе слабейшего предусловия для программы A и заданного постусловия Q вычисляется слабейшее предусловие wp(A, Q) — такое предусловие, что тройка Хоара {wp(A, Q)}A{Q} истинна и для любого предусловия P, если {P}A{Q} — истинная тройка Хоара, то формула P → wp(A, Q) истинна. В этом случае необходимо доказать, что вычисленное предусловие следует из заданного. Условие корректности имеет вид P → wp(A, Q), где P и Q — предусловие и постусловие программы

А соответственно. Вычисление слабейшего предусловия выполняется по программе от конца к началу последовательным применением правил к последним операторам программы. Например, правило для оператора присваивания имеет следующий вид: $wp(x := e, Q) \equiv Q(x \leftarrow e)$

В методе сильнейшего постусловия для программы А и заданного предусловия Р вычисляется сильнейшее постусловие sp(A, P) — такое постусловие, что тройка Хоара {P}A{sp(A, P)} истинна и для любого постусловия Q, если {P}A{Q} — истинная тройка Хоара, то формула sp(A, P) → Q истинна. В этом случае необходимо доказать, что заданное постусловие следует из вычисленного постусловия. Условие корректности имеет вид sp(A, P) → Q, где P и Q — предусловие и постусловие программы А соответственно. Вычисление сильнейшего постусловия выполняется по программе от начала к концу последовательным применение правил к первым операторам программы. Правило для оператора присваивания имеет следующий вид: $sp(x := e, P) \equiv \exists x'(P(x \leftarrow x') \wedge x = e(x \leftarrow x'))$

Здесь вводится новая переменная x', обозначающая значение переменной x до выполнения оператора присваивания. Введение новой переменной необходимо, так как выражение e в общем случае может зависеть от x, а старое значение x не всегда может быть выражено через новое. Данное правило утверждает, что если перед выполнением оператора присваивания было истинно предусловие P, то после выполнения оператора оно будет истинно для старого значения переменной x, а также новое значение x будет равно значению выражения e, вычисленному до присваивания.

Аналогичные правила существуют для других конструкций языка программирования.

Дедуктивная верификация требует наличия инвариантов циклов — утверждений, которые истинны при каждом входе в цикл и остаются истинными после выполнения итерации цикла, если они были истинными перед ее выполнением.

Таким образом, процесс дедуктивной верификации состоит из следующих шагов:

1. Составление формальной спецификации требований к программе.
2. Определение инвариантов всех циклов.
3. Генерация условий корректности.
4. Доказательство условий корректности.

При верификации программ большого объема возникают трудности, связанные с возможными ошибками в инвариантах циклов, а также с тем, что условия корректности являются большими формулами. Поэтому процесс верификации желательно автоматизировать. Генерация условий корректности является наиболее простым этапом дедуктивной верификации и может быть автоматизирована. Генераторы условий корректности (программы, выполняющие автоматическую генерацию условий корректности) применяются во многих системах дедуктивной верификации для различных языков программирования. Многие из таких систем основаны на методе слабейшего предусловия [6, 9, 4]. Но в некоторых случаях исчисление

сильнейшего постусловия может быть проще и, следовательно, более предпочтительным. Метод сильнейшего постусловия используется в VST-Floyd [5] для языка C. Также данный метод применялся в первой версии системы C-lightVer [33] для языка C.

В отличие от генерации условий корректности, поиск инвариантов циклов и доказательство условий корректности не поддаются полной автоматизации. Задача синтеза инвариантов в общем случае является алгоритмически неразрешимой. Тем не менее существует ряд эвристик [36, 37], позволяющих в некоторых случаях находить инварианты. Например, для синтеза инварианта цикла применяется ослабление постусловия цикла, которое в общем случае не является инвариантом цикла, но следует из него, а также усиление формулы вида $\neg b \rightarrow Q$ (где b — условие цикла, Q — постусловие цикла), которое является инвариантом цикла, так как постусловие Q должно быть истинным при завершении цикла, но, как правило, слишком слабым. Для усиления утверждения используются такие эвристики, как уменьшение области значений переменных, увеличение области действия квантора всеобщности и уменьшение области действия квантора существования в постусловии цикла. Для ослабления утверждения применяют двойственные эвристики. Также константы (или переменные) могут заменяться переменными или термами. Также применяют шаблоны [27]. В этом случае задается шаблон инварианта, содержащий параметры, и инвариант синтезируется потем нахождения значений этих параметров. Также для некоторых предметных областей такие методы были разработаны [17, 1].

Существуют также подходы, в которых выполняется усиление инвариантов циклов на основе неудачных попыток доказательства [25, 16, 18]. Если некоторое условие корректности не доказано, к инварианту добавляются дополнительные утверждения так, чтобы условие корректности могло быть доказано, и снова выполняется доказательство. Данный метод последовательно усиливает инвариант, пока он не будет достаточно сильным, чтобы доказать все условия корректности.

Для доказательства порожденных условий корректности применяются системы доказательства теорем. Существует два основных вида систем доказательства теорем [32]: автоматизированные, к которым относятся, в частности, SMT-решатели, и интерактивные. Преимуществом автоматизированных систем доказательства является то, что они не требуют участия человека в процессе поиска доказательства. Кроме того, SMT-решатели позволяют находить контрпримеры в случае, если условие корректности не является истинным, что может быть полезным при поиске ошибки в программе. Тем не менее, автоматизированные системы доказательства теорем не способны доказывать сложные формулы, в частости те, которые требуют доказательства по индукции. Поэтому доказательство, как правило, является интерактивным и заключается в применении пользователем тактик доказательства. Некоторые современные интерактивные системы доказательства теорем включают автоматизированные

инструменты [12], но в отличие от систем полностью автоматического доказательства требуют взаимодействия с пользователем.

Системы дедуктивной верификации, выполняющие автоматическую генерацию условий корректности и их автоматическое (или полуавтоматическое) доказательство, были разработаны для различных языков программирования. Например, для верификации Java-программ на основе спецификаций на языке JML разработан инструмент OpenJML [6], который генерирует условия корректности и доказывает их с помощью SMT-решателей. Для переменных с одинаковыми имена выполняется их маркировка в условиях корректности. Если условие корректности не доказано, OpenJML предоставляет найденный SMT-решателем контрпример. Автоматический поиск инвариантов циклов не поддерживается, инварианты задаются вручную на языке спецификаций JML.

Для языка C существует инструмент верификации VST-Floid [5], позволяющий выполнять верификацию в системе машинной поддержки доказательства Coq в полуавтоматическом режиме. VST-Floid использует сильнейшее постусловие для генерации условий корректности. Чтобы устранить кванторы существования, связывающие старые значения переменных, используется специальная каноническая форма утверждений, которая делает эти значения явными в предусловии, что позволяет упростить условия корректности. VST-Floyd предоставляет также набор тактик для манипулирования тройками Хоара.

Существуют также универсальные инструменты для дедуктивной верификации. Такие системы предоставляют промежуточный язык, в который могут транслироваться верифицируемые программы, написанные на других языках. Затем для полученной программы на этом языке система порождает и доказывает условия корректности. Примером такой системы является Boogie [4], разработанная для верификации объектно-ориентированных программ. Boogie принимает программу, представленную на промежуточном языке, выполняет для нее синтез инвариантов циклов и порождает для этой программы условия корректности, являющиеся формулами логики первого порядка. Порожденные условия корректности доказываются с помощью автоматических инструментов доказательства (по умолчанию используется Z3). Boogie используется инструментами для верификации программ на различных языках программирования, например, для языка Spec# [2] (C#, расширенного спецификациями), а также в верификаторе VCC [21] для языка C (в том числе для верификации параллельных программ).

Для проверки темпоральных требований к реактивным системам в [24] был разработан инструмент STeP * Stanford Temporal Prover), поддерживающий дедуктивную верификацию. Верифицируемые системы в STeP описываются программами на специальном языке SPL, включающем конструкции для параллелизма, недетерминированного выбора и параметризованные операторы. Затем SPL-программа представляется системой переходов. Модуль дедуктивной верификации включает правила верификации, сводящие доказательство корректности

программы относительно темпоральной формулы к доказательству условий корректности в логике первого порядка, статический анализатор текста программы для автоматической генерации инвариантов (свойств которые выполняются во всех состояниях для каждого пути исполнения программы), компонент доказательства теорем в логике первого порядка, который упрощает условия корректности и доказывает их, если это возможно, а также методы усиления инвариантов, которые применяются в случае доказательство выполнить не удалось. STeP включает инструмент интерактивного доказательства, который используется, когда доказательство не может быть выполнено автоматически. Инструмент интерактивного доказательства предоставляет правила вывода для темпоральной логики. STeP использует два подхода к генерации инвариантов: восходящий, в котором утверждения определяются на основе анализа программы, и нисходящий, в котором выполняется усиление инвариантов на основе недоказанных условий корректности с помощью вычисления слабейшего предусловия заданного инварианта относительно соответствующих переходов.

1.3 Isabelle/HOL

Isabelle — универсальный инструмент интерактивного доказательства теорем. Основу Isabelle составляет металогика [22], которая является интуиционистским фрагментом логики высшего порядка и содержит только импликацию $\Longrightarrow$, квантор всеобщности $\bigwedge$ и равенство $\equiv$. Металогика позволяет определять новые объектные логики. Isabelle включает несколько объектных логик, таких как логика высшего порядка (HOL), логика первого порядка (FOL) и теория множеств Цермело-Френкеля (ZF). Наиболее широко применяется Isabelle/HOL — специализация Isabelle для логики высшего порядка.

Isabelle/HOL предоставляет функциональный язык программирования, обеспечивающий возможность определять типы данных и функции, а также позволяет доказывать леммы и теоремы. HOL является типизированной логикой, система типов которой включает базовые типы данных, конструкторы типов, типы функций и переменные типа, имена которых начинаются с символа ‘. как в языках семейства ML. HOL поддерживает базовые конструкции функциональных языков программирования: if-, case- и let-выражения. Все определения типов, функции и теоремы должны быть включены в теорию. Каждая теория с именем T содержится в файле теории T.thy, который имеет следующий синтаксис:

```
theory T
imports T_1 ... T_n
begin
    definitions, theorems and proofs
end
```

где T_1, ..., T_n — имена теорий, которые импортирует данная теория.

HOL включает теории, содержащие определения базовых типов данных, таких как bool (логический тип), nat (тип натуральных чисел), int (тип целых чисел), real (тип вещественных чисел), list (тип списков) и т. п.

Пользовательские типы данных могут быть определены набором конструкторов следующим образом:

$$datatype\ (`a_1, ..., `a_n)\tau = C_1\ ''\tau''_{1,1}\ \ldots\ ''\tau''_{1,n_1}\ |\ \ldots\ |\ C_k\ ''\tau''_{k,1}\ \ldots\ ''\tau''_{k,n_k},$$

где C_1, ..., C_k — конструкторы, τ_{ij} — типы их аргументов. Для каждого типа Isabelle/HOL автоматически создает правило структурной индукции.

Нерекурсивные определения функций и констант могут быть введены с помощью ключевых слов `definition` и `abbreviation`. Основное отличие между ними состоит в том, что определение, введенное с помощью команды definition, необходимо явно раскрывать, используя теорему f_def, где f— имя определяемой константы. Имена констант, введенных с помощью команды abbreviation, раскрываются автоматически при синтаксическом анализа, они отсутствуют во внутреннем представлении термов, что позволяет упростить доказательства. Для типов данных также могут быть введены новые имена с помощью команды `type_synonum`.

Рекурсивные функции могут быть определены на основе сопоставления с образцом, в качестве образцов используются конструкторы значений типа данных. Isabelle/HOL требует, чтобы все функции были тотальными. В простейшем случае завершение функции доказывается автоматически. В более сложном случае оно может быть доказано вручную. Для каждой рекурсивной функции автоматически создается схема индукции, отражающая схему рекурсии и позволяющая упростить доказательство свойств этой функции.

Команды `lemma` и `theorem` начинают доказательство теоремы, задают название теоремы и доказываемое утверждение. Имя теоремы позволяет ссылаться на эту теорему после завершения доказательства и не является обязательным. Далее следует доказательство теоремы. Первоначально состояние доказательства состоит из одной подцели (утверждения, которое необходимо доказать для доказательства теоремы), совпадающей с основной целью. Доказательство может быть выполнено в виде скрипта доказательства, состоящего из последовательности команд, таких как `apply` и `subgoal`. Команда apply(m) применяет указанный метод доказательства m. Некоторые методы применяются только к первой подцели, другие по умолчанию работают со всеми подцелями. Чтобы применить такие методы только к первым n подцелям, необходимо использовать команду apply(m)[n].

Команда `subgoal` преобразует состояние цели в контекст доказательства и оставшееся заключение. Данная команда обеспечивает возможность задавать имена параметрам

и посылкам цели, что позволяет ссылаться на них в доказательстве. Например, команда subgoal premises $prems$ for $x\ y\ z$ задает имена x, y и z параметрам цели и делает посылки цели предположениями контекста с именем $prems$. Если имена параметров или посылок не заданы, они остаются недоступными в тексте доказательства. Имена параметров остаются внутренними.

Преимуществом системы Isabelle/HOL является то, что она совмещает мощную автоматизацию доказательств и возможность пошагового доказательства, которая является особенно полезной, когда доказательство не может быть выполнено полностью автоматически. Isabelle/HOL предоставляет ряд методов доказательства, некоторые из которых являются автоматическими, а другие позволяют выполнять отдельные шаги доказательства.

К методам пошагового доказательства относятся такие методы, как $cases$ и $induction$, позволяющие выполнять доказательства разбором случаев и по индукции соответственно, методы $rule$, $erule$ и т. п., позволяющие применить указанное правило, и другие методы. Методы $cases$ и $induction$ часто применяются для доказательства свойств рекурсивных функций. Модификатор $arbitrary$: метода $induction$ позволяет обобщить указанные переменные перед выполнением индукции. Модификатор $rule$: методов $cases$ и $induction$ позволяет указать правило разбора случаев или индукции соответственно, например, правило индукции для рекурсивной функции. Метод $rule$ применяет указанное правило к текущему состоянию доказательства. Если правило r имеет вид $P_1 \Longrightarrow \ldots \Longrightarrow P_n \Longrightarrow Q$, то команда apply$(rule\ r)$, примененная к цели вида $\bigwedge x_1 \ldots x_k . A_1 \Longrightarrow \ldots \Longrightarrow A_m \Longrightarrow C$, унифицирует Q и C и заменяет цель C новыми подцелями $\bigwedge x_1 \ldots x_k . U(A_1) \Longrightarrow \ldots \Longrightarrow U(A_m) \Longrightarrow U(P_1), \ldots, \bigwedge x_1 \ldots x_k . U(A_1) \Longrightarrow \ldots \Longrightarrow U(A_m) \Longrightarrow U(P_n)$, где U— найденный унификатор. Метод $erule$ применяет указанное правило следующим образом. Если правило r имеет вид $P_1 \Longrightarrow \ldots \Longrightarrow P_n \Longrightarrow Q$, то команда apply(erule r) к цели вида $\bigwedge x_1 \ldots x_k . A_1 \Longrightarrow \ldots \Longrightarrow A_m \Longrightarrow C$ унифицирует Q с C и P_1 с A_j для некоторого j и заменяет цель C новыми подцелями $\bigwedge x_1 \ldots x_k . U(A_1) \Longrightarrow \ldots \Longrightarrow U(A_{j-1}) \Box U(A_{j+1}) \Longrightarrow \ldots \Longrightarrow U(A_m) \Longrightarrow U(P_2), \ldots, \bigwedge x_1 \ldots x_k . U(A_1) \Longrightarrow \ldots \Longrightarrow U(A_{j-1}) \Longrightarrow U(A_{j+1}) \Longrightarrow \ldots \Longrightarrow U(A_m) \Longrightarrow U(P_n)$ где U — найденный унификатор. Методы пошагового доказательства могут быть полезны в случаях, когда автоматические методы не справляются.

Isabelle/HOL также предоставляет автоматические методы доказательства. Так метод $auto$ позволяет выполнять упрощения и может доказывать простые логические утверждения в классической логике. Данный метод применяется ко всем подцелям, и, если доказательство подцели не может быть выполнено полностью, $auto$ просто упрощает ее. В некоторых случаях применение $auto$ может занять много времени или сильно упростить подцель. В этом случае может применяться более простой и предсказуемый метод $simp$. Метод $simp$ упрощает посылки и заключение первой подцели, используя правила упрощения и посылки этой цели. Равенства, определяемые рекурсивными функциями, автоматически добавляются в набор пра-

вил упрощения. Для добавления или временного удаления правил упрощения в наборе могут использоваться соответственно модификаторы add: и del: метода $simp$. Метод $auto$ также имеет соответствующие модификаторы $simp\ add$: и $simp\ del$:. Метод $simp$ может расщеплять if- и case-выражения. Чтобы расщепить такое выражения в посылках подцели, необходимо указать соответствующее правило расщепления, например apply$(simp\ split : if_splits)$.

Для доказательства сложных логических утверждений Isabelle/HOL предоставляет метод доказательства $blast$. Метод $blast$ теоретически является полным для логики первого порядка, но на практике его возможности ограничены. Данный метод не выполняет упрощения и плохо работает с равенствами.

Формулы линейной арифметики могут быть доказаны с помощью метода $arith$. В целочисленной арифметике этот метод может даже доказывать формулы с кванторами. Методы $simp$ и $auto$ также могут доказывать многие формулы линейной арифметики, так как вызывают более простую и быструю версию $arith$.

Методы можно объединять в выражения, используя различные комбинаторы, такие как ”(последовательная композиция) и ”+”(повторить применение метода не менее одного раза). Также можно определять пользовательские методы [20].

После завершения доказательства теоремы все ее свободные переменные заменяются неизвестными, которые при использовании теоремы заменяются конкретными термами неявно в результате унификации или явно при использовании атрибута of. Выражение $th[of\, t_1\ \ldots t_m]$ позволяет выполнить замену неизвестных в теореме th термами $t_1,\ldots, t_m$ слева направо. Некоторые позиции могут быть пропущены, используя _ (нижнее подчеркивание). В доказательствах также используются атрибуты теорем OF и $THEN$. OF позволяет применять одну теорему к другим. Если $A \implies B$ и A' — утверждения теорем r и r' соответственно, то выражение $r[OF\, r']$ означает применение теоремы r к r', где r может рассматриваться как функция, возвращающая теорему B для заданной теоремы A. Теорема $r[THEN]$ является композицией правил r и r'.

Isabelle/HOL включает мощный инструмент автоматического доказательства `sledgehammer` [3], который параллельно запускает несколько внешних инструментов автоматического доказательства теорем (ATP) и SMT-решателей, выполняющих поиск доказательства. Результатом успешного применения `sledgehammer` является команда Isabelle/HOL, которая восстанавливает найденное внешним инструментом доказательство [10]. Также может быть указано приблизительное время выполнения команды. Задача, передаваемая внешнему инструменту доказательства, состоит из текущей цели и набора фактов, выбираемых эвристически из текущего контекста теории. Основным преимуществом `sledgehammer` является то, что он может находить необходимые для доказательства леммы, что не требует их поиска вручную. Перед вызовом sledgehammer рекомендуется вызвать $simp$

или *auto* для предварительного упрощения цели. Sledgehammer имеет ряд опций, которые позволяют задавать значения параметров, таких как используемые внешние инструменты доказательства, максимальное количество фактов, передаваемых внешним инструментам доказательства, временные ограничения и т. п. Также можно явно указать факты, которые необходимо передать внешним инструментам доказательства

Наиболее известной программной системой, верифицированной с помощью Isabelle/HOL является микроядро операционной системы общего назначения seL4 [31], реализованное на языке программирования C.

Система Isabelle/HOL была применена в ряде работ по моделированию программных систем. Например, в [23] в Isabelle/HOL была построена формальная модель архитектуры X86 и смоделированы ассемблерные инструкции для верификации операционных систем. В [13] разработана модель распределенных асинхронно взаимодействующих компонентов и определена их семантика, а также доказаны свойства структуры и поведения компонентов и их асинхронного взаимодействия. В работе [19] было формализовано доказательство корректности SAT-решателя, основанного на современных алгоритмах. В работе [26] выполнена формальная верификация Java-компилятора в Isabelle, основанная формализации языка Java.

Система Isabelle/HOL также применялась для дедуктивной верификации гибридных систем на основе дифференциальной динамической логики [15, 11]. В [15] авторы формализовали в системе Isabelle/HOL семантику команд, используя метод слабейшего предусловия, и правила вывода для дифференциальной динамической логики.

1.4 Требования к генератору условий корректности

Генератор условий корректности poST-программ представляет собой утилиту командной строки, которая генерирует теорию Isabelle/HOL по исходному коду poST-программы. Эта теория содержит порожденные условия корректности.

Требования к процесс-ориентированным программам формализуются в виде двух аннотаций. Первая аннотация (env) задает ограничения на изменения входных переменных средой (в частности объектом управления). Вторая аннотация— инвариант цикла управления (inv), который должен быть истинным при каждом входе в цикл управления. Первая аннотация, как правило, общая для всех требований, а инварианты цикла управления различны. Чтобы не порождать условия корректности отдельно для каждого требования, условия корректности параметризуются этими аннотациями. Кроме того условия корректности параметризуются инвариантами всех циклов на соответствующем пути в программе.

Имена переменных, процессов и состояний процессов в Isabelle/HOL представляются именованными константами: кодируются натуральными числами, и затем для этих констант вводятся имена.

Таким образом, были сформулированы следующие требования к генератору условий корректности:

1. Порождаемые условия корректности должны быть параметризованы аннотациями inv и env и инвариантами циклов, используемых в этом условии корректности. Порядок параметров должен быть определен для целей дальнейшего доказательства.
2. Однозначное кодирование и именование переменных, процессов и кодов процессов в Isabelle/HOL теории.
3. Необходимо порождать условия корректности для проверки отсутствия следующих ошибок: деление на ноль и выход за границы массива, нулевой шаг цикла FOR.
4. В случае наличия синтаксических ошибок в программе должны выводиться сообщения об ошибках.

2 Разработка и реализация генератора условий корректности

2.1 Язык аннотаций для процесс-ориентированных программ

В данном пункте описывается язык, используемый для формализации требований к процесс-ориентированным программам. Данный язык был введен ранее в [28].

Язык основан на типе данных *состояние изменений* (state), который хранит всю историю изменений состояния программы, и наборе функций над этим типом данных. Тип данных state определяется следующим набором конструкторов значений:

- emptyState : state — начальное состояние системы управления;
- toEnv : state → state — вход в цикл контроллера;
- setVarBool : state × variable × bool → state — присваивает значение переменной типа bool;
- setVarInt : state × variable × int → state — присваивает значение переменной типа int;
- setVarReal : state × variable × real → state — присваивает значение переменной типа real;
- setVarArrayBool : state × variable × int × bool → state — присваивает значение типа bool элементу массива;
- setVarArrayInt : state × variable × int × int → state — присваивает значение типа int элементу массива;
- setVarArrayReal : state × variable × int × bool → state — присваивает значение типа real элементу массива;
- setPstate : state × process × pstate → state — устанавливает состояние процесса для состояния системы управления;
- reset : state × process → state — сбрасывает локальный таймер процесса.

Для каждого изменения в программе определен соответствующий конструктор.

В аннотациях используется следующий набор предметно-ориентированных функций над состояниями изменений:

- getVarBool : state × variable → bool — возвращает значение переменной типа bool;
- getVarInt : state × variable → int — возвращает значение переменной типа int;
- getVarReal : state × variable → real — возвращает значение переменной типа real;
- getVarArrayBool : state × variable × int → bool — возвращает значение элемента массива типа bool;
- getVarArrayInt : state × variable × int → int — возвращает значение элемента массива типа int;
- getVarArrayReal : state × variable × int → real — возвращает значение элемента массива типа real;
- getPstate : state × process → pstate — возвращает текущее состояние процесса;

- substate : state × state → bool. Функция $substate(s', s)$ возвращает значение true, если $s' = s$ или существуют состояние s'' , конструктор c и значения $v_1, \ldots, v_n$, такие, что $s = c(s'', v_1, \ldots, v_n)$ и $substate(s', s'') = true$;
- toEnvNum : state × state → nat — возвращает количество применений конструктора toEnv, необходимого для получения второго состояния из первого;
- toEnvP : state → bool — проверяет, имеет ли состояние вид toEnv(s) для некоторого состояния s;
- ltime: state × process → nat — возвращает значение локального таймера процесса.

Предметная ориентированность этих функций заключается в том, что они позволяют естественным образом описывать требования к процесс-ориентированным программам.

Здесь типы bool, nat, int и real описывают множества логических констант (true и false), натуральных чисел, целых чисел и вещественных чисел, соответственно. Типы variable, process и pstate используются, соответственно, для кодирования имен переменных, процессов и состояний процессов процесс-ориентированной программы.

Введем следующие обозначения. Пусть setVar обозначает один из конструкторов setVarBool, setVarInt или setVarReal в зависимости от типа переменной. Аналогично setVarArray— один из конструкторов setVarArrayBool, setVarArrayInt или setVarArrayReal, getVar— одна из функций getVarBool, getVarInt или getVarReal, getVarArray— одна из функций getVarArrayBool, getVarArrayInt или getVarArrayReal.

2.2 Алгоритм генерации условий корректности

В этом разделе описывается алгоритм генерации условий корректности для языка poST, на котором основан генератор условий корректности.

Пусть $code$ — функция, которая возвращает коды переменных, процессов и состояний процессов.

Правила для выражений

Функция $expr(e, u)$, выполняющая символическое вычисление выражения e в состоянии изменений u определяется следующими правилами:

- $expr(c, u) = c$, если c — константа,
- $expr(x, u) = (getVar(s, code(x)))$, если x— переменная,
- $expr(x[i], u) = (getVarArray(u, code(x), expr(i, u)))$,
- $expr(\texttt{PROCESS}\ p\ \texttt{IN STATE STOP}, u) = (getPstate(s, code(p)) = \texttt{STOP})$,
- $expr(\texttt{PROCESS}\ p\ \texttt{IN STATE ERROR}, u) = (getPstate(s, code(p)) = ERROR)$,
- $expr(\texttt{PROCESS}\ p\ \texttt{IN STATE INACTIVE}, u) = (getPstate(u, code(p)) = STOP \vee getPstate(u, code(p)) = ERROR)$,

- $expr(\texttt{PROCESS } p \texttt{ IN STATE ACTIVE}, u) = (\neg(getPstate(u, code(p)) = STOP \vee getPstate(u, code(p)) = ERROR))$,
- $expr((e), u) = expr(e, u)$,
- $expr(e_1 \texttt{ op } e_2, u) = (expr(e_1, u)$ op $expr(e_2, u))$, если $op \in \{+, -, *\}$,
- $expr(e_1/e_2, u) = (expr(e_1, u)/expr(e_2, s))$ если хотя бы одно из выражений e_1 или e_2 имеет тип real
- $expr(e_1/e_2, u) = (expr(e_1, u)$ div $expr(e_2, u))$ если оба выражения e_1 и e_2 имеют тип int,
- $expr(e_1 ** e_2, u) = (expr(e_1, u)^{expr(e_2,u)})$,
- $expr(e_1 \texttt{ op } e_2, u) = (expr(e_1, u)$ op $expr(e_2, u))$, если $\texttt{op} \in <, >, <=, >=, =$,
- $expr(e_1 <> e_2, u) = (expr(e_1, u) \neq expr(e_2, u))$,
- $expr(e_1 \texttt{ AND } e_2, u) = (expr(e_1, u) \wedge expr(e_2, u))$,
- $expr(e_1 \texttt{ XOR } e_2, u) = (expr(e_1, u) \neq expr(e_2, u))$,
- $expr(e_1 \texttt{ OR } e_2, u) = (expr(e_1, u) \vee expr(e_2, u))$.

Правила для области определения выражений

Чтобы порождать условия корректности для проверки ошибок таких, как деление на ноль и выход за границы массива, определим функцию $D(e)$, которая для заданного выражения e вычисляет логическую формулу языка спецификаций, описывающую условие, при котором выражение e определено.

Данная функция определяется следующими правилами:

1. $D(c, u) =$ True, если c — константа
2. $D(x, u) =$ True, если x — переменная
3. $D(x[i], u) = (D(i, u) \wedge a \leq expr(i, s) \wedge expr(i, s) \leq b)$, где a и b нижний и верхний пределы индексов массива x соответственно
4. $D(\texttt{PROCESS } p \texttt{ IN STATE } q, u) =$ True, где $q \in \{STOP, ERROR, ACTIVE, INACTIVE\}$
5. $D((e), u) = D(e, u)$
6. $D(e_1 \texttt{ op } e_2, u) = (D(e_1, u) \wedge D(e_2, s))$, где $\texttt{op} \in \{+, -, *, **\}$
7. $D(e_1/e_2, s) = (D(e_1, u) \wedge D(e_2, u) \wedge expr(e_2, u) \neq 0)$
8. $D(e_1 \texttt{ op } e_2, u) = (D(e_1, s) \wedge D(e_2, u))$, где $\texttt{op} \in \{<, >, <=, >=, =, <>\}$
9. $D(e_1 \texttt{ AND } e_2, u) = (D(e_1, u) \wedge (expr(e_1, s) \longrightarrow D(e_2, u)))$
10. $D(e_1 \texttt{ XOR } e_2, u) == (D(e_1, s) \wedge D(e_2, u))$
11. $D(e_1 \texttt{ OR } e_2, u) = (D(e_1, u) \wedge (\neg expr(e_1, u) \longrightarrow D(e_2, u)))$

Правила для операторов

Простые операторы языка poST (операторы присваивания, изменения состояний процессов, сброса таймера) осуществляют преобразование текущего состояния системы управления. Также для формализации операторов EXIT и RETURN при обработке программы необходимо помнить, были ли встречены данные операторы. Для этого введем переменную $control$, хранящую информацию об этих операторах. Таким образом, предусловия и постусловия имеют вид $P \wedge u_{cur} = u \wedge control = ctrl$, где P — некоторое утверждение, u_{cur} — текущее состояние изменений. Переменная $control$ принимает одно из трех значений: NORMAL, EXIT или RETURN. Значение NORMAL означает, что на текущем пути не были встречены операторы EXIT и RETURN. Значение EXIT означает, что был встречен оператор EXIT. Значение RETURN означает, что на текущем пути был встречен оператор RETURN. В алгоритме генерации условий корректности формулы вида $P \wedge u_{cur} = u \wedge control = ctrl$ представляются в виде тройки (P, u, ctrl). Алгоритм определяется как рекурсивная функция $gen(P_1; \ldots; P_n, A)$, которая вычисляет сильнейшие постусловия для предусловий P_1,…, P_n и фрагмента программы A. Каждое предусловие соответствует некоторому пути в программе.

Пусть p_0 — код процесса, которому принадлежит первая конструкция (оператор или состояние процесса) обрабатываемого фрагмента программы, s_0 — код состояния процесса, которому принадлежит первый оператор рассматриваемого фрагмента программы. Переменная $pstates$ является двумерным массивом, хранящим коды состояний процессов. Первый индекс соответстует коду процесса, которому принадлежит состояние, а второй — номеру состояния в этом процессе. Пусть += обозначает операцию добавления элемента к списку, а операция +, примененная к спискам, обозначает их объединение. Опишем функцию gen, используя псевдокод.

Сначала определим функцию gen для простых операторов.

Для операторов присваивания, заимствованных из языка ST, данная функция определяется следующим образом:

function gen((P, u, NORMAL), x:=e)

vcs += $P \longrightarrow D(e,u)$

return $\{(P \wedge D(e,u), setVar(u, code(x), expr(e,u)), NORMAL)\}$

end function

function gen((P, u, NORMAL), x[i]:=e)

vcs += $P \longrightarrow D(i,u) \wedge a \leq expr(i,u) \wedge expr(i,u) \leq b \wedge D(e,u)$

return $\{(P \wedge D(i,u) \wedge a \leq expr(i,u) \wedge expr(i,u) \leq b \wedge D(e,u), setVarArray(u, code(x), expr(i,u), expr(e,u)), NORMAL)\}$

end function

Данные функции порождают условия корректности, утверждающие, что значения входящих в операторы выражений определены, и возвращают вычисленные сильнейшие постусловия. Первая функция соответствует простому присваиванию. В этом случае из предусловия должно следовать, что значение присваиваемого выражения e определено. Вторая функция соответствует присваиванию элементу массива. Вэтом случае должны быть определены выражение i, задающее индекс элемента, и присваиваемое выражение e. Также индекс должен принадлежать диапазону индексов массива.

Функция gen для операторов изменения состояния процесса, сброса таймера и управления другими процессами языка poST определяется следующими правилами:

1. gen((P, u, NORMAL), SET STATE s) = $\{(P, setPstate(u, p_0, code(s)), NORMAL)\}$,
2. gen((P, u, NORMAL), SET NEXT) = $\{(P, setPstate(u, p_0, s_0 + 1), NORMAL)\}$,
3. gen((P, u, NORMAL), RESET TIMER) = $\{(P, reset(u, p_0), NORMAL)\}$,
4. gen((P, u, NORMAL), START PROCESS p) =
   $\{(P, setPstate(setVar(...setVar(u, v_1, v_1')...v_n, v_n'), code(p), pstates[code(p)][0]), NORMAL)\}$,
5. gen((P, u, NORMAL), RESTART) =
   $\{(P, setPstate(setVar(...setVar(u, v_1, v_1')...v_n, v_n'), p_0, pstates[p_0][0]), NORMAL)$,
6. gen((P, u, NORMAL), STOP PROCESS p) = $(P, setPstate(u, code(p), STOP), NORMAL)\}$,
7. gen((P, u, NORMAL), STOP) = $\{(P, setPstate(u, p_0 STOP), NORMAL)\}$,
8. gen((P, u, NORMAL), ERROR PROCESS p) = $\{(P, setPstate(u, code(p), ERROR), NORMAL)\}$,
9. gen((P, u, NORMAL), ERROR) = $\{(P, setPstate(u, p_0, ERROR), NORMAL)\}$

В правилах 4 и 5 для операторов START PROCESS и RESTART v_1,..., v_n— инициализированные переменные в запускаемом процессе, v_1',..., v_n'— их начальные значения. Перед запуском процесса выполняется инициализация объявленных в нем переменных.

Отметим, что вычисленные сильнейшие постуствовия для простых операторов не содержат квантор существования по новой переменной, соответствующей значению до выполнения оператора. Утверждения в предусловии зависят от состояний изменений до выполнения оператора не преобразуется. Новое состояние изменений получается из состояния до выполнения оператора применением соответствующего конструктора типа $state$.

Далее определим функцию gen для последовательности операторов, а также операторов выбора, циклов и операторов EXIT и RETURN из языка ST.

Для последовательности операторов функция определяется следующим правилом:

gen(c, q w) = $gen(gen(c, q), w)$,

где c— предусловие, q— оператор, w— последовательность операторов.

Для списка предусловий функция имеет вид:

gen(c_1, ..., c_n, w) = $gen(c1, w)$ + ... + $gen(cn, w)$

Для условного оператора функция gen определяется следующим образом:

function gen((P, u, NORMAL), IF e THEN w ELSIF e_1 THEN w_1 ... ELSIF e_n THEN w_n ELSE w_{n+1} END_IF)

vcs+= $\{P \longrightarrow D(e,u)$;

$P \wedge D(e,u) \wedge \neg expr(e,u) \longrightarrow D(e_1,u)$;...;

$P \wedge D(e,u) \wedge \neg expr(e,u) \wedge D(e_1,u) \wedge \neg expr(e_1,u) \wedge ... \wedge D(e_{n-1},u) \wedge \neg expr(e_{n-1},u) \longrightarrow D(e_n,u)\}$

return $gen((P \wedge D(e,u) \wedge expr(e,u), u, NORMAL), w)$

+ $gen((P \wedge D(e,u) \wedge \neg expr(e,u) \wedge D(e_1,u) \wedge expr(e_1,u), u, NORMAL), w_1)$+...

+ $gen((P \wedge D(e,u) \wedge \neg expr(e,u) \wedge D(e_1,u) \wedge \neg expr(e_1,u) \wedge ... \wedge D(e_{n-1},u) \wedge \neg expr(e_{n-1},u) \wedge D(e_n,u) \wedge expr(e_n,u), u, NORMAL), w_n)$

+ $gen((P \wedge D(e,u) \wedge \neg expr(e,u) \wedge D(e_1,u) \wedge \neg expr(e_1,u) \wedge ... \wedge D(e_{n-1},u) \wedge \neg expr(e_{n-1},u) \wedge D(e_n,u) \wedge \neg expr(e_n,u), u, NORMAL), w_{n+1})$

end function

Из предусловия должно следовать, что значение первого логического выражения e определено. Также каждое выражение e_i, i=1,..., n должно быть определено, если ложны выражение e и все выражения e_1,..., e_{i-1}. Соответствующие условия корректности порождаются данной функцией. Затем для каждой ветви вычисляется сильнейшее постусловие.

Для оператора CASE функция имеет вид:

function gen((P, u, NORMAL), CASE e OF l_1: l_n: w_n ELSE w END_CASE)

vcs += $P \longrightarrow D(e,u)$

return $gen((P \wedge D(e,u) \wedge case(e,l_1,u), u, NORMAL), w_1)$+...

+ $gen((P \wedge D(e,u) \wedge case(e,l_n,u), u, NORMAL), w_n)$

+ $gen((P \wedge D(e,u) \wedge \neg case(e,l_1,u) \wedge ... \wedge \neg case(e,l_n,u), u, NORMAL), w)$

end function

Данная функция также порождает одно условие корректности, утверждающее, что выражение e определено, и вычисляет сильнейшее постусловие для каждой ветви.Здесь используется функция $case$, которая определяется следующим образом:

case(e, $l_1, ..., l_n$, u) = $expr(e,u) = l_1 \vee \ldots \vee expr(e,u) = l_n$

Для цикла WHILE функция определяется следующим образом.

function gen($\{(P_j, u_j, ctrl_j)$, j=1,..., n$\}$, {loopinv} WHILE e DO w END_WHILE)

res := $\{\}$

for i=1,..., n **do**

if $ctrl_i = NORMAL$ **then**

vcs += $P_i \longrightarrow loopinv(u_i)$

else

res += $P_i, u_i, ctrl_i$

end if

end for

vcs += $loopinv(u_0) \longrightarrow D(e, u_0)$

for $(P, u, ctrl) \in gen((loopinv(u_0) \wedge D(e, u_0) \wedge expr(e, u_0), u_0, NORMAL), w)$ **do**

case $ctrl$

NORMAL: vcs += $P \longrightarrow loopinv(u)$

EXIT: res += $(P, u, NORMAL)$

RETURN: res += $(P, u, ctrl)$

end case

end for

res += $(loopinv(u_0) \wedge D(e, u_0) \wedge \neg expr(e, u_0), u_0, NORMAL)$

return res

end function

Для каждого пути в программе до цикла, если ранее не были встречены операторы EXIT и RETURN ($ctrl = NORMAL$), то порождается условие корректности $P_i \longrightarrow loopinv(u_i)$, утверждающее, что из соответствующего предусловия P_i следует инвариант цикла $loopinv$ в соответствующем состоянии изменений u_i. Если же один из данных операторов был встречен ранее, то цикл не выполняется. Затем порождается условие корректности, утверждающее, что если инвариант истинен, то условие цикла e определено. Далее обрабатывается тело цикла. В этом случае условие цикла истинно. Если на некоторм пути в теле цикла не были встречены операторы EXIT и RETURN, то порождается условие корректности $P \longrightarrow loopinv(u)$. Если был встречен оператор EXIT, осуществляется выход из цикла и операторы, следующие за данным оператором цикла, выполняются. В этом случае переменная $control$ устанавливается в значение $NORMAL$. Если был встречен оператор RETURN, дальнейшие операторы не выполняются, значение переменной $control$ не изменяется. Затем обрабатывается оставшаяся часть программы. В этом случае условие цикла ложно. В данной функции вводится новая переменная u_0, обозначающая состояние изменение в начале итерации цикла и после завершения цикла.

Для оператора REPEAT функция gen пределяется аналогично. В отчие от цикла WHILE, в цикле REPEAT условие проверяется после выполнения тела цикла. Функция имеет вид

function gen({$P_j, u_j, ctrl_j$, j=1,..., n}, REPEAT w {loopinv} UNTIL e END_FOR)

res := {}

for i=1,..., n **do**

if $ctrl_i = NORMAL$ **then**

for $(P, u, ctrl) \in gen((P_i, u_i, ctrl_i), w)$ **do**

case $ctrl$

NORMAL: vcs += $P \longrightarrow loopinv(u)$

EXIT: res += $(P, u, NORMAL)$

RETURN: res += $(P, u, ctrl)$

end case

end for

else

res += $P_i, u_i, ctrl_i$

end if

end for

vcs += $loopinv(u_0) \longrightarrow D(e, u_0)$

for $(P, u, ctrl) \in gen((loopinv(u_0) \wedge D(e, u_0) \wedge expr(e, u_0), u_0, NORMAL), w)$ **do**

case $ctrl$

NORMAL: vcs += $P \longrightarrow loopinv(u)$

EXIT: res += $(P, u, NORMAL)$

RETURN: res += $(P, u, ctrl)$

end case

end for

res += $(loopinv(u_0) \wedge D(e, u_0) \wedge \neg expr(e, u_0), u_0, NORMAL)$

return res

end function

Функция gen для цикла FOR определяется следующим образом:

function gen({c_1;...; c_n}, {loopinv} FOR i := e_1 TO e_2 BY e_3 DO w END_FOR)

res := {}

for $(P, u, ctrl) \in gen(\{c_1; ...; c_n\}, i := e_1)$ **do**

if $ctrl = NORMAL$ **then**

vcs += $P \longrightarrow loopinv(u)$

else

res += $(P, u, ctrl)$

end if

end for

vcs += $loopinv(u_0) \longrightarrow D(e_2, u_0) \wedge D(e_3, u_0) \wedge expr(e3, u0)$≠0

l:= $gen((loopinv(u_0) \wedge D(e_2, u_0) \wedge D(e_3, u_0) \wedge expr(e_3, u_0) > 0 \wedge expr(i, u_0) \leq expr(e_2, u_0), u_0, NORMAL), w; i := i + e_3)$ +
$gen((loopinv(u_0) \wedge D(e_2, u_0) \wedge D(e_3, u_0) \wedge expr(e_3, u_0) < 0 \wedge expr(i, u_0) \geq expr(e_2, u_0), u_0, NORMAL), w; i := i + e_3)$

for $(P, u, ctrl) \in l$ **do**

case $ctrl$

NORMAL: vcs += $P \longrightarrow loopinv(u)$

EXIT: res += $(P, u, NORMAL)$

RETURN: res += $(P, u, ctrl)$

end case

end for

res += $(loopinv(u_0) \wedge D(e_2, u_0) \wedge D(e_3, u_0), u_0, , NORMAL)$

return res

end function

Функция цикла FOR также определяется аналогично функции для цикла WHILE, но перед входом в цикл счетчику присваивается начальное значение $i := e_1$, а также после каждой итерации цикла значение счетчика увеличивается на значение шага e_3. Также порождается условие корректности, утверждающие, что конечное значение e_2 и шаг цикла e_3 определены и шаг цикла не равен нулю. Данное определение функции gen предполагает, что счетчик, границы и шаг цикла не изменяются в теле цикла. Утверждение о том, что значение счетчика после завершения цикла больше (или меньше) конечного значения, не добавляется, так как стандарт IEC не определяет это значение.

Шаг цикла является необязательным параметром. Если шаг не задан, его значение полагается равным 1.

gen($\{c_1$;...; $c_n\}$, {loopinv} FOR i := e_1 TO e_2 DO w END_FOR) = gen($\{c_1$;...; $c_n\}$, {loopinv} FOR i := e_1 TO e_2 BY 1 DO w END_FOR)

Для операторов EXIT и RETURN функция gen определяется следущими правилами:

1. gen((P, u, NORMAL), EXIT) = (P, u, EXIT),
2. gen((P, u, NORMAL), RETURN) = (P, u, RETURN),
3. gen((P, u, ctrl), A) = (P, u, ctrl), если ctrl ≠ NORMAL

Правила 1 и 2 сохраняют информацию о встреченных операторах EXIT и RETURN в переменной $control$. Правило 3 осуществляет просачивание ее значения.

Для оператора таймаута языка poST функция gen определяется следующими правилами:

1. gen((P, u, NORMAL), TIMEOUT e THEN w END_TIMEOUT) =
   $gen((P \wedge e/t \leq ltime(u, p_0), u, NORMAL), w)$
   + $\{(P \wedge \neg e/t \leq ltime(u, p_0), u, NORMAL)\}$,
2. gen((P, u, NORMAL), TIMEOUT e THEN w END_TIMEOUT) =
   $gen((P \wedge (expr(e, u)–1) div t + 1 \leq ltime(u, p_0), u, NORMAL), w)$
   + $\{(P \wedge \neg((expr(e, u)–1) div\, t + 1 \leq ltime(u, p_0)), u, NORMAL)\}$,

где t — период активации процессов. Время в условиях корректности выражается в количестве итераций цикла управления. Поэтому в данных правилах выполняется деление

времени, заданного в операторе таймаута на период активации процессов с округлением вверх. В случае, когда значение таймера текущего процесса превышает указанное время, обрабатывается тело оператора таймаута. Если значение таймера текущего процесса не достигло указанного времени, тело таймаута не выполняется. Первое правило соответствует случаю, когда время в операторе таймаута задается константой. Второе правило соответствует случаю, когда время задается переменной.

Для состояний и процессов функция gen определяется следующими правилами:

1. gen((P, u, NORMAL), STATE n w END_STATE) = $gen((P, u, NORMAL), w)$,
2. gen((P, u, NORMAL), PROCESS p q_1 ... q_n END_PROCESS) = $gen((P \wedge getPstate(u, code(p)) = s_1, u, NORMAL), q_1)$ + ...
   + $gen((P \wedge getPstate(u, code(p)) = s_n, u, NORMAL), q_n)$
   + $(P \wedge getPstate(u, code(p)) = STOP, u, NORMAL)$
   + $(P$□$getPstate(u, code(p)) = ERROR, u, NORMAL)$, если q_i — тело состояния s_i.

При обработке процесса (правило 2) выполняется обработка всех его состояний. Также процесс может находиться в состоянии $STOP$ или $ERROR$. При обработке состояния s_i к предусловию добавляется подформула $getPstate(u, code(p)) = s_i$ и выполняется обработка тела состояния.

Функция gen для программы определяется следующим образом:

function gen(PROGRAM n q_1... q_n END_PROGRAM)
vcs+= $u_0 = setPstate(setVar(...setVar(emptyState, code(v_1), v'_1)...code(v_n)v'_n)$ code(p$_1$)$, pstates[code(p_1)][0]) \longrightarrow inv(u_0)$
u_1 := $setVar(...setVar(u_0, v_{e1}v'_{e1}value)...v_{ek}v'_{ek}value)$
for $(P, u, ctrl) \in gen((inv(u_0) \wedge env(u_1), u_1, NORMAL), q_1...q_n)$ **do**
vcs += $P \longrightarrow inv(u)$
end for
end function

где q_i — тело процесса p_i, v_i— инициализированные переменные, объявленные на уровне программы, v'_i— начальное значение переменной v_i.

Для программы сначала порождается условие корректности, соответствующее инициализации программы. В этом случае выполняется инициализация переменных программы и процессов, а затем первый процесс устанавливается в начальное состояние. После инициализации должен выполняться инвариант цикла управления inv. Далее входным переменным присваиваются произвольные значения, что соответствует означиванию их средой (для этого вводятся новые переменные $v'_{e1}value$,..., $v'_{ek}value$) и выполняется обработка процессов. Для каждого пути в программе порождается соответствующее условие корректности $P \longrightarrow inv(u)$.

Имена переменных, процессов и состояний процессов в Isabelle/HOL представляются именованными константами. Они кодируются последовательными натуральными числами, а затем для этих констант вводятся имена с помощью команды abbreviation. Использование этой команды для введения имен упрощает доказательства, так как не требует явно раскрывать определения.

Для обеспечения однозначности именования к именам переменных, объявленных в процессах, и состояний процессов добавляется имя процесса в качестве префикса. Каждая переменная v, объявленная на уровне программы, кодируется константой с именем v’. Каждый процесс p кодируется константой с именем p’. Каждая переменная v, объявленная в процессе p кодируется константой с именем p’v’, а состояние s процесса p— константой с именем p’s’. Выбор символа ‘ для разделения префиксов и имен переменных и состояний процессов основан на том, что данный символ не может использоваться в идентификаторах языка poST. Это позволяет обеспечить однозначное именование.

Для параметра условия корректности, соответствующего инварианту n-го цикла в состоянии s процесса p, используется имя p’s’loopinvn. Это позволяет однозначно определить, инварианту какого цикла соответствует этот параметр.

2.3 Реализация генератора условий корректности

Генератор условий корректности был реализован на языке Java с использованием фреймворка Xtext для синтаксического разбора текста программы на языке poST и работы с ее абстрактным синтаксическим деревом (AST). Стандартная архитектура проектов, основанных на Xtext, включает парсер и классы для хранения AST, генерируемые автоматически по описанию грамматики, генератор кода, компоненты семантического анализа и т. д. Xtext использует для внутреннего представления текстов программ EMFмодели[8]. Для правил грамматики Xtext создает соответствующие Java-классы, объекты которых представляют узлы в синтаксическом дереве и позволяют получить всю необходимую информацию о программе. Парсер принимает текст программы и создает ее AST в виде EMF модели. В случае успешного построения AST выполняется семантический анализ, а затем генерация кода.

Ядро языка poST было разработано в проекте [35] на основе фреймворка Xtext. Стандартные компоненты языка кроме кодогенератора (парсер, компоненты семантического анализа и т. д.) были взяты из этого проекта. В данной работе были созданы компонент генерации условий корректности, реализующий описанный ранее алгоритм, а также кодогенератор, который представляет собой генератор теории Isabelle/HOL, содержащей порожденные условия корректности. Компоненты генератора условий корректности и их взаимодействие приведены на рисунке 1.

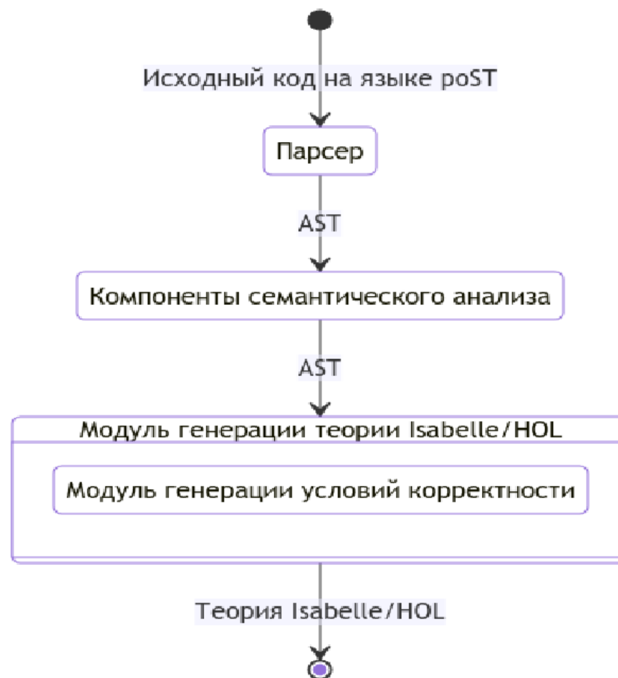

Рисунок 1 – Компоненты генератора условий корректности и их взаимодействие

Генератор условий корректности работает в режиме командной строки. На вход программе в качестве аргумента командной строки подается путь к файлу с расширением “.post”, содержащему исходный код программы на языке poST. Программа создает в текущей директории файл теории Isabelle/HOL, содержащей порожденные условия корректности. Пользовательский интерфейс программы приведен на рисунке 2. Руководство пользователя для генератора условий корректности приведено в приложении А.

```
C:\work\practice\turnstile>java -jar ..\vcgenerator.jar Turnstile.post
Code generation finished.
```

```
C:\work\practice\turnstile>java -jar ..\vcgenerator.jar Turnstile.post
Code generation aborted.
ERROR:Statement error: State can't be empty (Turnstile.post line : 37 column : 9)
WARNING:State should be LOOPED (Turnstile.post line : 37 column : 9)
WARNING:State is unreachable (Turnstile.post line : 41 column : 9)
```

Рисунок 2 – Пользовательский интерфейс генератора условий корректности

Для проверки отсутствия ошибок в poST-программе используется семантический анализатор, который был взят из проекта языка poST. В случае отсутствия ошибок выполняется генерация условий корректности. При наличии ошибок сообщения о них выдаются пользователю.

Для реализации генератора условий корректности были созданы классы для представления термов и формул, а также троек вида (P, u, st) представляющих пред- и постусловия. Для

работы алгоритма необходимо помнить контекст генератора условий корректности. Для этого был определен класс VCGeneratorState, хранящий следующую информацию:

- Типы переменных (сопоставление кодов имен переменных с их poST-типами). Для хранения информации о типах переменных используется ассоциативный массив (HashMap).
- Сопоставление имен переменных с их кодами для каждого процесса и для переменных, объявленных на уровне программы.
- Список программных констант.
- Значения констант (сопоставление кодов имен программных констант с их значениями).
- Константы, используемые в операторах таймаута (сопоставление кодов имен программных констант с их значениями, выраженными в количестве итераций цикла управления). Имена таких констант получаются из имен соответствующих констант, выражающих время в миллисекундах, добавлением постфикса TIMEOUT.
- Список кодов входных переменных.
- Списки инициализированных переменных для каждого процесса и для переменных, объявленных на уровне программы.
- Диапазоны индексов массивов
- Текущее состояние процесса (номер состояния в процессе)
- Список состояний для каждого процесса. Для каждого процесса хранится список имен состояний в порядке, в котором они определены в программе, и код каждого состояния.
- Период активации
- Счетчики для генерации кодов имен переменных, процессов и состояний процессов.
- Список порожденных условий корректности

Диаграмма классов модуля генерации условий корректности приведена на рисунке 3. В классы, представляющие выражения был добавлен метод generateExpression, реализующий описанную ранее функцию expr. Метод был объявлен в интерфейсе Expression, являющимся базовым для всех выражений, и переопределен для каждого класса выражений. Данный метод принимает в качестве параметров состояние изменений, в котором необходимо вычислить выражение, и контекст генератора условий корректности и возвращает терм, являющийся результатом символического вычисления выражения в соответствии с правилами для функции expr. Для вычисления сильнейшего постусловия в интерфейсе Statement также был метод, реализующий рассмотренную ранее функцию gen. Метод был переопределен в классах операторов. Данный метод принимает список предусловий для путей в программе, на которых выполняется данный оператор, и контекст генератора условий корректности и возвращает список вычисленных сильнейших постусловий.

Обработка процессов выполняется следующим образом. В контексте генератора данный процесс устанавливается в качестве текущего, текущее состояние процесса устанавливается

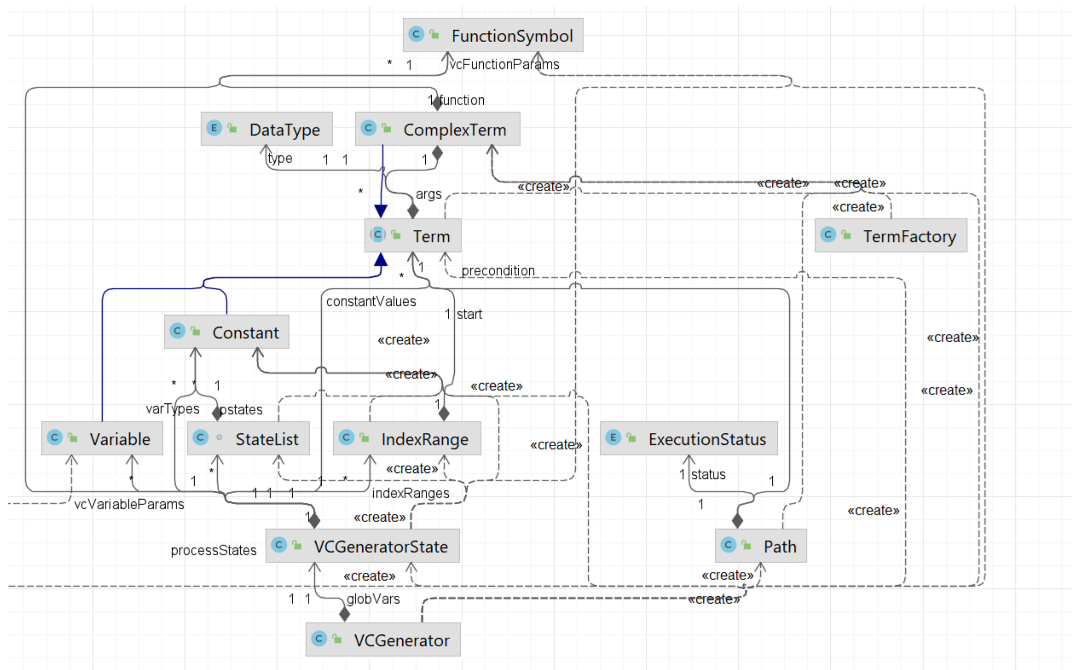

Рисунок 3 – Диаграмма классов модуля генерации условий корректности

в значение 0 и применяется описанный ранее алгоритм. После обработки каждого состояния процесса номер текущего состояния инкрементируется. При обработке программы сначала кодируются все переменные, объявленные на уроне программы, и процессы. Кодирование процесса помимо назначения ему кода также предполагает кодирование всех его переменных и состояний. Затем применяется алгоритм генерации для программы. Если входной файл содержит конфигурацию для poST-программы, то период активации извлекается из конфигурации. Иначе период активации задается в аргументах программы.

После выполнения генерации контекст генератора содержит список всех условий корректности и другую необходимую для создания теории Isabelle/HOL информацию: коды всех переменных, процессов и состояний процессов и программные константы.

Модуль генерации кода создает файл теории Isabelle/HOL с расширением “.thy”.Имя теории совпадает с именем входного файла программы на языке poST, если данное имя является допустимым именем теории Isabelle/HOL. Иначе выполняется кодирование имени. Теория содержит следующие элементы:

1. Определения констант, кодирующих имена переменных, процессов и состояний процессов.
2. Определения программных констант. Константы, имеющие в исходной программе тип данных TIME, в теории Isabelle/HOL представлены в миллисекундах.
3. Определения программных констант, используемых в операторах таймаута.
4. Порожденные условия корректности. Условия корректности могут иметь следующие параметры в указанном порядке:

а) Аннотация, соответствующая инварианту цикла управления

б) Аннотация, задающая ограничение на значения входных переменных программы

в) Аннотации, соответствующие циклам в тексте исходной программы. Порядок этих параметров соответствует порядку циклов исходной программе

г) Новые переменные для состояния системы управления и значений всех входных переменных исходной программы.

Каждое условие корректности имеет те и только те из этих параметров, которые используются в его определении.

Генератор условий корректности был протестирован на наборе модульных тестов для различных конструкций языка poST. Для тестирования генератора на целых программах был использован парсер из утилитных классов фреймворка Xtext для тестирования. В этих тестах синтаксический разбор текстов программ осуществлялся с помощью этого парсера. В тестах для конструкций нижнего уровня (выражений, операторов, состояний и процессов) соответствующие конструкции создавались программно.

3 Аппробация генератора условий корректности и доказательство условий корректности

3.1 Тестовый набор управляющих программ

Для апробации генератора условий корректности и разработки стратегий доказательства условий корректности был создан тестовый набор задач управления. Для каждой задачи была написана программа на языке poST и сформулирован набор требований. Требования были формализованы в виде аннотаций. Для каждой задачи были порождены и доказаны условия корректности.

Тестовый набор включает следующие 7 задач: ”Сушилка для рук ”Светофор на пешеходном переходе ”Вращающаяся дверь ”Эскалатор ”Турникет в метро ”Умный холодильник”и ”Термопот”.

Далее рассмотрим тестовую задачу управления турникетом в метро. Остальные задачи приведены в приложении Б.

В качестве управляемого объекта рассмотрим турникет 14. Турникет оснащен монетоприемником (coin acceptor), створками, управляемыми сигналом (open), которые блокируют проход и открываются после оплаты (paid). Оповещение проходящего о возможности прохода производится светодиодом (enter). Контроль открытия створок производится датчиком opened. Для контроля и упреждения попыток несанкционированного прохода турникет оснащен двумя оптическими датчиками, которые контролируют подход человека к турникету (PDin) и его проход через турникет (PDout). При получении оплаты монетоприемник блокируется. Разблокировка монетоприемника производится положительным фронтом сигнала reset.

Рисунок 4 – Турникет

Программа иправления турникетом на языке poST приведена в приложении В.

В программе объявлены входные переменные PdOut, paid и opened, выходные переменные open, reset и enter. и локальныя переменная passed. Переменная PdOut показывает, что пользователь прошел через турникет (сигнал датчика на выходе из турникета). Переменная paid показывает, выполнена ли оплата. Переменная opened показывает, что турникет открыт. Переменная open определяет сигнал открытия турникета. Переменная reset задает сигнал разблокировки монетоприемника. Переменная enter определяет, горит ли светодиод. В переменной passed запоминается проход пользователя.

В программе определены четыре процесса Init, Controller, EntranceController и Unlocker. Процесс Init имеет односостояние init, в котором он запускает процессы Controller и EntranceController и останавливается.

Процесс Controller обрабатывает сигналы PdOut и paid и в зависимости от них формирует сигнал open. Данный процесс имеет три состояния isClosed, minimalOpened и isOpened. В состоянии isClosed если выполнена оплата, турникет открывается, сбрасывается значение переменной passed и процесс переходит в состояние minimalOpened. В состоянии minimalOpened проход пользователя запоминается в переменной passed. В данном состоянии установлен таймаут, определяющий минимальное время, в течение которого должен быть открыт турникет. По истечении этого времени, если пользователь прошел, турникет закрывается и процесс переходит в состояние isClosed, иначе процесс переходит в состояние isOpened. В состоянии isOpened, если пользователь прошел, а также по таймауту, турникет закрывается и процесс переходит в состояние isClosed. Сумма временных интервалов в операторах таймаута в состояниях minimalOpened и isOpened определяет максимальное время, в течение которого может быть открыт турникет.

Процесс EntranceController обрабатывает входной сигнал opened и в зависимости от него формирует управляющие сигналы enter и reset. Процесс имеет два 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, подается сигнал разблокировки монетоприемника reset, запускается процесс Unlocker, контролирующий продолжительность сигнала reset, и процесс EntranceController переходит в состояние isClosed.

Процесс Unlocker имеет одно активное состояние unlock, тело которого состоит из оператора таймаута. По истечении 1 секунды значение переменной reset устанавливается в значение FALSE и процесс останавливается.

Период активации процессов равен 100 миллисекундам.

К данной программе предъявляются следующие требования:

1. После получения сигнала о проходе пользователя PdOut турникет должен быть закрыт не более, чем через 1 с.

2. Если турникет был закрыт и оплата не выполнена, то он не откроется, пока не будет выполнена оплата.
3. После появления с монетоприемника сигнала получения оплаты payed немедленно должен быть сформирован сигнал открытия турникета open.
4. Сигнал open должен быть в состоянии true не более 10 с.
5. Сигнал open должен быть в состоянии true не менее 1 с.
6. После появления сигнала opened и до его снятия должен гореть светодиод enter.
7. После запрета прохода должен быть разрешена работа монетоприемника

Для формализации требований используется только инвариант цикла, так как ограничений на значения входных переменных нет. Следующая формула является формальной аннотацией для первого требования:

$$\begin{aligned}
&\text{toEnvP}\ s \wedge \\
&(\forall s1 s2.\ \text{substate}\ s1\ s2 \wedge \text{substate}\ s2\ s \wedge \text{toEnvP}\ s1 \wedge \text{toEnvP}\ s2 \wedge \text{toEnvNum}\ s1\ s2 = 1 \wedge \\
&\text{toEnvNum}\ s2\ s {\geq} 10 - 1 \wedge \text{getVarBool}\ s1\ open' \wedge \text{getVarBool}\ s2\ PdOut' = True \longrightarrow \\
&(\exists s4.\ \text{toEnvP}\ s4 \wedge \text{substate}\ s2\ s4 \wedge \text{substate}\ s4\ s \wedge \\
&\text{toEnvNum}\ s2\ s4 \leq 10 - 1 \wedge \neg\, \text{getVarBool}\ s4\ open' \wedge \\
&(\forall s3.\ \text{toEnvP}\ s3 \wedge \text{substate}\ s2\ s3 \wedge \text{substate}\ s3\ s4 \wedge s3 \neq s4 \longrightarrow \text{getVarBool}\ s3\ open')))
\end{aligned}$$

3.2 Классификация требований

Выбор стратегии доказательства условий корректности зависит как от вида самого условия корректности, так и от класса требования, для которого оно доказывается. В [28] был представлен набор шаблонов требований к процесс-ориентированным программам, однако приведенная в той работе классификация покрывает не все требования к рассмотренному ранее тестовому набору задач. Поэтому был введен новый шаблон требований. Рассмотрим далее классы требований к процесс-ориентированным программам.

Первый шаблон определяет требования, утверждающие, что события происходят в одной точке цикла управления. Соответствующий шаблон p_1 имеет следующий вид:

$$p_1(s, vc_1, vc_2) \equiv \text{toEnvP}\ s \wedge (\forall s_1.\ \text{substate}\ s_1\ s \wedge \text{toEnvP}\ s_1 \wedge vc_1(s_1) \longrightarrow vc_2(s_1)),$$

где vc_1 и vc_2 описывают соответствующие события. Требование специфицирует, что если формула vc_1 истинна в некоторой точке, то и формула vc_2 также истинна в этой точке. К этому классу относится 6-е требование к программе управления турникетом. Формальная аннотация для этого требования имеет вид:

$$\text{toEnvP}\ s \wedge (\forall s1.\ \text{substate}\ s1\ s \wedge \text{toEnvP}\ s1 \wedge \text{getVarBool}\ s1\ opened' \longrightarrow \text{getVarBool}\ s1\ enter'),$$

где $vc_1(s1) \equiv \text{getVarBool}\ s1\ opened'$, $vc_2(s1) \equiv \text{getVarBool}\ s1\ enter'$.

Второй шаблон определяет требования, утверждающие, что после одной итерации цикла произойдет определенное событие. Этот шаблон имеет следующий вид:

$$\begin{aligned}
&p_2(s, vc_1, vc_2) \equiv \\
&\text{toEnvP}\, s \wedge \\
&(\forall s_1 s_2.\, \text{substate}\, s_1\, s_2 \wedge \text{substate}\, s_2\, s \wedge \text{toEnvP}\, s_1 \wedge \\
&\text{toEnvP}\, s_2 \wedge \text{toEnvNum}\, s_1\, s_2 = 1 \wedge vc_1(s_1, s_2) \longrightarrow vc_2(s_2)),
\end{aligned}$$

где vc_1 определяет ограничения на значения переменных в начале итерации, vc_2 описывает событие, которое должно произойти после итерации. Этому классу принадлежит 2-е требование к программе управления турникетом. Формальная аннотация для этого требования имеет вид:

$$\begin{aligned}
&\text{toEnvP}\, s \wedge \\
&(\forall s1 s2.\, \text{substate}\, s1\, s2 \wedge \text{substate}\, s2\, s \wedge \text{toEnvP}\, s1 \wedge \text{toEnvP}\, s2 \wedge \text{toEnvNum}\, s1\, s2 = 1 \wedge \\
&\neg\, \text{getVarBool}\, s1\, \mathit{open}' \wedge \neg\, \text{getVarBool}\, s2\, \mathit{paid}' \longrightarrow \neg\, \text{getVarBool}\, s2\, \mathit{open}'),
\end{aligned}$$

где $vc_1(s1, s2) \equiv \neg\, \text{getVarBool}\, s1\, \mathit{open}' \wedge \neg\, \text{getVarBool}\, s2\, \mathit{paid}'$, $vc_2(s2) \equiv \neg\, \text{getVarBool}\, s2\, \mathit{open}'$

Третий шаблон определяет требования, утверждающие, что если произошло событие E_1, то событие E_2 может произойти только не менее, чем через τ. Формула, описывающая событие E_1, определяет ограничения на значения переменных в двух состояниях изменений, время между которыми составляет одну итерацию цикла управления. Шаблон требований этого класса имеет вид:

$$\begin{aligned}
&p_3(s, , vc_1, vc_2) \equiv \\
&\text{toEnvP}\, s \wedge \\
&(\forall s_1 s_2 s_3.\, \text{substate}\, s_1\, s_2 \wedge \text{substate}\, s_2\, s_3 \wedge \text{substate}\, s_3\, s \wedge \text{toEnvP}\, s_1 \wedge \text{toEnvP}\, s_2 \wedge \\
&\text{toEnvP}\, s_3 \wedge \text{toEnvNum}\, s_1\, s_2 = 1 \wedge \text{toEnvNum}\, s_2\, s_3 < \tau \wedge vc_1(s_1, s_2) \longrightarrow vc_2(s_3),
\end{aligned}$$

где vc_1 описывает событие E_1, vc_2 описывает событие E_2. Этому классу принадлежит 5-е требование программе управления турникетом. Формальная аннотация для этого требования имеет вид:

$$\begin{aligned}
&\text{toEnvP}\, s \wedge \\
&(\forall s1 s2 s3.\, \text{substate}\, s1\, s2 \wedge \text{substate}\, s2\, s3 \wedge \text{substate}\, s3\, s \wedge toEnvPs1 \wedge \\
&\text{toEnvP}\, s2 \wedge \text{toEnvP}\, s3 \wedge \text{toEnvNum}\, s1\, s2 = 1 \wedge \text{toEnvNum}\, s2\, s3 < 10 \wedge \quad), \\
&\neg\, \text{getVarBool}\, s1\, \mathit{open}' \wedge \text{getVarBool}\, s2\, \mathit{open}' \longrightarrow \text{getVarBool}\, s3\, \mathit{open}'
\end{aligned}$$

где $vc_1(s1, s2) \equiv \neg\, \text{getVarBool}\, s1\, \mathit{open}' \wedge \text{getVarBool}\, s2\, \mathit{open}'$, $vc_2(s3) \equiv \text{getVarBool}\, s3\, \mathit{open}'$, $\tau \equiv 10$.

К четвертому классу относятся требования, утверждающие, что не более, чем через τ после определенного события E_1 должно произойти событие E2. Этот шаблон имеет следующий

вид

$$p_4(s, , vc_1, vc_2, vc_3) \equiv$$
$$\text{toEnvP}\ s \wedge$$
$$(\forall s_1.\ \text{substate}\ s_1\ s \wedge \text{toEnvP}\ s_1 \wedge \text{toEnvNum}\ s_1\ s \geq \tau \wedge vc1(s_1) \longrightarrow$$
$$\exists s_3.\ \text{toEnvP}\ s_3 \wedge \text{substate}\ s_1\ s_3 \wedge \text{substate}\ s_3\ s \wedge \text{toEnvNum}\ s_1\ s_3 \leq \tau \wedge vc_2(s_3) \wedge$$
$$\forall s_2.(\text{toEnvP}\ s_2 \wedge \text{substate}\ s_1\ s_2 \wedge \text{substate}\ s_2\ s3 \wedge s_2 \neq s_3 \longrightarrow vc_3(s_2))),$$

где формула vc_1 описывает событие E_1, vc_2 описывает событие E_2, vc_3 определяет ограничение на значения переменных после события E_1 до события E_2. Этому классу принадлежит 4-е требование к программе управления турникетом. Формальная аннотация для этого требования имеет вид:

$$\text{toEnvP}\ s \wedge$$
$$(\forall s1.\ \text{substate}\ s1\ s \wedge \text{toEnvP}\ s1 \wedge \text{toEnvNum}\ s1\ s \geq 100 \wedge \text{getVarBool}\ s1\ open' \longrightarrow$$
$$(\exists s3.\ \text{toEnvP}\ s3 \wedge \text{substate}\ s1\ s3 \wedge \text{substate}\ s3\ s \wedge \text{toEnvNum}\ s1\ s3 \leq 100 \wedge \neg\, \text{getVarBool}\ s3\ open' \wedge$$
$$(\forall s2.\ \text{toEnvP}\ s2 \wedge \text{substate}\ s1\ s2 \wedge \text{substate}\ s2\ s3 \wedge s2 \neq s3 \longrightarrow \text{getVarBool}\ s2\ open'))),$$

где $vc_1(s1) \equiv \text{getVarBool}\ s1\ open'$, $vc_2(s3) \equiv \neg\, \text{getVarBool}\ s3\ open'$, $vc_3(s2) \equiv \text{getVarBool}\ s2\ open'$, $\tau \equiv OPEN_DOOR_TIME_LIMIT'TIMEOUT$.

Шаблон пятого класса требований являются комбинацией шаблонов требований классов 2 и 4. Если произошло событие E_1, то не более, чем через τ должно произойти событие E_2. При этом формула, описывающая E_1, определяет ограничение на значения переменных в двух состояниях изменени, время между которыми составляет одну итерацию цикла. Шаблон p_5 имеет следующий вид:

$$p_5(s, , vc_1, vc_2, vc_3) \equiv$$
$$\text{toEnvP}\ s \wedge$$
$$(\forall s_1 s_2.\ \text{substate}\ s_1\ s_2 \wedge \text{substate}\ s_2\ s \wedge \text{toEnvP}\ s_1 \wedge \text{toEnvP}\ s_2 \wedge \text{toEnvNum}\ s_1\ s_2 = 1 \wedge$$
$$\text{toEnvNum}\ s_2\ s \geq \tau \wedge vc_1(s_1, s_2) \longrightarrow$$
$$\exists s_4.\ \text{toEnvP}\ s_4 \wedge \text{substate}\ s_2\ s_4 \wedge \text{substate}\ s4\ s \wedge \text{toEnvNum}\ s_2\ s_4 \leq \tau \wedge vc_2(s_4) \wedge$$
$$\forall s_3.(\text{toEnvP}\ s_3 \wedge \text{substate}\ s2\ s3 \wedge \text{substate}\ s_3\ s_4 \wedge s_3 \neq s_4 \longrightarrow vc_3(s_3))),$$

где формула vc_1 описывает событие E_1, vc_2 описывает событие E_2, vc_3 определяет ограничение на значения переменных после события E_1 до события E_2. К этому классу относится ранее рассмотренное первое требование к турникету, где $vc_1(s1, s2) \equiv \text{getVarBool}\ s1\ open' \wedge \text{getVarBool}\ s2\ PdOut' = True$, $vc_2(s4) \equiv \neg\, \text{getVarBool}\ s4\ open'$, $vc_3(s3) \equiv \text{getVarBool}\ s3\ open'$

Далее при описании дополнительного инварианта и стратегий доказательства условий корректности введенные в этом подразделе обозначения vc_1, vc_2, vc_3 используются для ссылок на соответствующие подформулы в требованиях. Также используются имена s_1, s_2, s_3 и т. д. для в качестве ссылок на соотвествующие переменные, связанные кванторами всеобщности в требованиях.

3.3 Дополнительный инвариант

Инвариант цикла управления, специфицирующий требования к программе содержат не достаточно информации для доказательства. Для доказательства условий корректности в инвариант необходимо включить дополнительные свойства программы, связывающие значения входных и выходных переменных с состояниями процессов, таймерами и значениями локальных переменных, не используемых в требованиях. Поэтому инвариант цикла управления для каждого требования имеет вид $r \wedge ei$, где r специфицирует требование к программе, а ei — дополнительный инвариант, имеющий вид $toEnvPs \wedge p_1 \wedge \ldots \wedge p_n$, где s — параметр, обозначающий состояние изменений, в котором должен выполняться инвариант, $p_1,\ldots,\ p_n$ — дополнительные свойства, зависящие от s. Рассмотрим, как формулируются дополнительные свойства для программ из рассматриваемого тестового набора.

В данной работе используются два подхода к определению вспомогательных свойств. Некоторые свойства определяются на основе программы без учета требований. Другие получаются усилением требований. На основе программы определялись следующие свойства:

1. Если в некотором состоянии процесса присутствует оператор таймаута и по таймауту процесс переходит в другое состояние или сбрасывается таймер, то значение таймера процесса в этом состоянии не превышает времени, указанного в операторе таймаута и выраженного в количестве итераций цикла управления. Шаблон данного свойства имеет вид $\forall s_1.\, \text{toEnvP}\, s_1 \wedge \text{substate}\, s_1\, s \wedge \text{getPstate}\, s_1\, p = q \longrightarrow \text{ltime}\, s_1\, p \leq t$, где p — некоторый процесс, q — состояние процесса p, содержащее оператор таймаута, t — время в операторе таймаута, выраженное в количестве итераций цикла управления.
2. Если некоторая переменная v используется только в процессе p, при переходе процесса p в состояние q переменная v имела значение v_0 или ей было присвоено данное значение и значение v не изменяется, пока процесс p не переходит в другое состояние, то, когда p находится в состоянии q, значение v равно v_0. Если q — начальное состояние процесса p или q= STOP и p не является начальным процессом, также требуется, чтобы v_0 было начальным значением переменной v. Шаблон данного свойства имеет вид $\forall s_1.\, \text{toEnvP}\, s_1 \wedge \text{substate}\, s_1\, s \wedge \text{getPstate}\, s_1\, p = q \longrightarrow \text{getVar}\, s_1\, v = v_0$.
3. Если первый процесс программы, находясь в начальном состоянии, выполняет переход в другое состояние или останавливается или другой процесс останавливает его и в дальнейшем первый процесс не переходит в начальное состояние и другие процессы не запускают его, то начальный процесс находится в начальном состоянии только перед первой итерацией цикла управления. Шаблон данного свойстава имеет вид $\forall s_1.\, \text{toEnvP}\, s_1 \wedge \text{substate}\, s_1\, s \longrightarrow (\text{getPstate}\, s_1\, p = q \longleftrightarrow \text{toEnvNum}\, emptyState\, s_1 = 1)$, где p — первый процесс программы, q — его начальное состояние.

4. Перед первой итерацией цикла управления все процессы кроме первого находятся в состоянии STOP. Шаблон данного свойства имеет вид $\forall s_1.\, \mathrm{toEnvP}\, s_1 \,\wedge\, \mathrm{substate}\, s_1\, s \,\wedge\, toEnvNumemptyStates_1 = 1 \longrightarrow \mathrm{getPstate}\, s_1\, p = STOP$, где p не является начальным процессом.
5. Каждый процесс всегда находится в одном из состояний, определенных для него в тексте программы, или в состоянии STOP, если процесс никогда не переходит в состояние ERROR. Шаблон данного свойства имеет вид $\forall s_1.\, \mathrm{toEnvP}\, s_1 \,\wedge\, \mathrm{substate}\, s_1\, s \longrightarrow \mathrm{getPstate}\, s_1\, p = q_1 \vee \ldots \vee \mathrm{getPstate}\, s_1\, p = q_n$, где q_1,..., q_n — состояния, в которых может находиться процесс p.

Свойства 3 и 4 необходимы для доказательства условий корректности в задачах «Турникет в метро» и «Холодильник», соответствующих путям в программах, на которых процесс Init запускает другие процессы и завершается. Свойство 5 использовалось для доказательства условий корректности в случае, когда процессы находятся в состоянии ERROR.

Далее рассмотрим используемые методы усиления требований. В некоторых случаях условия корректности не могут быть доказаны с помощью описанных далее стратегий доказательства. Это может означать как то, что требование не выполняется для программы, так и то, что инвариант не является достаточно сильным. После применения стратегий разбора случаев и упрощения условия корректности в некоторых случаях могут оставаться недоказанные утверждения. Используя эти утверждения могут быть сформированы предполагаемые свойства программы, выражающие связь между входными и выходными переменными и состояниями процессов и, возможно, локальными переменными, которые позволяют доказать ранее недоказанные условия корректности. Эти свойства добавляются в дополнительный инвариант. Подход к определению дополнительного инварианта на основе усиления требований позволяет упростить доказательство по сравнению с подходом, в котором на основе программы независимо от требований определяются свойства, необходимые для доказательства условий корректности, а затем эти свойства используются для доказательства, что требует применения более сложных стратегий доказательства.

Опишем подход к усилению требований классов 3, 4 и 5. Для усиления таких требований в дополнительный инвариант добавляются свойства для состояний процесса p, в котором используются переменные, входящие в требование. Эти свойства получаются из требования добавлением в качестве посылки утверждения о состоянии процесса и заменой константы, задающей время, другой константой или термом. В случае, когда состояние q процесса p содержит оператор таймаута, выполняется замена константы термом вида c + ltime s p, где c — некоторая константа, s — состояние изменений, для которого утверждается, что процесс p находится в состоянии q. Для доказательства требований класса 3 в дополнительный инвариант

необходимо добавить свойство вида

$$\forall s_3.\, \mathrm{toEnvP}\, s_3 \wedge \mathrm{substate}\, s_3\, s \wedge \mathrm{getPstate}\, s_3\, p = q \wedge \mathrm{ltime}\, s_3\, p \geq t \longrightarrow$$
$$(\forall s_1 s_2.\, \mathrm{toEnvP}\, s_1 \wedge \mathrm{toEnvP}\, s_2 \wedge \mathrm{substate}\, s_1\, s_2 \wedge \mathrm{substate}\, s_2\, s_3 \wedge$$
$$\mathrm{toEnvNum}\, s_1\, s_2 = 1 \wedge \mathrm{toEnvNum}\, s_2\, s_3 + 1 \leq \tau \longrightarrow \neg vc_1(s_1, s_2)$$

для состояния q такого, что, когда процесс рпереходит из q в другое состояние, формула vc_2 в заключении требования становится ложной и, следовательно, необходимо, чтобы посылка также была ложной. При этом, если переход в другое состояние осуществляется по таймауту, t является временем, заданным в операторе таймаута, выраженным в количестве итераций цикла управления. Усиленное свойство имеет вид:

$$\forall s_3.\, \mathrm{toEnvP}\, s_3 \wedge \mathrm{substate}\, s_3\, s \wedge \mathrm{getPstate}\, s_3\, p = q \longrightarrow$$
$$(\forall s_1 s_2.\, \mathrm{toEnvP}\, s_1 \wedge \mathrm{toEnvP}\, s_2 \wedge \mathrm{substate}\, s_1\, s_2 \wedge \mathrm{substate}\, s_2\, s_3 \wedge \tag{1}$$
$$\mathrm{toEnvNum}\, s_1\, s_2 = 1 \wedge \mathrm{toEnvNum}\, s_2\, s_3 + 1 \leq c + \mathrm{ltime}\, s_3\, p \longrightarrow \neg vc_1(s_1, s_2)$$

где константа c находится из равенства c + t = τ.

Рассмотрим теперь усиление требований класса 4. Так как условие $toEnvNums_1s \geq \tau$ в посылке требования не позволяет использовать истинность требования для предыдущей итерации для доказательства его истинности после выполнения текущей итерации в случае $toEnvNums_1s = \tau$. Поэтому требование преобразуется к виду

$$\forall s_4.\, \mathrm{toEnvP}\, s_4 \wedge \mathrm{substate}\, s_4\, s \longrightarrow$$
$$(\forall s_1.\, \mathrm{toEnvP}\, s_1 \wedge \mathrm{substate}\, s_1\, s_4 \wedge vc_1(s_1) \longrightarrow$$
$$(\exists s_3.\, \mathrm{toEnvP}\, s_3 \wedge \mathrm{substate}\, s_1\, s_3 \wedge \mathrm{substate}\, s_3\, s_4 \wedge \mathrm{toEnvNum}\, s_1\, s_3 \leq \tau \wedge vc_2(s_3) \wedge \tag{2}$$
$$(\forall s_2.\, \mathrm{toEnvP}\, s_2 \wedge \mathrm{substate}\, s_1\, s_2 \wedge \mathrm{substate}\, s_2\, s_3 \wedge s_2 \neq s_3 \longrightarrow vc_3(s_2))) \vee$$
$$\mathrm{toEnvNum}\, s_1\, s_4 < \tau \wedge (\forall s_2.\, \mathrm{toEnvP}\, s_2 \wedge \mathrm{substate}\, s_1\, s_2 \wedge \mathrm{substate}\, s_2\, s_4 \longrightarrow vc_3(s_2)))$$

Данное свойство обеспечивает истинность требования, так как при условии $\mathrm{toEnvNum}\, s1\, s \geq \tau$ второй дизъюнкт ложен и, следовательно, выполняется первый. Утверждение $(\forall s_2.\, \mathrm{toEnvP}\, s_2 \wedge \mathrm{substate}\, s_1\, s_2 \wedge \mathrm{substate}\, s_2\, s_4 \longrightarrow vc_3(s_2))$ во втором дизъюнкте необходимо, так как согласно требованию пока не произошло событие, описываемое формулой vc_2, должно выполняться vc_3. Для усиления данного свойства подобно тому, как это выполняется для требований третьего класса, для некоторых состояний процессов необходимо заменить константу термом, зависящим от значения таймера процесса.

Усиление свойства 2 выполняется следующим образом. Если в некотором состоянии q процесса p всегда выполняется условие vc_2, то второй дизъюнкт может быть устранен. В данном

случае свойство имеет вид:

$$\begin{aligned}
&\forall s_4.\, \text{toEnvP}\, s_4 \land \text{substate}\, s_4\, s \land \text{getPstate}\, s_4\, p = q \longrightarrow \\
&(\forall s_1.\, \text{toEnvP}\, s_1 \land \text{substate}\, s_1\, s_4 \land vc_1(s_1) \longrightarrow \\
&(\exists s_3.\, \text{toEnvP}\, s_3 \land \text{substate}\, s_1\, s_3 \land \text{substate}\, s_3\, s_4 \land \text{toEnvNum}\, s_1\, s_3 \leq \tau \land vc_2(s_3)\land \\
&(\forall s_2.\, \text{toEnvP}\, s_2 \land \text{substate}\, s_1\, s_2 \land \text{substate}\, s_2\, s_3 \land s_2 \neq s_3 \longrightarrow vc_3(s_2))))
\end{aligned} \tag{3}$$

В случае, когда состояние q процесса p содержит оператор таймаута, константа заменяется термом вида c + ltime s p и усиленное свойство имеет вид:

$$\begin{aligned}
&\forall s_4.\, \text{toEnvP}\, s_4 \land \text{substate}\, s_4\, s \longrightarrow \\
&(\forall s_1.\, \text{toEnvP}\, s_1 \land \text{substate}\, s_1\, s_4 \land vc_1(s_1) \longrightarrow \\
&(\exists s_3.\, \text{toEnvP}\, s_3 \land \text{substate}\, s_1\, s_3 \land \text{substate}\, s_3\, s_4 \land \text{toEnvNum}\, s_1\, s_3 \leq \tau \land vc_2(s_3)\land \\
&(\forall s_2.\, \text{toEnvP}\, s_2 \land \text{substate}\, s_1\, s_2 \land \text{substate}\, s_2\, s_3 \land s_2 \neq s_3 \longrightarrow vc_3(s_2)))\lor \\
&\text{toEnvNum}\, s_1\, s_4 < c + \text{ltime}\, s_4\, p \land (\forall s_2.\, \text{toEnvP}\, s_2 \land \text{substate}\, s_1\, s_2 \land \text{substate}\, s_2\, s_4 \longrightarrow vc_3(s_2))),
\end{aligned} \tag{4}$$

где константа c находится из равенства $c + t_{timeout} =$, $t_{timeout}$ — время в операторе таймаута, выраженное в количестве итераций цикла управления.

В случае, когда в состоянии q отсутствует оператор таймаута или $t_{timeout} > \tau$, второй дизъюнкт в заключении формулы 2 также устраняется. Усиленное свойство представляется в виде формулы 3.

Рассмотрим определение дополнительного инварианта на примере программы управления . Дополнительный инвариант для этой задачи является конъюнкцией следующих свойств:

1) $(\forall s1.\, \text{toEnvP}\, s1 \land \text{substate}\, s1\, s \land \text{getPstate}\, s1\, Controller' = Controller'minimalOpened' \longrightarrow \text{ltime}\, s1\, Controller' \leq 10)$;
2) $(\forall s1.\, \text{toEnvP}\, s1 \land \text{substate}\, s1\, s \land \text{getPstate}\, s1\, Controller' = Controller'isOpened' \longrightarrow \text{ltime}\, s1\, Controller' \leq 90$;
3) $(\forall s1.\, \text{toEnvP}\, s1 \land \text{substate}\, s1\, s \land (\text{getPstate}\, s1\, Controller' = Controller'isClosed' \lor \{getPstates1Controller' = STOP) \longrightarrow \neg\, \text{getVarBool}\, s1\, open')$;
4) $(\forall s1.\, \text{toEnvP}\, s1 \land \text{substate}\, s1\, s \land (\text{getPstate}\, s1\, Controller' = Controller'minimalOpened' \lor \text{getPstate}\, s1\, Controller' = Controller'isOpened') \longrightarrow \text{getVarBool}\, s1\, open')$;
5) $(\forall s1.\, \text{toEnvP}\, s1 \land \text{substate}\, s1\, s \land \text{getPstate}\, s1\, EntranceController' = EntranceController'isClosed' \longrightarrow \neg getVarBools1enter')$;
6) $(\forall s1.\, \text{toEnvP}\, s1 \land \text{substate}\, s1\, s \land \text{getPstate}\, s1\, EntranceController' = EntranceController'isOpened' \longrightarrow \text{getVarBool}\, s1\, enter')$;
7) $(\forall s1.\, \text{toEnvP}\, s1 \land \text{substate}\, s1\, s \longrightarrow \text{getPstate}\, s1\, Controller' \in Controller'isClosed', Controller'minimalOpened', Controller'isOpened', STOP)$;

8) $(\forall s1.\, \mathrm{toEnvP}\, s1 \wedge \mathrm{substate}\, s1\, s \longrightarrow \mathrm{getPstate}\, s1\, EntranceController' \in EntranceController'isClosed', EntranceController'isOpened', STOP)$;
9) $(\forall s4.\, \mathrm{toEnvP}\, s \wedge \mathrm{substate}\, s4\, s \wedge \mathrm{getPstate}\, s4\, Controller' = Controller'isClosed' \longrightarrow (\forall s1.\, \mathrm{toEnvP}\, s1 \wedge \mathrm{substate}\, s1\, s \wedge \mathrm{getVarBool}\, s1\, open' \longrightarrow (\exists s3.\, \mathrm{toEnvP}\, s3 \wedge \mathrm{substate}\, s1\, s3 \wedge \mathrm{substate}\, s3\, s4 \wedge \mathrm{toEnvNum}\, s1\, s3 \leq 100 \wedge \neg \mathrm{getVarBool}\, s3\, open' \wedge (\forall s2.\, \mathrm{toEnvP}\, s2 \wedge \mathrm{substate}\, s1\, s2 \wedge \mathrm{substate}\, s2\, s3 \wedge s2 \neq s3 \longrightarrow \mathrm{getVarBool}\, s2\, open'))))$;
10) $(\forall s4.\, \mathrm{toEnvP}\, s \wedge \mathrm{substate}\, s4\, s \wedge \mathrm{getPstate}\, s4\, Controller' = Controller'isOpened' \longrightarrow (\forall s1.\, \mathrm{toEnvP}\, s1 \wedge \mathrm{substate}\, s1\, s \wedge \mathrm{getVarBool}\, s1\, open' \longrightarrow (\exists s3.\, \mathrm{toEnvP}\, s3 \wedge \mathrm{substate}\, s1\, s3 \wedge \mathrm{substate}\, s3\, s4 \wedge \mathrm{toEnvNum}\, s1\, s3 \leq 100 \wedge \neg \mathrm{getVarBool}\, s3\, open' \wedge (\forall s2.\, \mathrm{toEnvP}\, s2 \wedge \mathrm{substate}\, s1\, s2 \wedge \mathrm{substate}\, s2\, s3 \wedge s2 \neq s3 \longrightarrow \mathrm{getVarBool}\, s2\, open')) \vee \mathrm{toEnvNum}\, s1\, s4 < 10 + \mathrm{ltime}\, s4\, Controller' \wedge (\forall s2.\, \mathrm{toEnvP}\, s2 \wedge \mathrm{substate}\, s1\, s2 \wedge \mathrm{substate}\, s2\, s4 \longrightarrow \mathrm{getVarBool}\, s2\, open')))$;
11) $(\forall s4.\, \mathrm{toEnvP}\, s \wedge \mathrm{substate}\, s4\, s \wedge \mathrm{getPstate}\, s4\, Controller' = Controller'minimalOpened' \longrightarrow (\forall s1.\, \mathrm{toEnvP}\, s1 \wedge \mathrm{substate}\, s1\, s \wedge \mathrm{getVarBool}\, s1\, open' \longrightarrow (\exists s3.\, \mathrm{toEnvP}\, s3 \wedge \mathrm{substate}\, s1\, s3 \wedge \mathrm{substate}\, s3\, s4 \wedge \mathrm{toEnvNum}\, s1\, s3 \leq 100 \wedge \neg \mathrm{getVarBool}\, s3\, open' \wedge (\forall s2.\, \mathrm{toEnvP}\, s2 \wedge \mathrm{substate}\, s1\, s2 \wedge \mathrm{substate}\, s2\, s3 \wedge s2 \neq s3 \longrightarrow \mathrm{getVarBool}\, s2\, open')) \vee \mathrm{toEnvNum}\, s1\, s4 < \mathrm{ltime}\, s4\, Controller' \wedge (\forall s2.\, \mathrm{toEnvP}\, s2 \wedge \mathrm{substate}\, s1\, s2 \wedge \mathrm{substate}\, s2\, s4 \longrightarrow \mathrm{getVarBool}\, s2\, open')))$;
12) $(\forall s3.\, \mathrm{toEnvP}\, s3 \wedge \mathrm{substate}\, s3\, s \wedge \mathrm{getPstate}\, s3\, Controller' = Controller'minimalOpened' \longrightarrow (\forall s1 s2.\, \mathrm{toEnvP}\, s1 \wedge \mathrm{toEnvP}\, s2 \wedge \mathrm{substate}\, s1\, s2 \wedge \mathrm{substate}\, s2\, s3 \wedge \mathrm{toEnvNum}\, s1\, s2 = 1 \wedge \mathrm{toEnvNum}\, s2\, s3 + 1 < \mathrm{ltime}\, s3\, Controller' \longrightarrow \neg(\neg \mathrm{getVarBool}\, s1\, open' \wedge \mathrm{getVarBool}\, s2\, open')))$;
13) $(\forall s3.\, \mathrm{toEnvP}\, s3 \wedge \mathrm{substate}\, s3\, s \wedge \mathrm{getPstate}\, s3\, Controller' = Controller'isOpened' \longrightarrow (\forall s1 s2.\, \mathrm{toEnvP}\, s1 \wedge \mathrm{toEnvP}\, s2 \wedge \mathrm{substate}\, s1\, s2 \wedge \mathrm{substate}\, s2\, s3 \wedge \mathrm{toEnvNum}\, s1\, s2 = 1 \wedge \mathrm{toEnvNum}\, s2\, s3 + 1 < 10 \longrightarrow \neg(\neg \mathrm{getVarBool}\, s1\, open' \wedge \mathrm{getVarBool}\, s2\, open')))$.

Здесь первые два свойства являются утверждениями о значении таймера процесса $Controller$ в состояниях $minimalOpened$ и $isOpened$ соответственно. Свойства 3-6 специфицируют значения переменных в различных состояниях процессов. Свойства 7 и 8 определяет, в каких состояниях могут находиться процессы. Свойства 9-11 являются усилением требования 4 для состояний $isClosed$, $isOpened$ и $minimalOpened$ процесса $Controller$ соответственно. Свойства 12 и 13 являются усилением требования для состояний $minimalOpened$ и $isOpened$ соответственно.

3.4 Доказательство условий корректности

Для доказательства условий корректности в Isabelle/HOL была определена теория, содержащая леммы, выражающие свойства функций для типа данных state, необходимые для доказательства условий корректности. Приведем некоторые из этих лемм.

- substate_refl: “$substate\ s1\ s1$” — рефлексивность substate.
- substate_trans: “$substate\ s1\ s2 \implies substate s2 s3 \implies substate s1 s3$” — транзитивность substate.
- substate_antisym: “$substate\ s1\ s2 \implies substate\ s2\ s1 \implies s1 = s2$” — антисимметричность substate.
- substate_toEnvNum_id: ”$substate\ s1\ s2 \wedge toEnvNum\ s1\ s2 = 0 \wedge toEnvP\ s1 \wedge toEnvP\ s2 \implies s1 = s2$— если время между двумя состояниями, удовлетворяющими предикату toEnvP (т. е. состояниями в точке выхода из цикла) равно нулю, то эти состояния равны.
- toEnvP_imp_gtime_gt_0: ”$toEnvPs \implies toEnvNumemptyStates > 0$— время между начальным состоянием системы управления и любым состоянием между итерациями цикла управления больше нуля.

Для доказательства этих лемм используются разбор случаев и индукция по переменным типа данных state, а также ранее доказанные леммы. Например, лемма substate_trans доказывается следующим образом (Листинг 1):

```
1    lemma substate_trans:
2  "substate s1 s2 ⇒ substate s2 s3 ⇒ substate s1 s3"
3    apply((induction s3); (simp split: if_splits))
4    done
```

Листинг 1 – Доказательство леммы substate_trans

Здесь выполняется индукция по переменной s3, а затем все подцели доказываются с помощью упрощения методом simp.

Более сложное доказательство имеет лемма substate_antisym. Рассмотрим ее доказательство (Листинг 2).

```
1    lemma substate_antisym:
2   "substate s1 s2 ⇒ substate s2 s1 ⇒ s1=s2"
3    apply((induction s2 arbitrary: s1); (metis substate.simps substate_refl
    substate_trans))
4    done
```

Листинг 2 – Доказательство леммы substate_antisym

Здесь перед выполнением индукции по s2 выполняется обобщение по s1. Дальнейшее доказательство не может быть выполнено с помощью методов, таких как simp и auto. Но с помощью sledgehammer для каждой подцели было найдено доказательство с помощью metis,

которому в качестве аргументов передается соответствующее правило упрощения, определяемое функцией substate, а также ранее доказанные леммы substate_refl и substate_trans.

Доказательство условий корректности в Isabelle//HOL выполняется в виде теорем. Для условий корректности, соответствующих пути инициализации программы, теоремы имеют вид: theorem “VC inv s0”, для остальных условий корректности теоремы имеют вид: theorem “VC inv env s0 v1_value …vn_value”, где VC — условие корректности, inv — конъюнкция инварианта цикла управления RInv, являющегося формальной аннотацией для требования к программе, и дополнительного инварианта extraInv, s0 — символическая переменная, обозначающая состояние изменений, v1_value,…, vn_value —символические переменные, соответствующие значениям входных переменных, установленным средой.

Определения условий корректности, требований и инвариантов в Isabelle/HOL вводятся с помощью команды definition и в доказательстве должны быть раскрыты методом unfold. Некоторые условия корректности могут быть доказаны в результате упрощения. К ним относятся, в частности, условия корректности, соответствующие пути в программе, на котором выполняется сброс таймера процесса, а затем срабатывает оператор таймаута в том же процессе, или запускается процесс, определенный в тексте программы после текущего и запущенный процесс находится не в начальном состоянии. Такие пути исполнения программы невозможны, и следовательно, условия корректности для них содержат ложные посылки. Также упрощение позволяет доказать условия корректности, соответствующие пути инициализации программы. Для упрощения условий корректности в Isabelle/HOL использовался метод simp.

Другие условия корректности не могут быть полностью доказаны в результате упрощения и требуют разработки специальных стратегий для их доказательства. Подход к доказательству таких условий корректности определяется прежде всего классом требования, для которого они доказываются. Пусть условие корректности VC имеет вид: $inv(s_0) \wedge env(s')\square P \longrightarrow inv(s)$, где s — состояние изменений после выполнения итерации цикла управления, s_0 — состояние изменений до выполнения этой итерации. Отметим, что все состояния изменений, используемые в требованиях, являются состояниями между итерациями цикла. Поэтому если для некоторого состояния s_1 выполнено условие $substates_1 s$ и $s_1 \neq s$, то выполнено условие $substates_1 s_0$. Доказательство условия корректности выполняется следующим образом. Сначала выполняется его доказательство для дополнительного инварианта в виде следующей леммы:

```
1    lemma VC_extraInv  “VC extraInv env s0 v1_value …”vn_value
```

Затем эта лемма используется для доказательства основной теоремы следующим образом (Листинг 3):

```
1 theorem  “VC inv env s0 v1\_value …vn\”_value,
2 apply(unfold VC_def inv_def RInv_def)
3 apply(rule impI)
```

```
apply(rule conjI)доказательство
 RInv(s)
using VC_extraInv by (auto simp add: VC_def)
```

Листинг 3 – Доказательство условия корректности

Здесь сначала с помощью метода unfold раскрываются определения условия корректности VC, инварианта inv, являющегося конъюнкцией требования и дополнительного инварианта, а также определение требования RInv. Затем применяются правила для импликации и конъюнкции для расщепления инварианта на требование и дополнительный инвариант. Далее выполняется доказательство требования. Последняя команда применяет лемму, в которой был доказан дополнительный инвариант.

Рассмотрим используемые стратегии доказательства для различных классов требований и дополнительного инварианта.

Доказательства условий корректности для требований, соответствующих первому шаблону, выполняется следующим образом. Доказательство некоторых условий корректности для таких требований может быть выполнено без использования дополнительного инварианта методом auto. В других случаях для доказательства требования используется дополнительный инвариант. В этом случае для доказательства условий корректности необходимо выполнить разбор случаев $s_1 = s$ и $s_1 \neq s$. Данный разбор случаев может быть выполнен с помощью команды apply(simp split: if_splits), которая выполняет расщепление if-выражений и упрощает подцели. В качестве примера рассмотрим доказательство одного из условий корректности для 6-го требования к программе управления турникетом (Листинг 4).

```
    theorem proof_6_351: "VC351 inv6 env s0 PdOut paid_value  opened_value"
  apply(unfold VC351_def)
  apply simp
  apply(unfold inv6_def R6_def)
  apply(rule impI)
  apply(erule conjE)
  apply(erule conjE)
  apply(rule conjI)
   apply(rule conjI)
    apply simp
   apply(unfold extraInv_def)[1]
  subgoal premises vc_prems
    apply(rule allI)
    apply(rule impI)
    apply(simp split: if_splits)
    using vc_prems(1,3)
     apply (meson substate_refl)
    using vc_prems(2) by auto
```

```
using VC351_extraInv by (auto simp add: VC351_def)
```

Листинг 4 – Доказательство условия корректности для требования класса 1

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 simp split: if_splits. Первая подцель соответствует случаю $s_1 = s$ и доказывается с использованием посылки условия корректности, соответствующей условиям в программе (prems(1)) и дополнительного инварианта (prems(3)). Доказательство apply (meson substate_refl) было найдено с помощью sledgehammer. Доказательство для случая $s_1 \neq s$ следует из того, что требование (prems(2)) было истинно перед выполнением текущей итерации цикла управления.

Доказательство условий корректности для требований, соответствующих шаблону 2, выполняется схожим образом. Для доказательства необходимо выполнить разбор случаев 1) $s_2 = s$ и 2) $s_2 \neq s$. Так как время между состояниями s_1 и s_2 составляет одну итерацию цикла управления ($toEnvNum s_1 s_2 = 1$), в первом случае выполняется условие $s_1 = s_0$. Если истинность утверждения $vc_1(s_0, s) \longrightarrow vc_2(s)$ может быть доказана в результате упрощения, то истинность требования в состоянии s следует из его истинности в состоянии s_0, то есть условие корректности может быть доказано без использования дополнительного инварианта. При этом, если для доказательства не требуется равенство $s_1 = s_0$ в случае $s_2 = s$, то доказательство может быть выполнено методом auto и выполнять разбор случаев явно не требуется. Если для доказательства условия корректности необходимо равенство $s_1 = s_0$ в случае $s_2 = s$, оно может быть доказано с помощью леммы substate_toEnvNum_id. Случай $s_2 \neq s$ может быть доказан с помощью auto. Для доказательства случая $s_2 = s$ требуется применение метода blast. В качестве примера рассмотрим доказательство одного из условий корректности для 2-го требования к программе управления турникетом (Листинг 5).

```
theorem proof_2_362: "VC362 inv2 env s0 PdOut_value paid_value opened_value"
  apply(unfold VC362_def inv2_def R2_def)
  apply(rule impI)
  apply(rule conjI)
   apply(rule conjI)
    apply simp
   apply((rule allI)+)
```

```
8    apply(rule impI)
9    apply(simp split: if_splits)
10  using substate_toEnvNum_id apply blast
11  apply auto[1]
12 using VC362_extraInv by (auto simp add: VC362_def)
```

Листинг 5 – Доказательство условия корректности для тре требования класса 2

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. Случай $s_2 \neq s$ доказывается методом auto. Затем применяется лемма, в которой доказан дополнительный инвариант.

В случае, когда для доказательства требования необходимо использовать дополнительный инвариант, доказательство выполняется аналогично доказательству условий корректности для требований первого класса, рассмотренному ранее.

Доказательства условий корректности для требований, соответствующих третьему шаблону, выполняется следующим образом. Если $vc_2(s)$ может быть доказана в результате упрощения, то условие корректности доказывается автоматически с помощью метода auto. Иначе необходимо выполнить разбор следующих случаев: 1) $s_2 = s_3 = s$, 2) $s_3 = s$ и $s_2 \neq s$ и 3) $s_3 \neq s$. Доказательство случая 1 выполняется аналогично случаю 1 для требований второго класса. Если из $vc_2(s_0)$ следует $vc_2(s)$, то доказательство случая 2 следует из того, что требование выполнялось в состоянии s_0. Тогда для доказательства этого случая может быть задано промежуточное утверждение toEnvNum $s_2\, s_0 < \tau$. Далее доказательство выполнялось автоматически методами blast и simp Случай 3 может быть доказан автоматически с помощью автоматических методов, таких как auto и blast. В качестве примера рассмотрим доказательство одного из условий корректности для 5-го требования к программе управления светофором (Листинг 6).

```
1 theorem proof_5_500: "VC500 inv5 env s0 PdOut_value paid_value opened_value"
2   apply(unfold VC500_def)
3   apply simp
4   apply(unfold inv5_def R5_def)
5   apply(rule impI)
6   apply(rule conjI)
7   apply(rule conjI)
8    apply simp
9   subgoal premises vc_prems
```

```
10    apply((rule allI)+)
11    apply(rule impI)
12    apply(simp split: if_splits del: One_nat_def)
13    subgoal for s1 s2
14      apply(rule cut_rl[of "toEnvNum s2 s0 <10"])
15      using vc_prems substate_refl apply blast
16      by simp
17    using vc_prems by blast
18 using VC500_extraInv by (auto simp add: VC500_def)
```

Листинг 6 – Доказательство условия корректности для требования класса 3

Начало доказательства выполняется аналогично рассмотренному ранее. 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 $s2\, s0$. С помощью этого утверждения доказательство выполняется методом blast/ Лемма substate_refl используется для вывода утверждения substate s0 s0, необходимого для доказательства данного случая. Промежуточное утверждение доказано методом simp. Случай 3 доказывается методом blast.

В случае, когда из истинности $vc_2(s_0)$ не следует $vc_2(s)$, истинности требования в состоянии s_0 не достаточно для доказательства его истинности в состоянии s и необходимо использовать дополнительный инвариант следующим образом. Пусть все программные переменные, входящие в v_1 используются только в процессе p и процесс p находится в состоянии q в состоянии изменений s_0. Тогда оказательство случая 2 выполняется с использованием вспомогательного свойства вида 1 для процесса p и состояния q.

Доказательство условий корректности для требований, соответствующих шаблону 4, выполняется следующим образом. Сначала доказывается истинность дополнительного инварианта в состоянии s. Пусть программные переменные, используемые в требовании, изменяются только в процессе p и в состоянии изменений s процесс p находится в состоянии q. Тогда требование доказывается с помощью вспомогательного свойства вида 4 для процесса p и состояния q. Доказательство для требований класса 5 выполняется аналогично.

Доказательство условий корректности для дополнительного инварианта выполняется следующим образом. Сначала расщепляется конъюнкция в дополнительном инварианте в посылке условия корректности и с помощью команды subgoal конъюнктам задается имя, на которое затем можно ссылаться при доказательстве. Затем расщепляется конъюнкция в дополнительном инварианте в заключении условия корректности и доказательство каждого конъюнкта выполняется отдельно. Если посылка конъюнкта ложна в состоянии s, например, если в посылке утверждается, что некоторый процесс p находится в состоянии, отличном от того, в котором p находится в состоянии изменений s, то данный конъюнкт может быть доказан

методом simp с использованием соответствующего конъюнкта для предыдущей итерации. Таким же образом могут быть доказаны утверждения о значении таймера процесса и значениях переменных в некотором состоянии процесса. При доказательстве утверждений о значениях переменных необходимо также использовать утверждение о значениях соответствующих переменных в состоянии, в котором процесс находился в состоянии изменений s_0. Свойства, являющиеся усилением требований класса 3,доказываются аналогично требованиям этого класса. Наиболее сложным является доказательство свойств, усиливающих требования классов 4 и 5.

Рассмотрим доказательство для требований класса 4. Для класса 5 доказательство выполняется аналогично. Данные свойства содержат квантор существования и требуют явного применения правил работы с кванторами. Пусть переменные, входящие в соответствующее требование, изменяются только в процессе p, p находится в состоянии q в состоянии изменений s, а в состоянии изменений s_0 p находится в состоянии q_0. Пусть факты, специфицирующие усиление соответствующего требования для состояний q и q_0 имеют соответственно имена ei(n) и ei(m). Рассмотрим общий случай, когда ei(n) и ei(m) имеют вид 4. Для доказательства выполняется разбор случаев $s_4 = s$ и $s_4 \neq s$. Затем для случая $s_4 = s$ выполняется разбор случаев $s_1 = s$ и $s_1 \neq s$. Случай $s_1 = s$ доказывается следующим образом. Если заключение доказываемого свойства представляет собой дизъюнкцию, то в случае, когда формула $vc_2(s)$ истинна, доказывается первый дизъюнкт, иначе доказывается второй дизъюнкт. Для квантора существования применяется правило exI[of _ s]. Для доказательства случая $s_1 \neq s$, если ei(m) представляет собой дизъюнкцию, выполняется ее расщепление. Когда истинен первый дизъюнкт он используется для доказательства первого дизъюнкта в заключении. Когда истинен второй дизъюнкт, выбор доказываемого дизъюнкта в заключении подцели и применение правила для квантора существования выполняется аналогично случаю $s_1 = s$. Случай $s_4 \neq s$ доказывается автоматически методом simp с использованием факта ei(n). В качестве примера рассмотрим фрагмент доказательства одного условий корректности программы управления турникетом для дополнительного инварианта.

```
1       apply(rule allI)
2       subgoal for s4
3         apply(cases ”s4 = s s0 PdOut_value paid_value opened_value”)
4          apply(rule impI)
5          apply(rule allI)
6         subgoal for s1
7           apply(cases ”s1 = s4”)
8            apply(rule impI)
9            apply(rule disjI2)
10           apply(rule conjI)
11            apply simp
```

```
12           apply(rule allI)
13          subgoal for s2
14            using substate_antisym[of s1 s2] ei(5) by auto
15          apply(insert ei(10))
16          apply(erule allE[of _ s0])
17          apply(erule impE)
18          using substate_refl apply simp
19          apply(erule allE[of _ s1])
20          apply(rule impI)
21          apply(erule impE)
22           apply(simp split: if_splits)
23          apply(rule disjI1)
24          apply(erule exE)
25          subgoal for s3
26            apply(rule exI[of _ s3])
27            by simp
28          done
29        using ei(12) by simp
30
```

Листинг 7 – Фрагмент доказательства условия корректности для дополнительного инварианта

Здесь сначала применяется правило для квантора всеобщности по переменной s4. Затем команда subgoal делает имя переменной s4 доступным в тексте доказательства. Далее выполняется разбор случаев s4=s и s4 $\neq$ s. В доказательстве случая s4 = s применяются правила для импликации и квантора всеобщности по переменной s1. Команда subgoal делает имя переменной s1 доступным в тексте доказательства. Затем выполняется разбор случаев s1 = s и s1 $\neq$ s. В доказательстве случая s1=s применяется правило для импликации, а затем правило disjI2 для доказательства второго дизъюнкта В случае доказательства второго дизъюнкта выполняется расщепление конъюнкции командой apply(rule conjI). Утверждение toEnvNum s1 s4 < ltime s4 fridgeDoorController' доказывается методом simp. Для доказательства конъюнкта $\forall s2.\ \text{toEnvP}\ s2 \wedge \text{substate}\ s1\ s2 \wedge \text{substate}\ s2\ s4 \longrightarrow \text{getVarBool}\ s2\ open'$ применяется метод auto и лемма substate_antisym. Значения параметров лемм указываются явно с помощью of. Для доказательства случая s1 $\neq$ s, к цели в качестве посылки добавляется факт ei(10), утверждающий, что свойство 9, необходимое для доказательства, истинно в состоянии s0. Затем с помощью правила allE связанная квантором всеобщности переменная s4 сопоставляется с s0, правило impE применяется для импликации, посылка которой доказывается с помощью simp, с помощью правила allE связанная переменная s1 сопоставляется с s1, затем снова применяется правило impE. Так как факт ei(10) не содержит дизъюнкцию, он используется для доказательства первого дизъюнкта в заключении подцели. Для этого с помощью правила exE получено состояние s3, которое затем используется для доказательства утверждения с

квантором существования. Случай $s4 \neq s$ доказывается методом simp с использованием факта ei(12).

ЗАКЛЮЧЕНИЕ

В работе были получены следующие результаты:

1. Проанализированы язык poST и возможности системы Isabelle/HOL.
2. Сформулированы требования к генератору условий корректности poST-программ.
3. Разработан алгоритм генерации условий корректности, основанный на стратегии сильнейшего постусловия.
4. Реализован генератор условий корректности.
5. Разработан тестовый набор управляющих программ, включающий 7 программ. Для каждой программы предложены требования, которые были формализованы в Isabelle/HOL.
6. Предложены стратегии доказательства условий корректности в системе Isabelle/HOL для различных классов требований.
7. Для каждой из программ тестового набора порождены и доказаны условия корректности.

Предложенный подход расширяет область применения метода дедуктивной верификации процесс-ориентированных программ, так как требования для программ тестового набора являются типовыми для широгого класса управляющих программ.

В дальнейшем планируется исследовать возможность автоматизации доказательства условий корректности poST-программ на основе разработанных стратегий доказательства.

По результатам работы были подготовлены доклады на Международную научную студенческую конференцию МНСК 2022 и МНСК 2023, доклад на X Международную научную конференцию "Математическое и компьютерное моделирование"и статьи на IEEE конференции EDM 2022 и EDM 2023.

Выпускная квалификационная работа выполнена мной самостоятельно и с соблюдением правил профессиональной этики. Все использованные в работе материалы и заимствованные принципиальные положения (концепции) из опубликованной научной литературы и других источников имеют ссылки на них. Я несу ответственность за приведенные данные и сделанные выводы.

Я ознакомлен с программой государственной итоговой аттестации, согласно которой обнаружение плагиата, фальсификации данных и ложного цитирования является основанием для не допуска к защите выпускной квалификационной работы и выставления оценки «неудовлетворительно».

Черненко Иван Михайлович

«. . . . .». . . . . .20 . . г. ________________________

(Подпись студента)

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Affine Loop Invariant Generation via Matrix Algebra / Y. Ji, H. Fu, B. Fang, H. Chen // Computer Aided Verification / Под ред. S. Shoham, Y. Vizel. — Cham : Springer International Publishing, 2022. — С. 257–281.
2. Barnett M., Leino K. R. M., Schulte W. The Spec# Programming System: An Overview // Construction and Analysis of Safe, Secure, and Interoperable Smart Devices / Под ред. G. Barthe, L. Burdy, M. Huisman и др. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2005. — С. 49–69.
3. Blanchette J. C. — Hammering Away: A User's Guide to Sledgehammer for Isabelle/HOL. — URL: `https://isabelle.in.tum.de/doc/sledgehammer.pdf` (дата обращения: 20.05.2023).
4. Boogie: A Modular Reusable Verifier for Object-Oriented Programs / M. Barnett, Bor-Yuh E. Chang, R. DeLine и др. // Formal Methods for Components and Objects / Под ред. F. S. de Boer, M. M. Bonsangue, S. Graf, Willem-Paul de Roever. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2006. — С. 364–387.
5. Cao Q., Beringer L., Gruetter S. et al. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. // Automated Reasoning. — 2018. — Т. 61. — С. 367–422.
6. Cok D. R. OpenJML: software verification for Java 7 using JML, OpenJDK, and Eclipse // arXiv preprint arXiv:1404.6608. — 2014.
7. Dijkstra E. W. A Discipline of Programming. — Prentice-Hall, Upper Saddle River, 1976.
8. EMF: eclipse modeling framework / D. Steinberg, F. Budinsky, E. M., M. Paternostro. — Pearson Education, 2008.
9. Filliâtre Jean-Christophe, Paskevich A. Why3 — Where Programs Meet Provers // Programming Languages and Systems / Под ред. M. Felleisen, P. Gardner. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2013. — С. 125–128.
10. Fleury M., Schurr Hans-Jörg. Reconstructing veriT proofs in Isabelle/HOL // arXiv preprint arXiv:1908.09480. — 2019.
11. Foster S., Gleirscher M., Calinescu R. Towards deductive verification of control algorithms for autonomous marine vehicles // 2020 25th International Conference on Engineering of Complex Computer Systems (ICECCS) / IEEE. — 2020. — С. 113–118.
12. Harrison J., Urban J., Wiedijk F. History of Interactive Theorem Proving // Computational Logic. — 2014.
13. Henrio L., Khan M. U. Asynchronous Components with Futures: Semantics and Proofs in Isabelle/HOL // Electronic Notes in Theoretical Computer Science. — 2010. — Т. 264, № 1. — С. 35–53. — Proceedings of the 7th International Workshop on Formal Engineering

approaches to Software Components and Architectures (FESCA 2010). URL: https://www.sciencedirect.com/science/article/pii/S1571066110000630.
14. Hoare C. A. R. An axiomatic basis for computer programming // Communications of the ACM. — 1969. — T. 12, № 10. — C. 576–580.
15. Huerta y Munive Jonathan Julián, Struth Georg. Predicate transformer semantics for hybrid systems: verification components for Isabelle/HOL // Journal of Automated Reasoning. — 2022. — T. 66, № 1. — C. 93–139.
16. Inductive invariant generation via abductive inference / I. Dillig, T. Dillig, B. Li, K. L. McMillan // Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications. — 2013.
17. Kovács L. Reasoning Algebraically About P-Solvable Loops // International Conference on Tools and Algorithms for Construction and Analysis of Systems. — 2008.
18. Leino R., Logozzo F. Loop Invariants on Demand // Proceedings of the the 3rd Asian Symposium on Programming Languages and Systems (APLAS'05). — T. 3780. — Springer Verlag, 2005. — November. — URL: https://www.microsoft.com/en-us/research/publication/loop-invariants-on-demand/.
19. Marić F. Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL // Theoretical Computer Science. — 2010. — T. 411, № 50. — C. 4333–4356. — URL: https://www.sciencedirect.com/science/article/pii/S0304397510004937.
20. Matichuk D., Wenzel M., Murray T. An Isabelle proof method language // International Conference on Interactive Theorem Proving. — 2014.
21. Moskal M. Verifying Functional Correctness of C Programs with VCC // NASA Formal Methods / Под ред. M. Bobaru, K. Havelund, G. J. Holzmann, R. Joshi. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2011. — C. 56–57.
22. Paulson L. C. The foundation of a generic theorem prover // Journal of Automated Reasoning. — 1989. — T. 5. — C. 363–397.
23. Refinement modeling and verification of secure operating systems for communication in digital twins / Z. Qian, G. Sun, X. Xing, G. Dhiman // Digital Communications and Networks. — 2022. — URL: https://www.sciencedirect.com/science/article/pii/S2352864822001602.
24. STeP: The Stanford Temporal Prover / Z. Manna, N. Bjørner, A. Browne и др. // TAPSOFT '95: Theory and Practice of Software Development / Под ред. P. D. Mosses, M. Nielsen, M. I. Schwartzbach. — Berlin, Heidelberg : Springer Berlin Heidelberg, 1995. — C. 793–794.
25. Stark J., Ireland A. Invariant Discovery via Failed Proof Attempts // Logic-Based Program Synthesis and Transformation / Под ред. P. Flener. — Berlin, Heidelberg : Springer Berlin Heidelberg, 1999. — C. 271–288.

26. Strecker M. Formal Verification of a Java Compiler in Isabelle // Proceedings of the 18th International Conference on Automated Deduction. — 2004.
27. Templates and recurrences: better together / J. Breck, J. Cyphert, Z. Kincaid, T. Reps // Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. — 2020. — С. 688–702.
28. A Temporal Requirements Language for Deductive Verification of Process-Oriented Programs / I. Chernenko, I. S. Anureev, N. O. Garanina, S. M. Staroletov // 2022 IEEE 23rd International Conference of Young Professionals in Electron Devices and Materials (EDM). — 2022. — С. 657–662.
29. Zyubin V. E. Hyper-automaton: a Model of Control Algorithms // 2007 Siberian Conference on Control and Communications. — 2007. — С. 51–57.
30. poST: A Process-Oriented Extension of the IEC 61131-3 Structured Text Language / V. E. Zyubin, A. S. Rozov, I. S. Anureev и др. // IEEE Access. — 2022.
31. seL4: Formal verification of an OS kernel / G. Klein, K. Elphinstone, G. Heiser и др. // Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. — 2009. — С. 207–220.
32. A survey on theorem provers in formal methods / M. S. Nawaz, M. Malik, Yi Li и др. // arXiv preprint arXiv:1912.03028. — 2019.
33. Автоматическая верификация C-программ на основе смешанной аксиоматической семантики / И. В. Марьясов, В. А. Непомнящий, А. В. Промский, Д. А. Кондратьев // Моделирование и анализ информационных систем. — 2013. — Т. 20, № 6. — С. 52–63.
34. Аджиев В. Мифы о безопасном ПО: уроки знаменитых катастроф. — 1998. — URL: http://www.osp.ru/os/1998/06/179592/ (дата обращения: 17.04.2023).
35. Башев В. PoST language core. — URL: https://github.com/v-bashev/post_core (дата обращения: 27.04.2023).
36. Камкин А. С. Введение в формальные методы верификации программ. — М. : Макс Пресс, 2018.
37. Непомнящий В. А., Рякин О. М. Прикладные методы верификации программ. — М. : Радио и связь, 1988.

ПРИЛОЖЕНИЕ А

Генератор условий корректности для языка poST

Руководство оператора

Листов 8

2023

АННОТАЦИЯ

Данный программный документ представляет собой руководство оператора по применению и эксплуатации генератора условий корректности для процесс-ориентированного языка poST.

В разделе ”Назначение программы”указаны назначение программы и ее функции. В разделе ”Условия выполнения программы”приведены условия, необходимые для работы программы (требования к программным средствам). В разделе ”Выполнение программы”приведено описание формата и возможных вариантов команд, обеспечивающих запуск программы. В разделе ”Сообщения оператору”приведено описание текстовых сообщений, выдаваемых в ходе выполнения программы.

Оформление программного документа «Руководство оператора» произведено по требованиям ГОСТ 19.505-79 «ЕСПД. Руководство оператора» и ГОСТ 19.105-78 «Единая система программной документации (ЕСПД). Общие требования к программным документам (с Изменением N 1)».

СОДЕРЖАНИЕ

1. Назначение программы 60
2. Условия выполнения программы 61
3. Выполнение программы 62
   3.1. Запуск программы 62
   3.2. Выполнение программы 62
   3.3. Завершение работы программы 63
4. Сообщения оператору 64

1. Назначение программы

Программа предназначена для генерации условий корректности программ на процесс-ориентированном языке poST. Программа по исходному коду poST-программы создает теорию для системы машинной поддержки доказательства Isabelle/HOL, содержащую условия корректности, параметризованные аннотацией инварианта цикла управления, аннотацией, определяющей ограничения на изменения входных переменных, и аннотациями циклов в программе, а также определения констант, используемых в условиях корректности.

Функцией данной программы является генерация теории Isabelle/HOL с условиями корректности.

2. Условия выполнения программы

Для выполнения программы необходимо, чтобы на компьютере пользователя была установлена виртуальная машина Java. Путь к Java должен быть добавлен в переменную среды PATH операционной системы.

3. Выполнение программы

3.1. Загрузка и запуск программы

Перед запуском программы необходимо скачать JAR-файл с генератором условий корректности vcgenerator.jar.

Для запуска программы на компьютере пользователя необходимо открыть командную строку, перейти в директорию, в которой необходимо создать файл теории для Isabelle/HOL, содержащей условия корректности, и запустить JAR-файл с программой. В качестве аргумента программе необходимо передать путь к файлу исходного кода poST-программы. Для этого необходимо выполнить команду

```
java -jar <путь к директории с JAR-файлом>/vcgenerator.jar <post_program>,
```

где <post_program> — путь к файлу с расширением «.post», который содержит исходный код программы на языке poST.

Также можно указать временной интервал, соответствующий периоду активации, с помощью опции -i. Для этого необходимо выполнить команду `java -jar <путь к директории с JAR-файлом>/vcgenerator.jar -i <interval> <post_program>` или `java -jar <путь к директории с JAR-файлом>/vcgenerator.jar <post_program> -i <interval>`,
где <post_program> — путь к файлу с расширением «.post», который содержит исходный код программы на языке poST.
<interval>— временной интервал, соответствующий периоду активации, указывается в таком же формате, как литералы типа TIME в языке poST, то есть T#<число_дней>d<число_часов>h<число_минут>m<число_секунд>s<число_миллисекунд>ms.

Если в файле poST-программы присутствует конфигурация для этой программы с указанием периода активации, то в качестве периода активации используется значение из конфигурации. Значение аргумента опции -i в этом случае не учитывается. Если в файле poST-программы отсутствует конфигурация или в ней не задано значение периода активации и указана опция -i, в качестве периода активации используется значение <interval>. Иначе в качестве периода активации используется значение по умолчанию 100 мс.

3.2. Выполнение программы

Во время выполнения программа создает в текущей директории файл с теорией Isabelle/HOL с условиями корректности. Имя созданной теории получается кодированием имени исходного файла с poST-программой без расширения с использованием символов, допустимых в имени теории Isabelle/HOL. Символы латинского алфавита и цифры остаются без

изменений. Символы, которые не могут содержаться в имени теории, заменяются на _<ASCII код символа>.

В случае наличия синтаксических ошибок в poST-программе пользователю выдаются сообщения об ошибках и условия корректности не генерируются. Если в текущей директории есть файл теории с таким же именем, он остается без изменений.

3.3. Завершение программы

Программа завершается после того, как будет выполнена генерация теории Isabelle/HOL, а также в случае наличия синтаксических ошибок в poST-программе.

4. Сообщения оператору

В ходе выполнения программы пользователю выдаются следующие сообщения:

1. В случае успешной генерации теории Isabelle/HOL с условиями корректности выдается сообщение «Code generation finished.».
2. В случае возникновения ошибок в ходе работы программы выдается сообщение «Code generation aborted.».
3. В случае наличия синтаксических ошибок выводятся соответствующие диагностические сообщения с привязкой к номеру строки в poST-программе. Сообщения выводятся в формате ERROR:<диагностическое сообщение> (<имя файла с исходным кодом> line : <номер строки> column : <номер символа в строке>). Данные сообщения сигнализируют, что пользователю необходимо исправить синтаксические ошибки в poST-программе и перезапустить генератор условий корректности
4. В случае наличия предупреждений выводятся соответствующие диагностические сообщения с привязкой к номеру строки в poST-программе. Сообщения выводятся в формате WARNING:<диагностическое сообщение> (<имя файла с исходным кодом> line : <номер строки> column : <номер символа в строке>).
5. Если poST-программа содержит вызовы функций, выдается сообщение ”Functions are not supported”
6. Если poST-программа содержит вызовы функциональных блоков, выдается сообщение ”Function blocks are not supported”

Первые два сообщения выводятся в стандартный поток вывода. Остальные сообщения выводятся в стандартный поток вывода сообщений об ошибках.

ПРИЛОЖЕНИЕ Б

Тестовый набор управляющих программ

Сушилка для рук

В качестве управляемого устройства в данной задаче рассматривается сушилка для рук. Устройство включает сенсор, показывающий, есть ли руки, вентилятор и обогреватель. Программа принимает входной сигнал с сенсора и, в зависимости от входного сигнала, управляет тепловентилятором. Если руки появляются, то тепловентилятор включается. Если руки убрали, то по прошествии определенного времени тепловентилятор выключается.

Программа управления сушилкой для рук представлена на листинге 8.

```
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
```

```
29 END_PROGRAM
```

Листинг 8 – Програма управления сушилкой для рук

В программе объявлены две переменные: входная переменная $hands$, показывающая наличие рук под тепловентилятором, и выходная переменная $dryer$, определяющая, включен ли тепловентилятор. Определен один процесс $Ctrl$ с двумя состояниями $waiting$ и $drying$. В состоянии $waiting$ проверяется наличие рук. Если руки есть тепловентилятор включается и процесс $Ctrl$ переходит в состояние $drying$. Если руки отсутствуют, тепловентилятор выключается. В состоянии $drying$ также проверяется наличие рук. Если руки есть, таймер процесса сбрасывается. В данном состоянии установлен таймаут. По истечении 1 секунды процесс $Ctrl$ переходит в состояние $waiting$. Период активации процесса равен 100 миллисекундам.

К программе предъявляются следующие требования:

1. Тепловентилятор должен включиться за время, приемлемое с точки зрения пользователя (не позже 0.2 секунды), после того, как под ним появляются руки.
2. Тепловентилятор никогда не включается произвольно (Если нет рук и тепловентилятор не включен, то он не включится до тех пор, пока не появятся руки).
3. Если руки убрали, то не более чем через 1 с тепловентилятор выключится, если руки за это время не появились вновь.
4. Если руки имеются и тепловентилятор включен, он не выключится.

Для формализации требований в данной задаче используется только инвариант цикла управления, так как ограничений на значения единственной входной переменной нет (руки могут появиться в любой момент). Например, инвариант цикла, являющийся формальной аннотацией для первого требования, имеет вид: $\text{toEnvP}\, s \wedge (\forall s1 s2.\, \text{substate}\, s1\, s2 \wedge \text{substate}\, s2\, s \wedge \text{toEnvP}\, s1 \wedge \text{toEnvP}\, s2 \wedge \text{toEnvNum}\, s1\, s2 = 1 \wedge \text{getVarBool}\, s1\, hands = False \wedge \text{getVarBool}\, s1\, dryer = False \wedge \text{getVarBool}\, s2\, hands = True \longrightarrow (\exists s4.\, \text{toEnvP}\, s4 \wedge \text{substate}\, s2\, s4 \wedge \text{substate}\, s4\, s \wedge \text{toEnvNum}\, s2\, s4 \leq 1 \wedge \text{getVarBool}\, s4\, dryer = True \wedge (\forall s3.\, \text{toEnvP}\, s3 \wedge \text{substate}\, s2\, s3 \wedge \text{substate}\, s3\, s4 \wedge s3 \neq s4 \longrightarrow \text{getVarBool}\, s3\, hands = True)))$

Светофор на пешеходном переходе

Управляемым устройством является пешеходный светофор с кнопкой, установленный на переходе. Его штатное состояние— «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 менее 10 с. Назад, то зеленый включится по истечении таймаута между переходами— 15 с. Если включился зеленый, он будет гореть в течение 30 с., затем включается красный. Таким образом, имеется один входной сигнал— «кнопка нажата» и один сигнал управления— «горит зеленый» Программа принимает входной сигнал и в зависимости от него управляет сигналом светофора.

Программа управления светофором на пешеходном переходе приведена на листинге 9.

```
PROGRAM Controller
  VAR_INPUT
    requestButton: BOOL;
  END_VAR
  VAR_OUTPUT
    trafficLight: BOOL;
  END_VAR
  VAR
    requestButtonPressed: BOOL;
  END_VAR

  VAR CONSTANT
    GREEN : BOOL := TRUE;
    RED : BOOL := FALSE;
    PRESSED : BOOL := TRUE;
    NOT_PRESSED : BOOL:= FALSE;
    MINIMAL_RED_TIME_LIMIT  : TIME := T#10s;
    RED_TO_GREEN_TIME_LIMIT : TIME := T#5s;
    GREEN_TIME_LIMIT : TIME := T#30s;
  END_VAR

  PROCESS controller
    STATE minimalRed
      IF requestButton THEN
        requestButtonPressed := TRUE;
      END_IF
      TIMEOUT MINIMAL_RED_TIME_LIMIT  THEN
        SET NEXT;
      END_TIMEOUT
    END_STATE

    STATE redAfterMinimalRed
      IF requestButtonPressed OR requestButton THEN
        SET NEXT;
      END_IF
    END_STATE

    STATE redToGreen
      TIMEOUT RED_TO_GREEN_TIME_LIMIT THEN
        trafficLight := GREEN;
        requestButtonPressed := FALSE;
        SET NEXT;
```

```
      END_TIMEOUT
    END_STATE

    STATE green
      TIMEOUT GREEN_TIME_LIMIT THEN
        trafficLight := RED;
        SET STATE minimalRed;
      END_TIMEOUT
    END_STATE
  END_PROCESS
END_PROGRAM

```

Листинг 9 – Программа управления светофором на пешеходном переходе

В данной программе объявлены три логических переменных: входная переменная $requestButton$, показывающая, нажата ли кнопка, выходная переменная $trafficLight$, которая определяет сигнал управления, и локальная переменная $requestButtonPressed$. Определен один процесс $controller$, который имеет четыре состояния: $minimalRed$, $redAfterMinimalRed$, $redToGreen$ и $green$. Когда процесс находится в состоянни $minimalRed$, т. е. во время минимального интервала красного сигнала светофора, нажатие кнопки запоминается в переменной $requestButtonPressed$. Также в состоянии $minimalRed$ установлен таймаут, определяющий минимальноую продолжительность красного сигнала светофора. По истечении таймаута процесс переходит в состояние $redAfterMinimalRed$. Нажатие кнопки запоминается для того, чтобы по истечении таймаута светофор начал переключение на зеленый. В состоянии $redAfterMinimalRed$ процесс проверяет, нажата ли кнопка в данный момент ли была нажата ранее в состоянии $minimalRed$. Если кнопка была нажата, процесс переходит в состояние $redToGreen$. Тело состояния $redToGreen$ состоит из оператора таймаута, определяющего задержку переключения светофора с красного на зеленый. По истечении этого таймаута сбрасывается значение переменной $requestButtonPressed$, включается зеленый сигнал светофора и процесс переходит в состояние $green$. Тело состояния $green$ состоит из оператора таймаута, определяющего продолжительность зеленого сигнала светофора. По истечении этого таймаута включается красный сигнал, и процесс переходит в состояние $minimalRed$. Период активации процесса равен 100 миллисекундам

К данной программе предъявляются следующие требования:

1. Если горел красный свет и кнопку нажали, то не более, чем через Tr загорится зеленый свет.
2. Если только что загорелся зеленый, то зеленый будет гореть не менее Tg секунд.
3. Если только что загорелся зеленый, то за время не более Tg он переключится на красный.
4. Если только что загорелся красный, то он горит постоянно, пока не нажмут на кнопку.

5. Если загорелся красный, то красный будет гореть не менее Tr секунд.

Здесь Tr — максимальное время, в течение которого пешеход ожидает включения зеленого сигнала светофора после нажатия на кнопку, равное 15 с., Tg — продолжительность зеленого сигнала светофора, равная 30 с. Tr равно сумме программных констант MINIMAL_RED_TIME_LIMIT и RED_TO_GREEN_TIME_LIMIT, Tg равно программной константе GREEN_TIME_LIMIT.

Для формализании требований также используется только инвариант цикла управления, так как ограничений на единственную входную переменную requestButton нет (кнопка может быть нажата в любой момент). Например, инвариант, являющийся формальной аннотацией для первого требования имеет вид: $\text{toEnvP}\, s \wedge (\forall s1 s2.\, \text{substate}\, s1\, s2 \wedge \text{substate}\, s2\, s \wedge \text{toEnvP}\, s1 \wedge \text{toEnvP}\, s2 \wedge \text{toEnvNum}\, s1\, s2 = 1 \wedge \text{toEnvNum}\, s2\, s \geq Tr \wedge \text{getVarBool}\, s1\, trafficLight = RED \wedge \text{getVarBool}\, s1\, requestButton = NOT_PRESSED \wedge \text{getVarBool}\, s2\, requestButton = PRESSED \longrightarrow (\exists s4.\, \text{toEnvP}\, s4 \wedge \text{substate}\, s2\, s4 \wedge \text{substate}\, s4\, s \wedge \text{toEnvNum}\, s2\, s4 \leq Tr \wedge \text{getVarBool}\, s4\, trafficLight = GREEN \wedge (\forall s3.\, \text{toEnvP}\, s3 \wedge \text{substate}\, s2\, s3 \wedge \text{substate}\, s3\, s4 \wedge s3 \neq s4 \longrightarrow \text{getVarBool}\, s3\, trafficLight = RED)))$

Вращающаяся дверь

В данной задаче в качестве управляемого устройства рассматривается вращающаяся дверь. Вращающиеся двери устанавливают на входах в здания с большим потоком посетителей. Устройство состоит из трех- или четырехсекционной двери, вращающейся вокруг вертикальной оси, привода (rotation) и тормоза (brake) для мгновенной 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 по пришествии определенного времени вращение останавливается. Датчик давления регистрирует давление на секционные перегородки. Вращение приостанавливается на небольшое время, когда на перегородки оказывается давление.

Программа управления вращающейся дверью на языке poST приведена на листинге 10.

```
PROGRAM RevolvingDoor
    VAR_INPUT
      user : BOOL;
      pressure : BOOL;
    END_VAR

    VAR_OUTPUT
      rotation : BOOL;
```

```
9       brake : BOOL;
10    END_VAR
11
12    VAR CONSTANT
13      SUSPENSION_TIME : TIME := T#1s;
14      DELAY : TIME := T#1s;
15    END_VAR
16
17    PROCESS Controller
18      STATE motionless
19        IF user THEN
20          IF pressure THEN
21            brake := TRUE;
22            SET STATE suspended;
23          ELSE
24            rotation := TRUE;
25            SET NEXT;
26          END_IF
27        END_IF
28      END_STATE
29
30      STATE rotating
31        IF pressure THEN
32          rotation := FALSE;
33          brake := TRUE;
34          SET NEXT;
35        ELSIF user THEN
36          RESET TIMER;
37        END_IF
38        TIMEOUT DELAY  THEN
39          rotation := FALSE;
40          SET STATE motionless;
41        END_TIMEOUT
42      END_STATE
43
44      STATE suspended
45        IF pressure THEN
46          RESET TIMER;
47        END_IF
48        TIMEOUT SUSPENSION_TIME THEN
49          brake := FALSE;
50          rotation := TRUE;
51          SET STATE rotating;
```

```
52          END_TIMEOUT
53        END_STATE
54      END_PROCESS
55    END_PROGRAM
```

Листинг 10 – Программа управления вращающейся дверью

В программе объявлены четыре логических переменных: входные переменные $user$ и $pressure$ и выходные переменне $rotation$ и $brake$. Переменная $user$ показывает присутствие пользователя. Переменная $preesure$ показывает, оказывается ли давление на секционные перегородки. Переменная $rotation$ определяет, вращается ли дверь. Переменная $brake$ оределяет сигнал торможения. В программе определен один процесс $Controller$, который имеет три состояния $motionless$, $rotating$ и $suspended$. В состоянии $motionless$ процесс проверяет, есть ли пользователь. Если пользователь присутствует, проверяется сигнал датчика давления. Если давление на секционные перегородки оказывается, процесс переходит в состояние $suspende$. Иначе дверь начинает вращаться и процесс переходит в состояние $rotating$. В состоянии $rotating$ также проверяется сигнал датчика давления. Если давление есть, вращение останавливается, подается сигнал торможения и процесс переходит в состояние $suspended$. Иначе при наличии пользователя сбрасывается таймер процесса. В данном состоянии установлен таймаут. По истечении 1 секунды при отсутствии пользователя вращение останавливается и процесс переходит в состояние $motionless$. В состоянии $suspended$ также проверяется сигнал датчика давления. Если давление есть, таймер процесса сбрасывается. В данном состоянии установлен таймаут. По истечении 1 секунды вращение возобновляется и процесс переходит в состояние $rotating$. Период активации процесса равен 100 миллисекундам.

Для программы были сформулированы следующие требования:

1. При входе пользователя дверь начинает вращаться, 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 более, чем через 1 с. вращение остановится, если за это время пользователи не появятся вновь.
4. Если на секционные перегородки оказывается давление, то вращение приостанавливается не менее, чем на 1 с.
5. Если на секционные перегородки перестали оказывать давление, то не более, чем через 1 с. вращение возобновится.
6. Запрещена одновременная подача сигналов rotation и brake.

Для формализации требований используется только инвариант цикла уравления, так как ограничений на входные переменные user и pressure (пользователь может появить-

ся и оказывать давление на секционные перегородки в любой момент). Например, инвариант цикла, являющийся формальной аннотацией для первого требования, имеет вид: $\text{toEnvP}\, s \wedge (\forall s1 s2.\, \text{substate}\, s1\, s2 \wedge \text{substate}\, s2\, s \wedge \text{toEnvP}\, s1 \wedge \text{toEnvP}\, s2 \wedge \text{toEnvNum}\, s1\, s2 = 1 \wedge \text{getVarBool}\, s1\, rotation = False \wedge \text{getVarBool}\, s2\, user \wedge \neg \text{getVarBool}\, s2\, pressure \longrightarrow \text{getVarBool}\, s2\, rotation = True)$

Эскалатор

В данной задаче управляемым устройством является эскалатор. Эскалатор может двигаться как вверх, так и вниз. Направление движения определяется переключателем направления (directionSwitch). Лента эскалатора начинает движение при присутствии пользователей и прекращает движение при их отсутствии. Наличие пользователей перед эскалатором на верхнем и нижнем этажах регистрируется датчиками (userAtTop, userAtBottom). При попадании предметов между ступенями, которое регистрируется датчиком stuck, движение прекращается. Возобновление движения происходит по прошествии определенного времени после удаления застрявшего предмета. Движение также прекращается при нажатии тревожной кнопки (alarmButton).

Программа управления эскалатором на языке poST приведена на листинге 11.

```
1  PROGRAM Escalator
2
3    VAR_INPUT
4      userAtTop : BOOL;
5      userAtBottom : BOOL;
6      directionSwitch : BOOL;
7      alarmButton: BOOL;
8      stuck: BOOL;
9    END_VAR
10
11   VAR_OUTPUT
12     up : BOOL;
13     down : BOOL;
14   END_VAR
15
16   VAR CONSTANT
17     UP : BOOL := TRUE;
18     DOWN : BOOL := FALSE;
19     DELAY : TIME := T#2m;
20     SUSPENSION_TIME : TIME := T#1s;
21   END_VAR
22
```

```
23  VAR
24    direction : BOOL;
25    moving : BOOL;
26  END_VAR
27
28  PROCESS Ctrl
29    STATE motionless
30      IF alarmButton THEN
31        SET STATE emergency;
32      ELSIF stuck THEN
33        SET STATE stuckState;
34      ELSIF userAtTop OR userAtBottom THEN
35        IF directionSwitch = UP THEN
36          up := TRUE;
37          moving := TRUE;
38          direction := UP;
39          SET STATE goUp;
40        ELSE
41          down := TRUE;
42          moving := TRUE;
43          direction := DOWN;
44          SET STATE goDown;
45        END_IF
46      ELSE
47        direction := directionSwitch;
48      END_IF
49    END_STATE
50
51    STATE goUp
52      IF alarmButton THEN
53        SET STATE emergency;
54      ELSIF stuck THEN
55        SET STATE stuckState;
56      ELSIF userAtTop OR userAtBottom THEN
57        RESET TIMER;
58      END_IF
59      TIMEOUT DELAY THEN
60        up := FALSE;
61        moving := FALSE;
62        direction := directionSwitch;
63        SET STATE motionless;
64      END_TIMEOUT
65    END_STATE
```

```
66
67    STATE goDown
68      IF alarmButton THEN
69        SET STATE emergency;
70      ELSIF stuck THEN
71        SET STATE stuckState;
72      ELSIF userAtTop OR userAtBottom THEN
73        RESET TIMER;
74      END_IF
75      TIMEOUT DELAY THEN
76        down := FALSE;
77        moving := FALSE;
78        direction := directionSwitch;
79        SET STATE motionless;
80      END_TIMEOUT
81    END_STATE
82
83    STATE stuckState
84      up := FALSE;
85      down := FALSE;
86      IF alarmButton THEN
87        SET STATE emergency;
88      ELSIF stuck THEN
89        RESET TIMER;
90      END_IF
91      TIMEOUT SUSPENSION_TIME THEN
92        IF moving THEN
93          IF direction = UP THEN
94            up := TRUE;
95            SET STATE goUp;
96          ELSE
97            down := TRUE;
98            SET STATE goDown;
99          END_IF
100       ELSE
101         SET STATE motionless;
102       END_IF
103     END_TIMEOUT
104   END_STATE
105
106   STATE emergency
107     up := FALSE;
108     down := FALSE;
```

```
109     END_STATE
110   END_PROCESS
111 END_PROGRAM
```

Листинг 11 – Программа управления эскалатором

В программе объявлены входные переменные $userAtTop$, $userAtBottom$, $directionSwitch$, $alarmButton$ и $stuck$, выходные переменные up и $down$ и локальные переменные $direction$ и $moving$. Переменные $userAtTop$ и $userAtBottom$ показывают присутствие пользователя перед эскалатором на верхнем и нижнем этажах соответственно. Переменная $directionSwitch$ показывает направление движения эскалатора. Переменная $alarmButton$ показывает, нажата ли тревожная кнопка. Переменная $stuck$ показывает застревание предметов между ступенями эскалатора. Переменные up и $down$ определяют управляющие сигналы движения эскалатора вверх и вниз соответственно. В переменных $moving$ и $direction$ запоминается двигался ли эскалатор и в каком направлении, чтобы после приостановки эскалатор продолжил движение в том же направлении. В программе определен один процесс, которые имеет пять состояний: $motionless$, $goUp$, $goDown$, $stuckState$ и $emergency$. Во всех состоянияхх кроме $emergency$ проверяется, нажата ли тревожная кнопка. Если тревожная кнопка нажата, процесс переходит в состояние $emergency$. Если тревожная кнопка не нажата, проверяется значение переменной $stuck$. Если между ступенями эскалатора застевают предметы, то процесс переходит в состояние $stuckState$ или сбрасывает таймер, если процесс ранее находился в этом состоянии. Иначе в состоянии $motionless$ проверяется присутствие пользователя на верхнем или нижнем этаже. Если пользователь есть,в зависимости от значения переменной $directionSwitch$ эскалатор начинает движении вверх или вниз и процесс переходит соответственно в состояние $goUp$ или $goDown$. В состояниях $goUp$ и $goDown$ также проверяется присутствие пользователя. Если пользователь присутствует таймер сбрасывается. В этих состояниях установлен таймаут, по истечении которого движение останавливается и процесс переходит в состояние $motionless$. В состоянии $stuckState$ движение останавливается. В этом состоянии установлен таймаут. В состоянии $emergency$ движение эскалатора останавливается. Период активации процесса равен 100 миллисекундам.

Для данной программы были сформулированы следующие требования:

1. Если пользователи отсутствуют, то не более, чем через 2 мин. лента эскалатора прекращает движение, если пользователи за это время не появились вновь.
2. Если есть пользователь, эскалатор двигался и тревожная кнопка не нажата, то эскалатор продолжит движение в том же направлении.
3. При попадании предметов между ступенями движение прекращается не менее, чем на 1 с.

4. При нажатии тревожной кнопки движение должно прекратиться за приемлемое время (не позже 0.2 секунды).

Для формализации требований используется только инвариант цикла управления, так как ограничений на значения входных переменных нет. Следующая формула является формальной аннотацией для первого требования: $\text{toEnvP}\ s \wedge (\forall s1 s2.\ \text{substate}\ s1\ s2 \wedge \text{substate}\ s2\ s \wedge \text{toEnvP}\ s1\ \wedge\ \text{toEnvP}\ s2\ \wedge\ \text{toEnvNum}\ s1\ s2\ =\ 1\ \wedge\ \text{toEnvNum}\ s2\ s\ \wedge\ DELAY'TIMEOUT\ \wedge\ (\text{getVarBool}\ s1\ up'\ =\ True \vee \text{getVarBool}\ s1\ down'\ =\ True) \wedge \neg\, \text{getVarBool}\ s2\ userAtTop' \wedge \neg\, \text{getVarBool}\ s2\ userAtBottom' \longrightarrow (\exists s4.\ \text{toEnvP}\ s4\ \wedge\ \text{substate}\ s2\ s4\ \wedge\ \text{substate}\ s4\ s\ \wedge\ \text{toEnvNum}\ s2\ s4 \leq DELAY'TIMEOUT \wedge (\text{getVarBool}\ s4\ up' = False \wedge \text{getVarBool}\ s4\ down' = False\ \vee\ \text{getVarBool}\ s4\ userAtTop'\ \vee\ \text{getVarBool}\ s4\ userAtBottom')\ \wedge\ (\forall s3.\ \text{toEnvP}\ s3\ \wedge\ \text{substate}\ s2\ s3 \wedge \text{substate}\ s3\ s4 \wedge s3 \neq s4 \longrightarrow (\text{getVarBool}\ s3\ up' = True \vee \text{getVarBool}\ s3\ down' = True) \wedge \neg\, \text{getVarBool}\ s3\ userAtTop' \wedge \neg\, \text{getVarBool}\ s3\ userAtBottom')))$

Холодильник

В данной задаче в качестве управляемого устройства рассматривается холодильник. Холодильник состоит из холодильной и морозильной камер и меет два компресса. Температура вхолодильной камере регистрируется датчиками fridgeTempGreaterMin и fridgeTempGreaterMax, показывающие, превышает ли температура минимальное и максимальное значения 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).

Программа управления холодильником на языке poST приведена на листинге 12.

```
1  PROGRAM Fridge
2
3    VAR_INPUT
4      fridgeTempGreaterMin : BOOL;
5      fridgeTempGreaterMax : BOOL;
6      freezerTempGreaterMin : BOOL;
7      freezerTempGreaterMax : BOOL;
8      fridgeDoor : BOOL;
9    END_VAR
10
11   VAR_OUTPUT
```

```
12    fridgeCompressor : BOOL;
13    freezerCompressor : BOOL;
14    lighting : BOOL;
15    doorSignal : BOOL;
16  END_VAR
17
18  VAR CONSTANT
19    OPEN : BOOL := TRUE;
20    CLOSED : BOOL := FALSE;
21
22    OPEN_DOOR_TIME_LIMIT : TIME := T#30s;
23  END_VAR
24
25  PROCESS Init
26    STATE begin
27      START PROCESS FridgeDoorController;
28      START PROCESS FridgeCompressorController;
29      START PROCESS FreezerCompressorController;
30      STOP;
31    END_STATE
32  END_PROCESS
33
34  PROCESS FridgeDoorController
35    STATE closed
36      IF fridgeDoor = OPEN THEN
37        lighting := TRUE;
38        SET NEXT;
39      END_IF
40    END_STATE
41
42    STATE open
43      IF fridgeDoor = CLOSED THEN
44        lighting := FALSE;
45        SET STATE closed;
46      END_IF
47      TIMEOUT OPEN_DOOR_TIME_LIMIT THEN
48        doorSignal := TRUE;
49        SET NEXT;
50      END_TIMEOUT
51    END_STATE
52
53    STATE longOpen
54      IF fridgeDoor = CLOSED THEN
```

```
      lighting := FALSE;
      doorSignal := FALSE;
      SET STATE closed;
    END_IF
  END_STATE
END_PROCESS

PROCESS FridgeCompressorController
  STATE checkTemp LOOPED
    IF fridgeTempGreaterMax THEN
      fridgeCompressor := TRUE;
    ELSIF NOT fridgeTempGreaterMin THEN
      fridgeCompressor := FALSE;
    END_IF
  END_STATE
END_PROCESS

PROCESS FreezerCompressorController
  STATE checkTemp LOOPED
    IF freezerTempGreaterMax THEN
      freezerCompressor := TRUE;
    ELSIF NOT freezerTempGreaterMin THEN
      freezerCompressor := FALSE;
    END_IF
  END_STATE
END_PROCESS
END_PROGRAM
```

Листинг 12 – Программа управления холодильником

В программе объявлены входные переменные $fridgeTempGreaterMin$, $fridgeTempGreaterMax$, $freezerTempGreaterMin$, $freezerTempGreaterMax$ и $fridgeDoor$ и выходные переменные $fridgeCompressor$, $freezerCompressor$, $lighting$ и $doorSignal$. Первые четыре входных переменных показывают, превышает ли температура в холодильной и морозильной камерах минимальное и максимальное значения. Переменная $fridgeDoor$ показывает, открыта ли дверь холодильной камеры. Выходные переменные $fridgeCompressor$ и $freezerCompressor$ определяют сигналы управления компрессорами холодильника и морозильника соответственно. Переменная $lighting$ определяет, включено ли освещение. Переменная $doorSignal$ задает сигнал о том, что долго открыта дверь холодильника.

В программе определены четыре процесса $Init$, $FridgeDoorController$, $FridgeCompressorController$ и $FreezerCompressorController$. Процесс $Init$ имеет одно состояние $begin$, в котором он запускает остальные процессы и останавливается.

Процесс $fridgeDoorController$ обрабатывает входной сигнал $fridgeDoor$ и в зависимости от него формирует управляющие сигналы $lighting$ и $doorSignal$. Процесс имеет три состояния $closed$, $open$ и $longOpen$. В состоянии $closed$, если дверь холодильника открыта, включается освещение и процесс переходит в состояние $open$. В состоянии $open$, если дверь холодильника закрыта, освещение выключается и процесс переходит в состояние $closed$. В этом состоянии установлен таймаут. По истечении 30 секунд подается сигнал и процесс переходит в состояние $longOpen$. В состоянии $longOpen$, если дверь закрыта, освещение выключается, прекращает подаваться сигнал о том, что долго открыта дверь, и процесс переходит в состояние $closed$.

Процесс $FridgeCompressorController$ управляет компрессором в зависимости от температуры в холодильной камере. В процессе определено состояние $CheckTemp$. Если температура в холодильной камере превышает максимальное значение, включается компрессор $fridgeCompressor$. Если температура ниже минимального значения, компрессор выключается.

Процесс $FreezerCompressorController$ работает аналогично процессу $FridgeCompressorController$, но контролирует температуру в морозильной камере.

Период активации процессов равен 100 миллисекундам.

Для данной программы были сформулированы следующие требования:

1. При открытии двери холодильника включается освещение.
2. При закрытии двери холодильника выключается освещение.
3. Если дверь холодильника открыта, то не более, 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 температура в холодильнике превышает максимальную, включается компрессор.

Для формализации требований используется только инвариант цикла управления, так как ограничений на значения входных переменных нет. Следующая формула является формальной аннотацией для первого требования: $\text{toEnvP}\, s \wedge (\forall s1 s2.\, \text{substate}\, s1\, s2 \wedge \text{substate}\, s2\, s \wedge \text{toEnvP}\, s1 \wedge \text{toEnvP}\, s2 \wedge \text{toEnvNum}\, s1\, s2 = 1 \wedge \text{getVarBool}\, s1\, fridgeDoor' = CLOSED' \wedge \text{getVarBool}\, s2\, fridgeDoor' = OPEN' \longrightarrow \text{getVarBool}\, s2\, lighting')$

Термопот

В данной задаче в качестве управляемого устройства рассматривается 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 ремя кипячения крышка блокируется сигналом (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 соответственн.

Программа управления термопотом на языке poST приведена на листинге 13.

```
PROGRAM Thermopot
  VAR_INPUT
    temperature : INT;
    button1, button2, button3: BOOL;
    boilingButton: BOOL;
  END_VAR

  VAR_OUTPUT
    heater : BOOL;
    lid : BOOL;
    selectedTemp : INT;
    boilingMode, maintainingMode : BOOL;
  END_VAR

  VAR CONSTANT
    LOCKED : BOOL := TRUE;
    UNLOCKED : BOOL := FALSE;
    PRESSED : BOOL := TRUE;

    BOILING_POINT: INT := 100;
    TEMP1 : INT := 98;
    TEMP2 : INT := 85;
    TEMP3 : INT := 60;
  END_VAR
```

```
25
26  PROCESS Init
27    STATE begin
28      START PROCESS TempSelection;
29      START PROCESS HeaterController;
30      STOP;
31    END_STATE
32  END_PROCESS
33
34  PROCESS TempSelection
35    STATE tempSelection LOOPED
36      IF button1 = PRESSED THEN
37        selectedTemp := TEMP1;
38      ELSIF button2 = PRESSED THEN
39        selectedTemp := TEMP2;
40      ELSIF button3 = PRESSED THEN
41        selectedTemp := TEMP3;
42      END_IF
43    END_STATE
44  END_PROCESS
45
46  PROCESS HeaterController
47    STATE begin
48      IF boilingButton = PRESSED THEN
49        boilingMode := TRUE;
50        SET NEXT;
51      END_IF
52    END_STATE
53
54    STATE heating
55      heater := TRUE;
56      lid := LOCKED;
57      IF temperature >= BOILING_POINT THEN
58        heater := FALSE;
59        lid := UNLOCKED;
60        boilingMode := FALSE;
61        maintainingMode := TRUE;
62        SET NEXT;
63      END_IF
64    END_STATE
65
66    STATE maintaining
67      IF boilingButton = PRESSED THEN
```

```
68        maintainingMode := FALSE;
69        boilingMode := TRUE;
70        SET STATE heating;
71      ELSE
72        IF temperature >= selectedTemp THEN
73          heater := FALSE;
74        ELSIF temperature < selectedTemp - 5 THEN
75          heater := TRUE;
76        END_IF
77      END_IF
78    END_STATE
79  END_PROCESS
80 END_PROGRAM
81
```

Листинг 13 – Программа управления термопотом

В программе объявлены входные переменные $temperature$, $button1$, $button2$, $button3$ и $boilingButton$ и выходные переменные $heater$, lid, $selectedTemp$, $boilingMode$ и $maintainingMode$. Переменная $temperature$ показывает температуру воды. Переменные $buuton1$, $button2$ и $button3$ показывают, нажата ли кнопка выпора соответствующего температурного режима. Переменная $boilingButton$ показывает, нажата ли кнопка кипячения. Выходная переменная $heater$ определяет, включен ли нагревательный элемент. Переменная lid определяет сигнал блокировки крышки термопота. Переменная $selectedTemp$ хранит выбранную температуту. Переменные $boilingMode$ и $maintainingMode$ определяют индикаторы режимов кипячения и поддержания температуры соответственно.

В программе определены три процесса: $Init$, $TempSelection$ и $HeaterController$. Процесс $Init$ имеет одно состояние $begin$, в котором он запускает остальные процессы и останавливается. В процессе $TempSelection$ определено состояние $tempSelection$, в котором для различных температурных режимов последовательно проверяется , нажата ли соответствующая кнопка. Если некоторая кнопка нажата, то переменной $selectedTemp$ устанавливается значение, соответствующее данному температурному режиму.

Процесс $HeaterController$ имеет три состояния: $begin$, $heating$ и $maintaining$. В состоянии $begin$, если нажата кнопка кипячения, включается индикатор кипячения и процесс переходит в состояние $heating$. В состоянии $heating$ включается нагревательный элемент и блокируется крышка. Когда температура кипения достигнута, нагревательный элемент выключается, крышка разблокируется, выключается индикатор кипячения и включается индикатор поддержания температуры, процесс переходит в состояние $maintaining$. В состоянии $maintaining$, если кнопка кипячения нажата, выключается индикатор поддержания температуры и включается индикатор кипячения, процесс переходит в состояние $heating$. Если кнопка

кипячения не нажата, температура поддерживается в определенном диапазоне. Если достигнута максимальная температура, нагреватель выключается. Когда температура становится ниже минимального значения, нагревательный элемент включается.

К данной программе были сформулированы следующие требования:

1. Пока требуемая температура не достигнута, крышка заблокирована.
2. При достижении заданной температуры нагревательный элемент отключается.
3. нагревательный элемент включается при понижении температуру воды более, чем на 5 градусов ниже заданной.
4. При нажатии одной из кнопок выбора температурного режима выбирается соответствующая требуемая температура.
5. Блокировка автоматического кипячения воды привключении термолота в сеть. (Если кнопка кипячения не нажата, то нагрев не включится.)

Для формализации требоований используется только инвариант цикла управления, так как ограничений на значения входных переменных нет. Следующая формула является формальной аннотацией для первого требования; $\text{toEnvP}\, s \wedge (\forall s1 s2.\, \text{substate}\, s1\, s2 \wedge \text{substate}\, s2\, s \wedge \text{toEnvP}\, s1 \wedge \text{toEnvP}\, s2 \wedge \text{toEnvNum}\, s1\, s2 = 1 \wedge \text{getVarBool}\, s1\, boilingMode' \wedge \text{getVarInt}\, s2\, temperature' < BOILING_POINT' \longrightarrow \text{getVarBool}\, s2\, lid' = LOCKED')$

ПРИЛОЖЕНИЕ В

Программа управления турникетом на языке poST

```
PROGRAM Turnstile

  VAR_INPUT
    PdOut : BOOL;
    paid : BOOL;
    opened : BOOL;
  END_VAR

  VAR_OUTPUT
    open : BOOL;
    reset : BOOL;
    enter : BOOL;
  END_VAR

  VAR
    passed : BOOL;
  END_VAR

  PROCESS Init
    STATE init
      START PROCESS Controller;
      START PROCESS EntranceController;
      STOP;
    END_STATE
  END_PROCESS

  PROCESS Controller
    STATE isClosed
      IF paid THEN
        open := TRUE;
        passed := FALSE;
        SET NEXT;
      END_IF
    END_STATE

    STATE minimalOpened
      IF PdOut THEN
        passed := TRUE;
      END_IF
```

```
40      TIMEOUT T#1s THEN
41        IF passed THEN
42          open := FALSE;
43          SET STATE isClosed;
44        ELSE
45          SET NEXT;
46        END_IF
47      END_TIMEOUT
48    END_STATE
49
50    STATE isOpened
51      IF PdOut THEN
52        open := FALSE;
53        SET STATE isClosed;
54      END_IF
55      TIMEOUT T#9s THEN
56        open := FALSE;
57        SET STATE isClosed;
58      END_TIMEOUT
59    END_STATE
60  END_PROCESS
61
62  PROCESS EntranceController
63    STATE isClosed
64      IF opened = TRUE THEN
65        enter := TRUE;
66        SET NEXT;
67      END_IF
68    END_STATE
69
70    STATE isOpened
71      IF opened = FALSE THEN
72        enter := FALSE;
73        reset := TRUE;
74        START PROCESS Unlocker;
75        SET STATE isClosed;
76      END_IF
77    END_STATE
78  END_PROCESS
79
80  PROCESS Unlocker
81    STATE unlock
82      TIMEOUT T#1s THEN
```

```
83         reset := FALSE;
84         STOP;
85       END_TIMEOUT
86     END_STATE
87   END_PROCESS
88 END_PROGRAM
```

Листинг 14 – Программа управления турникетом в метро