Федеральное государственное бюджетное образовательное учреждение высшего образования «Саратовский государственный технический университет имени Гагарина Ю.А.»

Институт прикладных информационных технологий и коммуникаций

Кафедра информационно-коммуникационных систем и программной инженерии

Направление 09.03.04 Программная инженерия

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

РАЗРАБОТКА ТЕСТОВОГО ПРИМЕРА МОДЕЛИ ПРОГРАММЫ ДЛЯ ВЕРИФИКАТОРА PAT

Выполнила студентка
Носырева Мария Алексеевна
курс 4
группа бПИНЖ41

Руководитель работы:
к.ф-м.н., доцент кафедры ИКСП
Хворостухина Екатерина Владимировна

Допущен к защите
Протокол № ____22____ от _____3 июня_____2020 г.

Зав. кафедрой ИКСП д.т.н., профессор _____А.А. Сытник

Саратов 2020

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

На настоящий момент аппаратные и программные системы занимают большое место в жизни человека. Общество всё больше зависит от программных и аппаратных систем, которые помогают нам практически во всех аспектах повседневной жизни. Часто мы можем даже не осознавать, что такие системы в это вовлечены. Некоторые функции управления в современных автомобилях основаны на встроенных программных решениях, например, торможение, подушки безопасности, круиз-контроль и так далее. Мобильные телефоны, системы связи, медицинские устройства, аудио- и видеосистемы и бытовая техника в целом содержат большое количество программного обеспечения. Кроме того, в системах транспорта, производства и управления всё шире применяются встроенные программные решения, обеспечивающие гибкость и экономичность. Выходит, что огромная ответственность “ложится на плечи” программ, которые управляют определёнными устройствами.

Следовательно, из-за постоянного роста автоматизации жизни всё более обостряется необходимость преждевременного обнаружения и устранения ошибок в программах. Это имеет большое значение, так как чем раньше ошибка будет обнаружена, тем меньше вреда она принесёт. В ряде приложений ошибки не критичны, например, в офисных приложениях или веб-сервисах. Они сводятся к лёгким моральным травмам пользователей и репутационным издержкам производителя ПО, эти ошибки

достаточно легко обнаружить и выполнить восстановление системы, а также их возможно быстро исправить, например, выпустив патч, либо просто поправив код, например, веб-сервиса на сервере. В таких случаях важнее не избавится от всех ошибок, а как можно быстрее вывести свой продукт на рынок. Но существует ряд систем с повышенными требованиями к надёжности, в которых ошибка в программе может принести вред как со стороны вопроса безопасности, так и с экономической: если ошибка, которая была допущена на этапе проектирования устройства, обнаружится только после старта производства, то появится необходимость в её исправлении уже не исключительно в проекте, но и во всех устройствах, которые были произведены. В истории описано большое количество случаев ошибок, которые привели к большим финансовым потерям. Один из таких случаев произошёл в июне 1996 года: французы запускали ракету Ariane-5, которая взорвалась спустя 40 секунд после старта из-за неверной работы бортового программного обеспечения. На разработку ракеты было потрачено 7 млрд. долларов, ущерб от потери ракеты составил примерно 500 млрд. долларов. Кроме экономической стороны, ошибка в работе систем с повышенными требованиями к надёжности может даже причинить вред здоровью людей, и даже иметь летальные последствия. Примером такой ошибки может быть случай, который произошёл в Саудовской Аравии. В феврале 1991 года ракетный комплекс Patriot не смог перехватить ракету Scud. ЗРК, прикрывавший авиабазу, увидел ракету, но не сумел попасть в нее из-за ошибки в собственном математическом обеспечении. Из-за особенностей хранения

чисел в системе опорное значение времени при округлении искажалось, что накапливало рассогласование примерно на треть секунды за 100 часов непрерывной работы комплекса. Ракета попала на американскую авиабазу. Взрыв убил 28 человек и около сотни получили ранения. Ошибка эта была уже известна до момента трагедии, но из-за начавшейся войны обновить ПО просто не успевали. Носители с «патчем» прибыли только на следующий день после ракетного удара.

Хочется уточнить, что системы с повышенными требованиями к надёжности это не только спутники, самолёты и т.п. Это в том числе системы, в которых:

- затруднено исправление ошибок (потребительская электроника);
- велик масштаб использования (та же потребительская электроника, важные веб-сервисы, которыми пользуются миллионы человек);
- высокая степень доверия человека.

Именно поэтому, так как возрастают размеры и сложности аппаратных и программных систем, необходимо обеспечить процесс валидации систем с использованием методов и инструментов, которые могут облегчить автоматический анализ корректности. Кроме того, верификация системы, исправление ошибок на этапе ее проектирования экономически целесообразно, позволяя экономить время и средства на производство некорректно работающей системы. В то время как тестирование ручным способом не только зачастую не способно обеспечить полной проверки, но и требует наличия уже готового, произведенного продукта.

Поэтому важно изучать методы формальной верификации моделей систем, позволяющей на этапе проектирования производить валидацию модели.

Целью данной работы является разработка тестового примера системы для верификации на основе Model Checking (метода проверки модели). Для достижения данной цели были поставлены следующие задачи:

- описать теоретические сведения о формальной верификации и основах темпоральной логики;
- сравнить существующие инструменты верификации, использующие данный метод;
- на основе полученных данных выбрать наиболее подходящий;
- разработать модель системы «Электрическая плита»;
- сформулировать требования к системе «Электрическая плита»;
- описать модель и требования к ней с помощью выбранного инструмента проверки моделей;
- провести верификацию в выбранном инструменте на построенной модели системы «Электрическая плита».

Данная работа состоит из четырёх глав. В первой главе рассказывается о методах валидации систем, особое внимание уделяется автоматному подходу и методу проверки моделей. Во второй главе производится анализ и сравнение инструментов, использующих метод проверки моделей. На основе анализа происходит выбор необходимого инструмента. В третьей главе описывается моделирование системы «Электрическая плита» и составление требований к ней на

языке линейной темпоральной логики для последующей верификации. В четвёртой главе описывается процесс реализации модели и спецификации в PAT и верификация модели для описанных требований.

Глава 1. Теоретические сведения

1.1 Методы валидации систем

Валидация систем – это исследование и обоснование того, что спецификация ПО и само ПО через реализованную в нём функциональность удовлетворяет требованиям пользователей. Валидация систем является деятельностью всё возрастающей значимости. Это способ поддерживать контроль качества систем. Верификация – это исследование и обоснование того, что программа соответствует своей спецификации [1].

Самыми важными методами валидации систем являются:

- экспертный анализ – процесс получения оценки на основе знания и опыта экспертов;
- тестирование – эффективный метод проверки того, что заданная реализация системы согласуется с абстрактной спецификацией. По своей сути тестирование может быть применено только после реализации прототипа системы;
- симуляция основывается на модели, которая описывает вероятное поведение системы (эта модель в определённом смысле исполнима, при этом программный инструмент (называемый симулятором) имеет возможность определить поведение системы по отношению к некоторым сценариям, то есть пользователь получает определённое понимание того, как система откликается на стимулы; симуляция полезна в основном для быстрой, первоначальной оценки качества системы);
- формальная верификация основывается на математическом (логическом) моделировании программ и

требованиям к ним. Одним из подходов к формальной верификации является проверка моделей. Проверка моделей – это автоматизированный метод, который для заданной модели поведения системы с конечным числом состояний и логического свойства, которое записано в подходящем логическом формализме (обычно в темпоральной логике), проверяет справедливость этого свойства в данной модели.

1.2 Методы формальной верификации

Рассмотрим различные виды формальных методов проверки правильности программ [4,5].

- «Полное тестирование». В основе системного тестирования должно лежать обоснование, почему выполнение предложенного набора тестов будет гарантировать правильность программы. Как правило, в той или иной форме это обоснование сводится к подтверждению выполнения всех пунктов спецификации [13]. Существуют различные варианты обоснования: метод «черного ящика» – это полное покрытие входных данных, то есть мы составляем набор тестов так, чтобы перебрать все возможные входные данные программы и метод «белого ящика» – полное покрытие кода программы в той или иной интерпретации, например, полное покрытие операторов программы, то есть если мы выполним все тесты нашего набора, то через каждый оператор программы мы пройдём хотя бы один раз [14]. Плюсы тестирования очевидны. Во-первых, мы проверяем ту программу, которая будет использоваться в будущем. Во-вторых, не требуется знания дополнительных инструментальных средств, т.е. тестировщику необходимы знания для работы с программой, которую он тестирует,

соответственно тестировщик может быть достаточно неквалифицированным. Также средства откладки позволяют достаточно быстро локализовать источник ошибки и устранить её, что ускоряет переход от нахождения ошибки к её устранению. На первый взгляд, всё, что нужно, это составить правильный набор тестов, который даёт нам полное покрытие и провести эти тесты. Если бы это было возможно, мы бы получили идеальный механизм проверки правильности программ. Кроме этого, в ряде случаев невозможно создать условия для тестирования нашей системы. Особенно часто это случается при тестировании встроенных программ, поскольку для тестирования необходимо создать натурный образец системы и воспроизвести её физическое окружение. Эта проблема частично решается с помощью имитационного моделирования, о котором речь пойдёт далее.

- Имитационное моделирование. Имитационное моделирование – это метод, который позволяет строить модели, которые описывают процессы таким образом, как бы они проходили в действительности. Такую модель можно «проиграть» во времени, как для одной проверки, так и для заданного их множества. При этом итоги станут определяться случайным характером процессов. По этим сведениям, можно получить достаточно устойчивую статистику. Имитационное моделирование можно рассматривать как разновидность экспериментальных испытаний.

- Доказательство теорем. В данном подходе мы рассуждаем о программе как о математическом объекте. Доказательство теорем – это методика система и её

предполагаемые свойства, представляются в виде формул в некоторой математической логике. Данная логика задаётся формальной системой, определяющей набор аксиом и правил вывода. Доказательство теоремы представляет собой процесс построения из аксиом системы доказательства заданного свойства. Шаги доказательства соотносятся с аксиомами и правилами и возможно с некоторыми выведенными определениями, то есть промежуточными леммами. Система и проверяемые свойства описываются с помощью формул, включающих в себя атомарные утверждения (аксиомы), связанные при помощи так называемых логических связок. Система формальных правил вывода используется для вывода одних формул из других, то есть собственно доказательства теорем. Для доказательства правильности программы необходимо показать, что из формулы, описывающей программу, следует формула, описывающая её свойство. Данный метод является чрезвычайно сложным и требует очень высокой квалификации человека, осуществляющего проверку. Кроме того, метод становится практически не применимым в системах с параллельно работающими процессами.

- Статический анализ. Статический анализ программ в отличие от доказательства теорем это количественный анализ. Это более прагматичный и грубый подход, который основан на анализе исходного текста программы без её выполнения. Как правило, решение задачи статического анализа так или иначе сводится к решению задачи достижимости одной вершины из другой в графе управления программой. Известно, что эта задача неразрешима в общем

случае, поэтому статический анализ это всегда поиск компромисса между возможностями (тем, что можно проанализировать) и потребностями (тем, что нужно проанализировать).

- Верификация на моделях. Верификация программ на моделях – это методика, основанная на построении конечной модели системы и проверки, выполняются ли на заданной модели требуемые свойства. В общем, проверка выполняется как исчерпывающий поиск по пространству состояний, который гарантированно заканчивается в силу того, что построенная модель конечна. Подробнее данный метод будет рассмотрен далее.

- Динамическая верификация. Динамическая верификация [7] – это методика, при использовании которой протоколы выполнения программы проверяются на соответствие заданной спецификации. Эта разновидность верификации применяется для проверки поведения программы во время выполнения, например, при отсутствии доступа к коду программы или при исследовании правильности взаимодействия со сторонними компонентами. Хотя верификация протокола конкретного запуска программы не может гарантировать выполнения заданных свойств при любых значениях входных параметров, при правильном подборе тестовых сценариев её результаты могут оказаться достаточно точными. При этом трудоёмкость динамической верификации не зависит от сложности верифицируемой программы, а только лишь от сложности проверяемой спецификации и размера протоколов.

1.3 Конечный автомат и автоматный подход

Под «автоматом» в дальнейшем будет подразумеваться «конечный автомат». Для того, чтобы ввести понятие «конечный автомат», необходимо рассмотреть понятие «абстрактный автомат».

Абстрактный автомат [2] – это математическая модель дискретной системы и описывается шестикомпонентный набором:

$$A=\{X,Y,S,\delta,\lambda,s_0\}, \text{ где}$$

- X – это множество входных сигналов (входной алфавит автомата). Множество входных сигналов представляет собой множество воздействий, которые влияют на поведение исследуемой системы.
- Y – это множество выходных сигналов (выходной алфавит автомата). Множество выходных сигналов представляет собой множество реакций системы на внешние воздействия, интересующих исследователя, иными словами, множество величин, характеризующих поведение системы.
- S – это множество состояний автомата (множество внутренних состояний). Множество состояний – это множество величин, которые не являются входными воздействиями, но от которых также зависит реакция системы.
- $\delta: X \times S \rightarrow S$ – это функция переходов из одного состояния автомата в другое. Функция переходов определяет, в какое состояние перейдёт система в зависимости от входного сигнала и текущего состояния.

- $\lambda : X \times S \rightarrow Y$ – это функция выходов. Функция выходов определяет выходной сигнал системы в зависимости от входного сигнала и текущего состояния.
- $s_0 \in S$ – это начальное состояние автомата, то есть состояние, в котором система начинает функционировать.

Автомат называется конечным, если множества X, Y, S – конечны.

Для задания автомата существует два основных способа: табличный и графовый.

Табличный способ представляет собой табличное описание конечного автомата. Для табличного способа используются две таблицы: таблица переходов и таблица выходов. В таблице переходов каждая строка (столбец) соответствует состоянию входов, а каждый столбец (строка) внутреннему состоянию. Каждая клетка соответствует внутреннему состоянию автомата, в который он должен перейти в следующий момент времени. Вид таблицы выходов зависит от модели автомата (Мили, Мура). Для автомата Мили столбцы соответствуют внутренним состояниям, строки – входным сигналам, в ячейках – выходные сигналы. Для автомата Мура таблица выходов не строится, так как выходной сигнал не зависит от входного. Для автомата Мура строится совмещённая таблица переходов и выходов.

Графовый способ (диаграмма переходов) представляет собой графическое описание конечного автомата. При графовом способе автомат представляется ориентированным графом, вершины которого соответствуют состояниям автомата, а дуги – переходам из состояния в состояние, над которыми принято подписывать соответствующие входные и

выходные сигналы.

Автоматное программирование [9] – это парадигма программирования, при использовании которой программа или ее часть осмысливается как система автоматизированных объектов управления. Особенность такого программирования заключается в явном выделении состояний и переходов между ними. Процесс исполнения программы заключается в последовательных переходах между состояниями и выполнении определенной секции кода (одной и той же для каждого состояния или перехода). Такая технология программирования является удобной при написании определенного класса программ и их последующей верификации.

Данный подход отличается в лучшую сторону от традиционного способа программирования тем, что позволяет явно представлять состояния системы [10]. Одна из центральных идей автоматного программирования состоит в отделении описания логики поведения (при каких условиях необходимо выполнить те или иные действия) от описания его семантики (собственно смысла каждого из действий). Кроме того, описание логики при автоматном подходе жестко структурировано. Эти свойства как раз и делают автоматное описание сложного поведения наглядным и ясным. Это позволяет лучше анализировать работу программ, вносить в них изменения, осуществлять отладку и поиск ошибок. Наличие явно выраженных состояний в программах позволяет упростить процесс их верификации.

1.4 Описание метода Model Checking

Данный раздел рассматривает проверку моделей

программ, построенных на основе автоматного подхода.

Верификация моделей – это набор идей и методов для построения моделей работающих программ, математической формулировки требований к ним, которые отражают правильность их работы (спецификацию), и создания алгоритмов для формальной проверки (доказательства или опровержения) этих требований [6].

Model Checking (проверка моделей) — это метод формальной верификации, который позволяет для заданной модели поведения некоторой системы с конечным числом состояний и логического свойства (требования) проверить, будет ли выполняться это свойство в рассматриваемых нами состояниях данной модели [3]. Подобно компьютерной шахматной программе, которая проверяет все возможные ходы, программный инструмент, который выполняет проверку модели, систематически анализирует все возможные системные сценарии. Таким образом, можно показать, что данная модель системы действительно удовлетворяет определённому свойству.

Основная идея данного метода состоит в запуске алгоритмов, которые исполняются компьютером, для проверки корректности проверяемых систем [18]. Верификация системы состоит из следующих частей:

- построение математической модели анализируемой системы;
- представление проверяемых свойств в виде формального текста (называемого спецификацией);
- построение формального доказательства наличия или отсутствия у системы проверяемого свойства.

Реализация идеи этого метода осуществляется с помощью специальных программ – верификаторов [16]. Эти программы в качестве входных данных получают модель, которая описана на языке выбранного верификатора, и свойство, которое необходимо проверить (рисунок 1). В качестве выходных данных программа-верификатор формирует либо сообщение о том, что свойство, которое мы указали, на заданной нами модели выполняется, либо предоставляет контрпример, который показывает, при каких условиях могло возникнуть данное несоответствие. Контрпример представляет собой некий сценарий (набор определённых действий, условий), в котором модель ведёт себя нежелательным образом. Сценарий, который приводит к нарушению проверяемого свойства. Как правило, это означает, что модель ошибочна и подлежит пересмотру. Хотя в некоторых случаях это может означать неверно заданные формальные требования.

Проверка модели даёт разработчику возможность обнаружить ошибку и исправить или модель, или требования. Если же в итоге ни одной ошибки не было обнаружено, то разработчик может улучшить описание модели (изменить модель, сделав её, например, более близкой к реальности, приняв к сведению больший набор свойств), как правило, за счёт увеличения её размера.

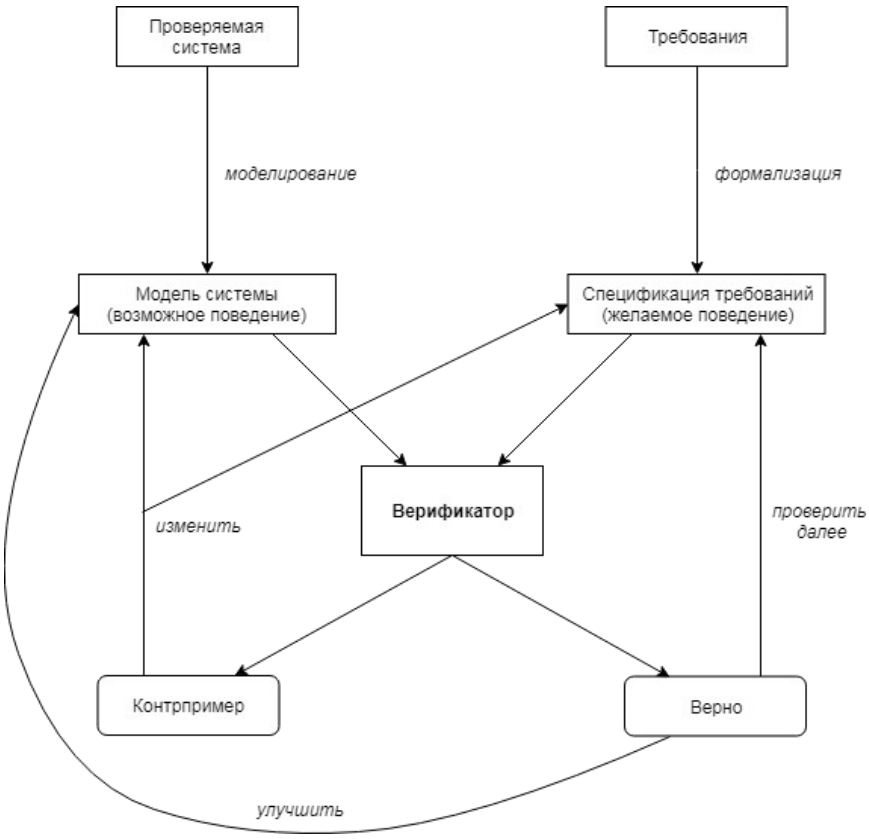

Рисунок 1 - Схема работы метода Model Checking

Основное отличие метода проверки моделей от классической (или хоаровской) верификации заключается в том, что первый метод допускает проверку динамических свойств программ - это те свойства, которые можно записать при помощи темпоральной (временной) логики, а второй метод проводит проверку на соответствие состояние переменных на выходе из программы условиям, которые накладываются на их входное состояние.

Достоинства метода верификации “проверка моделей”:

- проверка моделей полностью автоматическая и быстрая;
- проверка моделей может быть применена к частичным реализациям, поэтому она может предоставлять полезную информацию относительно корректности систем, даже если система полностью не определена;

- программы проверки моделей имеют весьма дружественный интерфейс и легко используются;
- проверка моделей позволяет генерировать контрпримеры, которые впоследствии можно использовать для помощи в отладке.

1.5 Моделирование системы

Как следует из названия рассматриваемого метода формальной верификации – проверка моделей – данный метод работает не напрямую с системой (программой), а с её моделью. Как правило, математическая модель системы представляет собой граф, вершины которого называются состояниями, и изображают ситуации (или классы ситуаций), в которых может находиться система в различные моменты времени; рёбра которого могут иметь метки, изображающие действия, которые может исполнять система.

Функционирование системы изображается в данной модели переходами по рёбрам графа от одного состояния к другому. Если проходимое ребро имеет метку, то эта метка изображает действие системы при переходе от состояния в начале ребра к состоянию в его конце.

Рассмотрим, как построить такую модель [12]. Первым шагом для заданной системы создаётся спецификация - некий набор свойств, которым она должна удовлетворять. Так как в методе проверки моделей проверка спецификации происходит на модели, необходимо, чтобы в модели присутствовали все свойства, которые описаны в спецификации. Таким образом, модель системы не должна быть чрезмерно упрощённой, она должна отражать те аспекты системы, которые имеют отношение к проверяемым

свойствам и должна сохранять все свойства моделируемой системы, представляющие интерес для анализа. В то же время, незачем добавлять в модель те детали системы, которые никаким образом не влияют на исход выполнения требуемых свойств, то есть модель системы не должна быть чрезмерно детальной, так как излишняя сложность модели может вызвать существенные вычислительные проблемы при её формальном анализе.

Работу системы формально можно представить с помощью модели Крипке — это граф переходов, в котором для каждого состояния системы известен набор предикатов, которые выполняются в этом состоянии [17].

Пусть задано множество элементарных высказываний AP (Atomic Propositions). Моделью Крипке над множеством AP называется четвёрка (S,S_0, R, L), где:

- S – непустое множество состояний;
- $S_0 \subseteq S$ – множество начальных состояний;
- $R \subseteq S \times S$ – отношение переходов;
- L: $S \rightarrow 2^{AP}$- функция разметки, помечающая каждое состояние множеством истинных в этом состоянии высказываний.

Структуры Крипке используются для моделирования реагирующих (реактивных) систем – систем, работающих в «бесконечном цикле» и взаимодействующих со своим окружением.

1.6 Описание свойств модели

Реагирующие системы можно характеризовать непрерывным взаимодействием с окружением. Они реагируют на стимулы, которые посылаются извне.

Основной функцией таких систем является поддержание взаимодействия с окружением, а не преобразование информации.

Реагирующие системы отличны от обычных программ преобразования данных (их часто называют трансформационными программами) [11]. Трансформационные программы принимают на вход исходные данные, производят вычисления и выдают результат на выход, после чего работу завершают. Программы поиска экстремума функции, решения дифференциального уравнения численным методом – всё это трансформационные программы. Фактически программы преобразования данных являются функциями, осуществляющими отображение входных данных в выходные, и верификация такой программы состоит в проверке правильности преобразования возможных начальных состояний программы в заключительные её состояния.

Реагирующие системы, напротив, функционируют бесконечно, они контролируют окружение, реагирую на внешние события – получение сообщения, щелчок кнопкой мыши, нажатие клавиши и т.п.[8] В большинстве случаев реагирующие системы не завершаются. Останов таких систем обычно связан с поломкой или коллизией (блокировкой) и является ошибкой. Как правило, реагирующие системы являются частями больших систем, с которыми они взаимодействуют. Операционные системы, протоколы коммуникации, планировщики, контроллеры, параллельные взаимодействующие программы, драйверы – всё это примеры реагирующих систем. Верификация

реагирующей системы не может быть сведена к проверке отношения между входным сигналом и выходным после прихода системы в заключительное состояние: такие системы имеют начальное состояние, но не имеют заключительного состояния, они вообще не должны останавливаться. Поэтому свойства систем принято относить к взаимному порядку событий в системе - поведению системы в течение времени. Требования, которые налагаются на реагирующие системы разработчиками, делятся на два типа:

- свойства безопасности: означают, что что-то нежелательное не случится ни при каких вариантах работы системы;
- свойства живучести: означают, что что-то желаемое при всех вариантах работы рано или поздно произойдёт.

Для того, чтобы достаточно точно описывать эти типы свойств, были определены некие логические формализмы. Самыми известными являются темпоральные логики, которые описывают спецификации систем. С помощью темпоральной логики возможна формулировка свойств поведения систем во времени [15]. Существует несколько интерпретаций темпоральной логики, которые зависят от того, каким образом рассматриваются временные изменения систем.

1) Линейная темпоральная логика (LTL) позволяет формулировать свойства исполнимых вычислительных последовательностей систем. Помимо обычных связок в этой логике используются темпоральные операторы G (Globally), F (in the Future), X (next time) и U(Until), которые интерпретируются следующим образом:

a) G φ – условие φ истинно всегда;

b) Fφ – условие φ истинно сейчас или станет истинным когда-нибудь в будущем;

c) X φ – условие φ истинно в следующий момент времени;

d) φ U ψ – условие φ истинно до тех пор пока не станет истинным условие ψ.

2) Ветвящаяся темпоральная логика (CTL) позволяет пользователю писать формулы, включающие некую зависимость от выборов, которые совершаются в системе в процессе выполнения. Она позволяет формулировать свойства о возможных последовательностях выполнения, стартующих в некотором состоянии.

Глава 2. Сравнение и анализ инструментов проверки моделей

2.1 Сравнение инструментов проверки моделей

Для проверки моделей используются специальные инструменты [19,20, 25-33]. Выделим несколько наиболее известных.

- SPIN. Он, в качестве моделей, использует системы, которые описаны на языке Promela.
- SMV и его обновлённая версия NuSMV. Они осуществляют символическую проверку моделей.
- PAT. Среда для поддержки составления, моделирования и анализа моделей систем.
- PRISM. Является вероятностным символьным верификатором.
- BLAST. Является инструментом анализа C-программ.

- Java Pathfinder. Свободный инструмент, который используется для верификации многопоточных программ на Java.
- CADP. Это набор инструментов, который используется для построения и анализа распределённых систем и протоколов.
- UPPAAL. Набор инструментов, который используется для моделирования, верификации и валидации встроенных систем и систем реального времени. Основа модели – это расширенные временные автоматы.
- KRONOS. Данный верификатор используется для проверки встроенных систем, а также систем реального времени. Основа модели — это расширенные временные автоматы. Инструмент позволяет использовать для описания проверяемых у модели свойств логику с явным временем.

Рассмотрим каждый инструмент по отдельности.

- SPIN [21,22]

Это инструмент формальной верификации распределённых систем.

Проводит проверку с помощью прямого перебора, проверяя спецификацию модели в оперативном режиме. Начал разрабатываться он в 1980 году, а с 1991 находится в свободном распространении, а также имеет открытый код. Этот инструмент очень популярен. Используется во многих организация мира.

У инструмента SPIN есть три основных режима работы.

1) Режим симулятора. Он позволяет осуществить быстрое создание прототипа с управляемым, случайным или интерактивным моделированием.

2) Режим верификатора. Он доказывает правильность и корректность требований, которые указаны пользователем.

3) Режим аппроксимации. Он подтверждает правильность больших по объёму протоколов с максимальным покрытием пространства состояний.

Данный инструмент вполне можно использовать, например, для отслеживания каких-либо ошибок логического дизайна в операционных системах, протоколах обмена данными, в системах переключателей, параллельных и распределённых алгоритмах.

Для описания проверяемых у модели свойств инструмент SPIN использует линейную темпоральную логику (LTL). Кроме этого, он поддерживает использование языка программирования C как фрагмента спецификации модели системы. Это предоставляет возможность проведения прямой верификации уровня реализации спецификации ПО, при помощи использования SPIN как драйвера и логического инструмента для проведения верификации высокоуровневых темпоральных свойств.

Этот инструмент поддерживает язык Promela (сокращение от PROcess MEta Language – язык метапроцессов) — это C-подобный язык для спецификации описания программных систем. Язык позволяет динамически создавать параллельные процессы для моделирования, например, распределённых систем.

Инструмент SPIN работает оперативно (на лету), что означает, что он избавляет от необходимости предварительно строить глобальный граф состояний или модель Крипке, в качестве условия для верификации каких-либо свойств

программной системы. Это позволяет проверить очень большие модели системы. Для ускорения верификации используется техника частичного порядка, кроме этого техника слияния состояний - она производит слияние внутренних (наблюдаемых) переходов процесса для уменьшения количества достижимых состояний модели.

- NuSMV [23,24]

Это инструмент, применяемый для символьной верификации моделей, который является обновлённой версией SMV. Он был разработан как система для верификации моделей систем с открытым кодом.

Данный инструмент может быть использован для верификации промышленных разработок, как ядро составных верификационных инструментов и как основа для тестирования других технологий формальной верификации. Инструмент NuSMV проводит верификацию, сочетая техники использования бинарных диаграмм решений (BDD) и проверку моделей систем, которая основана на SAT.

NuSMV позволяет проверять системы конечных состояний на соответствие спецификациям во временных логиках LTL и CTL. Язык ввода NuSMV предназначен для описания систем конечных состояний, которые варьируются от полностью синхронных до полностью асинхронных. Язык NuSMV (как и язык SMV) обеспечивает модульное иерархическое описание и определение повторно используемых компонентов. Основная цель языка NuSMV – описать (используя выражения в исчислении высказываний) переходное отношение конечной структуры Крипке. Это обеспечивает большую гибкость, но в то же время может

создавать опасность несоответствия (для неопытных пользователей).

Поскольку NuSMV предназначен для описания конечных автоматов, единственными типами данных в языке являются конечные, то есть булевы, скалярные, битовые векторы и фиксированные массивы базовых типов данных.

Если провести сравнение между SMV и NuSMV, то можно обнаружить, что инструмент NuSMV предоставляет такие дополнительные возможности как: взаимодействие конечных автоматов, анализ инвариантов, проверка модели LTL.

- PAT [25,26]

PAT (Process Analysis Toolkit) — это автономная среда для поддержки составления, моделирования и анализа моделей систем (параллельных систем реального времени и других возможных областей).

Данный инструмент имеет дружественный интерфейс, специальный редактор модели и анимированный симулятор. PAT реализует библиотеку методов проверки моделей, предназначенных для проверки взаимоблокировки, достижимости, свойств LTL с допущениями о справедливости, проверка вероятностной модели и т.д. В PAT также реализованы методы оптимизации, например, частичное уменьшение порядка, ограниченная проверка модели, параллельная проверка модели и вероятностная проверка модели. PAT поддерживает как явную проверки модели состояния, так и символическую модель.

Основные функции PAT:

1) удобная среда редактирования для представления моделей;

2) удобный симулятор для интерактивного и визуального моделирования поведения системы; случайным моделированием, пошаговым моделированием под руководством пользователя и т.д.;

3) простая проверка для анализа взаимоблокировки, анализа достижимости, линейная временная логическая проверка состояния или события и уточнение проверки;

4) широкий спектр встроенных примеров, начиная от эталонных систем и заканчивая новыми разработанными алгоритмами.

PAT поставляется с полным и удобным графическим интерфейсом, для того чтобы инструмент было легче использовать.

Графический интерфейс пользователя состоит из следующих частей, позволяющих пользователям редактировать, моделировать и проверять свойства модели соответственно.

1) PAT Editor.

Он обеспечивает более предпочтительный и интеллектуальный способ редактирования вашей модели. Кроме этого, PAT редактор также обеспечивает набор “горячих клавиш” и полный набор функций редактирования.

2) PAT Simulator.

Симулятор PAT позволяет пользователям интерактивно и визуально моделировать поведение системы. Симулятор состоит из четырёх частей: панель инструментов сверху, панель взаимодействия, панель данных (показывает значение

переменных в выбранном состоянии) и граф.

3) PAT Verifier.

Верификатор PAT позволяет пользователям проверять утверждения, перечисленные в модели. Результат проверки будет отображаться в разных цветах, таких как красный для утверждения, которое не действительно, и зелёный для утверждения, которое является действительным.

- PRISM [27]

Это широко используемый, мощный инструмент для формального моделирования и анализа систем, которые демонстрируют случайное или вероятностное поведение. Он используется для анализа систем из разных областей применения, включая протоколы связи и мультимедиа, рандомизированные распределённые алгоритмы, протоколы безопасности, биологические системы и многие другие.

PRISM может строить и анализировать несколько типов вероятностных моделей:

1) цепи Маркова с дискретным временем (DTMC);

2) цепи Маркова с непрерывным временем (CTMC);

3) марковские процессы принятия решений (МДП);

4) вероятностные автоматы (ПА);

5) синхронизированные автоматы (PTA).

Модели описываются с использованием языка PRISM. PRISM обеспечивает поддержку автоматического анализа широкого спектра количественных свойств этих моделей, например, «какова вероятность сбоя, приводящего к выключению системы в течении 4 часов? », «каково ожидаемое наихудшее время, необходимое для завершения алгоритма? ». Язык спецификации свойств включает

временную логику CSL, PLTL, PCTL.

PRISM включает в себя современные символические структуры данных и алгоритмы, основанные на BDD (двоичные диаграммы принятия решений) и MTBDD (многотерминальные двоичные диаграммы принятия решений). Он также включает механизм моделирования дискретных событий, обеспечивающий поддержку приближенной/статистической проверки моделей, и реализации различных методов анализа, таких как количественное уточнение абстракции и уменьшение симметрии.

- Java PathFinder [29]

Данный инструмент используется для верификации многопоточных Java программ. По сути, это методы проверки моделей, которые реализованы на основе виртуальной Java машины. Это значит, что Java PathFinder выполняет программу не один раз, как обычная виртуальная машина, а по всем возможным путям, связанным с переключением потоков планировщиком. На вход Java Pathfinder принимает программы, написанные на языке Java. Он обнаруживает такие ошибки как тупики, необработанные исключения, а кроме этого нарушения условий, которые заданы пользователем в виде assert выражений.

Данный инструмент был разработан в NASA. Применяется в NASA Ames Research Center.

- CADP [24,30]

Это набор инструментов для проектирования асинхронных параллельных систем, таких как протоколы связи, распределённые системы, асинхронные схемы,

многопроцессорные архитектуры, веб-сервисы и так далее. CADP может применяться к любой системе, которая включает асинхронные параллелизм, т.е. к любой системе, поведение которой может быть смоделировано как набор параллельных процессов, управляемых семантикой чередования.

Целью инструментария CADP является облегчение проектирования надежных систем путем использования методов формального описания вместе с программными инструментами для моделирования, быстрой разработки приложений, проверки и генерации тестов.

CADP предлагает довольно широкий спектр функциональных возможностей: от пошагового моделирования до массовой параллельной проверки моделей. Обеспечивает эффективную компиляцию, симуляцию, формальную верификацию и тестирование свойств, которые записаны на языке LOTOS. LOTOS может описывать как асинхронные параллельные процессы, так и сложные структуры данных.

Кроме этого, CADP включает в себя программу EVALUATOR (данная программа используется для проведения верификации оперативно (на лету)) и XTL — это язык, который реализует различные темпоральные логики.

- BLAST [28]

BLAST (Berkeley Lazy Abstraction Software Verification Tool) – это средство проверки моделей, которое используется для верификации программ, написанных на таком языке программирования как C и предназначен для решения задачи достижимости.

Цель BLAST – убедиться, что программное обеспечение удовлетворяет поведенческим свойствам используемых им интерфейсов. Для этого BLAST использует метод абстракции предикатов для построения абстрактной модели, которая впоследствии будет проверяться на свойства безопасности. Данный инструмент способен строить абстракции оперативно (на лету) с требуемой точностью, которая устанавливается в ходе анализа. Если дана программа на C, функция main (так называемая «точка входа») и имя метки, инструмент проверяет, существует ли такое выполнение программы, которое начинается в точке входа и заканчивается в месте, отмеченном заданной меткой.

Инструмент анализирует программу с помощью подхода CEGAR (Counter-Example Guided Abstraction Refinement) – уточнения абстракции на основе контрпримеров. BLAST реализует CEGAR методом «ленивой абстракции» («Lazy Abstraction»). Суть этого метода заключается в том, что после осуществления анализа контрпримера (потенциального ошибочного пути) не осуществляется построение всего абстрактного дерева для всей программы «с нуля». Вместо этого, дерево пересматривается частично, только для тех путей, в которых его пересмотр мог бы повлиять на наличие ошибочных состояний. При анализе пути контрпримера алгоритм ленивой абстракции находит в нём самую первую с конца пути вершину, из которой ошибка уже недостижима. Таким образом, данный метод позволяет выполнять меньше работы, не уточняя ту часть абстракции, которая не должна измениться после анализа контрпримеров, вместо того чтобы пересматривать её всю целиком.

- UPPAAL [31,32]

Это инструментальная среда для моделирования, симуляции и верификации систем реального времени, моделируемых как сети синхронизированных автоматов, расширенных типами данных (целые числа, массивы и т.д.).

Типичные области применения включают в себя контроллеры реального времени и, в частности, протоколы связи, в которых аспекты синхронизации имеют решающее значение. Требования корректности задаются в подъязыке TCTL. Данная среда свободно доступна для некоммерческих целей. UPPAAL состоит из трёх основных частей: язык описания, симулятор, средство проверки моделей.

Язык описания представляет собой недетерминированный командный язык с типами данных (например, целые числа, массивы и т.д.). Он служит языком моделирования для описания поведения системы как сетей автоматов, расширенных с помощью переменных часов и данных.

Симулятор – это инструмент проверки, который позволяет исследовать возможные динамические исполнения системы на ранних стадиях проектирования (или моделирования).

Средство проверки моделей может проверять свойство инварианта и достижимости, исследуя пространство состояний системы.

Два основных критерия дизайна Uppaal – эффективность и простота использования. Применение технологии поиска «на лету» имеет решающее значение для эффективности проверки модели в данной инструментальной среде. Другим

важным ключом к эффективности является применение символической техники, которая сводит проблемы верификации к проблемам эффективной манипуляции и решения ограничений. Чтобы упростить моделирование и откладку, средство проверки моделей Uppaal может автоматически генерировать диагностическую трассировку, которая объясняет, почему свойство удовлетворяется (или не удовлетворяется) описанию системы. Диагностические трассы, сгенерированные средством проверки моделей, могут автоматически загружаться в симулятор, который может использоваться для визуализации и исследования трассы.

- KRONOS [33]

Это инструмент, разработанный с целью проверки сложных систем реального времени.

Типичные области применения включают в себя протоколы связи в реальном времени, синхронизированные асинхронные схемы и гибридные системы.

В KRONOS компоненты систем реального времени моделируются временными автоматами (это автоматы, которые расширены конечным набором действительных таймеров, которые используются для выражения временных ограничений), а требования к корректности, выражаются во временной логике TCTL. Алгоритмы проверки моделей, которые проверяют, удовлетворяет ли система свойству, указанному в формуле TCTL, могут быть классифицированы в две общие категории в соответствии со стратегией, применяемой для исследования пространства состояний: обратный анализ и прямой анализ. Алгоритмы обратного анализа выполняют обратный поиск графа достижимости для

вычисления набора предшественников заданного набора состояний. Алгоритмы прямого анализа создают множество преемников, выполняя прямое исследование графа.

Данный инструмент проверки моделей является свободно распространяемым.

2.2 Результаты сравнительного анализа

В таблице 1 приведены результаты сравнения инструментов проверки моделей по таким критериям как: режим, язык моделирования, тип используемой темпоральной логики, наличие графического интерфейса, лицензия и используемая операционная система.

Таблица 1 - Сравнение инструментов проверки моделей

Верификатор	Режим	Язык моделирования	Тип темпоральной логики	GUI	Лицензия	ОС
SPIN	- Симуляция - Верификация - Апроксимация	Promela	LTL	+	Свободная	Linux, Mac OS X, Windows
NuSMV	Верификация	SMV input language	CTL, LTL	-	Свободная	Linux, Mac OS X, Windows
PRISM	Верификация вероятностн	PEPA, PRISM language	CSL, PLTL, PCTL	+	Свободная	Linux, Mac OS X,

	ых систем					Windows

Продолжение таблицы 1

PAT	-Симуляция -Верификация	CSP#, Timed CSP, Probabilistic CSP	LTL	+	Свободная	Windows
Java Pathfinder	Виртуальная машина	Java	unknown	+	Свободная	Linux, Mac OS X, Windows
CADP	Верификация	LOTOS, LNT	AFMC, MCL, XTL	+	Свободная для некоммерческого использования	Linux, Mac OS X, Solaris, Windows
BLAST	Верификация	C	Monitor automata	+ (плагин для Eclipse)	Свободная	Windows, связанные с Unix
UPPAAL	-Симуляция -	Timed Automata,	TCTL	+	Свободная для	Linux, Mac OS

	Верификация систем реального времени	C subset			некоммерческого использования	X, Windows

Продолжение таблицы 1

KRONOS	Верификация систем реального времени	Timed automata	TCTL	-	Свободная для некоммерческого использования	Linux, Solaris, Windows (требуется Cygwin)

На основе проведённого сравнения и анализа инструментов, использующих метод проверки моделей, для дальнейшей работы был выбран инструмент PAT, так как он имеет дружественный графический интерфейс, который включает и редактор моделей, и анимированный симулятор, кроме этого, он является довольно современным инструментом и позволяет выразить требуемые свойства поведения (т.н. «темпоральные свойства») на языке логики линейного времени LTL.

Глава 3. Разработка модели электрической плиты

3.1 Описание модели системы управления электрической плитой

Для дальнейшей работы необходимо спроектировать модель электрической плиты (варочной поверхности). Рассмотрим часть автоматной программы, которая отвечает за логику управления электрической плитой. Электрическая плита в данном случае представляет собой варочную поверхность, которая содержит две конфорки и дисплей. Количество температурных режимов в данной модели изменяется от нуля до шести. Регулировать температурные режимы каждой горелки можно с помощью кнопок «+» (увеличить температурный режим на единицу) и «-» (уменьшить температурный режим на единицу). При попытке увеличить максимальное значение температурного режима нажатием кнопки «+» ничего не происходит. Аналогично и для минимального значения температурного режима – нажатие кнопки «-» игнорируется.

Пользователь может задать таймер работы нагрева конфорок. Если время на таймере истекло, то система выключается. Также предусмотрена возможность блокировки и разблокировки дисплея плиты от случайных нажатий. В случае попадания каких-либо предметов на индикаторы варочной поверхности, система выдаёт звуковой сигнал и выключается.

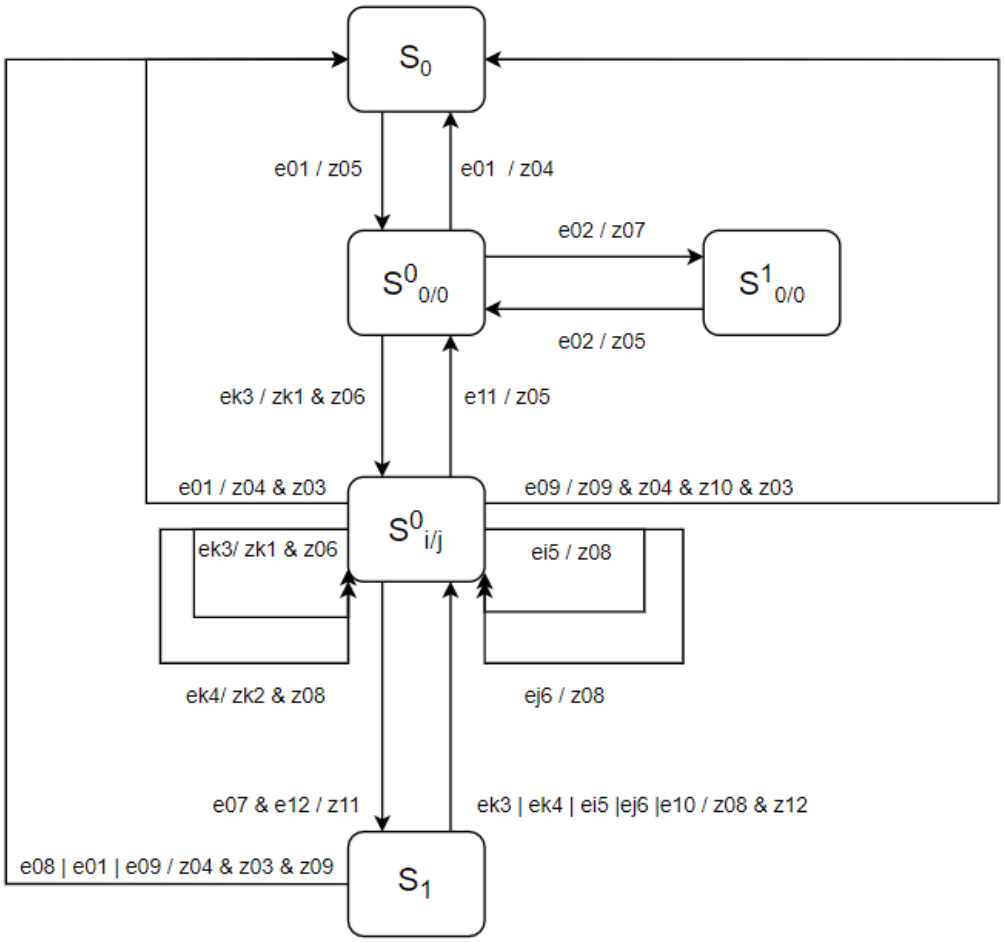

Рисунок 2 - граф переходов автомата "Электрическая плита"

На рисунке 2 изображён граф переходов автомата «Электрическая плита» со следующими входными воздействиями (где k=1,2, i,j=0,..,6):

- e01 - нажата кнопка включить/выключить;
- e02 - нажата кнопка блокировки плиты;
- ek3 - нажата кнопка “+” на k-конфорке;
- ek4 - нажата кнопка “-” на k-конфорке;
- ei5 – выбран i-режим для 1 конфорки;
- ej6 – выбран j-режим для 2 конфорки;
- e07 – запуск таймера;
- e08 – таймер истёк;
- e09 – попадание предмета на индикаторы;
- e10 – сброс таймера;

- e11 – обе конфорки находятся в начальном состоянии;
- e12 – нажата кнопка включения таймера.

Выходными воздействиями являются:

- zk1 - увеличить температуру нагревателя k-конфорки;
- zk2 - уменьшить температуру нагревателя k-конфорки;
- z03 - сброс в начальное состояние всех конфорок;
- z04 - индикация выключения на дисплее электрической плиты;
- z05 - индикация готовности на дисплее электрической плиты;
- z06 - индикация нагрева на дисплее электрической плиты;
- z07 - индикация блокировки на дисплее электрической плиты;
- z08 - индикация выполнения на дисплее электрической плиты;
- z09 – звуковой сигнал;
- z10 – индикация неисправности на дисплее электрической плиты;
- z11 – запуск таймера;
- z12 – сброс таймера.

Состояния автомата:

- S_0 – плита выключена;
- $S^0_{0/0}$ – плита готова к работе;
- $S^1_{0/0}$ – плита заблокирована;
- $S^0_{i/j}$ – нагрев конфорок;

- S_1 – ожидание во время нагрева.

Данный автомат реагирует на нажатие кнопок на дисплее (при включении, при указании температурного режима, при блокировке), оповещая о результате действия путём индикации на дисплее электрической плиты или с помощью звукового сигнала. Информация о текущих состояниях модели отображается на дисплее плиты.

3.2 Спецификация системы

Для того чтобы описать спецификацию LTL-формул, нужно выделить атомарные предложения AP = {working, $modeBurner1_i$}, где working означает, что электрическая плита в данный момент включена и работает, а $modeBurner1_i$ означает i-ый режим 1 конфорки ($0 \leq i \leq 6$).

Сформулируем следующие требования к модели «Электрическая плита» на языке LTL.

- Достижимость свойства «Нажатие кнопки включения электрической плиты».

$$F[e01] \tag{1}$$

- Достижимость свойства «плита не заблокирована».

$$F[\neg S^1_{0/0}] \tag{2}$$

- Во всех будущих состояниях после включения электрической плиты в следующем состоянии не будет запущен таймер, пока не будет включён нагреватель на одной из конфорок.

$$G[S^0_{0/0} \rightarrow X(\neg e07 U S^0_{i/j})] \tag{3}$$

- После запуска таймера в следующем состоянии таймер не будет сброшен, пока не будет нажата какая-либо кнопка, относящаяся к регулировке температурного режима

конфорок, или пока какой-либо предмет не попадёт на индикаторы варочной поверхности.

$$(e07 \rightarrow X(\neg e10\, U(ek3 | ek4 | ei5 | ej6 \vee e09 ¿¿¿ \quad (4)$$

- Во всех будущих состояниях если процесс вошёл в состояние «плита заблокирована», то когда-нибудь в будущем он из него выйдет, и плита будет разблокирована.

$$G[S^1_{0/0} \rightarrow F \neg S^1_{0/0}] \quad (5)$$

- Система будет работать, пока кто-то не нажмёт кнопку выключения или не будет закончено время на таймере (в случае если таймер запущен), или пока какой-либо предмет не попадёт на индикаторы варочной поверхности.

$$working \rightarrow (working\, U(e01 | e08 | e09 ¿¿ \quad (6)$$

- Свойство «Система работает» будет истинным бесконечное число раз на всех траекториях системы.

$$G\, F[working] \quad (7)$$

- Хотя бы в одном состоянии в будущем будет включен нагреватель для первой конфорки.

$$F ¿ \quad (8)$$

- По истечении таймера система переходит в нулевое состояние «Плита выключена».

$$G ¿ \quad (9)$$

- При попадании какого-либо предмета на индикаторы система переходит в состояние «Плита выключена».

$$G[e09 \rightarrow F\, S_0] \quad (10)$$

- Регулировка температурного режима 1 конфорки. Температурный режим не может быть меньше 0 и больше максимального температурного режима, который равен 6.

$$G[0 \le modeBurner1 \le modeBurner1_{max}] \quad (11)$$

Данные требования и будут проверяться с помощью инструмента проверки моделей PAT.

Глава 4. Реализация и верификация

4.1 Описание работы в среде PAT

Описывать модель в среде PAT мы будем с помощью языка CSP# (произносится как «CSP sharp», сокращение от Communicating Sequential Programs) [26]. Данный язык моделирования объединяет высокоуровневые операторы моделирования такие как (условный или недерминированный) выбор, прерывание, (алфавитная) параллельная композиция, чередование, скрытие, асинхронный канал передачи сообщений с предпочтительными для программиста низкоуровневыми конструкциями, такими как переменные, массивы, if-then-else, while и т.д. Это обеспечивает большую гибкость при моделировании систем. Например, взаимодействие между процессами может быть основано либо на общей памяти (с использованием глобальных переменных), либо на передаче сообщений (с использованием асинхронной передачи сообщений) [37]. Операторы высокого уровня основаны на классической алгебре процессов.

Языковые конструкции CSP# можно разделить на следующие группы.

- Первая группа – это базовое подмножество операторов CSP, включая префиксы событий, внутренние/внешние выборы, алфавитную синхронизацию шагов блокировки, условное ветвление, прерывание, рекурсию и т.д.

- Вторая группа включает в себя те языковые конструкции, которые можно рассматривать как «синтаксический сахар» (для CSP), включая глобальные переменные и асинхронные каналы. CSP способен моделировать общие переменные или асинхронные каналы как процессы. Тем не менее, специальные языковые конструкции обеспечивают удобство использования и могут повысить эффективность проверки.
- Третья группа представляет собой набор аннотаций к событиям. Известно, что алгебра процессов, такая как CSP или CCS, определяет только безопасность. Аннотации событий предлагают гибкий способ моделирования справедливости, используя основанный на событиях композиционный язык.
- Последняя группа – это язык для формулировки утверждений, который позже может быть автоматически проверен с использованием встроенных верификаторов.

Начать разработку следует с объявления необходимых констант и переменных. Для этого необходимо использовать следующие зарезервированные в PAT слова.

- #define является зарезервированным словом для определения константы или свойства. Константу можно задать только целочисленным значением (как положительным, так и отрицательным) и логическим значением (истина или ложь).
- var – это зарезервированное ключевое слово для представления переменных или массивов. Язык ввода PAT слабо типизирован, и поэтому при объявлении переменной не требуется никакой информации о типе переменной.

Для нашей модели необходимо объявить следующие константы и переменные.

- Константа M. Данная константа используется для хранения числа температурных режимов. В нашем случае равна шести.
- Константа timer. Данная константа используется для хранения максимального значения времени на таймере. Время в данном случае исчисляется в секундах. В данной системе задано равным 5.
- Переменная time. Данная переменная хранит значение, соответствующее времени, прошедшему с момента запуска таймера в системе. Изначально задано равным 0.
- Переменная buttonTimer. Данная переменная хранит значение обозначающее нажатие кнопки запуска таймера. Она имеет значение false, если кнопка не нажата, и значение true, если кнопка нажата.
- Переменная buttonOn. Данная переменная хранит значение, обозначающее нажатие кнопки включения. Она имеет значение false, если кнопка не нажата, и значение true, если кнопка нажата.
- Переменная buttonBlock. Данная переменная хранит значение, обозначающее нажатие кнопки блокировки. Она имеет значение false, если кнопка не нажата, и значение true, если кнопка нажата.
- Переменные mode1 и mode2. Данные переменные хранят значения от 0 до M и обозначают выбор температурного режима для первой и второй конфорки соответственно. В данной системе значения заданы от 0 до 6.

- Переменные buttonPlus и buttonPlus2. Данные переменные хранят значения, обозначающие нажатие кнопок “+” на первой и на второй конфорке соответственно. Они имеют значение false, если кнопки не нажаты, и значение true, если кнопки нажаты.
- Переменные buttonMinus и buttonMinus2. Данные переменные хранят значения, обозначающие нажатие кнопок “-” на первой и на второй конфорке соответственно. Они имеют значение false, если кнопки не нажаты, и значение true, если кнопки нажаты.
- Переменная someObjectOnStove. Данная переменная хранит значение, обозначающее какой-либо предмет, находящийся на варочной поверхности вне конфорки. Она имеет значение false, если на варочной поверхности нет посторонних предметов, true – если есть.
- Переменные modeBurner1 и modeBurner2. Данные переменные хранят значения от 0 до M и обозначают текущий температурный режим первой и второй конфорки соответственно. В данной системе значения заданы от 0 до 6.
- Переменная working. Данная переменная хранит значение, соответствующее текущему рабочему состоянию электрической плиты (включена/выключена). Она имеет значение false, если плита выключена, и значение true, если плита включена.
- Переменная blocking. Данная переменная хранит информацию о состоянии блокировки электрической плиты. Она имеет значение false, если плита не заблокирована, и значение true, если плита заблокирована.

Объявление переменных в PAT представлено на рисунке 3.

```
#define M 6;
#define timer 5;

var time = 0;
var buttonTimer = false;
var buttonOn = true;
var buttonBlock = false;

var mode1:{0.. M} = 0;
var mode2:{0.. M} = 1;
var buttonPlus = true;
var buttonPlus2 = false;
var buttonMinus = false;
var buttonMinus2 = true;
var someObjectOnStove = false;

var modeBurner1:{0.. M} = 0;
var modeBurner2:{0.. M} = 0;

var working = false;
var blocking = false;

```

Рисунок 3 - Объявленные переменные в PAT

Далее нам необходимо объявить в модели процессы. Процесс в PAT определяется как уравнение в следующем синтаксисе:

$P(x_1, x_2, \ldots, x_n) = \exp,$

где P – это имя процесса, $x_1, x_2, \ldots, x_n$ – это необязательный список параметров процесса, а exp – это выражение процесса. Выражение процесса определяет вычислительную логику процесса. Процесс без параметров записывается как P() или P.

В данном контексте существует также понятие события. Событие – это абстракция наблюдения. Для данного процесса P ниже описывается процесс, который сначала выполняет e, а затем ведёт себя так, как указано процессом P.

$e \rightarrow P,$

где e – это событие. Имя события – это произвольная строка формы ('a' .. 'z' | 'A' .. 'Z' | '_') ('a' .. 'z' | 'A' .. 'Z '|' 0'.. '9' | '_') *. Однако имена глобальных переменных, имена глобальных констант, имена процессов, имена параметров процесса не могут использоваться в качестве имени события.

К событию может быть присоединён блок операторов последовательной программы (который может содержать локальные переменные, if-then-else, while, математическую функцию и т.д.). Последовательная программа рассматривается как атомарное действие. То есть, нет чередования других процессов до завершения последовательной программы. Другими словами, после запуска последовательная программа продолжает выполняться до тех пор, пока не завершится без прерывания. С другой точки зрения, событие можно рассматривать как помеченный фрагмент кода. Имя события используется для построения значимых контрпримеров.

В данной модели задано пять процессов.

- Процесс Stove(). Данный процесс описывает собственно работу системы электрической плиты.
- Процесс Burner(). Данный процесс описывает работу нагревателей на конфорках.
- Процесс Burner1(). Данный процесс описывает работу нагревателя на первой конфорке.
- Процесс Burner2(). Данный процесс описывает работу нагревателя на второй конфорке.
- Процесс Timer(). Данный процесс описывает работу таймера в данной системе.

На рисунке 4 представлено описание процесса Stove()

нашей модели в инструменте PAT и таких событий как:

- StartStove – событие обозначающее включение электрической плиты;
- BlockStove – событие обозначающее блокировку дисплея электрической плиты;
- UnblockStove – событие обозначающее разблокировку дисплея электрической плиты;
- OnBurner – событие обозначающее включение нагрева одной из конфорок электрической плиты;
- SomeObjectOnStove – событие обозначающее, что на индикаторах варочной поверхности электрической плиты находится посторонний предмет;
- FinishStove – событие обозначающее выключение электрической плиты.

```
Stove() = [working == false && buttonOn == true]
            StartStove{working = true; buttonOn = false} -> Stove()
        [] [working == true && ((blocking == false && buttonBlock == true) || (blocking == true && buttonBlock == false))]
            BlockStove{blocking = true; buttonBlock = false} -> Stove()
        [] [working == true && blocking == true]
            BlockStove{blocking = true} -> Stove()
        [] [working == true && blocking == true && buttonBlock == true]
            UnblockStove{blocking = false; buttonBlock = false} -> Stove()

        [][working == true && blocking != true && buttonBlock == false
            && ((mode1 > 0 || buttonPlus == true) || (mode2 > 0 || buttonPlus2 == true))]
            OnBurner{working=true}->Burner()

        [] [working == true && someObjectOnStove == true]
            SomeObjectOnStove{buttonOn = false;working = false;blocking=false;buttonBlock=false;buttonPlus = false;buttonMinus = false;
                              buttonPlus2 = false;buttonMinus2 = false;modeBurner1 = 0;modeBurner2 = 0;someObjectOnStove = false;} -> FinishStove -> Stove()
        [] [working == true && buttonOn == true]
            FinishStove{buttonOn = false;working = false;blocking=false;buttonBlock=false;buttonPlus = false;buttonMinus = false;
                        buttonPlus2 = false;buttonMinus2 = false;modeBurner1 = 0;modeBurner2 = 0;someObjectOnStove = false;} -> Stove()
;
```

Рисунок 4 - Описание процесса Stove() в PAT

На рисунке 5 представлено описание процесса Burner() нашей модели в инструменте PAT и таких событий как:

- OnBurner1 – событие обозначающее включение нагрева на первой конфорке электрической плиты;
- OnBurner2 – событие обозначающее включение нагрева на первой конфорке электрической плиты;

- OffBurner – событие обозначающее выключение нагревателя на обеих конфорках электрической плиты;
- StartTimer – событие обозначающее включение таймера.

```
Burner() = [(mode1 > 0 || buttonPlus == true)]OnBurner1{working=true}->Burner1()
        [][(mode2 > 0 || buttonPlus2 == true)]OnBurner2{working=true}->Burner2()
        [][modeBurner1 == 0 && buttonPlus == false && buttonMinus == false && mode1 == 0 && modeBurner2 == 0
                && buttonPlus2 == false && buttonMinus2 == false && mode2 == 0]OffBurner{working = true;} -> Stove()
        [] [((modeBurner1 > 0 || modeBurner2 > 0) && buttonPlus == false && buttonMinus == false && buttonPlus2 == false
                && buttonMinus2 == false && mode1 == 0 && mode2 == 0 && buttonTimer == true)]
            StartTimer{time = 0;} -> Timer();
```

Рисунок 5 - Описание процесса Burner() в PAT

На рисунке 6 представлено описание процесса Burner1() нашей модели в инструменте PAT и таких событий как:

- OnModeBurner1 – событие обозначающее включение определённого температурного режима для первой конфорки электрической плиты;
- PlusModeBurner1 – событие обозначающее увеличение температурного режима для первой конфорки электрической плиты на единицу;
- MinusModeBurner1 – событие обозначающее уменьшение температурного режима для первой конфорки электрической плиты на единицу;
- WaitBurner1 – событие обозначающее ожидание действия со стороны пользователя для первой конфорки электрической плиты;
- OffBurner1 – событие обозначающее выключение нагревателя на первой конфорке электрической плиты.

```
Burner1() = [mode1 > 0 && buttonPlus == false]OnModeBurner1 {modeBurner1 = mode1; mode1 = 0} -> Burner1()
        [] [modeBurner1!=M && buttonPlus == true]PlusModeBurner1{modeBurner1++; buttonPlus=false} -> Burner1()
        [] [modeBurner1!=0 && buttonMinus == true]MinusModeBurner1{modeBurner1--; buttonMinus=false} -> Burner1()
        [] [modeBurner1 > 0 && buttonPlus == false && buttonMinus == false && mode1 == 0]WaitBurner1{working = true} -> Burner()
        [] [modeBurner1 == 0 && buttonPlus == false && buttonMinus == false && mode1 == 0]OffBurner1{working = true} -> Burner();
```

Рисунок 6 - Описание процесса Burner1() в PAT

На рисунке 7 представлено описание процесса Burner2()

нашей модели в инструменте PAT и таких событий как:

- OnModeBurner2 – событие обозначающее включение определённого температурного режима для второй конфорки электрической плиты;
- PlusModeBurner2 – событие обозначающее увеличение температурного режима для второй конфорки электрической плиты на единицу;
- MinusModeBurner2 – событие обозначающее уменьшение температурного режима для второй конфорки электрической плиты на единицу;
- WaitBurner2 – событие обозначающее ожидание действия со стороны пользователя для второй конфорки электрической плиты;
- OffBurner2 – событие обозначающее выключение нагревателя на второй конфорке электрической плиты.

```
Burner2() = [mode2 > 0 && buttonPlus2 == false]OnModeBurner2{modeBurner2 = mode2; mode2 = 0} -> Burner2()
        [] [modeBurner2!=M && buttonPlus2 == true]PlusModeBurner2{modeBurner2++; buttonPlus2=false} -> Burner2()
        [] [modeBurner2!=0 && buttonMinus2 == true]MinusModeBurner2{modeBurner2--; buttonMinus2=false} -> Burner2()
        [] [modeBurner2 > 0 && buttonPlus2 == false && buttonMinus2 == false && mode2 == 0]WaitBurner2{working = true} -> Burner()
        [] [modeBurner2 == 0 && buttonPlus2 == false && buttonMinus2 == false && mode2 == 0]OffBurner2{working = true} -> Burner();
```

Рисунок 7 - Описание процесса Burner2() в PAT

На рисунке 8 представлено описание процесса Timer() нашей модели в инструменте PAT и таких событий как:

- TimeIsUp – событие обозначающее, что время на таймере подошло к концу и плита будет выключена;
- Tick – событие обозначающее, что прошла секунда на таймере;
- ResetTimer – событие обозначающее сброс времени на таймере и его остановку;

- SomeObjectOnStove – событие обозначающее, что на варочной поверхности электрической плиты находится посторонний предмет.

```
Timer() = [(time == timer)]
            TimeIsUp{time = 0;working = false;blocking=false;buttonTimer = false;buttonOn = false;buttonBlock=false;buttonPlus = false;buttonMinus = false;
                     buttonPlus2 = false;buttonMinus2 = false;modeBurner1 = 0;modeBurner2 = 0;someObjectOnStove = false;} -> FinishStove -> Stove()
        [][(buttonPlus == false && buttonMinus == false && buttonPlus2 == false && buttonMinus2 == false && time != timer)]
            Tick{if (time < timer) time++;} -> Timer()
        [][buttonPlus == true || buttonMinus == true || buttonPlus2 == true || buttonMinus2 == true || buttonTimer == false]
            ResetTimer{time = 0;} -> Stove()
        [][someObjectOnStove == true]
            SomeObjectOnStove{buttonOn = false;working = false;blocking=false;buttonBlock=false;buttonPlus = false;buttonMinus = false;
                              buttonPlus2 = false;buttonMinus2 = false;modeBurner1 = 0;modeBurner2 = 0;someObjectOnStove = false;} -> FinishStove -> Stove()
;
```

Рисунок 8 - Описание процесса Timer() в PAT

Теперь мы можем с помощью симулятора PAT попробовать интерактивно и визуально смоделировать поведение системы.

```
1   #define M 6;
2   #define timer 5;
3
4   var time = 0;
5   var buttonTimer = true;
6   var buttonOn = true;
7   var buttonBlock = false;
8
9   var mode1:{0.. M} = 2;
10  var mode2:{0.. M} = 0;
11  var buttonPlus = false;
12  var buttonPlus2 = true;
13  var buttonMinus = false;
14  var buttonMinus2 = false;
15  var someObjectOnStove = false;
16
17  var modeBurner1:{0.. M} = 0;
18  var modeBurner2:{0.. M} = 0;
19
20  var working = false;
21  var blocking = false;
22
23
```

Рисунок 9 - Объявление переменных в PAT

Например, при начальных данных, представленных на рисунке 9, мы получим граф, представленный на рисунке 10. Изначально наша электрическая плита была выключена, произошло нажатие кнопки «On», о чём говорит переменная buttonOn, значение которой равно true. Далее, режим для первой конфорки изначально задан как второй температурный режим, то есть мы включаем первую

конфорку во втором температурном режиме. У нас нажата кнопка «+» для второй конфорки, таким образом, мы увеличиваем температурный режим второй конфорки на 1 и в результате получаем первый температурный режим для второй конфорки. Кнопка включения таймера была нажата, о чём говорит переменная buttonTimer, значение которой равно true. Таким образом, после окончания времени на таймере, система выдаёт звуковой сигнал об окончании времени и выключает электрическую плиту.

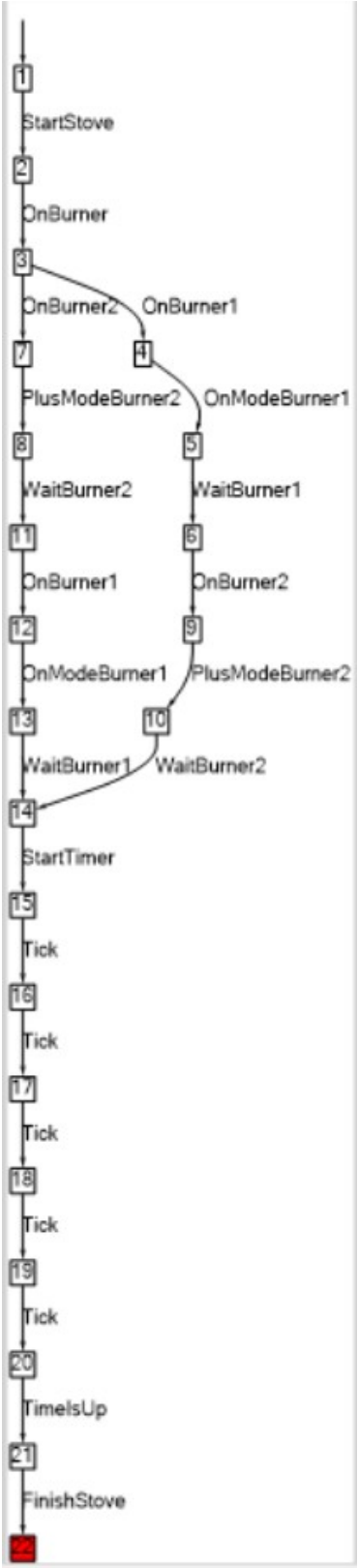

Рисунок 10 - Визуальное представление работы системы

В результате прохода значения переменных данной модели представлены на рисунке 11.

```
The environment is:
Variables:
working=false;
buttonPlus=false;
blocking=false;
buttonPlus2=false;
mode1=0;
time=0;
buttonMinus2=false;
someObjectOnStove=false;
modeBurner1=0;
modeBurner2=0;
buttonBlock=false;
buttonMinus=false;
buttonTimer=false;
buttonOn=false;
mode2=0;
```

Рисунок 11 - значения переменных в PAT

В результате проделанной работы мы получили описание модели электрической плиты в средстве проверки моделей PAT. Данная модель будет использована для верификации в среде PAT.

4.2 Верификация модели

Верификация в среде PAT происходит с помощью таких выражений как утверждения. Утверждение – это запрос о поведении системы. Симулятор PAT и верификаторы проверяют утверждение во время выполнения. Утверждение задаётся с помощью зарезервированного ключевого слова assert. PAT поддерживает полный набор линейной темпоральной логики (LTL), а также классические отношения уточнения/эквивалентности. В PAT мы можем проверять следующие свойства.

- Тупиковая свобода. Тупиковая свобода [34] – это свойство, утверждающее, что система никогда не может оказаться в ситуации, когда прогресс невозможен. Это свойство относится к системам, которые должны работать бесконечно. Набор должным образом определённых

конечных состояний должен быть свободным от тупиков. Следующее утверждение спрашивает, свободен ли тупик P() или нет.

#assert P() deadlockfree;

где и assert, и deadlockfree являются зарезервированными ключевыми словами. Средство проверки модели PAT выполняет алгоритм Depth-First-Search (поиск в глубину) или Breath-First-Search (поиск в ширину), чтобы повторно исследовать непосещённые состояния, пока не будет найдено состояние взаимоблокировки (то есть состояние без дальнейшего перемещения, за исключением успешно завершённого состояния) или пока не будут посещены все состояния.

- Свобода от дивергенции [25]. Дивергенцией называется бесконечная последовательность τ-действий (другими словами зацикливание, в этом случае система не выполняет никаких внешних действий и не останавливается). Следующее утверждение спрашивает, является ли P() свободным от дивергенции или нет.

#assert P() divergencefree;

где и assert, и divergencefree являются зарезервированными ключевыми словами. Для данного процесса, внутренние переходы могут выполняться вечно, не вызывая каких-либо полезных событий, например, P = (a -> P) \ {a}. Дивергентная система обычно нежелательна.

- Детерминированность [34]. Под жёсткой детерминированностью процессов понимается однозначная предопределённость, то есть у каждого следствия должна

быть строго определённая причина. Следующее утверждение спрашивает, является ли P() детерминированным или нет.

#assert P() deterministic;

где и assert, и deterministic являются зарезервированными ключевыми словами. Для данного процесса, если он детерминированный, то для любого состояния нет двух исходящих переходов, ведущих в разные состояния, но с одинаковыми событиями.

- Не завершающийся процесс. Следующее утверждение спрашивает, является ли P() не завершающимся или нет.

#assert P() nonterminating;

где и assert, и nonterminating являются зарезервированными ключевыми словами. Средство проверки модели PAT выполняет алгоритм Depth-First-Search (поиск в глубину) или Breath-First-Search (поиск в ширину), чтобы повторно исследовать непосещенные состояния, пока не будет найдено состояние завершения (то есть состояние без дальнейшего перемещения, включая успешно завершенное состояние) или все состояния были посещены.

- Достижимость. Достижимость процессом какого-либо состояния. Следующее утверждение спрашивает, может ли процесс P() достичь состояния, при котором выполняется некоторое заданное условие.

#assert P() reaches condition;

где и assert, и reaches являются зарезервированными ключевыми словами, а condition – это некоторое определённое свойство. Чтобы определить является ли утверждение истинным или нет, средство проверки модели

PAT выполняет алгоритм поиска в глубину, чтобы повторно исследовать непосещённые состояния, пока не будет найдено состояние, при котором условие является истинным, или пока не будут посещены все состояния.

В утверждениях, как уже упоминалось ранее, можно использовать формулы линейной темпоральной логики (LTL) [36]. Например, следующее утверждения спрашивает, удовлетворяет ли процесс P() формуле LTL.

#assert P() |= F;

где F – это формула LTL, синтаксис которой определяется как следующие правила,

F = e | prop | []F | <> F | X F | $F_1 U F_2$ | $F_1 R F_2$

где

- e – это событие.
- prop – это преопределённое предложение.
- [] читается как «Globally» (также может быть записано как «G» в PAT). Означает, что свойство F должно быть истинно во всех будущих состояниях.
- <> читается как «Future» (также может быть записано как «F» в PAT). Означает, что свойство F должно стать истинным хотя бы в одном состоянии в будущем.
- X читается как «Next». Означает, что свойство F должно быть истинным в состоянии, непосредственно следующим за данным.
- U читается как «Until». Означает, что свойство F_2 должно выполниться в некотором состоянии в будущем (возможно, в текущем), свойство F_1 обязано выполняться во всех состояниях до обозначенного (не включительно).

- R читается как «Release» (также может быть записано как «V» в PAT). Означает, что свойство F_1 освобождает свойство F_2, если F_2 истинно, пока не наступит момент, когда F_1 первый раз станет истинно (или всегда, если такого момента не наступит). Иначе, F_1 должно хотя бы раз стать истинным, пока F_2 не стало истинным в первый раз.

Неформально утверждение верно тогда и только тогда, когда каждое выполнение системы удовлетворяет формуле. Учитывая формулу LTL, средство проверки модели PAT сначала вызывает процедуру для генерации автомата Бюхи [17], который эквивалентен отрицанию формулы. Проще говоря, мы ищем нарушение свойства, так как легче проверять его нарушение, чем выполнение (потому что достаточно найти хотя бы один контрпример). Нарушение свойства описывается автоматом, распознающим неправильное поведение – автоматом Бюхи. Затем автомат Бюхи ищет пересечения с внутренней моделью процесса, чтобы определить, верна ли формула для всех системных выполнений или нет. Если в автомате Бюхи невозможно выполнить ни один переход, то происходит откат: в данной ветви автомата не удалось найти контрпример.

Формально, детерминированный автомат Бюхи [17] это кортеж A=(Q,Σ,δ,q0,F), который состоит из следующих компонентов:

- Q – конечное множество (элементы Q называются состояниями A);
- Σ – конечное множество, называемое алфавитом A;
- δ: Q × Σ → Q – функция, называемая функцией переходов от A;

- q0 является элементом множества Q, называемым начальным состоянием A;
- F⊆Q – условие приемки (автомат A принимает именно те прогоны, в которых хотя бы одно из бесконечно часто встречающихся состояний находится в F).

В недетерминированном автомате Бюхи функция перехода δ заменяется переходным отношением Δ, которое возвращает набор состояний, а одиночное состояние q0 заменяется набором I начальных состояний.

Для симуляции модели электрической плиты переменные в PAT были объявлены так, как представлено на рисунке 12. Из объявления следует, что изначально электрическая плита выключена. После нажатия кнопки включения она была включена. После включения нагревателей на конфорках и регулировки температур устанавливается 2 температурный режим на первой конфорке, а на второй конфорке 1. Так как нажата кнопка запуска таймера, запускается таймер и происходит выключение системы.

```
#define M 6;
#define timer 5;

var time = 0;
var buttonTimer = true;
var buttonOn = true;
var buttonBlock = false;

var mode1:{0.. M} = 3;
var mode2:{0.. M} = 0;
var buttonPlus = false;
var buttonPlus2 = true;
var buttonMinus = true;
var buttonMinus2 = false;
var someObjectOnStove = false;

var modeBurner1:{0.. M} = 0;
var modeBurner2:{0.. M} = 0;

var working = false;
var blocking = false;
```

Рисунок 12 - объявление переменных в PAT

Для проверки мы объявили несколько утверждений.

- #define someone_PressOn (buttonOn==true && working == false) – это утверждение обозначает нажатие кнопки включения при выключенной электрической плите.
- #define someone_PressButton (buttonPlus == true || buttonMinus == true || mode1 > 0 || buttonPlus2 == true || buttonMinus2 == true || mode2 > 0) – это утверждение обозначает нажатие какой-либо кнопки, относящейся к регулировке температурного режима конфорок.
- #define someone_Block ((blocking == true && buttonBlock == false) || (blocking == false && buttonBlock == true)) – это утверждение обозначающее, что дисплей электрической плиты был заблокирован от случайного нажатия.
- #define notBlock (blocking == false && buttonBlock == false) – это утверждение обозначает, что дисплей электрической плиты не заблокирован.

- #define SystemWork (working == true) – это утверждение обозначает, что электрическая плита в данный момент работает.
- #define someone_PressOff (buttonOn==true && working == true) – это утверждение обозначает нажатие кнопки выключения при включённой электрической плите.
- #define someone_OnBurner (modeBurner1 > 0 || modeBurner2 > 0) – это утверждение обозначает включение нагревателя одной из конфорок или обеих одновременно.
- #define someone_OnBurner1 (modeBurner1 > 0) – это свойство утверждение включение нагревателя на первой конфорке.
- #define RangeOfModeBurner1 ((modeBurner1 >= 0) && (modeBurner1 <= M)) – это утверждение обозначает, что температурный режим первой конфорки в данный момент находится в промежутке между 0 и константой M равной 6.

Для данной модели для процесса Stove() мы проверяем заданные в спецификации свойства.

- #assert Stove() reaches someone_PressOn – в данном случае проверяется достижимость свойства «Нажатие кнопки включения электрической плиты». Для заданных переменных мы получаем результат, представленный на рисунке 13, который говорит, что утверждение истинно.

```
Output
********Verification Result********
The Assertion (Stove() reaches someone_PressOn) is VALID.
The following trace leads to a state where the condition is satisfied.
<init>

********Verification Setting********
Admissible Behavior: All
Search Engine: First Witness Trace using Depth First Search
System Abstraction: False

********Verification Statistics********
Visited States:1
Total Transitions:0
Time Used:0,0011659s
Estimated Memory Used:8552,736KB
```

Рисунок 13 - Результат проверки

- #assert Stove() |= G (someone_PressOn -> X (! StartTimer U OnBurner)) – в данном случае проверяется, что во всех будущих состояниях после включения электрической плиты в следующем состоянии не будет запущен таймер пока не будет включён нагреватель на одной из конфорок. Для заданных переменных мы получаем результат, представленный на рисунке 14, который говорит, что утверждение истинно.

```
Output
********Verification Result********
The Assertion (Stove() |= G ( someone_PressOn-> X (! StartTimer U OnBurner))) is VALID.

********Verification Setting********
Admissible Behavior: All
Search Engine: Strongly Connected Component Based Search

********Verification Statistics********
Visited States:27
Total Transitions:26
Time Used:0,0063965s
Estimated Memory Used:41460,112KB
```

Рисунок 14 - Результат проверки

- #assert Stove() |= (StartTimer -> X (!ResetTimer U (someone_PressButton || SomeObjectOnStove))) – в данном случае проверяется, что после запуска таймера в следующем состоянии таймер не будет сброшен пока не будет нажата какая-либо кнопка относящаяся к регулировке

температурного режима конфорок или на индикаторы варочной поверхности не попадёт какой-либо предмет. Для заданных переменных мы получаем результат, представленный на рисунке 15, который говорит, что утверждение истинно.

```
Output
********Verification Result********
The Assertion (Stove() |= ( StartTimer-> X (! ResetTimer U ( someone_PressButton|| SomeObjectOnStove)))) is VALID.

********Verification Setting********
Admissible Behavior: All
Search Engine: Loop Existence Checking - The negation of the LTL formula is a safety property!
System Abstraction: False

********Verification Statistics********
Visited States:0
Total Transitions:0
Time Used:0,0006845s
Estimated Memory Used:8544,424KB
```

Рисунок 15 - Результаты проверки

- #assert Stove() |= G (someone_Block -> F ! someone_Block) – в данном случае проверяется, что во всех будущих состояниях если процесс вошёл в состояние «плита заблокирована», то когда-нибудь в будущем он из этого состояния выйдет и плита будет разблокирована. Для заданных переменных мы получаем результат, представленный на рисунке 16, который говорит, что утверждение истинно.

```
Output
********Verification Result********
The Assertion (Stove() |= G ( someone_Block-> F ! someone_Block)) is VALID.

********Verification Setting********
Admissible Behavior: All
Search Engine: Strongly Connected Component Based Search

********Verification Statistics********
Visited States:25
Total Transitions:25
Time Used:0,0111627s
Estimated Memory Used:41386,56KB
```

Рисунок 16 - Результаты проверки

- #assert Stove() reaches notBlock – в данном случае проверяется, что процесс достигает состояния «плита не

заблокирована». Для заданных переменных мы получаем результат, представленный на рисунке 17, который говорит, что утверждение истинно.

```
Output
********Verification Result********
The Assertion (Stove() reaches notBlock) is VALID.
The following trace leads to a state where the condition is satisfied.
<init>

********Verification Setting********
Admissible Behavior: All
Search Engine: First Witness Trace using Depth First Search
System Abstraction: False

********Verification Statistics********
Visited States:1
Total Transitions:0
Time Used:0,0012861s
Estimated Memory Used:8569,064KB
```

Рисунок 17 - Результаты проверки

- #assert Stove() |= (SystemWork -> (SystemWork U (someone_PressOff || TimeIsUp || SomeObjectOnStove))) – в данном случае проверяется, что система будет работать, пока кто-то не нажмёт кнопку выключения или не будет закончено время на таймере (в случае если таймер запущен), или на индикаторы варочной поверхности не попадёт какой-либо предмет. Для заданных переменных мы получаем результат, представленный на рисунке 18, который говорит, что утверждение истинно.

```
Output
********Verification Result********
The Assertion (Stove() |= ( SystemWork->( SystemWork U ( someone_PressOff|| TimeIsUp|| SomeObjectOnStove)))) is VALID.

********Verification Setting********
Admissible Behavior: All
Search Engine: Loop Existence Checking - The negation of the LTL formula is a safety property!
System Abstraction: False

********Verification Statistics********
Visited States:0
Total Transitions:0
Time Used:0,0010362s
Estimated Memory Used:8540,336KB
```

Рисунок 18 - Результаты проверки

- #assert Stove() |= G F SystemWork – в данном случае проверяется, что свойство «Система работает» будет

истинным бесконечное число раз на всех траекториях системы. Для заданных переменных мы получаем результат, представленный на рисунке 19, который говорит, что утверждение ложно и верификатор предоставляет контрпример при котором данное утверждение не срабатывает.

```
Output
********Verification Result********
The Assertion (Stove() |= G F SystemWork) is NOT valid.
A counterexample is presented as follows.
<init -> StartStove -> OnBurner -> OnBurner2 -> PlusModeBurner2 -> WaitBurner2 -> OnBurner1 -> OnModeBurner1 -> MinusModeBurner1 -> WaitBurner1 ->
StartTimer -> Tick -> Tick -> Tick -> Tick -> Tick -> TimeIsUp -> FinishStove>

********Verification Setting********
Admissible Behavior: All
Search Engine: Strongly Connected Component Based Search

********Verification Statistics********
Visited States:22
Total Transitions:20
Time Used:0,004795s
Estimated Memory Used:41449,464KB
```

Рисунок 19 - Результаты проверки

Автомат Бюхи выглядит следующим образом (рисунок 20).

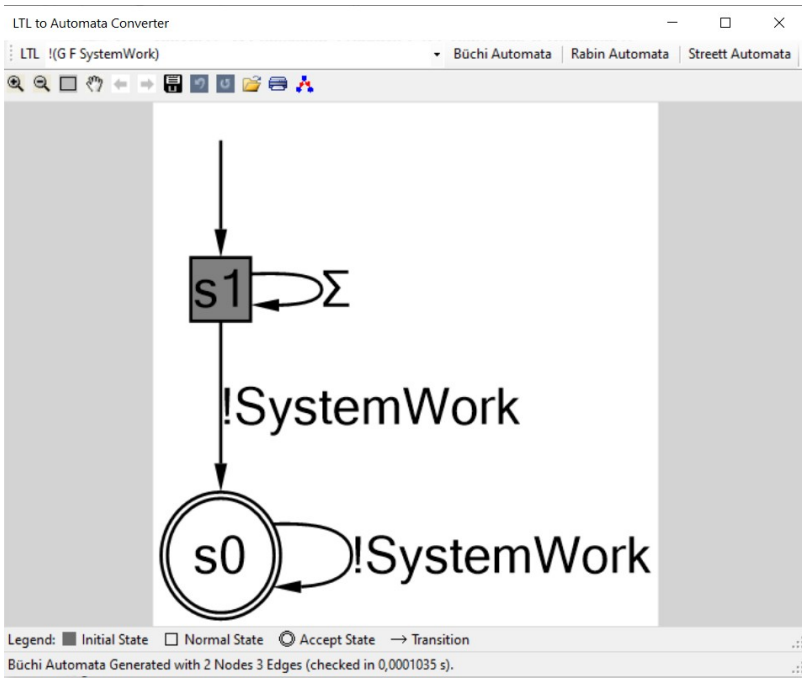

Рисунок 20 - Автомат Бюхи

В PAT можно построить контрпример, на котором свойство не выполняется (то есть путь, ведущий к нарушению

свойства). Для данного свойства контрпример выглядит следующим образом (рисунок 21).

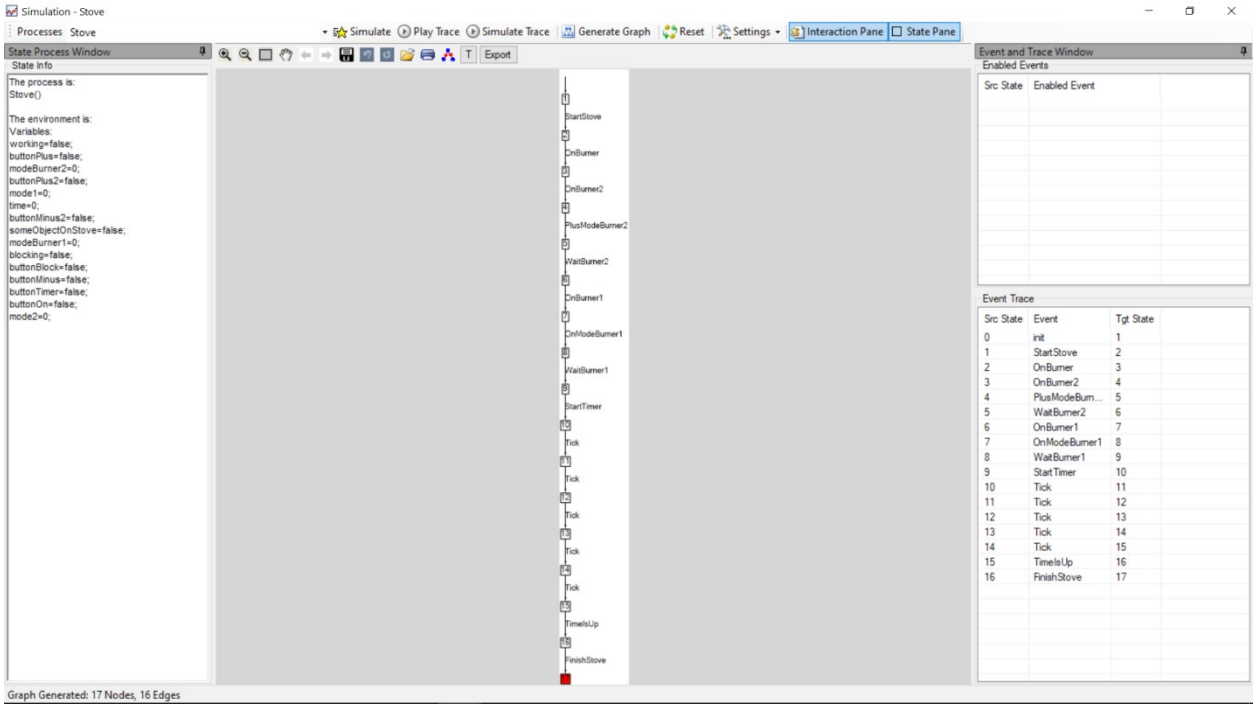

Рисунок 21 - Симуляция контрпримера

Данное свойство было заведомо задано неверно, так как система не может быть постоянно во включенном, работающем состоянии (в какой-то момент времени система перейдёт в состояние «Плита выключена»).

- #assert Stove() |= <> someone_OnBurner1 – в данном случае проверяется, что хотя бы в одном состоянии в будущем будет включен нагреватель для первой конфорки. Для заданных переменных мы получаем результат, представленный на рисунке 22, который говорит, что утверждение истинно.

```
Output
********Verification Result********
The Assertion (Stove() |= <> someone_OnBurner1) is VALID.

********Verification Setting********
Admissible Behavior: All
Search Engine: Loop Existence Checking - The negation of the LTL formula is a safety property!
System Abstraction: False

********Verification Statistics********
Visited States:8
Total Transitions:9
Time Used:0,001518s
Estimated Memory Used:8577,072KB
```

Рисунок 22 - Результаты проверки

- Stove |= G (TimeIsUp -> F FinishStove) – в данном случае проверяется, что во всех будущих состояния по истечении времени на таймере система переходит в состояние «Плита выключена». Для заданных переменных мы получаем результат, представленный на рисунке 23, который говорит, что утверждение истинно.

```
Output
********Verification Result********
The Assertion (Stove() |= G ( TimeIsUp-> F FinishStove)) is VALID.

********Verification Setting********
Admissible Behavior: All
Search Engine: Strongly Connected Component Based Search

********Verification Statistics********
Visited States:24
Total Transitions:24
Time Used:0,0076145s
Estimated Memory Used:41402,936KB
```

Рисунок 23 - Результаты проверки

- Stove |= G (SomeObjectOnStove -> F FinishStove) – в данном случае проверяется, что при попадании какого-либо предмета на индикаторы варочной поверхности система переходит в состояние «Плита выключена». Для заданных переменных мы получаем результат, представленный на рисунке 24, который говорит, что утверждение истинно.

```
Output
********Verification Result********
The Assertion (Stove() |= G ( SomeObjectOnStove-> F FinishStove)) is VALID.

********Verification Setting********
Admissible Behavior: All
Search Engine: Strongly Connected Component Based Search

********Verification Statistics********
Visited States:25
Total Transitions:25
Time Used:0,0058263s
Estimated Memory Used:41429,216KB
```

Рисунок 24 - Результаты проверки

- #assert Stove() |= G(RangeOfModeBurner1) – в данном случае проверяется регулировка температурного режима первой конфорки. Во всех будущих состояниях температурный режим не может быть меньше 0 и больше максимального температурного режима, который равен 6. Для заданных переменных мы получаем результат, представленный на рисунке 25, который говорит, что утверждение истинно.

```
Output
********Verification Result********
The Assertion (Stove() |= G ( RangeOfModeBurner1)) is VALID.

********Verification Setting********
Admissible Behavior: All
Method: Refinement Based Safety Analysis using DFS - The LTL formula is a safety property!
System Abstraction: False

********Verification Statistics********
Visited States:23
Total Transitions:23
Time Used:0,0242086s
Estimated Memory Used:8745,216KB
```

Рисунок 25 - Результаты проверки

Построение автомата Бюхи по LTL-формуле происходит следующим образом. LTL-формула приводится в негативную нормальную форму, в которой отрицание применяется только к пропозициональным переменным. В формуле не

встречаются подформулы вида Fφ и Gφ, так как их нужно заменить на

$$F\varphi = true\, U\, \varphi, \tag{13}$$

$$G\varphi = false\, R\, \varphi. \tag{14}$$

Для приведения LTL-формулы в негативную нормальную форму можно воспользоваться следующими тождествами для темпоральных операторов:

$$(\varphi\, U\, \psi) \equiv (\varphi) R (\psi)\ ; \tag{15}$$

$$(\varphi\, R\, \psi) \equiv (\varphi) U (\psi)\ ; \tag{16}$$

$$(X\varphi) \equiv X(\varphi). \tag{17}$$

Для построения данного автомата были предприняты следующие шаги.

- Исходная формула

$G[0 \le modeBurner1 \le modeBurner1_{max}]$

- Избавление от операторов F и G с помощью формул 13-14. В данном случае, оператор Gφ становится оператором вида false R φ

$false\, R[0 \le modeBurner1 \le modeBurner1_{max}]$

- Отрицание полученной формулы. Этот этап необходим для последующей проверки правильности исходной формулы

$\neg(false\, R\left[0 \le modeBurner1 \le modeBurner1_{max}\right])$

- Приведение полученной формулы к ННФ (негативной нормальной форме). Для этого необходимо воспользоваться формулами 15-17 – отрицание должно применяться к пропозициональным переменным

$$true\, U \neg\left[0 \le modeBurner1 \le modeBurner1_{max}\right]¿.$$

Построенный для данной формулы автомат Бюхи в PAT выглядит следующим образом (рисунок 26).

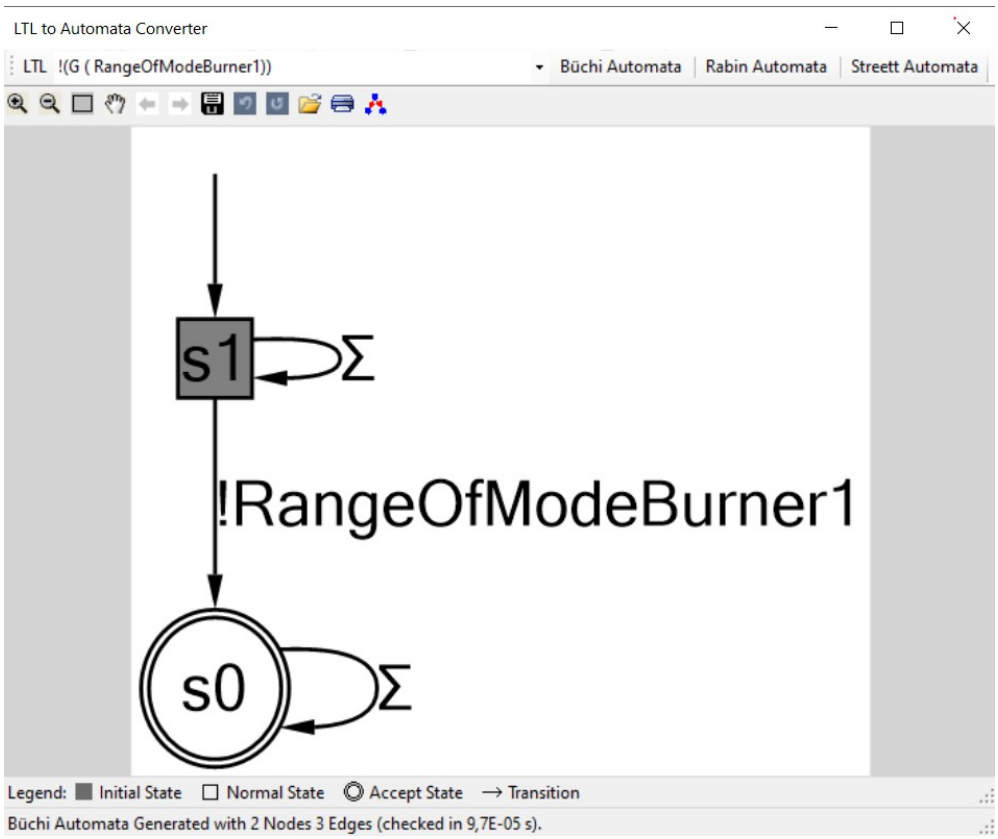

Рисунок 26 - Автомат Бюхи

Объявление описанных свойств и утверждений в верификаторе PAT представлено на рисунке 27. Полный листинг описанной модели приведен в приложении А.

```
#define someone_PressOn (buttonOn==true && working == false);
#assert Stove() reaches someone_PressOn;
#assert Stove() |= G (someone_PressOn -> X (!StartTimer U OnBurner));

#define someone_PressButton (buttonPlus == true || buttonMinus == true
            || mode1 > 0 || buttonPlus2 == true || buttonMinus2 == true || mode2 > 0);

#assert Stove() |= (StartTimer -> X (!ResetTimer U (someone_PressButton || SomeObjectOnStove)));

#define someone_Block ((blocking == true && buttonBlock == false)
                           || (blocking == false && buttonBlock == true));
#assert Stove() |= G (someone_Block -> F !someone_Block);

#define notBlock (blocking == false && buttonBlock == false);
#assert Stove() reaches notBlock;

#define SystemWork (working == true);
#define someone_PressOff (buttonOn==true && working == true);
#assert Stove() |= (SystemWork -> (SystemWork U (someone_PressOff || TimeIsUp || SomeObjectOnStove)));
#assert Stove() |= G F SystemWork;

#define someone_OnBurner (modeBurner1 > 0 || modeBurner2 > 0);
#assert Stove() |= G ((StartStove && X FinishStove)->(someone_OnBurner && (someone_OnBurner U FinishStove)));
#define someone_OnBurner1 (modeBurner1 > 0);
#assert Stove() |= <> someone_OnBurner1;

#define RangeOfModeBurner1 ((modeBurner1 >= 0) && (modeBurner1 <= M));
#assert Stove() |= G(RangeOfModeBurner1);

#assert Stove |= G (TimeIsUp -> F FinishStove);

#assert Stove |= G (SomeObjectOnStove -> F FinishStove);
```

Рисунок 27 - объявление свойств и утверждений в PAT

ЗАКЛЮЧЕНИЕ

Современная жизнь неотрывно связана с различными техническими и информационными системами, которые обслуживают самые разнообразные сферы деятельности человека. В связи с этим всё большую важность приобретают проблемы валидации таких систем с целью устранения ошибок. Устранение ошибок на этапе проектирования системы позволяет зачастую не только сэкономить финансовые ресурсы, но и не подвергать риску здоровье и жизни людей, поэтому особое значение имеют методы формальной верификации, одним из которых является метод проверки модели (Model Checking).

В рамках выпускной квалификационной работы был разработан тестовый пример системы для верификатора PAT: были изучены и описаны теоретические сведения о видах верификации, основы линейной темпоральной логики; особое внимание было уделено методу Model Checking; был произведен сравнительный анализ существующих инструментов верификации, использующих этот метод, на основе которого был выбран наиболее подходящий инструмент проверки модели - PAT; разработана модель системы «Электрическая плита»; сформулированы требования к системе «Электрическая плита»; описана модель и требования к ней с помощью выбранного инструмента проверки моделей PAT; а также проведена верификация построенной модели системы «Электрическая плита» в данном инструменте.

Результаты этой работы могут использоваться в качестве

учебно-методического обеспечения на дисциплинах «Теория автоматов и формальных языков», «Верификация программного обеспечения». Методические указания для проведения занятия на тему «Верификация автоматных программ с помощью верификатора PAT» представлены в ПРИЛОЖЕНИИ Б.

Результаты работы докладывались и обсуждались на XVI международной научно-практической конференции «Проблемы управления в социально-экономических и технических системах» в секции «Управление в технических системах». Работа была отмечена дипломом 3 степени [38].

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Вельдер С. Э., Лукин М. А., Шалыто А. А, Яминов Б. Р ВЕРИФИКАЦИЯ АВТОМАТНЫХ ПРОГРАММ Учебное пособие – Санкт-Петербург, 2011 – 242с.

2. Шульга Т.Э. Теория автоматов и формальных языков [Текст]: учеб. пособие для студ. вузов, обуч. по спец. «Программная инженерия» и «Прикладная информатика» – Саратов: Сарат. гос. техн. ун-т, 2015 – 83 с.

3. Вельдер С.Э., Шалыто А.А. - Верификация простых автоматных программ на основе метода model checking // опубликовано в материалах XV Международной научно-методической конференции «Высокие интеллектуальные технологии и инновации в образовании и науке» – СПбГ ПУ. 2008, с. 285–288.

4. Карпов Ю.Г. Model Checking. Верификация параллельных и распределенных программных систем – Санкт-Петербург, 2010 – 560 с.

5. Lukin M., Velder S., Shalyto A., Yaminov B. Verification of automata-based programs – St.Petersburg, 2011 – 102 с.

6. Камкин А. С. Введение в формальные методы верификации программ: учебное пособие – Москва, 2018. – 272 с

7. Степанов О. Г. МЕТОД АВТОМАТИЧЕСКОЙ ДИНАМИЧЕСКОЙ ВЕРИФИКАЦИИ АВТОМАТНЫХ ПРОГРАММ // опубликовано в материалах Научно-технический вестник СПбГУИТМО. Вып. 53. Автоматное программирование. 2008 – с. 221–229.

8. Поликарпова Н. И., Шалыто А. А. Автоматное программирование. 2008. — 167 с.

9. Кузьмин Е. В., Соколов В. А. Моделирование, спецификация и верификация «автоматных» программ // Программирование. 2008. №1. –Ярославль, 2008 – с. 38-60

10. Шалыто А. А. Парадигма автоматного программирования //Научно-технический вестник СПбГУ ИТМО. Автоматное программирование. Вып. 53., 2008 – с. 3–23.

11. Harel D., Pnueli A. On the development of reactive systems / In «Logic and Models of Concurrent Systems». NATO Advanced Study Institute on Logic and Models for Verification and Specification of Concurrent Systems. Springer Verlag, 1985 – с. 477–498.

12. Карпов Ю. Имитационное моделирование систем. Введение в моделирование с AnyLogic 5. – СПб.: БХВ-Петербург, 2005 – с. 400

13. Синицын С.В., Налютин Н.Ю. Верификация программного обеспечения: Курс лекций. – М.: МИФИ (ГУ), 2006 – 158 с.

14. Бейзер Б, Тестирование черного ящика. Технологии функционального тестирования программного обеспечения и систем. – СПб.:Питер, 2004 – 320 с.

15. Pnueli A, The temporal logic of programs // 18thIEEE Symposium on Foundations of Computer Science. 1977 – с. 46–57

16. Кларк Э. М., Грамберг О., Пелед Д. Верификация моделей программ. Model Checking – МЦНМО, 2002 — 416 с.

17. Миронов А. М. Верификация программ методом Model Checking, – Москва, 2012 — 84 с

18. Миронов А. М., Жуков Д. Ю., Математическая модель и методы верификации программных систем, Интеллектуальные системы, том. 9 –Москва, 2005 – с. 209-252

19. Frappier M., Fraikin B., Chossart R., Chane-Yack-Fa R., Ouenzar M. Comparison of Model Checking Tools for Information Systems Université de Sherbrooke – Québec, Canada, 2010 – 85 с.

20. «List of model checking tools» [Электронный ресурс]. – Режим доступа https://en.wikipedia.org/wiki/List_of_model_checking_tools

21. «Spin» [Электронный ресурс]. – Режим доступа http://spinroot.com/

22. Holzmann G. J. The Spin Model Checker: Primer and Reference Manual. – Addison-Wesley, 2003 – с. 608

23. «NuSMV» [Электронный ресурс]. – Режим доступа http://nusmv.fbk.eu/

24. Havelund K., Majumdar R. Model Checking Software // 15th International SPIN Workshop – Los Angeles, CA, USA, August 10-12, 2008 – с. 343

25. «PAT» [Электронный ресурс]. – Режим доступа https://pat.comp.nus.edu.sg/

26. J. Sun, Y. Liu, J. S. Dong, Y. Liu, L. Shi, É. André. Modeling and Verifying Hierarchical Real-Time Systems Using Stateful Timed CSP. The ACM Transactions on Software Engineering and Methodology (TOSEM) 2013 – с. 29

27. «PRISM» [Электронный ресурс]. – Режим доступа https://www.prismmodelchecker.org/

28. «BLAST» [Электронный ресурс]. – Режим доступа https://en.wikipedia.org/wiki/BLAST_model_checker

29. «Java PathFinder» [Электронный ресурс]. – Режим доступа http://javapathfinder.sourceforge.net/

30. «CADP» [Электронный ресурс]. – Режим доступа https://cadp.inria.fr/

31. «UPPAAL» [Электронный ресурс]. – Режим доступа http://www.uppaal.org/

32. M. Pereira Júnior, G. Vaz Alves A Study Towards the Application of UPPAAL Model Checker // Conference: WEIT 2015 - 3rd Workshop-School on Theoretical Computer Science, At Porto Alegre, RS – Brazil, 2015 – с. 9

33. «KRONOS» [Электронный ресурс]. – Режим доступа http://www-verimag.imag.fr/DIST-TOOLS/TEMPO/kronos/

34. Колчин А. В. РАЗРАБОТКА ИНСТРУМЕНТАЛЬНЫХ СРЕДСТВ ДЛЯ ПРОВЕРКИ ФОРМАЛЬНЫХ МОДЕЛЕЙ // // опубликовано в материалах ISSN 1727-4907. Проблеми програмування. № 2-3. Спеціальний випуск – Институт кибернетики им. В.М. Глушкова НАН Украины, 2008 – 622-626 с.

35. Бурдонов И. Б., Косачев А. С., Кулямин В. В. Безопасность, верификация и теория конформности // опубликовано в материалах второй международной научной конференции по проблемам безопасности и противодействия терроризму – МГУ 2006", М., МЦНМО, 2007 – с 135-158.

36. Хворостухина Е.В. Математическая логика [Текст]: учеб. пособие для студентов бакалавриата по направлениям подготовки 09.03.01 «Информатика и вычислительная

техника», 09.03.04 «Программная инженерия» – Саратов, 2018.

37. Хоар Ч. Взаимодействующие последовательные процессы: Пер. с англ.– М.: Мир, 1989 – с. 264

38. Носырева М.А. Разработка тестового примера автоматной программы средствами PAT // Проблемы управления в социально-экономических и технических системах. Сборник научных статей – Саратов: Наука, 2020 (в печати).

ПРИЛОЖЕНИЕ А

Листинг программы, описывающей модель «Электрическая плита»

```
#define M 6; // количество температурных режимов
#define timer 5; // время, заданное на таймере
var time = 0; // время, прошедшее с момента запуска таймера
var buttonTimer = true; // нажатие кнопки таймера
var buttonOn = true; // нажатие кнопки включения/выключения
var buttonBlock = false; // нажатие кнопки блокировки
var mode1:{0.. M} = 2; // выбор температурного режима для 1 конфорки
var mode2:{0.. M} = 0; // выбор температурного режима для 2 конфорки
var buttonPlus = false; // нажатие кнопки увеличения на 1 температурного режима 1 конфорки
var buttonPlus2 = true; // нажатие кнопки увеличения на 1 температурного режима 2 конфорки
var buttonMinus = false; // нажатие кнопки уменьшения на 1 температурного режима 1 конфорки
var buttonMinus2 = false; // нажатие кнопки уменьшения на 1 температурного режима 2 конфорки
var someObjectOnStove = false; // индикация попадания постороннего предмета на плиту
var modeBurner1:{0.. M} = 0; // текущий температурный режим 1 конфорки
var modeBurner2:{0.. M} = 0; // текущий температурный режим 2 конфорки
```

var working = false; // рабочее состояние конфорки
var blocking = false; // состояние блокировки плиты
Stove() = [working == false && buttonOn == true]StartStove{working = true; buttonOn = false} -> Stove() // включение электрической плиты
[] [working == true && ((blocking == false && buttonBlock == true) || (blocking == true && buttonBlock == false))]BlockStove{blocking = true; buttonBlock = false} -> Stove() //блокировка электрической плиты
[] [working == true && blocking == true]BlockStove{blocking = true} -> Stove()
[] [working == true && blocking == true && buttonBlock == true]UnblockStove{blocking = false; buttonBlock = false} -> Stove() // разблокировка электрической плиты
[][working == true && blocking != true && buttonBlock == false && ((mode1 > 0 || buttonPlus == true) || (mode2 > 0 || buttonPlus2 == true))]OnBurner{working=true}->Burner() // включение нагревателя на одной из конфорок
[] [working == true && someObjectOnStove == true]SomeObjectOnStove{buttonOn = false;working = false;blocking=false;buttonBlock=false;buttonPlus = false;buttonMinus = false;
buttonPlus2 = false;buttonMinus2 = false;modeBurner1 = 0;modeBurner2 = 0;someObjectOnStove = false;} -> FinishStove -> Stove() // попадание постороннего предмета на индикаторы электрической плиты
[] [working == true && buttonOn == true] FinishStove{buttonOn = false;working =

false;blocking=false;buttonBlock=false;buttonPlus = false;buttonMinus = false; buttonPlus2 = false;buttonMinus2 = false;modeBurner1 = 0;modeBurner2 = 0;someObjectOnStove = false;} -> Stove() //выключение электрической плиты
;
Burner() = [(mode1 > 0 || buttonPlus == true)]OnBurner1{working=true}->Burner1() // включение нагревателя на 1 конфорке
[][(mode2 > 0 || buttonPlus2 == true)]OnBurner2{working=true}->Burner2() // включение нагревателя на 2 конфорке
[][modeBurner1 == 0 && buttonPlus == false && buttonMinus == false && mode1 == 0 && modeBurner2 == 0 && buttonPlus2 == false && buttonMinus2 == false && mode2 == 0]OffBurner{working = true;} -> Stove()
[] [((modeBurner1 > 0 || modeBurner2 > 0) && buttonPlus == false && buttonMinus == false && buttonPlus2 == false && buttonMinus2 == false && mode1 == 0 && mode2 == 0 && buttonTimer == true)]StartTimer{time = 0;} -> Timer(); // запуск таймера
Burner1() = [mode1 > 0 && buttonPlus == false]OnModeBurner1 {modeBurner1 = mode1; mode1 = 0} -> Burner1()
[] [modeBurner1!=M && buttonPlus == true]PlusModeBurner1{modeBurner1++; buttonPlus=false} -> Burner1() // нажатие кнопки "+" на дисплее для 1 конфорки
[] [modeBurner1!=0 && buttonMinus == true]MinusModeBurner1{modeBurner1--; buttonMinus=false} -> Burner1() // нажатие кнопки "-" на дисплее для 1 конфорки

[] [modeBurner1 > 0 && buttonPlus == false && buttonMinus == false && mode1 == 0]WaitBurner1{working = true} -> Burner()

[] [modeBurner1 == 0 && buttonPlus == false && buttonMinus == false && mode1 == 0]OffBurner1{working = true} -> Burner(); // выключение 1 конфорки

Burner2() = [mode2 > 0 && buttonPlus2 == false]OnModeBurner2{modeBurner2 = mode2; mode2 = 0} -> Burner2()

[] [modeBurner2!=M && buttonPlus2 == true]PlusModeBurner2{modeBurner2++; buttonPlus2=false} -> Burner2() // нажатие кнопки "+" на дисплее для 2 конфорки

[] [modeBurner2!=0 && buttonMinus2 == true]MinusModeBurner2{modeBurner2--; buttonMinus2=false} -> Burner2() // нажатие кнопки "-" на дисплее для 2 конфорки

[] [modeBurner2 > 0 && buttonPlus2 == false && buttonMinus2 == false && mode2 == 0]WaitBurner2{working = true} -> Burner()

[] [modeBurner2 == 0 && buttonPlus2 == false && buttonMinus2 == false && mode2 == 0]OffBurner2{working = true} -> Burner(); // выключение 2 конфорки

Timer() = [(time == timer)] // истёк таймерTimeIsUp{time = 0;working = false;blocking=false;buttonTimer = false;buttonOn = false;buttonBlock=false;buttonPlus = false;buttonMinus = false; buttonPlus2 = false;buttonMinus2 = false;modeBurner1 = 0;modeBurner2 = 0;someObjectOnStove = false;} -> FinishStove -> Stove()

```
            [][(buttonPlus == false && buttonMinus == false &&
buttonPlus2 == false && buttonMinus2 == false && time !=
timer)]Tick{if (time < timer) time++;} -> Timer()
            [][buttonPlus == true || buttonMinus == true ||
buttonPlus2 == true || buttonMinus2 == true || buttonTimer ==
false]ResetTimer{time = 0;} -> Stove() //сброс таймера
            [][someObjectOnStove == true]
SomeObjectOnStove{buttonOn = false;working =
false;blocking=false;buttonBlock=false;buttonPlus =
false;buttonMinus = false; buttonPlus2 = false;buttonMinus2 =
false;modeBurner1 = 0;modeBurner2 = 0;someObjectOnStove =
false;} -> FinishStove -> Stove();
 //попадание постороннего предмета на индикаторы
#define someone_PressOn (buttonOn==true && working ==
false);
#assert Stove() reaches someone_PressOn;
#assert Stove() |= G (someone_PressOn -> X (!StartTimer U
OnBurner));
#define someone_PressButton (buttonPlus == true ||
buttonMinus == true || mode1 > 0 || buttonPlus2 == true ||
buttonMinus2 == true || mode2 > 0);
#assert Stove() |= (StartTimer -> X (!ResetTimer U
(someone_PressButton || SomeObjectOnStove)));
#define someone_Block ((blocking == true && buttonBlock ==
false) || (blocking == false && buttonBlock == true));
#assert Stove() |= G (someone_Block -> F !someone_Block);
#define notBlock (blocking == false && buttonBlock == false);
#assert Stove() reaches notBlock;
#define SystemWork (working == true);
```

```
#define someone_PressOff (buttonOn==true && working ==
true);
#assert Stove() |= (SystemWork -> (SystemWork U
(someone_PressOff || TimeIsUp || SomeObjectOnStove)));
#assert Stove() |= G F SystemWork;
#define someone_OnBurner (modeBurner1 > 0 || modeBurner2
> 0);
#assert Stove() |= G ((StartStove && X FinishStove)-
>(someone_OnBurner && (someone_OnBurner U FinishStove)));
#define someone_OnBurner1 (modeBurner1 > 0);
#assert Stove() |= <> someone_OnBurner1;
#define RangeOfModeBurner1 ((modeBurner1 >= 0) &&
(modeBurner1 <= M));
#assert Stove() |= G(RangeOfModeBurner1);
#assert Stove |= G (TimeIsUp -> F FinishStove);
#define Mode0AllBurner (modeBurner1 == 0 && modeBurner2
== 0);
#assert Stove |= G (SomeObjectOnStove -> F Mode0AllBurner);
```

ПРИЛОЖЕНИЕ Б

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ ПО ТЕМЕ: «ВЕРИФИКАЦИЯ АВТОМАТНЫХ ПРОГРАММ В ВЕРИФИКАТОРЕ PAT»

В качестве верифицируемой системы рассмотрим систему «Электрическая плита».

Система «Электрическая плита» представляет собой варочную поверхность, которая состоит из двух конфорок и дисплея. Для каждой конфорки в зависимости от температуры нагрева устанавливается температурный режим. Количество температурных режимов в данной модели изменяется от нуля до шести. Регулировать температурные режимы каждой конфорки можно с помощью кнопок «+» (увеличить температурный режим на единицу) и «-» (уменьшить температурный режим на единицу). При попытке увеличить (уменьшить) максимальное (минимальное) значение температурного режима ничего не происходит. Помимо этого, пользователь может задать таймер работы нагрева конфорок. Если время на таймере подошло к концу, то система выключается. Также предусмотрена возможность блокировки и разблокировки дисплея плиты от случайных нажатий. В случае попадания каких-либо предметов на индикаторы варочной поверхности, система выдаёт звуковой сигнал и выключается.

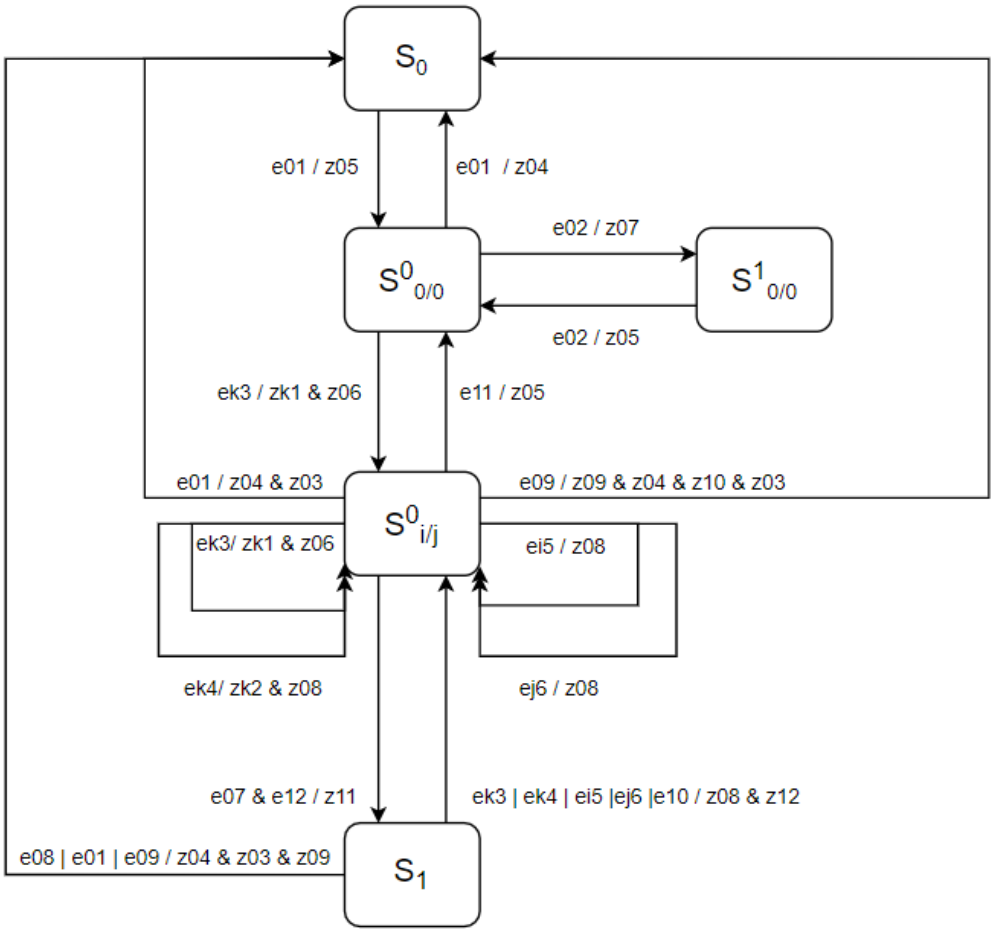

Рисунок Б.1 - Граф переходов автомата "Электрическая плита"

На рисунке Б.1 изображён граф переходов автомата «Электрическая плита» со следующими входными воздействиями (где k=1,2, i,j=0,..,6):

- e01 - нажата кнопка включить/выключить;
- e02 - нажата кнопка блокировки плиты;
- ek3 - нажата кнопка “+” на k-конфорке;
- ek4 - нажата кнопка “-” на k-конфорке;
- ei5 – выбран i-режим для 1 конфорки;
- ej6 – выбран j-режим для 2 конфорки;
- e07 – запуск таймера;
- e08 – таймер истёк;
- e09 – попадание предмета на индикаторы;
- e10 – сброс таймера;

- e11 – обе конфорки находятся в начальном состоянии;
- e12 – нажата кнопка включения таймера.

Выходными воздействиями являются:

- zk1 - увеличить температуру нагревателя k-конфорки;
- zk2 - уменьшить температуру нагревателя k-конфорки;
- z03 - сброс в начальное состояние всех конфорок;
- z04 - индикация выключения на дисплее электрической плиты;
- z05 - индикация готовности на дисплее электрической плиты;
- z06 - индикация нагрева на дисплее электрической плиты;
- z07 - индикация блокировки на дисплее электрической плиты;
- z08 - индикация выполнения на дисплее электрической плиты;
- z09 – звуковой сигнал;
- z10 – индикация неисправности на дисплее электрической плиты;
- z11 – запуск таймера;
- z12 – сброс таймера.

Состояния автомата:

- S_0 – плита выключена;
- $S^0_{0/0}$ – плита готова к работе;
- $S^1_{0/0}$ – плита заблокирована;
- $S^0_{i/j}$ – нагрев конфорок;

- S_1 – ожидание во время нагрева.

Основные требования к модели системы «Электрическая плита» были сформулированы следующим образом и представлены в виде формул линейной темпоральной логики LTL (логика, в которой учитывается временной аспект).

- Достижимость свойства «Нажатие кнопки включения электрической плиты» (1).
- Достижимость свойства «плита не заблокирована» (2).
- Во всех будущих состояниях после включения электрической плиты в следующем состоянии не будет запущен таймер, пока не будет включён нагреватель на одной из конфорок (3).
- После запуска таймера в следующем состоянии таймер не будет сброшен, пока не будет нажата какая-либо кнопка, относящаяся к регулировке температурного режима конфорок, или пока какой-либо предмет не попадёт на индикаторы варочной поверхности (4).
- Во всех будущих состояниях если процесс вошёл в состояние «плита заблокирована», то когда-нибудь в будущем он из него выйдет, и плита будет разблокирована (5).
- Система будет работать, пока кто-то не нажмёт кнопку выключения или не будет закончено время на таймере (в случае если таймер запущен), или пока какой-либо предмет не попадёт на индикаторы варочной поверхности (6).
- Свойство «Система работает» будет истинным бесконечное число раз на всех траекториях системы (7).

- Хотя бы в одном состоянии в будущем будет включен нагреватель для первой конфорки (8).
- По истечении таймера плита выключится (9).
- При попадании какого-либо предмета на индикаторы плита выключится (10).
- Регулировка температурного режима 1 конфорки. Температурный режим не может быть меньше 0 и больше максимального температурного режима, который равен 6 (11).

После разработки модели системы и формулировки требований к ней проводится формальная верификация на основе метода проверки модели.

Формальная верификация основывается на математическом моделировании программ и требований к ним. Одним из подходов к формальной верификации является проверка моделей.

Model Checking (проверка моделей) — это метод формальной верификации, который позволяет для заданной модели поведения некоторой системы с конечным числом состояний и логического свойства (требования) проверить, будет ли выполняться это свойство в рассматриваемых нами состояниях данной модели. Реализация идеи этого метода осуществляется с помощью специальных программ – верификаторов. Эти программы в качестве входных данных получают модель, которая описана на языке выбранного верификатора, и свойство, которое необходимо проверить. В качестве выходных данных программа-верификатор формирует либо сообщение о том, что свойство, которое мы указали, на заданной нами модели выполняется, либо

предоставляет контрпример, который показывает, при каких условиях могло возникнуть данное несоответствие. Контрпример представляет собой некий сценарий (набор определённых действий, условий), в котором модель ведёт себя нежелательным образом. Сценарий, который приводит к нарушению проверяемого свойства.

Таким образом, для начала проведения верификации системы, необходимо перевести граф переходов автомата системы «Электрическая плита» на язык программирования, понятный для верификатора. В данном случае, для используемого в работе верификатора PAT таким языком является язык CSP#.

Так как кодирование графа переходов «Электрическая плита» является автоматным подходом к программированию, то каждое состояние автомата – это отдельная процедура (процесс).

- Процесс Stove(). Данный процесс описывает собственно работу системы электрической плиты.
- Процесс Burner(). Данный процесс описывает работу нагревателей на конфорках.
- Процесс Burner1() и Burner2(). Данные процессы описывают работу нагревателя на первой и второй конфорке.
- Процесс Timer(). Данный процесс описывает работу таймера в данной системе.

Процессы в процедурах описаны согласно графу переходов разработанного ранее автомата. Процессы вызывают друг друга, что обеспечивает переходы между состояниями автомата. На рисунке Б.2 стрелкой указаны вызовы процесса Burner().

```
Burner() = [(mode1 > 0 || buttonPlus == true)]OnBurner1{working=true}->Burner1() // включение нагревателя на 1 конфорке
        [][(mode2 > 0 || buttonPlus2 == true)]OnBurner2{working=true}->Burner2() // включение нагревателя на 2 конфорке
        [][modeBurner1 == 0 && buttonPlus == false && buttonMinus == false && mode1 == 0 && modeBurner2 == 0
                && buttonPlus2 == false && buttonMinus2 == false && mode2 == 0]OffBurner{working = true;} -> Stove()
        [] [((modeBurner1 > 0 || modeBurner2 > 0) && buttonPlus == false && buttonMinus == false && buttonPlus2 == false
                && buttonMinus2 == false && mode1 == 0 && mode2 == 0 && buttonTimer == true)]
            StartTimer{time = 0;} -> Timer(); // запуск таймера
```

Рисунок Б.2 - Процесс Burner()

Требования к системе также необходимо перевести на язык CSP#. Они описываются после объявления всех констант, переменных и процессов (рисунок Б.3).

```
#define someone_PressOn (buttonOn==true && working == false);
#assert Stove() reaches someone_PressOn;
#assert Stove() |= G (someone_PressOn -> X (!StartTimer U OnBurner));

#define someone_PressButton (buttonPlus == true || buttonMinus == true
            || mode1 > 0 || buttonPlus2 == true || buttonMinus2 == true || mode2 > 0);

#assert Stove() |= (StartTimer -> X (!ResetTimer U (someone_PressButton || SomeObjectOnStove)));

#define someone_Block ((blocking == true && buttonBlock == false)
                          || (blocking == false && buttonBlock == true));
#assert Stove() |= G (someone_Block -> F !someone_Block);

#define notBlock (blocking == false && buttonBlock == false);
#assert Stove() reaches notBlock;

#define SystemWork (working == true);
#define someone_PressOff (buttonOn==true && working == true);
#assert Stove() |= (SystemWork -> (SystemWork U (someone_PressOff || TimeIsUp || SomeObjectOnStove)));
#assert Stove() |= G F SystemWork;

#define someone_OnBurner (modeBurner1 > 0 || modeBurner2 > 0);
#assert Stove() |= G ((StartStove && X FinishStove)->(someone_OnBurner && (someone_OnBurner U FinishStove)));
#define someone_OnBurner1 (modeBurner1 > 0);
#assert Stove() |= <> someone_OnBurner1;

#define RangeOfModeBurner1 ((modeBurner1 >= 0) && (modeBurner1 <= M));
#assert Stove() |= G(RangeOfModeBurner1);

#assert Stove |= G (TimeIsUp -> F FinishStove);

#assert Stove |= G (SomeObjectOnStove -> F FinishStove);
```

Рисунок Б.3 - Объявление проверяемых свойств

Полный листинг данной программы приведён в приложении А.

Верификация в среде PAT происходит с помощью таких выражений как утверждения. PAT проверяет утверждение во время выполнения. Утверждение задаётся с помощью ключевого слова assert.

Для того чтобы перейти в режим верификации в PAT необходимо на панели инструментов нажать кнопку «Verification» или нажать кнопку F7 на клавиатуре. После открывается окно «Verification» (рисунок Б.4).

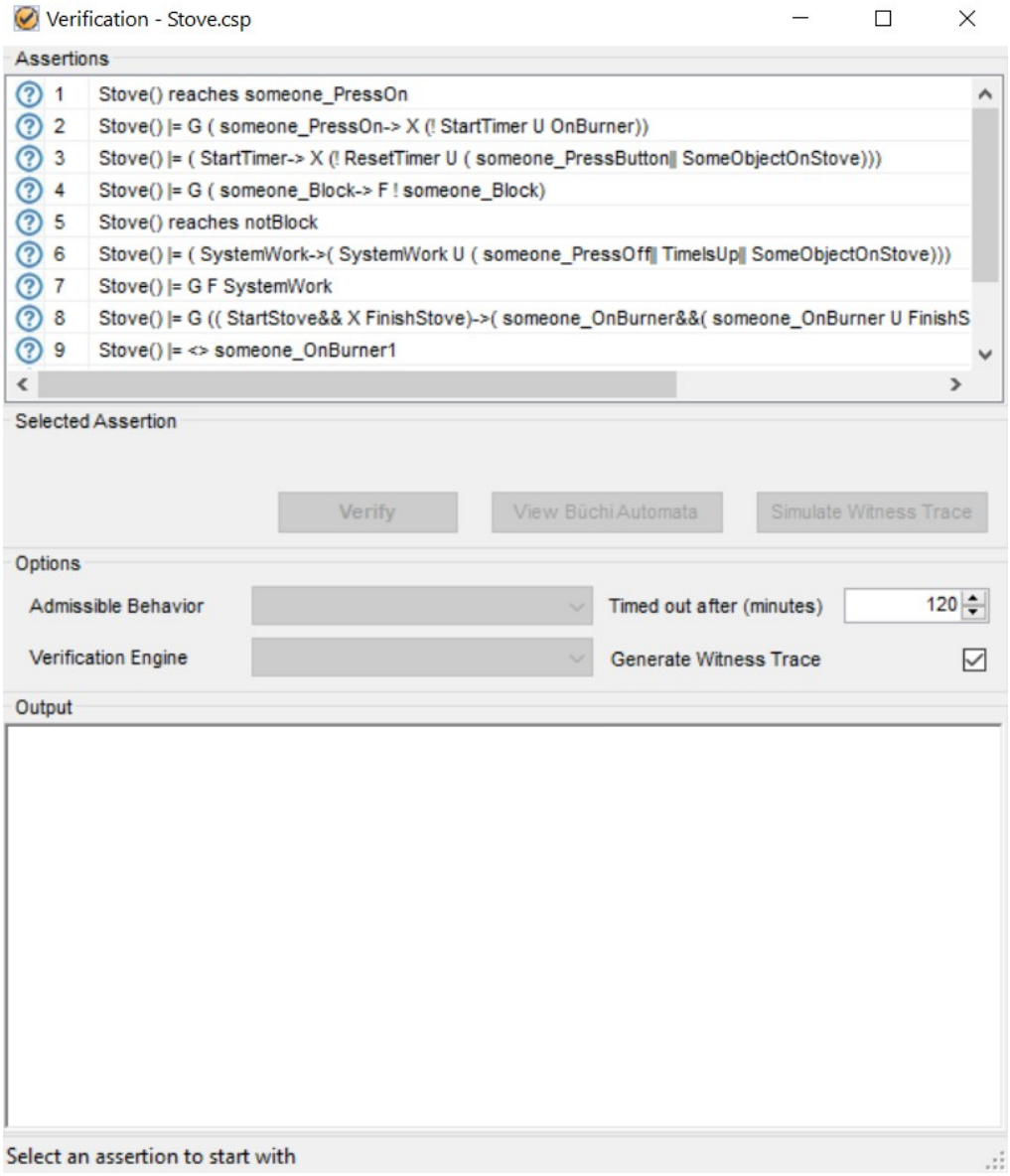

Рисунок Б.4 - Окно "Verification"

В окне «Verification» представлен список утверждений (проверяемых свойств), три кнопки: «Verify» (проверка свойства), «View Buchi Automata» (построить автомат Бюхи), «Simulate Witness Trace» (смоделировать контрпример). Результат верификации будет доступен в нижней части окна.

- #assert Stove() |= (SystemWork -> (SystemWork U (someone_PressOff || TimeIsUp || SomeObjectOnStove))) – в данном случае проверяется, что система будет работать, пока кто-то не нажмёт кнопку выключения или не будет закончено время на таймере (в случае если таймер запущен) или на

индикаторы варочной поверхности не попадёт какой-либо предмет. Для заданных переменных мы получаем результат, представленный на рисунке Б.5, который говорит, что утверждение истинно.

```
Output
********Verification Result********
The Assertion (Stove() |= ( SystemWork->( SystemWork U ( someone_PressOff|| TimeIsUp|| SomeObjectOnStove)))) is VALID.

********Verification Setting********
Admissible Behavior: All
Search Engine: Loop Existence Checking - The negation of the LTL formula is a safety property!
System Abstraction: False

********Verification Statistics********
Visited States:0
Total Transitions:0
Time Used:0,0010362s
Estimated Memory Used:8540,336KB
```

Рисунок Б.5 - Результаты проверки

- #assert Stove() |= G F SystemWork – в данном случае проверяется, что свойство «Система работает» будет истинным бесконечное число раз на всех траекториях системы. Для заданных переменных мы получаем результат, представленный на рисунке Б.6, который говорит, что утверждение ложно и верификатор предоставляет контрпример, при котором данное утверждение не срабатывает.

```
Output
********Verification Result********
The Assertion (Stove() |= G F SystemWork) is NOT valid.
A counterexample is presented as follows.
<init -> StartStove -> OnBurner -> OnBurner2 -> PlusModeBurner2 -> WaitBurner2 -> OnBurner1 -> OnModeBurner1 -> MinusModeBurner1 -> WaitBurner1 ->
StartTimer -> Tick -> Tick -> Tick -> Tick -> Tick -> TimeIsUp -> FinishStove>

********Verification Setting********
Admissible Behavior: All
Search Engine: Strongly Connected Component Based Search

********Verification Statistics********
Visited States:22
Total Transitions:20
Time Used:0,004795s
Estimated Memory Used:41449,464KB
```

Рисунок Б.6 - Результаты проверки

Автомат Бюхи выглядит следующим образом (рисунок Б.7).

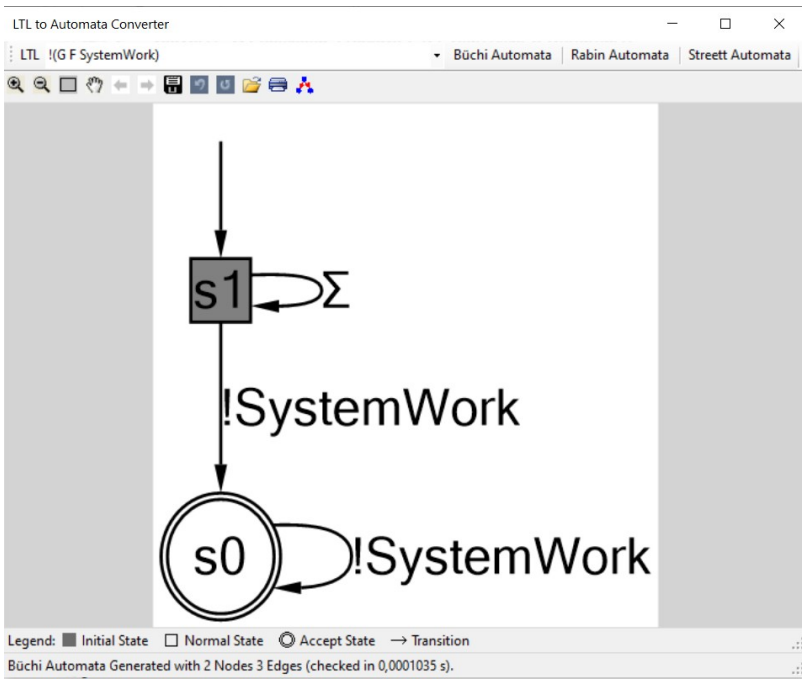

Рисунок Б.7 - Автомат Бюхи

В PAT можно построить контрпример, на котором свойство не выполняется (то есть путь, ведущий к нарушению свойства). Для данного свойства контрпример выглядит следующим образом (рисунок Б.8).

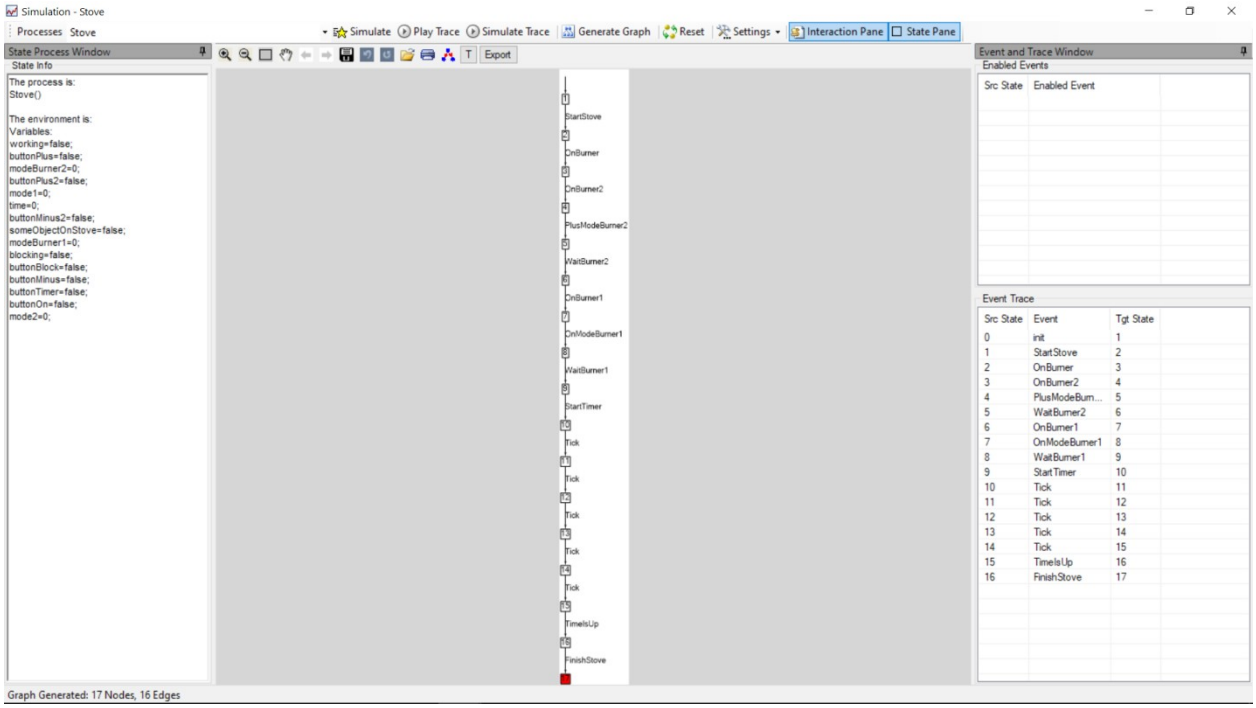

Рисунок Б.8 - Симуляция контрпримера

Данное свойство было заведомо задано неверно, так как система не может быть постоянно во включенном,

работающем состоянии (в какой-то момент времени система перейдёт в состояние «Плита выключена»).