Санкт-Петербургский национальный исследовательский университет
информационных технологий, механики и оптики

На правах рукописи

Егоров Кирилл Викторович

Генерация управляющих автоматов на основе генетического программирования и верификации

Специальность 05.13.11. Математическое и программное обеспечение
вычислительных машин, комплексов и компьютерных сетей

Диссертация на соискание ученой степени
кандидата технических наук

Научный руководитель:
доктор технических наук,
профессор А. А. Шалыто

Санкт-Петербург – 2013 г.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Актуальность проблемы. Людям всегда хотелось уметь проверять правильность программ, но не просто убедиться, что программа правильно работает в конечном числе случаев, а уметь формально доказывать, что ее поведение соответствует спецификации. Метод проверки того, что программа соответствует заданной спецификации (обладает необходимыми свойствами) и называется **верификацией** [7]. На практике, полная верификация системы намного сложнее, чем ее создание. Поэтому верификация не всегда оправдана с точки зрения затрат, и проще исправлять ошибки по мере их обнаружения во время работы системы. Однако ошибки в некоторых системах могут обойтись слишком дорого. Например, в программах управления транспортом, медицинским оборудованием, военной техникой и другими объектами ошибки могут привести к гибели людей или большим финансовым потерям.

Проверке качества программного обеспечения (ПО) ответственных систем, казалось бы, должно было уделяться значительное внимание, но до недавнего времени это было не так, о чем свидетельствуют многочисленные аварии и сбои в системах.

Ошибка аппарата *Therac-25* была, пожалуй, одной из самых дорогих ошибок в современной истории. Аппарат поступил в продажу в конце 1982 года, одиннадцать таких машин были установлены в Северной Америке (пять – в США и шесть – в Канаде). С июня 1985 по январь 1987 года ошибка в программной части привела к гибели троих пациентов из шести, получивших передозировку радиации [17].

За три года работы *Therac-25* было обнаружено два программных дефекта [38]:

- логическая ошибка в обновлении параметров, когда оператор менял состояние аппарата;

- проверка безопасности не срабатывала, когда 8-битный счетчик переполнялся и достигал нуля каждые 256 итераций.

Такие ошибки, зависящие от скорости работы оператора, сложно обнаруживать и воспроизводить. Первый дефект возникал при выполнении двух условий [38]:

- оператору необходимо внести изменения одновременно в параметрах режима и уровня энергии;
- оператору необходимо успеть внести данные изменения в течение восьми секунд.

Возникновение второй ошибки завесило от случайности. Она воспроизводилось только тогда, когда клавиша нажималась в момент достижения нуля счетчиком, поэтому ее было практически невозможно обнаружить.

Другая программная ошибка привела к одной из самых дорогих катастроф, которая произошла 4.06.1996 года во время первого запуска новой ракеты-носителя *Arian-5*, разработанной Европейским космическим агентством. Запуск прошел неудачно – ракета самоуничтожилась через 37 секунд после старта из-за неверной работы бортового ПО [17].

Было установлено, что авария *Arian-5* была вызвана необработанным исключением при преобразовании 64-битного значения с плавающей запятой в 16-битное целое значение со знаком [39].

К сожалению, отечественное освоение космоса также не обошлось без инцидентов, произошедших по вине ПО [53]. «В начале 70-х годов прошлого советскую марсианскую программу преследовали хронические неудачи. Из целой армады межпланетных станций только одной удалось совершить мягкую посадку, однако ни одной фотографии аппарат так и не передал. Оказывается, на аппараты марсианской серии впервые были установлены бортовые цифровые ЭВМ».

С каждым годом доля аппаратной части систем и устройств уменьшается, а программной увеличивается, как, например, это имеет место в автомобилях. Возрастающая сложность ПО приводит к увеличению числа ошибок. Для обеспечения корректности и надежности работы ответственных систем большое значение имеют различные методы тестирования и верификации, позволяющие выявлять ошибки на разных этапах разработки и сопровождения ПО.

На практике тестирование применяется в большинстве случаев. Однако оно позволяет находить ошибки, но не доказывает их отсутствие [26]. Правильность работы программы, используя данный подход, можно подтвердить только при конечном числе входных данных. Однако некоторые ошибки появляются крайне редко и сложно воспроизводятся. Поэтому, для того чтобы исключить возможность их появления, требуется рассмотреть практически все возможные варианты поведения системы.

Указанную проблему решает верификация – метод доказательства того, что программа соответствует спецификации. На практике часто используется метод, который называется верификацией на моделях (*Model Checking*). При этом спецификация задается в виде темпоральных формул, число которых в общем случае не известно. Поэтому соответствие программы спецификации может быть обеспечено, но это не будет доказательством правильности программы. При таком подходе используется логический формализм и математическое описание модели.

Процесс верификации *разработанных программ* делится на четыре части. Построение формальной модели по программе. Построение спецификации – формальная запись свойств, которые требуется проверить. Верификация – проверка выполнения спецификации на модели. Определения соответствия между ошибкой в модели и ошибкой в программе.

Сложность такого подхода заключается как в построении модели по программе, так и в самой верификации. Часто создание программы оказывается проще, чем ее проверка. Для вновь разрабатываемых программ целесообразно сначала построить модель, и только после верификации построить по ней программу.

Для автоматных программ [15], как для существующих, так и для вновь разрабатываемых, может быть применен один и тот же подход к верификации. Автоматная программа представляет собой модель, пригодную для верификации. Поэтому не требуется дополнительно преобразование программы в модель и определения соответствия между ошибкой в модели и в программе [9]. Модель автоматной программы представляет собой управляющий автомат или систему таких автоматов. Для автоматов этого класса характерно наличие событий, условий переходов, задаваемых булевыми формулами, и выходных воздействий, которые могут быть произвольными. В диссертации предполагается, что поведение программ задается одним автоматом, а при описании поведения системой автоматов, они могут быть представлены одним автоматом за счет их «произведения» [8].

Для многих задач автоматы удается строить эвристически, однако существуют задачи, для которых такое построение затруднительно или невозможно [23, 24, 31]. В рамках работ [18] был предложен подход к построению управляющих конечных автоматов (далее автоматов) по тестовым примерам на основе эволюционных алгоритмов. Его особенность состоит в том, что функция приспособленности и операции скрещивания и мутации не зависят от рассматриваемой задачи. При использовании такого подхода при создании автомата выделяются события (*e1, e2, ...*), входные переменные (*x1, x2, ...*) и выходные воздействия (*z1, z2, ...*). В качестве тестов для автомата рассматривались пары последовательностей: первая описывает события и входные переменные, поступающие на вход автомата,

а вторая – выходные воздействия, которые должен вырабатывать автомат при обработке этих событий.

Особь эволюционного алгоритма представляет собой автомат, у которого не фиксированы выходные воздействия на переходах, а указано только их число. Для заполнения «скелета» автомата выходными воздействиями используется метод расстановки пометок [18]. После их расстановки вычисляется функция приспособленности, которая учитывает, насколько хорошо особь проходит тесты. Лучшие особи скрещиваются либо случайным образом, либо с сохранением той части автомата, на которой тесты выполняются. Процесс повторяется до тех пор, пока не появится особь, проходящая все тесты.

Предложенный подход обладает тем недостатком, что при построении неправильного (ошибочного) автомата пользователю приходится снова и снова модифицировать тесты или добавлять новые, пока не будет построен требуемый автомат. Такие действия могут занять много времени, так как построение сложного автомата эволюционными алгоритмами требует рассмотрения большого числа поколений [5, 10, 24]. Поэтому может оказаться, что построение автомата вручную займет меньше времени, чем использование эволюционного алгоритма.

В любом случае, даже построив кажущийся правильным и проходящий все тесты автомат, нельзя гарантировать его поведение при других входных воздействиях. Под правильностью подразумевается корректное поведение построенного автомата при любых возможных вариантах входных событий и входных переменных. Поэтому построение «всеобъемлющих» тестов практически невозможно.

Какие должны быть выходные воздействия при поступлении на вход автомата той последовательности событий, которая не была описана в тестах? Ответ на данный вопрос можно дать двумя способами. Первый – если автомат ведет себя неправильно при определенной

последовательности входных событий и переменных, то добавляется новый тест и автомат строится заново. Второй – использовать верификацию модели при построении автоматов.

Первый вариант плох тем, что приходится вручную искать ошибки в построенном автомате, и тем, что ошибка может быть не обнаружена, а второй – позволяет описывать бесконечное число вариантов поведения автомата. Второй подход используется в настоящей диссертации.

Известны всего несколько работ, в которых генетическое программирование применялось совместно с верификацией [32 – 37]. В них верификация учитывалась только в функции приспособленности, и вклад каждой темпоральной формулы был дискретен: ноль – формула не выполняется, единица – если она выполняется. Выбор начального поколения, мутации и скрещивания были случайны и не учитывали верификацию. В результате рост функции приспособленности следующего поколения был обусловлен только отбором особей, «проходящих» большее число темпоральных формул.

Существуют и другие работы, которые строят автоматы эволюционными алгоритмами [1, 12, 13, 31, 40, 41, 48 – 51], однако большинство из них решают определенную задачу или класс задач, не предлагая решения для общего случая.

Настоящая работа является развитием *совместных* работ автора и Ф.Н. Царева [19, 52, 68], в которых применялись эволюционные алгоритмы для построения автоматов по тестовым примерам и темпоральным формулам. В этих работах было предложено совместно применять тесты и темпоральные формулы. Результат верификации учитывался только в функции приспособленности, аналогично работам [32 – 37]. В работах [19, 52, 68] была предложена технология построения автоматов по тестам и темпоральным формулам. Основные отличия настоящей работы от предыдущих:

1. Вклад каждой темпоральной формулы в функцию приспособленности лежит на отрезке [0; 1], а не дискретен – ноль или единица. Это позволяет лучше учесть приспособленность особи.
2. Темпоральные формулы учитываются не только в функции приспособленности, но и при операциях скрещивания и мутации.
3. Предложен алгоритм генерации автоматов на основе генетического программирования с учетом контрактов, применение которых для автоматных программ было предложено в работе [2].
4. Предложен алгоритм генерации автоматов на основе генетического программирования по сценариям работы и верификации.

Цель работы – генерации автоматов на основе генетического программирования и верификации.

При этом **задачи, решаемые в диссертации**, состоят в следующем:

1. Предложить функцию приспособленности, учитывающую верификацию, проводимую в процессе выполнения алгоритма генетического программирования.
2. Разработать операции скрещивания и мутации, учитывающие верификацию.
3. Разработать алгоритм построения автоматов на основе генетического программирования с учетом контрактов, которые являются разновидностью темпоральных формул.
4. Разработать алгоритм построения автоматов на основе генетического программирования по сценариям работы и верификации.

5. Разработать верификатор, приспособленный для поддержки генерации автоматов на основе генетического программирования.
6. Разработать технологию и инструментальное средство для генерации автоматов с помощью генетического программирования и верификации.

Научная новизна. На защиту выносятся следующие новые научные результаты:

1. Предложена функция приспособленности, учитывающая выполнение формулы на части автомата. В известных работах учитывалось только выполнение или невыполнение каждой темпоральной формулы.
2. Операции скрещивания и мутации отличаются от известных тем, что в ходе их выполнения используется верификация.
3. Алгоритм генерации автоматов с учетом контрактов, который позволяет упростить запись некоторых темпоральных формул и ускорить построение автоматов по сравнению с применением темпоральных формул общего вида.
4. Алгоритм генерации автоматов по темпоральным формулам и сценариям работы, который позволяет ускорить построение автоматов по сравнению с применением тестов.

Методы исследований. В работе используются методы теории автоматов, теории графов, дискретной математики и генетического программирования.

Достоверность разработанных алгоритмов, методов и технологий, полученных в диссертации, подтверждается точной постановкой задач, корректным формулированием параметров и результатов работы генетического алгоритма, численными оценками скорости работы различных подходов, путем экспериментальных исследований.

Практическое значение работы состоит в том, что предложенные методы позволяют разрабатывать программы, которые соответствуют спецификации и не требуют тестирования и верификации как самостоятельных этапов разработки программ. В случае внесения изменений в спецификацию, такие программы легко модифицируются, так как можно заново построить автомат, который будет соответствовать старым и новым требованиям. Предложенная технология разработки автоматных программ и реализованное средство для их построения позволяют строить «правильные» программы для ответственных систем, или когда построение автомата вручную сложнее автоматического.

Внедрение результатов работы. Результаты диссертации были получены при выполнении научно-исследовательских работ на кафедрах «Компьютерные технологии» и «Технологии программирования» НИУ ИТМО по следующим государственным контрактам: «*Разработка методов машинного обучения на основе генетических алгоритмов для построения управляющих конечных автоматов*» (государственный контракт № П2174 от 09.11.2009 г. по Федеральной целевой программе «Научные и научно-педагогические кадры инновационной России» на 2009 – 2013 годы), «*Применение методов искусственного интеллекта в разработке управляющих программных систем*» (государственный контракт № П2236 от 11.11.2009 г. по Федеральной целевой программе «Научные и научно-педагогические кадры инновационной России» на 2009 – 2013 годы). Результаты, полученные в диссертации, были внедрены при построении модуля *Top Traffic Monitor* для определения узлов сети с максимальным трафиком в программном продукте, выпускаемом компанией ООО *ЭВЕЛОПЕРС* (Санкт-Петербург). Результаты работы используются в учебном процессе на кафедре «Компьютерные технологии» НИУ ИТМО в курсе «Автоматное программирование».

Апробация результатов работы. Основные результаты диссертации докладывались на следующих конференциях: 32-я конференция молодых ученых и специалистов Института проблем передачи информации им. А.А. Харкевича РАН «Информационные технологии и системы» (2009), VII и VIII межвузовская конференция молодых ученых (СПбГУ ИТМО, 2010, 2011), 14-th Annual Graduate Students Workshop («Genetic and Evolutionary Computation Conference» GECCO – 2011, Dublin, ACM, 2011), вторая и третья межвузовская научная конференция по проблемам информатики СПИСОК-2011, СПИСОК-2012 (СПбГУ, 2011, 2012), I Всероссийский конгресс молодых ученых (СПбГУ ИТМО, 2012).

Личный вклад автора. Решение задач диссертации, разработанные алгоритмы и их реализация принадлежат лично автору.

Публикации. По теме диссертации опубликовано 11 печатных работ, в том числе четыре статьи, из которых три в журналах из перечня ВАК.

Структура и объем работы. Диссертация состоит из введения, четырех глав, заключения и четырех приложений. Список литературы содержит 74 наименования. Объем работы – 150 страниц, 33 рисунка и 13 таблиц.

В первой главе приводится описание основных идей автоматного программирования, верификации моделей и верификации автоматных программ. Рассказывается, как традиционная верификация моделей может быть применена для проверки корректности автоматной программы.

Во второй главе излагаются методы построения автоматов с помощью генетического программирования и верификации. Сначала рассматривается построение автоматов по темпоральным формулам и тестовым примерам, при этом верификация учитывается в функции приспособленности, операциях мутации и скрещивания. Вклад в функцию приспособленности каждой темпоральной формулы не просто ноль – формула не выполнятся и единица – формула выполняется, а зависит от

выполнения формулы на части автомата. Далее рассматривается расширение предлагаемого подхода на случай сценариев работы и контрактов. Приводятся результаты вычислительных экспериментов по построению автоматов на двух примерах: автомат управления дверьми лифта и автомат управления часами с будильником.

В третьей главе приводится сравнение трех подходов построения автоматных программ с использованием верификации. Первый – построение «вручную» и последующая верификация, второй – генетическое программирование по тестам с учетом верификации только в функции приспособленности, третий – генетическое программирование по сценариям работы и контрактам с учетом верификации в функции приспособленности, операциях скрещивания и мутации.

Первый подход обладает тем недостатком, что необходима дополнительная верификация, а при обнаружении ошибки необходимо вносить изменения в автомат. Второй и третий подходы лишены этого недостатка. Однако второй подход строит автоматы медленнее третьего. В этой главе также описываются разработанные верификатор и инструментальное средство для поддержки генетического программирования с использованием верификации.

Четвертая глава посвящена результатам внедрения технологии и разработанного инструментального средства при построении модуля для определения узлов сети с максимальным трафиком. В первой части главы описываются этот продукт и графы управления модулями, а во второй – преобразование графов в автоматы и обратное преобразование. Рассматривается построение с помощью генетического программирования и верификации одного из модулей, предназначенного для определения узлов сети с максимальным трафиком.

ГЛАВА 1. АВТОМАТНОЕ ПРОГРАММИРОВАНИЕ И ВЕРИФИКАЦИЯ АВТОМАТНЫХ ПРОГРАММ

В этой главе описаны основные идеи автоматного программирования, приведены преимущества программирования с явным выделением состояний по сравнению с традиционным подходом к созданию программ. Вторая часть главы описывает метод верификации *Model Checking*, который часто применяется на практике для проверки соответствия программ спецификации. Приведены основные принципы и методы верификации автоматных программ.

1.1. Автоматное программирование

Автоматное программирование – это парадигма программирования, при использовании которой программа или ее фрагмент осмысливается как система автоматизированных объектов управления. Особенность такого программирования состоит в явном выделении состояний[1] 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 изложена в работах [20 – 22] и является удобной при создании класса управляющих программ и их последующей верификации.

Этот подход отличается от традиционного способа программирования тем, что позволяет явно представлять состояния системы. Это обеспечивает возможность лучше анализировать работу программ, вносить в них изменения, осуществлять отладку и поиск ошибок в терминах автоматов. Все это обусловлено тем, что человеку удобно мыслить в рамках автоматной модели, где, например, вложенность

1 Автоматное программирование также называют «программированием с явным выделением состояний».

автоматов представляет собой разбиение логики системы на уровни, а отдельно выделенное состояние представляет собой состояние системы. Наличие явно выделенных состояний в автоматных программах позволяет упростить процесс их верификации, а самое главное, что верифицируется непосредственно модель и не требуется дополнительного построения модели по коду программы.

Автоматный подход к построению программ предполагает, что практически вся логика сосредоточена в управляющем автомате. При этом вспомогательная часть (например, реализация функций входных и выходных воздействий) должна быть очень простой. При традиционном подходе ветвления программы реализуются флагами и условными операторами, причем нет четких правил, где в коде надо располагать логику. В результате, полученную программу сложно читать, вносить в нее изменения и верифицировать, так как флаги очень расплывчато делят систему на отдельные части.

1.1.1. Технология автоматного программирования

Технология автоматного программирования (Switch-технология*)* – это технология разработки ПО, которая основана на методе программирования с явным выделением состояний. При таком подходе к созданию программ выделяются три типа объектов: система управления, объекты управления и поставщики событий.

Система управления представляет собой автомат или систему взаимодействующих автоматов. В диссертации предполагается, что поведение программ задается одним автоматом, а при описании поведения системой автоматов, они могут быть представлены одним автоматом за счет их «произведения». Автомат состоит из множества состояний и переходов между ними [20]. Каждый переход описывается начальным состоянием, конечным состоянием, событием, условием и выходными

воздействиями. События генерируются поставщиками событий. Система управления совершает переход по событию и при выполнении условия. Для проверки условия перехода запрашиваются значения входных переменных у объектов управления. Описанная система, как правило, называется реагирующей[2] [16] или событийной.

При программной реализации поставщики событий и объекты управления могут быть связаны между собой. На рисунке 1 схематично представлена структура взаимодействия компонент при программировании в рамках технологии автоматного программирования.

При поступлении события от поставщика автомат, находящийся в каком-либо состоянии, или автомат, вложенный в текущее состояние родительского автомата, может совершить переход, помеченный этим событием. При этом переход осуществляется только при выполнении определенных условий, значения входных переменных для которых запрашиваются у объектов управления. Условия на переходах могут быть сколь угодно сложными, например, быть сложной булевой формулой. При переходе по событию автомат может воздействовать на объекты управления, выполняя набор определенных действий.

2 В русскоязычной литературе также употребляется термин «реактивная» система.

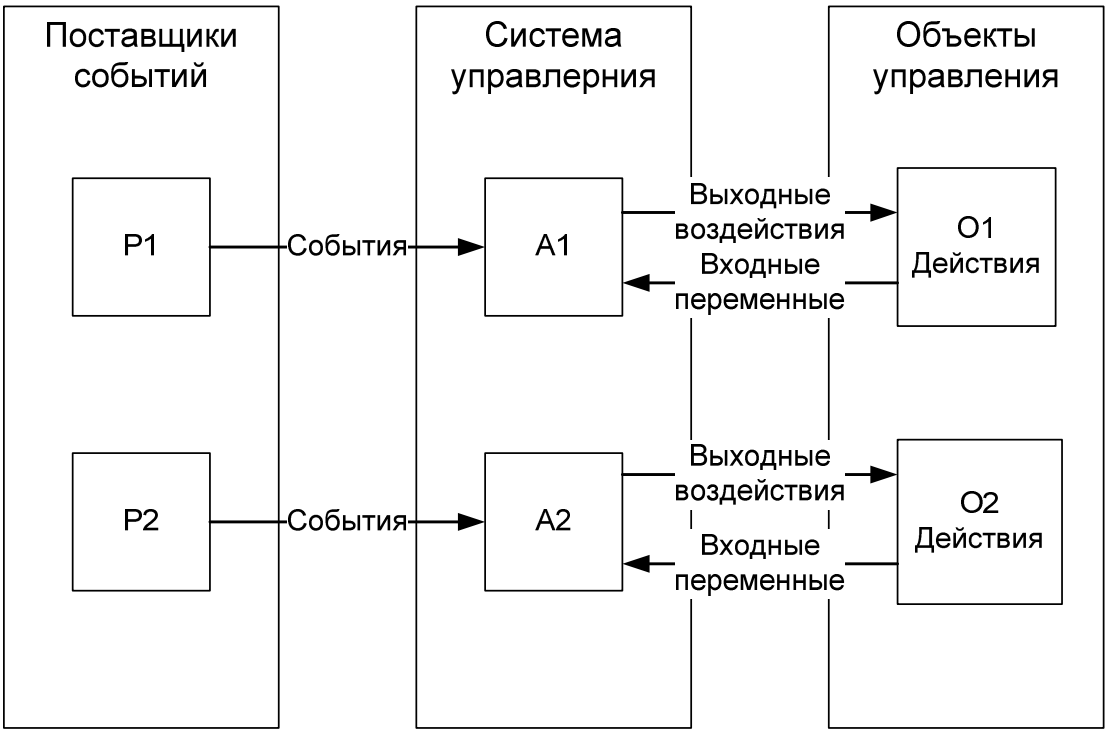


Рисунок 1 – Схема работы реагирующей системы

Поставщики событий могут быть как внешней средой, так и объектами управления.

Объекты управления по выходным воздействиям автомата реализуют те или иные действия и формируют входные переменные для автомата. Так как практически вся логика вынесена в автомат, то имеется возможность верифицировать только его, в то время как объекты управления могут подвергаться тестированию (например, покрытие *unit*-тестами).

Формально, можно рассматривать поставщики событий и объекты управления не как две разные сущности, а как абстрактную внешнюю среду, с которой взаимодействует автомат. Разбиение на два типа компонент обусловлено различными функциями объектов и их семантической составляющей. Для улучшения качества и понимания всей системы в целом, рассматривается не единственный объект управления и поставщик событий, а их множество.

Далее рассматривается автомат управления, так как его генерация с помощью генетического программирования и верификации является целью диссертации.

1.1.2. Управляющий конечный автомат

Автомат содержит состояния, переходы, условия на переходах и выходные воздействия. Формально он представляется следующим набором $<S, T, E, X, Z, s_0, L>$, где

- s_0 – начальное состояние;
- S – конечное множество состояний;
- E – множество событий;
- X – множество булевых входных переменных;
- Z – множество выходных воздействий произвольного вида;
- $T \subseteq S \times E \times 2^X \times S$ – множество переходов;
- $L : T \to 2^Z$ – функция выходных воздействий.

Автоматы могут описываться по-разному (например, таблично или графически). На рисунке 2 приведен пример простейшего автомата, заданного графом переходов. В этом автомате переход из состояния *s1* в состояние *s2* происходит при поступлении события *e1* и истинности переменной *x1* объекта управления *o1*, а переход из *s2* в *s1* – при возникновении события *e2* и истинности переменной *x2* объекта управления *o1.* При первом переходе вызывается действие *z1*, а при втором – действие *z2* того же объекта управления.

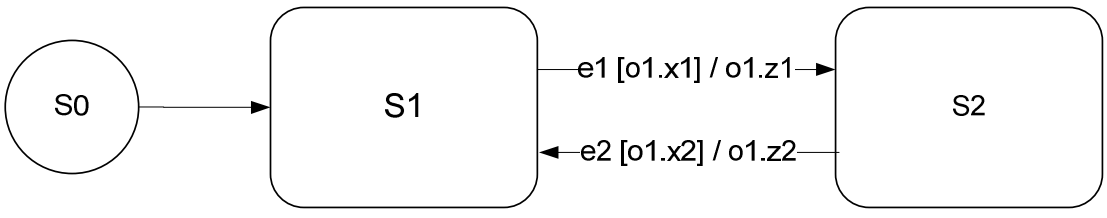


Рисунок 2 – Пример автомата из трех состояний

При автоматном подходе к созданию программ они могут достаточно просто верифицироваться, так как для них не требуется строить другие модели с конечным числом состояний. При верификации программ, написанных традиционным путем (без явного выделения состояний), потребовалось бы преобразовать программу к виду, понятному существующим верификаторам. При этом были бы утеряны некоторые данные и связи в программе, так как пришлось бы перейти на другой уровень абстракции. При создании системы в рамках технологии автоматного программирования автоматная модель может верифицироваться без преобразований или с преобразованиями, которые не приводят к потере данных.

1.2. Верификация автоматных программ

В первой части раздела представлены основы метода *Model Checking*, определяются модель Крипке и автомат Бюхи, дается определение языка логики линейного времени (*Linear Time Logic, LTL*) и в общих чертах приводится идея алгоритма верификации на моделях. Вторая часть описывает верификацию автоматных программ.

1.2.1. Традиционная верификация моделей на основе метода *Model Checking*

В настоящее время наиболее практичным методом верификации является *Model Checking* [7, 29]. В этом методе процесс верификации состоит из четырех частей.

1. Построение формальной модели по программе.
2. Построение спецификации – формальная запись свойств, которые требуется проверить.
3. Верификация – проверка выполнения спецификации на модели.

4. Определения соответствия между ошибкой в модели и ошибкой в программе.

Первая часть состоит в преобразовании программы в формальную модель с конечным числом состояний. Вторая часть (спецификация) – формальная запись утверждений, которые требуется проверить. В третьей части выполняется собственно верификация – алгоритмическая проверка выполнения спецификации для модели. Если верификация выявила ошибку, то в четвертой части осуществляется преобразование ошибки в модели в ошибку в программе, которая в дальнейшем устраняется.

В дальнейшем рассматриваются модели в виде конечных автоматов [4, 9]. При автоматном подходе модель программы представляется как множество состояний, которая в любой момент времени может быть только в одном состоянии. Каждое состояние имеет конечное описание. Имеется выделенное начальное состояние. Особый интерес в рамках настоящей работы представляют реагирующие (реактивные) системы [16], которые постоянно взаимодействуют с окружающей средой. Система переходит из состояния в состояние, и в каждый момент времени может выполняться только один переход. Такой переход означает изменение глобального состояния системы.

Для реагирующих систем характерно бесконечное выполнение – они работают бесконечно долго. Вычисление такой системы – бесконечная последовательность состояний, где каждое следующее состояние получается некоторым переходом из предыдущего.

1.2.1.1. Представление модели

При верификации традиционных программ модель задается диаграммой *Крипке*, которая представима графом переходов (S, T, s_0, L, F) [7]:

- s_0 – начальное состояние;

- S – конечное множество состояний;
- $T \subseteq S \times S$ – множество переходов;
- $L : S \to 2^{AP}$, где AP – множество атомарных высказываний;
- $F \subseteq S$ – множество допускающих состояний.

Путь в этом графе, состоящий из состояний и переходов, является *допускающим*, если существует состояние из множества *F*, принадлежащее данному пути и встречающееся бесконечно часто.

Для модели Крипке есть эквивалентная ей модель – **автомат Бюхи**. Формально он определяется (*S, T, s_0, E, F*) следующим образом:

- s_0 – начальное состояние;
- S – конечное множество состояний;
- E – конечное множество меток переходов;
- $T \subseteq S \times E \times S$ – множество переходов;
- $F \subseteq S$ – множество допускающих состояний.

Путь в автомате Бюхи определяется так же, как и в модели Крипке, только переход осуществляется в случае выполнения $T(s_{i-1}, e, s_i)$, где e – метка перехода.

Для преобразования модели Крипке (*S, T, s_0, L, F*) в автомат Бюхи (*S, T, s_0, E, F*) достаточно взять в качестве элементов множества меток E множество атомарных высказываний ($E = 2^{AP}$) и добавить другое начальное состояние с переходом в начальное состояние из модели Крипке. После этого переход $T(s_{i-1}, e, s_i)$ выполняется в том и только в том случае, когда состояние s_i в модели Крипке помечено символом *e*. При этом $e = L(s_i)$.

1.2.1.2. Временная логика *LTL*

Темпоральные логики [16] являются формализмом, который описывает последовательность переходов между состояниями системы. Темпоральные логики позволяют формализовать высказывания типа «состояние рано или поздно будет достигнуто» или «после состояния *s1*

автомат перейдет в состояние *s2*». Для записи такого вида утверждений, кроме обычных булевых операторов, используются специальные темпоральные операторы и пропозициональные переменные [7, 16].

Логика линейного времени *LTL* (*Linear Time Logic*) является подмножеством более выразительной логики CTL^* [7, 55]. Синтаксис языка *LTL* включает в себя темпоральные операторы, пропозициональные переменные *Prop* и булевы связки (*&, |,* ¬). Причем пропозициональные переменные интерпретируются как $I: Prop \rightarrow \{True, False\}$.

Логика линейного времени расширяет классическую логику, добавляя временные операторы. В этой логике время линейно и изоморфно натуральным числам. Модель можно представить как последовательность состояний, индексированных натуральными числами [54]. Пропозициональные переменные могут быть истинными или ложными в каждый момент времени (рисунок 3).

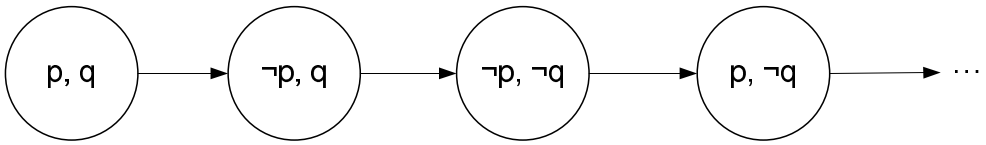


Рисунок 3 – Последовательность состояний в логике *LTL*

Для составления утверждений (формул) о событиях в будущем применяются следующие темпоральные операторы:

- ***X*** *(ne**X**t)*, где *Xp* – утверждает о выполнении предиката *p* в следующий момент времени;
- ***F*** *(in the **F**uture)*, где *Fp* – утверждает о выполнении предиката *p* в какой-то момент в будущем;
- ***G*** *(**G**lobally in the future)*, где *Gp* – утверждает о выполнении предиката *p* в любой момент времени (всегда);
- ***U*** *(**U**ntil)*, где *pUq* – утверждает о выполнении предиката *q* в некотором состоянии автомата, для которого во всех предыдущих выполняется предикат *p*;

- ***R*** ***(Release)***, где pRq – утверждает, что либо предикат p выполняется в некотором состоянии, а во всех предыдущих выполняется предикат q, либо предикат q выполняется во всех состояниях.

Множество *LTL*-формул можно задать следующим рекурсивным образом:

- *True, False*;
- *Prop* – множество пропозициональных переменных;
- Если φ и ψ – *LTL*-формулы, то:
  - $\varphi\&\psi$, $\varphi|\psi$ и $\neg\varphi$ – формулы;
  - $F\varphi$, $X\varphi$, $G\varphi$, $\varphi R\psi$ и $\varphi U\psi$ – формулы.

Отличие логики *LTL* от логики *CTL** состоит в том, что в последней присутствуют кванторы пути $\forall$ (любой путь) и $\exists$ (существует путь). Логика линейного времени говорит обо всех путях, поэтому квантор $\forall$ в формулах опущен. Таким образом, в *LTL* утверждения должны выполняются для всех путей. В этой логике доказательство можно строить от противного и проверять существование пути, на котором будет выполняться отрицание данной формулы. Если такой путь не найден, то формула выполнима.

1.2.1.3. Построение автомата Бюхи по отрицанию *LTL*-формулы

Выше было сказано, что проверку *LTL*-формулы на всех путях можно заменить проверкой существования пути, на котором выполняется отрицания *LTL*-формулы. Как по *LTL*-формуле, так и по ее отрицанию можно построить автоматы Бюхи. Эта задача *PSPACE*-полна [8]. Рассмотрим алгоритм, предложенный в работе [28], который обеспечивает трансляцию любой *LTL*-формулы в автомат Бюхи. Этот алгоритм используется в некоторых верификаторах с незначительными изменениями.

Перед применением этого алгоритма *LTL*-формула приводится в *негативную нормальную форму* [7], в которой отрицание применяется только к пропозициональным переменным. Можно считать, что в *LTL*-формулах не встречаются подформулы вида $F\varphi$ и $G\varphi$, так как их всегда можно заменить на $True\ U\ \varphi$ и $False\ R\ \varphi$ соответственно. Для приведения *LTL*-формулы в негативную нормальную форму можно воспользоваться следующими тождествами для темпоральных операторов:

- $\neg(\varphi\ U\ \psi) \equiv (\neg\varphi)\ R\ (\neg\psi)$;
- $\neg(\varphi\ R\ \psi) \equiv (\neg\varphi)\ U\ (\neg\psi)$;
- $\neg(X\varphi) \equiv X(\neg\varphi)$.

Алгоритм построения автомата Бюхи по *LTL*-формуле в негативной нормальной форме основывается на построении графа специального вида, каждая вершина в котором соответствует подформуле исходной формулы. Построение такого графа состоит в последовательном расщеплении вершин. Вместо одной вершины для операторов $|, U, R$ может быть создано две вершины, а для остальных – одна [43].

Расщепление вершины для темпоральных операторов U и R основывается на тождествах $\varphi\ U\ \psi \equiv \psi\ |\ (\varphi\ \&\ X(\varphi\ U\ \psi))$ и $\varphi\ R\ \psi \equiv \psi\ \&\ (\varphi\ |\ X(\varphi\ R\ \psi))$ соответственно.

После построения такого графа он преобразуется в автомат Бюхи, но в отличие от классического автомата Бюхи, полученный автомат содержит не одно множество допускающих состояний, а столько, сколько существует подформул вида $\varphi\ U\ \psi$. Особенность такого автомата состоит в том, что, для допущения некоторого слова, цикл, соответствующий бесконечному суффиксу, должен проходить по состояниям из каждого допускающего множества [25].

Существует множество модификаций данного алгоритма. Они все основываются на преобразовании формулы перед трансляцией с целью минимизации числа состояний. Например, в работе [42] приводится ряд

преобразований, которые позволяют получить автомат Бюхи меньшего размера.

В настоящей работе используется транслятор *LTL2BA* [57, 58]. Данное средство реализует алгоритм трансляции *LTL*-формулы в автомат Бюхи, описанный в работе [27]. Транслятор *LTL2BA* принимает на вход *LTL*-формулу, используя синтаксис верификатора *Spin* [30]. На выход выдается автомат Бюхи. Было реализовано преобразование *LTL*-формулы, используемой в настоящей работе, во входной язык транслятора, а затем построение автомата Бюхи по результату работы *LTL2BA*.

1.2.1.4. Проверка пустоты языка, допускаемого пересечением автоматов Бюхи

Выше было показано, что модель Крипке и проверяемое свойство описываются автоматами Бюхи, что позволяет выполнять верификацию за счет пересечения однотипных автоматов.

В качестве доказательства невыполнимости *LTL*-формулы на автомате Бюхи модели можно проверить, что автомат пересечения автоматов Бюхи модели и отрицания *LTL*-формулы не допускает ни одно слово.

В связи с тем, что автоматы Бюхи допускают бесконечные слова, то, как доказано в работе [7], для пустоты языка пересечения двух автоматов достаточно доказать, что ни одно допускающее состояние в автомате пересечения не принадлежит сильной компоненте связанности, которая достижима из начального состояния. Это эквивалентно тому, что через допускающее состояние не проходит цикл. Тем самым, если найден цикл, достижимый из начального состояния, то путь, на котором не выполняется *LTL*-формула, будет являться *контрпримером*.

Для обнаружения сильно связанных компонент, может быть применен алгоритм Тарьяна [8], но при верификации чаще применяют

двойной обход в глубину [7], так как для его реализации не требуется построение пересечения автоматов целиком. В этом случае состояния автомата пересечения строятся по мере их достижения, что очень важно для больших моделей.

Общая идея алгоритма такова. При достижении первым обходом в глубину допускающего состояния в автомате пересечения, запускаем второй обход в глубину для поиска цикла, проходящего через найденное состояние. В случае, если допускающее состояние оказалось достижимо из самого себя, то найден цикл. Следовательно, на автомате Бюхи, представляющем модель программы, исходная *LTL*-формула не выполняется – в программе обнаружена ошибка.

Приведем рекурсивный алгоритм двойного обхода в глубину на псевдокоде из работы [7].

```
procedure emptiness
    for all q_i ∈ Q do
        dfs1(q_i);
    terminate(False);
end procedure

procedure dfs1(q)
    local q';
    hash(q);
    for all последователей q' вершины q do
        if q' не содержится в хэш-таблице then dfs1(q');
    if accept(q) then dfs2(q);
end procedure

procedure dfs2(q)
    local q';
    flag(q);
    for all последователей q' вершины q do
        if q' в стеке dfs1 then terminate(True);
        else if q' не является помеченной then dfs2(q');
        end if;
end procedure
```

Приведенный алгоритм работает следующим образом: когда первый обход в глубину (*Depth-first search, DFS*) покидает состояние, он вызывает второй *DFS* для обнаружения циклов. Если второй *DFS* пришел в

состояние, содержащееся в стеке первого, то цикл найден. Тогда стек первого *DFS* содержит конечный префикс контрпримера, а второй обнаружил цикл, который является бесконечным суффиксом. Таким образом, язык пересечения автоматов Бюхи не пуст. Доказательство алгоритма приведено в работе [7].

Как уже отмечалось, достоинство рассмотренного алгоритма состоит в том, что можно сразу не строить автомат пересечения, а по мере достижения его состояний с помощью обхода в глубину. Если утверждение не выполняется, то этот подход позволяет обнаружить контрпример до того, как будет построено пересечение автоматов целиком.

1.2.2. Особенности верификации автоматных программ

В разд. 1.2.1 был рассмотрен метод верификации автомата Бюхи, который может быть построен из модели Крипке. Однако управляющие автоматы в автоматной программе не является ни моделью Крипке, ни автоматом Бюхи.

Для преобразования управляющего автомата в автомат Бюхи необходимо пометить переходы пропозициональными переменными. Они не указываются явно, а являются высказываниями о переходах или состояниях, которые выполняются всегда независимо от последовательности переходов автомата. Такие утверждения называются *предикатами* [3, 4].

В работах [3, 4, 44 – 47] предполагалось, что поставщики событий рассматриваются как компоненты, которые на каждом шаге могут вернуть любое событие и никак не связаны с объектами управления. При этом только одно событие может быть послано за единицу времени. Таким образом, о поставщиках событий верификатор знает только названия событий и считает, что они не хранят информации о состояниях системы.

Объекты управления рассматриваются как внешняя среда, на которую воздействует автомат посредством вызова определенных действий на переходах. Однако результат действий таких вызовов не учитывается – считаем, что объекты управления не обладают памятью. Поэтому условия на переходах в разные моменты времени могут выполняться или не выполняться независимо от результата предыдущих переходов.

В управляющем автомате при переходе из состояния в состояние никаких конкретно предикатов не указано. Однако можно самим выбрать в качестве предикатов те утверждения, которые требуется проверять при переходах. При этом для того, чтобы иметь возможность применять алгоритм двойного обхода в глубину, на утверждения о переходе накладываются ограничения, что они выполняются всегда, независимо от предыдущих переходов. Такого рода предикаты можно рассматривать в качестве пропозициональных переменных в автомате Бюхи.

Рассмотрим несколько примеров предикатов, которые допустимы для проверки:

- переход совершен по событию *p1.e1*;
- на переходе вызвано действие *o1.z1*;
- при входе в состояние вызвано действие *o1.z2*.

Таким образом, в качестве предикатов подойдут утверждения о вызванных действиях, последовательности вызовов, событиях и аналогичные утверждения.

Как было предложено в работах [4, 44 – 47], в качестве основных предикатов используются следующие утверждения:

- *wasEvent(e)* – переход совершен по событию *e*;
- *isInState(s)* – переход совершен в состояние *s*;
- *wasInState(s)* – переход совершен из состояния *s*;
- *cameToFinalState()* – при переходе автомат перешел в завершающее состояние;

- *wasAction(z)* – во время перехода было вызвано действие *z*;
- *wasFirstAction(z)* – во время перехода первым вызванным действием было *z*.

Выразительности таких предикатов может не хватать для проверки утверждений, которые могут потребоваться. Поэтому в верификаторе, реализованном в настоящей работе, обеспечена возможность создания собственных предикатов. Это позволяет не строить хитрые утверждения, использующие стандартные предикаты и логические операторы. Например, для проверки утверждения «действие *o1.z2* вызывается через одно после *o1.z1*» достаточно написать метод проверки предиката на языке *Java*. Таким образом, можно проверять такого рода утверждения, не разбивая переход на несколько частей или используя комбинацию стандартных предикатов.

При этом утверждения об управляющем автомате записываются *LTL*-формулами, содержащими предикаты. Например, формулы могут быть такими: *G(wasEvent(p1.e1))* или *F(isInState(A1.s1))*. Следовательно, синтаксис *LTL*-формул остался таким же, как в разд. 1.2.1.2.

Алгоритм верификации управляющего автомата практически такой, как при верификации автомата Бюхи. Он также использует двойной обход в глубину для проверки пустоты языка пересечения двух автоматов (управляющего автомата и автомата Бюхи для отрицания *LTL*-формулы). Единственное отличие, что предикаты не указаны на переходах, а вычисляются во время верификации – преобразование управляющего автомата в автомат Бюхи осуществляется не явно, а «на лету». Если бы все возможные предикаты заранее вычислили для каждого перехода, то в точности получился бы автомат Бюхи.

Изложенное в настоящей главе реализовано в верификаторе *AutomataVerificator*, который доступен по адресу: http://code.google.com/p/automataverificator/.

Выводы по главе 1

1. Рассмотрены основные идеи автоматного программирования, его преимущество перед традиционным созданием программ. Приведено описание управляющего конечного автомата.
2. Приведено описание автомата Бюхи и языка логики линейного времени (*LTL*). Приведен алгоритм верификации автомата Бюхи.
3. Рассмотрено как традиционная верификация моделей может быть перенесена на управляющие автоматы. Для этого необходимо преобразовать управляющий автомат в автомат Бюхи, что делается не явно, а «на лету» в процессе верификации. Приводятся основные предикаты автоматной программы, которые могут проверяться.

ГЛАВА 2. ПРИМЕНЕНИЕ ГЕНЕТИЧЕСКОГО ПРОГРАММИРОВАНИЯ И ВЕРИФИКАЦИИ ДЛЯ ПОСТРОЕНИЯ АВТОМАТОВ

Рассмотрим метод построения автоматов с помощью генетического программирования и верификации. Как отмечалось ранее, существуют различные методы построения автоматов по тестовым примерам, сценариям работы или по темпоральным формулам. Однако совместное их применение было исследовано недостаточно. Например, в работах [19, 52] вклад темпоральных формул ограничивался лишь функцией приспособленности. В настоящей работе предлагается использовать верификацию на различных стадиях алгоритма генетического программирования [24].

2.1. Модификация метода построения автоматов по темпоральным формулам и тестовым примерам

В настоящем разделе описана модификация метода, которая основана на алгоритме построения автоматов на основе тестовых примеров из работ [18, 52], однако предлагаемый подход использует верификацию в алгоритме генетического программирования не только в функции приспособленности, как в работе [52], но и в операциях скрещивания и мутации. Рассматриваемую модификацию в дальнейшем будем называть *предлагаемым методом*.

Так же, как и при создании автоматной модели только на основе тестов, запись *LTL*-формул не предполагает реализации объектов управления и поставщиков событий заранее – они могут быть созданы и после создания модели. Однако можно создавать модель при уже готовой реализации поставщиков событий и объектов управления.

Отметим, что до генерации автомата известны только входные воздействия, входные условия и выходные воздействия. Это означает, что

нельзя использовать в *LTL*-формулах предикаты о состояниях, так как заранее ничего неизвестно о структуре автомата, который будет построен.

2.1.1. Входные и выходные данные

При генерации автоматов по тестам и *LTL*-формулам, исходными данными являются:

- список событий $\{e_1, e_2, ..., e_v\}$;
- список входных переменных $\{x_1, x_2, ..., x_w\}$;
- список выходных воздействий $\{z_1, z_2, ..., z_t\}$;
- максимальное ожидаемое число состояний k;
- набор тестов $\{\{Input[1], Answer[1]\}\ldots\{Input[n], Answer[n]\}\}$;
- набор *LTL*-формул.

Результатом работы как известного, так и предлагаемого алгоритмов генетического программирования, является автомат с числом состояний не больше k, который проходит все тесты и удовлетворяет всем *LTL*-формулам.

В этих алгоритмах каждый тест содержит последовательность событий *Input[i]*, поступающих на вход автомата, и соответствующую ей эталонную последовательность выходных воздействий *Answer[i]*. Причем заранее не известно отношение между элементами из *Input[i]* и *Answer[i]*. Тест с номером i считается пройденным, если, подав на вход автомата последовательность *Input[i]*, автомат вызовет в точности последовательность из *Answer[i]*.

Каждый переход совершатся по событию при выполнении условия. Условие записывается в виде логической формулы, которая задает ограничения на значения входных переменных. Например, пометка перехода может иметь вид: *T [!x1 && x2]* – переход совершается по событию *T* при условии невыполнимости *x1* и выполнимости *x2*. При

создании автоматной модели на основе тестов, такой переход считается отличным от перехода просто по событию *T* без условий.

Однако, оба перехода (с условием и без него) идентичны для предиката по событию – *wasEvent(T)* («*переход совершен по событию T*»). При верификации, если переход был совершен по событию *T* или этому же событию, но с некоторым условием, то в обоих случаях данный предикат будет выполнен. Заметим, что можно вводить предикаты и на условия на переходах, тогда такие переходы будут отличаться и для верификатора. Например, можно добавить предикат *wasTrue(x1)* (условие *x1* верно), тогда он не будет выполняться на переходе *T [!x1 && x2]*, но будет верен для перехода *T [x1 && x2]*.

2.1.2. Особь алгоритма генетического программирования

В настоящей работе используется такое же представление автоматов в виде особи алгоритма генетического программирования, как и в работах [18, 19, 52].

Особь при генетическом программировании представлена автоматом, который содержит список переходов для каждого из состояний и номер начального состояния. Переходы задаются событиями и числом выходных воздействий, которые необходимо вызвать у объекта управления.

Таким образом, в особи кодируется только «скелет» (рисунок 4) автомата, а конкретные выходные воздействия, вырабатываемые на переходах, определяются с помощью алгоритма расстановки пометок [18].

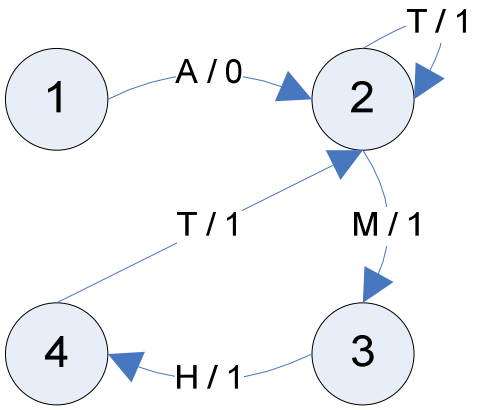


Рисунок 4 – Пример «скелета» автомата

Идея алгоритма расстановки выходных воздействий состоит в следующем: каждый из тестовых примеров запускается на «скелете» автомата, для каждого перехода запоминаются выходные воздействия, после прохождения всех тестов выбираются те выходные воздействия для перехода, которые чаще всего встречаются.

2.1.3. Функция приспособленности

В генетическом программировании ключевую роль играет функция приспособленности. Она позволяет оценивать особи: чем больше функция приспособленности, тем лучше особь. Таким образом, независимо от выбора стратегии генетического программирования, функция приспособленности лучшей особи в популяции, как правило, должна расти. В любом случае поиск автоматной модели считается завершенным, когда найдена особь, для которой значение функции приспособленности превышает некоторое целевое значение, заданное заранее.

Особь, проходящая все тесты и удовлетворяющая всем темпоральным формулам, является требуемой. Это означает, что ее функция приспособленности должна быть больше, чем у особи, не проходящей определенное число тестов или не удовлетворяющей некоторым из темпоральных формул.

Отметим, что, так же как и в работах [18, 52], в функции приспособленности должно учитываться общее число переходов в автомате, однако вклад числа переходов должен быть меньше, чем формул или тестов. Это связано с тем, что важнее найти «правильную» модель, а не модель с наименьшим числом состояний.

В настоящей работе функция приспособленности является суммой трех частей. Каждая из них представляет собой вклад одного из критериев: успешность прохождения тестов, выполнимость *LTL*-формул, число переходов в автомате.

Напомним, что вклад тестов в функцию приспособленности основан на расстоянии Левенштейна (редакционном расстоянии) [11]. Для его вычисления автомату подаются события из *Input[i]*. Затем для каждой *i*-ой последовательность событий автомат генерирует последовательность выходных воздействий *Output[i]* (для различных особей эти последовательности могут быть различными и не совпадать с *Answer[i]*). Для определения вклада тестов вычисляется величина

$$FF_1 = \frac{\sum_{i=1}^{n}\left(1 - \frac{ED(Output[i], Answer[i])}{\max(|Output[i]|, |Answer[i]|)}\right)}{n},$$

где $ED(S_1, S_2)$ – редакционное расстояние между двумя строками S_1 и S_2. Отметим, что область значений данной функции – это отрезок [0; 1]. При этом большее значение функции достигается на автомате, «лучше» проходящем тесты.

Для того чтобы выделять особи, проходящие все тесты, и те, которые проходят только часть из них, предлагалось вычислять функцию приспособленности для тестов по формуле: $FF_{test} = \begin{Bmatrix} 0.5 \cdot N \cdot FF_1, FF_1 < 1 \\ N, FF_1 = 1 \end{Bmatrix}$,

где N – «важность» прохождения всех тестов по сравнению с другими критериями. Например, для задачи, рассмотренной в разд. 2.3.2, $N = 100$.

Вклад *LTL*-формул в общую функцию приспособленности в простейшем случае оценивается как доля выполненных формул для рассматриваемой особи [52]. Этот вклад можно оценить как $\mathrm{FF_{LTL}} = \mathrm{M} \cdot \frac{\mathrm{m_1}}{\mathrm{m_2}}$, где M – «важность» выполнения всех формул, m_1 – число успешно выполненных *LTL*-формул, а m_2 – общее число формул. Таким образом, чем больше число верных формул, тем больше их вклад в функцию приспособленности, а при выполнимости всех *LTL*-формул их вклад становится равным M. Например, для задачи, рассмотренной в разд. 2.3.2, $M = 10$.

Функция приспособленности зависит не только от того, насколько «хорошо» автомат работает на тестах и удовлетворяет формулам, но и числа переходов, которые он содержит. Таким образов, функцию приспособленности можно вычислить по формуле

$$\mathrm{FF} = \mathrm{FF_{test}} + \mathrm{FF_{LTL}} + \frac{\mathrm{C} - \mathrm{T}}{10 \cdot \mathrm{C}},$$

где T – число переходов в автомате, C – число заведомо большее числа переходов, а FF_{test} и FF_{LTL} – вклады тестов и формул соответственно. Эта функция приспособленности устроена таким образом, что при одинаковом значении $FF_{test} + FF_{LTL}$, отражающем «прохождение» автоматом тестов и *LTL*-формул, преимущество будет иметь модель, содержащая меньше переходов.

Алгоритм генетического программирования сначала строит особь, проходящую все тесты и *LTL*-формулы, а потом пытается уменьшить число переходов. Для обеспечения этого требуется подбирать значение C в приведенной выше формуле таким образом, чтобы вклад успешно пройденного теста и удовлетворение темпоральной формуле был больше, чем, например, уменьшение на единицу числа переходов.

Оценим время вычисления функции приспособленности. Для редакционного расстояния время пропорционально произведению длин последовательностей, для которых оно вычисляется. Таким образом, время вычисления функции приспособленности для тестов $O(\sum_{i=1}^{n}|\text{Output[i]}|\cdot|\text{Answer[i]}|)$. Заметим, что добавление в набор тестов «префиксов» тестов не увеличивает время вычисления функции приспособленности, так как достаточно вычислить редакционное расстояние только для «самых больших» тестов, а для их префиксов редакционное расстояние взять из таблицы динамического программирования.

Оценить время верификации одной *LTL*-формулы можно только примерно. Если n – длина формулы, то в худшем случае будет построен автомат Бюхи с $n \times 2^n$ состояниями [27]. Следовательно, пересечение автоматов модели и отрицания *LTL*-формулы будет содержать $n \times 2^n \times V$ состояний, где V – число вершин в модели. Заметим, что верификатор, описанный в главе 3, использует средство *LTL2BA* [57] для построения автомата Бюхи по *LTL*-формуле, которое преобразует формулу перед построением автомата и модифицирует автомат после. Поэтому построенный автомат Бюхи не будет экспоненциально расти от длины формулы. Как правило, верифицируемые формулы достаточно компактны, что не будет приводить к автоматам с большим числом состояний.

Так как *LTL*-формулы задаются заранее, то их преобразование в автоматы Бюхи производится один раз. Однако пересечения этих автоматов с автоматом модели придется выполнять каждый раз при вычислении функции приспособленности новой полученной особи.

Заметим, что при верификации не всегда требуется целиком обходить автомат пересечения. Если модель не удовлетворяет темпоральной формуле, то контрпример может быть найден задолго до обхода всего

автомата. Но даже, когда формула выполняется, автомат пересечения может быть меньше, чем $n \times 2^n \times V$, так как некоторые вершины могут оказаться недостижимы.

Из сказанного следует, что процесс верификации в худшем случае может занимать много времени и, чем больше формул используется при построении автомата модели, тем дольше будет вычисляться функция приспособленности конкретной особи. Так как в каждом поколении могут быть тысячи особей, то выбор верификатора и его эффективность в плане производительности крайне важны. Это же учитывалось при создании верификатора из главы 3.

2.1.3.1. Учет результата верификации в функции приспособленности

Вклад каждого теста оценивается по редакционному расстоянию между эталонной выходной последовательностью и выходной последовательностью, сгенерированной особью. Тем самым, каждый тест вносит не просто ноль или единицу в функцию приспособленности, показывая, что тест пройден или не пройден на автомате, а некоторое вещественное число из отрезка [0; 1].

Предлагается оценивать вклад каждой *LTL*-формулы аналогичным образом – сделать вклад каждой формулы не дискретным, а вещественным из отрезка [0; 1]. Обычно верификаторы умеют только сообщать, что формула верна или приводить контрпример. Разработанный в настоящей работе верификатор *AutomataVerificator* был реализован таким образом, чтобы можно было помечать переходы в процессе двойного обхода в глубину. В результате, когда во время первого обхода состояние автомата покидается, все переходы, которые ведут из него, помечаются как *проверенные*. Это означает, что они точно не лежат на пути, опровергающем *LTL*-формулу.

Предлагается в качестве вклада *LTL*-формул в функцию приспособленности использовать отношение числа проверенных переходов к числу достижимых переходов. Тогда вклад *LTL*-формул записывается как $FF_{LTL} = M \cdot \sum_{i=1}^{m} t_i / T' / m$, где m – число *LTL*-формул; T' – число достижимых переходов в особи; t_i – число переходов, помеченных как проверенные при верификации i-ой формулы; M – «важность» *LTL*-формул. Чем больше переходов было отмечено в процессе верификации, тем больше вклад формулы в функцию приспособленности. Следовательно, особь является более приспособленной.

Приведем пример вычисления вклада одной *LTL*-формулы. Пусть верифицируется автомат из шести состояний, представленный на рисунке 5. Цифрой «1» выделена его часть (вершины 1, 2 и 3 автомата), на которой не удалось обнаружить контрпример («проверенные» переходы), а цифрой «2» – часть автомата, на которой формула опровергается (вершины 0, 4 и 5). Тогда вклад формулы будет 3/7, где три – число «проверенных» переходов, а семь – число переходов в автомате.

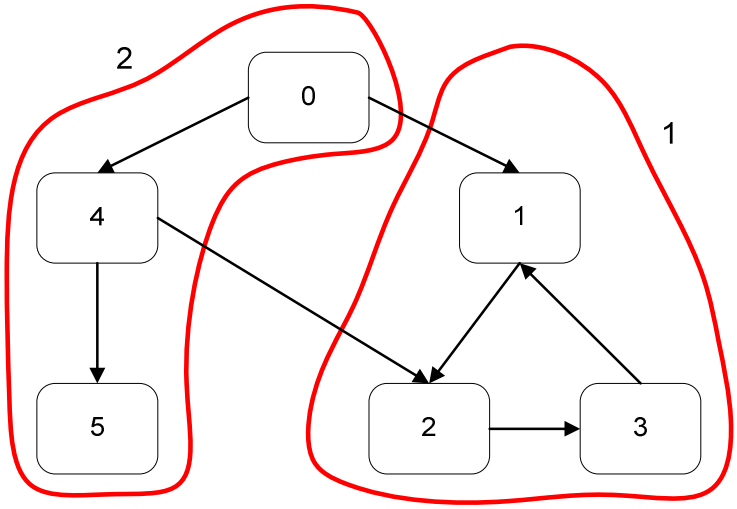


Рисунок 5 – Пример вычисления вклада *LTL*-формулы в функцию приспособленности

Заметим, что порядок обхода в глубину состояний и переходов автомата не гарантируется, поэтому могло получиться так, что контрпример мог быть найден сразу. Тогда вклад формулы в функцию приспособленности оказался бы нулевым, так как ни один из переходов не был бы помечен.

2.1.4. Мутация

Как и в предыдущих работах [18, 19, 52], мутация применяется к переходам и бывает трех видов: заменить конечное состояние перехода, заменит входное событие и изменить число выходных воздействий. Также она может удалить переход. Однако ранее мутация применялась случайным образом, а в настоящей работе предлагается учитывать верификацию.

Выполнимость или невыполнимость *LTL*-формул позволяет только отбирать «лучшие» особи, но не позволяет «улучшать» популяцию путем скрещивания или мутации, так как этот процесс был бы случайным и не гарантировал бы увеличения числа верных утверждений в следующем поколении. Следовательно, учитывая только число выполненных *LTL*-формул, не получается влиять на результат скрещивания или мутации, однако именно эти операции и приводят к росту значения функции приспособленности.

Для устранения этого недостатка при выполнении операции мутации предлагается использовать контрпример, построенный верификатором. Он представляет собой список вершин и переходов автомата, которые опровергают *LTL*-формулу. По сути, это путь в автоматной модели, на котором выполняется отрицание формулы. Имея информацию о таком пути, можно увеличить вероятность мутации для одного или нескольких переходов из контрпримера. Благодаря этому перестанет существовать данный контрпример для *LTL*-формулы, и увеличатся шансы новой особи соответствовать большему числу *LTL*-формул.

Опишем подробнее алгоритм операции мутации. При верификации выбирается самый длинный по числу переходов контрпример. Затем при создании новой особи с переходами из этого контрпримера могут происходить следующие действия:

- при копировании перехода в новую особь с некоторой вероятностью у него одновременно меняются: входное событие, число выходных воздействий и конечное состояние;
- при обычной мутации, если она заключается в удалении перехода у вершины, удаляется тот переход, который лежит на пути, опровергающем формулу.

Конечно, такие действия не гарантируют увеличение значения функции приспособленности у особи в следующем поколении. Однако, они позволяют убрать путь, на котором выполняется отрицание *LTL*-формулы, а тогда, вероятно, исходная формула будет выполняться на всех возможных путях.

Нельзя автоматически проанализировать семантику формулы и понять, как надо исправить переход для того, чтобы формула стала выполняться. Поэтому одновременно изменяются входное событие, число выходных воздействий и конечное состояние перехода, так как заранее не известно, какой из предикатов в *LTL*-формуле выполняется или не выполняется на данном пути.

Приведем пример операции мутации с учетом верификации. Пусть одна из особей в поколении оказалась такой, как представлена на рисунке 6.

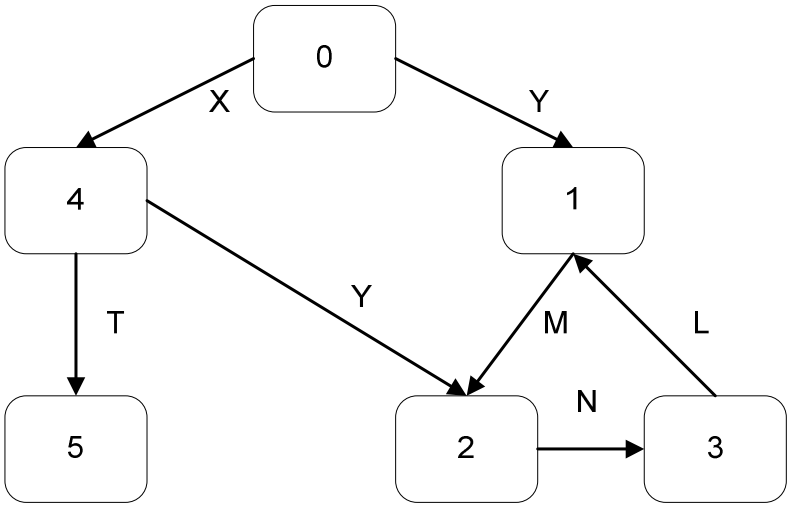


Рисунок 6 – Пример автомата

Пусть необходимо проверить утверждение *G(!wasEvent(T))*, которое означает, что никогда не будет обработано событие *T*. Так как алгоритм верификации заключается в поиске контрпримера, опровергающего формулу, то ищется путь, на котором выполняется отрицание формулы: *!G(!wasEvent(T))*. По этому отрицанию строится автомат Бюхи, представленный на рисунке 7.

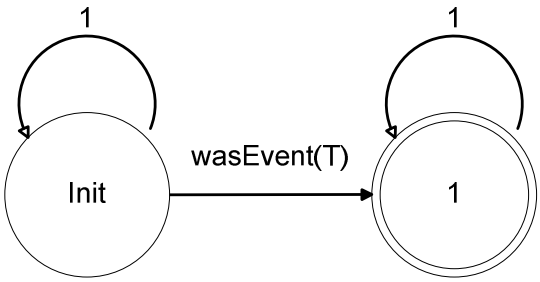


Рисунок 7 – Автомат Бюхи для формулы *!G(!wasEvent(T))*

Построенный автомат перейдет в допускающее состояние «1» в том и только в том случае, когда автомат модели совершит переход по событию *T*, так как только тогда предикат *wasEvent(T)* будет верен. Если же автомат модели не будет содержать такого перехода, или он будет недостижим из начального состояния, то автомат Бюхи никогда не сможет перейти в состояние «1» и темпоральная формула будет выполняться.

Не будем явно строить пересечение двух автоматов. Автомат Бюхи, построенный по отрицанию формулы, будет совершать переходы по петле из состояния «*Init*», пока будет обходиться автомат модели (состояния особи с номерами 0, 1, 2, 3, 4). Только после того, как автомат модели перейдет по событию *T* из состояния «4» в состояние «5», автомат, допускающий отрицание *LTL*-формулы, сможет сделать переход из состояния «*Init*» в состояние «1». Так как из состояния модели с номером 5 переходов больше нет, то автомат пересечения запустит второй обход в глубину. В результате работы верификатора в модели будет найден путь, опровергающий формулу (рисунок 8).

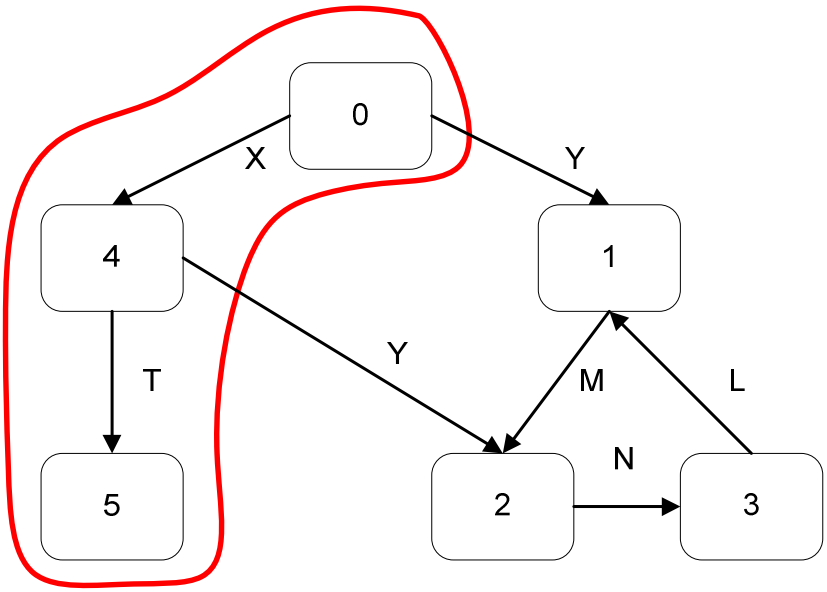


Рисунок 8 – Путь в модели, опровергающий *LTL*-формулу

Таким образом, если удалить любой переход в найденном контрпримере, переход по событию *T* перестанет существовать или быть достижимым. В то же время, может помочь и мутация, изменяющая событие на переходе, но, в нашем случае, подвергаться мутации должен именно переход по событию *T*.

Конечно, модель может оказаться не такой простой, и мутация окажется не такой эффективной. Например, если бы был переход из

состояния модели «2» в состояние «5» по событию *T*, то сначала мог быть найден путь, проходящий через состояния 0, 1, 2, 5 (рисунок 9). В таком случае мутация устранила бы данный контрпример, а формула все равно не будет выполняться. Однако, в следующем поколении верификатор найдет путь, проходящий через состояния 0, 4, 5, и устранит второй контрпример. Тем самым за два поколения найдется особь, удовлетворяющая заявленной формуле.

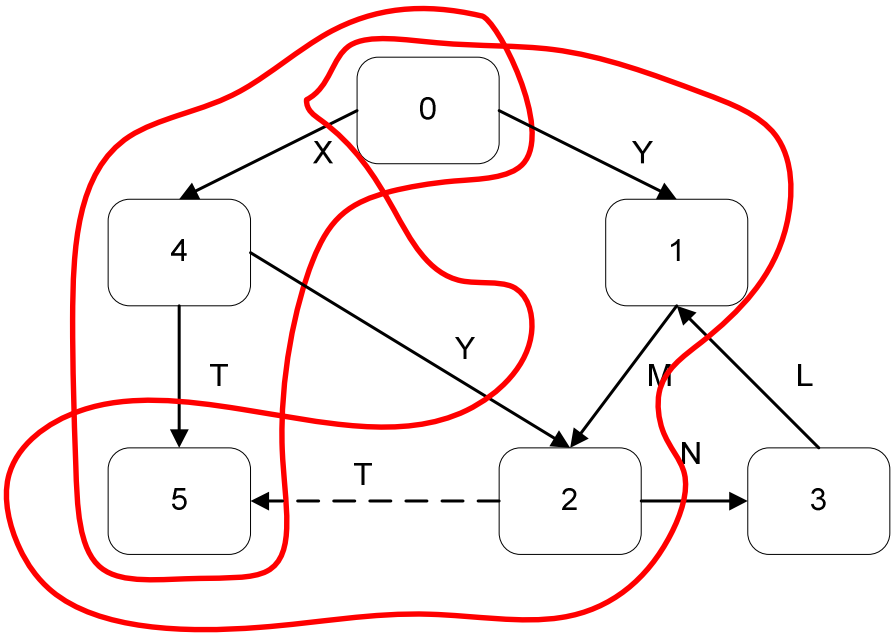


Рисунок 9 – Модель с двумя контрпримерами

В общем случае мутация остается случайным процессом, просто вероятность мутации переходов из контрпримера выше, чем остальных. При этом отметим, что устранение контрпримера может нарушить прохождение какого-нибудь теста.

2.1.5. Скрещивание

Операция скрещивания может выполняться одним из трех способов: случайно, по тестам и с учетом результата верификации. Первые два способа описаны в работах [18, 52].

Случайное скрещивание самое простое – выбираются две особи и их списки переходов «смешиваются». В результате получается особь, в которой часть переходов от одного родителя, а часть от другого.

Скрещивание по тестам основано на том, что часть автомата, на которой «хорошо» проходят тесты, переходит в новую особь без изменений, а остальные переходы «смешиваются», так же как при случайном скрещивании. Такое скрещивание позволяет сохранить часть автомата, на которой проходятся тесты, тем самым не уменьшая функцию приспособленности.

2.1.5.1. Скрещивание с учетом результата верификации

Как было изложено выше, алгоритм верификации автоматов основан на двойном обходе в глубину. Таким образом, когда первый обход в глубину покидает состояние, то подграф, образованный просмотренными состояниями и переходами, соответствует *LTL*-формуле.

Напомним, что если состояние в автомате-пересечении является допускающим, то второй обход в глубину проверяет, лежит ли данное допускающее состояние в сильной компоненте связанности. Если состояние не допускающее, или цикл, проходящий через него, не был обнаружен, то оно покидается. Так как алгоритм верификации использует обход в глубину, то все достижимые состояния из покидаемого состояния также просмотрены, и можно пометить все исходящие переходы как *проверенные*.

Приведем псевдокод предлагаемого алгоритма – жирным шрифтом выделен фрагмент, отличающий его от обычного двойного обхода в глубину, помечающий исходящие переходы как *проверенные*:

```
procedure emptiness
    for all q0 ϵ Q0 do
        dfs1(q0);
    terminate(False);
end procedure
```

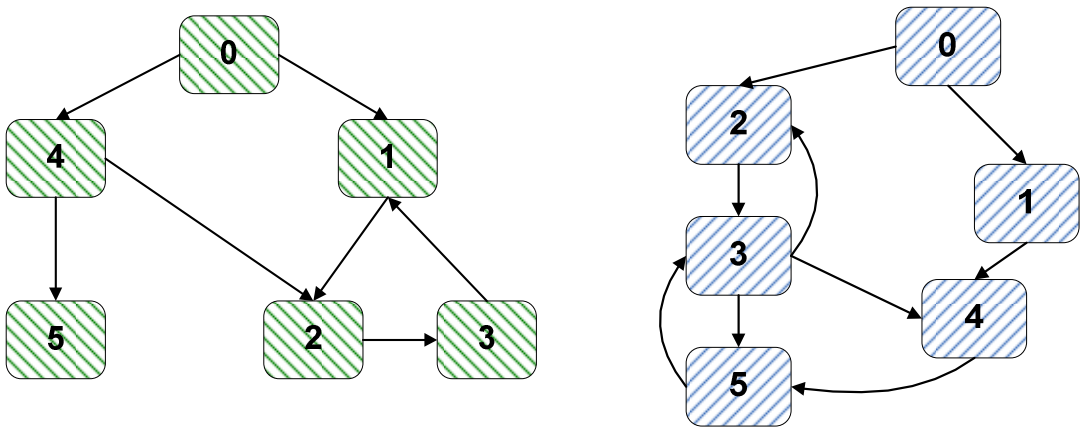
```
procedure dfs1(q)
    local q';
    hash(q);
    for all последователей q' вершины q do
        if q' не содержится в хэш-таблице then dfs1(q');
    if accept(q) then dfs2(q);
    for all переходы t из вершины q do
        markAsVerified(t);
end procedure

procedure dfs2(q)
    local q';
    flag(q);
    for all последователей q' вершины q do
        if q' в стеке dfs1 then terminate(True);
        else if q' не является помеченной then dfs2(q');
        end if;
end procedure
```

После пометки переходов, как соответствующих темпоральной формуле, можно проводить скрещивание таким образом, чтобы помеченные переходы перешли в новую особь без изменений.

Так как обычно проверяются несколько формул, то при скрещивании можно брать объединение или пересечение помеченных переходов для разных формул. Возможно, что какая-то формула выполняется, и все переходы помечены, тем самым объединением помеченных переходов будут все переходы. В то же время, не определено в какой последовательности проверяются переходы, и контрпример может быть найден сразу для неверной формулы. Тем самым, ни один переход не будет помечен, и пересечение будет пусто. Для того чтобы устранить обе эти проблемы предлагается брать объединение (или же пересечение) не для всех формул, а только для случайной их выборки.

Приведем пример такого скрещивания. Предположим, что каждая особь популяции содержит по шесть состояний, они отмечены номерами от 0 до 5 (рисунок 10).



а б

Рисунок 10 – Две особи, построенные генетическим программированием

Каждая из них не удовлетворяет всем *LTL*-формулам, но, помечая переходы в процессе верификации, можно получить некий подграф из переходов и состояний, на котором часть формул выполняется. Данный подграф перейдет в новую особь без изменений, а остальные переходы случайным образом перемешаются.

Такое скрещивание приведено на рисунке 11, выделенная часть подграфа образована помеченными переходами.

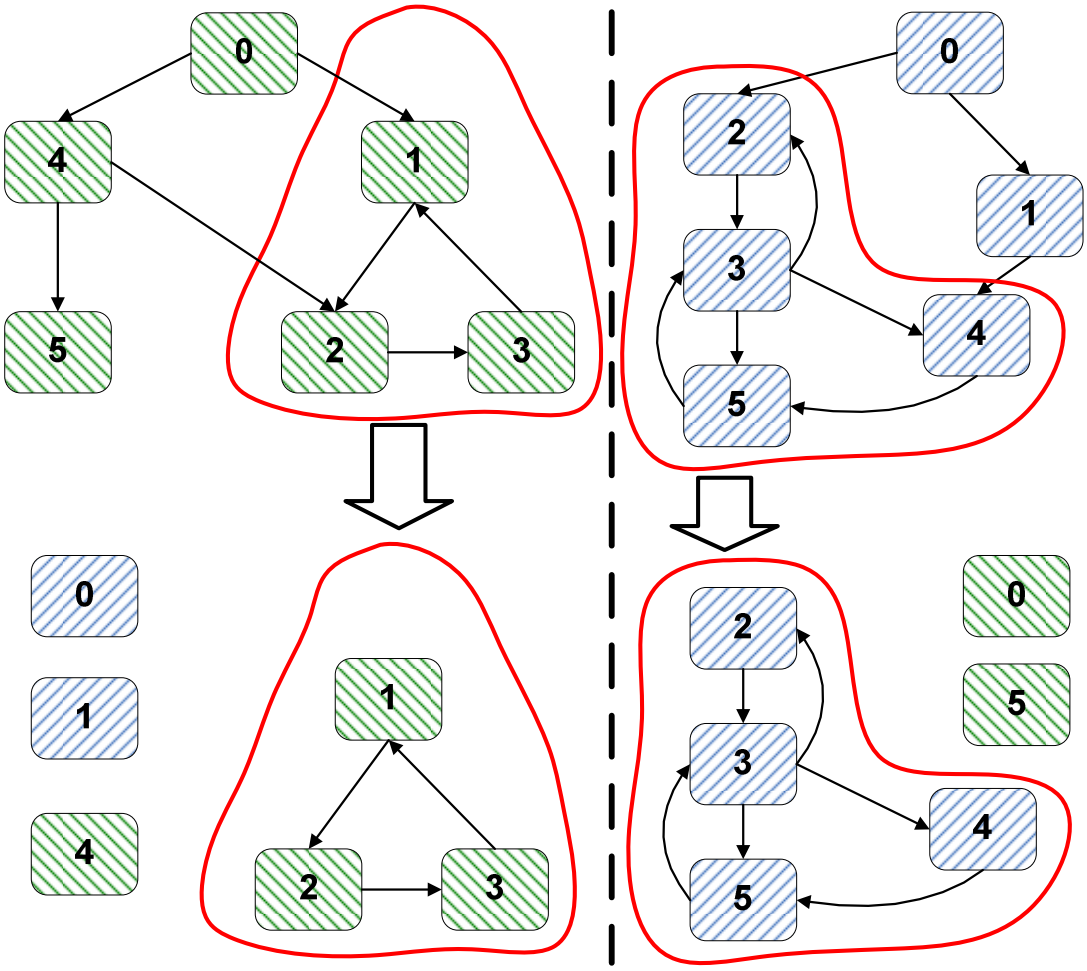


Рисунок 11 – Скрещивание с учетом результата верификации

В результате такого скрещивания часть автомата, на которой выполняются некоторые формулы, сохранится, что позволит увеличить функцию приспособленности.

2.2. Модификации алгоритма построения автоматов на основе генетического программирования и верификации

В диссертации предлагается несколько модификаций предложенного алгоритма построения автоматов. По своей сути, они являются подмножеством уже рассмотренных случаев, но, благодаря своей специфичности, позволяют упростить формализацию спецификации и ускорить построение автомата. В первой части раздела рассмотрим построение автоматов по контрактам, причем определим их через *LTL*-

формулы. Вторая часть описывает переход от тестовых примеров к сценариям работы.

2.2.1. Построение автоматов по контрактам

В настоящем разделе предлагается совместно с *LTL*-формулами использовать контракты [2, 14]. Обычно контракт состоит из предусловия, постусловия и инварианта, но может содержать только часть из них. Классическое понимание «программирования по контрактам» в объектно-ориентированном программировании:

- *предусловие* – ожидание метода объекта на входные параметры и состояние объекта при его вызове;
- *постусловие* – обязательства метода при его завершении;
- *инвариант* – условие должно выполняться на протяжении всего времени жизни экземпляра класса.

Например, контракты могут проверять диапазон входных переменных или накладывать определенные условия на возвращаемые значения методов. Инвариант может проверять, что переменная не модифицируется, что значение флага остается неизменным и т. п. Такие проверки можно было бы делать и неявно в коде программы, но задание их в явном унифицированном виде позволяет писать «чистый код», отделяя логику программы от проверки контрактов.

При автоматном подходе контракты могут накладываться как на состояния, так и на переходы и группы состояний. В таком случае, предусловие на переход может определять то, что если на переходе выполнен предикат p_1, то на предыдущем был верен p_2. Постусловие – если на переходе выполнен предикат p_1, то на следующем верен p_2. Инвариант – на переходе всегда выполняется предикат p_1. Аналогично для состояний: предусловие – до входа в состояние s верен предикат p_1,

постусловие – после выхода из состояния s верен предикат p_1, инвариант – в состоянии s верен предикат p_1.

Далее приведем формальное определение контрактов, но заметим, что одно из преимуществ контрактов перед *LTL*-формулами то, что они проще для понимания и для их записи не требуется знание языка *LTL*. Однако, язык контрактов уже, чем язык *LTL*.

При автоматическом построении автомата управления заранее не известно о состояниях автомата, поэтому не представляется возможным использование контрактов для состояний или групп состояний. Определим контракты через *LTL*-формулы. Язык *LTL* состоит из пропозициональных переменных, стандартных булевых и специальных темпоральных операторов [7]. Для настоящей работы важны два из них:

- ***X*** *(ne**X**t)*, где *Xp* – утверждает о выполнении предиката p в следующий момент времени;
- ***G*** *(**G**lobally in the future)*, где *Gp* – утверждает о выполнении предиката p в любой момент времени (всегда).

Определим **предусловие** как $G(Xp_1 \rightarrow p_2)$ – если на следующем шаге выполнено p_1, то выполнено p_2; **постусловие** как $G(p_1 \rightarrow Xp_2)$ – если выполнено p_1, то на следующем шаге выполнено p_2; **инвариант** как $G(p_1 \rightarrow p_2)$ – если выполнено p_1, то выполнено p_2. Например, предусловие на переход можно определить как $G((!e \,\&\&\, Xe) \rightarrow p)$ или как $G(Xe \rightarrow p)$, постусловие как $G((e \,\&\&\, X!e) \rightarrow Xp)$ или как $G(e \rightarrow Xp)$, инвариант как $G(e \rightarrow p)$, где e – «переход по событию e», p – предикат.

Как было изложено выше, по любой *LTL*-формуле можно построить автомат Бюхи, а алгоритм верификации основан на проверке пустоты языка пересечения, допускаемого автоматом модели и отрицанием *LTL*-формулы [7]. Можно показать, что для верификатора постусловие и предусловие оказываются одинаковыми, так как их отрицание представляются «похожими» автоматами Бюхи. В принципе, контрактом можно назвать

любую *LTL*-формулу, отрицание которой приведет к заранее заданной структуре автомата Бюхи. Под «заранее заданной структурой» будем понимать недетерминированный автомат, который эквивалентен автомату контракта с точностью до пометок на переходах. На рисунке 12 представлены автоматы для предусловия (а), постусловия (б) и инварианта (в) для перехода по событию *e*.

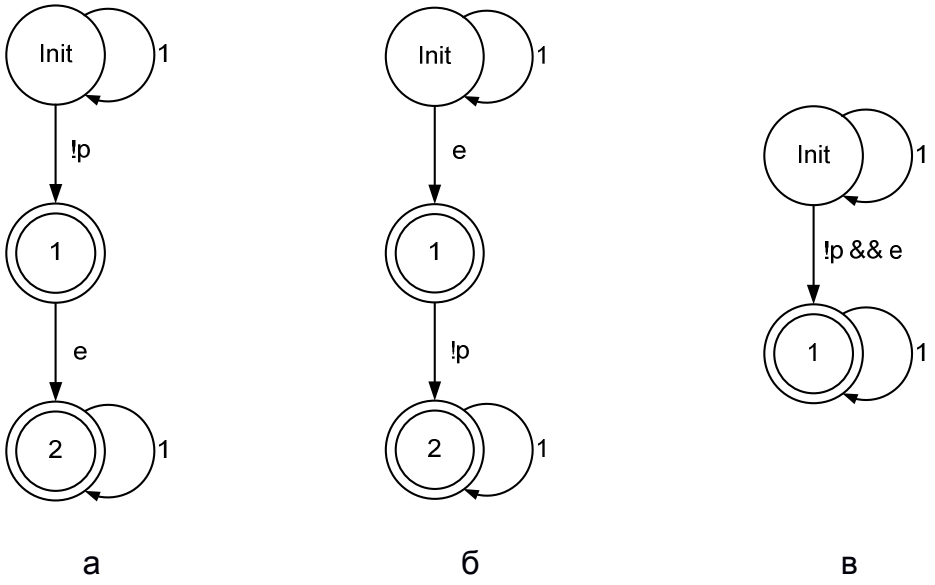


Рисунок 12 – Автоматы Бюхи, построенные для отрицания *LTL*-формул *G(Xe → p)* (а), *G(e → Xp)* (б), *G(e → p)* (в)

Так как каждый контракт представляется *LTL*-формулой, то функция приспособленности осталась такой же, как в случае темпоральных формул, но операция мутации и скрещивания должны подвергнуться некоторым изменениям.

2.2.1.1. Мутация

При верификации произвольных темпоральных формул заранее не известна их семантика. Это приводит к тому, что, обнаружив контрпример в автомате, невозможно определить, какой переход нарушает формулу. Когда автомат строится на основе контрактов, точно известно, какие переходы в контрпримере нарушают его. Например, для инварианта

последний переход нарушает его, а для предусловия и постусловия – последний и предпоследний. В результате такой априорной информации упрощается процесс модификации автомата с целью соблюдения контракта.

Как было описано в разд. 2.1.4, переходы из контрпримера с бо́льшей вероятностью подвергаются мутации, чем остальные переходы. В случае обычных *LTL*-формул выбирался любой переход из контрпримера, так как было не известно какой переход нарушает формулу. В случае же контрактов, мутации подвергается только последний или предпоследний переход, а следовательно, изменив его (входное воздействие, число выходных или его конечное состояние), данный контрпример не будет больше существовать.

Такой алгоритм мутации позволяет увеличить вклад темпоральной формулы в функцию приспособленности. Это объясняется тем, что вклад каждой формулы вычисляется как отношение проверенных переходов к общему числу переходов. Тогда, разрушив контрпример, процесс верификации больше не будет проваливаться на данных переходах и станет проверять автомат дальше, тем самым число «хороших» переходов увеличится. Однако мутация может ухудшить прохождение других формул или тестовых примеров, что естественно для генетического программирования, когда рост функции приспособленности нельзя гарантировать.

2.2.1.2. Скрещивание

Операция скрещивания отличается от обычного скрещивания по *LTL*-формулам аналогично мутации. Напомним, что скрещивание по формулам позволяло сохранять только те переходы, которые проверены верификатором и не лежат ни на одном контрпримере. Переход считается *проверенным*, если верификатор при первом обходе в глубину вернулся в

начальное состояние перехода. Таким образом, часть переходов автомата помечаются как проверенные – на них *LTL*-формула выполняется; часть переходов лежит на контрпримере, опровергающем формулу, а часть не проверена, и про них ничего не известно. Множество проверенных переходов и множество переходов, лежащих на контрпримерах («плохие» переходы), могут пересекаться.

В случае контрактов, не все переходы контрпримера помечаются как нарушающие формулу, а только один последний для инварианта и два последних для предусловия и постусловия. Таким образом, множество «плохих» переходов, содержит меньшее число элементов, чем в случае обычных *LTL*-формул. В результате, разность проверенных и «плохих» переходов, которые переходят в новую особь без изменения, оказывается не меньше случая темпоральных формул.

2.2.2. Сценарии работы

В некоторых случаях можно сузить понятие тестовых примеров до сценариев работы. Напомним, что *i*-ый тестовый пример представляет собой последовательность событий *Input[i]* и соответствующую последовательность выходных воздействий *Answer[i]*. Причем заранее не известно соответствие между *j*-ым входом *Input[i][j]* и выходными воздействиями из *Answer[i]*. Предлагается перейти к интуитивно понятному представлению теста, когда список действий на каждое из событий заранее известен – такой тест называется сценарием работы.

Позитивный сценарий работы представляет собой последовательность пар: входное воздействие и соответствующий ему список выходных воздействий. Это в некоторой степени сужение тестовых примеров, которое позволяет ускорить построение автомата. Различие между тестами и сценариями показаны на рисунке 13.

e_1	e_2	e_3	e_4	e_5
$a_1, a_2, a_3, a_4, a_1, a_5, a_3$				

а

e_1	e_2	e_3	e_4	e_5
a_1	a_2	a_3, a_4		a_1, a_5, a_3

б

Рисунок 13 – Тестовые примеры (а) и сценарии работы (б)

Использование позитивных сценариев работы не вносит изменений в функцию приспособленности. Однако требуется небольшая модификация представления особи генетического программирования и алгоритма расстановки пометок. Напомним, что в случае тестовых примеров каждый переход не содержит выходные воздействия, а только их число. В случае сценариев, число выходных воздействий перехода в особи не фиксировано. Алгоритм расстановки пометок выбирает то множество выходных воздействий для перехода автомата, которое чаще всего встречается.

Такая модификация особи генетического программирования приводит к упрощению мутации. Так как «скелет» автомата не содержит число выходных воздействий, то этот параметр не подвергается мутации. Только конечное состояние перехода или входное воздействие может быть изменено.

Применение сценариев работы ускоряет построение автомата, так как не требуется с использованием мутаций определять правильное число выходных воздействий перехода. Выходные воздействия определяются алгоритмом расстановки пометок, который может правильно расставить выходные воздействия в «скелете» автомата, в то время как в случае тестовых примеров для правильной расстановки требуется нужное число выходных воздействий на всех переходах автомата.

2.2.2.1. Негативные сценарии

Негативный сценарий представляет собой последовательность событий *nInput[i]*, которая в противоположность позитивной последовательности не должна выполняться в автомате. При этом все

префиксы негативного сценария выполняются, только последний переход не должен совершаться. Для негативного сценария не важны выходные воздействия, так как они определяются в позитивных сценариях.

Вклад в функцию приспособленности каждого негативного сценария дискретен. Если существует последовательность событий в автомате, на которой сценарий выполняется, то он проходит, и его вклад – *-1*; если последовательности не существует, вклад сценария равен нулю. Вклад всех негативных сценариев считается как отношение суммы вкладов каждого сценария к их общему числу. При этом функция приспособленности принимает вид:

$$FF=FF_{ptest}+FF_{ntest}+FF_{LTL}+\frac{C-T}{10\cdot C}$$

Здесь FF_{ptest} – вклад позитивных сценариев работы, FF_{LTL} – вклад *LTL*-формул, FF_{ntest} – вклад негативных сценариев, T – число переходов автомата, C – число заведомо большее числа переходов.

Скрещивание и мутация могут учитывать негативные сценарии таким образом, что последний переход, который позволил негативному сценарию пройти, не может сохраняться в новой особи. Тем самым, последние переходы негативных сценариев с большей вероятностью подвергаются мутации и не переходят в новую особь без изменений, подобно скрещиванию по тестам или *LTL*-формулам (разд. 2.1.5).

Как уже неоднократно отмечалось в настоящей работе, нельзя утверждать о правильности построенной системы только на основе позитивных тестов. Это же утверждение переносится и на сценарии работы, так как они не имеют такой же выразительности как временные утверждения.

2.3. Экспериментальные исследования

В работе исследовалось построение двух автоматов: автомат управления часами с будильником и автомат управления дверьми лифта.

Каждый эксперимент принимает на вход *LTL*-формулы, множество тестовых примеров или сценарии работы. Кроме спецификации, алгоритм генетического программирования имеет следующие параметры:

- размер начальной популяции;
- число состояний у автоматов в начальном поколении;
- ожидаемое число переходов;
- доля особей, переходящих в следующее поколение (остальные будут получены с помощью операции скрещивания);
- число поколений до «малой» мутации;
- число поколений до «большой» мутации;
- вероятность мутации особи.

Под «малой» мутацией понимается сохранение 1/10 лучших особей и случайной мутацией 9/10 поколения. Такая мутация выполняется, если функция приспособленности не растет в течение заданного числа поколений. «Большая» мутация означает создание новой популяции, если функция приспособленности не растет в течение заданного числа поколений. Такие мутации направлены на преодоление локальных максимумов, когда функция приспособленности лучшей особи перестает расти, но не все формулы или тесты выполняются.

Для оценки эффективности рассмотренных выше алгоритмов измеряется число вычислений функции приспособленности. Выбор данного критерия обусловлен тем, что он не зависит от размера начальной популяции, вычислительных мощностей компьютера и т. п. Например, если рассматривать число поколений генетического программирования в качестве оценки эффективности алгоритма, то оно будет различаться в зависимости от размера популяции.

Заметим, что предлагаемый подход не позволяет строить автомат только на основе *LTL*-формул, так как тесты и сценарии работы используются для расстановки выходных воздействий на переходах.

Применение только темпоральных формул приводит к слишком медленному росту функции приспособленности. Только совместное использование тестов и *LTL*-формул позволяет добиться прогнозируемого роста функции приспособленности и заранее заданного поведения построенного автомата.

2.3.1. Автомат управления дверьми лифта

Предложенные алгоритмы построения автоматов и операций скрещивания и мутаций исследовались на задаче построения автомата управления дверьми лифта [45]. Двери лифта могут открываться и закрываться. Если при их закрытии появилось препятствие, то требуется прекратить закрытие и открыть двери. Как и любой лифт, рассматриваемый может сломаться в процессе открытия или закрытия дверей, и тогда необходим звонок в аварийную службу.

Заметим, что после поломки лифта не предусмотрено возвращение автомата в работоспособное состояние – в модели не предусмотрено событие «ремонт окончен». В этом случае предполагается, что программа будет перезапущена, и автомат перейдет в начальное состояние.

Поставщик событий для автомата управления лифтом содержит пять событий [45]:

- *e11* – нажата кнопка «Открыть двери»;
- *e12* – нажата кнопка «Закрыть двери»;
- *e2* – двери лифта полностью открылись или закрылись;
- *e3* – закрытию дверей мешает препятствие;
- *e4* – поломка дверей лифта.

Объект управления имеет три выходных воздействия [45]:

- *z1* – приступить к открытию дверей;
- *z2* – приступить к закрытию дверей
- *z3* – сообщить в аварийную службу о поломке лифта.

Поведение дверей лифта может быть описано автоматом (рисунок 14), построенным вручную в работе [45].

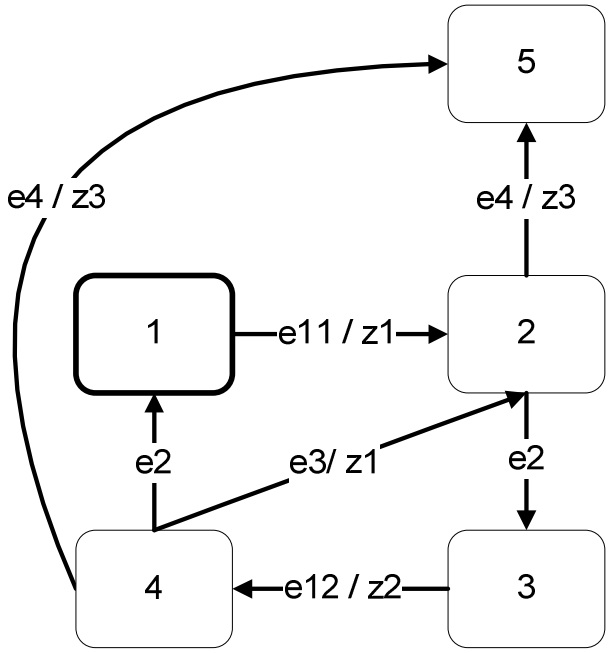


Рисунок 14 – Автомат управления дверьми лифта

Начальное состояние имеет номер «1» и выделено жирно.

2.3.1.1. Система тестовых примеров и *LTL*-формул

В систему тестов для построения автомата управления дверьми лифта входят девять тестов [45]. Они описывают различные варианты поведения дверей лифта и представлены в таблице 1.

Таблица 1 – Тесты для автомата управления дверьми лифта

Тест	Комментарий
Input: e11, e2, e12, e2 *Answer: z1, z2*	Процесс открытия и закрытия дверей лифта без возникновения препятствий и поломок.
Input: e11, e2, e12, e2, e11, e2, e12, e2 *Answer: z1, z2, z1, z2*	Предыдущий тест повторенный дважды.

Тест	Комментарий
Input: e11, e2, e12, e3, e2, e12, e2 *Answer: z1, z2, z1, z2*	Процесс закрытие дверей при возникновении препятствия. Дверь закрывается после повторного нажатия кнопки «Закрыть двери».
Input: e11, e2, e12, e2, e11, e2, e12, e3, e2, e12, e2 *Answer: z1, z2, z1, z2, z1, z2*	Возникновение препятствия при втором закрытии дверей. Двери закрывается со второго раза.
Input: e11, e2, e12, e3, e2, e12, e3, e2, e12, e2 *Answer: z1, z2, z1, z2, z1, z2*	Две попытки закрыть двери при возникновении препятствия, двери успешно закрываются с третьего раза.
Input: e11, e4 *Answer: z1, z3*	Процесс поломки дверей при их открытии.
Input: e11, e2, e12, e4 *Answer: z1, z2, z3*	Процесс поломки дверей при их закрытии.
Input: e11, e2, e12, e2, e11, e4 *Answer: z1, z2, z1, z3*	Процесс поломки дверей при повторном открытии.
Input: e11, e2, e12, e3, e4 *Answer: z1, z2, z1, z3*	Процесс поломки дверей при возникновении препятствия.

Совместно с тестами применялось 11 *LTL*-формул (таблица 2) [19].

Таблица 2 – *LTL*-формулы, описывающие автомат управления дверьми лифта

Формула	**Комментарий**
G(wasEvent(e11) => wasAction(z1))	При переходе по событию *e11* вызывается действие *z1*. В терминах модели: при обработке события «Открыть двери», обязательно будет начато открытие дверей. Контракт – инвариант.
G(wasEvent(e12) <=> wasAction(z2))	Переход по событию *e12* совершается в том и только том случае, когда вызывается действие *z2*. В терминах модели: при нажатии кнопки «Закрыть двери» будет начато их закрытие. Закрытие дверей может начаться только при нажатии кнопки «Закрыть двери». Контракт – инвариант.
G(wasEvent(e4) <=> wasAction(z3))	Переход по событию *e4* совершается в том и только в том случае, когда вызывается действие *z3*. В терминах модели: звонок в аварийную службу будет произведен тогда и только тогда, когда лифт сломается. Контракт – инвариант.

Формула	Комментарий
G(wasEvent(e3) => wasAction(z1))	При переходе по событию *e3* вызывается действие *z1*. В терминах модели: если препятствие мешает закрыть двери, то дверь начнет открываться. Контракт – инвариант.
G(wasEvent(e2) => X[wasEvent(e11) || wasEvent(e12)])	При переходе по событию *e2* следующий переход будет по событию *e11* или *e12*. В терминах модели: если дверь успешно открылась или закрылась, то следующим обработанным событием может быть только «Открыть двери» или «Закрыть двери». Контракт – постусловие.
G(wasEvent(e11) => X[wasEvent(e2) or wasEvent(e4)])	При переходе по событию *e11* следующий переход будет по событию *e2* или *e4*. В терминах модели: если была нажата кнопка «Открыть двери», то следующее событие будет либо успешное открытие дверей, либо дверь сломалась. Контракт – постусловие.

Формула	Комментарий
G(wasAction(z1) => X[wasEvent(e2) or wasEvent(e4)])	При переходе по событию *z1* следующий переход будет по событию *e2* или *e4*. В терминах модели: если дверь начала открываться, то либо она успешно откроется, либо сломается. Контракт – постусловие.
G(wasEvent(e12) => X[wasEvent(e2) or wasEvent(e3) or wasEvent(e4)])	При переходе по событию *e12* следующий переход будет по событию *e2* или *e3* или *e4*. В терминах модели: если нажали кнопку «Закрыть двери», то либо двери закроются, либо препятствие помешает их закрыть, либо лифт сломается. Контракт – постусловие.
G(wasAction(z1) => X[U(!wasAction(z1), wasAction(z2) or wasEvent(e4))])	Если при переходе вызвано действие *z1*, то не будет больше перехода с таким же выходным воздействием, пока не будет перехода с *z2* или не совершится переход по событию *z4*. В терминах модели: если двери начали открываться, то они не начнут повторное открытие пока не закроются или сломаются.

Формула	Комментарий
G(wasAction(z2) => X[U(!wasAction(z2), wasAction(z1) or wasEvent(e4))])	Если при переходе было вызвано действие *z2*, то не будет больше перехода с таким же выходным воздействие, пока не произойдет переход с *z1* или не совершится переход по событию *e4*. В терминах модели: если дверь начала закрываться, то она не будет снова закрываться до тех пор, пока она не будет отрываться или не сломается.
!F(wasEvent(e4) and X(F(wasEvent(e11) || wasEvent(e12) || wasEvent(e2) || wasEvent(e3) || wasEvent(e4))))	Отрицание того, что в будущем после перехода по событию *e4* будут когда-либо совершены переходы по событиям *e11*, *e12, e2*, *e3* или *e4*. В терминах модели: не верно, что после поломки лифта будут обработаны события «Открыть двери», «Закрыть двери», «Успешное открытие дверей», «Успешное закрытие дверей», «Препятствие мешает закрыть двери», «Лифт сломался». Следовательно, лифт не может быть починен.

При построении автомата, управляющего дверьми лифта, не получается использовать только *LTL*-формулы без тестовых примеров или сценариев работы. Применять только тестовые примеры тоже не удается, так как по ним строится неправильный автомат, что будет рассмотрено ниже.

2.3.1.2. Результаты экспериментов

Для сравнения различных способов построения автоматов (на основе тестов, на основе тестов и *LTL*-формул, на основе тестов, *LTL*-формул и контрактов, на основе сценариев работы, *LTL*-формул и контрактов) было проведено несколько экспериментов. Каждый эксперимент проводился по 1000 раз. Параметры каждого эксперимента были идентичны (таблица 3).

Таблица 3 – Параметры эксперимента построения автомата, управляющего дверьми лифта

Параметр эксперимента	**Значение**
Размер начальной популяции	2000
Число состояний автоматов в начальном поколении	6
Ожидаемое число переходов	7
Доля особей, переходящих в следующее поколение. Остальные будут получены с помощью скрещивания	10 %
Число поколений до «малой» мутации	70
Число поколений до «большой» мутации	100
Вероятность мутации особи	1 %

В результате экспериментов, использующих только тестовые примеры, более чем в 99 % случаях получался неправильный автомат. Один из таких автоматов представлен на рисунке 15, «жирная» вершина – начальное состояние.

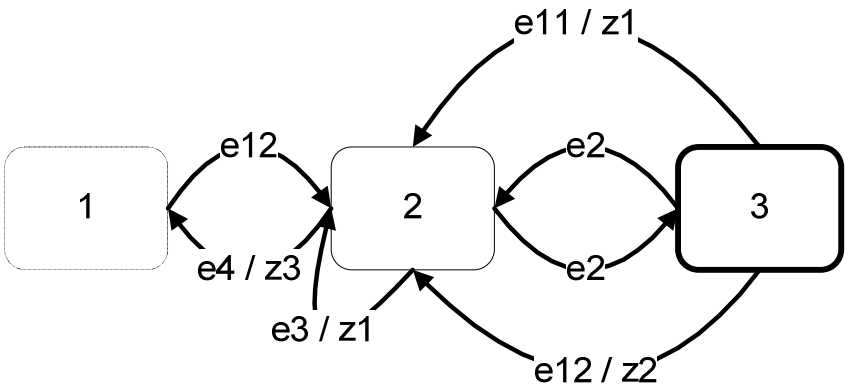


Рисунок 15 – Граф переходов автомата управления дверьми лифта, построенного только на основе тестов

Ошибка в представленном автомате заключается в том, что после поломки лифта дверь может снова начать закрываться, а затем лифт начнет функционировать как работоспособный. Также данный автомат может повторно начать закрывать или открывать двери после их закрытия или открытия соответственно.

Отметим, что теоретически мог бы быть построен и автомат из одного состояния, все переходы которого являлись бы петлями. Такой автомат также проходил бы все тесты, но, естественно, был бы абсолютно неверным.

Указанных проблем можно избежать, применяя верификацию. При построении автомата совместно на основе тестов и *LTL*-формул был построен правильный автомат, изоморфный автомату, изображенному на рисунке 14.

В этом случае функция приспособленности вычислялась по формуле

$$\mathrm{FF} = \mathrm{FF}_{\mathrm{test}} + \mathrm{FF}_{\mathrm{test}} \cdot \mathrm{FF}_{\mathrm{LTL}} + \frac{\mathrm{C} - \mathrm{T}}{10 \cdot \mathrm{C}}.$$

В отличие от формулы из разд. 2.1.3, из вклада тестов (FF_{test}) удалена надбавка за прохождения всех тестов и $N = M = 1$, так как для автомата, проходящего все тесты, могут не выполняться *LTL*-формулы. В приведенной формуле отражено то, что прохождение тестов тоже важно, и

автомат, не проходящий все тесты, но удовлетворяющий всем *LTL*-формулам, имеет нулевую функцию приспособленности. Последняя компонента формулы отражает вклад числа переходов. Значение C было выбрано равным 100, и вклад числа переходов в ходе вычисления функции приспособленности достигал максимума 0.093 на особях, проходящих все тесты и *LTL*-формулы.

Эксперименты **первого** вида заключались в построении автомата управления дверьми лифта только на основе тестов. Значение функции приспособленности построенного автомата было равно 1.093. В качестве численной оценки скорости работы алгоритма измерялось число вычислений функции приспособленности при нахождении автомата. Среднее значение вычислений функции приспособленности равно 7.479×10^4. Минимальное число вычислений – 2.184×10^4. Максимальное число – 2.999×10^5. Среднеквадратичное отклонение – 2.54×10^4. Плотность распределения вероятности числа вычислений функции приспособленности при построении автомата только на основе тестов представлена на рисунке 16.

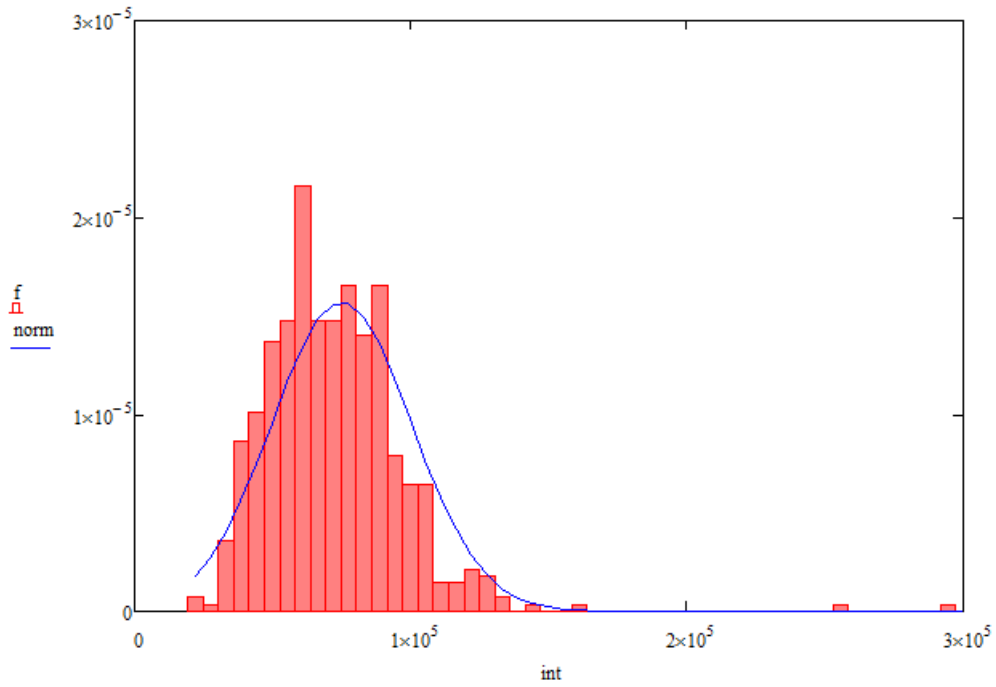


Рисунок 16 – Плотность распределения вероятности числа вычислений функции приспособленности при построении автомата управления дверьми лифта только на основе тестов

Второй вид экспериментов заключался в построении автомата, управляющего дверьми лифта на основе тестов и *LTL*-формул. Значение функции приспособленности построенного автомата – 2.093. Среднее значение вычислений функции приспособленности равно 8.372×10^5. Минимальное число вычислений – 6.331×10^4. Максимальное число – 5.912×10^6. Среднеквадратичное отклонение – 7.57×10^5. Плотность распределения вероятности числа вычислений функции приспособленности при построении автомата на основе тестов и *LTL*-формул представлена на рисунке 17.

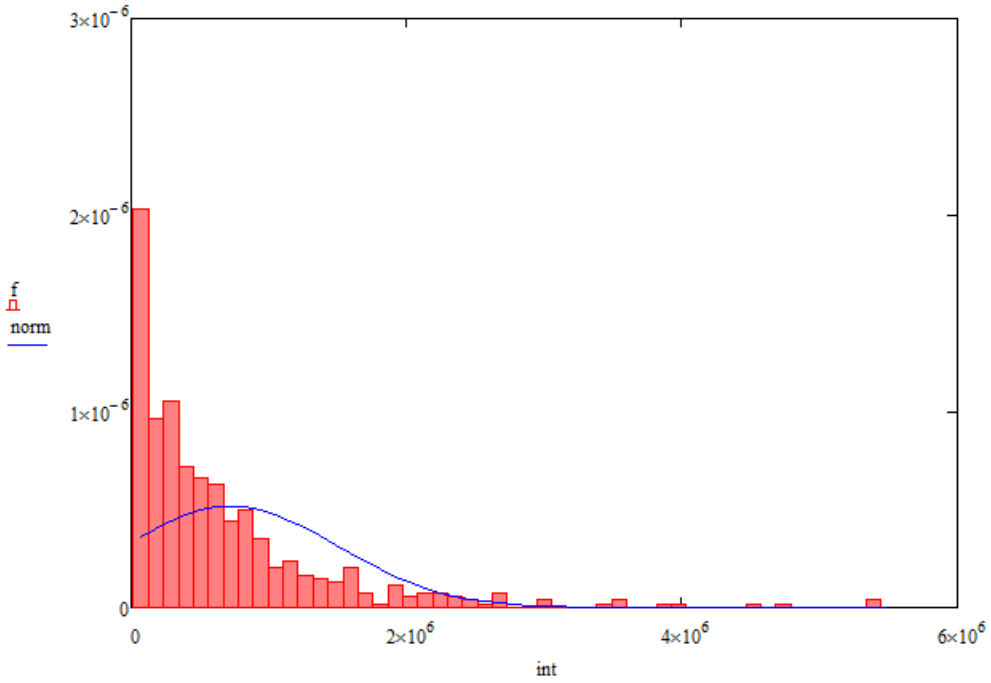


Рисунок 17 – Плотность распределения вероятности числа вычислений функции приспособленности при построении автомата управления дверьми лифта на основе тестов и *LTL*-формул

Несмотря на то, что число вычислений функции приспособленности для экспериментов с *LTL*-формулами оказалось больше практически в 10 раз, все 1000 построенных автоматов были правильными. Однако из 1000 построений автоматов только на основе тестов, только девять из них не содержали ошибок.

Третий вид экспериментов заключался в построении автомата на основе тестов, *LTL*-формул и контрактов. Контракты и *LTL*-формулы взяты из таблицы 2, так как контракты – это *LTL*-формулы специального вида. Среди контактов пять инвариантов и четыре постусловия. Значение функции приспособленности построенного автомата – 2.093. Среднее значение вычислений функции приспособленности равно 7.109×10^5. Минимальное число вычислений – 6.153×10^4. Максимальное число – 4.589×10^6. Среднеквадратичное отклонение – 6.32×10^5.

Из результатов экспериментов третьего вида следует, что применение контрактов позволило сократить время построения автомата. Причем остальные параметры экспериментов также уменьшились.

Четвертый вид экспериментов состоял в построении автомата на основе позитивных и негативных сценариев работы совместно с *LTL*-формулами. Позитивные сценарии работы были получены путем преобразования каждого тестового примера из таблицы 1. При этом для каждого из них было построено соответствие между входом (*Input*) и выходом (*Answer*). В качестве примера приведем первый позитивный сценарий: e11/z1, e2, e12/z2, e2.

30 негативных сценариев было добавлено автором (таблица 4).

Таблица 4 – Негативные сценарии работы для автомата управления дверьми лифта

Негативный сценарий	**Комментарий**
e12	Эти четыре сценария свидетельствуют о том, что в стартовом состоянии двери лифта закрыты, поэтому нельзя обработать никакую команду, кроме *e11* (открыть двери лифта)
e2	
e3	
e4	
e11, e11	После обработки команды *e11* (открыть двери), следующее событие может быть одно из двух: либо *e2* (двери открылись), либо *e4* (двери сломались).
e11, e12	
e11, e3	

Негативный сценарий	Комментарий
e11, e2, e11	После полного открытия дверей, следующая обработанная команда может быть только *e12* (закрыть двери).
e11, e2, e2	
e11, e2, e3	
e11, e2, e4	
e11, e2, e12, e11	Если двери лифта закрываются, то не может обработаться команда *e11* (открыть двери).
e11, e2, e12, e12	Если двери лифта закрываются, то не может повторно обработаться команда *e12* (закрыть двери).
e11, e2, e12, e3, e11	Если препятствие мешает закрыть двери, то двери могут либо сломаться, либо открыться.
e11, e2, e12, e3, e12	
e11, e2, e12, e3, e3	
e11, e2, e12, e2, e12	Если двери закрылись, то следующая единственно возможная обработанная команда *e11* (открыть двери).
e11, e2, e12, e2, e2	
e11, e2, e12, e2, e3	
e11, e2, e12, e2, e4	
e11, e2, e12, e4, e11	Если лифт сломался в процессе закрытия дверей, то больше никакие другие события не могут быть обработаны.
e11, e2, e12, e4, e12	
e11, e2, e12, e4, e2	
e11, e2, e12, e4, e3	
e11, e2, e12, e4, e4	
e11, e4, e11	Если лифт сломался в процессе открытия дверей, то больше никакие другие события не могут быть обработаны.
e11, e4, e12	
e11, e4, e2	
e11, e4, e3	
e11, e4, e4	

Значение функции приспособленности построенного автомата – 2.093. Среднее значение вычислений функции приспособленности равно 1.616×10^5. Минимальное число вычислений – 3.808×10^4. Максимальное число – 8.162×10^5. Среднеквадратичное отклонение – 1.102×10^5.

Применение сценариев позволило ускорить построение автомата более чем в четыре раза по сравнению с тестовыми примерами. Однако, для описания запрещенного поведения автомата потребовалось записать 30 негативных сценариев. Более того, все приведенные сценарии имеют максимальное число переходов равное пяти, что не гарантирует правильного поведения на более длинных сценариях, которые требуют обязательного применения верификации.

Сценарии могут обладать избыточностью и быть описаны *LTL*-формулами. Например, последние 10 негативных сценариев из таблицы 4 могут быть записаны в виде одной *LTL*-формулы: $G(e4 \rightarrow !X(e11 \text{ or } e12 \text{ or } e2 \text{ or } e3 \text{ or } e4))$, которая означает, что после события *e4* не могут быть события *e11, e12, e2, e3* и *e4*. В работе использовалась еще более общая формула: $!F(e4 \text{ and } X(F(e11 \text{ or } e12 \text{ or } e2 \text{ or } e3 \text{ or } e4)))$ – неверно, что в будущем после события *e4* когда-либо обрабатываются *e11, e12, e2, e3* или *e4* (последняя формула в таблице 2).

Результаты экспериментов сведены в таблицу 5. В ней не приведены результаты экспериментов только с тестовыми примерами, так как правильный автомат получался только в 1 % случаев и, следовательно, сравнение алгоритмов некорректно.

Таблица 5 – Число вычислений функции приспособленности при генерации автомата управления дверьми лифта

	Эксперимент		
	Тесты и *LTL*-формулы	Тесты, *LTL*-формулы и контракты	Сценарии, *LTL*-формулы и контракты
Среднее значение	8.372×10^5	7.109×10^5	1.616×10^5
Среднеквадратичное отклонение	7.57×10^5	6.32×10^5	1.102×10^5
Минимальное значение	6.331×10^4	6.153×10^4	3.808×10^4
Максимальное значение	5.912×10^6	4.589×10^6	8.162×10^5

Полученные результаты позволяют утверждать, что, несмотря на «избыточность» негативных сценариев, применение сценариев работы позволило добиться четырехкратного ускорения времени построения автоматов управления дверьми лифта. *XML*-описание параметров эксперимента, тестовых примеров, сценариев работы и *LTL*-формул приведены в приложении 1. Результат работы эксперимента представляет собой *UniMod*-модель автомата в *XML*-формате (приложение 2).

2.3.2. Автомат управления часами с будильником

Как и в работах [18, 52] генерация автоматов исследовалась на задаче построения автомата, управляющего часами с будильником [15]. Эти часы позволяют менять режим работы устройства, настраивать текущее время и время срабатывания будильника. Для этого служат три кнопки на корпусе часов (рисунок 18).

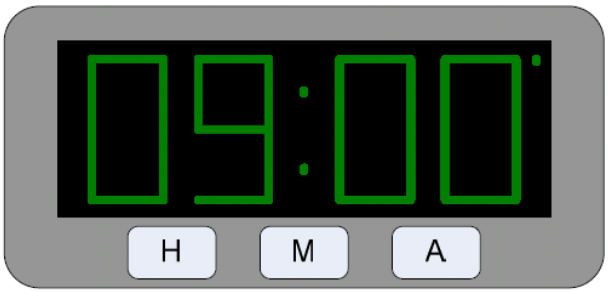


Рисунок 18 – Внешний вид часов с будильником

Часы с будильником работают в двух режимах: режим часов, и режим будильника. Кнопка «А» служит для перехода из одного режима в другой и обратно, а также для выключения сигнала будильника. Кнопка «Н» предназначена для увеличения текущего времени или времени срабатывания будильника на один час. Кнопка «М» предназначена для увеличения текущего времени или времени срабатывания будильника на одну минуту. Будильник срабатывает, как только время уставки совпадает с текущим временем. Звонок автоматически выключается через минуту или нажатием кнопки «А». Текущее время увеличивается на одну минуту таймером, который посылает событие ровно раз в минуту [15].

Множество событий состоит из четырех элементов [15]:

- *H* – нажатие кнопки «Н»;
- *M* – нажатие кнопки «М»;
- *A* – нажатие кнопки «А»;
- *T* – срабатывание таймера (раз в минуту).

Объект управления содержит две входные переменные [15]:

- *x1* – верно ли, что время уставки будильника совпадает с текущим временем?
- *x2* – верно ли, что текущее время превышает время уставки будильника ровно на одну минуту?

Объект управления содержит семь выходных воздействий:

- *z1* – перевести текущее время на один час;

- *z2* – перевести текущее время на одну минуту;
- *z3* – перевести время срабатывания будильника на один час;
- *z4* – перевести время срабатывания будильника на одну минуту;
- *z5* – увеличить текущее время на минуту;
- *z6* – включить сигнал будильника;
- *z7* – выключить сигнал будильника.

Часы с будильником описываются автоматом (рисунок 19), который построен вручную в работе [15].

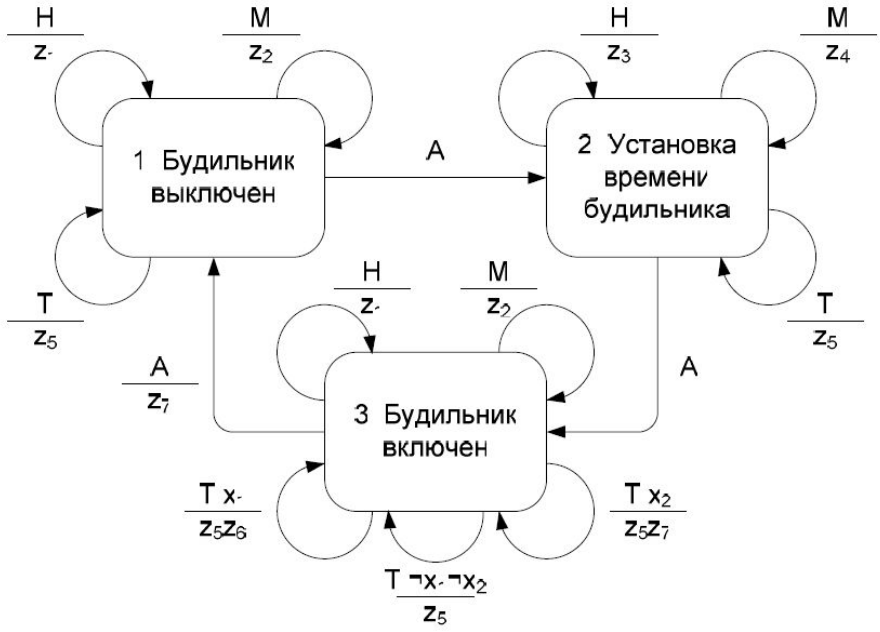


Рисунок 19 – Автомат управления часами с будильником

Начальное состояние автомата – «1 Будильник выключен».

2.3.2.1. Система тестовых примеров и *LTL*-формул

В систему тестов для построения автомата управления часами с будильником входят 38 тестов, они подробно описаны в работе [18], и поэтому в диссертации не приводятся. В настоящей работе они не были изменены для того, чтобы оценить, какой вклад дает верификация. В

таблице 6 приведены *LTL*-формулы, по которым выполняется верификация [19].

Таблица 6 – *LTL*-формулы часов с будильником

Формула	**Комментарий**
G(wasEvent(T) <=> wasFirstAction(z5))	Переход по событию *T* происходит тогда и только тогда, когда первым вызывается действие *z5*. В терминах модели: срабатывание таймера приводит к прибавлению минуты к текущему времени. Это может быть выполнено только по срабатыванию таймера.
G(wasEvent(H) <=> [wasAction(z1) or wasAction(z3)])	Переход по событию *H* происходит тогда и только тогда, когда вызываются действия *z1* или *z3*. В терминах модели: нажатие кнопки «H» приводит к увеличению текущего времени или времени будильника на один час. Это может быть выполнено только по нажатию кнопки «H».
!F(wasAction(z1) and wasAction(z3))	Неверно, что в будущем будет совершен переход и вызваны одновременно действия *z1* и *z3*. В терминах модели: нельзя одновременно увеличить время часов и время будильника на час.

Формула	Комментарий
G(wasEvent(M) <=> [wasAction(z2) or wasAction(co.z4)])	Переход по событию *M* происходит тогда и только тогда, когда вызываются действия *z2* или *z4*. В терминах модели: нажатие кнопки «M» приводит к увеличению времени часов или времени будильника на одну минуту. Это может быть выполнено только по нажатию кнопки «M».
!F(wasAction(z2) and wasAction(z4))	Неверно, что в будущем будет совершен переход и вызваны одновременно действия *z2* и *z4*. В терминах модели: нельзя одновременно увеличить время часов и время будильника на минуту.
G([wasEvent(A) and wasAction(z7)] => X(R[wasEvent(A), !wasAction(z3) and !wasAction(z4)]))	При переходе по событию *A* и вызове действия *z7* не могут вызываться действия *z3* и *z4*, пока не произойдет событие *A*. В терминах модели: если будильник был отключен, то нельзя выставлять время будильника, не нажав кнопку «A».

Формула	Комментарий
G([wasAction(z3) or wasAction(z4)] => X(R[wasEvent(A), !wasAction(z6) and !wasAction(z7)]))	При переходе и вызове действия *z3* или *z4* не будут вызваться действия ни *z6*, ни *z7* до тех пор, пока не будет обработано событие *A*. В терминах модели: если выставляется время будильника, то звонок не будет включен и не будет выключен, пока не будет нажата кнопка «А».
G([wasAction(z1) or wasAction(z2)] => X(R[wasEvent(A), !wasAction(z3) and !wasAction(z4)]))	При переходе и вызове действия *z1* или *z2* не могут быть вызваны действия *z3* и *z4* до перехода по событию *A*. В терминах модели: если выставляется время часов, то нельзя выставлять время будильника, не нажав кнопку «А».
G([wasAction(z3) or wasAction(z4)] => X(R[wasEvent(A), !wasAction(z1) and !wasAction(z2)]))	При переходе и вызове действия *z3* или *z4* не могут быть вызваны действия *z1* и *z2* до перехода по событию *A*. В терминах модели: если выставляется время будильника, то, не нажав «А», нельзя выставлять время часов.
!F(wasAction(z6) and wasAction(z7))	При любом переходе не могут быть вызваны действия *z6* и *z7* одновременно. В терминах модели: нельзя одновременно включить и отключить сигнал будильника.

Формула	Комментарий
F(!(wasEvent(A) and !wasAction(z7)))	Не верно, что произойдет переход по событию *A* и не будет вызвано действия *z7*. В терминах модели: кнопка «A» не может нажиматься бесконечно, не отключив сигнал будильника. Иначе, нажатие кнопки «A» рано или поздно отключит будильник.

2.3.2.2. Результаты экспериментов

Для сравнения различных способов построения автоматов было проведено три вида экспериментов. В первом случае автомат строился только на основе тестов. Во втором – на основе тестов и *LTL*-формул, когда верификация использовалась только в функции приспособленности и при операции мутации. В третьем – верификация использовалась на всех стадиях алгоритма генетического программирования (в функции приспособленности и в операциях скрещивания и мутации). Эксперимент каждого вида запускался по 1000 раз. В отличие от генерации автомата управления дверьми лифта, правильный автомат строился при каждом запуске, даже без использования верификации. Входные параметры каждого эксперимента были идентичны (таблица 7).

Таблица 7 – Параметры эксперимента построения автомата, управляющего часами с будильником

Параметр эксперимента	**Значение**
Размер начальной популяции	2000
Число состояний автоматов в начальном поколении	4
Ожидаемое число переходов	14
Доля особей, переходящих в следующее поколение. Остальные будут получены с помощью операции скрещивания	10 %
Число поколений до «малой» мутации	70
Число поколений до «большой» мутации	100
Вероятность мутации особи	1 %

В результате всех трех видов экспериментов получался автомат изоморфный построенному вручную с точностью до названия состояний (рисунок 20), в котором из начального (отмечено «жирным») достижимы только три состояния из четырех. Этот автомат изоморфен построенному вручную, если удалить недостижимое состояние.

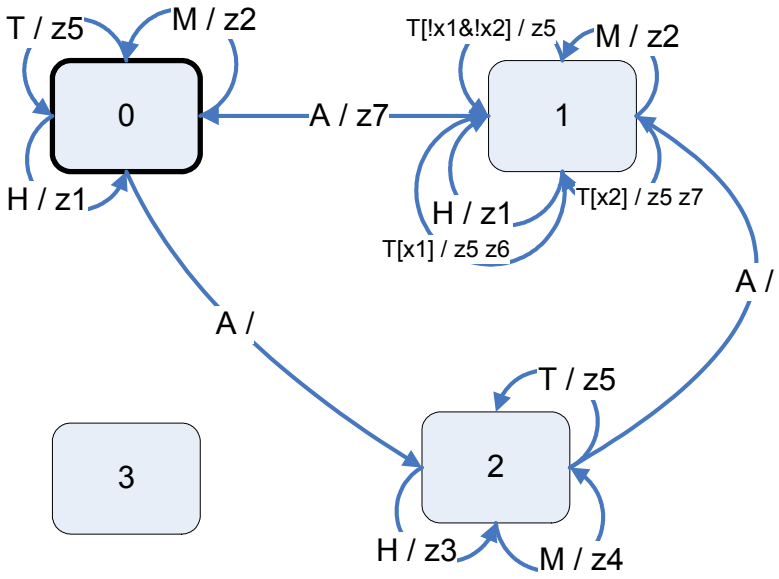


Рисунок 20 – Автомат, построенный с помощью алгоритма генетического программирования

Вклад тестов выбирался равным 100, а вклад *LTL*-формул – 10 (N = 100, M = 10, C = 100 из разд. 2.1.3). Вклад числа переходов достигал максимума 0.086 на особях, проходящих все тесты и формулы.

Как было отмечено выше, **первый** вид экспериментов состоял в построении автомата управления часами с будильником только на основе 38 тестов [52]. Значение функции приспособленности построенного автомата 100.086. В качестве оценки скорости работы алгоритма измерялось число вычислений функции приспособленности. Среднее значение вычислений функции приспособленности оказалось равным 1.45×10^6. Минимальное число вычислений – 2.561×10^5. Максимальное число – 9.24×10^6. Среднеквадратичное отклонение – 1.106×10^6. Плотность распределения вероятности числа вычислений функции приспособленности при построении автомата только на основе тестов представлена на рисунке 21.

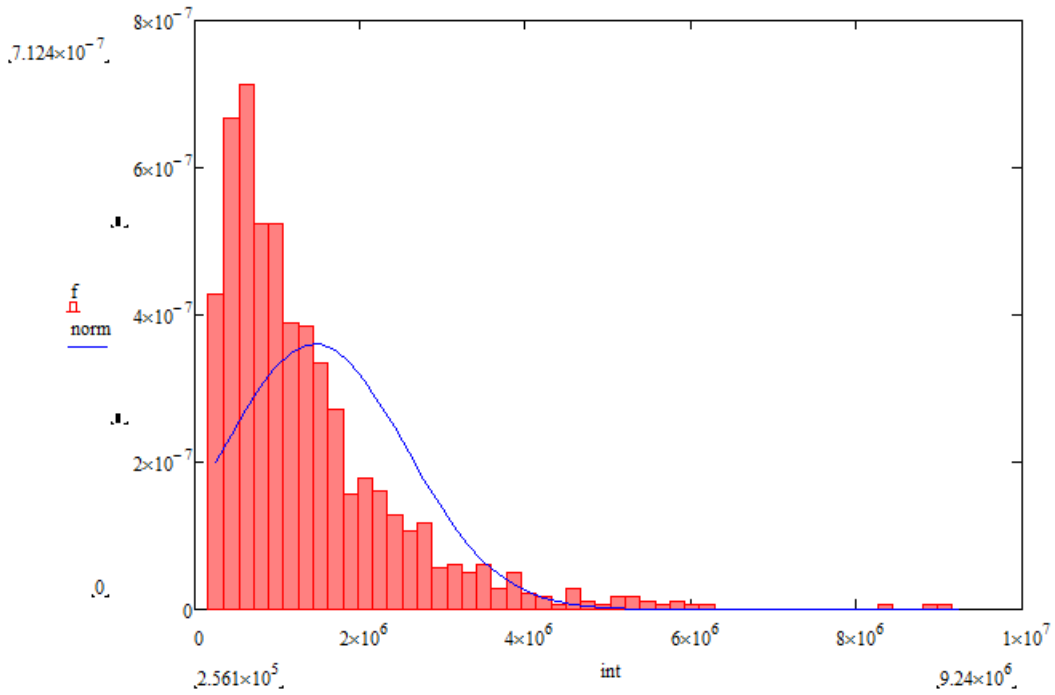


Рисунок 21 – Плотность распределения вероятности числа вычислений функции приспособленности при построении автомата только на основе тестов

Второй вид экспериментов заключался в построении автомата, управляющего часами с будильником на основе тестов и *LTL*-формул (разд. 2.3.2.1). При этом верификация учитывалась при вычислении функции приспособленности и при выполнении операции мутации. Вклад *LTL*-формул в функции приспособленности рассматривался как отношение выполнимых формул к общему числу формул, как это выполнялось до настоящей работы. Значение функции приспособленности построенного автомата – 110.086. Среднее значение вычислений функции приспособленности оказалось равным 2.425×10^6. Минимальное число вычислений – 2.182×10^5. Максимальное число – 1.949×10^7. Среднеквадратичное отклонение – 2.311×10^6. Плотность распределения вероятности числа вычислений функции приспособленности при построении автомата на основе тестов и *LTL*-формул представлена на рисунке 22.

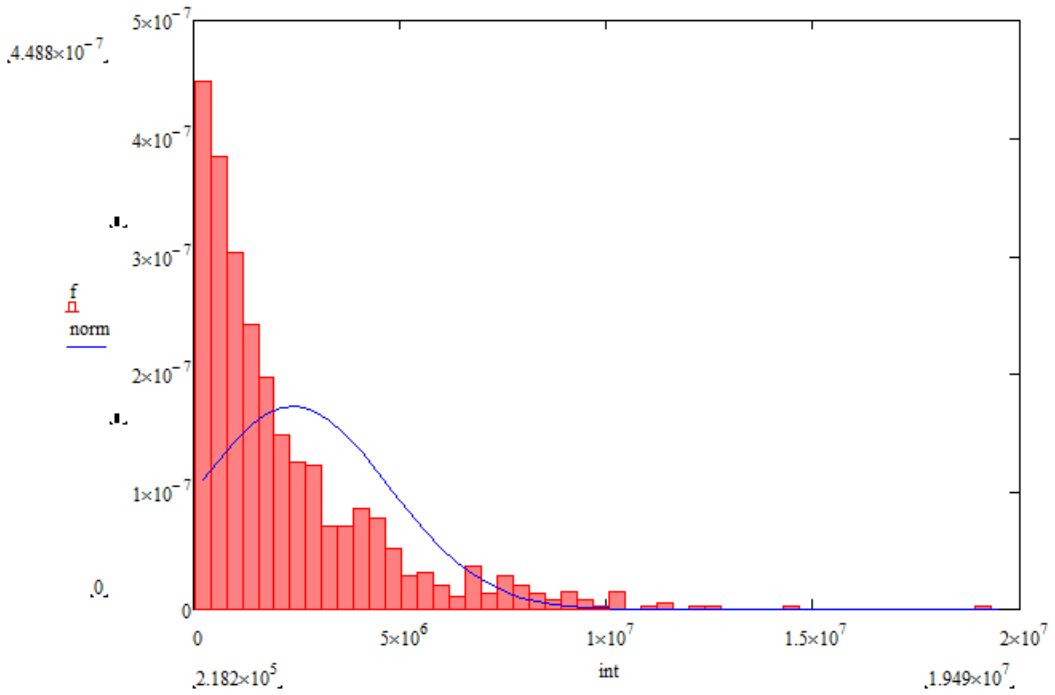


Рисунок 22 – Плотность распределения вероятности числа вычислений функции приспособленности на основе тестов и *LTL*-формул (верификация учитывалась в функции приспособленности и при операции мутации)

Третий вид экспериментов состоял в построении автомата, управляющего часами с будильником на основе тестов и *LTL*-формул (разд. 2.3.2.1), когда верификация учитывалась не только в функции приспособленности и при выполнении операции мутации, но и при скрещивании. Значение функции приспособленности построенного автомата – 110.086. Среднее значение вычислений функции приспособленности оказалось равным 1.832×10^6. Минимальное число вычислений – 2.038×10^5. Максимальное число – 1.301×10^7. Среднеквадратичное отклонение – 1.662×10^6. Плотность распределения вероятности числа вычислений функции приспособленности представлена на рисунке 23.

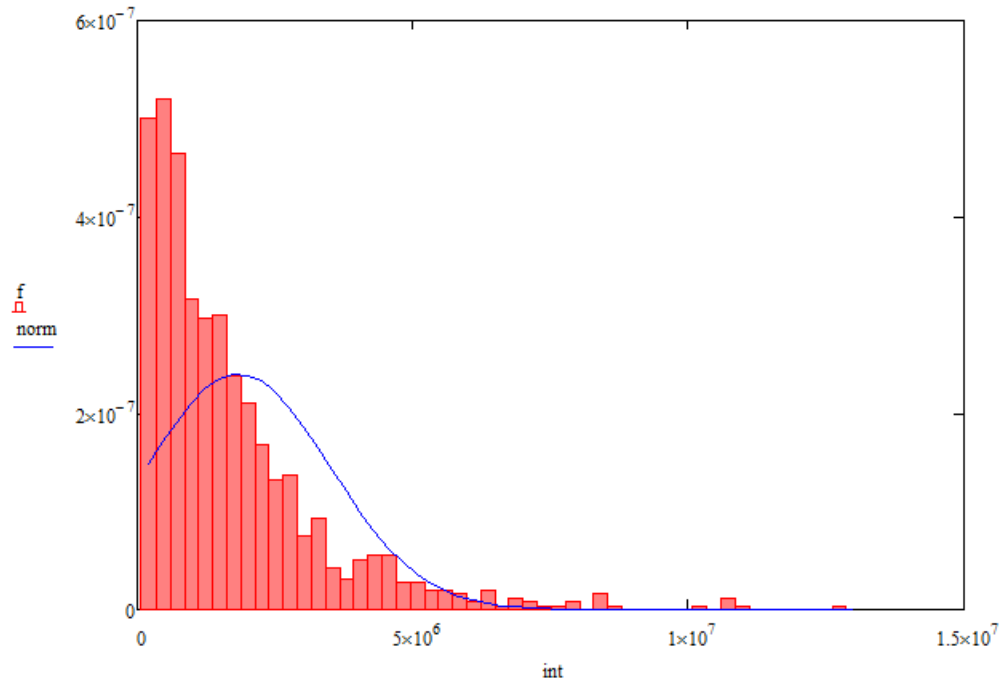


Рисунок 23 – Плотность распределения вероятности числа вычислений функции приспособленности на основе тестов и *LTL*-формул

Результаты всех экспериментов сведены в таблице 8.

Таблица 8 – Число вычислений функции приспособленности при генерации автомата управления часами с будильником

	Эксперименты		
	Первый вид	Второй вид	Третий вид
Среднее значение	1.45×10^6	2.425×10^6	1.832×10^6
Среднеквадратичное отклонение	1.106×10^6	2.311×10^6	1.662×10^6
Минимальное значение	2.561×10^5	2.182×10^5	2.038×10^5
Максимальное значение	9.24×10^6	1.949×10^7	1.301×10^7

Отметим, что число вычислений функции приспособленности для экспериментов с *LTL*-формулами (второй и третий виды экспериментов)

оказалось больше, чем без их использования. При сравнении второго и третьего видов экспериментов, преимущество имеет последний.

Замедление построения автомата при применении *LTL*-формул можно объяснить тем, что автомат, построенный только на основе тестов, является «правильным».

Поясним, почему использование верификации привело к увеличению числа вычислений функции приспособленности, хотя ожидалось обратное за счет учета верификации в функции приспособленности и в операциях скрещивания и мутации. В наборе *LTL*-формул встречается много формул вида $G(A => B)$. Они выполняются всегда, если выражение A никогда не встречается – вклад формулы этого вида единица. Следовательно, новая особь, для которой стало выполняться выражение A, но не выполняется B, стала хуже родителя, для которого A не выполнялось никогда. Такой автомат будет иметь меньшую функцию приспособленности, чем автомат из предыдущего поколения. Однако новый автомат может проходить те же тесты, что и родитель, и служить «прародителем» автомата, для которого формула снова станет выполняться.

Исходя из результатов экспериментов, можно сделать вывод, что в случае, когда тесты строят «правильный» автомат, использование верификации замедляет этот процесс. Однако можно уменьшать влияние *LTL*-формул за счет параметра M (разд. 2.1.3) в функции приспособленности, так как, используя только тесты, все равно нельзя быть уверенными, что автомат будет построен правильно.

В случае, когда только по тестам не удается построить автомат, одновременно проходящий все тесты и все *LTL*-формулы, имеется возможность его построения за счет изменения влияния формул (M) и тестов (N) в функции приспособленности.

Если автомат на основе тестов строится неправильным, то необходимо удалить «надбавку» за прохождение всех тестов. При этом

формулу для учета влияния тестов можно записать в виде: $FF_{test}=N \cdot FF_1$. Это не позволяет популяции оставаться локальном максимуме функции приспособленности, когда проходят все тесты, но формулы выполняются не все.

К сожалению, даже применение 38 тестов и 11 *LTL*-формул позволили построить автомат, соответствующий спецификации, но в нем содержалась ошибка. В работе [56] показано, что формула *G(wasEvent(z6) => F(wasEvent(z7))* – «после срабатывания будильника, он когда-нибудь будет выключен», не выполняется. Существует такая последовательность событий, при которой будильник будет звучать всегда. Для этого достаточно во время сигнала будильника перевести время. Это можно исправить путем отключения будильника (*z7*) при нажатой кнопке *M* или *H* в третьем состоянии автомата, или путем изменения значения входной переменной *x2,* которая будет сравнивать не время срабатывания будильника с текущим временем, а использовать независимый таймер.

Выводы по главе 2

1. Разработан алгоритм построения управляющих автоматов на основе генетического программирования по *LTL*-формулам и тестовым примерам. Основное отличие предлагаемого подхода от известных состоит в том, что верификация используется в функции приспособленности, операциях скрещивания и мутации.
2. Предложена функции приспособленности, в которой вклад каждой *LTL*-формулы не дискретен (ноль или единица), а может учитывать выполнение *LTL*-формулы на части автомата.
3. Предложена операция мутации особей генетического программирования, при которой учитывается контрпример, нарушающий *LTL*-формулу.

4. Разработана операция скрещивания особей генетического программирования, учитывающая результат прохождения *LTL*-формул. Эта операция сохраняет ту часть особи, для которой формула выполняется, и скрещивает случайным образом оставшуюся часть особи.
5. Впервые предложен алгоритм генерации автоматов с учетом контрактов. Описание системы при помощи контрактов проще, чем запись *LTL*-формул произвольного вида. Использование контрактов позволяет ускорить процесс построения автоматов, по сравнению с применением *LTL*-формул произвольного вида.
6. Рассмотрено расширение алгоритма построения автоматов с помощью генетического программирования с учетом сценариев, которые позволяют ускорить генерацию автоматов, по сравнению с применением тестов.
7. Предлагаемые алгоритмы исследованы при построении двух автоматов управления. На автомате управления дверьми лифта продемонстрировано, что только совместное использование верификации и тестовых примеров приводит к правильному автомату.

ГЛАВА 3. ТЕХНОЛОГИЯ ПОСТРОЕНИЯ АВТОМАТНЫХ ПРОГРАММ НА ОСНОВЕ ГЕНЕТИЧЕСКОГО ПРОГРАММИРОВАНИЯ И ВЕРИФИКАЦИИ

В настоящей главе сначала рассматривается традиционный подход к построению автоматных программ с их дальнейшей верификацией. Особенность такого подхода состоит в том, что если в программе будет обнаружена ошибка, то необходимо внесение изменений либо в спецификацию, либо в автомат, а затем повторная верификация.

Далее в работе рассматриваются два подхода к генерации автоматов: первый – по тестам, второй – по сценариям работы и контрактам с учетом верификации. На основе второго подхода автором созданы технология и соответствующее инструментальное средство для построения автоматов с помощью генетического программирования и верификации.

3.1. Верификация автоматных программ и верификатор *AutomataVerificator*

Рассмотрим традиционный подход к разработке и верификации автоматных программ, представленный на рисунке 24 в виде схемы. Для ее реализации используются:

- инструментальное средство *UniMod*, разработанное на кафедре «Компьютерные технологии» НИУ ИТМО, которое позволяет визуально разрабатывать автоматы и генерировать код, им соответствующий [6]. Это средство доступно по адресу http://unimod.sourceforge.net/;
- верификатор *AutomataVerificator* [64]., который позволяет верифицировать автоматы, утверждения о которых записаны на языке *LTL*. Это средство доступно по адресу http://code.google.com/p/automataverificator/.

Перейдем к описанию предложенной схемы.

Рисунок 24 – Схема, представляющая традиционное построение и верификацию автоматных программ

1. Собираются требования к будущей программе, и создается спецификация. Она описывает поведение программы и ее свойства, которые должны выполняться. Это требуется для того, что бы в дальнейшем была возможна проверка утверждений о программе. Иначе во

время верификации не понятно, какие свойства программы требуется проверять, какие из них должны выполняться, а какие нет.

2.1. При помощи инструментального средства *UniMod* разрабатывается автоматная программа: создаются поставщики событий, объекты управления и управляющий автомат.

2.2. Это средство позволяет сохранить модель (автомат) в *XML*-формате.

3.1. По спецификации, записанной в произвольной форме, выделяется словесное описание требований. Например, может быть выделено требование: «После возникновения аппаратной ошибки система отменит последнюю операцию».

3.2. Из требований выделяются атомарные высказывания (предикаты), соответствующие утверждениям о переходах и состояниях в модели. Например, требование из пункта 3.1 может быть записано в виде двух атомарных высказываний: «После события *p1.e1*» и «Рано или поздно будет вызвано действие *o1.z1*», где *p1.e1* – событие, посылаемое при аппаратной ошибке, а *o1.z1* – откат последней операции.

3.3. На основе атомарных высказываний могут быть записаны требования в виде *LTL*-формул. Если выразительная способность языка *LTL* не позволяет записать требования в виде *LTL*-формул, то они должны быть переформулированы или исключены из верификации.

4. После построения модели и *LTL*-формул, начинается работа верификатора.

4.1. Сначала верификатор преобразует *UniMod*-модель в свое внутренне представление.

4.2. По отрицанию каждой *LTL*-формулы строится автомат Бюхи.

4.3. Верификатор проверяет пустоту языка, допускаемого пересечением двух автоматов: автомата модели и автомата Бюхи, построенного по отрицанию *LTL*-формулы.

4.4. Если язык пуст, то верификатор подтверждает выполнение требования. Тогда считается, что модель удовлетворяет заявленной спецификации. При внесении изменений в модель требуется повторная верификация для проверки сохранения соответствия новой модели требованиям. После добавления новых требований в спецификацию требуется повторение цикла разработки программы.

4.5. Если язык не пуст, то верификатор выдает контрпример, показывающий нарушение требования.

5. Если требование нарушено, то, возможно, найдена ошибка в автомате. Тогда требуется внесение изменения в модель, ее преобразование во внутреннее представление верификатора, а затем повторная верификация. Если же контрпример не является ошибкой в модели, то или имеет место неправильная формулировка требований или не учтены особенности реализации поставщиков событий и объектов управления. Тогда необходимо устранить указанные проблемы и выполнить повторную верификацию.

Элементы схемы 2.1, 2.2 реализованы в инструментальном средстве *UniMod* [6], а элементы 4.1 – 4.5 реализованы в верификаторе *AutomataVerificator* [64]. Остальные элементы реализуются вручную.

3.2. Построение автоматных программ на основе генетического программирования по тестам

Рассмотрим подход к созданию автоматных программ на основе генетического программирования по тестам (рисунок 25).

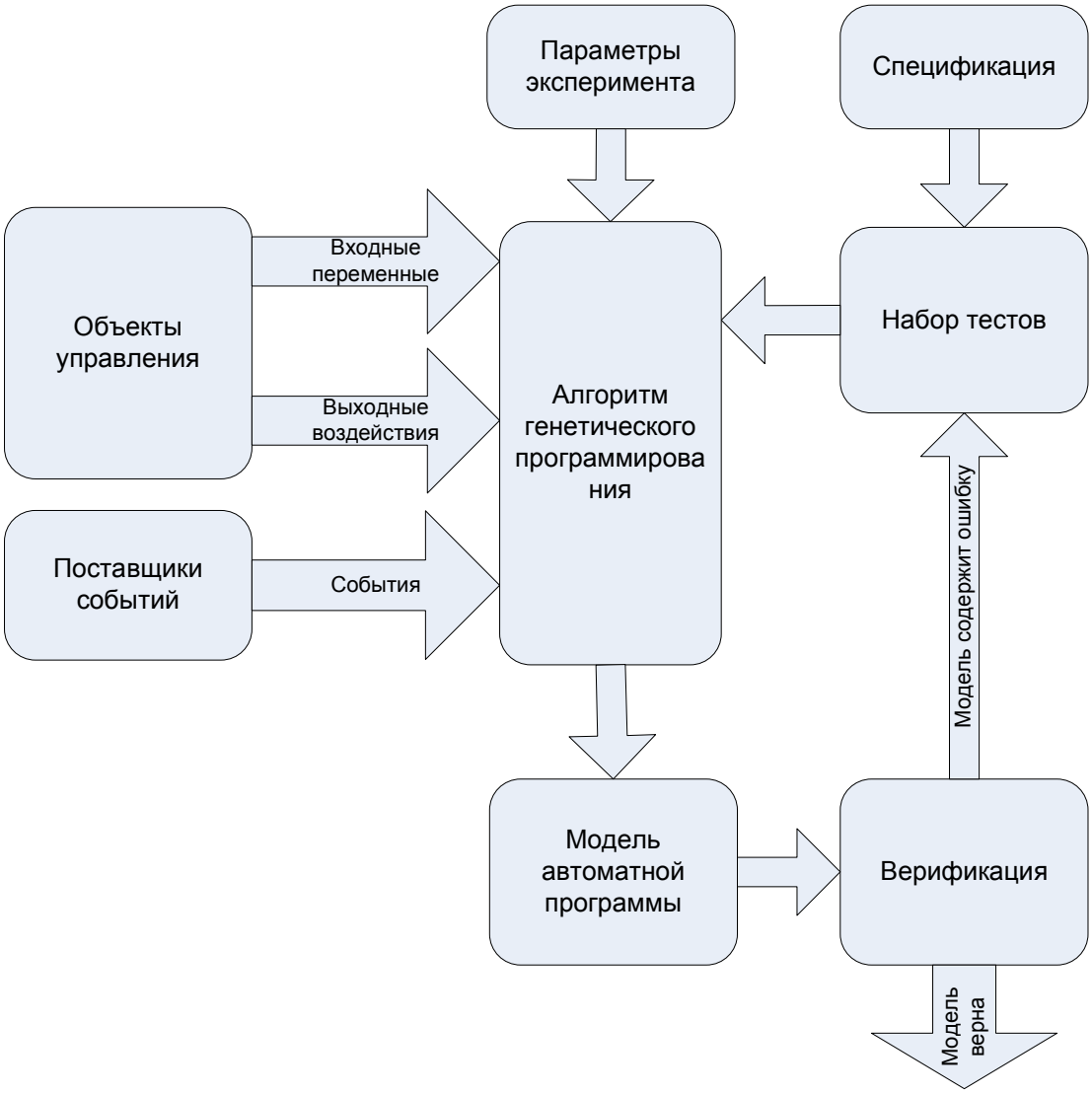


Рисунок 25 – Схема, представляющая построение автоматных программ на основе генетического программирования по тестам

На вход алгоритму генетического программирования подаются списки событий, входных переменных и выходных воздействий. Входные переменные и выходные воздействия поступают от «Объекта управления», который либо реализован заранее, либо известны его методы. События, которые обрабатываются моделью, поступают от «Поставщика событий». В общем случае можно использовать несколько объектов управления и несколько поставщиков событий.

Далее по спецификации программы или из других априорных знаний о ее поведении строится набор тестов. При их записи используются события и входные переменные (входные данные теста – *Input[i]*) и выходные воздействия (эталонная последовательность выходных данных – *Answer[i]*).

На вход алгоритму генетического программирования подаются параметры эксперимента, такие как размер популяции, вероятность мутации, число поколений до «малой» и «большой» мутаций и т. д.

Далее алгоритм генетического программирования строит автоматную модель. При построении автомата на основе моделирования, для каждой задачи приходится разрабатывать свой алгоритм генетического программирования – в частности, строить функцию приспособленности, учитывающую специфику задачи. При генерации автоматов рассматриваемым способом, алгоритм генетического программирования одинаков для любых задач – для каждой новой задачи необходим только новый набор тестов, входных и выходных воздействий и, возможно, параметры алгоритма. Построенная модель проходит все тесты, однако нельзя быть уверенными в ее правильности, поэтому необходима дополнительная верификация.

Если модель оказалась верной, то ее можно использовать в реальной системе. Однако если верификатор обнаружил несоответствие модели и спецификации, то необходимо добавлять новые тесты во входной набор и выполнять построение заново. В худшем случае, корректная модель никогда не будет построена, так как нет гарантии, что такой цикл разработки когда-нибудь завершится.

Заметим, что можно вручную модифицировать модель для того, чтобы она соответствовала спецификации, однако это может оказаться сложнее, чем создание модели вручную «с нуля».

3.3. Технология создания автоматных программ на основе генетического программирования и верификации

В настоящей работе предлагается использовать верификацию при генерации автомата модели. На рисунке 26 представлена схема, соответствующая предложенной технологии.

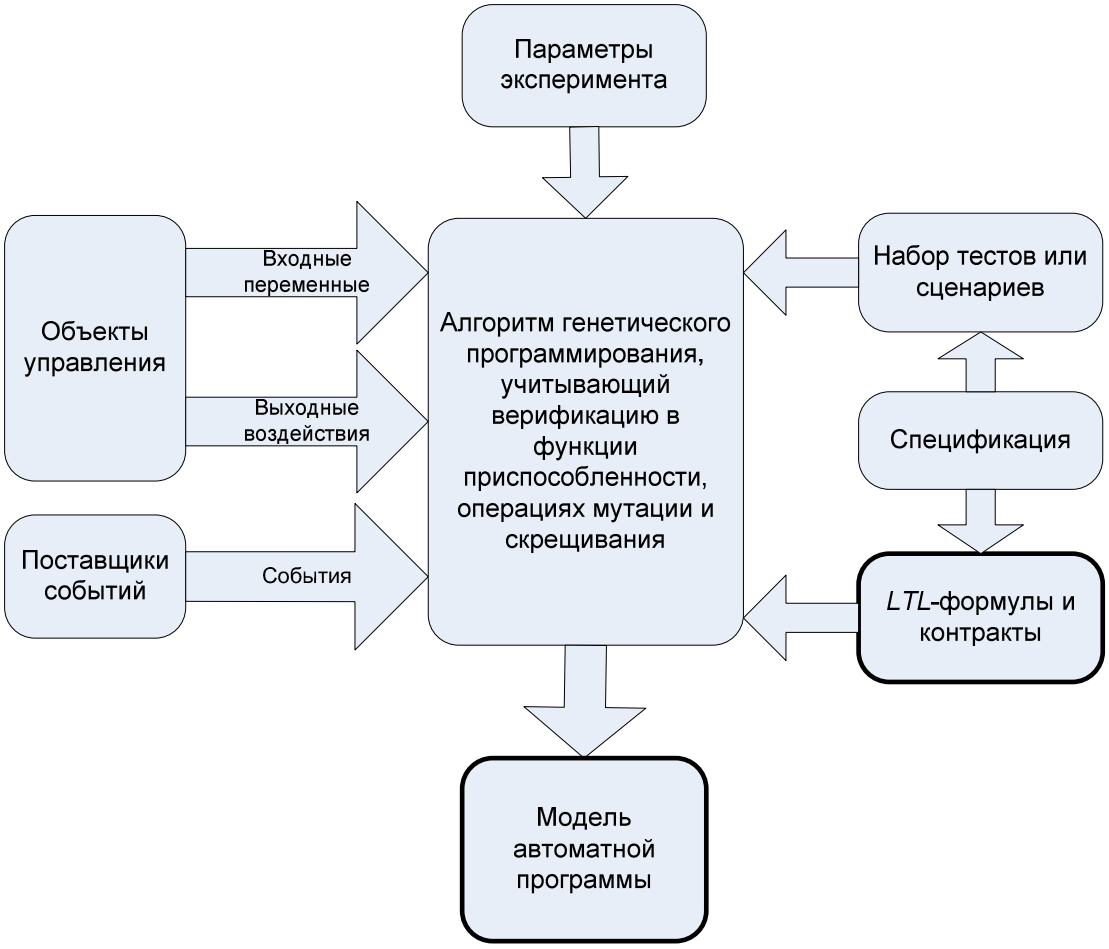


Рисунок 26 – Схема, соответствующая технологии создания автоматных программ на основе генетического программирования и верификации

В отличие от подхода, основанного на тестах, входными данными в предлагаемой технологии являются *LTL*-формулы и контракты. Они формулируются на основе спецификации. Если автомат может быть построен, то он не требует дополнительной верификации, так как каждая особь в каждом поколении верифицируется, и в качестве результата работы

алгоритма выбирается та, у которой функция приспособленности максимальна. Это означает, что автомат проходит все тесты и удовлетворяет всем *LTL*-формулам – модель соответствует спецификации, и дополнительная проверка не требуются.

Таким образом, если набор тестов и набор *LTL*-формул непротиворечивы, то автоматная модель будет построена, и она будет соответствовать заданной спецификации. При необходимости внесений изменений в требования необходимо их внести в набор тестов и в *LTL*-формулы, после этого, если автомат будет построен, то он снова соответствует спецификации и не требует дополнительной проверки.

Перейдем к описанию алгоритма генетического программирования. В работе использовался классический вариант алгоритма со стратегий эллитизма: в каждом поколении часть особей с наибольшей функцией приспособленности переходит в новое поколение без изменений. Остальные особи подвергаются операциям скрещивания и мутации, в том числе и предлагаемыми в диссертации. На рисунке 27 представлена схема работы алгоритма генетического программирования.

Рисунок 27 – Схема работы алгоритма генетического программирования

Алгоритм генетического программирования случайным образом генерирует начальное поколение. Затем для каждой особи выполняется алгоритм расстановки пометок выходных воздействий. После того, как для каждого перехода известны его выходные воздействия, выполняется верификация *LTL*-формул. Затем вычисляется функция приспособленности, в которой учитываются как тесты (сценарии), так и *LTL*-формулы. Если на какой-нибудь особи достигнуто требуемое значение функции приспособленности, то алгоритм заканчивается и данная особь считается результатом работы алгоритма генетического программирования. Если же требуемое значение не достигнуто, то в новое поколение переходят лучшие особи без изменений, и для всего поколения производятся операции скрещивания и мутации. После этого шаги алгоритма повторяются с расстановки пометок выходных воздействий.

3.4. Инструментальное средство для поддержки технологии генерации автоматов на основе генетического программирования и верификации

Предлагаемая технология реализована на языке *Java* и является дополнением к реализации *GABP* (*Genetic Automata-Based Programming*) из работы [52]. При этом были добавлены два пакета: первый – для чтения тестов, сценариев, *LTL*-формул и контрактов, второй – для проведения верификации. Диаграмма пакетов приведена на рисунке 28.

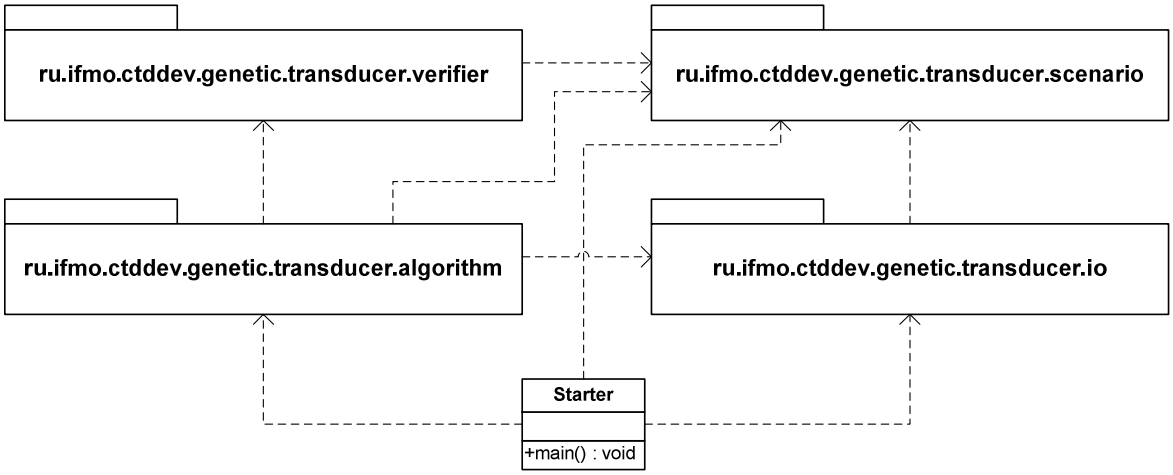


Рисунок 28 – Диаграмма пакетов

В пакет `ru.ifmo.ctddev.genetic.transducer.scenario` добавлен класс `TestGroup`, который хранит набор тестов (сценариев) и *LTL*-формул для конкретной группы. Теперь алгоритм генетического программирования на вход получает не список тестов, а список объектов данного класса.

Класс `FitnessCalculator` реализует вычисление функции приспособленности на основе редакционного расстояния и относительного числа верифицированных переходов. Приведем код метода вычисления функции приспособленности:

```
public double calcFitness(FST fst) {
    int transitionCount = fst.getUsedTransitionsCount();
    if (transitionCount == 0) {
        return 0.0;
    }
```

```
fst.unmarkAllTransitions();
fst.doLabelling(this.tests);

verifier.configureStateMachine(fst);
int[] verRes = verifier.verify();

double res = 0; //fitness
int i = 0;      //group number
for (TestGroup group : groups) {
    double pSum = 0;   //positive tests sum
    int nSum = 0;      //negative tests sum
    int cntOk = 0;

    for (Path p : group.getTests()) {
        String[] output = fst.transform(p.getInput());
        String[] answer = p.getOutput();
        double t;
        if (output == null) {
            t = 1;
        } else {
            t = editDistance(output, answer)
                 / Math.max(output.length, answer.length);
        }
        pSum += 1 - t;
        if (same(output, answer)) {
            cntOk++;
        }
    }

    for (Path p : group.getNegativeTests()) {
        if (!fst.validateNegativeTest(p.getInput())) {
            nSum++;
        }
    }

    int testsSize = group.getTests().size();
    int negativeTestsSize =
            group.getNegativeTests().size();
    int formulasSize = group.getFormulas().size();

    double testsFF = TESTS_COST;
    if (testsSize > 0) {
        testsFF = (cntOk == testsSize)
            ? TESTS_COST
            : TESTS_COST * (pSum / testsSize);
    }
    double negativeTestsFF = (negativeTestsSize != 0)
            ? (nSum * 1.0) / negativeTestsSize
            : 0;
    double ltlFF = FORMULAS_COST;
    if (formulasSize > 0) {
```

```
                ltlFF = FORMULAS_COST * verRes[i] / formulasSize
                        / transitionCount;
            }
            res += testsFF + testsFF * ltlFF
                    - negativeTestsFF * testsFF;
            i++;
        }
        return res + 0.0001 * (100 - fst.getTransitionsCount());
    }
```

Сначала выполняется алгоритм расстановки пометок выходных воздействий [18, 52], после этого полученный автомат с проставленными воздействиями верифицируется, определяется число успешно «пройденных» формул для каждой группы. Также при верификации помечаются переходы, принадлежащие контрпримерам, если они обнаруживаются. После этого вычисляется редакционное расстояние для тестов в каждой группе, суммируются вклады тестов (сценариев) и *LTL*-формул (контрактов).

Класс `FST`, реализующий автомат Мили, хранит номер начального состояния, число состояний, набор переходов для каждого состояния, набор входных событий и набор выходных воздействий:

```
private final int initialState;
private final int stateNumber;
private Transition[][] states;
private final String[] setOfInputs;
private final String[] setOfOutputs;
```

В этом классе также реализованы операции мутации, скрещивания и алгоритм расстановки пометок. Их реализацию приведена в работе [52]. В настоящей работе в данный класс было добавлено удаление перехода из контрпримера во время мутации и скрещивания, учитывающее результат верификации. Приведем часть кода, отвечающую за скрещивание с учетом верификации:

```
public FST[] crossOverBasedOnLtl(FST that) {
    return crossOverByMarked(that, new ITransitionChecker() {
        public boolean isMarked(Transition t) {
            return t.isVerified() && !t.isUsedByVerifier();
        }
```

```
    });
}

public FST[] crossOverByMarked(FST that,
                               ITransitionChecker check) {
    int initialState1;
    int initialState2;

    if (RANDOM.nextBoolean()) {
        initialState1 = this.initialState;
        initialState2 = that.initialState;
    } else {
        initialState1 = that.initialState;
        initialState2 = this.initialState;
    }

    Transition[][] states1 = new Transition[stateNumber][];
    Transition[][] states2 = new Transition[stateNumber][];

    for (int i = 0; i < stateNumber; i++) {
        int degree1 = this.states[i].length;
        int degree2 = that.states[i].length;

        states1[i] = new Transition[degree1];
        states2[i] = new Transition[degree2];
        int p1 = 0;
        int p2 = 0;

        ArrayList<Transition> list =
                new ArrayList<Transition>();
        for (Transition t : this.states[i]) {
            if (check.isMarked(t)) {
                states1[i][p1++] = t.copy(setOfInputs,
                        setOfOutputs, stateNumber);
            } else {
                list.add(t);
            }
        }
        for (Transition t : that.states[i]) {
            if (check.isMarked(t)) {
                states2[i][p2++] = t.copy(setOfInputs,
                        setOfOutputs, stateNumber);
            } else {
                list.add(t);
            }
        }
        Collections.shuffle(list);

        degree1 -= p1;
        degree2 -= p2;
```

```
            for (int j = 0; j < degree1; j++) {
                states1[i][p1++] = list.get(j)
                    .copy(setOfInputs, setOfOutputs, stateNumber);
            }
            for (int j = 0; j < degree2; j++) {
                states2[i][p2++] = list.get(degree1 + j)
                    .copy(setOfInputs, setOfOutputs, stateNumber);
            }
            states1[i] = removeDuplicates(states1[i]);
            states2[i] = removeDuplicates(states2[i]);
        }

        return new FST[] {
                new FST(states1, initialState1,
                        setOfInputs, setOfOutputs),
                new FST(states2, initialState2,
                        setOfInputs, setOfOutputs)
        };
    }
```

Класс `Transition` представляет собой переход в автомате, он хранит входное воздействие, число выходных воздействий и номер состояния, в которое перейдет автомат по этому переходу. Класс представляет набор методов, но в данной работе интересен тот, который копирует переход в новую особь при скрещивании. Он реализован, как описано в разд. 2.1.5:

```
public Transition copy(String[] setOfInputs,
                       String[] setOfOutputs,
                       int stateNumber) {
    if (isUsedByVerifier()
            && !used
            && (RANDOM.nextDouble() < 0.1)) {
        return new Transition(
            setOfInputs[RANDOM.nextInt(setOfInputs.length)],
            mutateOutputSize(outputSize, setOfOutputs.length),
            RANDOM.nextInt(stateNumber));
    }
    return new Transition(input, outputSize, newState);
}
```

В пакете `ru.ifmo.ctddev.genetic.transducer.io` размещены классы для чтения тестов (сценариев) и *LTL*-формул (контрактов) из *XML*-файла и записи модели в файл в формате *UniMod* [6].

Класс `TestsReader` позволяет читать из *XML*-файла параметры алгоритма (вероятность мутации, размер популяции, ожидаемое число

состояний, число поколений до мутации и т. д.), набор тестов и *LTL*-формул, разбитых по группам. Класс `OneGrouptestsReader` позволяет читать входные данные для метода, игнорируя разбивку по группам (создавая всего одну группу).

Класс `UnimodModelWriter` сохраняет особь в файл в *XML*-формате *UniMod*-модели. В результате, созданную автоматную модель можно сразу запускать в инструментальном средстве *UniMod* при условии, что реализованы поставщики событий и объекты управления.

В пакете `ru.ifmo.ctddev.genetic.transducer.verifier` размещены классы, относящиеся к верификации: трансляция формул в автомат Бюхи, преобразование особи генетического программирования в объект-автомат, принимаемый верификатором, и вызов процесса верификации.

Класс `ModifiableAutomataContext` является представлением верифицируемой автоматной программы. Он содержит объект управления, поставщика событий и автомат модели. Его особенность в том, что автомат может изменяться, так как у каждой особи он свой, в то время как поставщик событий и объект управления остаются неизменными, создаются один раз за время генерации автомата. Приведем поля из данного класса, они являются интерфейсами, описанными в используемом верификаторе:

```
private IControlledObject co;
private IEventProvider ep;
private IStateMachine<IState> machine;
```

Класс `VerifierFactory` предоставляет методы для трансляции формул в автомат Бюхи, для представления особи в виде автомата, принимаемого используемым верификатором, и метод для верификации. Трансляция формул происходит один раз при запуске программы, так как они не изменяются в процессе работы алгоритма генетического программирования. Следовательно, их можно преобразовывать в автомат

Бюхи только один раз. После верификации модели соответствующий метод возвращает суммарное число верифицированных переходов для каждой группы и помечает переходы в автомате, являющиеся контрпримерами для формул:

```
void prepareFormulas(TestGroup[] groups);
void configureStateMachine(FST fst);
int[] verify();
```

Вызов процесса верификации предполагает, что автомат `FST` был преобразован в модель, принимаемую на вход верификатором. Для этого необходимо вызвать метод `void configureStateMachine(FST fst)`. После этого автомат верифицируется методом `int[] verify()`, который возвращает суммарное число верных переходов для каждой группы. Приведем код данного метода:

```
public int[] verify() {
    int[] res = new int[preparedFormulas.length];
    IVerifier<IState> verifier = new SimpleVerifier<IState>(
            context.getStateMachine(null).getInitialState());
    int usedTransitions = fst.getUsedTransitionsCount();

    for (int i = 0; i < preparedFormulas.length; i++) {
        int marked = 0;
        int length = preparedFormulas[i].length;
        int barrier = Math.max(Const.MIN_USED_TESTS,
                               length / 10);

        for (IBuchiAutomata buchi : preparedFormulas[i]) {
            counter.resetCounter();
            List<IIntersectionTransition> list;
            if (RANDOM.nextInt(length) < barrier) {
                list = verifier.verify(buchi, predicates,
                                       marker, counter);
            } else {
                list = verifier.verify(buchi, predicates,
                                       counter);
            }
            if ((list != null) && (!list.isEmpty())) {
                ListIterator<IIntersectionTransition> iter =
                        list.listIterator(list.size());
                int failTransitions = buchi.size() - 1;

                for (int j = 0; iter.hasPrevious()
                     && (j < failTransitions);) {
```

```
                    IIntersectionTransition t =
                            iter.previous();
                    if ((t.getTransition() != null)
                            && (t.getTransition().getClass()
                            == AutomataTransition.class)) {
                        AutomataTransition trans =
                                t.getTransition();
                        if (trans.getAlgTransition() != null){
                            trans.getAlgTransition()
                                .setUsedByVerifier(true);
                            j++;
                        }
                    }
                }
                marked += counter.countVerified() / 2;
            } else {
                marked += usedTransitions;
            }
        }
        res[i] = marked;
    }
    return res;
}
```

Класс `AutomataTransition` является представлением перехода для верификатора. Также данный класс сохраняет ссылку на переход в особи для того, чтобы по контрпримеру верификатора совершить обратное преобразование в переход модели.

3.4.1. Использование инструментального средства

Инструментальное средство компилируется и упаковывается в один *jar*-файл. Следующая команда используется для единичного построения автомата:

```
$ java -jar gabp.jar experiment.xml
```

Файл *experiment.xml* содержит параметры алгоритма генетического программирования, тесты или сценарии, *LTL*-формулы и контракты. Пример такого файла приведен в приложении 1.

Программа пишет в консоль минимальную и максимальную функцию приспособленности для каждого из поколений. Каждые 200 поколений алгоритма генетического программирования программа

выводит лучший результат в файл *genN.xml*, где *N* – номер поколения. После построения особи, проходящей все тесты или сценарии, удовлетворяющей всем *LTL*-формулам и имеющей ожидаемое число переходов, программа сохраняет ее в файле *best.xml*. Формат выходных файлов инструментального средства совпадает с форматом *UniMod* [6]. Пример такого файла приведен в приложении 2.

Исходный код программного средства размещен в открытом доступе по адресу: http://code.google.com/p/gabp/.

Выводы по главе 3

1. Рассмотрен традиционный подход к созданию автоматных программ и верификации управляющих автоматов.
2. Рассмотрено сравнение построений автоматных программ только по тестовым примерам и совместное применение тестов или сценариев и верификации. Показано, почему применение верификации приводит к построению автомата с заранее заданным поведением, в отличие от использования только тестов.
3. Приведено описание основных частей разработанного инструментального средства *GABP* для автоматического построения автоматов. Описаны детали реализации использования верификации в алгоритме генетического программирования.

ГЛАВА 4. ВНЕДРЕНИЕ РЕЗУЛЬТАТОВ РАБОТЫ ПРИ ПОСТРОЕНИИ МОДУЛЯ ДЛЯ ОПРЕДЕЛЕНИЯ УЗЛОВ СЕТИ С МЕКСИМАЛЬНЫМ ТРАФИКОМ

Настоящий раздел посвящен внедрению результатов диссертационной работы в программный продукт, выпускаемом ООО «ЭВЕЛОПЕРС», на примере создания модуля для определения узлов сети с максимальным трафиком.

NetFlow – сетевой протокол, предназначенный для учета сетевого трафика [59, 61, 62, 63]. Фактически *NetFlow*-сообщение представляет собой метаинформацию о потоках в сети. Например, *NetFlow*-сообщение может содержать входящий *IP*-адрес, входящий порт, исходящий *IP*-адрес, исходящий порт, число переданных байт, число пакетов и т. д. Существуют несколько версий *NetFlow* протокола. Наиболее распространенные на данный момент: *версия 5* [59], *версия 9* [61] (далее *nfv5* и *nfv9*) и *IPFIX* (*Internet Protocol Flow Information eXport*) [62, 63]. В пятой версии протокола поля фиксированы, а в девятой – поля определяются шаблоном и могут настраиваться. Шаблон *NetFlow* определяет порядок полей, их тип, длину и позицию в сообщении. *NetFlow* представляет собой бинарный формат, который воспринимается далеко не всеми системами.

NetFlow-сообщения могут посылать различные сетевые устройства, такие как: маршрутизаторы, роутеры, межсетевые экраны и т. п. Далее все такие устройства будем объединять названием – *NetFlow*-экспортер или экспортер. *NetFlow*-экспортеры, поддерживающие протокол *nfv9*, позволяют выбрать один из стандартных шаблонов или создать свой. Это позволяет, в отличие от протокола *nfv5*, вводить новые поля и уменьшить размер *NetFlow*-сообщения за счет посылки только необходимых полей.

Syslog – стандарт отправки и регистрации сообщений о происходящих в системе событиях (то есть создания логов) [60]. Каждое

syslog-сообщение имеет приоритет, заголовок и текстовое тело сообщения. В отличие от бинарного формата *NetFlow*, *syslog* поддерживается различными системами и может быть прочтен человеком.

Рассматриваемый программный продукт предназначен для мониторинга сети и поиска потенциальных сетевых атак или угроз. Данный продукт позволяет обрабатывать сотни тысяч *NetFlow*-сообщений в секунду, агрегировать их и конвертировать в *syslog*-сообщения. Основная особенность указанного приложения состоит в том, что входящих *NetFlow*-сообщений много, а исходящих *syslog*-сообщений мало. При этом *NetFlow*-сообщения не сохраняются на диск, а обрабатываются на лету.

За счет формата выходных данных в виде *syslog*-сообщений, рассматриваемый программных продукт совместим с большинством *SIEM*-системами (*Security Information and Event Management*), – системами мониторинга, управления, анализа и хранения информации и событий информационной безопасности. За счет агрегации и исключения лишних *NetFlow*-сообщений рассматриваемое приложение позволяет сократить объем обрабатываемых и хранящихся данных в *SIEM*-системах.

Данный продукт позволяет подключать модули с различными правилами обработки *NetFlow*-сообщений. Автором настоящей работы предложена технология автоматического создания таких правил по сценариям работы и *LTL*-формулам. Для этого необходимо не только создать модель автоматной программы, но и конвертировать ее в граф потока управления и данных, используемый для описания модуля.

4.1. Принцип работы приложения

Приложение состоит из двух основных частей: сервер и контроллер. Обе компоненты общаются между собой по протоколу *TCP/IP* и могут располагаться на различных машинах.

Контроллер представляет собой приложение, которое позволяет настраивать сервер и управлять им посредством посылки таких команд как: включить/выключить модуль, добавить/удалить исходящий *syslog* сервер и т. п. Контроллер имеет две реализации: консольный вариант и веб-приложение, позволяющее настраивать сервер через интернет-браузер.

Как отмечалось ранее, основная задача приложения состоит в том, что бы преобразовывать *NetFlow*-сообщения в *syslog*. Этой задачей занимается компонента «сервер». Сервер принимает на вход *NetFlow*-сообщения по *UDP* протоколу от одного или нескольких *NetFlow*-экспортеров. Подсистема обработки входящих сообщений обрабатывает *NetFlow*-сообщение и передает сообщение правилам (*rule modules*) и конвертерам (*converter modules*), которые представляют собой динамически подключаемые библиотеки. Существует отдельный тип временных правил (*kron rules*), которые срабатывают по таймеру и обрабатывают данные, накопленные обычными правилами.

Каждое правило получает на вход *NetFlow*-сообщения, а на выход выдает одно или несколько тоже *NetFlow*-сообщений, но, как правило, определенных внутренними шаблонами. Эти сообщения попадают на вход другим правилам или конвертерам. Конвертер по одному *NetFlow*-сообщению форматирует один *syslog*. *Syslog*-сообщения попадают в очередь отправки сообщений, из которой они забираются подсистемой отправки сообщений.

4.1.1. Правила обработки *NetFlow*-сообщений

Как было сказано выше, конвертация *NetFlow*-сообщений в *syslog*-сообщения определяется правилами (*rules*) и конвертерами (*converters*). Конвертер реализуется достаточно просто: он переводит бинарное *NetFlow*-сообщение определенного шаблона в *syslog*-сообщение. Например, формат *syslog*-сообщения может быть «*ключ = значение*» для

каждого из полей. Тогда сообщение может выглядеть как следующая строка:

«*May 27 19:45:00 src_ip=192.168.63.112 src_port=80 bytes_in=4562*», в то время как *NetFlow*-сообщение представляет собой последовательность байт, где тело сообщения будет включать в себя четыре байта для входящего *IP*-адреса, два байта для порта и четыре байта для поля *bytes_in*.

Для настоящей работы интерес представляют правила обработки и конвертации *NetFlow*-сообщений. Каждое правило обрабатывает сообщения только определенных шаблонов, может накапливать, агрегировать, фильтровать и т. п. Правила бывают двух видов: *content-base* и *kron-base*. Первые принимают сообщения и их накапливают, вторые по таймеру берут накопленные сообщения и выдают на выход одно или несколько сообщений согласно реализованной в них логике. Существуют следующие основные элементы правил:

- входящий шаблон *NetFlow*-сообщения;
- исходящий шаблон *NetFlow*-сообщения;
- *mapper* – правила сопоставления полей шаблонов входящего *NetFlow*-сообщения и внутреннего;
- *collector* – структура для хранения и поиска *NetFlow*-сообщений по ключу.

Каждое *content-base* правило имеет один или несколько входящих шаблонов *NetFlow*-сообщений или же не имеет их вовсе. Как отмечалось ранее, шаблон точно определяет позицию каждого поля в *NetFlow*-сообщении, но в случае, когда шаблоны явно не заданы, поиск необходимых полей может осуществляться «на лету», если входящее сообщение имеет необходимые поля.

Сообщения, передаваемые внутри правил, а также между ними и конвертерами, имеют заданные внутренние шаблоны, которые однозначно

определяют позиции полей в *NetFlow*-сообщении и обеспечивают их быстрый поиск.

Mapper реализует функцию отображения полей входящих шаблонов на внутренний шаблон. Он извлекает поля из входящего *NetFlow*-сообщения и выдает сообщение, определенное внутренним шаблоном.

Collector сохраняет в памяти *NetFlow*-сообщения по простому или составному ключу, где в качестве простого ключа обычно выступает одно из полей, а в качестве составного – вектор из нескольких полей. При появлении сообщения с существующим ключом коллектор может выполнять различные операции над полями хранимого сообщения (сложение, вычитание и т. п.). Например, ключом коллектора может быть входящий *IP*-адрес (*src_ip*), а значением – число принятых байт (*bytes_in*). Тогда, если определена операция суммирования для поля *bytes_in*, то в любой момент времени можно будет узнать число принятых байт для определенного *IP*-адреса.

Kron-base политики не имеют входящих сообщений, они вызываются с определенным интервалом и обращаются к коллекторам *content-base* политик. Если вернуться к предыдущему примеру, где коллектор суммирует входящий трафик (*bytes_in*), то *kron-base* политика может вызываться раз в 30 секунд и забирать из коллектора по 10 *IP*-адресов с максимальным входящим трафиком и очищать коллектор. Таким образом, приведенный модуль состоит из одного *content-base* правила и одного *kron-base* правила и выдает каждые 30 секунд по 10 адресов с максимальным входящим трафиком (рисунок 29).

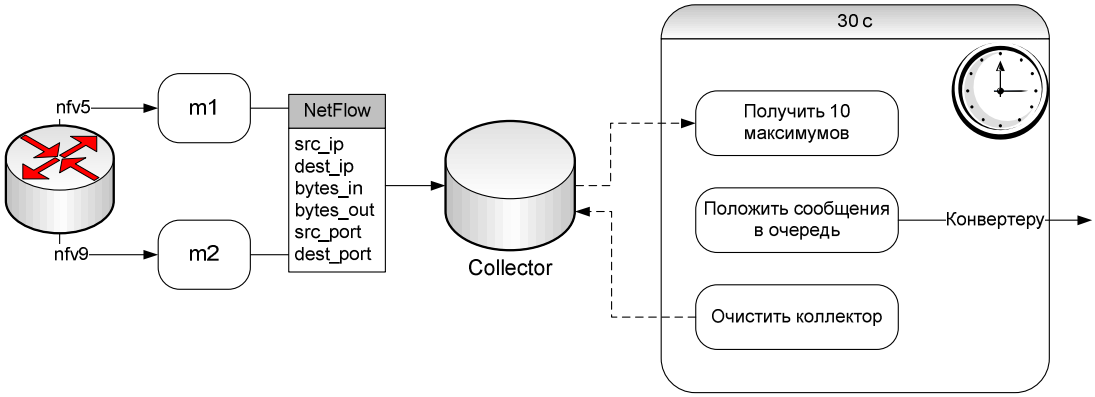


Рисунок 29 – Модуль: Top host traffic monitor

4.1.2. Граф представления модуля

Модули можно разрабатывать «как придется», а можно строить граф, основанный на элементах из разд. 4.1.1, аналогично традиционному и автоматному подходам к созданию программ.

Модуль может быть описан графом потока управления и данных (*control and data flow graph, CDFG*). Этот граф, как и традиционный, состоит из узлов и ребер. Узлы выполняют некие операции над *NetFlow*-сообщениями, а ребра показывают последовательность выполнения и зависимости данных. Узлы бывают следующих типов: *template, mapper, collector, condition, kron*, а ребра двух: *control flow* и *data flow*.

Ребра *control flow* показывают последовательность выполнения, а ребра *data flow* показывают, что выходные данные одного узла являются входными данными для другого. На рисунке 30 приведен пример части графа. Сплошными стрелками обозначены *control flow* ребра, а пунктирными *data flow*. В приведенном примере управление передается сначала *m1*, затем *c1*, затем *c2*. При этом данные для *c2* берутся из *m1*, а для *k1* – из *c2*.

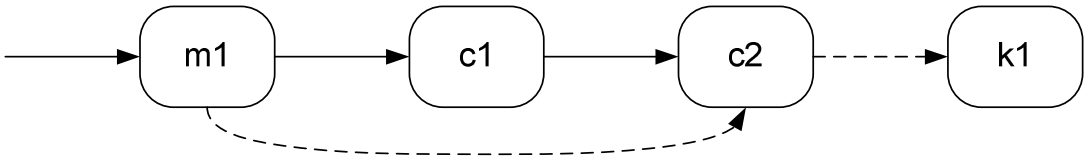


Рисунок 30 – Пример графа потока управления и данных

Каждый из узлов имеет атрибуты: список полей для шаблонов, правила сопоставления полей шаблонов для *mapper* и *collector*, простой или составной ключ для коллектора, выражение для узла типа *condition*...

Шаблон описывается списком из пар: код поля и его длина. Шаблон может выглядеть следующим образом:

```
nfv9_ipv4_src_addr 4
nfv9_ipv4_dst_addr 4
nfv9_in_bytes      4
nfv9_in_pkts       4
nfv9_protocol      4
```

Объект *mapper* представляет собой функцию отображения одного шаблона на другой. Отображение сопоставляет поля входящего и исходящего шаблонов (*match rule*), например:

```
dest_ip = extract_field(tmpl, nfv9_ipv4_dst_addr)
bytes_in = extract_field(tmpl, nfv9_in_bytes)
packets_in = extract_field(tmpl, nfv9_in_pkts)
```

Каждая функция *extract_field* извлекает из шаблона *tmpl* поле, указанное вторым аргументом. В приведенном примере извлекаются поля *dst_address*, *in_bytes* и *in_pkts* из сообщения, а остальные игнорируются.

Collector характеризуется ключом и хранимыми значениями, например, коллектор можно объявить следующим образом:

```
key_simple extract_field(tint, nfv9_ipv4_dst_addr)
accumulate extract_field(tint, nfv9_in_bytes)
accumulate extract_field(tint, nfv9_in_pkts)
```

В приведенном примере коллектор принимает шаблон *tint*, суммирует число входящих байт и пакетов, группируя их по простому ключу *nfv9_ipv4_dst_addr*. Таким образом, коллектор содержит число

принятых байт и пакетов от конкретного адреса за определенный интервал времени.

Объект *condition* содержит выражение, при истинности которого совершается переход по одному ребру, а при ложности – по-другому. Например, можно обрабатывать только сообщения о конкретном адресе:

```
extract_field(tmpl, nfv9_ipv4_src_addr) == 192.168.1.100
```

Правило *kron-base* представляет собой подграф, в который не ведет ни одно *control flow* ребро из основного графа, а только *data flow* ребра (рисунок 31). В отличие от *content-base* правил, которые обрабатывают каждое входящее сообщение, *kron-base* правило вызывается через фиксированный интервал времени и обращается как к коллекторам *content-base* правил, так и к своим внутренним структурам.

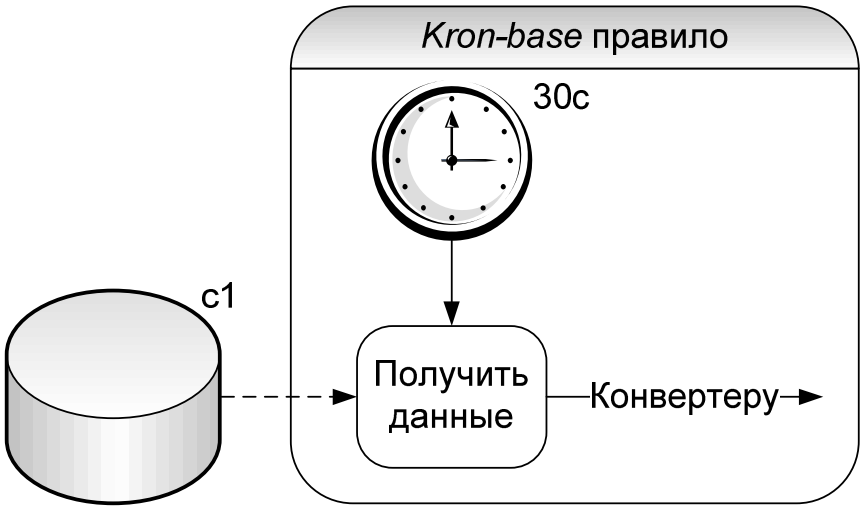


Рисунок 31 – Пример: *kron-base* правило

В приведенном примере *kron-base* правило вызывается каждые 30 с, берет данные из коллектора *c1* и передает их на конвертер.

4.2. Построение модуля на основе генетического программирования и верификации

Существует множество однотипных задач, когда необходимо агрегировать *NetFlow*-сообщения по сложному ключу из различных полей

и суммировать одно или несколько полей. Такие модули обычно возвращают *N*-максимумов среди всех агрегированных сообщений с фиксированным интервалом. Например, можно собирать сообщение по ключу из полей *<exp_ip, src_ip, dest_ip>*, а суммировать поля *bytes_in* и *packets_in*. Каждое такое сообщение показывает: сколько байт и пакетов передавалось между узлами сети.

Одна из таких задач состоит в определении узлов в сети, которые принимают больше всего трафика (по числу байт или числу пакетов) или же создают больше всего соединений.

В таблице 9 приведены поля *syslog*-сообщения для модуля «*Top Traffic Monitor*». Поля с номерами 1 – 8 используются для агрегации, поля 11 – 13 суммируются, для поля 9 вычисляется побитовое *ИЛИ*.

Таблица 9 – Поля *syslog*-сообщения модуля «*Top Traffic Monitor*»

№	**Сокращенное название**	**Описание поля**
1	*exp_ip*	*IPv4* адрес *NetFlow*-экспортера
2	*input_snmp*	*SNMP* индекс входящего интерфейса *NetFlow*-экспортера
3	*output_snmp*	*SNMP* индекс исходящего интерфейса *NetFlow*-экспортера
4	*protocol*	Транспортный протокол (например, *TCP = 6, UDP = 17*)
5	*src_ip*	Исходящий *IPv4* адрес
6	*src_port*	Исходящий порт
7	*dest_ip*	Входящий *IPv4* адрес
8	*dest_port*	Исходящий порт
9	*tcp_flag*	Побитовое *ИЛИ TCP* флагов
10	*packets_in*	Число принятых пакетов
11	*bytes_in*	Число принятых байт

№	Сокращенное название	Описание поля
12	*flow_count*	Число соединений
13	*percent_of_total*	Процент от общего числа переданных через экспортер байт

Как отмечалось ранее, приложение совместимо с различными *SIEM*-системами, которые могут, как просуммировать некоторые поля сообщений, например, поля *bytes_in* по *exp_ip* и *src_ip*, так и рассматривать такие сообщения по одному.

В настоящей работе была построена автоматная модель такого модуля, которая затем была переведена в модель приложения. Как описано в разд. 4.1.2, приложение работает с графом потока управления и данных (*CDFG*). Однако, автоматная модель содержит поставщики событий и объекты управления. Таким образом, необходимо преобразование графа в автомат и обратно.

Основная идея преобразования *CDFG* в автомат состоит в том, что узлы графа становятся объектами управления, действия над этими объектами переносятся из узлов на ребра, а входные/выходные данные объектов становятся событиями. *Data flow* ребра переносятся в автомат, если они совпадают с *control flow* ребрами графа. Остальные *data flow* ребра заменяются на «объектный» вызов входных и выходных воздействий. Например, если было ребро из *mapper m1* в коллектор *c2* и оба объекта работают с шаблоном *tmpl*, то на переходе в *c2* укажем *m1.tmpl/c2.mc2* – данные берутся из *m1* и сохраняются согласно правилу *mc2* коллектора *c2*. *Kron-base* правило заменяется на подграф в автомате, который имеет одно входящее ребро из стартового состояния по событию таймера.

Заметим, что реализация автомата не будет существовать – это только модель, которая не имеет смысла и не может выполняться без реализации *CDFG*.

Далее приведем взаимно однозначное соответствие между объектами управления автомата и узлами *CDFG*. Определим, что будет являться поставщиками событий, какие возможны события и действия.

4.2.1. Описание объектов управления

В качестве объектов управления будем рассматривать следующие узлы графа *CDFG*:

- *m1 – nfv5 mapper*;
- *m2 – nfv9/IPFIX mapper*;
- *c1* – коллектор для суммирования *bytes_in*, *packets_in*, *flow_count* и сбору *tcp_flag* по вектору из следующих полей: *exp_ip, protocol, src_ip, dest_ip, protocol, src_port, dest_port, input_snmp, output_snmp*;
- *c2* – коллектор для суммирования *bytes_in* для различных *IP*-адресов экспортеров (*exp_ip*);
- *k1 – kron-base* правило для поиска узлов с максимальным трафиком за определенный интервал времени (например, 30 секунд).

Первый mapper используется для извлечения необходимых полей из *NetFlow*-сообщений версии 5. Второй *mapper* требуется для *NetFlow*-сообщений версии 9. Оба объекта реализуют отображение шаблонов экспортеров во внутренний шаблон *tmpl*.

Первый коллектор собирает все сообщения, агрегируя их по составному ключу из следующих полей: *exp_ip, protocol, src_ip, dest_ip, protocol, src_port, dest_port, input_snmp, output_snmp*. Коллектор суммирует число байт, число пакетов, число соединений и вычисляет побитовое *ИЛИ*

для *TCP*-флагов. *N*-максимумов для каждого из экспортеров будут узлами с максимальным трафиком. Коллектор хранит данные в отсортированном формате по *exp_ip* и *bytes_in*.

Второй коллектор собирает сообщения по простому ключу – *IP*-адрес экспортера (*exp_ip*). Коллектор суммирует число байт, тем самым, каждая запись содержит суммарный трафик, прошедший через экспортер. Это позволяет вычислить процент от общего трафика для записей из первого коллектора. Коллектор хранит данные в отсортированном формате по *exp_ip*.

Правило *kron-base* с одинаковым интервалом выполняет следующие действия: проходит по второму коллектору (по всем экспортерам) и для каждого экспортера находит *N*-максимумов в первом коллекторе.

4.2.2. Входные и выходные воздействия

Рассмотрим, какие бывают события в автоматной модели, как они связаны с *CDFG*. События бывают следующего вида:

- *nfv5* – поступило *NetFlow*-сообщение версии 5;
- *nfv9* – поступило *NetFlow*-сообщение версии 9 или *IPFIX*;
- *tmpl* – получено *NetFlow*-сообщение внутреннего шаблона;
- *k1.t* – событие таймера (каждые 30 секунд) для *kron-base* правила;
- *c2.next* – получена следующая запись второго коллектора;
- *c2.end* – все записи второго коллектора прочитаны;
- *c1.next* – получить следующую запись из первого коллектора;
- *c1.end* – первый коллектор пуст, следующая запись относится к другому экспортеру или же уже найдены *N*-максимумов.

События *nfv5* и *nfv9* появляются в том случае, когда пришло *NetFlow*-сообщение от экспортера. Внутренний шаблон *tmpl* используется

уже после *mapper*-ов и всегда имеет одну и ту же структуру. Событие *tmpl* последовательно обрабатывается обоими коллекторами.

Событие таймера *t* появляется каждые 30 секунд и передает управление *kron-base* правилу *k1*.

События *c2.next* и *c2.end* используются для прохода по второму коллектору. Первое событие обрабатывается в том случае, когда коллектор еще содержит элементы, второе в случае, когда все элементы уже просмотрены.

События *c1.next* и *c1.end* применяются для прохода по элементам первого коллектора с *IP*-адресом экспортера из второго коллектора. Первое событие появляется только тогда, когда существуют еще не обработанные записи в первом коллекторе с экспортером, полученном при последнем событии *c2.next*. Второе событие появляется тогда, когда коллектор пуст, или все записи для текущего экспортера обработаны, или уже найдены *N*-максимумов.

Действия над объектами управления бывают следующих видов:

- *m1.mr1* – отображение полей *nfv5* во внутренний шаблон;
- *m2.mr2* – отображение полей *nfv9* во внутренний шаблон;
- *c1.mc1* – добавить записи в первый коллектор;
- *c2.mc2* – добавить запись во второй коллектор;
- *c1.exp* – взять в качестве текущего экспортера текущий ключ второго коллектора;
- *c1.sl* – отправить текущее *NetFlow*-сообщение из первого коллектора в очередь для конвертации его в *syslog*;
- *c1.clean* – очистить первый коллектор;
- *c2.clean* – очистить второй коллектор;
- *k1.reset* – перезапустить таймер.

Действия *m1.mr1* и *m2.mr2* отображают входящий шаблон во внутренний. Результатом такого отображения является шаблон *tmpl*, независимо от шаблона входящего *NetFlow*-сообщения.

Действие *c1.mc1* агрегирует сообщения с шаблоном *tmpl* по ключу из полей: *exp_ip, protocol, src_ip, dest_ip, protocol, src_port, dest_port, input_snmp, output_snmp*. Значение для ключа является сумма *bytes_in, packates_in, flow_count* и побитовое *ИЛИ* для *TCP* флагов.

Действие *c2.mc2* агрегирует сообщения с шаблоном *tmpl* по простому ключу *exp_ip*. Значение для ключа является сумма *bytes_in*.

Как для *mc1*, так и для *mc2* шаблон *tmpl* приходит от *m1* или *m2*. Если рассматривать *CDFG*, то должны были быть нарисованы четыре *data flow* ребра от *m1* и *m2* к *c1* и *c2*.

Действие *c1.exp* сохраняет текущий экспортер второго коллектора в качестве ключа для итерирования по первому коллектору. То есть по событию *c1.next* будут приходить только сообщения с данным экспортером.

Действие *c1.sl* посылает текущее сообщение первого экспортера в очередь для конвертации *NetFlow* в *syslog*.

Действия *c1.clean* и *c2.clean* очищают первый и второй коллекторы соответственно. Коллекторы должны очищаться каждые 30 секунд, после того как получены *N*-максимумов для каждого из экспортеров.

Действие *k1.reset* перезапускает таймер для *kron-base* правило после обработки всех экспортеров.

4.2.3. Спецификация модуля

При построении модуля *Top Traffic Monitor* для определения узлов сети, принимающих или отправляющих максимальный трафик, было использовано девять позитивных и 27 негативных сценариев и девять *LTL*-формул.

В таблице 10 приведены указанные позитивные сценарии работы.

Таблица 10 – Позитивные сценарии работы модуля *Top Traffic Monitor*

Сценарий	**Комментарий**
nfv5/m1.mr1; tmpl/c1.mc1; tmpl/c2.mc2	Последовательность обработки сообщения *nfv5*: оно сначала попадает в объект *mapper*, а затем в первый коллектор, после этого – во второй.
nfv9/m2.mr2; tmpl/c1.mc1; tmpl/c2.mc2	Последовательность обработки сообщения *nfv9*: оно сначала попадает в объект *mapper*, а затем в первый коллектор, после этого – во второй.
nfv5/m1.mr1; tmpl/c1.mc1; tmpl/c2.mc2; nfv9/m2.mr2; tmpl/c1.mc1; tmpl/c2.mc2	Последовательность обработки двух разных сообщений: сначала сообщение *nfv5*, а затем *nfv9*.
k1.t; c2.end/c1.clean, c2.clean, k1.reset	Сценарий работы *kron-base* правила, при котором второй коллектор пустой: ничего не обрабатывается.
k1.t; c2.next/c1.exp; c1.end; c2.end/c1.clean, c2.clean, k1.reset	Сценарий работы *kron-base* правила, при котором второй коллектор содержит один экспортер, а первый – пустой.
k1.t; c2.next/c1.exp; c1.next/c1.sl; c1.end; c2.end/ c1.clean, c2.clean, k1.reset	Сценарий работы *kron-base* правила, при котором оба коллектора содержат по одной записи с одинаковым экспортером.
k1.t; c2.next/c1.exp; c1.next/c1.sl; c1.end; c2.next/c1.exp; c1.next/c1.sl; c1.end; c2.end/c1.clean, c2.clean, k1.reset	Сценарий работы *kron-base* правила, при котором второй коллектор содержит два экспортера, а первый коллектор содержит две записи с двумя различными экспортерами.

Сценарий	Комментарий
k1.t; c2.next/c1.exp; c1.next/c1.sl; c1.next/c1.sl; c1.end; c2.end/c1.clean, c2.clean, k1.reset	Сценарий работы *kron-base* правила, при котором второй коллектор содержит один экспортер, а первый коллектор две записи для этого экспортера.
k1.t; c2.next/c1.exp; c1.next/c1.sl; c1.next/c1.sl; c1.end; c2.next/c1.exp; c1.next/sl; c1.end; c2.end/c1.clean, c2.clean, k1.reset	Сценарий работы *kron-base* правила, при котором второй коллектор содержит два экспортера, а первый – две записи для одного экспортера и одну запись для другого.

В таблице 11 приведены 27 негативных сценариев работы.

Таблица 11 – Негативные сценарии работы модуля *Top Traffic Monitor*

Сценарий	Комментарий
tmpl	Из стартового состояния могут быть либо обработаны входящие *NetFlow*-сообщения, либо передано управление в *kron-base* правило.
c2.next	
c1.next	
c1.end	
nfv5, nfv9	Если было получено и обработано сообщение *nfv5* или *nfv9*, то дальше сообщение передается в коллектор, а другие сообщения не могут выбираться из очереди входящих сообщений для обработки. Также не может быть передано управление в *kron-base* правило.
nfv5, t	
nfv5, c2.next	
nfv5, c1.next	
nfv5, c1.end	
nfv9, nfv5	
nfv9, t	
nfv9, c2.next	
nfv9, c1.next	
nfv9, c1.end	

Сценарий	Комментарий
nfv5, tmpl, nfv5	После обработки сообщения первым коллектором, единственное, что может выполнять автомат – это передача сообщения во второй коллектор. Следовательно, не могут быть обработаны другие сообщения или быть передано управление в *kron-base* правило.
nfv5, tmpl, t	
nfv5, tmpl, c2.next	
nfv9, tmpl, nfv9	
nfv9, tmpl, t	
nfv9, tmpl, c2.next	
t, nfv5	После передачи управления в *kron-base* правило, новые сообщения не могут обрабатываться. Также сообщения из первого коллектора могут обрабатываться, только если они присутствуют во втором коллекторе.
t, nfv9	
t, tmpl	
t, c1.next	
t, c1.end	
t, c2.next, c2.next	Экспортер из второго коллектора не может быть пропущен, пока не будут обработаны *N*-максимумов из первого коллектора.
t, c2.next, c1.next, c2.next	Нельзя перейти к обработке следующего экспортера из второго коллектора, если первый из них еще содержит записи с текущим экспортером и еще не был обработан *N*-ый максимум.

В таблице 12 приведены девять *LTL*-формул.

Таблица 12 – *LTL*-формулы для модуля *Top Traffic Monitor*

Сценарий	**Комментарий**
G(wasEvent(nfv5) => wasAction(m1.mr1))	Если пришло сообщение *nfv5*, то оно будет преобразовано во внутреннее сообщение с шаблоном *tmpl*.
G(wasEvent(nfv9) => wasAction(m2.mr2))	Если пришло сообщение *nfv9*, то оно будет преобразовано во внутреннее сообщение с шаблоном *tmpl*.
G(wasEvent(c2.next) => wasAction(c1.exp))	Если второй коллектор еще содержит записи, то выставляем следующую запись в качестве ключа для первого коллектора.
G(wasEvent(c2.end) => (wasAction(c1.clean) and wasAction(c2.clean) and wasAction(k1.reset))	Если все экспортеры из второго коллектора обработаны, то необходимо очистить оба коллектора и перезапустить таймер *kron-base* правила.
G(wasEvent(c1.next) => wasAction(c1.sl))	Если первый коллектор еще содержит записи для текущего экспортера, то отправить сообщение в очередь для конвертеров.
G((wasEvent(c2.end) or wasAction(c2.mc2)) => X(wasEvent(nfv5) or wasEvent(nfv9) or wasEvent(k1.t)))	Если закончена обработка экспортеров во втором коллекторе, или сообщение было добавлено во второй коллектор, то следующим сообщением может быть либо обработка входящего сообщения (*nfv5* или *nfv9*), либо будет передано управление в *kron-base* правило.

Сценарий	Комментарий
G(wasEvent(c2.next) => X(G(wasEvent(c1.next)) or U(wasEvent(c1.next), wasEvent(c1.end))))	Если был обработан следующий экспортер из второго коллектора, то будут обрабатываться сообщения из первого коллектора до тех пор, пока не закончатся сообщения, или не будут найдены *N*-максимумов, или сообщения для данного экспортера закончатся.
G((wasEvent(nfv5) or wasEvent(nfv9)) => X(R(wasEvent(tmpl) and wasAction(c2.mc2), !wasEvent(nfv5) and !wasEvent(nfv9) and !wasEvent(k1.t))))	Если было получено сообщение, то новые сообщения не будут обрабатываться, и не будет передано управление в *kron-base* правило, до тех пор, пока сообщение не окажется во втором коллекторе.
G(wasEvent(k1.t) => X(R(wasEvent(c2.end), !wasEvent(nfv5) and !wasEvent(nfv9) and !wasEvent(k1.t))))	Если было передано управление в *kron-base* правило, то не будут обрабатываться новые сообщения, и не будет еще раз передано управление в *kron-base* правило, пока не будут обработаны все экспортеры из второго коллектора.

4.2.4. Результаты построения модуля

Автомат управления строился 1000 раз по сценариям работы и тестовым примерам из разд. 4.2.3. Все автоматы получились изоморфные друг другу с точностью до нумерации состояний. Общие параметры алгоритма генетического программирования представлены в таблице 13.

Таблица 13 – Параметры алгоритма генетического программирования при построении модуля *Top Traffic Monitor*

Параметр эксперимента	**Значение**
Размер начальной популяции	2000
Число состояний у автоматов в начальном поколении	7
Ожидаемое число переходов	9
Доля особей, переходящих в следующее поколение. Остальные будут получены с помощью операции скрещивания	10 %
Число поколений до «малой» мутации	50
Число поколений до «большой» мутации	70
Вероятность мутации особи	2 %

В качестве численной оценки скорости работы алгоритма измерялось число вычислений функции приспособленности при нахождении автомата. Среднее значение вычислений функции приспособленности оказалось равным 3.012×10^5. Минимальное число вычислений – 8.318×10^4. Максимальное число – 2.096×10^6. Среднеквадратичное отклонение – 1.904×10^5.

Автомат, полученный в результате генераций, представлен на рисунке 32. Начальное состояние выделено жирным. Два из семи состояний не имели входящих и исходящих переходов (были не достижимы), и поэтому удалены из автомата.

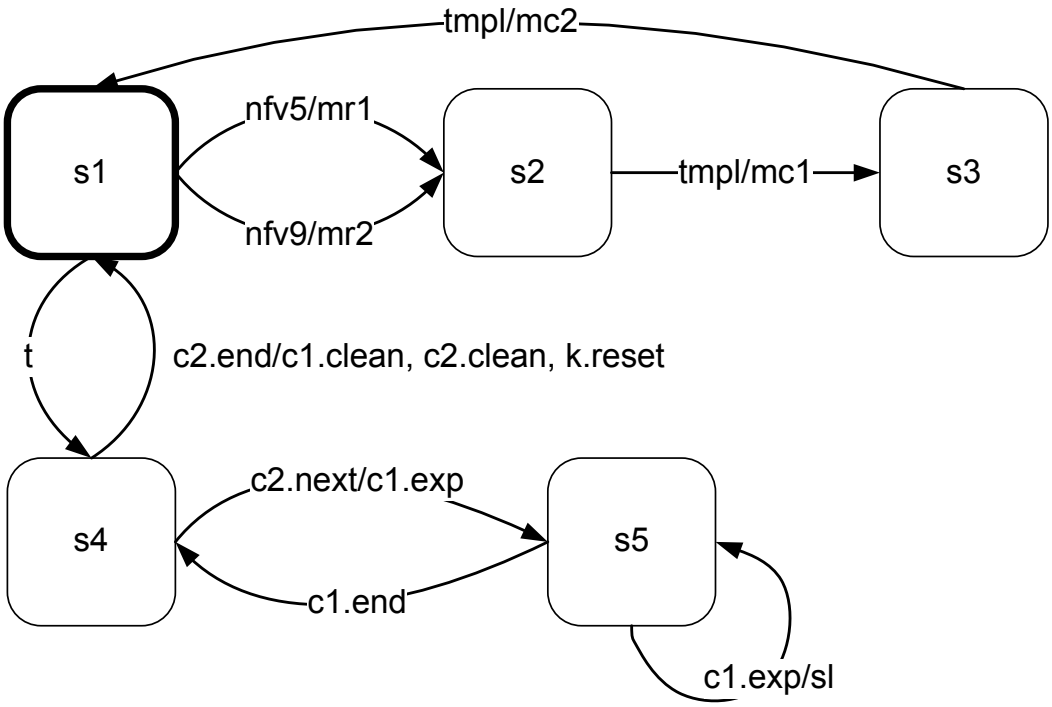


Рисунок 32 – Автомат управления модулем *Top Traffic Monitor*, построенный с помощью алгоритма генетического программирования

Как отмечалось выше, модули для приложения представляют собой граф потока управления и данных. Поэтому необходимо обратное преобразование из автомата в *CDFG*. Граф управления после преобразования автомата представлен на рисунке 33.

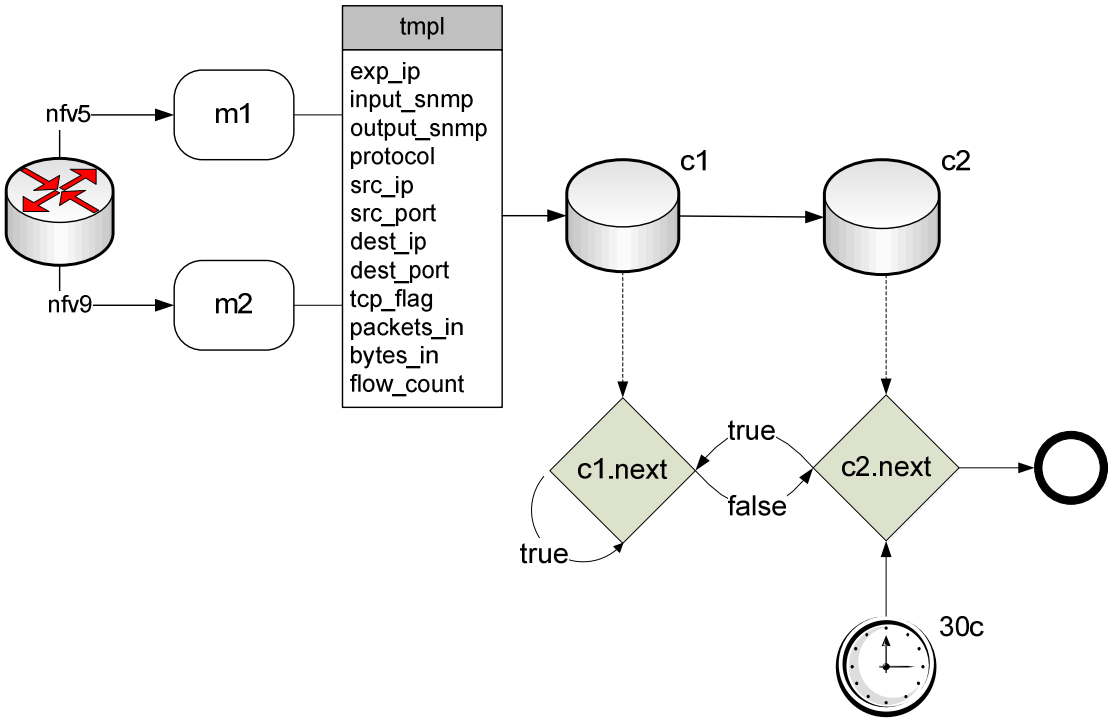


Рисунок 33 – Граф управления модулем *Top Traffic Monitor*

Полученный граф модуля принимает на вход сообщения с шаблонами двух видов: *nfv5* и *nfv9*. После поступления сообщения из него извлекаются поля *exp_ip, protocol, src_ip, dest_ip, protocol, src_port, dest_port, input_snmp, output_snmp, bytes_in, packets_in, flow_count*, из которых формируется новое сообщение с внутренним *NetFlow* шаблоном *tmpl*. Это сообщение сначала поступает на вход первому коллектору, а затем второму. Первый коллектор агрегирует сообщения по сложному ключу {*exp_ip, protocol, src_ip, dest_ip, protocol, src_port, dest_port, input_snmp, output_snmp*}, второй коллектор по простому ключу, состоящему из одного поля *exp_ip*. *Data flow* ребра из *m1* в *c1, c2* и из *m2* в *c1, c2* не изображены в графе, так как они однозначно определяются типом исходящих и входящих данных узлов.

Kron-base правило выполняется каждые 30 секунд. Он обращается ко второму коллектору, последовательно извлекая оттуда *IP*-адрес экспортера и общее число байт, пройденное через него. Для каждого экспортера из второго коллектора извлекается *N*-максимумов из первого коллектора.

Полученные сообщения посылаются в очередь сообщений для конвертеров.

XML-описание параметров эксперимента, сценариев работы и *LTL*-формул приведены в приложении 3. Результат работы эксперимента представляет собой *UniMod*-модель автомата в *XML*-формате (приложение 4).

Модуль, идентичный построенному, поставляется вместе с описанным программным продуктом под названием «*Top Traffic Monitor*». Существуют аналогичные модули для определения *N*-максимумов числа соединений и числа пакетов: «*Top Connections Monitor*» и «*Top Packets Monitor*» соответственно. Все модули могут быть построены автоматически, как было продемонстрировано на примере, рассмотренным ваше. Испытания программного продукта показали, что построенный автомат соответствует спецификации, а модуль работает правильно без дополнительной отладки.

Выводы по главе 4

1. Рассмотрен программный продукт, выпускаемый *ООО «ЭВЕЛОПЕРС»*, его основные части и модули. Приведено описание графа потока управления и данных, описывающего поведение модуля для рассматриваемого продукта.
2. Разработанное инструментальное средство и технология для генерации автоматов на основе генетического программирования и верификации применены при построении модуля для программного продукта. Был создан набор сценариев работы и набор *LTL*-формул, по которым построен автомат управления модулем. Этот автомат преобразован в граф потока управления и данных. Испытания программного продукта показали, что

построенный модуль работает правильно без дополнительной отладки. Тем самым была продемонстрирована возможность применения предложенного подхода для генерации автомата, применяемого на практике.

ЗАКЛЮЧЕНИЕ

В ходе диссертационной работы получены следующие результаты, обладающие научной новизной:

1. Предложена функция приспособленности, учитывающая верификацию. В отличие от известных, в ней вклад каждой *LTL*-формулы не ноль или единица, а значение в интервале [0; 1], что более точно оценивает выполнение формулы.
2. Разработаны операции мутации и скрещивания, учитывающие верификацию, что позволяет строить автомат быстрее, чем при учете верификации только в функции приспособленности.
3. Разработан алгоритм генерации автоматов с учетом контрактов. Ранее при генерации автоматов контракты не использовались. Так как контракты – это *LTL*-формулы определенного вида, то их применение позволяет ускорить генерацию автоматов с учетом верификации.
4. Разработан алгоритм генерации автоматов по сценариям работы и верификации. Применение сценариев позволяет ускорить генерацию автоматов по сравнению с генерацией автоматов по тестам.
5. Разработан верификатор *AutomataVerificator*, приспособленный для поддержки генерации автоматов на основе генетического программирования.
6. Разработана технология генерации автоматов с помощью генетического программирования и верификации.
7. На основе разработанных верификатора и технологии создано инструментальное средство *GABP* для генерации автоматов с помощью генетического программирования и верификации.
8. Это инструментальное средство было использовано для генерации автомата управления модулем *Top Traffic Monitor* для поиска узлов

сети с максимальным трафиком, который входит в программный продукт, выпускаемый *ООО «ЭВЕЛОПЕРС»* (Санкт-Петербург). Испытания этого продукта показали, что построенный автомат соответствует спецификации, а модуль работает правильно без дополнительной отладки.

СПИСОК ИСТОЧНИКОВ

ПЕЧАТНЫЕ ИЗДАНИЯ НА РУССКОМ ЯЗЫКЕ

1. *Александров А. В., Казаков С. В. Сергушичев А. А., Царев Ф. Н. Шалыто А. А.* Генерация конечных автоматов для управления моделью беспилотного самолета // Научно-технический вестник СПбГУ ИТМО. 2011. № 2, с. 3 – 11. http://is.ifmo.ru/works/2011/Vestnik/72-2/01-Aleksandrov-Kazakov-Tsarev-Shalyto.pdf
2. *Борисенко А., Федотов П., Степанов О., Шалыто А.* Разработка надежного программного обеспечения со сложным поведением / Сборник трудов конференции 5th Central and Eastern European Software Engineering Conference in Russia. М.: 2009, с. 125 – 128.
3. *Васильева К. А., Кузьмин Е. В., Соколов В. А.* Верификация автоматных программ с использованием LTL // Моделирование и анализ информационных систем. 2007. № 1, с. 3 – 14. http://is.ifmo.ru/verification/_ltl_aut_ver_1.pdf
4. *Вельдер С. Э., Лукин М. А., Шалыто А. А., Яминов Б. Р.* Верификация автоматных программ. СПб.: Наука, 2011. http://is.ifmo.ru/verification/velder_verification_posobie_nauka.pdf
5. *Гладков Л. А., Курейчик В. В., Курейчик В. М.* Генетические алгоритмы. М.: Физматлит, 2006.
6. *Гуров В. С., Мазин М. А., Нарвский А. С., Шалыто А. А.* UML. SWITCH-технология. Eclipse // Информационно-управляющие системы. 2004. № 6, с. 12 – 17. http://is.ifmo.ru/works/uml-switch-eclipse/
7. *Кларк Э., Грамберг О., Пелед Д.* Верификация моделей программ. М.: МЦНМО, 2002.
8. *Кормен* Т., *Лейзерсон* Ч., *Ривест* Р., *Штайн К.* Алгоритмы. Построение и анализ. М.: Вильямс, 2005.

9. *Кузьмин Е. В., Соколов В. А.* О верификации «автоматных» программ / Актуальные проблемы математики и информатики. Сборник статей к 20-летию факультета ИВТ ЯрГУ им. П. Г. Демидова. Ярославль: ЯрГУ. 2006, с. 27 – 32. http://is.ifmo.ru/verification/_verautpr.pdf
10. *Курейчик В. М.* Генетические алгоритмы. Состояние, проблемы, перспективы // Известия РАН. Теория и системы управления. 1999. № 1, с. 144 – 160.
11. *Левенштейн В. И.* Двоичные коды с исправлением выпадений, вставок и замещений символов // Доклады Академии наук СССР. 1965. № 4, с. 845 – 848.
12. *Лобанов П. Г., Шалыто А. А.* Использование генетических алгоритмов для автоматического построения конечных автоматов в задаче о «Флибах» / Сборник докладов 4-ой Всероссийской научной конференции «Управление и информационные технологии» (УИТ-2006). СПбГЭТУ «ЛЭТИ». 2006, с. 144 – 149. http://is.ifmo.ru/works/flib
13. *Люгер Дж.* Искусственный интеллект: стратегии и методы решения сложных проблем. М.: Вильямс, 2003.
14. *Мейер Б.* Объектно-ориентированное конструирование программных систем. М.: Интернет-университет информационных технологий, 2005.
15. *Поликарпова Н. И., Шалыто А. А.* Автоматное программирование. СПб.: Питер, 2009. http://is.ifmo.ru/books/_book.pdf
16. *Смелянский Р. Л.* Применение темпоральной логики для спецификации поведения программных систем // Программирование. 1993. № 1, с. 3 – 28.
17. *Тэллес М., Хсих Ю.* Наука отладки. М.: КУДИЦ-Образ, 2003.
18. *Царев Ф. Н.* Метод построения автоматов управления системами со сложным поведением на основе тестов с помощью генетического

программирования / Материалы Международной научной конференции «Компьютерные науки и информационные технологии». Саратов: СГУ. 2009, с. 216 – 219.

19. *Царев Ф. Н.* Методы построения конечных автоматов на основе эволюционных алгоритмов. Диссертация на соискание ученой степени кандидата технических наук. СПб.: НИУ ИТМО. 2012. http://is.ifmo.ru/disser/tsarev_disser.pdf
20. *Шалыто А. А.* Автоматное программирование / Материалы конференции с международным участием «Технические и программные средства систем управления, контроля и измерения» (УКИ-2010). ИПУ РАН. 2010, с. 156 – 167. http://cmm.ipu.ru/proc/Шалыто%20A.A.%20.pdf
21. *Шалыто А. А.* Switch-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. http://is.ifmo.ru/books/switch/1
22. *Шалыто А. А.* Логическое управление. Методы аппаратной и программной реализации. СПб.: Наука, 2000. http://is.ifmo.ru/books/log_upr/1

ПЕЧАТНЫЕ ИЗДАНИЯ НА АНГЛИЙСКОМ ЯЗЫКЕ

23. *Angeline P. J., Pollack J.* Evolutionary Module Acquisition / Proceedings of the Second Annual Conference on Evolutionary Programming. 1993. http://www.demo.cs.brandeis.edu/papers/ep93.pdf
24. *Chambers L.* Practical Handbook of Genetic Algorithms. Complex Coding Systems. Volume III. CRC Press. 1999.
25. *Courcoubetis C., Vardi M., Wolper P., Yannakakis M.* Memory-Efficient Algorithms for the Verification of Temporal Properties / Proceedings of conference «Formal Methods in System Design». 1992, pp. 275 – 288.

26. Dahl O. J., Dijkstra E. W., Hoare C. A. R. Structured Programming. Academic Press. London. 1972.
27. *Gastin P., Oddoux D.* Fast LTL to Büchi Automata Translation / 13th Conference on Computer Aided Verification (CAV'01). 2001, pp. 53 – 65.
28. *Gerth R., Peled D., Vardi M. Y.,Wolper P.* Simple On-the-fly Automatic Verification of Linear Temporal Logic / Proceedings of the 15th Workshop on Protocol Specification. Testing and Verification. Warsaw. 1995, pp. 3 – 18.
29. *Hoffman L.* Talking Model-Checking Technology // Communications of the ACM. 2008. № 7, pp. 110 – 112.
30. *Holzmann G. J.* The Model Checker *SPIN* // IEEE Transactions on software engineering. 1997. Vol. 23. Issue 5, pp. 279 – 295.
31. *Jefferson D., Collins R., Cooper C., Dyer M., Flowers M., Korf R., Taylor C., Wang A.* The Genesys System: Evolution as a Theme in Artificial Life / Proceedings of Second Conference on Artificial Life. MA: Addison-Wesley. 1992, pp. 549 – 578. www.cs.ucla.edu/~dyer/Papers/AlifeTracker/Alife91Jefferson.html
32. *Johnson C.* Genetic Programming with Fitness based on Model Checking. Lecture Notes in Computer Science. Springer Berlin / Heidelberg. 2007. Volume 4445/2007, pp. 114 – 124.
33. *Katz G., Peled D.* Model Checking-Based Genetic Programming with an Application to Mutual Exclusion / Proceedings of the 14th International Conference, Tools and Algorithms for the Construction and Analysis of Systems (TACAS-2008), Budapest. Springer Berlin Heidelberg. 2008, pp 141 – 156.
34. *Katz G., Peled D.* Genetic Programming and Model Checking: Synthesizing New Mutual Exclusion Algorithms / Proceedings of the 6th International Symposium, Automated Technology for Verification and

Analysis (ATVA-2008), Seoul. Springer Berlin Heidelberg. 2008, pp 33 – 47.

35. *Katz G., Peled D.* Synthesizing Solutions to the Leader Election Problem using Model Checking and Genetic Programming / Proceedings of the 5th International Haifa Verification Conference (HVC-2009), Haifa. Springer Berlin Heidelberg. 2009, pp 117 – 132.
36. *Katz G., Peled D.* Code Mutation in Verification and Automatic Code Correction / Proceedings of the 16th International Conference, Tools and Algorithms for the Construction and Analysis of Systems (TACAS-2010), Paphos. Springer Berlin Heidelberg. 2010, pp 435 – 450.
37. *Katz G., Peled D.* MCGP: A Software Synthesis Tool Based on Model Checking and Genetic Programming / Proceedings of the 8th International Symposium, Automated Technology for Verification and Analysis (ATVA 2010), Singapore. Springer Berlin Heidelberg. 2010, pp 359 – 364.
38. *Leveson N., Turner C.S.* An Investigation of the Therac-25 Accidents // IEEE Computer. Vol. 26. 1993. No. 7, p. 18 – 41.
39. *Lions J.L., et al.* ARIANE 5 Flight 501 Failure, Report by the Inquiry Board. Paris, 1996.
40. *Miller J.* The Coevolution of Automata in the Repeated Prisoner's Dilemma. Working Paper. Santa Fe Institute. 1989 // Journal of Economic Behavior & Organization. 1996. Vol. 29. Issue 1, pp. 87 – 112.
41. *Reynolds C. W.* Competition, Coevolution and the Game of Tag / Proceedings of Artificial Life IV. Cambridge. MA: MIT Press. 1994, pp. 59 − 69. http://www.red3d.com/cwr/papers/1994/alife4.html
42. *Somenzi F., Bloem R.* Efficient Büchi Automata from LTL Formulae / 12th Conference on Computer Aided Verification (CAV'00). 2000, pp. 248 – 263.
43. *Van Wyk C.* An LTL Verification System Based on Automata Theory. University of Stellenbosch, South Africa. 1999, pp. 1 – 70.

Ресурсы сети Интернет

44. Государственный контракт «Разработка технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода». Промежуточный отчет по I этапу «Выбор направления исследований и базовых компонентов». СПбГУ ИТМО, 2007.
    http://is.ifmo.ru/verification/_2007_01_report-verification.pdf
45. Государственный контракт «Разработка технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода». Промежуточный отчет по II этапу «Теоретические исследования поставленных перед НИР задач». СПбГУ ИТМО, 2007.
    http://is.ifmo.ru/verification/_2007_02_report-verification.pdf
46. Государственный контракт «Разработка технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода». Промежуточный отчет по III этапу «Экспериментальные исследования поставленных перед НИР задач». СПбГУ ИТМО, 2007.
    http://is.ifmo.ru/verification/_2007_02_report-verification.pdf
47. Государственный контракт «Разработка технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода». Заключительный отчет по IV этапу «Обобщение и оценка результатов исследований». СПбГУ ИТМО, 2007.
    http://is.ifmo.ru/verification/_2007_02_report-verification.pdf
48. Государственный контракт «Технология генетического программирования для генерации автоматов управления системами со сложным поведением». Промежуточный отчет по этапу I «Выбор

направлений исследований и базовых компонентов». СПбГУ ИТМО, 2007. http://is.ifmo.ru/genalg/_2007_01_report-genetic.pdf

49. Государственный контракт «Технология генетического программирования для генерации автоматов управления системами со сложным поведением». Промежуточный отчет по этапу II «Теоретические исследования поставленных перед НИР задач». СПбГУ ИТМО, 2007. http://is.ifmo.ru/genalg/_2007_02_report-genetic.pdf
50. Государственный контракт «Технология генетического программирования для генерации автоматов управления системами со сложным поведением». Промежуточный отчет по этапу III «Экспериментальные исследования поставленных перед НИР задач». СПбГУ ИТМО, 2007. http://is.ifmo.ru/genalg/_2007_03_report-genetic.pdf
51. Государственный контракт «Технология генетического программирования для генерации автоматов управления системами со сложным поведением». Промежуточный отчет по этапу IV «Обобщение и оценка результатов исследований». СПбГУ ИТМО, 2007. http://is.ifmo.ru/genalg/_2007_04_report-genetic.pdf
52. Государственный контракт «Разработка методов машинного обучения на основе генетических алгоритмов для построения управляющих конечных автоматов». Промежуточный отчет по этапу I. СПбГУ ИТМО, 2009. http://is.ifmo.ru/science/_nk-385-1-1.pdf. Выполнен при участии автора.
53. *Золотов Е., Карпов М.* Четыре способа сломать космические аппарат. 16.12.2010. http://old.computerra.ru/vision/29928/
54. *Конев Б. Ю.* Введение в моделирование и верификацию аппаратных и программных систем. Computer Science клуб при ПОМИ РАН. Слайды лекций. 2007. http://logic.pdmi.ras.ru/~infclub/?q=courses/verification

55. *Миронов А. М.* Математическая теория программных систем. http://intsys.msu.ru/staff/mironov/mthprogsys.pdf
56. *Ульянцев В. И.* Отчет о верификации программы управления часами с будильником. НИУ ИТМО. 2012. http://is.ifmo.ru/verification/2013/alarm_clock_verification.pdf
57. *LTL2BA* project. http://www.lsv.ens-cachan.fr/~gastin/ltl2ba/
58. *LTL2BA4J* project. http://www.sable.mcgill.ca/~ebodde/rv/ltl2ba4j/
59. *NetFlow* Export Datagram Format. http://www.cisco.com/en/US/docs/net_mgmt/netflow_collection_engine/3.6/user/guide/format.pdf
60. *RFC 3164*. The BSD syslog Protocol. http://www.ietf.org/rfc/rfc3164.txt
61. *RFC 3954*. Cisco Systems NetFlow Services Export Version 9. http://www.ietf.org/rfc/rfc3954.txt
62. *RFC 5101*. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information. http://tools.ietf.org/html/rfc5101
63. *RFC 5102*. Information Model for IP Flow Information Export. http://tools.ietf.org/html/rfc5102

Публикации автора

Статьи в журналах из перечня ВАК

64. *Егоров К. В., Шалыто А. А.* Методика верификации автоматных программ // Информационно-управляющие системы. 2008. № 5, с. 15 – 21. http://is.ifmo.ru/works/_egorov.pdf
65. *Егоров К. В., Шалыто А. А.* Разработка верификатора автоматных программ // Научно-технический вестник СПбГУ ИТМО. 2008. Вып. 53, с. 177 – 188. http://is.ifmo.ru/works/2008/Vestnik/53/15-verification-of-automata-based-programs-tool-development.pdf

66. *Егоров К. В., Царев Ф. Н., Шалыто А. А.* Применение генетического программирования для построения автоматов управления системами со сложным поведением на основе обучающих примеров и спецификации // Научно-технический вестник СПбГУ ИТМО. 2010. № 5 (69), с. 81 – 86. http://is.ifmo.ru/works/_2010_10_13_egorov.pdf

Другие статьи автора

67. *Егоров К. В., Царев Ф. Н., Шалыто А. А.* Совместное применение генетического программирования и верификации для построения автоматов управления системами со сложным поведением // Труды СПИИРАН. 2010. Вып. 15, с. 123 – 135.

Материалы конференций с участием автора

68. *Егоров К. В., Царев Ф. Н.* Совместное применение генетического программирования и верификации моделей для построения автоматов управления системами со сложным поведением / Сборник трудов 32-й конференции молодых ученых и специалистов «Информационные технологии и системы» (ИТИС-2009). М.: Институт проблем передачи информации им. А. А. Харкевича РАН. 2009,
с. 77 – 82. http://is.ifmo.ru/genalg/_2010_01_14_egorov_tsarev.pdf
69. *Егоров К. В., Царев Ф. Н., Парфенов В. Г.* Совместное применение генетического программирования и верификации моделей для построения автоматов управления системами со сложным поведением / Труды XVII Всероссийской научно-методической конференции «Телематика-2010». Т. 2. СПбГУ ИТМО. 2010, с. 344, 345.
70. *Егоров К. В., Царев Ф. Н.* Применение генетического программирования для построения автоматов управления системами со сложным поведением на основе верификации моделей и

обучающих примеров / Материалы второй межвузовской научной конференции по проблемам информатики «СПИСОК-2011». СПб.: ВВМ. 2011, с. 343 – 350.
http://is.ifmo.ru/works/2011/SPISOK/07-Egorov-Tsarev.pdf

71. *Егоров К. В., Шалыто А. А.* Применение генетического программирования для построения автоматов управления системами со сложным поведением на основе контрактов и тестовых примеров / Материалы второй межвузовской научной конференции по проблемам информатики «СПИСОК-2011». СПб.: ВВМ. 2011, с. 351 – 355. http://is.ifmo.ru/works/2011/SPISOK/08-Egorov-Shalyto.pdf
72. *Егоров К. В., Шалыто А. А.* Применение генетического программирования для построения автоматов управления системами со сложным поведением на основе контрактов и тестовых примеров / Сборник научных трудов VI-ой Международной научно-практической конференции «Интегрированные модели и мягкие вычисления в искусственном интеллекте. М.: Физмалит. 2011, с. 610 – 615.
73. *Egorov K., Tsarev F.* Finite State Machine Induction using Genetic Programming Based on Testing and Model Checking / Proceedings of the 2011 GECCO Conference Companion on Genetic and Evolutionary Computation. NY.: ACM. 2011, pp. 759 – 762.
http://is.ifmo.ru/articles_en/2011/GECCO2011-Tsarev-Egorov-FSM-induction.pdf
74. *Егоров К. В., Царев Ф. Н., Шалыто А. А.* Построение автоматов управления системами со сложным поведением на основе верификации и сценариев работы / Материалы третьей всероссийской научной конференция по проблемам информатики «СПИСОК-2012». СПб.: ВВМ. СПбГУ. 2012, с. 411 – 414.
http://is.ifmo.ru/works/2012/SPISOK/egorov.pdf

ПРИЛОЖЕНИЕ 1. *XML*-ОПИСАНИЕ ДАННЫХ ДЛЯ ЭКСПЕРИМЕНТОВ ПО ГЕНЕРАЦИИ АВТОМАТА УПРАВЛЕНИЯ ДВЕРЬМИ ЛИФТА

```
<program>
  <parameters>
    <fixedOutput>true</fixedOutput>
    <stateNumber>6</stateNumber>
    <populationSize>2000</populationSize>
    <partStay>0.1</partStay>
    <mutationProbability>0.01</mutationProbability>
    <timeSmallMutation>70</timeSmallMutation>
    <timeBigMutation>100</timeBigMutation>
    <desiredFitness>0.093</desiredFitness>
  </parameters>

  <inputSet>e11, e12, e2, e3, e4</inputSet>
  <outputSet>z1, z2, z3</outputSet>

  <group>
    <formulas>
      <ltl>G( !wasEvent(ep.e11) || wasAction(co.z1) )</ltl>
      <ltl>G( (!wasEvent(ep.e12) || wasAction(co.z2)) and
(!wasAction(co.z2) || wasEvent(ep.e12)) )</ltl>
      <ltl>G( (!wasEvent(ep.e4) || wasAction(co.z3)) and (!wasAction(co.z3)
|| wasEvent(ep.e4)))</ltl>
      <ltl>G( !wasEvent(ep.e3) || wasAction(co.z1) )</ltl>
      <ltl>G( !wasEvent(ep.e2) || X(wasEvent(ep.e11) || wasEvent(ep.e12))
)</ltl>
      <ltl>G( !wasEvent(ep.e11) or X(wasEvent(ep.e4) or wasEvent(ep.e2))
)</ltl>
      <ltl>G( !wasAction(co.z1) or X(wasEvent(ep.e2) or wasEvent(ep.e4))
)</ltl>
      <ltl>G( !wasEvent(ep.e12) or X(wasEvent(ep.e2) or wasEvent(ep.e3) or
wasEvent(ep.e4)) )</ltl>
      <ltl>G( !wasAction(co.z1) or X( U(!wasAction(co.z1), wasAction(co.z2)
or wasEvent(ep.e4)) ) )</ltl>
      <ltl>G( !wasAction(co.z2) or X( U(!wasAction(co.z2), wasAction(co.z1)
or wasEvent(ep.e4)) ) )</ltl>

      <ltl>!F(wasEvent(ep.e4) and X(F(wasEvent(ep.e11) || wasEvent(ep.e12)
|| wasEvent(ep.e2) || wasEvent(ep.e3))))</ltl>
    </formulas>

    <tests>
      <test>
        <input>e11, e2, e12, e2</input>
        <output>z1, z2</output>
        <ptest>e11/z1; e2; e12/z2; e2</ptest>
      </test>
      <test>
        <input>e11, e2, e12, e2, e11, e2, e12, e2</input>
        <output>z1, z2, z1, z2</output>
        <ptest>e11/z1; e2; e12/z2; e2; e11/z1; e2; e12/z2; e2</ptest>
      </test>
      <test>
        <input>e11, e2, e12, e3, e2, e12, e2</input>
        <output>z1, z2, z1, z2</output>
```

```
        <ptest>e11/z1; e2; e12/z2; e3/z1; e2; e12/z2; e2</ptest>
      </test>
      <test>
        <input>e11, e2, e12, e2, e11, e2, e12, e3, e2, e12, e2</input>
        <output>z1, z2, z1, z2, z1, z2</output>
        <ptest>e11/z1; e2; e12/z2; e2; e11/z1; e2; e12/z2; e3/z1; e2;
e12/z2; e2</ptest>
      </test>
      <test>
        <input>e11, e2, e12, e3, e2, e12, e3, e2, e12, e2</input>
        <output>z1, z2, z1, z2, z1, z2</output>
        <ptest>e11/z1; e2; e12/z2; e3/z1; e2; e12/z2; e3/z1; e2; e12/z2;
e2</ptest>
      </test>
      <test>
        <input>e11, e4</input>
        <output>z1, z3</output>
        <ptest>e11/z1; e4/z3</ptest>
      </test>
      <test>
        <input>e11, e2, e12, e4</input>
        <output>z1, z2, z3</output>
        <ptest>e11/z1; e2; e12/z2; e4/z3</ptest>
      </test>
      <test>
        <input>e11, e2, e12, e2, e11, e4</input>
        <output>z1, z2, z1, z3</output>
        <ptest>e11/z1; e2; e12/z2; e2; e11/z1; e4/z3</ptest>
      </test>
      <test>
        <input>e11, e2, e12, e3, e4</input>
        <output>z1, z2, z1, z3</output>
        <ptest>e11/z1; e2; e12/z2; e3/z1; e4/z3</ptest>
      </test>
    </tests>

    <negativeTests>
      <ntest>e12</ntest>
      <ntest>e2</ntest>
      <ntest>e3</ntest>
      <ntest>e4</ntest>

      <ntest>e11, e11</ntest>
      <ntest>e11, e12</ntest>
      <ntest>e11, e3</ntest>

      <ntest>e11, e2, e11</ntest>
      <ntest>e11, e2, e2</ntest>
      <ntest>e11, e2, e3</ntest>
      <ntest>e11, e2, e4</ntest>

      <ntest>e11, e2, e12, e11</ntest>
      <ntest>e11, e2, e12, e12</ntest>

      <ntest>e11, e2, e12, e3, e11</ntest>
      <ntest>e11, e2, e12, e3, e12</ntest>
      <ntest>e11, e2, e12, e3, e3</ntest>

      <ntest>e11, e2, e12, e2, e12</ntest>
      <ntest>e11, e2, e12, e2, e2</ntest>
      <ntest>e11, e2, e12, e2, e3</ntest>
      <ntest>e11, e2, e12, e2, e4</ntest>
```

```
      <ntest>e11, e2, e12, e4, e11</ntest>
      <ntest>e11, e2, e12, e4, e12</ntest>
      <ntest>e11, e2, e12, e4, e2</ntest>
      <ntest>e11, e2, e12, e4, e3</ntest>
      <ntest>e11, e2, e12, e4, e4</ntest>

      <ntest>e11, e4, e11</ntest>
      <ntest>e11, e4, e12</ntest>
      <ntest>e11, e4, e2</ntest>
      <ntest>e11, e4, e3</ntest>
      <ntest>e11, e4, e4</ntest>
    </negativeTests>

  </group>
</program>
```

ПРИЛОЖЕНИЕ 2. *UniMod*-МОДЕЛЬ АВТОМАТА УПРАВЛЕНИЯ ДВЕРЬМИ ЛИФТА

```
<model name="Model1">
  <controlledObject class="ControlledObject" name="o1"/>
  <eventProvider class="EventPrivider" name="p1">
    <association clientRole="p1" targetRef="A1"/>
  </eventProvider>
  <rootStateMachine>
    <stateMachineRef name="A1"/>
  </rootStateMachine>
  <stateMachine name="A1">
    <configStore class="DistinguishConfigManager"/>
    <association clientRole="A1" supplierRole="o1" targetRef="o1"/>
    <state name="Top" type="NORMAL">
      <state name="s0" type="NORMAL"/>
      <state name="s1" type="NORMAL"/>
      <state name="s2" type="NORMAL"/>
      <state name="s3" type="NORMAL"/>
      <state name="s4" type="INITIAL"/>
      <state name="s5" type="NORMAL"/>
    </state>
    <transition Event="e2" sourceRef="s0" targetRef="s2">
    </transition>
    <transition Event="e4" sourceRef="s0" targetRef="s5">
      <outputAction ident="o1.z3"/>
    </transition>
    <transition Event="e2" sourceRef="s1" targetRef="s4">
    </transition>
    <transition Event="e3" sourceRef="s1" targetRef="s0">
      <outputAction ident="o1.z1"/>
    </transition>
    <transition Event="e4" sourceRef="s1" targetRef="s5">
      <outputAction ident="o1.z3"/>
    </transition>
    <transition Event="e12" sourceRef="s2" targetRef="s1">
      <outputAction ident="o1.z2"/>
    </transition>
    <transition Event="e11" sourceRef="s4" targetRef="s0">
      <outputAction ident="o1.z1"/>
    </transition>
  </stateMachine>
</model>
```

ПРИЛОЖЕНИЕ 3. *XML*-ОПИСАНИЕ ДАННЫХ ДЛЯ ЭКСПЕРИМЕНТОВ ПО ГЕНЕРАЦИИ МОДУЛЯ *TOP TRAFFIC MONITOR*

```
<program>
  <!--
    objects:
    m1 - nfv5 mapper
    m2 - nfv9/ipfix mapper

    c1 - collector <exp, src, dst, ...> in_bytes | in_pkts | flows
    c2 - collector <exp>, in_bytes

    EVENTS:
    nfv5 – сообщение nfv5
    nfv9 – сообщение nfv9/ipfix
    tmpl – внутренний шаблон <exp_ip, src_addr, dst_addr, src_port,
dst_port...>
    t – событие таймера >= 60s
    c2_next – извлечение следующего элемента из коллектора c2
    c2_empty – коллектор c2 пуст
    e1 - c1.next[exp=curExp]
    e2 - c1.next[exp!=curExp] || c1.empty

    ACTIONS:
    mr1, mr2 - match rules (extract_field(...))
    mc1, mc2 - collect
    sl – отправить сообщение в очередь конвертеров
    c1_exp - curExp = c2_next
    c1_clean – очистить коллектор c1
    c2_clean – очистить коллектор c2
    t_reset – перезапустить таймер
  -->

  <parameters>
    <fixedOutput>true</fixedOutput>
    <stateNumber>7</stateNumber>
    <populationSize>2000</populationSize>
    <partStay>0.1</partStay>
    <mutationProbability>0.02</mutationProbability>
    <timeSmallMutation>50</timeSmallMutation>
    <timeBigMutation>70</timeBigMutation>
    <desiredFitness>0.0091</desiredFitness>
  </parameters>

  <inputSet>nfv5, nfv9, tmpl, t, c2_next, c2_empty, e1, e2</inputSet>
  <outputSet>mr1, mr2, mc1, mc2, sl, c1_exp, c1_clean, c2_clean,
             t_reset</outputSet>

  <group>
    <formulas>
      <ltl>G( !wasEvent(ep.nfv5) or wasAction(co.mr1) )</ltl>
      <ltl>G( !wasEvent(ep.nfv9) or wasAction(co.mr2) )</ltl>
      <ltl>G( !wasEvent(ep.c2_next) or wasAction(co.c1_exp) )</ltl>
      <ltl>G( !wasEvent(ep.c2_empty) or ( wasAction(co.c1_clean)
           and wasAction(co.c2_clean) and wasAction(co.t_reset) ) )</ltl>
      <ltl>G( !wasEvent(ep.e1) or wasAction(co.sl) )</ltl>
```

```
      <ltl>G( !(wasEvent(ep.c2_empty) or wasAction(co.mc2)) or
X(wasEvent(ep.nfv5) or wasEvent(ep.nfv9) or wasEvent(ep.t)) )</ltl>

      <ltl>G( !wasEvent(ep.c2_next) or ( G(wasEvent(ep.e1)) or
           U(wasEvent(ep.e1), wasEvent(ep.e2)) ) )</ltl>

      <ltl>G( !(wasEvent(ep.nfv5) or wasEvent(ep.nfv9)) or
          X(R( wasEvent(ep.tmpl) and wasAction(co.mc2), !wasEvent(ep.nfv5)
          and !wasEvent(ep.nfv9) and !wasEvent(ep.t) )) )</ltl>

      <ltl>G( !wasEvent(ep.t) or X(R( wasEvent(ep.c2_empty),
           !wasEvent(ep.nfv5) and
           !wasEvent(ep.nfv9) and
           !wasEvent(ep.t) )) )</ltl>
    </formulas>
    <tests>
      <test>
        <input>nfv5, tmpl, tmpl</input>
        <output>mr1, mc1, mc2</output>
        <ptest>nfv5/mr1; tmpl/mc1; tmpl/mc2</ptest>
      </test>
      <test>
        <input>nfv9, tmpl, tmpl</input>
        <output>mr2, mc1, mc2</output>
        <ptest>nfv9/mr2; tmpl/mc1; tmpl/mc2</ptest>
      </test>
      <test>
        <input>nfv5, tmpl, tmpl, nfv9, tmpl, tmpl</input>
        <output>mr1, mc1, mc2, mr2, mc1, mc2</output>
        <ptest>nfv5/mr1; tmpl/mc1; tmpl/mc2; nfv9/mr2; tmpl/mc1;
tmpl/mc2</ptest>
      </test>

      <test>
        <input>t, c2_empty</input>
        <output>c1_clean, c2_clean, t_reset</output>
        <ptest>t; c2_empty/c1_clean,c2_clean,t_reset</ptest>
      </test>
      <test>
        <input>t, c2_next, e2, c2_empty</input>
        <output>c1_exp, c1_clean, c2_clean, t_reset</output>
        <ptest>t; c2_next/c1_exp; e2;
c2_empty/c1_clean,c2_clean,t_reset</ptest>
      </test>

      <test>
        <input>t, c2_next, e1, e2 c2_empty</input>
        <output>c1_exp, sl, c1_clean, c2_clean, t_reset</output>
        <ptest>t; c2_next/c1_exp; e1/sl; e2;
c2_empty/c1_clean,c2_clean,t_reset</ptest>
      </test>
      <test>
        <input>t, c2_next, e1, e2, c2_next, e1, e2, c2_empty</input>
        <output>c1_exp, sl, c1_exp, sl, c1_clean, c2_clean,
t_reset</output>
        <ptest>t; c2_next/c1_exp; e1/sl; e2; c2_next/c1_exp; e1/sl; e2;
c2_empty/c1_clean,c2_clean,t_reset</ptest>
      </test>

      <test>
        <input>t, c2_next, e1, e1, e2, c2_empty</input>
        <output>c1_exp, sl, sl, c1_clean, c2_clean, t_reset</output>
```

```
        <ptest>t; c2_next/c1_exp; e1/sl; e1/sl; e2;
c2_empty/c1_clean,c2_clean,t_reset</ptest>
      </test>
      <test>
        <input>t, c2_next, e1, e1, e2, c2_next, e1, e2, c2_empty</input>
        <output>c1_exp, sl, sl, c1_exp, sl, c1_clean, c2_clean,
t_reset</output>
        <ptest>t; c2_next/c1_exp; e1/sl; e1/sl; e2; c2_next/c1_exp; e1/sl;
            e2; c2_empty/c1_clean,c2_clean,t_reset</ptest>
      </test>
    </tests>

    <negativeTests>
      <ntest>tmpl</ntest>
      <ntest>c2_next</ntest>
      <ntest>e1</ntest>
      <ntest>e2</ntest>

      <ntest>nfv5, nfv9</ntest>
      <ntest>nfv5, t</ntest>
      <ntest>nfv5, c2_next</ntest>
      <ntest>nfv5, e1</ntest>
      <ntest>nfv5, e2</ntest>

      <ntest>nfv9, nfv5</ntest>
      <ntest>nfv9, t</ntest>
      <ntest>nfv9, c2_next</ntest>
      <ntest>nfv9, e1</ntest>
      <ntest>nfv9, e2</ntest>

      <ntest>nfv5, tmpl, nfv5</ntest>
      <ntest>nfv5, tmpl, t</ntest>
      <ntest>nfv5, tmpl, c2_next</ntest>

      <ntest>nfv9, tmpl, nfv9</ntest>
      <ntest>nfv9, tmpl, t</ntest>
      <ntest>nfv9, tmpl, c2_next</ntest>

      <ntest>t, nfv5</ntest>
      <ntest>t, nfv9</ntest>
      <ntest>t, tmpl</ntest>
      <ntest>t, e1</ntest>
      <ntest>t, e2</ntest>

      <ntest>t, c2_next, c2_next</ntest>
      <ntest>t, c2_next, e1, c2_next</ntest>
    </negativeTests>
  </group>
</program>
```

ПРИЛОЖЕНИЕ 4. *UniMod*-МОДЕЛЬ АВТОМАТА УПРАВЛЕНИЯ МОДУЛЕМ *TOP TRAFFIC MONITOR*

```
<model name="Model1">
  <controlledObject class="ControlledObject" name="o1"/>
  <eventProvider class="EventPrivider" name="p1">
    <association clientRole="p1" targetRef="A1"/>
  </eventProvider>
  <rootStateMachine>
    <stateMachineRef name="A1"/>
  </rootStateMachine>
  <stateMachine name="A1">
    <configStore class="DistinguishConfigManager"/>
    <association clientRole="A1" supplierRole="o1" targetRef="o1"/>
    <state name="Top" type="NORMAL">
      <state name="s0" type="INITIAL"/>
      <state name="s1" type="NORMAL"/>
      <state name="s2" type="NORMAL"/>
      <state name="s3" type="NORMAL"/>
      <state name="s4" type="NORMAL"/>
      <state name="s5" type="NORMAL"/>
      <state name="s6" type="NORMAL"/>
    </state>
    <transition Event="nfv9" sourceRef="s0" targetRef="s6">
      <outputAction ident="o1.mr2"/>
    </transition>
    <transition Event="nfv5" sourceRef="s0" targetRef="s6">
      <outputAction ident="o1.mr1"/>
    </transition>
    <transition Event="t" sourceRef="s0" targetRef="s2">
    </transition>
    <transition Event="e1" sourceRef="s1" targetRef="s1">
      <outputAction ident="o1.s1"/>
    </transition>
    <transition Event="e2" sourceRef="s1" targetRef="s2">
    </transition>
    <transition Event="c2_empty" sourceRef="s2" targetRef="s0">
      <outputAction ident="o1.c1_clean"/>
      <outputAction ident="o1.c2_clean"/>
      <outputAction ident="o1.t_reset"/>
    </transition>
    <transition Event="c2_next" sourceRef="s2" targetRef="s1">
      <outputAction ident="o1.c1_exp"/>
    </transition>
    <transition Event="tmpl" sourceRef="s4" targetRef="s0">
      <outputAction ident="o1.mc2"/>
    </transition>
    <transition Event="tmpl" sourceRef="s6" targetRef="s4">
      <outputAction ident="o1.mc1"/>
    </transition>
  </stateMachine>
</model>
```