Министерство образования и науки Российской Федерации
Санкт-Петербургский государственный университет
информационных технологий, механики и оптики

Факультет Информационных технологий и программирования
Направление (специальность) прикладная математика и информатика
Квалификация (степень) бакалавр прикладной математики и информатики
Специализация -----------------------
Кафедра Компьютерных технологий **Группа** 4538

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

Разработка верификатора автоматных программ

Автор квалификационной работы *Егоров К.В.* (подпись)

Руководитель *Шалыто А.А.* (подпись)

К ЗАЩИТЕ ДОПУСТИТЬ
Зав. кафедры *Васильев В.Н.* (подпись)

« » ____________________ 2008 г.

Санкт-Петербург, 2008 г.

Квалификационная работа выполнена с оценкой ________________________________

Дата защиты «   » ______________________ 2008 г.

Секретарь ГАК ________________________________

Листов хранения ______________________________

Чертежей хранения ____________________________

СОДЕРЖАНИЕ

Введение....5
ГЛАВА 1. Автоматное программирование и верификация автоматов....9
1.1. Автоматное программирования....9
1.1.1. *Switch*-технология....9
1.1.2. Инструментальное средство *UniMod*....12
1.2. Традиционная верификация моделей на основе метода *Model Checking*..14
1.2.1. Представление модели....14
1.2.2. Временная логика *LTL*....17
1.2.3. Построение автомата Бюхи по *LTL*-формуле....18
1.2.4. Проверка пустоты пересечения языков....19
1.3. Существующие верификаторы, основанные на автоматном подходе....22
1.3.1. *Spin*....22
1.3.2. *Bogor*....23
1.4. Особенности автоматных программ....23
Выводы по главе 1....24
ГЛАВА 2. Обзор методов верификации автоматных программ....26
2.1. Представление автоматной модели....26
2.2. Язык *LTL* для описания поведения автоматных программам....28
2.3. Особенности верификации автоматных программ....30
2.4. Верификация автоматных программ на многоядерном компьютере....32
2.4.1. Расширение верификатора *Spin* для многоядерного компьютера....33
Выводы по главе 2....34
ГЛАВА 3. Методика верификации автоматных программ....36
3.1. Параллельная верификация автоматных программ....36
3.2. Методика верификации автоматных программ....39
Выводы по главе 3....43
ГЛАВА 4. Описание верификатора....44
4.1. Описание основных классов верификатора....44

4.2. Применение основных классов верификатора..46
4.2.1. Построение модели автоматной программы..46
4.2.2. Построение *LTL*-формул..47
4.2.3. Трансляция *LTL*-формулы в автомат Бюхи..49
4.2.4. Верификация модели автоматной программы..49
Выводы по главе 4..51
ГЛАВА 5. Верификация автоматных программ..53
5.1. Пример верификации модели автоматной программы..53
5.2. Анализ работы верификатора на примере игры «Побег»..56
5.2.1. Анализ модели игрового мира..57
5.2.2. Верификация тактик полицейских машин..61
5.3. Верификация модели банкомата..62
5.3.1. Результат верификации модели банкомата..63
5.3.2. Пример верификации модели банкомата по методике верификации автоматных программ..66
5.4. Анализ эффективности верификации на многоядерном компьютере.......69
Выводы по главе 5..71
Заключение..73
Литература..74

ВВЕДЕНИЕ

С момента появления первых программ людям хотелось уметь проверять их правильность. Причем не просто удостовериться, что программа работает на конечном числе примеров, а уметь формально доказывать, что поведение программы соответствует спецификации. Метод проверки того, что аппаратная или программная система соответствует заявленной спецификации (обладает необходимыми свойствами) и называется **верификацией.** К сожалению, полностью верифицировать систему обычно намного сложнее, чем ее создать. Поэтому в не очень ответственных системах верификация не всегда оправдана, и проще исправлять ошибки по мере их обнаружения во время работы системы. Однако существуют такие системы, в которых ошибки нельзя допускать или они могут обойтись слишком дорого. Например, системы управления транспортом (самолеты, поезда), медицинское оборудование, военные программы, финансовые программы и многие другие области, ошибки в которых могут привести к гибели людей или слишком большим убыткам.

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 катастрофы. Так, одна из самых дорогих катастроф произошла 4 июня 1996 года во время первого запуска новой ракеты-носителя *Ариан-5,* разработанной Европейским космическим агентством. Запуск прошел неудачно – ракета самоуничтожилась через 37 с после старта из-за неверной работы бортового ПО. Эта неудача произошла по причине ошибки преобразования 64-битного числа с плавающей запятой в 16-битное целое. Такое упущение привело к потере 800 миллионов долларов.

К сожалению, отечественное освоение космоса также не обошлось без инцидентов, произошедших по вине ПО. «В начале 70 годов прошлого столетия между СССР и США разгоралась космическая гонка, кто первым совершит

мягкую посадку на Марс. 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»[1]. Первый полет прошел неудачно – на траекторию полета к Марсу станция не перешла по причине ввода в вычислительную бортовую машину ошибочного значения времени запуска двигателя. Из-за ошибки в разряде, двигатель должен был запуститься не через несколько десятков минут, как предусматривала программа полета, а через полторы сотни часов. Второй полет также не принес никакой научной ценности из-за ошибки бортового компьютера. Хотя аппарат и смог добраться до Марса, однако мягкая посадка не была осуществлена. Последняя неудача была уже в 1989 году, когда станция *Фобос-2* не смогла выполнить поставленную задачу. Наиболее вероятная причина отказа: «одновременное «зависание» двух каналов бортовой ЦВМ и, как следствие, потеря ориентации и закрутка аппарата с остаточными на этапах разворотов угловыми скоростями»[2].

Одним из основных методов проверки программы на наличие ошибок является тестирование. На практике оно применяется в большинстве случаев. Однако «тестирование позволяет показать наличие ошибок, но не их отсутствие»[3]. При таком подходе к проверке, можно удостовериться в правильности работы программы только при определенном ее поведении или каком-то конечном числе входных данных. Однако существуют ошибки, которые могут появляться крайне редко, особенно при параллельном исполнении программы. Поэтому, для того чтобы исключить возможность их появления, требуется рассмотреть все возможные варианты поведения системы, что невозможно.

Формальная верификация – метод проверки того, что программа

1 http://stump-workshop.blogspot.com/2007/12/blog-post_13.html

2 Лантратов К. На Марс! // Новости космонавтики, №21, 1996, http://martiantime.narod.ru/History/lant2.htm

3 Э. Дейкстра, 1970г.

соответствует спецификации, который основан на строгих математических принципах. При таком подходе используется логический формализм и математическое описание модели. Автоматическая верификация – это формальная верификация, проводимая машиной без вмешательства человека. Далее в работе под словом «верификация» будет пониматься автоматическая верификация.

Процесс верификации, как правило, делится на три части. Моделирование программы – преобразование программы в формальную модель для последующей верификации. Спецификация – формальная запись свойств, которые требуется проверить. Собственно верификация. Все эти части связаны между собой – алгоритмы верификации зависят от способа построения модели и способа записи свойств системы.

Цель настоящей работы состоит в **создании верификатора для модели автоматных программ, созданных при помощи инструментального средства *UniMod*** [1, 2]. Особенности этого класса программ позволяют избежать первой части верификации – построение модели, так как каждая такая программа уже представляет собой модель, пригодную для проверки определенных утверждений о ней без ее модификации. Требования к программе будем формулировать в виде формул темпоральной логики *LTL* (*Linear Time Logic*).

В ходе работы **создан верификатор**, который не использует уже существующие верификаторы, применение которых связано с преобразованием модели автоматной программы в модель, описываемую на языке верификатора. При применении такого языка после доказательства невыполнимости утверждения об автомате (построении контрпримера) пришлось бы совершать обратное преобразование в автоматную программу.

Для создания нового верификатора необходимо разработать собственное представление модели, которая не использует внутреннее представление модели в инструментальном средстве *UniMod*. Это было сделано для того,

чтобы и при создании новых средств для написания программ в рамках автоматного программирования можно было бы применять данный верификатор. При этом необходимо рассмотреть применимость для автоматных программ алгоритмов верификации при использовании логики *LTL*. Это уже рассматривалось в работе [3], но для уже имеющихся верификаторов.

В связи с ростом числа ядер персональных компьютеров, возникает задача **распараллеливания процесса верификации автоматных программ**. Поэтому еще одна задача работы – создание верификатора, который мог бы равномерно распределять нагрузку на ядра и работал бы эффективнее на многоядерном компьютере по сравнению с одноядерным.

ГЛАВА 1. АВТОМАТНОЕ ПРОГРАММИРОВАНИЕ И ВЕРИФИКАЦИЯ АВТОМАТОВ

В первой части данной главы рассматривается понятие «автоматное программирование», где оно применяется и почему оно удобно для верификации. Вторая часть посвящена формальному способу описания модели программы и ее поведения при помощи темпоральной логики.

1.1. Автоматное программирования

Автоматное программирование – это парадигма программирования, при использовании которой программа или ее фрагмент осмысливается как система автоматизированных объектов управления. Особенность такого программирования заключается в явном выделении состояний[4] 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 была предложена в работе [1] и является удобной при написании определенного класса программ и их последующей верификации.

Данный подход отличается в лучшую сторону от традиционного способа программирования тем, что позволяет явно представлять состояния системы. Это позволяет лучше анализировать работу программ, вносить в них изменения, осуществлять отладку и поиск ошибок. Это обусловлено тем, что человеку свойственно мыслить в рамках автоматной модели, где, например, вложенность автоматов представляет собой разбиение логики системы на уровни. Наличие явно выраженных состояний в программах позволяет упростить процесс их верификации.

1.1.1. *Switch*-технология

Switch-технология – технология разработки программного обеспечения,

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

которая основана на использовании структуры систем логического управления и методе программирования с явным выделением состояний. При таком подходе к созданию программ выделяются три типа объектов: поставщики событий, система управления и объекты управления.

Система управления представляет собой конечный автомат или систему взаимодействующих автоматов. Автомат – это множество состояний и переходов между ними. Каждый переход помечен событием, при котором он может осуществиться, и условием, выполнимость которого требуется для перехода. Поставщики событий генерируют события, а система управления по каждому событию может совершать переход, запрашивая определенные свойства у объектов управления для проверки условия перехода. Такая система называется реагирующей[5] или событийной.

При программной реализации поставщики событий и объекты управления могут быть связаны между собой. На рис. 1 схематично представлена структура взаимодействия компонент при программировании в рамках *Switch*-технологии. Стоит отметить, что в реальной программе может быть создано несколько моделей, образующих мультиагентную систему, которые имеют общие поставщики событий или объекты управления.

При поступлении события от поставщика автомат, находящийся в каком-либо состоянии, или автомат, вложенный в текущее состояние родительского автомата, может совершить переход, помеченный данным событием. При этом переход осуществляется только при выполнении определенных условий, данные для которых запрашиваются у объектов управления. При переходе по событию, а также при входе в состояние, автомат может воздействовать на объекты управления, выполняя набор определенных действий.

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

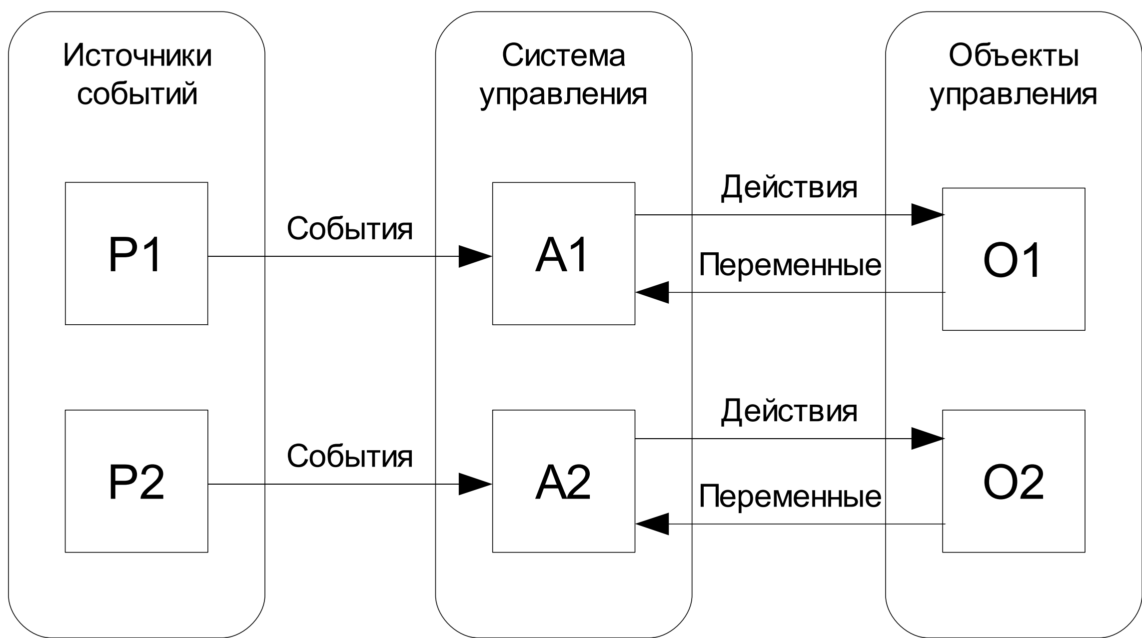

Рис. 1. Схема работы реагирующей системы

На рис. 2 изображен автомат, содержащий одно начальное и два состояния с переходами между ними. 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*. Причем, при первом переходе вызывается действие *z1*, а при втором – действие *z2* того же объекта управления.

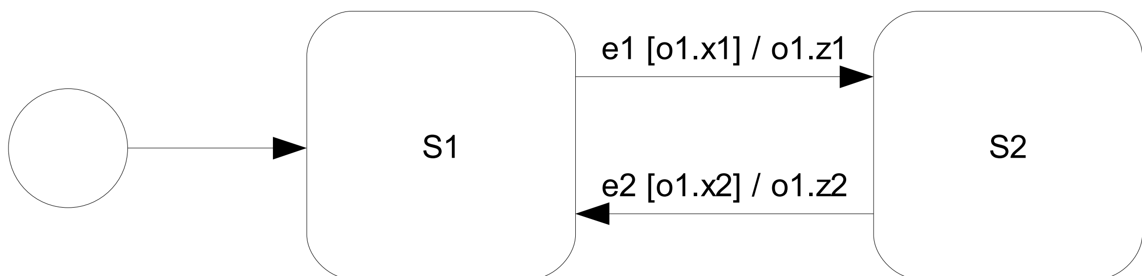

Рис. 2. Пример автомата

При таком подходе к написанию программ они могут достаточно просто верифицироваться, так как для них не требуется строить другие модели с конечным числом состояний. При верификации программ на языках типа *Java* или *C++,* написанных традиционным путем (без явного выделения состояний),

потребовалось бы преобразовать программу к виду, понятному существующим верификаторам. При этом были бы утеряны определенные данные и связи в программе, так как пришлось бы перейти на другой уровень абстракции. Автоматная модель, которая строится при создании системы в рамках *Switch*-технологии, может верифицироваться без изменений или с изменениями, которые не приводят к потере данных о ней.

1.1.2. Инструментальное средство *UniMod*

UniMod – инструментальное средство, обеспечивающее визуальное проектирование автоматных программ на основе *Switch*-технологии. Это позволяет вынести практически всю логику программы в автоматы, а остальные классы разбить на два типа: поставщики событий и объекты управления. *UniMod* написан на языке программирования *Java* и встраивается в среду разработки *Eclipse* как дополнительный модуль (*plug-in*) [2].

UniMod предоставляет возможность написания собственных поставщиков событий и объектов управления, а конечные автоматы и их связь с указанными компонентами генерируется по визуальной модели. Поставщики событий и объекты управления в инструментальном средстве *UniMod* – это классы, написанные на языке программирования *Java*. Для того чтобы класс стал поставщиком событий, достаточно унаследовать его от интерфейса `com.evelopers.unimod.runtime.EventProvider`. При этом каждая статическая переменная типа `String` будет являться событием для конечного автомата. Поставщики событий принято обозначать *p1*, *p2* и т.д., а события – *e1*, *e2* и т.д.

Для того чтобы класс стал объектом управления, его следует наследовать от интерфейса `com.evelopers.unimod.runtime.ControlledObject`. Объекты управления принято называть *o1*, *o2* и т.д., методы-действия – *z1*, *z2* и т.д., а методы, возвращающие значения (входящие переменные) – *x1*, *x2* и т.д. Такая нотация была принята для того, чтобы на переходах конечного автомата

можно было отразить сложную логику, так как, применяя «говорящие» названия переменных и других символов, все условия и действия не смогли бы разместиться на переходах при визуальном отображении. Состояния автомата можно называть *s1*, *s2*, и т.д., но для них целесообразно применять и «говорящие» названия.

Особенность автоматов, разрабатываемых с помощью указанного инструментального средства, состоит в том, что у каждого автомата должно быть только одно начальное состояние и одно завершающее или завершающего состояния может не быть вообще. У каждого автомата могут быть вложенные автоматы, что достигается вложением одних автоматов в состояния других. При этом вложенный автомат может совершать переход в том и только в том случае, если родительский автомат находится в состоянии, в которое вложен ребенок. Вся иерархическая система автоматов обрабатывает события последовательно – в каждый момент времени системы может обрабатываться только одно событие. Также важно то, что при поступлении события, его может обработать только один автомат. Таким образом, за один шаг работы система автоматов может совершить только один переход или не совершить вообще. Во втором случае ни один автомат не имеет возможности обработать событие или не имеет допустимых переходов по нему.

Как уже отмечалось выше, инструментальное средство *UniMod* позволяет визуально отображать взаимодействия между поставщиками событий, конечными автоматами и объектами управления. Также имеется возможность создания иерархии конечных автоматов. Данное средство позволяет по построенной модели генерировать *Java*-код или *XML*-файл. Первый подход является компилятивным, а второй – интерпретационным. Поведение автоматной программы от подхода не зависит, и выбор первого или второго метода остается за программистом.

В данной работе для верификации автоматной модели используется *XML*-файл, а внутренне представление модели в инструментальном средстве

UniMod не требуются. Таким образом, верификатору нет необходимости ничего знать о *UniMod,* кроме *XML*-файла. Тем не менее, реакция иерархии автоматов на поступающие события была реализована как это принято в инструментальном средстве *UniMod.* Отказ от использования внутреннего представления модели программы данного инструментального средства, позволит в дальнейшем верифицировать автоматные программы, написанные не только при помощи указанного инструментального средства, но и использующие любые другие средства проектирования, реализующие *Switch*-технологию.

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

Для верификации модели, в первую очередь, необходимо определиться, что такое модель. Это может быть представление о программе в определенный момент или значение переменных, регистров, или состояние файлов…. Также при верификации обычной программы, возникает проблема уровня разбиения: строка кода на языке программирования высокого уровня или строка на ассемблере, например.

В настоящей работе будет рассматриваться модель в виде конечного автомата, причем, как отмечалось выше, не будет требоваться специальные преобразования над исходной системой, для того чтобы получить указанную модель. Таким образом, в данном разделе рассмотрим только теоретическую часть традиционной верификации модели, а применять ее для автоматного программирования можно будет с изменениями, о которых пойдет речь в следующей главе.

1.2.1. Представление модели

В рамках указанного подхода модель программы представляется как множество состояний, причем система в любой момент времени может быть

только в одном состоянии. Семантика состояния – глобальный снимок всей системы или мгновенное описание системы. Каждое состояние имеет конечное описание. Имеется выделенное начальное состояние. Особый интерес в рамках настоящей работы представляют реагирующие системы, 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. Таким образом, система в общем случае формально может быть представлена в виде графа переходов (S, T, s_0, L, F), который называется *моделью Крипке* [4].

- S – конечное множество состояний;
- $T \subseteq S \times S$ – множество переходов;
- s_0 – начальное состояние;
- $L: S \rightarrow 2^{AP}$, где AP – множество атомарных высказываний;
- $F \subseteq S$ – множество допускающих состояний.

Тогда путь в этом графе $\pi = s_0, s_1, s_2, \ldots s_n, \ldots$, для которого выполнено $T(s_{i-1}, s_i)$, будет последовательностью вычислений системы. Путь будет *допускающим*, если существует состояние из множества F, такое что оно встречается бесконечно часто.

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

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

- $F \subseteq S$ – 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 качества 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 $e = L(s_i)$.

При представлении модели в качестве автомата Бюхи самое простое, что можно проверить, это достижимость «хорошего» состояния или недостижимость «плохого». Состояние достижимо, если существует путь из начального состояния в него. Система переходов автомата описывает все состояния и переходы системы. Поэтому можно формулировать **условие корректности, как достижимость состояния, а недостижимость – как некорректность**. Анализ достижимости проверяется обходом в глубину или в ширину и не представляет особого интереса для данной работы.

Анализ достижимости – это простое условие для проверки. Хочется уметь проверять более сложные условия, такие как, например, «устройство готово к работе бесконечно часто», «после отправки события, оно будет получено» или «каждый процесс, который хочет войти в критическую секцию, рано или поздно войдет в нее». Для высказываний такого типа требуется более богатый формализм.

Темпоральные логики как раз и являются тем формализмом, который описывает последовательность переходов между состояниями реагирующей 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.

1.2.2. Временная логика *LTL*

Логика линейного времени *LTL* (*Linear Time Logic*), является подмножеством более выразительной логики CTL^*, о которой можно прочесть в работах [4, 5]. Синтаксис логики линейного времени включает в себя множество пропозициональных переменных *Prop*, булевы связки ($\neg$, $\vee$, $\wedge$) и темпоральные операторы. Причем, пропозициональные переменные интерпретируются как $I: Prop \rightarrow \{True, False\}$.

Логика линейного времени расширяет классическую логику, добавляя временные операторы, для того чтобы можно было судить о разных моментах времени. В этой логике время линейно и изоморфно натуральным числам. Модель можно представить как последовательность состояний, индексированных натуральными числами [6]. Пропозициональные переменные могут быть истинными или ложными в каждый момент времени (рис. 3).

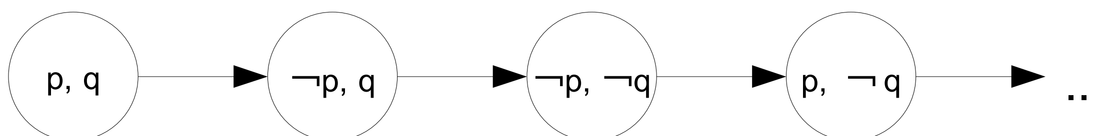

Рис. 3. Последовательность состояний в логике LTL

Для составления утверждений о времени событий применяются следующие темпоральные операторы:

- **X** (ne**X**t) – «*Xp*» – в следующий момент выполнено *p*;
- **F** (in the **F**uture) – «*Fp*» – в некоторый момент в будущем будет выполнено *p*;
- **G** (**G**lobally in the future) – «*Gp*» – всегда в будущем выполняется *p*;
- **U** (**U**ntil) – «*pUq*» – существует состояние, в котором выполнено *q* и до него во всех предыдущих выполняется *p*;
- **R** (**R**elease) – «*pRq*» – либо во всех состояниях выполняется *q*, либо

существует состояние, в котором выполняется p, а во всех предыдущих выполнено q.

Множество *LTL*-формул таково:

- пропозициональные переменные *Prop*;
- *True, False*
- φ и ψ – формулы, то
  - $\neg\varphi$, $\varphi\vee\psi$, $\varphi\wedge\psi$ – формулы;
  - $X\varphi$, $F\varphi$, $G\varphi$, $\varphi U\psi$, $\varphi R\psi$ – формулы.

Отличие логики *LTL* от логики *CTL** состоит в том, что в последней присутствуют кванторы пути $\forall$ и $\exists$, которые говорят о любом пути и о существовании пути соответственно. Логика линейного времени говорит о всех путях, и квантор $\forall$ опущен. Таким образом, логика *LTL* предполагает, что некоторое утверждение будет выполняться для всех путей. Поэтому можно строить доказательство от противного и проверять существование пути, на котором будет выполняться отрицание данной формулы. Если такой путь не будет найден, то формула выполнима.

1.2.3. Построение автомата Бюхи по *LTL*-формуле

По *LTL*-формуле можно построить автомат Бюхи. К сожалению, данная задача *PSPACE*-полна. Рассмотрим алгоритм, предложенный в работе [7], который обеспечивает трансляцию *LTL*-формулы в автомат Бюхи. Этот алгоритм применяется в некоторых верификаторах с незначительными изменениями.

Перед применением указанного алгоритма *LTL*-формула приводится в *негативную нормальную формулу* [4], в которой отрицание применяется только к пропозициональным переменным. Также можно считать, что в формуле 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 основывается на построение графа специального вида, каждая вершина в котором соответствует подформуле исходной формулы. Построение такого графа заключается в последовательном расщеплении вершин. Вместо одной вершины для операторов $\vee$, *U*, *R* может быть создано две вершины, а для остальных – одна [8].

Расщепление вершины для темпоральных операторов *U* и *R* основывается на тождествах $\varphi\ \mathrm{U}\ \psi \equiv \psi \vee (\varphi \wedge \mathrm{X}(\varphi\ \mathrm{U}\ \psi)$ и $\varphi\ \mathrm{R}\ \psi \equiv \psi \wedge (\varphi \vee \mathrm{X}(\varphi\ \mathrm{R}\ \psi)$.

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 $\varphi\ \mathrm{U}\ \psi$. Особенность такого автомата состоит в том, что для допущения некоторого слова, цикл, соответствующий бесконечному суффиксу, он должен проходить по состояниям из каждого допускающего множества. Это вносит некоторые изменения в процесс верификации [9].

Существует множество модификаций данного алгоритма. Они все основываются на преобразовании формулы перед трансляцией с целью уменьшения числа состояний. Например, в работе [10] приводится ряд преобразований, которые позволяют получить автомат Бюхи меньшего размера.

1.2.4. Проверка пустоты пересечения языков

Для доказательства невыполнимости некоторой формулы логики линейного времени на автомате Бюхи можно проверить, что пересечение языков верифицируемого автомата Бюхи и языка отрицания *LTL*-формулы пусто. Для

этого требуется доказать, что язык автомата пересечения пуст. Из сказанного следует, что алгоритм верификации может быть следующим: строится автомат Бюхи для верифицируемой программы; по отрицанию формулы *LTL* строим автомат Бюхи, затем строим пересечение автоматов и после этого проверяем, что он не допускает ни одного слова.

В связи с тем, что рассматриваются бесконечные слова, то, как доказано в работе [4], для пустоты пересечения достаточно доказать, что ни одно допускающее состояние не принадлежит сильной компоненте связанности, которая достижима из начального состояния. Или, что тоже самое, не существует цикла, проходящего через допускающее состояние. Таким образом, при нахождении цикла, достижимого из начального состояния, будет построен контрпример – путь, на котором не выполняется *LTL*-формула.

Для обнаружения сильно связанных компонент, может быть применен алгоритм Тарьяна[6], но при верификации обычно применяют двойной обход в глубину [4], преимущество которого состоит в том, что для его реализации не требуется построение пересечения автоматов целиком. При его реализации можно строить состояния пересечения автоматов по мере их достижения. Это дает преимущество на больших моделях.

Общая идея алгоритма такова: обходим в глубину автомат пересечения языков, при достижении допускающего состояния для проверки достижимости самого себя запускаем второй обход в глубину из данного состояния. Если оказалось, что допускающее состояние достижимо из самого себя, то цикл найден, и следовательно, исходная *LTL*-формула не выполняется на автомате Бюхи, представляющем модель программы.

Приведем рекурсивный алгоритм двойного обхода в глубину на псевдокоде из работы [4].

```
procedure emptiness
    for all q0 ∈ Q0 do
```

6 http://en.wikipedia.org/wiki/Tarjan's_strongly_connected_components_algorithm

```
        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);
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 *DFS*, то цикл найден. Тогда стек первого *DFS* содержит конечный префикс контрпримера, а второй обнаружил цикл, который служит бесконечным суффиксом. Таким образом, язык пересечения автоматов Бюхи не пуст.

Доказательство алгоритма приведено в работе [4].

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

пересечение автоматов целиком.

1.3. Существующие верификаторы, основанные на автоматном подходе

Существует некоторое число верификаторов, которые умеют проверять выполнимость или невыполнимость *LTL*-формулы на автомате Бюхи. Для данной работы интерес представляют те из них, которые для верификации используют алгоритм, указанный в разд. 1.2.4. Рассмотрим верификаторы *Spin* и *Bogor*, так как их применение было изучено для верификации автоматных программ в работе [3].

1.3.1. Spin

Верификатор *Spin* позволяет верифицировать программы, написанные на высокоуровневом языке *Promela*. Требования к программе записываются на языке *LTL*.

В разд. 1.2 работы [3] предложен метод проверки автоматных программ, использующий данный верификатор. При этом был разработан инструмент, позволяющий по автоматной программе строить модель на языке *Promela*, выполнять автоматическое преобразование записанных вручную на языке *Promela* проверяемых требований в автомат Бюхи, автоматически выполнять верификацию полученной модели и производить построение контрпримера по модели.

Особенность модели на языке *Promela*, полученной в результате преобразования автоматной модели, состоит в том, что она считает поставщики событий и объекты управления внешней средой. При этом она не учитывает результат действий объектов управления и считает, что поставщики событий могут посылать любые события. В дальнейшем авторы указанной работы предполагают устранить этот недостаток.

1.3.2. Bogor

Верификатор *Bogor* предоставляет возможность проверять модели, написанные на языке *BIR* (*Bogor Input Representation*). Требования к программе записываются на языке *LTL*. Во входной язык верификатора *Bogor* можно добавлять новые типы и абстракции, а сам верификатор разделен на модули, реализующие различные аспекты верификации (алгоритм обхода, кодирование состояния и т.д.). Эти модули достаточно просто можно заменять, и за счет этого изменять логику верификатора, не переписывая его заново. Таким образом, *Bogor* представляет гибкий верификатор, который достаточно легко настраивается для верификации конкретного типа задач [3].

В работе [3] язык *BIR* был расширен. Это позволило верификатору работать с *UniMod*-моделью как с объектом нового типа. Созданному средству на вход подается *XML*-файл и *LTL*-формула, записанная на входном языке верификатора. На выход верификатор выдает текстовый список действий, которые привели к нарушению требований, или сообщение об удачном завершении верификации [3].

При использовании данного верификатора, автоматная модель (также как и при применении верификатора *Spin*) считает поставщики событий и объекты управления внешней средой и не учитывает их внутреннюю реализацию.

1.4. Особенности автоматных программ

В разд. 1.2 был рассмотрен традиционный метод верификации автомата Бюхи, который может быть построен из модели Крипке. Однако автоматная программа не является ни моделью Крипке, ни автоматом Бюхи.

Для выделения модели из автоматной программы, пригодной для верификации, необходимо пометить переходы и (или) состояния автомата пропозициональными переменными. Однако семантика таких переменных должна быть как-то связана с автоматной программой, иначе ценность таких утверждений будет отсутствовать.

Если рассматривать модель автоматной программы как модель Крипке, то в каждом состоянии такой модели должны выполняться определенные пропозициональные переменные. Выполнимость или невыполнимость таких переменных не зависит от переходов. Это сильно сужает выразительность утверждений о модели автоматной программы, так как в таком случае можно утверждать только о неизменных свойствах состояния. Таковыми являются, например, вызванные действия или название состояния.

Если считать модель автоматной программы автоматом Бюхи, то потребуются пропозициональные переменные для переходов автомата. Это также внесет ограничение в модель, так как можно будет делать высказывания только о переходах.

В следующей главе будет рассмотрена модель автоматной программы, которая была предложена главе 1 работы [3] и является, в некотором смысле, объединением модели Крипке и автомата Бюхи. Такая модель позволяет составлять утверждения как о переходах автомата, так и о его состояниях. При этом пропозициональные переменные не указываются явно, а являются высказываниями о переходах или состояниях, которые выполняются всегда независимо от последовательности переходов автомата.

Выводы по главе 1

В первой части главы рассматривались такие понятия как автоматное программирование и *Switch*-технология. Они реализованы в инструментальном средстве *UniMod*.

Вторая часть посвящена верификации автоматов Бюхи, утверждения к которым записываются на языке *LTL*. Описаны автоматы Бюхи, логика *LTL* и приведен алгоритм верификации.

В третьей части рассматривались существующие верификаторы, применение которых для верификации модели автоматных программ было уже изучено.

Последняя часть показывает различия между моделью Крипке, автоматом Бюхи и моделью автоматной программы. Это приводит к необходимости анализа применимости к автоматной программе способа верификации, изложенного в разд. 1.2.

ГЛАВА 2. ОБЗОР МЕТОДОВ ВЕРИФИКАЦИИ АВТОМАТНЫХ ПРОГРАММ

В данном разделе рассматриваются алгоритмы верификации автоматных программ. Как уже отмечалось в предыдущей главе, один из подходов в верификации автоматной программы – это преобразование ее модели во входной язык одного из существующих верификаторов и последующее применение его для доказательства выполнимости или невыполнимости формулы.

Как показывает практика, программы, написанные при помощи инструментального средства *UniMod*, имеют модель из небольшого числа состояний. Поэтому применения «тяжеловесных» верификаторов для анализа такой модели может быть не оправданно, тем более что при их применении в настоящее время не учитываются результаты действий объектов управления. При верификации считается, что они могут возвращать все что угодно, а также поставщики событий в любой момент времени могут послать любое событие. Поэтому, на взгляд автора, имеет смысл только проверка достаточно простых утверждений; например, «до события *e1* всегда обрабатывалось событие *e2*» или «автомат вызывает действие *o1.z1* после обработки события *e1*», так как, даже если верификатор скажет, что данное утверждение не выполняется, это, скорее всего, не будет свидетельствовать об ошибке в модели автоматной программы из-за достаточно существенных допущений.

2.1. Представление автоматной модели

В настоящей работе для построения модели используется *XML*-файл, который создается при помощи инструментального средства *UniMod*. Из этого файла разработанный автором верификатор извлекает информацию о связях между поставщиками событий, объектами управления и конечными автоматами. Рассмотрим, в чем эти три типа компонент отличаются и в чем их

сходство с *UniMod*-объектами.

В работе [3] предполагалось, что поставщики событий рассматриваются как объекты, которые на каждом шаге возвращают любое событие и никак не связаны с объектами управления. 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 состоянии системы.

Объекты управления рассматриваются как внешняя среда, на которую воздействует конечные автоматы посредством вызова определенных действий при переходах. Только, в отличие от реагирующей системы из *UniMod*-проекта, не учитывается результат действий таких вызовов. При запросе входных переменных в представлении системы верификатором объекты управления возвращают все что угодно. Считается, что условия на переходах в разные моменты времени могут выполняться или не выполняться независимо от результата предыдущих переходов.

Такой подход, когда поставщики событий и объекты управления рассматриваются как внешняя среда [3], которая не зависит от предыдущих действий и переходов, ограничивает выразительную возможность любого верификатора. При этом не всегда найденный контрпример невыполнимости некоторой формулы будет свидетельствовать об ошибке в автоматной программе, а чаще только об ошибке в представлении модели верификатором. Однако создание автоматных программ, в которых «хитрая» логика не выносится за пределы автоматов в объекты управления, позволяет делать утверждения о модели более осмысленными и частично устраняет приведенный недостаток модели используемой верификатором.

Такая модель реагирующей системы отличается от той, которая была описана в разд. 1.2.1, тем, что там в качестве модели, рассматривался автомат Бюхи, полученный из модели Крипке, в котором все переходы, которые ведут в состояние, оказывались помечены одинаковыми предикатами. В автоматной

программе в зависимости от перехода, по которому попали в данное состояние, предикаты будут отличаться. Однако рассмотренный алгоритм верификации был представлен для автомата Бюхи (разд. 1.2.4) и не учитывал такую особенность. Поэтому он может быть применен и для данной модели.

2.2. Язык *LTL* для описания поведения автоматных программам

Для верификации автоматных программ требуется определить какой язык темпоральной логики использовать. Как уже отмечалось ранее, в настоящей работе используется язык *LTL*. Однако такой язык ничего не говорит о свойствах автоматной программы. Он может только сказать о выполнимости или невыполнимости фиксированных предикатов в какой-нибудь момент в будущем. Для автомата Бюхи предикаты – это утверждения, которые верны или неверны для конкретного перехода из одного состояния в другое, а *LTL*-формула говорит о том, что некоторое утверждение верно для всех переходов.

В модели автоматной программы 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 из разд. 1.2.4, на утверждение о переходе накладывается ограничение, что оно может говорить только о текущем переходе и должно выполняться всегда, независимо от предыдущих переходов. Тогда такого рода предикат ничем не будет отличаться от предикатов, помечающих переходы в автомате Бюхи. Так как для перехода фиксированы начальное и конечное состояния, то в качестве предиката также может быть использовано утверждение о предыдущем и текущем состояниях.

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 работе [3], 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 верификатор, разработанный в настоящей работе, добавлена возможность создавать собственные предикаты. 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 одну функцию на языке *Java*, которой доступна информация о переходе. Таким образом, можно легко проверять такого рода утверждения, не разбивая переход на несколько частей или используя комбинацию предопределенных предикатов.

При этом утверждения об автоматной программе записываются обычной строкой, например, «*G(wasEvent(p1.e1))*» или «*F(isInState(A1.s1))*». Таким образом, синтаксис *LTL*-формул остался таким же как в разд. 1.2.2. Это позволяет записывать утверждения о программе человеку, который знает только

синтаксис и семантику языка *LTL*.

2.3. Особенности верификации автоматных программ

Процесс верификации, описанный в разд. 1.2, заключался в следующем: модель программы представляли в виде автомата Бюхи, по *LTL*-формуле строилось ее отрицание; новую формулу преобразовывали в автомат Бюхи; строили пересечение двух автоматов; доказывали, что язык пересечения автоматов пуст. Если найден контрпример, то заданная формула не выполняется, иначе она верна. Алгоритм верификации автоматных программ практически такой же. Однако требует пояснения интерпретация модели, описанной в разд. 2.1 как автомат Бюхи, а также переходов между состояниями, когда рассматривается не один автомат, а система вложенных автоматов.

Особенность *LTL*-формул описана в разд. 2.2, в котором подробно описаны предикаты, применяемые для верификации автоматных программ. Таким образом, модель автоматной программы можно интерпретировать в качестве автомата Бюхи. Когда в этом автомате совершается переход, то предикаты могут выполняться или не выполняться в зависимости от перехода.

Опишем алгоритм верификации **одного автомата**:

1. Модель программы представляется в виде автомата Бюхи.
2. Строится отрицание *LTL*-формулы.
3. По отрицанию *LTL*-формулы строится автомат Бюхи с переходами, помеченными специально введенными предикатами.
4. Производится двойной обход в глубину неявного пересечения двух автоматов Бюхи. Для построения пересечения выполняются следующие действия:
   4.1. Перебираются все переходы верифицируемого автомата.
   4.2. Перебираются все возможные переходы автомата, построенного по *LTL*-формуле, для перехода верифицируемого автомата, полученного на шаге 4.1.

4.3. Состояние помечается допускающим, если состояние автомата, построенного по *LTL*-формуле, является допускающим[7].

Построение автомата Бюхи по *LTL*-формуле в настоящей работе было реализовано двумя способами. Первый реализовал алгоритм из разд. 1.2.2, а второй – при помощи транслятора *LTL2BA* [11]. Данное средство использует алгоритм трансляции *LTL*-формулы в автомат Бюхи, описанный в работе [12]. Транслятор *LTL2BA* принимает на вход *LTL*-формулу, используя синтаксис верификатора *Spin*. 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 известного транслятора.

Для верификации **системы вложенных автоматов** применяется такой же алгоритм, только в качестве верифицируемого автомата строится автомат, состояния которого содержат информацию о состояниях всех автоматов иерархической системы. Каждое состояние нового автомата представляет собой дерево, структура которого совпадает со структурой системы вложенных автоматов. В узлах дерева находятся состояния, в которых размещены соответствующие конечные автоматы. Узел может быть активным, если соответствующий автомат может обрабатывать события, и неактивным, если автомат не может обрабатывать события. Переходы из такого состояния могут совершаться по переходам одного из активных внутренних состояний. При этом активные и неактивные состояния могут вычисляться следующим образом:

1. Состояние активно, если состояние, в которое вложен автомат, является родителем и активно.
2. Состояние неактивно, если состояние, в которое вложен автомат, не является родителем или неактивно.

При таком подходе, если состояние неактивно, то все вложенные в него состояния также неактивны. Это позволяет строить переходы из такого

---

7 Состояния в автомате Бюхи, построенного по *LTL*-формуле, помечаются допускающими согласно работе [7].

сложного состояния, не просматривая все узлы дерева, а только активные.

Такая структура является неявным пересечением автоматов. Однако преимущество этого алгоритма состоит в том, что он позволяет строить пересечение системы автоматов не сразу, а по мере их посещения при обходе в глубину. Это позволяет обнаружить контрпример до того, как будет построено полное пересечение автоматов.

От пересечения автоматов нельзя отказаться, так как если требуется делать утверждения о состоянии системы автоматов в целом, то необходимо как-то представлять такое глобальное состояние. Поэтому, если требуется доказывать свойство конкретного автомата, то верификатор, разрабатываемый в настоящей работе, предоставляет возможность как проверять утверждения об отдельном автомате иерархической системы, так и обо всей системе в целом.

Не всегда утверждение об одном автомате имеет тот же результат, что и утверждение о системе вложенных автоматов. Например, утверждение «$G(isInState(A2.s1) \rightarrow X(isInState(A2.s2)))$» (Если автомат *A2* находится в состояние *s1*, то следующим состоянием будет *s2*) вполне может быть истинным для автомата *A2*. Однако если автомат *A2* вложен в *A1*, то это утверждение не будет выполняться для такой системы автоматов, так как если автомат *A2* находится в состоянии *s1*, то следующий переход может совершить автомат *A1* и тогда утверждение не будет выполнено.

2.4. Верификация автоматных программ на многоядерном компьютере

Существует несколько способов распараллеливания некоторого вычислительного процесса. Как правило, они делятся на распределенные и параллельные вычисления. Распределенные вычисления характеризуются вычислениями на нескольких отдельных машинах, соединенных сетью. Недостаток такого подхода состоит в том, что скорость обмена информации по сети очень низка по сравнению с передачей данных внутри одного компьютера.

Поэтому требуется разбивать задачу на независимые подзадачи, которым бы требовался малый объем переданных данных для параллельной работы.

Существует реализация верификатора *Spin*, которая работает на нескольких машинах [13, 14] а также работы, предлагающие альтернативные алгоритмы для распределенной верификации [15]. Создание такой версии верификаторов было обусловлено распределением памяти между несколькими компьютерами, что приводило к значительным сложностям. В результате эффективность проверки такой же формулы на той же модели была эффективнее на одной машине, чем распределенная версия.

На данной момент проблема с памятью не является существенной, особенно с появлением 64-битных процессоров и винчестеров значительного объема. Благодаря этому предпочтение отдается параллельным вычислениям, подразумевая под ними выполнимость на одном компьютере. Поэтому в 2007 году появилась версия верификатора *Spin*, предназначенная для многоядерной системы [16]. В этой работе приводятся сравнение скоростей передачи данных с использованием сети, многопроцессорной системы и многоядерного компьютера, что является основным фактором предпочтения версии верификатора для многоядерного компьютера перед остальными версиями.

2.4.1. Расширение верификатора *Spin* для многоядерного компьютера

Работа [16] состояла в расширении функциональных возможностей верификатора *Spin* для работы на многоядерном компьютере, причем авторы этой работы хотели внести как можно меньше изменений в однопоточную версию уже существующего верификатора. Также там была поставлена задача о равномерной нагрузке процессоров. Было предложено два алгоритма для распараллеливания процесса двойного обхода в глубину.

Первый алгоритм заключается в следующем. Первый поток обходит состояния и при обнаружении допустимых, заносит их в очередь. Второй поток извлекает допустимые состояния из очереди и запускает из них вложенный

обход в глубину для обнаружения циклов. При этом новые состояния вычисляются для каждого потока в отдельности. Преимущество такого подхода состоит в том, что он не требует блокировок. Однако его сложно расширить на большее число ядер, чем два.

Идея второго алгоритма основана на том, что ядра (потоки) связаны в логическое кольцо, и только соседи имеют общую рабочую очередь. Каждый поток может передавать работу только соседу справа, используя для этого их общую рабочую очередь. Для равномерной нагрузки на процессоры, каждое ядро может строить дерево обхода в глубину только определенного размера. Таким образом, когда глубина дерева превышает некоторое значение d, поток записывает новое состояние в рабочую очередь, и соседний поток будет строить свое дерево обхода в глубину, начиная с текущего состояния.

При обнаружении ошибки возникает проблема построения контрпримера. Это объясняется тем, что каждый поток знает только о локальном стеке, который начинается не из стартовой вершины. В работе [17] приводится способ решения данной проблемы. Общая идея такова: предлагается хранить «дерево стеков». При передаче состояния соседнему потоку также передается соответствующий указатель на дерево. Когда поток покидает состояние, он добавляет запись в «дерево стеков». В тот момент, когда необходимость в записи пропадает, она может быть удалена.

Выводы по главе 2

В первой части главы рассматривалась верифицируемая модель[8]. Показаны различия между такой моделью и моделью автоматной программы. Приведены отличия верифицируемой модели от модели Крипке и автомата Бюхи.

Вторая часть главы посвящена отличию языка *LTL* для описания

8 Верифицируемая модель – модель, утверждения о которой, проверяет верификатор. В настоящей работе отличия верифицируемой модели от модели автоматной программы в том, что объекты управления и поставщики событий рассматриваются в качестве «внешней среды».

поведения верифицируемой модели от предложенного в разд. 1.2.2.

В третьей части главы был приведен алгоритм верификации автоматных программ. Показаны его отличия от верификации автомата Бюхи.

В последней части рассматривался метод распараллеливания двойного обхода в глубину, используемый в верификаторе *Spin*.

ГЛАВА 3. МЕТОДИКА ВЕРИФИКАЦИИ АВТОМАТНЫХ ПРОГРАММ

В настоящей главе предложена методика верификации автоматных программ на основе метода *Model Checking* (глава 1) и его особенностей при верификации автоматных программ (глава 2). Новым зрения является также распараллеливание процесса верификации, о котором пойдет речь в разд. 3.1.

В этой и в последующих главах под *UniMod*-моделью будем понимать всю реагирующую систему целиком, как она действует в инструментальном средстве *UniMod*. Под верифицируемой моделью или моделью автоматной программы будем понимать только конечный автомат из *UniMod*-модели, для которого поставщики событий и объекты управления рассматриваются в качестве «внешней среды», которая ничего не знает о последовательности переходов автомата, и результат их работы может быть любым.

3.1. Параллельная верификация автоматных программ

В настоящей работе была реализована предложена автором модификация алгоритма двойного обхода в глубину, предназначенная для многоядерного компьютера. Двойной *DFS* также начинается из стартовой вершины и ищет допускающее состояние, принадлежащее какой-нибудь компоненте связанности. Однако, в отличие от однопоточной версии, состояния автомата обходят параллельно несколько потоков. Каждый поток совершает собственный обход в глубину из стартового состояния. Для того, что бы потоки не посещали одни и те же состояния, они помечают уже посещенные.

Однако, как и в алгоритме из работы [18], многопоточность приводит к некоторому усложнению в способе хранения стека обхода в глубину. Это требуется для того, чтобы не дублировать общую часть стека разными потоками. В многопоточной версии обхода в глубину вместо стека применяется дерево, где каждый путь от корня до листа – это стек одного из потоков. При посещении всех состояний, достижимых из данного, оно удаляется из дерева в

том и только том случае, если у него нет детей. Это означает, что теперь состояние удаляется только если оно не лежит ни в одном стеке. Удаление состояния из дерева означает удаление его из списка детей его родителя.

Если при обходе в глубину поток t вернулся в состояние s, из которого не ведут переходы в непосещенные состояния, то поток не возвращается в предыдущее состояние, а сначала проверяет, есть ли у данного состояния дети. Наличие ребенка означает, что состояние s находится в стеке другого потока q (возможно в стеке нескольких потоков). Поэтому поток t не переходит в предыдущее состояние, а сначала пытается помочь потоку q. Для этого он обходит в глубину поддерево с корнем в состояние s, пока не обнаружит такое состояние s', из которого ведут переходы в непосещенные состояния. При обнаружении указанного состояния поток t возобновляет *DFS* по непосещенным состояниям. Если у состояния s нет детей, то первый поток, обнаруживший это, удаляет его. Если s – допускающее состояние, то указанный поток запускает *DFS* для поиска цикла.

Поиск цикла, проходящего через допускающее состояние, может быть запущен не обязательно тем же потоком, который обнаружил данное состояние. Второй *DFS* имеет собственный стек и собственное множество посещенных им состояний. Это исключает коллизии с остальными потоками. Если обнаружен контрпример, то поток заканчивает поиск и информирует все оставшиеся потоки.

Подход, когда поток не сразу покидает состояние, а сначала помогает другому завершить поиск, позволяет им не простаивать. Например, если в графе переходов есть точка сочленения[9], то поток, первый посетивший ее, был бы вынужден в одиночку обходить все вершины из второй компоненты связанности, полученной удалением точки сочленения.

В начале работы предложенного алгоритма дерево содержит только стартовую вершину, и все потоки начинают поиск, начиная с нее. Поиск

9 Точка сочленения – это вершина в графе, удаление которой увеличивает число компонент связанности.

заканчивается, если одним из потоков обнаружен контрпример или дерево стало пустым. Таким образом, предложенный метод можно назвать «тройной обход в глубину». Первый обход – поиск по общему дереву вершины, из которой существует переход в непосещенное состояние. Второй обход – обход непосещенных состояний и поиск допускающих. Второй обход добавляет вершину в общее дерево при совершении перехода в вершину и может удалять их при покидании, если она не используется другим потоком. Третий обход – поиск цикла, проходящего через допускающее состояние.

Приведем предложенный алгоритм на псевдокоде:

```
1.  TreeNode {
2.      State state             //состояние автомата
3.      TreeNode parent         //предыдущее состояние в стеке
4.      Set<TreeNode> children //множество детей
5.      boolean wasLeft         //были ли исследованы все
    состояния, достижимые из state
6.  }
7.
8.  void dfs(TreeNode root) {
9.    TreeNode node = root;
10.   visited.add(root.state);
11.   while (node != null && не найден контрпример) {
12.     State child = node.state.next;
13.     if (child != null) {
14.       if (!visited.contains(child)) {
15.         visited.add(child);
16.         node = node.createChild(child);
17.       }
18.     } else if (∃ childNode: childNode.wasLeft == false) {
19.       node = childNode;
20.     } else {
21.       leaveNode(node);
```

```
22.        node = node.parent;
23.      }
24.      terminate(False);
25. }
26.
27. void leaveNode(TreeNode node) {
28.   if (node.wasLeft.compareAndSet(false, true)) {
29.     node.parent.removeChild(node);
30.     if (node.state – допускающее) {
31.       ищем цикл, проходящий через node.state обходом в глубину
32.     }
33.   }
34. }
```

Строки 1–6 описывают структуру данных для хранения узлов дерева. В строке 12 получаем неисследованный переход из состояния *state*. В сроке 14 проверяется, что при совершении такого перехода не посещается уже посещенное состояние. Заметим, что строки 14 и 15 должны выполняться атомарно, иначе может получиться так, что два потока одновременно придут в одно и тоже состояние. В строке 18 проверяется возможность перейти в состояние *childNode*, которое лежит в стеке другого потока. Если все состояния, достижимые из *state*, уже исследованы, то оно покидается в строке 21. Состояние удаляет тот поток, который первый пометил состояние как покинутое (строка 28), и он же запускает *DFS* для поиска цикла.

3.2. Методика верификации автоматных программ

Приведем методику использования созданного верификатора. На рис. 4 изображена схема процесса верификации программ, созданных при помощи инструментального средства *UniMod*.

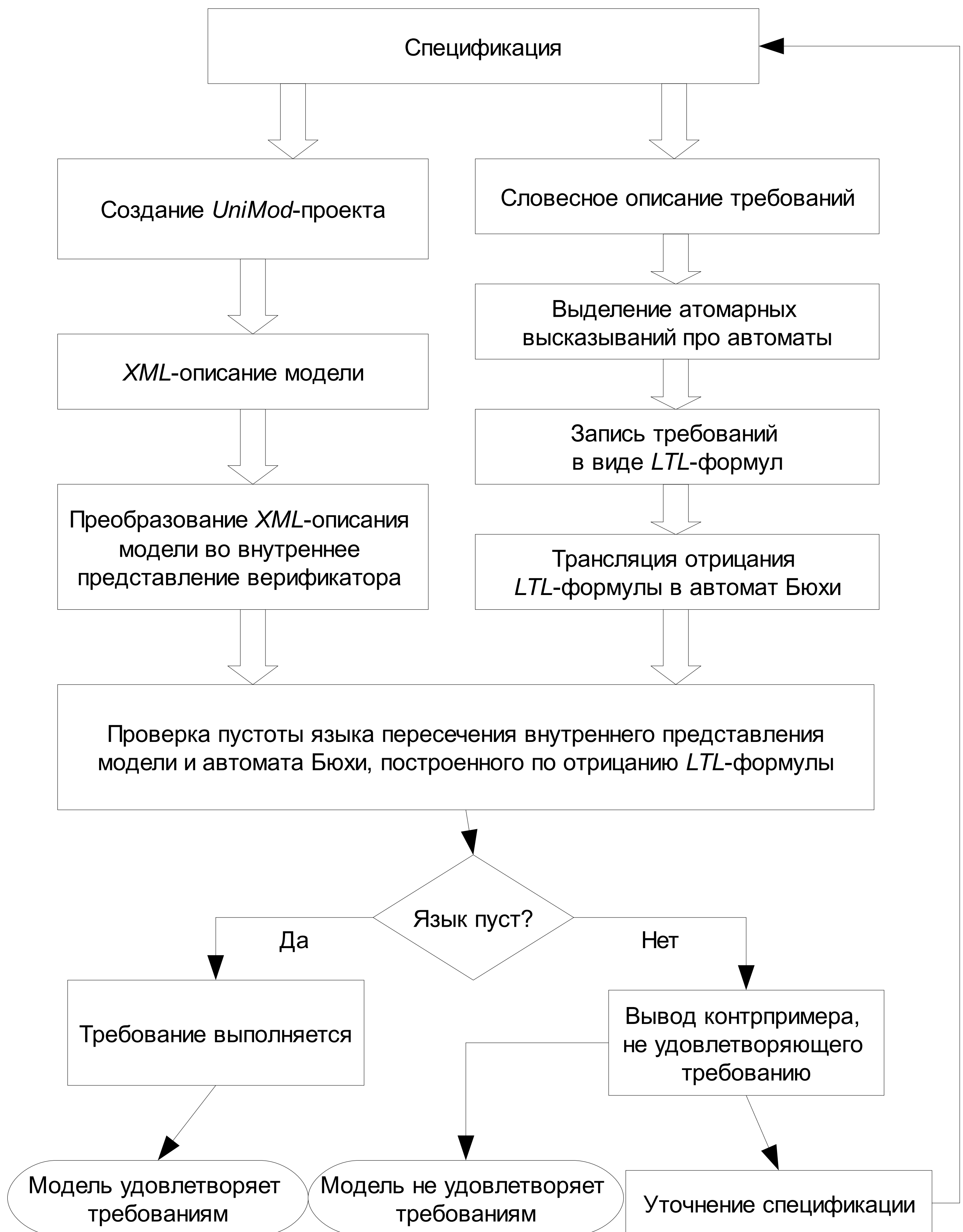

Рис. 4. Методика верификации UniMod-моделей

Создается спецификация будущей программы. 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.

После создания спецификации возможны два варианта: сначала создать *UniMod*-проект, а затем записать для него словесные требования для проверки верификатором, или же сначала сформулировать проверяемые требования, а затем создать программу. Возможно также, что спецификация уже включает в себя четко сформулированные требования об автоматной программе, выполнение которых планируется проверять.

После создания *UniMod*-проекта его модель сохраняется в виде *XML*-файла, который и будет использоваться верификатором для проверки утверждений. *XML*-файл автоматически создает инструментальное средство *UniMod*. Поэтому можно ожидать, что в нем нет ошибок, свойственных построению в ручную.

После словесного описания требований из них выделяются атомарные высказывания (предикаты), соответствующие утверждениям о переходах и состояниях в *UniMod*-модели. Например, требование «После возникновения аппаратной ошибки система отменит последнюю операцию» может быть переформулировано в следующее высказывание про автоматы: «После события *p1.e10*, рано или поздно будет вызвано действие *o1.z10*», где *p1.e10* – событие, посылаемое при аппаратной ошибке, а *o1.z10* – откат последней операции.

Такие преобразования над утверждениями позволяют записать требования в виде *LTL*-формул. Если выразительная способность языка *LTL* не позволяет записать требования в виде *LTL*-формул, то они должны быть переформулированы.

После того как имеется *XML*-описание модели и запись *LTL*-формул, начинается работа созданного верификатора. При верификации верификатор подтверждает выполнимость *LTL*-формулы или выдает контрпример в виде

последовательности переходов автомата пересечения автомата модели и автомата Бюхи, построенного по *LTL*-формуле. Пользователь может самостоятельно «настроить» верификатор под свои требования, указав какие алгоритмы во время процесса верификации будут использоваться. Также пользователю доступен выбор между верификацией одного автомата или всей иерархической системы автоматов целиком.

Продолжим описание работы верификатора. Верификатор читает *XML*-файл и строит по нему модель для верификации. Заметим, что утверждения для проверки верификатором записываются про *UniMod*-модель, которая отличается от верифицируемой как было описано в разд. 2.1.

Затем по *LTL*-формуле строится ее отрицание и оно транслируется в автомат Бюхи. Трансляция в автомат Бюхи может быть осуществлена как разработанным автором транслятором, так и при помощи транслятора *LTL2BA*. Оба способа трансляции автоматические и пользователь верификатора может не заметить, который них был использован для построения автомата Бюхи.

Затем верификатор совершает двойной обход в глубину по пересечению модели автоматной программы и автомата Бюхи, полученного по *LTL*-формуле. Причем пересечение двух автоматов строится не сразу, а по мере посещения состояний автомата пересечения. Как уже говорилось ранее, при невыполнимости формулы это позволяет обнаружить контрпример, не строя пересечение автоматов в целом.

Двойной обход в глубину реализован в однопоточной и многопоточной формах. Результат работы обеих версий одинаков: контрпример, опровергающий утверждение, или же подтверждение выполнимости формулы. Однако контрпримеры однопоточной и многопоточной версий могут отличаться, что не является существенным.

Если верификатор обнаружил контрпример, то, возможно, найдена ошибка в *UniMod*-модели. Тогда требуется внесение изменения в *UniMod*-модель, ее сохранение в виде *XML*-файла, а затем повторная

верификация. Если же контрпример не является ошибкой в *UniMod*-модели из-за неправильной формулировки требований или же из-за неучтенной внутренней реализации поставщиков событий и объектов управления, то необходимо уточнение утверждения, повторная его запись в виде *LTL*-формулы и повторная верификация.

Если верификатор подтвердил выполнимость всех *LTL*-формул, то можно полагать, что модель удовлетворяет заявленным утверждениям. Повторная верификация может быть запущена при внесении в модель изменений, с целью проверки заявленных утверждений.

Для простоты повторной верификации предлагается оформлять каждое проверяемое утверждение в виде отдельного *Unit*-теста. Тогда при внесении изменения в *UniMod*-модель будет иметься возможность повторного запуска всех тестов. При их выполнимости можно будет утверждать, что модель соответствует заявленным требованиям.

Выводы по главе 3

В первой части главы автором предложена модификация алгоритма двойного *DFS* для работы на многоядерном компьютере. Предлагаемый алгоритм может быть реализован без использования блокировок потоков (*lock-free*), что позволяет избежать простоев потоков и затрат на их синхронизацию.

Во второй части предложена методика верификации автоматных программ, используя созданный верификатор. Предлагается создавать и верифицировать *UniMo*d-проекты согласно схеме, описанной в этой части главы.

ГЛАВА 4. ОПИСАНИЕ ВЕРИФИКАТОРА

В настоящей главе дано описание структуры *Java*-классов верификатора и их назначение. Приведены примеры их использования для верификации модели автоматной программы, созданной с помощью инструментального средства *UniMod*.

4.1. Описание основных классов верификатора

Приведем основные классы и пакеты верификатора, знание назначения которых может потребоваться для применения верификатора.

Таблица 1. Структура классов верификатора

Package	Class	Комментарий
ru.ifmo.automata		Классы, представляющие конечные автоматы.
ru.ifmo.automata.statemashine		Классы, представляющие модель конечного автомата, аналогичную *UniMod*-модели.
ru.ifmo.automata.statemashine.impl	UnimodXmlReader	Класс, предназначенный для построения модели конечного автомата по *XML*-файлу.
	AutomataContext	Класс, хранящий информацию о поставщиках событий, объектах управления и автоматах.
ru.ifmo.ltl.buchi		Классы, представляющие автомат Бюхи, построенный по *LTL*-формуле.
ru.ifmo.ltl.buchi.translator		Классы, предоставляющие возможность преобразовывать *LTL*-формулу в автомат Бюхи.
	SimpleTranslator	Реализация транслятора

Package	Class	Комментарий
		LTL- формулы в автомат Бюхи.
	`Ltl2baTranslator`	Реализация преобразования *LTL*-формулы в автомат Бюхи при помощи транслятора *LTL2BA*.
`ru.ifmo.ltl.converter`	`LtlParser`	Класс для распознавания *LTL*- формулы.
`ru.ifmo.ltl.grammar`		Внутреннее представление *LTL*- формулы.
	`LtlNode`	*LTL*- формула представляется в виде дерева, где вершины – это наследники данного класса.
`ru.ifmo.ltl.grammar.predicate`		Классы, методы которых могут являться предикатами. Метод считается предикатом, если он возвращает тип *boolean* или *Boolean* и помечен аннотацией *@Predicate.*
	`PredicateFactory`	Класс с методами-предикатами. Предназначен для составления утверждений об одном автомате.
	`ComplexPredicateFactory`	Класс с методами-предикатами для составления утверждений об иерархической системе автоматов.
`ru.ifmo.test.verifier`		Классы, предназначенные для упрощения создания набора тестов для *UniMod*- проектов.
`ru.ifmo.util`		Вспомогательные классы для работы со структурами данных.
`ru.ifmo.util.concurrent`		Структуры данных для одновременной работы в нескольких

Package	Class	Комментарий
		потоках.
`ru.ifmo.verifier`		Классы для проверки утверждений об автоматных программах.
`ru.ifmo.verifier.automata`		Классы для работы с пересечением автоматов, в которых определяются допускающие состояния, лежащие в сильной компоненте связанности.
`ru.ifmo.verifier.impl`	`SimpleVerifier`	Реализация однопоточной версии верификатора. Использует метод двойного обхода в глубину.
`ru.ifmo.verifier.concurrent`	`MultiThreadVerifier`	Реализация многопоточной версии верификатора. Использует модификацию метода двойного обхода в глубину из разд. 3.1.

4.2. Применение основных классов верификатора

В настоящем разделе опишем несколько способов применения классов верификатора для проверки утверждений о модели автоматной программы. Для проверки утверждений об автоматной модели необходима инициализация некоторого числа классов. Это может показаться не очень удобным, однако такой подход позволяет «настроить» верификатор под конкретную автоматную модель.

4.2.1. Построение модели автоматной программы

Класс *UnimodXmlReader* позволяет из *XML*-файла строить модель автоматной программы. Конструктор класса принимает либо имя файла в виде строки *String*, либо файл типа *File*. После создания экземпляра данного класса можно с помощью его методов получить представление автомата (табл. 2).

Таблица 2. Класс UnimodXmlReader

Метод	Комментарий
`IStateMashine<? extends IState> readRootStateMashine()`	Чтение главного автомата.
`Map<String, IEventProvider> readEventProviders()`	Чтение поставщиков событий.
`Map<String, IControlledObject> readControlledObjects()`	Чтение объектов управления.
`Map<String, ? extends IStateMashine<? extends IState>> readStateMashines()`	Чтение всех автоматов.

Напомним, что в верифицируемой модели поставщики событий и объекты управления рассматриваются как внешняя среда. Модель ничего не знает об их внутренней реализации, а только о названиях событий, действий и переменных.

Для удобной навигации по поставщикам событий, объектам управления и автоматам применяется класс *AutomataContext,* конструктор которого принимает на вход класс *UnimodXmlReader* и сохраняет все объекты. Доступ к ним может быть осуществлен по имени, с помощью вызовов соответствующих методов (табл. 3).

Таблица 3. Класс AutomataContext

Метод	Комментарий
`IControlledObject getControlledObject(String name)`	Получение объекта управления по имени.
`IEventProvider getEventProvider(String name)`	Получение поставщика событий по имени.
`IStateMashine<? extends IState> getStateMashine(String name);`	Получение конечного автомата по имени.

4.2.2. Построение *LTL*-формул

Для построения *LTL*-формул необходим класс, методы которого

являются предикатами. Для определения методов-предикатов, применяется аннотация *@Predicate.* Методы, помеченные ею и возвращающие тип *boolean* или *Boolean,* автоматически считаются предикатами. Параметры метода могут быть объекты модели автоматной программы, такие как состояния, события, автоматы и т.д. Например, если метод-предикат принимает в качестве параметра состояние автомата, то такой предикат записывается следующим образом:

```
1. @Predicate
2. Boolean predicate(IState s) {
3.     /*
4.      * Проверить что-нибудь про состояние s.
5.      * При проверке известно предыдущее состояние state
6.      * и переход transition.
7.      */
8. }
```

После создания требуемых предикатов, формулы могут записываться в виде строки, например, такой как «*!G(predicate(A1.s1))*». Верификатор поймет, что пользователь хочет проверить формулу: «неверно, что всегда выполнено *predicate(A1.s1)*».

Для трансляции формулы из строки во внутреннее представление верификатора применяется класс *LtlParser.* Конструктор класса принимает на вход *AutomataContext* и объект с методами-предикатами. Класс предоставляет возможность трансляции строки типа *String* во внутреннее представление *LTL*-формулы. Например, таким образом:

```
1. PredicateFactory predicates = new PredicateFactory();
2. IAutomataContext context = new AutomataContext(
       new UnimodXMLReader("A1.xml"));
3. IltlParser parser = new LtlParser(context, predicates);
4. LtlNode ltlNode = parser.parse("!wasEvent(p1.e1)");
```

В приведенном примере объект *ltlNode* является деревом, представляющим *LTL*-формулу.

4.2.3. Трансляция *LTL*-формулы в автомат Бюхи

Трансляция *LTL*-формулы в автомат Бюхи может быть осуществлена двумя способами.

Первый – применение транслятора написанного автором на языке *Java*. Для этого требуется создать класс типа *SimpleTranslator*, который имеет один *public* метод *IBuchiAutomata translate(LtlNode root).* Этот метод принимает на вход *LTL*-формулу и возвращает автомат Бюхи.

Второй – использование класса, написанного на языке *Java*. Его отличие от первого способа состоят в том, что он сначала преобразует формулу во входной язык транслятора *LTL2BA*, а затем выполняется обратное преобразование из автомата, записанного на языке *Promela* верификатора *Spin.*

Так как оба транслятора наследуют один и тот же интерфейс *ITranslator*, то пользователю верификатора может быть незаметно, какая реализация используется.

4.2.4. Верификация модели автоматной программы

Теперь опишем целиком последовательность вызовов для проверки выполнимости *LTL*-формулы.

Приведем пример верификации одного автомата *A1* однопоточным верификатором:

```
1. // Создание предикатов.
2. PredicateFactory predicates = new PredicateFactory();
3. // Чтение модели из файла "A1.xml".
4. IAutomataContext context = new AutomataContext(
                 new UnimodXMLReader("A1.xml"));
5. // Создание транслятора LTL-формул во внутреннее
           представление верификатора.
6. ILtlParser parser = new LtlParser(context, predicates);
7. // Создание транслятора LTL-формул в автомат Бюхи.
8. ITranslator translator = new SimpleTranslator();
9. // Получение автомата с именем A1.
```

```
10.IStateMashine<? extends IState> stateMashine =
                context.getStateMashine("A1");
11.// Создание верификатора для автомата A1.
12.IVerifier<IState> verifier = new SimpleVerifier<IState>(
           stateMashine.getInitialState(), parser, translator);
13.// Получение контрпримера. Если список пустой,
14.// то формула выполняется.
15.List<IInterNode> stack =
           verifier.verify("G(!wasEvent(p1.e2))", predicates);
```

Заметим, что верификатор ничего не знает об автомате, а только о его начальном состоянии. Поэтому верификация системы автоматов проходит аналогично, однако для получения начального состояния системы автоматов вызывается метод *createInitialState(rootStateMashine)* класса *ComplexStateFactory*, где *rootStateMashine* – главный автомат.

Приведем пример верификации одного автомата *A1* многопоточным верификатором:

```
1. // Создание предикатов.
2. PredicateFactory predicates =
          new MultiThreadPredicateFactory<ComplexState>(
          new PredicateFactory());
3. // Чтение модели из файла "A1.xml".
4. IAutomataContext context = new AutomataContext(
          new UnimodXMLReader("A1.xml"));
5. // Создание транслятора LTL-формул во внутреннее
      представление верификатора.
6. ILtlParser parser = new LtlParser(context, predicates);
7. // Создание транслятора LTL-формул в автомат Бюхи.
8. ITranslator translator = new SimpleTranslator();
9. // Получение автомата с именем A1.
10.IStateMashine<? extends IState> stateMashine =
          context.getStateMashine("A1");
11.// Создание многопоточного верификатора для автомата A1.
12.// Число потоков используемых при верификации ==
```

```
13.// Runtime.getRuntime().availableProcessors()
14.IVerifier<IState> verifier =
            new MultiThreadVerifier<IState>(
                stateMashine.getInitialState(), parser,
                translator, stateMashine.getStates().size());
15.// Получение контрпримера. Если список пустой,
16.// то формула выполняется.
17.List<IInterNode> stack =
        verifier.verify("G(!wasEvent(p1.e2))", predicates);
```

Можно заметить, что процессы инициализации однопоточного и многопоточного верификатора отличаются только в двух строках. Первое отличие – использование разных классов предикатов (строка 2). Второе – применение разных верификаторов (строка 14).

Многопоточный верификатор выбирает число потоков равное числу процессоров, доступных виртуальной машине. Однако можно самому задать их число в конструкторе класса.

В связи с тем, что для большинства проверяемых утверждений необходима одна и та же последовательность инициализаций объектов, то были реализованы специальные классы, наследующие *junit.framework.TestCase.* Они создают необходимые объекты, и им требуется только передать имя *XML*-файла и название верифицируемого автомата (системы автоматов). Благодаря этому, каждому тесту доступен уже проинициализированный верификатор, реализующий интерфейс *IVerifier.* Однако пользователь всегда может самостоятельно настроить верификатор под свои требования.

Выводы по главе 4

Первая часть главы посвящена описанию структуры классов верификатора, разработанного в настоящей работе. Приведены основные классы, которые могут быть использованы при верификации.

Во второй части главы показан процесс инициализации классов

верификатора для последующей проверки модели автоматной программы. Тем самым продемонстрирована последовательность шагов, необходимых пользователю верификатора для проверки утверждений о *UniMod*-проекте.

ГЛАВА 5. ВЕРИФИКАЦИЯ АВТОМАТНЫХ ПРОГРАММ

В этой главе рассматриваются несколько примеров верификации *UniMod*-проектов согласно методике, изложенной в разд. 3.2.

Во время разработки настоящего проекта проводилось тестирование всех частей верификатора на *UniMod*-проектах [19, 21]. На автоматной модели данных проектов были проверенны некоторые свойства, анализ которых будет выполнен в настоящей главе.

5.1. Пример верификации модели автоматной программы

Приведем пример верификации модели рубильника, автомат которого предложен на рис. 2. Рубильник может быть в состояниях «включен» и «выключен», которым соответствуют состояния автомата *s1* и *s2* соответственно. Рассмотрим верификацию такого автомата согласно методике, изложенной в разд. 3.2.

Рассмотрим вариант, когда требуется проверить утверждение «Рубильник никогда не будет выключен». При выделении предикатов из данного утверждения, оно перепишется следующим образом: «Автомат никогда не попадет в состояние *s2*». На языке *LTL* утверждение записывается следующим образом: «*G(!isInState(A1.s2))*».

Заметим, что предложенное утверждение не выполняется. При предоставлении верификатору *XML*-описания модели и *LTL*-формулы для проверки, будет выдан контрпример. Результат работы верификатора будет следующим:

```
LTL: G(!isInState(A1.s2))
Buchi:
    initial t1
    BuchiNode t1
        -->[true] t1
        -->[(isInState(A1.s2)] t2
    BuchiNode t2
```

```
        -->[true] t2

DFS1:
    [s0, t1] → [s1, t1] → [s2, t1] → [s2, t2]
DFS2:
    [s2, t2] → [s2, t2]
```

Здесь *s0* – начальное состояние автомата модели, а *t1* и *t2* – состояния автомата Бюхи, полученного трансляцией отрицания *LTL*-формулы.

Теперь рассмотрим работу верификатора, которая привела к такому результату.

Сначала верификатор по *XML*-описанию строит модель, изображенную на рис. 2:

```
<model name="Model1">
  <controlledObject class="ru.ifmo.Manager" name="o1"/>
  <eventProvider class="ru.ifmo.EventProvider" name="p1">
    <association clientRole="p1" targetRef="A1"/>
  </eventProvider>
  <rootStateMachine>
    <stateMachineRef name="A1"/>
  </rootStateMachine>
  <stateMachine name="A1">
    <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>
    <transition sourceRef="s0" targetRef="s1"/>
    <transition event="e1" guard="o1.x1" sourceRef="s1" targetRef="s2">
      <outputAction ident="o1.z1"/>
    </transition>
    <transition event="e1" guard="o1.x2" sourceRef="s2" targetRef="s1">
      <outputAction ident="o1.z2"/>
    </transition>
  </stateMachine>
</model>
```

Для опровержения утверждения верификатор строит автомат Бюхи отрицания формулы (он представлен на рис. 5). Отрицание строится исходя из

тождества «$\neg(G\varphi) \equiv F(\neg\varphi) \equiv True\ U\ \varphi$».

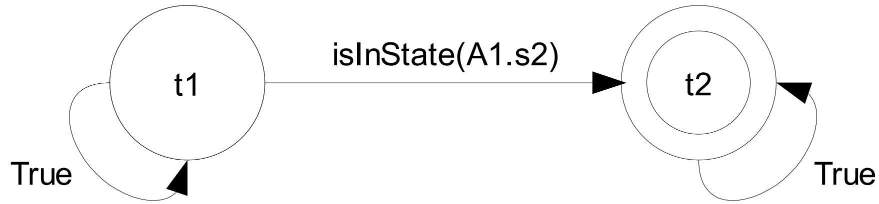

Рис. 5. Автомат Бюхи для формулы F(isInState(A1.s2)

Не будем приводить пересечение автоматов, а покажем вариант переходов модели автоматной программы и автомата Бюхи, которые приводят к обнаружению контрпримера. На рис. 6 изображена последовательность переходов, где серым цветом помечаются текущие состояния автоматов.

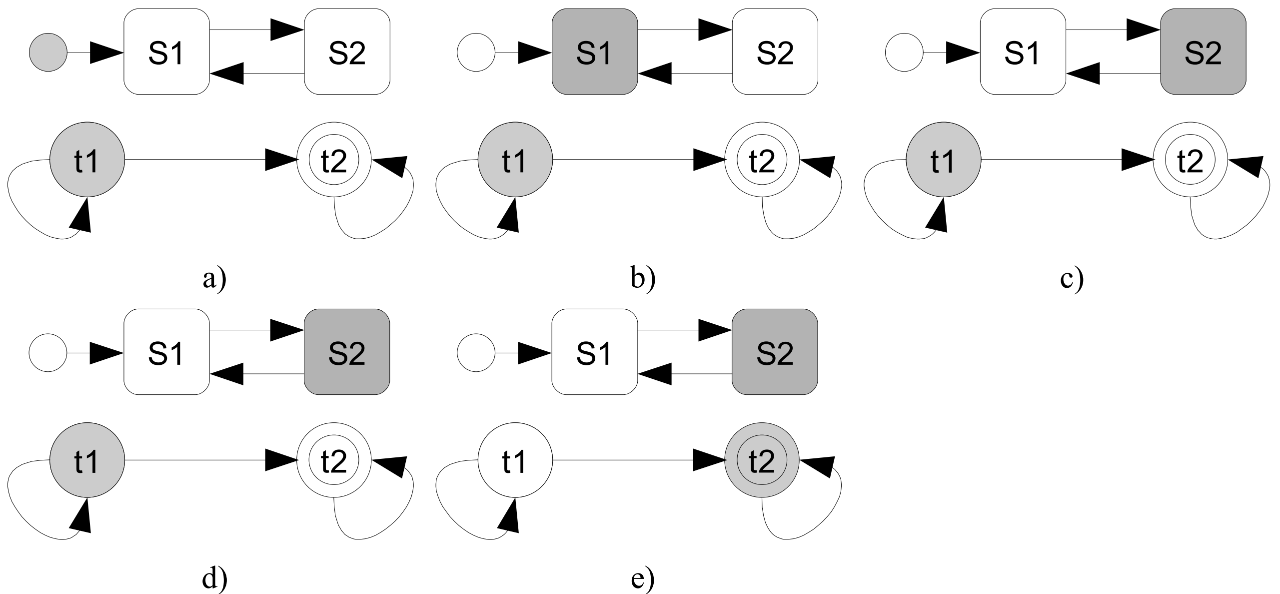

Рис. 6. Последовательность переходов модели и автомата Бюхи при верификации

В начале оба автомата находятся в стартовых состояниях (рис. 6, a). Затем автомат модели совершает переход в состояние *s1* (рис. 6, b). Автомат Бюхи при этом может совершить переход из состояния *t1* в *t1* (рис. 6, b). После этого автомат модели может не совершать перехода. Тогда автомат Бюхи опять останется в состоянии *t1.* При этом автомат пересечения оказался бы в уже посещенном состоянии. Таким образом, автомат модели переходит в состояние *s2* (рис. 6, c), а автомат Бюхи, например, может опять остаться в состоянии *t2* (рис. 6, d). Если автомат модели не совершит переход, то автомат Бюхи

вынужден будет сделать переход в состояние *t2* (рис. 6, е).

При такой последовательности переходов автомат пересечения оказался бы в допускающем состоянии, и поэтому из него будет когда-нибудь запущен второй обход в глубину, если раньше не будет найден другой контрпример. Цикл обнаруживается несложно – достаточно автомату модели не совершать перехода из состояния *s2*. При этом автомат Бюхи все-равно останется в допускающем состоянии *t2*. Следовательно автомат пересечения остался бы в том же самом состоянии. Цикл найден.

Предложенный в работе верификатор при обнаружении контрпримера, возвращает последовательность переходов, которая приводит к опровержению *LTL*-формулы. Для приведенного примера эта последовательность будет иметь вид: [*s0, t1*] → [*s1, t1*] → [*s2, t1*] → [*s2, t2*] → [*s2, t2*], где *s0* – начальное состояние автомата модели.

5.2. Анализ работы верификатора на примере игры «Побег»

Игра «Побег» – это игра для одного игрока, задача которого состоит в том, что бы уехать на своей машине от машин полицейских. Игрок управляет одной машиной, а полицейскими – компьютер. Полицейские машины образуют мультиагентную систему, взаимодействуя между собой для поимки игрока. В работе используется четыре тактики координаций действий полицейских.

Программы была написана на базе инструментального средства *UniMod*. Каждая тактика полицейских представляет отдельную систему автоматов. Также был создан отдельный автомат – модель игрового мира.

Авторами работы [19] проводилось визуальное тестирование и анализ работы программы. Для этого был создан отдельный компонент, позволяющий рисовать траектории движения полицейских машин. По такому графическому представлению можно было приблизительно говорить о тактиках движения полицейских. Автоматическая верификация модели не проводилась [20].

В ходе выполнения настоящей работы проводилась проверка только

отдельных агентов (систем вложенный автоматов). Это обусловлено тем, что на данный момент верификатор не учитывает взаимодействия между автоматами, выполняющимися параллельно, так как при этом пространство допустимых состояний было бы просто декартовым произведением пространств состояний автоматов, что обычно не так.

5.2.1. Анализ модели игрового мира

Модель игрового мира представляет систему из трех вложенных друг в друга автоматов, нескольких поставщиков событий и нескольких объектов управления. В настоящей работе были проверены свойства двух автоматов как по отдельности, так и в составе иерархической системы.

На рис. 7 представлен базовый автомат управлением состоянием игры. Автомат состоит из начального состояния, конечного и четырех обычных. В состояние «*Start*» вложен автомат *A2*.

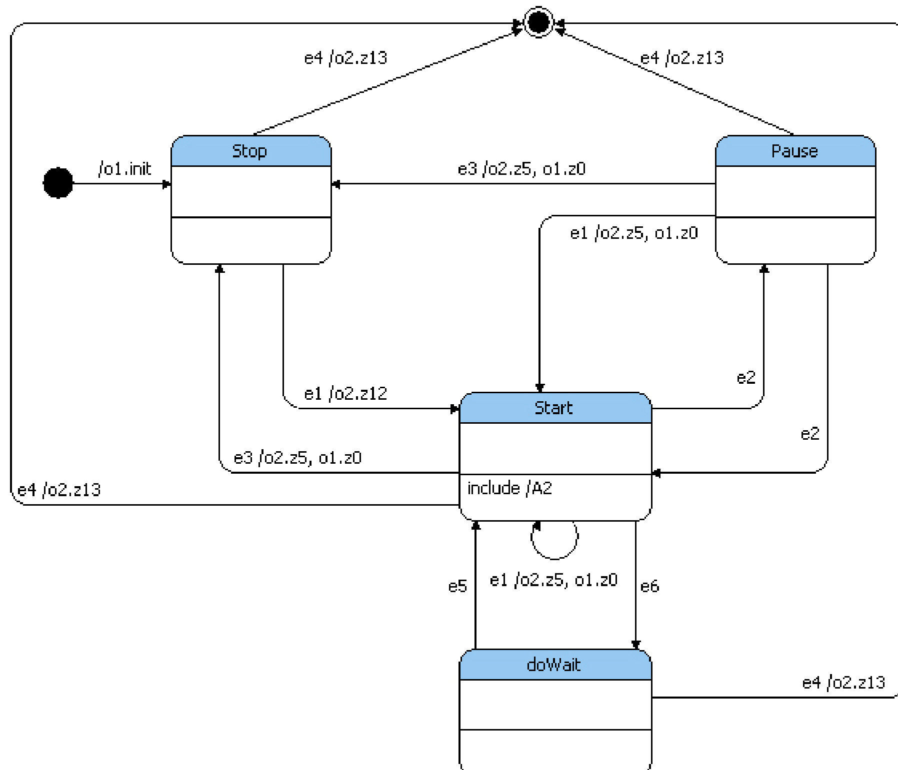

Рис. 7. Автомат A1

Разберем подробно верификацию одного утверждения, проверенного созданным верификатором согласно методике из разд. 3.2.

Пусть требуется проверить утверждение «*При переходе игры в состояние «Пауза», рано или поздно игра возобновится или пользователь закроет приложение*». При выделении предикатов из данного утверждения получается следующее высказывание: «*Если автомат A1 находится в состоянии Pause, то рано или поздно он перейдет в состояние Start или в состояние s2*», где *s2* – завершающее состояние главного автомата *A1*. Такое утверждение можно записать в виде *LTL*-формулы «*G(isInState(A1.Pause) => F(isInState(A1.Start) || isInState(A1.s2)))*».

При проверке предложенной формулы верификатор сообщил, что она верна:

```
LTL: G(!isInState(A1.Pause) || F(isInState(A1.Start) || isInState(A1.s2)))
Buchi:
    initial t1
    BuchiNode t1
        -->[true] t1
        -->[(!isInStateA1Start && !isInStateA1s2 && isInStateA1Pause)] t2
    BuchiNode t2
        -->[(!isInStateA1Start && !isInStateA1s2)] t2

Stack is empty
```

Рассмотрим его работу при проверке данной формулы. Сначала верификатор строит верифицируемую модель по *XML*-описанию. Затем отрицание *LTL*-формулы транслируется в автомат Бюхи (рис. 8).

Рис. 8. Автомат Бюхи для отрицания формулы
G(isInState(A1.Pause) => F(isInState(A1.Start) || isInState(A1.s2)))

После этого строится пересечение двух автоматов и обходятся в глубину все состояния автомата пересечения. Однако автомат пересечения строится не сразу, а последовательно совершаются переходы сначала автомата модели, затем автомата Бюхи по переходам, для которых верны предикаты. Как только автомат модели перейдет в состояние «*Pause*», автомат Бюхи может перейти в состояние *t2*. Однако не существует цикла, не проходящего через состояния «*Start*» и *s2.* Таким образом, цикл не будет найден, а, следовательно, не существует контрпримера, опровергающего формулу.

В результате верификатор переберет все состояния пересечения двух автоматов и не сможет найти цикла, проходящего через допускающее состояние. При этом верификатор подтвердит, что формула выполняется.

Также для автомата *A1* были проверены следующие *LTL*-формулы. Результаты их проверки разработанным верификатором представлены ниже:

- *G(!isInState(A1.s2)), G(!cameToFinalState()), G(!wasEvent(p2.e4))* – это эквивалентные утверждения о том, что автомат *A1* никогда не попадет в завершающее состояние. Они неверны.
- *F(isInState(A1.s2))* – это отрицание предыдущих высказываний, утверждающее, что автомат *A1* рано или поздно попадет в завершающее состояние. Однако данное утверждение также неверно, так как существует цикл, который не проходит через завершающее состояние и достижим из стартового состояния.
- *G(isInState(A1.Pause) => !X(isInState(A1.doWait)))* – утверждение, что если автомат *A1* находится в состоянии «*Pause*», то следующее состояние не может быть «*doWait*». Данное утверждение выполняется.

Приведенные выше утверждения об автомате *A1* верны также и для иерархической системы автоматов. Рассмотрим простой автомат *A3*, представленный на рис. 9. Этот автомат имитирует простую Марковскую цепь для порождения случайного числа машин полиции. Некоторые утверждения

верны для данного автомата при рассмотрении его независимо. Однако они же становятся неверными при верификации системы автоматов рассматриваемой модели.

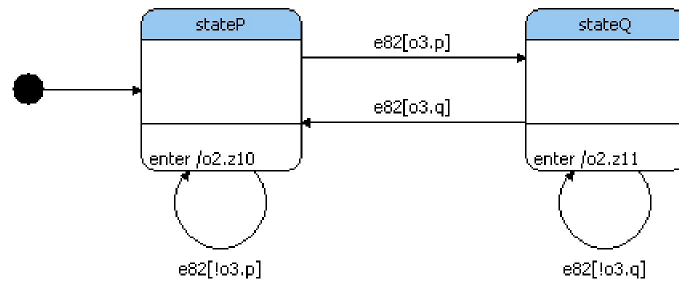

Рис. 9. Автомат A3

Вот примеры таких свойств:

- *G(isInState(A3.s1) || X(wasEvent(p3.e82))), G(wasInState(A3.s1) || wasEvent(p3.e82))*[10] – утверждение, что либо автомат *A3* находится в начальном состоянии, либо следующее событие будет *p3.e82*. При рассмотрении только автомата *A3* оба утверждения выполнены, однако если рассматривать систему автоматов, то следующее обработанное событие может быть другим – любое событие автомата *A2*, в состояние которого вложен *A3*.
- *G(wasInState(A3.stateP) => X(wasEvent(p3.e82)))* – утверждение, что если автомат *A3*, находился в состоянии «*stateP*», то следующее обработанное событие будет *p3.e82*. Данное утверждение также оказывается неверным для системы автоматов, так как следующее событие может обработать автомат *A2*.

На первый взгляд может показаться, что эти два утверждения очень просты, но они демонстрируют трудность верификации автоматной модели и недостатки верифицируемой модели. Это связанно с тем, что верифицируя один автомат и систему автоматов, получаются разные результаты. Заметим, что на самом деле эти утверждения все-таки неверны, так как при входе в состояния «*stateP*» или «*stateQ*» вызываются действия *o2.z10* и *o2.z11* соответственно.

10 *s1* – начальное состояние автомата *A3*.

Указанные действия инициируют событие, которое приводит к выходу из состояния, в которое вложен автомат *A3* рассматриваемой модели. Таким образом, при верификации системы автоматов, случайно получается правильный результат. Однако могло быть и наоборот, если бы вызываемые действия были другими.

5.2.2. Верификация тактик полицейских машин

В игре «Побег» было реализовано четыре тактики полицейских машин. Их цель – столкнуться с машиной игрока. На рис. 10 представлен автомат управлением состоянием полицейской машины (*A1*). Данный автомат используется во всех тактиках, отличие которых заключается во вложенном в состояние «*move to player*» автомате *A2*.

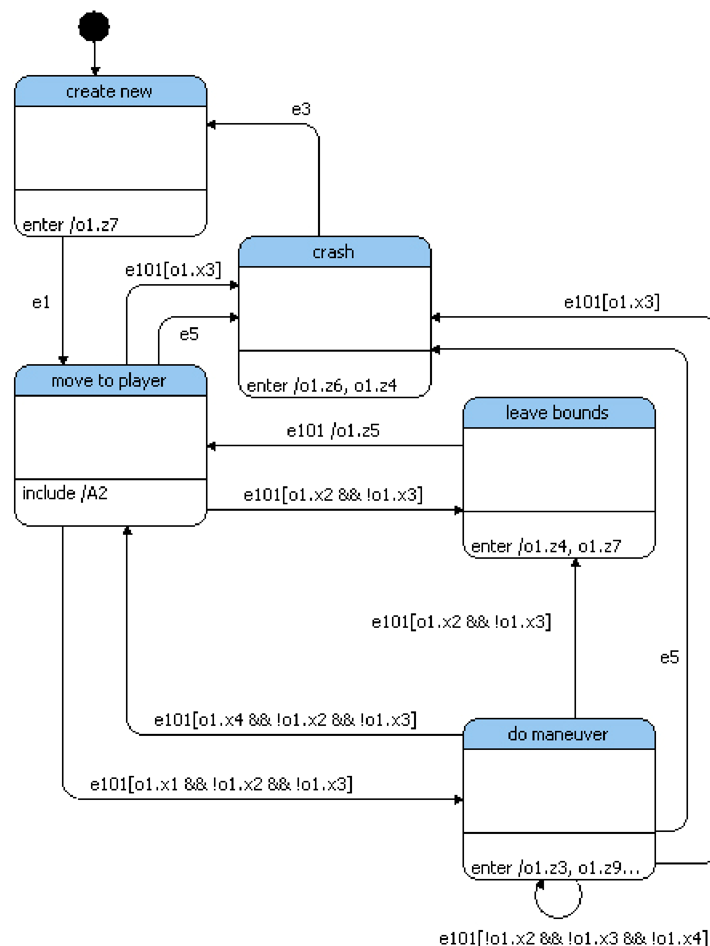

Рис. 10. Автомат управления машиной полиции A1

Приведем несколько верных утверждений, которые были проверены на данной модели:

- *G(!wasEvent(p1.e2))* – утверждение говорит о том, что никогда не будет обработано событие *p1.e2*. Утверждение выполняется, так как данное событие присутствует у поставщика событий *p1*, но не обрабатывается ни в одном состоянии;
- *G(isInState(A1[«do maneuver»]) => wasInState(A1[«move to player»]) || wasInState(A1[«do maneuver»]))* – утверждение о том, что попасть в состояние «*do maneuver*» можно только из состояния «*move to player*» или из самого себя;
- *G(isInState(A1[«crash»]) => F(isInState(A1[«move to player»])))* – утверждение о том, что если машина попала в состояние «*crash*», то рано или поздно она вернется в состояние «*move to player*».

Однако, есть ряд утверждений, для которых верификатор обнаруживает контрпример:

- *G(isInState(A1[«crash»]) => wasEvent(p1.e5))* – утверждение означает, что в состояние «*crash*» можно попасть только по событию *p1.e5*. Однако это не так – в данное состояние можно прийти и по событию *p1.e101*.
- G*(isInState(A1[«do maneuver»]) => F(isInState(A1[«move to player»])))* – утверждение о том, что если автомат находится в состоянии «*do maneuver*», то рано или поздно он окажется в состоянии «*move to player*». Верификатор обнаруживает контрпример, так как в верифицируемой модели автомат может находиться в состоянии «*do maneuver*» бесконечно долго и никогда не покинет его. Однако это не так, так как благодаря вызываемым действиям число переходов из данного состояния в самого себя ограниченно.

5.3. Верификация модели банкомата

С помощью созданного верификатора был верифицирован

UniMod-проект «Моделирование работы банкомата» [21]. В работе [22] были проверены некоторые свойства банкомата из проекта [23]. Однако данный проект не был реализован с использованием инструментального средства *UniMod.* Поэтому он не мог быть автоматически верифицирован созданным верификатором. В настоящей работе были проверенны свойства, аналогичные рассмотренным авторами работы [22], для модели банкомата, созданной с помощью инструментального средства *UniMod* [21].

Банкомат в этой работе представляет собой модель устройства, позволяющего автоматизировать операции по выдаче и переводу денег. Пользователь может совершать следующие операции: вставить карту, забрать карту, осуществить просмотр доступных средств, снять наличные.

Реализация банкомата состоит из двух частей: серверной и клиентской. Серверная реализована без использования автоматов. Клиентская часть состоят из двух вложенных конечных автоматов *AClient* и *AServer.* Автомат *AServer* посылает на сервер такие запросы клиента как авторизация, запрос баланса, снятие средств со счета. Состояния автомата *AClient* являются глобальными состояниями банкомата после какого-либо действия пользователя, например, вывода на экран информации или операции с картой клиента.

В разд. 5.3.1 рассмотрены результаты верификации банкомата, а в разд. 5.3.2 приведен пример внесения изменения в проект, которое приведет к нарушению спецификации. Однако действуя по методике верификации автоматных программ из разд. 3.2, такая ошибка будет сразу обнаружена.

5.3.1. Результат верификации модели банкомата

Конечный автомат *AServer* представлен на рис. 11. Он представляет собой простой автомат, который читает запрос клиента, обрабатывает его, посылает запрос серверу и выдает ответ клиенту.

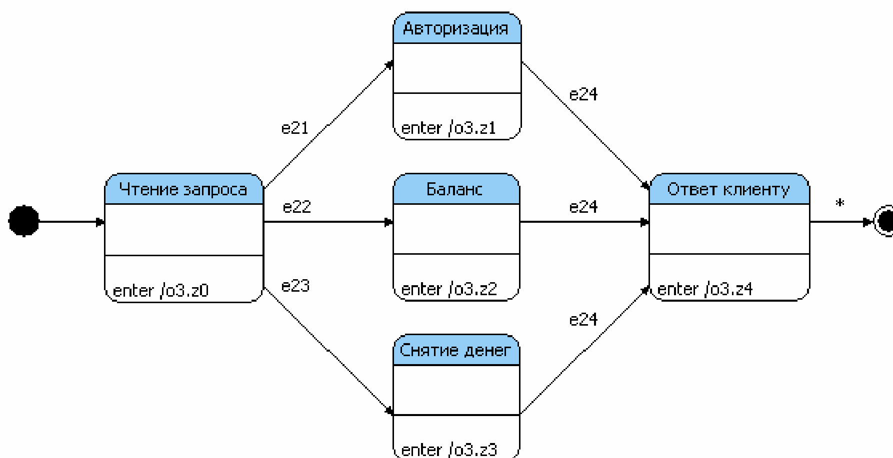

Рис. 11. Автомат AServer

Для автомата *AServer* проверенно свойство: «Получив запрос, автомат рано или поздно вернет ответ». На языке *LTL* данное свойство записывается следующим образом: «*G(isInState(AServer[«Чтение запроса»]) => F(isInState(AServer[«Ответ клиенту»])))*». Верификатор показал, что данное свойство выполняется.

Конечный автомат *AClient* представлен на рис. 12.

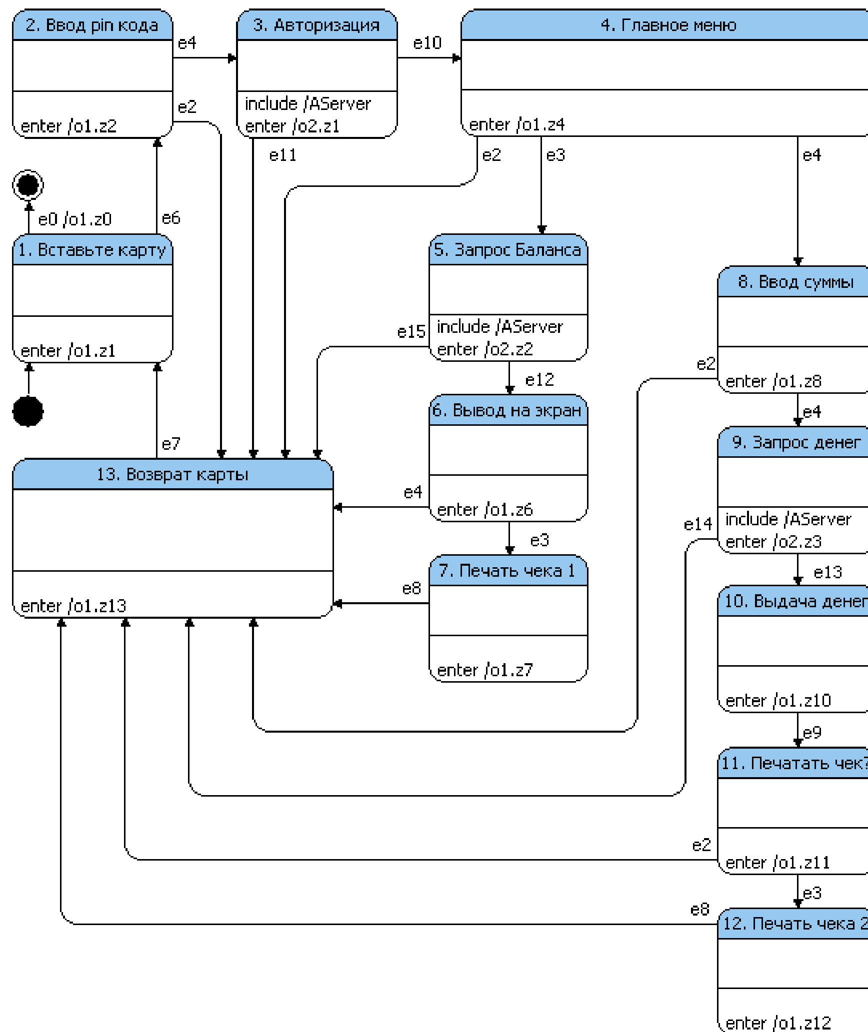

Рис. 12. Автомат AClient

Была проведена верификация автомата *AClient*. Результаты некоторых проверенных утверждений представлены ниже:

- «*G(wasEvent(p1.e6) => F(isInState(AClient[«Возврат карты»])))*» – утверждение о том, что вставив карту в банкомат, клиент обязательно ее получит обратно (событие *p1.e6* означает, что карта вставлена в банкомат). Утверждение выполняется.
- «*wasEvent(p3.e10) R !wasInState(AClient[«Главное меню»])*» – утверждение о том, что клиент не может попасть в главное меню, пока неверен *pin*-код (событие *p3.e10* означает, что *pin*-код верный). Верификатор доказал выполнимость утверждения.
- «*G(isInState(AClient[«Запрос денег»]) => X(isInState(AClient[«Выдача*

денег»])))» – утверждение о том, что после запроса денег, следующим состоянием будет выдача денег. Данное утверждение не выполняется, так как у клиента на счете может быть недостаточно денег, и запрос будет отклонен. Верификатор показал, что возможен вариант развития событий, когда после запроса наличных банкомат сразу вернул карту.

- «*wasEvent(p3.e10) R (!isInState(AClient[«Запрос баланса»]) && !isInState(AClient[«Запрос денег»]))*» – утверждение о том, что пользователь не может запросить баланс или снятие наличных до тех пор, пока не пройдет авторизацию. Верификатор доказал верность данного утверждения.

В результате верификации модели клиентской части банкомата не было обнаружено неверных утверждений, которые бы доказывали некорректную работу устройства.

5.3.2. Пример верификации модели банкомата по методике верификации автоматных программ

В предыдущем разделе было приведено утверждение «*Пользователь не может запросить снятие наличные или запросить баланс до тех пор, пока не пройдет авторизацию*». При выделении из утверждения предикатов получаем: «*Автомат AClient не попадет в состояние «Запрос баланса» или в состояние «Запрос денег» до тех пор, пока не произойдет событие p3.e10*».

Утверждение записывается в виде *LTL*-формулы следующим образом: «*wasEvent(p3.e10) R (!isInState(AClient[«Запрос баланса»]) && !isInState(AClient[«Запрос денег»]))*». Используется оператор *R (Release),* а не *U (Until)*, так как событие *p3.e10* может вообще не произойти из-за недоступности сервера или из-за того, что пользователь забыл свой *pin*-код.

Предположим, что кто-нибудь внес в автомат *AClient* еще один переход из состояния «*возврат карты*» в состояние «*главное меню*» по событию *e2*. Такое изменение модели нарушает спецификацию модели банкомата, так как

появляется возможность снять наличные или запросить баланс без авторизации.

В этом случае при изменении модели и повторном запуске всех проверенных утверждений, верификатор на утверждение «*Пользователь не может запросить снятие наличные или запросить баланс до тех пор, пока не пройдет авторизацию*», выдаст сообщение о найденном контрпримере:

```
LTL: wasEvent(p3.e10) R (!isInState(AClient["Запрос баланса"]) && !
isInState(AClient["Запрос денег"]))
Buchi:
    initial t1
    BuchiNode t1
        -->[isInState(AClient["Запрос баланса"]) || isInState(AClient["Запрос
денег"])] t2
        -->[!wasEvent(p3.e10)] t1
    BuchiNode t2
        -->[true] t2

DFS1:
    [s0, e1] → [«Вставьте карту», t1] →[«Ввод pin кода», t1] →
[«Авторизация», t1] → [«Возврат карты», t1] →[«Главное меню», t1] →
[«Запрос Баланса», t2]
DFS2:
    [«Запрос Баланса», t2] → [«Запрос Баланса», t2]
```

Верификатор обнаружил контрпример (рис. 13). На данном рисунке приведена последовательность переходов, приводящая к нарушению спецификации.

Теперь поясним работу верификатора, которая привела к обнаружению контрпримера.

Когда модель была правильной и утверждение выполнялось, верификатор сообщал, что контрпример не обнаружен. Однако когда был добавлен неправильный переход, новая *UniMod*-модель была снова сохранена в *XML*-файл. Затем была повторно запущена верификация. Верификатор по *XML*-описанию модели построил верифицируемую модель с новым переходом – внутренне представление верифицируемого автомата.

Затем верификатор транслировал *LTL*-формулу в автомат Бюхи (рис. 14).

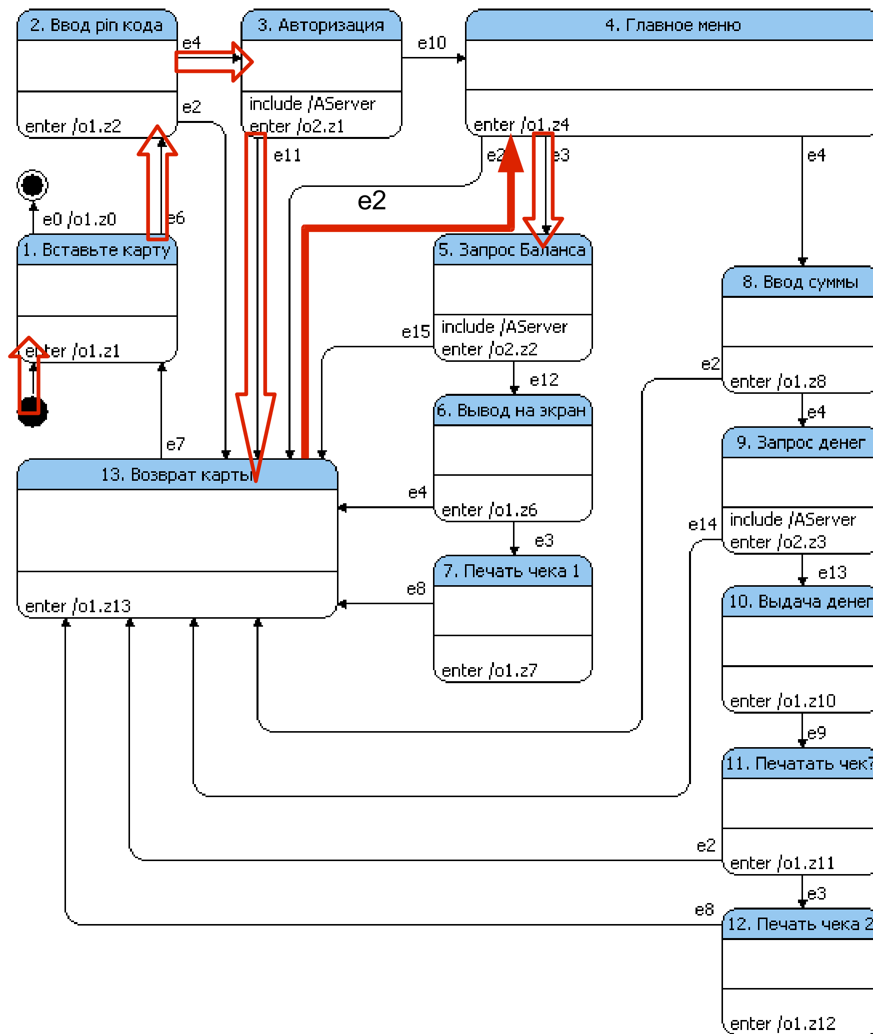

Рис. 13. Измененная модель банкомата. Крупными стрелками выделена последовательность переходов, нарушающая спецификацию

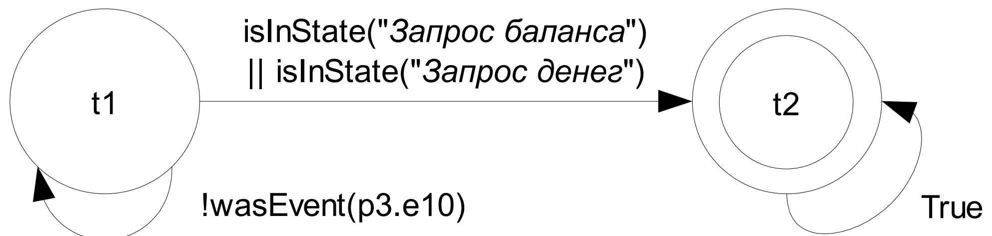

Рис. 14. Автомат Бюхи для отрицания формулы «wasEvent(p3.e10) R (!isInState(AClient[«Запрос баланса»]) && !isInState(AClient[«Запрос денег»]))»

После этого верификатор обходит в глубину состояния пересечения двух автоматов. Пересечение строится не сразу, а последовательно совершаются переходы сначала автомата модели, а затем автомата Бюхи, построенного по

LTL-формуле. В конце концов, достигается состояние модели «*Запрос Баланса*». При этом автомат Бюхи сможет перейти в состояние *t2*, так как условие на переходе будет выполнено.

Таким образом, автомат пересечения окажется в допускающем состоянии [*«Запрос Баланса», t2*]. Тогда будет найден цикл, проходящий через это состояние. Тем самым верификатор вернет обнаруженный контрпример.

5.4. Анализ эффективности верификации на многоядерном компьютере

Тестирование верификатора проводилось на двухъядерном компьютере. Верификатор был реализован двумя способами: однопоточная и многопоточная версия. Эти версии сравнивались.

Сложность анализа эффективности верификаторов заключалась в том, что не существует реальной автоматной модели, написанной на инструментальном средстве *UniMod*, число состояний которой было бы велико. В большинстве проектов число состояний одного автомата модели редко превышает 10. Это приводит к тому, что проверка *LTL*-формул занимает очень мало времени, которое измеряется в сотнях миллисекунд.

Сравнение с уже существующими верификаторами не проводилось, так как сложно оценивать время, затраченное именно на процесс поиска контрпримера, когда общее время верификации невелико.

Для увеличения общего числа состояний автоматной модели, верифицировалась иерархическая система автоматов из работы [19]. Были проверены те же утверждения, которые были приведены в разд. 5.2, а также их конъюнкции. Оценивалось время, затраченное на поиск контрпримера однопоточной версией верификатора и многопоточной. При этом отметим, что для многопоточной версии часть времени уходит на синхронизацию или на реализацию *lock-free* структур данных.

Тестирование показало, что многопоточная версия в большинстве

случаев обнаруживает контрпример быстрее. Даже на небольших примерах выигрыш в среднем составляет 20%. Однако, на некоторых примерах верификатор не показал практически никакого выигрыша, а на других – заканчивал свою работу более чем в два раза быстрее. Последний результат объясняется тем, что несколько потоков обходят состояния в другой последовательности чем один. Благодаря этому они могут быстрее обнаружить контрпример.

В случае выполнимости формул многопоточная версия на моделях с небольшим числом состояний не показала никакого выигрыша. Для сравнения двух реализаций верификатора на выполнимых формулах требовалось построение модели из бóльшого числа состояний. Так как инструментальное средство *UniMod* – это все-таки средство визуального построения программ, то создание в нем такого автомата представляется невозможным. Поэтому было автоматически создано несколько моделей с бóльшим числом состояний и переходов. При этом были сгенерированны несколько *XML*-файлов, представляющих модели с числом переходов от 1000 до 30000.

Модели создавались эмуляцией последовательности переходов автомата: в начале автомат находится в стартовом состоянии, затем он начитает совершать переходы из состояния в состояние. В новое состояние автомат переходит с вероятностью 10%, а в уже существующее – с вероятностью 80%. Каждому такому переходу назначается произвольное событие поставщика событий и вызов нескольких действий объекта управления. Построение автомата заканчивается, когда достигается определенное число переходов.

Каждая автоматически созданная модель представляет собой единственный автомат с одним поставщиком событий и одним объектом управления. Для таких автоматов проверялось утверждение «*никогда не будет обработано действие o1.z49 и никогда не будет обработано событие p1.e49*», которое выполняется для всех созданных моделей и записывается на языке *LTL* следующим образом: «*G(!wasAction(p1.e49)) && G(!wasEvent(p1.e49))*». Время

работы двух версий верификаторов представлены на рис. 15.

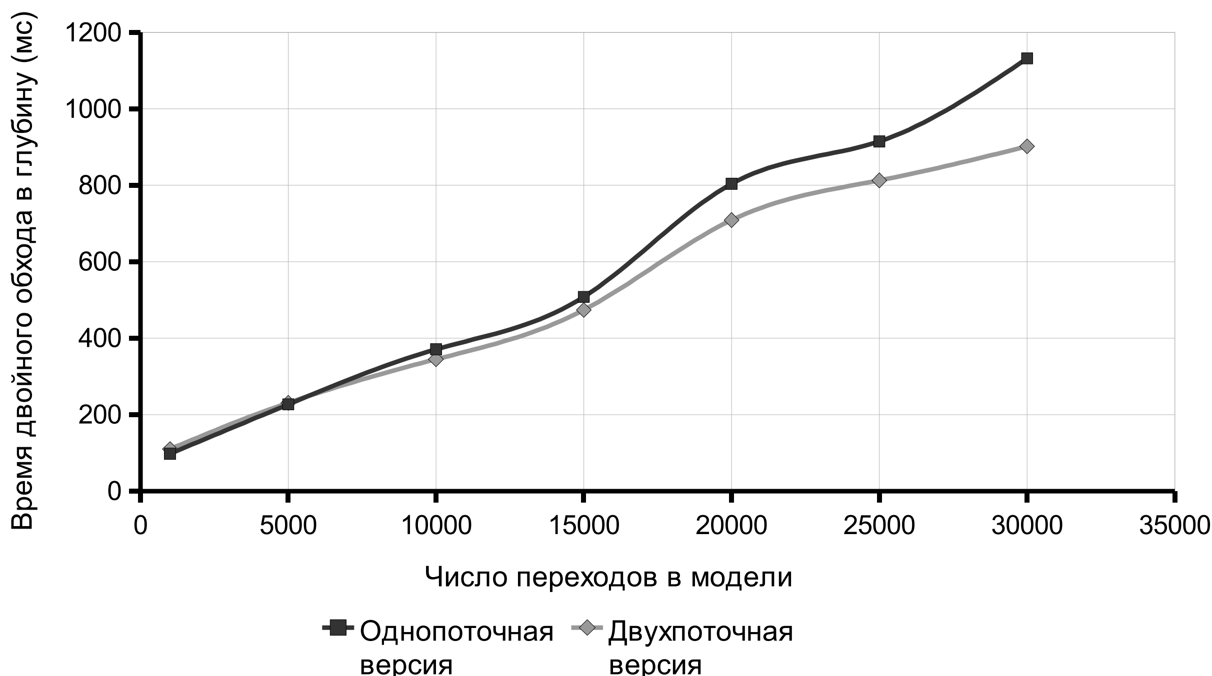

Рис. 15. Время работы верификаторов от числа переходов автоматной модели

Из рассмотрения графиков следует, что выигрыш многопоточной версии при проверке выполнимых формул появляется на моделях с большим числом переходов. Однако даже на модели с числом переходов 30000 время верификации невелико. Заметим, что двойной обход в глубину совершается на пересечении двух автоматов (автомата Бюхи, построенного по *LTL*-формуле, и автомата модели). Поэтому число состояний и переходов увеличивается в несколько раз на больших утверждениях, что может приводить к значительному увеличению времени верификации.

Выводы по главе 5

В первой части главы рассматривался пример поиска контрпримера. Была продемонстрирована работа верификатора при проверке утверждения об автоматной программе на примере верификации простого автомата.

Во второй части главы были приведены результаты верификации моделей автоматной программы из *UniMod*-проекта [19]. Рассматривались

примеры *LTL*-формул, показывающие недостатки верифицируемой модели.

Третья часть посвящена верификации банкомата из проекта [20]. Была проведена проверка ряда утверждений про автоматы данного проекта. В модель была специально внесена ошибка, и верификатор смог обнаружить нарушение спецификации.

В последней части главы приведены результаты сравнения многопоточной версии верификатора с однопоточной, которые показали преимущество применения многопоточной версии для моделей автоматных программ с большим числом переходов.

ЗАКЛЮЧЕНИЕ

В настоящей работе разработан верификатор автоматных программ, создаваемых с помощью инструментального средства *UniMod*. Он позволяет верифицировать модели автоматных программ, которые строятся по *XML*-файлу, генерируемому указанным средством. Требования к модели записываются на языке *LTL*. Верификатор предоставляет набор классов и интерфейсов на языке программирования *Java* для проверки выполнимости *LTL*-формул.

Возможно применение разработанного верификатора не только после создания программы целиком, но и во время ее разработки. Предложен ряд тестов с утверждениями об автоматной модели в процессе ее проектирования, проверку которых можно осуществлять, например, после внесения изменений в модель. Созданы специальные классы, позволяющие оформлять проверяемые формулы в виде отдельных тестов.

В верификатор встроена возможность расширять множество пропозициональных переменных, что позволило строить любые *LTL*-формулы, удовлетворяющие требованиям, описанным в разд. 2.2.

Реализовано построение модели по *XML*-файлу, которая соответствует модели, используемой в инструментальном средстве *UniMod*. Построенная модель отличается от *UniMod*-модели и от автомата Бюхи, различие которых описано в разд. 2.1. Предложенный верификатор проверяет утверждения, сделанные над данной моделью.

Трансляция *LTL*-формулы в автомат Бюхи была реализована двумя способами. Первый способ – автором разработан транслятор. Второй – использование транслятора *LTL2BA* [11]. Для использование этого средства были реализованы преобразование *LTL*-формулы в его входной язык этого транслятора и преобразование автомата Бюхи, полученного на его выходе, в автомат Бюхи разработанного верификатора. При дальнейшей работе

планируется усовершенствовать разработанный транслятор, что, возможно, позволит отказаться от транслятора *LTL2BA*.

Использование транслятора *LTL2BA* приводит к потеря мультиплатформенности верификатора, так как он написан на языке программирования *C* и поэтому требует отдельно скомпилированной версии для каждой платформы. Применение этого транслятора также вносит дополнительную сложность во внедрение верификатора в процесс создания автоматной программы. При дальнейшей работе над верификатором планируется превратить его в отдельный модуль, который можно будет присоединять к проекту на подобии *JUnit*-тестов.

Как отмечалось в разд. 1.2.4, проверка выполнимости *LTL*-формулы заключается в проверке пустоты пересечения языка, допускаемого автоматом Бюхи верифицируемой модели, и отрицания *LTL*-формулы. Проверка была реализована алгоритмом «двойного обхода в глубину» и предложенной автором модификацией данного метода для работы на многоядерном компьютере. Тестирование показало, что даже на моделях, содержащих меньше 1000 переходов, многопоточная версия двойного обхода в глубину находит контрпример быстрее однопоточной. На моделях с большим числом переходов время работы однопоточной реализации настоящего верификатора превышает многопоточной для любых *LTL*-формул.

Из изложенного следует, что цель настоящей работы выполнена – разработан верификатор, позволяющий проверять утверждения над автоматными программами.

Литература

1. *Шалыто А. А.* Switch-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. http://is.ifmo.ru/books/switch/1/
2. *UniMod project*. http://unimod.sourceforge.net

3. *Разработка технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода.* Отчет по контракту о верификации автоматных программ. Второй этап. СПб: СПбГУ ИТМО, 2007.
http://is.ifmo.ru/verification/_2007_02_report-verification.pdf
4. *Кларк Э., Грамберг О., Пелед Д.* Верификация моделей программ: Model Checking. М.: МЦНМО, 2002.
http://is.ifmo.ru/verification/_klark_gamberg_pered_verification.djvu
5. *Миронов А. М.* Математическая теория программных систем. http://intsys.msu.ru/staff/mironov/mthprogsys.pdf
6. *Конев Б. Ю.* Введение в моделирование и верификацию аппаратных и программных систем. Computer Science клуб при ПОМИ РАН. Слайды лекций (1 – 3, 11). 2007.
http://logic.pdmi.ras.ru/~infclub/?q=courses/verification
7. *Gerth R., Peled D., Vardi M. Y.,Wolper P.* Simple On-the-fly Automatic Verification of Linear Temporal Logic / Proc. of the 15th Workshopon Protocol Specification, Testing, and Verification, Warsaw, Poland: Chapman Hall, June 1995, pp. 3–18. http://citeseer.ist.psu.edu/gerth95simple.html
8. *Van Wyk C.* An LTL Verification System Based on Automata Theory. University of Stellenbosch, South Africa. 1999, pp. 1–70. http://citeseer.ist.psu.edu/vanwyk99ltl.html
9. *Courcoubetis C., Vardi M., Wolper P., Yannakakis M.* Memory-Efficient Algorithms for the Verification of Temporal Properties / Formal Methods in System Design. 1992, pp. 275–288.
10. *Somenzi F., Bloem R.* Efficient Büchi Automata from LTL Formulae / 12th Conference on Computer Aided Verification (CAV'00). 2000, pp. 248-263.
11. *LTL 2 BA project*. http://www.lsv.ens-cachan.fr/~gastin/ltl2ba/
12. *Gastin P., Oddoux D.* Fast LTL to Büchi Automata Translation / 13th Conference on Computer Aided Verification (CAV'01). 2001, pp. 53–65.

13. *Lerda F., Sisto R.* Distributed-Memory Model Checking with SPIN / Proc. of the 5th International SPIN Workshop 1680. 1999, pp. 3–17.
14. *Barnat J., Brim L., Stribrna J.* Distributed LTL Model-Checking in SPIN // Lecture Notes in Computer Science. 2057. 2001. pp. 1–17.
15. *Lafuente A.* Simplified distributed LTL model checking. Technical Report 00176, Institut fur Informatik. University Freiburg, Germany. July 2002.
16. *Holzmann G. J., Bošnački D.* The Design of a Multi-Core Extension of the SPIN Model Checker // IEEE Transactions on Software Engineering. V. 33. Issue 10. 2007, pp. 659–674.
17. *Holzmann G. J.* A Stack-Slicing Algorithm for Multi-Core Model Checking // Electronic Notes in Theoretical Computer Science (ENTCS). V. 198. Issue 1. 2008, pp. 3–16.
18. *Holzmann G. J.* A Stack-Slicing Algorithm for Multi-Core Model Checking. // Electronic Notes in Theoretical Computer Science (ENTCS). V. 198. Issue 1. 2008, pp. 3–16.
19. *Егоров К. В., Райков П. М.* Игра «Побег». СПбГУ ИТМО. 2007. http://is.ifmo.ru/unimod-projects/la_redada/
20. *Егоров К. В., Райков П. М., Шалыто А. А.* Применение автоматного подхода для создания одного класса мультиагентных систем. Материалы VI научно-технической конференции «Научное программное обеспечение в образовании и научных исследованиях». СПбГПУ: 2008, с. 46–52.
21. *Козлов В. А., Комалёва О. А.* Моделирование работы банкомата. СПбГУ ИТМО. 2006. http://is.ifmo.ru/unimod-projects/bankomat/
22. *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
23. *Первушин Е. В., Шалыто А. А.* Моделирование банкомата. СПбГУ ИТМО. 2003. http://is.ifmo.ru/projects/bankomat/