Опубликовано в Научно-техническом вестнике СПбГУ ИТМО. 2008. Выпуск 53. Автоматное программирование, с. 177-189.

УДК 004.4'242

РАЗРАБОТКА ВЕРИФИКАТОРА АВТОМАТНЫХ ПРОГРАММ

К.В. Егоров, А.А. Шалыто

(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

В статье описан разработанный авторами верификатор автоматных программ, созданных при помощи инструментального средства для поддержки автоматного программирования *UniMod*. При его использовании, в отличие от известных верификаторов, отсутствует необходимость описывать модель на входном языке верификатора. Требования к программе записываются на языке темпоральной логики линейного времени *LTL* (*Linear Time Logic*). При использовании такой логики верификация осуществляется за счет пересечения автомата-произведения модели и автомата *Бюхи*, построенного для отрицания *LTL*-формулы.

Ключевые слова: верификация программ, верификация на моделях, автоматное программирование, автомат Бюхи

Введение

С момента появления первых программ требовалось проверять их правильность, причем не просто удостовериться, что программа работает на конечном числе тестов, а уметь формально доказывать, что ее поведение соответствует спецификации.

Метод проверки того, что программная система соответствует заявленной спецификации (обладает необходимыми свойствами или удовлетворяет определенным требованиям (утверждениям)), называется *верификацией*. К сожалению, верифицировать систему обычно намного сложнее, чем ее создать. Поэтому для не очень ответственных систем верификация не всегда оправдана, и в них проще исправлять ошибки по мере обнаружения при тестировании и в процессе работы. Однако существуют такие системы, в которых ошибки нельзя допускать или они могут обойтись слишком дорого [1].

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» (Э. Дейкстра). 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. Поэтому, чтобы исключить возможность их появления, требуется рассмотреть все возможные варианты поведения системы, что при тестировании невозможно.

Наиболее практичным в настоящее время является метод верификации, названный *Model Checking* [2, 3]. При его использовании процесс верификации состоит из трех частей: моделирование программы – преобразование программы в формальную модель с конечным числом состояний для последующей верификации, спецификация – формальная запись утверждений, которые требуется проверить, и собственно верификация. Эти части связаны между собой – алгоритмы верификации зависят от способа построения модели и способа записи требований. При использовании этого метода для программ, написанных традиционно, возникают три проблемы.

1. Как для произвольной программы построить адекватную модель с конечным числом состояний?
2. Как переформулировать требования к программе (системе) в требования к модели?
3. Как при обнаружении ошибки (построении контрпримера) перейти от модели к программе?

Ответ [4, 5] на все эти вопросы может быть найден, если программы являются автоматными [6]. Здесь имеет место та же ситуация, что и при контроле аппаратуры, которая при сложной логике не может быть проверена, если она не спроектирована специальным образом с учетом контролепригодности.

Цель настоящей работы состоит в разработке верификатора для автоматных программ, созданных при помощи инструментального средства *UniMod* [7].

Особенности этого класса программ позволяют решить первую и третью проблемы верификации, так как каждая автоматная программа уже сама по себе является моделью, которая, в отличие от традиционно написанных программ, пригодна для проверки определенных утверждений о ней либо без ее модификации, как в настоящей работе, либо за счет модификации, которая может быть выполнена автоматически. Вторая проблема в случае автоматных программ решается при проектировании автоматов, устраняя тем самым семантический разрыв между требованиями к программе и модели, который имеет место для традиционно написанных программ.

В настоящей работе требования к программе формулируются в виде формул темпоральной логики линейного времени (*Linear Time Logic, LTL*). Это определяет используемый алгоритм верификации на основе пересечения автоматов *Бюхи* [3].

В ходе работы *создан верификатор*, не использующий 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 из модели в автоматную программу.

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.

Как отмечено выше, цель настоящей работы состоит в разработке верификатора автоматных программ, созданных при помощи *Switch*-технологии [6] в инструментальном средстве *UniMod* [7]. В таких программах 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, считывая значения входных переменных у объектов управления для проверки условия перехода. Такая система называется реагирующей или событийной [8].

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*) [7].

Как отмечалось выше, при верификации программ на языках типа *Java* или *C++,* написанных традиционным путем (без явного выделения состояний), требуется вручную строить по программе модель и описывать ее на языке, понятном используемому верификатору. При этом могут быть утеряны определенные данные и связи в программе, так как приходится переходить на другой уровень абстракции.

Возможны два подхода к использованию автоматной модели для верификации:

- ее формальное преобразование к виду, определяемому выбранным верификатором [5];
- создание верификатора, в котором применяется автоматная модель или некоторое уже существующее ее представление.

В настоящей работе используется второй подход, при котором автоматные программы создаются с помощью инструментального средства *UniMod,* а для верификации применяется *XML*-описание автоматов, являющееся внутренним представлением графов переходов автоматов в указанном средстве.

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. Другое достоинство автоматных программ, резко упрощающее их верификацию, – возможность достаточно просто переформулировать требования к системе в высказывания об автоматах, так как в этом случае при проектировании программы строится модель ее поведения, которая может применяться и при верификации.

В настоящей работе верифицируется не вся автоматная программа, а только ее модель, представленная в общем случае системой вложенных автоматов. При этом поставщики событий и объекты управления рассматриваются в качестве «внешней среды», которая ничего не помнит о последовательности переходов рассматриваемого автомата и вызванных действиях. Таким образом, в любой момент времени может быть совершен любой переход из текущего состояния автомата. Такой же подход рассматривается в работе [9].

Для описания требований к автоматным программам будем применять, как отмечено выше, язык *LTL*. В нем время линейно и дискретно. Синтаксис *LTL* включает в себя пропозициональные переменные *Prop*, булевы связки ($\neg$, Λ, V) и темпоральные операторы. Последние применяются для составления утверждений о событиях в будущем.

Будем использовать 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*
- если φ и ψ – формулы, то
  - $\neg\varphi$, $\varphi\Lambda\psi$, $\varphi V\psi$ – формулы;
  - $X\varphi$, $F\varphi$, $G\varphi$, $\varphi U\psi$, $\varphi R\psi$ – формулы.

Оказывается, что как модель автоматной программы, так и *LTL*-формулу можно представить в виде автомата *Бюхи*. Формально он определяется пятеркой (*S, E, T, s0, F*), где

- *S* – конечное множество состояний;
- *E* – множество меток переходов;
- $T \subseteq S \times E \times S$ – множество переходов;
- *s0* – начальное состояние;

- $F \subseteq S$ – множество допускающих состояний.

Тогда путь в этом графе $\pi = s0, s1, s2, \ldots sn, \ldots,$ для которого выполнено $T(si-1, e, si)$, где e – метка перехода, будет последовательностью вычислений системы. Путь является *допускающим*, если существует состояние из множества F, встречающееся бесконечно часто.

Подробно трансляция *LTL*–формулы в автомат *Бюхи* описана в работах [3, 10, 11].

При этом отметим, что в автоматной программе модель поведения может содержать один автомат или являться системой вложенных автоматов. При использовании рассматриваемого подхода по системе автоматов строится автомат-произведение [12]. Автомат или автомат-произведение представляет собой автомат *Бюхи*, в котором метка на переходе – это выполнимость определенного предиката. Под предикатом будем понимать утверждение о текущем переходе, например, вызванные автоматом действия в объектах управления или состояние, в которое перешел автомат.

Для доказательства невыполнимости некоторой *LTL*-формулы на автомате *Бюхи* можно проверить, что пересечение верифицируемого автомата *Бюхи* и автомата *Бюхи,* соответствующего отрицанию *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 автоматной программы, по отрицанию *LTL*-формулы строится автомат *Бюхи*, затем строится автомат пересечения, а после этого проверяется, что этот автомат не допускает ни одного слова.

В связи с тем, что рассматриваются бесконечные слова, то, как доказано в работе [3], для пустоты пересечения достаточно доказать, что ни одно допускающее состояние не принадлежит компоненте сильной связности, которая достижима из начального состояния (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.

При верификации обычно применяют двойной обход в глубину [3], преимущество которого состоит в том, что для реализации этого алгоритма не требуется построение автомата-пересечения целиком – можно строить состояния пересечения автоматов по мере их достижения. Это дает выигрыш на больших моделях.

Общая идея алгоритма такова: обходим в глубину автомат пересечения, 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. Следовательно, исходная *LTL*-формула не выполняется на автомате *Бюхи*, представляющем модель программы, и найден контрпример.

Как было отмечено выше, алгоритм верификации *одного автомата* может быть записан следующим образом.

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:
5. Перебираются все переходы верифицируемого автомата.
6. Перебираются все возможные переходы автомата, построенного по отрицанию *LTL*-формулы, для перехода верифицируемого автомата, полученного на шаге 4.1.

Для верификации *системы вложенных автоматов* применяется такой же алгоритм, только в качестве верифицируемого автомата строится автомат-произведение [12], состояния которого содержат информацию о состояниях всех автоматов иерархи-

ческой системы. Каждое состояние нового автомата представляет собой дерево, структура которого совпадает со структурой системы вложенных автоматов. В узлах дерева размещены состояния, в которых находятся 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 только 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, все вложенные в него состояния также неактивны. Это позволяет строить переходы из такого сложного состояния, просматривая не все узлы дерева, а только активные.

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

От произведения автоматов нельзя отказаться, так как если требуется делать утверждения о состоянии системы автоматов в целом, то необходимо иметь представление такого глобального состояния. Верификатор, разработанный в настоящей работе, предоставляет возможность проверять утверждения как об отдельном автомате иерархической системы, так и обо всей системе в целом.

Отметим, что проверка утверждения для одного автомата не всегда дает тот же результат, что проверка этого же утверждения для системы автоматов, содержащей этот автомат. 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.

Приведем методику использования созданного верификатора. На рис. 1 изображена схема процесса верификации автоматных программ, созданных при помощи инструментального средства *UniMod*.

Разрабатывается спецификация будущей программы. Она описывает поведение программы и требования к ней, которые должны выполняться. Это требуется для того, чтобы в дальнейшем была возможна проверка утверждений о программе.

После создания спецификации возможны два варианта: сначала создать *UniMod*-проект, а затем записать для него словесные требования к системе, проверяемые 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.

Модель *UniMod*-проекта сохраняется в виде *XML*-файла, который и будет использоваться верификатором для проверки утверждений. *XML*-файл автоматически генерируется инструментальным средством *UniMod* при создании программы. Поэтому можно ожидать, что в нем нет ошибок, свойственных построению вручную.

После словесного описания требований из них выделяются атомарные высказывания (предикаты), соответствующие утверждениям о переходах и состояниях в *UniMod*-модели. Например, требование системе «*После возникновения аппаратной ошибки система отменит последнюю операцию*» может быть переформулировано в высказывание об автомате: «*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 к модели в виде *LTL*-формул. Если выразительная способность языка *LTL* не позволяет записать требования в виде *LTL*-формул, то они должны быть переформулированы.

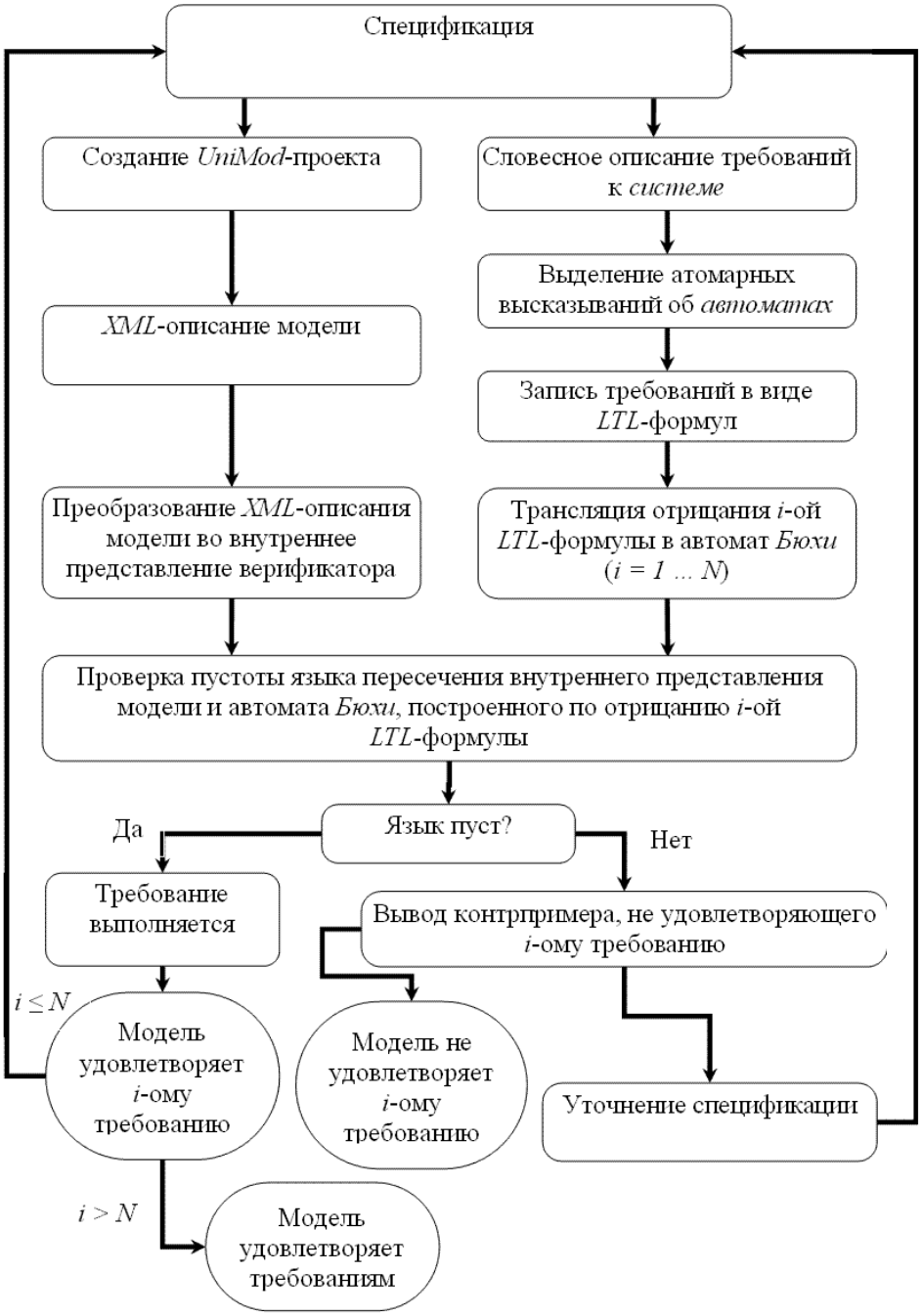

Рис. 1. Методика верификации *UniMod*-моделей

По *XML*-описанию модели и *LTL*-формулам (их отрицаниям) начинается работа созданного верификатора. В ходе работы верификатор подтверждает выполнимость утверждения или выдает контрпример в виде последовательности переходов автомата пересечения автомата модели и автомата *Бюхи*, построенного по инверсии *LTL*-формулы. Пользователю доступен выбор между верификацией одного автомата или иерархической системы автоматов.

Продолжим описание работы верификатора. Верификатор читает *XML*-файл и автоматически строит по нему модель для верификации.

Затем по *LTL*-формуле строится ее отрицание, и оно транслируется в автомат *Бюхи*. Трансляция в автомат *Бюхи* может быть осуществлена как разработанным авторами транслятором, так и при помощи транслятора *LTL2BA* [13].

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

Двойной обход в глубину может быть реализован в однопоточной и многопоточной (см. последний раздел статьи) формах. Результат работы обеих версий одинаков – контрпример, опровергающий утверждение, или же подтверждение выполнимости формулы.

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 требованиям. Повторная верификация может быть запущена при внесении в модель изменений с целью проверки заявленных свойств.

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 утверждать, что модель соответствует заявленным требованиям.

Во время разработки верификатора проводилось тестирование всех его частей на *UniMod*-проектах [14, 15]. На автоматной модели этих проектов были проверенны некоторые свойства. Верификатор доказал верные утверждения и опроверг неверные, тем самым подтвердив возможность своего применения.

Проект [15] реализует банкомат (рис. 2, 3), позволяющий пользователю совершать такие операции, как снятие наличных денег и просмотр доступных средств на счете. Модель банкомата представляет собой двухуровневую структуру автоматов, где автомат *AServer* вложен в автомат *AClient*. Рассмотрим одно из проверенных утверждений про банкомат: «*Пользователь не может запросить снятие наличные или запросить баланс до тех пор, пока не пройдет авторизацию*». При выделении из утверждения предикатов получаем утверждение про автомат: «*Автомат AClient не попадет в состояние «Запрос баланса» или в состояние «Запрос денег» до тех пор, пока не произойдет событие p3.e10*». В утверждении применяется предикат об обработке события *p3.e10*, а не посещение состояния «*Авторизация*», так как это событие означает про-

хождение авторизации, а состояние – обращение к серверу для проверки правильности введения *pin*-кода.

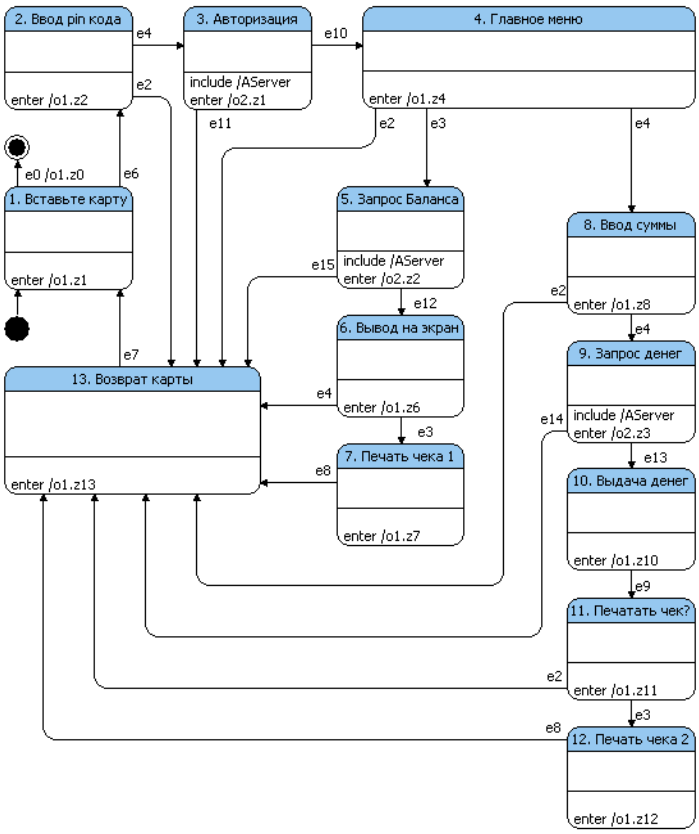

Рис. 2. Модель банкомата (автомат *AClient*)

Верифицируемое утверждение записывается в виде LTL-формулы вида «`wasEvent(p3.e10) R (!isInState(AClient[«Запрос баланса»]) && !isInState(AClient[«Запрос денег»]))`».

Здесь 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. Данное утверждение выполняется для модели банкомата.

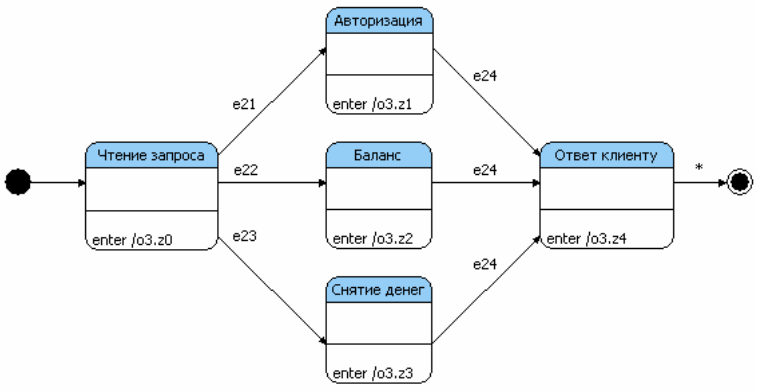

Рис. 3. Модель банкомата (воженный автомат AServer)

Предположим, что кто-то внес в автомат *AClient* еще один переход из состояния «*Возврат карты*» в состояние «*Главное меню*» по событию *e2* (рис. 4). Такое изменение модели нарушает спецификацию банкомата, так как появляется возможность снять наличные или запросить баланс без авторизации. При верификации измененной модели верификатор обнаруживает контрпример, нарушающий спецификацию. На рис. 4 путь, соответствующий контрпримеру, выделен крупными стрелками.

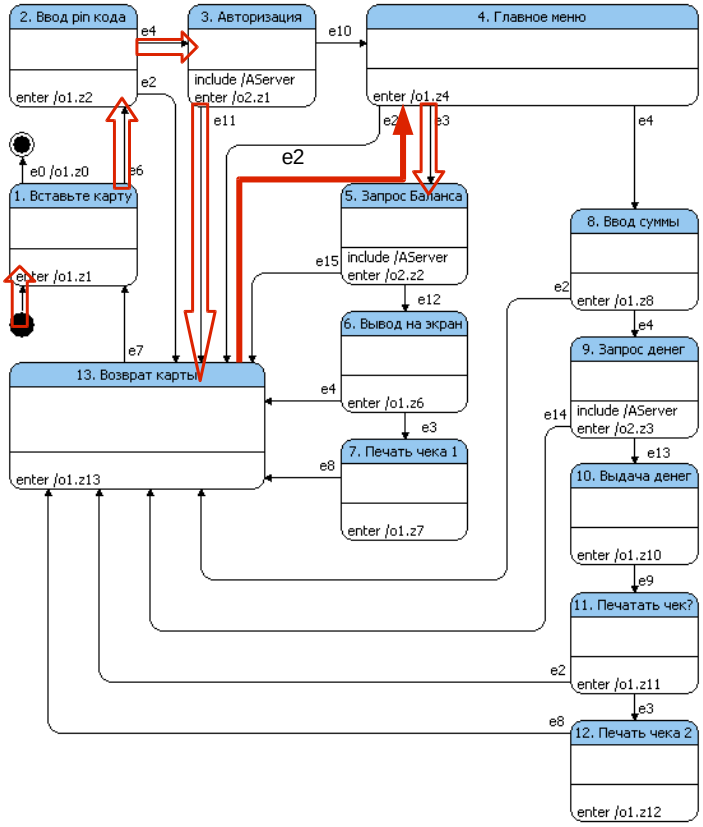

Рис. 4. Измененная модель банкомата. Крупными и жирной стрелками выделена последовательность переходов, нарушающая спецификацию

Этот контрпример верификатор выдает в виде последовательности переходов: [*s1, s1*] → [«*Вставьте карту*», *s1*] → [«*Ввод pin-кода*», *s1*] → [«*Авторизация*», *s1*] → [«*Авторизация*», «*Чтение запроса*»] → [«*Авторизация*», «*Авторизация*»] → [«*Авторизация*», «*Ответ клиенту*»] → [«*Авторизация*», *s2*] → [«*Возврат карты*», *s2*] → [«*Главное меню*», *s2*] → [«*Запрос Баланса*», *s2*]. В квадратных скобках указаны состояния автоматов *AClient* и *AServer*; *s1* – стартовое состояние автомата, а *s2* – завершающее.

Из предложенной последовательности переходов следует, что достичь состояния «*Запрос баланса*» можно, не пройдя авторизацию, так как в ней отсутствует переход по событию *p3.e10*. Достижение данного состояния таким способом свидетельствует о возможности запросить баланс без прохождения авторизации.

Существует реализация верификатора *Spin*, которая работает на нескольких машинах [16, 17], а также работы, предлагающие альтернативные алгоритмы для распределенной верификации [18]. Создание таких верификаторов было обусловлено распределением памяти между несколькими компьютерами. Однако это приводило к значительным сложностям.

В настоящее время проблема с памятью не является существенной, особенно с появлением 64-битных процессоров и винчестеров значительного объема. Благодаря

этому предпочтение отдается параллельным вычислениям, понимая под ними выполнимость на одном компьютере. Поэтому в 2007 г. появилась версия верификатора *Spin*, предназначенная для многоядерной системы [19, 20]. В ней был реализован алгоритм распараллеливания двойного обхода в глубину.

В настоящей работе предлагается другой алгоритм многопоточной версии двойного обхода в глубину. Приведем основную идею этого алгоритма.

Состояния автомата пересечения автомата модели и автомата *Бюхи*, построенного по *LTL*-формуле, обходят в глубину одновременно несколько потоков. У них существует общее множество посещенных состояний. Каждый поток не имеет собственного стека, а формируется общее «дерево стеков». Путь от корня до листа в таком дереве – стек одного из потоков. Как и при обычном обходе в глубину (*Depth-first search, DFS*), обходятся только непосещенные состояния. 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, обнаружившей 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, при котором поток не сразу покидает состояние, а сначала помогает другому завершить поиск, позволяет им не простаивать. Например, если в графе переходов существует точка сочленения, то поток, посетивший ее первым, 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 с нее. Поиск 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 такой 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.

Анализ эффективности многопоточной версии верификатора проводился путем сравнения ее с однопоточной версией. Для этого были автоматически созданы модели автоматных программ с различным числом переходов и состояний. Для них проверялась выполнимая формула. При этом при обходе в глубину посещались все состояния (рис. 5).

Из рассмотрения графиков следует, что выигрыш многопоточной версии при проверке выполнимых формул появляется на моделях с большим числом переходов. Однако даже на модели с 30000 переходов время верификации невелико.

Эксперименты также показали, что число состояний и переходов в автомате пересечения увеличивается в несколько раз на сложных *LTL*-формулах. Это может приводить к значительному увеличению времени верификации.

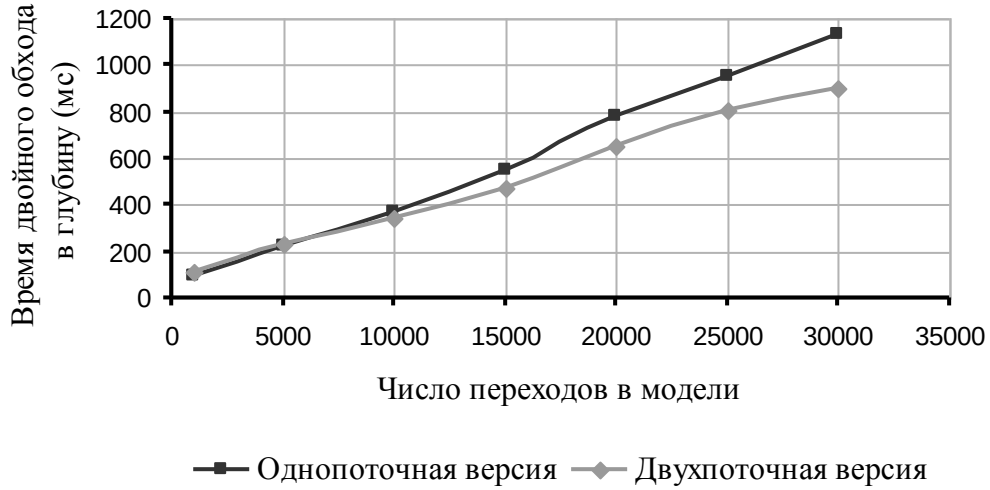

Рис. 5. Зависимость времени работы верификатора от числа переходов автоматной модели

В настоящей работе разработан верификатор автоматных программ, создаваемых при помощи инструментального средства *UniMod*. Он позволяет верифицировать модели программ, которые автоматически строятся верификатором по *XML*-описанию, создаваемому указанным средством по автоматной модели программы. Требования к модели записываются на языке *LTL*. Верификатор предоставляет набор классов и интерфейсов на языке программирования *Java* для проверки выполнимости *LTL*-формул.

При создании верификатора были решены следующие подзадачи:

- трансляция XML-описания модели во внутреннее представление верификатора;
- трансляция LTL-формулы в автомат Бюхи;
- проверка пустоты языка пересечения модели и автомата Бюхи, построенного по отрицанию LTL-формулы.

Для трансляции *LTL*-формулы в автомат *Бюхи* был реализован собственный транслятор и использовался уже существующий транслятор *LTL2BA* [13]. Проверка пустоты языка пересечения двух автоматов осуществляется с помощью двойного обхода в глубину и его многопоточной модификации.

Применение автоматного подхода к написанию программ и созданного верификатора позволяет разрабатывать более надежное программное обеспечение по сравнению с традиционным подходом. Предлагается использовать созданный верификатор для построения и проверки *Unit*-тестов, которые можно запускать на любой стадии жизненного цикла проекта.

Литература

1. Hoffman L. In Search of Dependable Design // Communications of the ACM. – 2008. – V. 51. – № 7. – P. 14–16.

2. Hoffman L. Talking Model-Checking Technology // Communications of the ACM. – 2008. – V. 51. – № 7. – P. 110–112.
3. Кларк Э., Грамберг О., Пелед Д. Верификация моделей программ: Model Checking. М.: МЦНМО, 2002. – Режим доступа: http://is.ifmo.ru/verification/_klark_gamberg_pered_verification.djvu
4. Корнеев Г.А., Парфенов В.Г., Шалыто А.А. Верификация автоматных программ / Тезисы докладов Международной научной конференции, посвященной памяти профессора А. М. Богомолова. «Компьютерные науки и технологии». – Саратов: СГУ, 2007. – С. 66–69. – Режим доступа: http://is.ifmo.ru/verification/_KNIT-2007.pdf
5. 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. – Режим доступа: http://is.ifmo.ru/verification/_ltl_aut_ver_1.pdf
6. Шалыто А.А. Switch-технология. Алгоритмизация и программирование задач логического управления. – СПб: Наука, 1998. – Режим доступа: http://is.ifmo.ru/books/switch/1/
7. Гуров В.С., Мазин М.А., Нарвский А.С., Шалыто А.А. UML. SWITCH-технология. Eclipse // Информационно-управляющие системы. – 2004. – № 6. – С. 12–17. – Режим доступа: http://is.ifmo.ru/works/UML-SWITCH-Eclipse.pdf
8. Harel D., et al. Statemate: A Working Environment for the Development of Complex Reactive Systems // IEEE Trans. Software Eng. –1990. – № 4. – P. 403–414.
9. Разработка технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода. Второй этап. СПбГУ ИТМО, 2007. – Режим доступа: http://is.ifmo.ru/verification/_2007_02_report-verification.pdf
10. 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, 1995. – P. 3–18. – Режим доступа: http://citeseer.ist.psu.edu/gerth95simple.html
11. Courcoubetis C., Vardi M., Wolper P., Yannakakis M. Memory-Efficient Algorithms for the Verification of Temporal Properties / Formal Methods in System Design. – 1992. – P. 275–288.
12. Хопкрофт Д., Мотвани Р., Ульман Д. Введение в теорию автоматов, языков и вычислений. – М.: Вильямс, 2002.
13. LTL 2 BA project. – Режим доступа: http://www.lsv.ens-cachan.fr/~gastin/ltl2ba/
14. Егоров К.В., Райков П.М. Игра «Побег». – СПбГУ ИТМО, 2007. – Режим доступа: http://is.ifmo.ru/unimod-projects/la_redada/
15. Козлов В.А., Комалева О.А. Моделирование работы банкомата. – СПбГУ ИТМО, 2006. – Режим доступа: http://is.ifmo.ru/unimod-projects/bankomat/
16. Lerda F., Sisto R. Distributed-Memory Model Checking with SPIN / Proc. of the 5th International SPIN Workshop. – 1999. – P. 3–17.
17. Barnat J., Brim L., Stribrna J. Distributed LTL Model-Checking in SPIN // Lecture Notes in Computer Science. – 2001. – № 2057. – P. 1–17.
18. Lafuente A. Simplified distributed LTL model checking. Technical Report 00176, Institut fur Informatik. University Freiburg. Germany, 2002.
19. Holzmann G.J., Bošnački D. The Design of a Multi-Core Extension of the SPIN Model Checker // IEEE Transactions on Software Engineering. – 2007. – V. 33. – Issue 10. – P. 659–674.
20. Holzmann G.J. A Stack-Slicing Algorithm for Multi-Core Model Checking // Electronic Notes in Theoretical Computer Science (ENTCS). – 2008. – V. 198. – Issue 1. – P. 3–16.