Санкт-Петербургский государственный университет информационных технологий, механики и оптики
Кафедра компьютерных технологий

Б.Р. Яминов

Сравнение методов верификации *UniMod*-моделей

Магистерская диссертация

Научный руководитель – профессор А.А. Шалыто

Санкт-Петербург
2009

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В настоящее время программные и аппаратные системы занимают все большее место в жизни человека и используются повсюду: в медицине, транспорте, бытовой технике, космических полетах и т. д. Даже в легковых автомобилях наличие бортовых компьютеров уже не удивляет, не говоря уже о самолетах и кораблях. Таким образом, все больше ответственности возлагается на программы, управляющие теми или иными устройствами.

В результате постоянно увеличивающейся автоматизации жизни человека все более остро возникает необходимость заранее обнаруживать и устранять ошибки в управляющих программах. Причем, чем раньше ошибки будут найдены, тем меньше вреда они принесут. Это касается не только безопасности, но и экономической стороны вопроса: если допущенная при проектировании устройства ошибка будет найдена лишь после запуска производства, то возникнет необходимость в исправлении ошибки не только в проекте, но и во всех уже произведенных устройствах.

Автоматный подход [1] хорошо подходит для программирования управляющих систем, поскольку автоматы позволяют просто и наглядно представлять поведение системы.

Таким образом, возникла необходимость автоматически проверять соблюдение управляющей автоматной программой заданных свойств. Это стало бы неоспоримым аргументом для использования автоматного подхода при программировании управляющих систем.

В университете СПбГУ ИТМО на кафедре «Компьютерные технологии» проводятся исследования в этой области. При этом было разработано несколько методов верификации управляющих систем, реализованных при помощи автоматного подхода. Будем называть такие системы «автоматными» [2]. Более того, для каждого метода было разработано инструментальное средство, позволяющее **автоматически**

доказывать или опровергать темпоральные свойства автоматной системы. Будем называть такие инструментальные средства «верификаторами».

В настоящей работе *разработан новый метод верификации* управляющих программ, построенных на основе автоматного подхода. Он на примерах сравнивается с другими методами верификации [2 – 5].

ГЛАВА 1. МЕТОД ВЕРИФИКАЦИИ *UNIMOD.VERIFIER*

В этой главе сначала объясняется смысл задачи верификации автоматных систем. После этого описывается метод и инструментальное средство *UniMod.Verifier* и приводится пример его использования.

1.1. Постановка задачи и описание подхода

1.1.1. Верификация и *Model Checking*

Задача верификации состоит в том, чтобы проверить, удовлетворяет ли программа предъявляемым к ней требованиям. Для решения этой задачи существует несколько подходов, один из которых – *Model Checking* [6]. Это подход, который позволяет автоматически доказывать или опровергать свойства программы. При этом использования могут формулироваться и проверяться темпоральные свойства [6] – свойства, которые описывают поведение программы во времени.

Подход *Model Checking* основывается на том, что число элементарных состояний программы конечно. Поэтому теоретически можно представить работу программы как граф переходов из одних элементарных состояний в другие. Такой граф называют моделью *Крипке* [6]. В каждом элементарном состоянии выполняется определенный набор предикатов. Тогда, если проверяемое свойство сформулировано в терминах таких предикатов, то можно автоматически исследовать граф состояний на предмет выполнения свойства. Более того, если свойство не выполняется, можно определить состояния или цепочки состояний, приводящие к нарушению свойства.

Хотя теоретически все программы имеют конечное число состояний, ограниченное объемом выделенной программе памяти, для любой нетривиальной программы это число слишком велико. И верификатору не хватило бы ресурсов и времени для того, чтобы обойти все состояния и

проверить выполнимость свойства программы. Это не позволяет применять подход *Model Checking* к программам непосредственно.

Для того чтобы решить эту проблему, по *программе строится ее модель*. Модель может абстрагироваться от необязательных деталей реализации программы и описывать только существенные моменты. Полученная модель имеет меньшую сложность, меньшее число состояний и поддается автоматической проверке.

Таким образом, при использовании подхода *Model Checking*, выделяются следующие этапы проверки свойства программы.

1. Построение модели программы.
2. Формирование требований к построенной модели с использованием темпоральной логики, которые должны отражать требования к исходной программе.
3. Автоматическая проверка требований на модели. Если требования не выполняются, то выдается отчет о том, как была получена ошибка в модели.
4. Интерпретация ошибки в модели в термины исходной программы.

Из изложенного следует, что хотя *Model Checking* и позволяет автоматически верифицировать свойства, реально требуется много ручной работы, и обычно лишь третий шаг выполняется автоматически. Таким образом, верификация программы – это трудоемкая ручная работа, и не исключены ошибки при выполнении первого, второго и четвертого шагов.

Для автоматической проверки темпорального свойства на модели (третий шаг) в мире было разработано много верификаторов [7], таких как, например, *SPIN*, *SMV*, *Bogor*, причем для многих из них код открыт. Эти верификаторы, как правило, верифицирует свойства программ, записанные с использованием лишь определенного типа темпоральной логики, из которых можно выделить два основных: *LTL* и *CTL* [6]. Каждый верификатор может

обрабатывать модели, описанные на его входном языке. Например, для верификатора *SPIN* – это язык *Promela*, а для верификатора *Bogor* – язык *BIR*.

Таким образом, например, для того чтобы верифицировать программу с помощью верификатора *SPIN*, требуется описать модель программы на языке *Promela*, и задать свойства для проверки в темпоральной логике *LTL*.

1.1.2. Применение *Model Checking* для верификации автоматных систем

При использовании автоматного подхода в программе выделяется управляющая система, источники событий и объекты управления. Основная логика работы программы моделируется в управляющей системе, тогда как детали реализации различной функциональности скрываются в объектах управления и источниках событий. Таким образом, в автоматной программе при ее проектировании строится модель поведения, отделенная от деталей реализации.

В общем случае ставится задача верификации управляющей системы, состоящей из системы взаимосвязанных автоматов. Так как обычно такая система содержит не так много элементарных состояний, как программа, построенная традиционным путем, то такая система может быть верифицирована и либо без построения более абстрактной модели, либо с ее автоматическим построением. Рассматриваемые в настоящей работе верификаторы были созданы для автоматической верификации автоматных систем.

Разработанные верификаторы автоматных систем используют разные подходы, однако существуют и общие вопросы к верификации автоматных систем.

Первый из них – использовать существующий верификатор программ, или разработать новый верификатор. Выбор существующего верификатора определяет темпоральную логику, в которой будет возможность формулировать свойства автоматной системы. Как правило, каждый из

существующих верификаторов работает лишь с одной логикой. Кроме того, при использовании существующего верификатора, необходимо преобразовывать автоматную систему во входной язык верификатора.

Второй вопрос – выделение элементарных состояний автоматной системы, или, другими словами, преобразование исходной системы в модель Крипке. Во время работы автоматной системы происходят переходы автоматов в состояния, обрабатываются события, вызываются выходные воздействия, управление передается в другие автоматы. При этом можно считать переход автомата из одного состояния в другое за атомарное действие, а можно разбить его на несколько элементарных действий (вычисление условия на переходе, вызов каждого выходного воздействия, вызов вложенного автомата и т. д.) Различные методы верификации разбивают переходы на элементарные состояния по-разному.

При использовании известного верификатора, выделение элементарных состояний происходит на этапе трансляции автоматной системы во входной язык верификатора. На входном языке верификатора программируются все элементарные состояния и переходы между ними.

Третий вопрос – задание набора предикатов, которые могут использоваться в темпоральных формулах. При этом чем шире этот набор, тем более точные свойства автоматной системы можно верифицировать. Примеры предикатов: «автомат `A` находится в состоянии `s1`», «произошло событие `e1`», «было вызвано действие `z1`» и т. д.

Четвертый вопрос – интерпретация отчета об ошибке (контпримера) и преобразование его в термины исходной автоматной системы. При использовании существующих верификаторов, отчет об ошибке выдается в терминах модели, сформулированной на входном языке верификатора. Возникает необходимость преобразовать этот отчет в термины исходной автоматной системы и отобразить пользователю верификатора.

Поэтому в настоящей работе разрабатывается верификатор автоматных программ, построенных с помощью инструментального средства *UniMod* [9].

Это инструментальное средство, позволяющее визуально создавать автоматные программы, а также запускать их. При этом автоматная система изображается на *UML*-диаграммах, а объекты управления и источники событий программируются на языке программирования *Java*.

Разработанный верификатор *UniMod.Verifier* основан на известном верификаторе *Bogor*, так как этот верификатор позволяет расширять входной язык за счет введения новых типов данных.

1.2. Инструментальное средство *UniMod.Verifier*

1.2.1. Общее описание

Инструментальное средство *UniMod.Verifier* использует существующий верификатор *Bogor* [10, 11]. Входной язык этого верификатора, который называется *BIR*, как отмечено выше, может быть расширен новыми, в том числе сложными, типами данных. Новый тип представляет собой класс, который имеет свое внутреннее состояние. У класса можно вызывать действия, которые могут изменить внутреннее состояние, или запрашивать значения тех или иных внутренних переменных. При условии выполнения новым классом некоторых условий, верификатор *Bogor* может верифицировать модели, которые используют объекты нового класса.

Возможность расширения входного языка позволяет абстрагироваться от лишних деталей описания верифицируемой модели. Это приводит к меньшему числу элементарных состояний модели, а, следовательно, к более простой и быстрой верификации.

В *UniMod.Verifier* был реализован новый класс, содержащий описание автоматной системы. Этот класс имеет лишь одно действие «`step`»: обработать очередное событие. При вызове этого действия класс недетерминированно выбирает событие и передает его на обработку автоматной системе. Благодаря использованию нового класса, описание модели на входном языке верификатора *Bogor* сводится к тривиальному

бесконечному циклу, состоящему из одного состояния. В этом состоянии у объекта нового класса вызывается действие `step`. На языке *BIR* это записывается следующим образом:

```
AutomataModel.type model;

main thread MAIN() {
  loc init:
    do invisible {
      model := AutomataModel.create();
    } goto loop;

  loc loop:
    do {
      AutomataModel.step(model);
    } goto loop;
}
```

Основное достоинство такого подхода состоит в том, что описание автоматной системы на входном языке верификатора стало тривиальным и одинаковым для любой автоматной системы. Поведение модели и хранение состояний автоматов реализуется программно, и один раз для всех автоматных систем. Отсутствует необходимость генерировать новые промежуточные входные данные для каждой верифицируемой автоматной системы.

Кроме того, в *UniMod.Verifier* используется еще одно оригинальное решение. При использовании известных верификаторов поведение автоматной системы необходимо программировать на входном языке внешнего верификатора (например, *NuSMV*, *SPIN*). Это поведение может не совпадать с правилами интерпретации автоматных систем инструментальным средством *UniMod*. В этом случае может возникнуть проблема, когда верификатор доказал некоторое свойство автоматной системы, которое не выполняется при применении интерпретатора *UniMod*.

UniMod.Verifier при верификации использует интерпретатор автоматных систем *UniMod*. При этом гарантируется, что верифицируется то же самое, что затем исполняется инструментальным средством *UniMod*.

Схема взаимодействия верификатора *Bogor*, инструментального средства *UniMod* и *UniMod.Verifier* изображена на рис. 1.

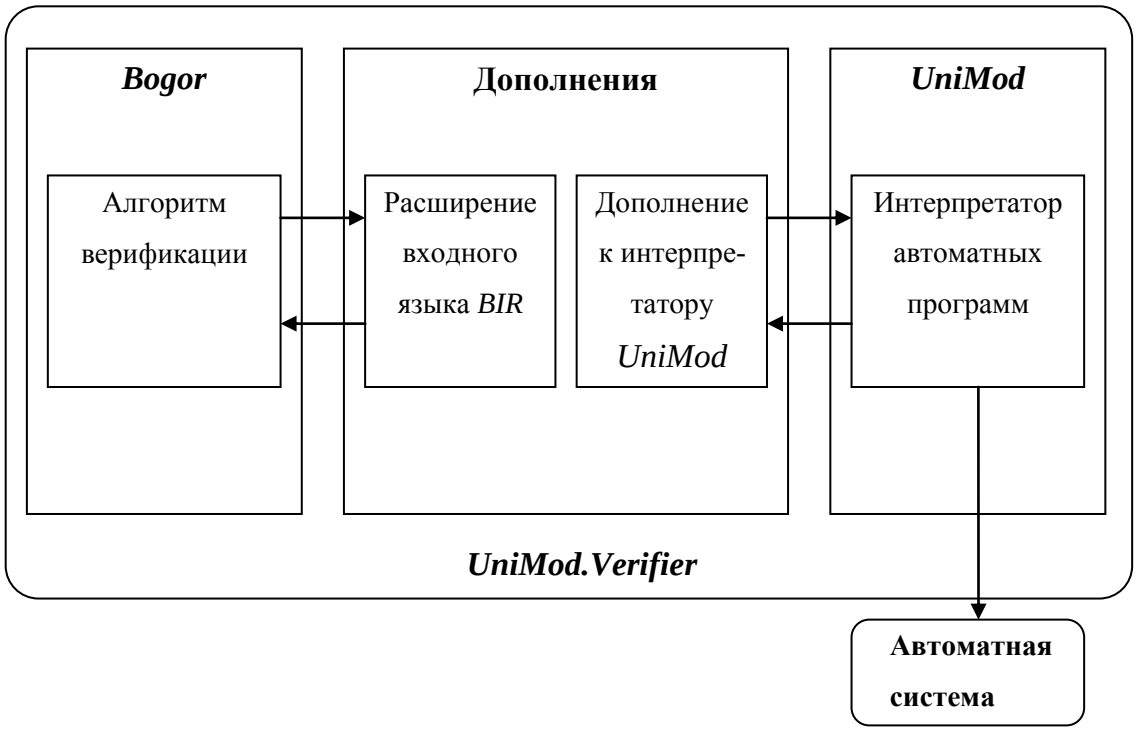

Рис. 1. Схема взаимодействий компонент верификатора *UniMod.Verifier*

1.2.2. Выделение атомарных состояний

При верификации программы необходимо точно определять ее глобальное состояние на каждом шаге верификации, для того чтобы остановить процесс, когда программа вернулась в уже пройденное состояние.

В верификаторе *UniMod.Verifier* глобальным состоянием автоматной системы считается набор текущих состояний, содержащихся в ней автоматов. Таким образом, во-первых, предполагается, что в разные моменты времени автоматная система ведет себя одинаково, если в эти моменты автоматы системы находятся в одних и тех же состояниях. Во-вторых – что переходы автоматов из одних состояний в другие происходят атомарно. Точнее – обработка одного события автоматной системой происходит за один шаг.

Вообще говоря, можно разделять каждый переход на множество действий, таких как вычисление входных переменных, вызов каждого выходного воздействия, и т. д. Однако в верификаторе *UniMod.Verifier* не делается такого деления на элементарные состояния.

1.2.3. Верифицируемые свойства

Выразительные возможности свойств, которые могут проверяться с использованием применяемого верификатора, зависят от набора поддерживаемых методом предикатов. К таким предикатам относятся, например, «произошло событие `e1`», «было вызвано действие `z1`», «автомат `A1` находится в состоянии `s1`». Кроме того, важен тип темпоральной логики, при помощи которой эти предикаты соединяются в формулы, предназначенные для верификации.

Верификатор *Bogor*, а, следовательно, и *UniMod.Verifier*, позволяют верифицировать свойства, сформулированные в темпоральной логике *LTL*.

В качестве предикатов автоматное расширение языка *BIR* предоставляет следующие функции:

- `wasEvent(e)` – верно, если в последнем шаге было выбрано для обработки событие `e`, и неверно в противном случае;
- `wasInState(sm, s)` – выполняется, если перед последним шагом автомат `sm`, находился в состоянии `s`;
- `isInState(sm, s)` – выполняется, если после совершения последнего шага автомат `sm` находится в состоянии `s`;
- `cameToState(sm, s)` – выполняется, если после совершения последнего шага автомат `sm` изменил свое состояние на `s`. Это то же, что `(isInState(sm, s) && !wasInState(sm, s))`;
- `cameToFinalState()` – верно, если после совершения шага головной автомат модели вошел в свое конечное состояние. Это означает, что автоматная система завершила работу.
- `wasAction(z)` – выполняется, если в ходе выполнения шага было вызвано выходное воздействие `z`;
- `wasFirstAction(z)` – выполняется, если в ходе выполнения шага первым вызванным действием было `z`;

- `wasLastAction(z)` – выполняется, если в ходе выполнения шага последним вызванным действием было `z`;
- `getActionIndex(z)` – показывает номер действия в списке действий, вызванных в ходе выполнения последнего шага. Этот предикат предназначен для того, чтобы формулировать утверждения, задающие порядок вызова действий объектов управления в автоматной программе;
- `wasTrue(g)` – выполняется, если в ходе выполнения последнего шага один из переходов был помечен условием `g`, и его значение было определено как `True`. Выражение `g` в общем случае описывает формулу, а не одну переменную. Например, `g = «!o1.x1 && o1.x2»`;
- `wasFalse(g)` – верно, если в ходе выполнения последнего шага на одном из переходов встречается условие `g`, и его значение было определено как `False`.

1.2.4. Преобразование контрпримера

Поскольку верификатор работает напрямую с автоматной системой, контрпример сразу же выражен в терминах исходной системы, и преобразования не требуется.

1.2.5. Описание инструментального средства

Опишем, как верифицировать автоматные программы при помощи *UniMod.Verifier*.

Этот верификатор предназначен для работы с диаграммами автоматных программ, разработанными в инструментальном средстве *UniMod*. Пусть задана автоматная программа в среде *UniMod*, которую требуется верифицировать. Подробные инструкции по разработке программ в указанной среде приведены на сайте http://unimod.sourceforge.net. Ниже это описано кратко.

Сначала необходимо создать *XML*-описание автоматной программы. Для этого на диаграмме одного из автоматов надо нажать правой кнопкой мыши, и в контекстном меню выбрать *Export to runtime XML*. Это действие изображено на рис. 22.

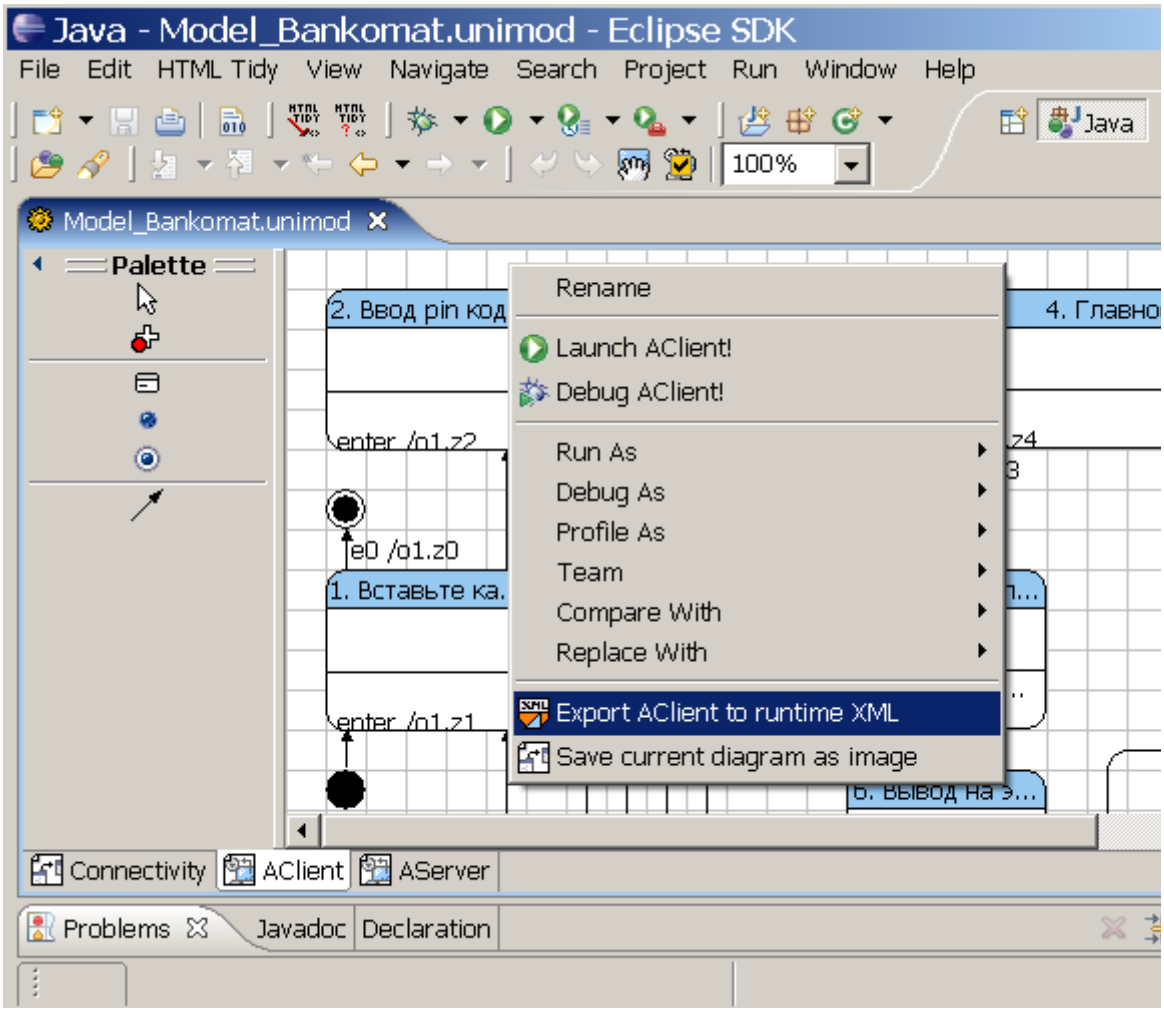

Рис. 2. Создание *XML*-описания автоматной программы

В открывшемся диалоге необходимо выбрать головной автомат программы, и указать путь, куда сохранить *XML*-описание.

Далее необходимо записать формулу для верификации. Эта формула сохраняется в файл `Unimod.bir`, который содержит, кроме того, информацию о верифицируемой модели. Для записи формулы в рассматриваемом верификаторе используются следующие операторы:

- `LTL.always` – (G) всегда;
- `LTL.eventually` – (F) когда-нибудь;
- `LTL.next` – (X) в следующий момент во времени;

- `LTL.until` – (`U`) до тех пор, пока;
- `LTL.weakUntil` – (`W`) до тех пор пока или всегда: `p W q = (p U q) || G (p && !q)`.
- `LTL.release` – (`R`) освобождение: `p R q = !(!p U !q)`;
- `LTL.negation` – отрицание;
- `LTL.equivalence` – эквивалентно;
- `LTL.implication` – следует;
- `LTL.conjunction` – И;
- `LTL.disjunction` – ИЛИ.

Формула записывается в файл `Unimod.bir` в виде функции. Объясним на примере формат этой функции:

```
fun NoPinNoMoney() returns boolean =
  LTL.temporalProperty(
    Property.createObservableDictionary(
      Property.createObservableKey("correct_pin",
        AutomataModel.wasEvent(model, "e10")),
      Property.createObservableKey("give_money",
        AutomataModel.wasAction(model, "o1.z10"))
    ),

    LTL.weakUntil (
      LTL.negation(LTL.prop("give_money")),
      LTL.prop("correct_pin")
    )
  );
```

Указанная функция выражает темпоральную формулу `!o1.z10 W e10`. В начале функции объявляются предикаты. Например, "`correct_pin`" – имя предиката, а `AutomataModel.wasEvent(model, "e10")` – его значение. Такой предикат верен тогда, когда последнее обработанное событие `e10`.

После объявления предикатов, записывается, как вложенные вызовы функций, формула с использованием этих предикатов.

Теперь вызывается верификатор. Это делается командой:

```
verifier.cmd Test.connectivity NoPinNoMoney
```

Здесь `Test.connectivity` – имя файла описания автоматной системы в формате *UniMod* с расширением `connectivity`, а `NoPinNoMoney` – имя функции для верификации, описанной в файле `Unimod.bir`.

В результате вызова этой команды, в стандартный вывод будет печататься служебная информация о процессе верификации. Когда верификация завершится, либо выведется сообщение об успешном завершении, либо будет выведен контрпример.

Контрпример описывает по шагам состояния автоматов, например:

```
Stack of transitions leading to the error:
Model [ step [0] event [null] guards [null] transitions [null] actions [null]
states [null] ] fsaState [bad$accept_init]
Model [ step [0] event [] guards [] transitions [] actions [] states
[(/AClient:9. Запрос денег/AServer) - (Top); (/AClient:5. Запрос
Баланса/AServer) - (Top); (/AClient:3. Авторизация/AServer) - (Top);
(/AClient) - (Top)] ] fsaState [bad$accept_init]

Model [ step [1] event [*] guards [] transitions [s1#1. Вставьте
карту#*#true] actions [o1.z1] states [(/AClient:9. Запрос денег/AServer) -
(Top); (/AClient:5. Запрос Баланса/AServer) - (Top); (/AClient:3.
Авторизация/AServer) - (Top); (/AClient) - (1. Вставьте карту)] ] fsaState
[bad$accept_init]

Model [ step [2] event [e6] guards [true->true] transitions [1. Вставьте
карту#2. Ввод pin кода#e6#true] actions [o1.z2] states [(/AClient:9. Запрос
денег/AServer) - (Top); (/AClient:5. Запрос Баланса/AServer) - (Top);
(/AClient:3. Авторизация/AServer) - (Top); (/AClient) - (2. Ввод pin кода)] ]
fsaState [bad$accept_init]

Model [ step [3] event [e2] guards [true->true] transitions [2. Ввод pin
кода#13. Возврат карты#e2#true] actions [o1.z13] states [(/AClient:9. Запрос
денег/AServer) - (Top); (/AClient:5. Запрос Баланса/AServer) - (Top);
(/AClient:3. Авторизация/AServer) - (Top); (/AClient) - (13. Возврат карты)]
] fsaState [bad$accept_init]

Model [ step [4] event [e7] guards [true->true] transitions [13. Возврат
карты#1. Вставьте карту#e7#true] actions [o1.z1] states [(/AClient:9. Запрос
денег/AServer) - (Top); (/AClient:5. Запрос Баланса/AServer) - (Top);
(/AClient:3. Авторизация/AServer) - (Top); (/AClient) - (1. Вставьте карту)]
] fsaState [bad$accept_init]

Done!
```

1.3. Пример использования

Приведем пример использования верификатора *UniMod.Verifier*. Будем верифицировать различные свойства автоматной системы, моделирующей работу банкомата.

1.3.1. Описание модели банкомата

Банкомат – это устройство, автоматизирующее операции по выдаче и переводу денег, хранящихся в банке, лицу, которому они принадлежат.

Идентификация каждого клиента происходит при помощи имеющейся у него карты банка и соответствующего карте секретного *pin*-кода. Банкомат осуществляет следующие операции:

- идентифицирует клиента;
- позволяет посмотреть доступные средства;
- позволяет снимать деньги;
- связывается с банком.

Автоматная модель банкомата состоит из двух автоматов. Автомат `AClient` управляет пользовательским интерфейсом, а автомат `AServer` проводит операции со счетами и связывается с банком. Кроме того, в системе работают следующие источники событий и объекты управления. Источники событий:

- `HardwareEventProvider` – системные события, генерируемые оборудованием.
- `HumanEventProvider` – события, инициируемые пользователем.
- `ServerEventProvider` – ответы на запросы, поступающие от сервера.
- `ClientEventProvider` – запросы, поступающие на сервер.

Объекты управления:

- `FormPainter` – визуализация работы.
- `ServerQuery` – отправляет запросы на сервер.
- `ServerReply` – отвечает на клиентские запросы.

Модель банкомата разработана при помощи инструментального средства *UniMod*. На рис. 3 изображена схема связей модели банкомата. На этой схеме приведены все события и выходные воздействия, содержащиеся в модели.

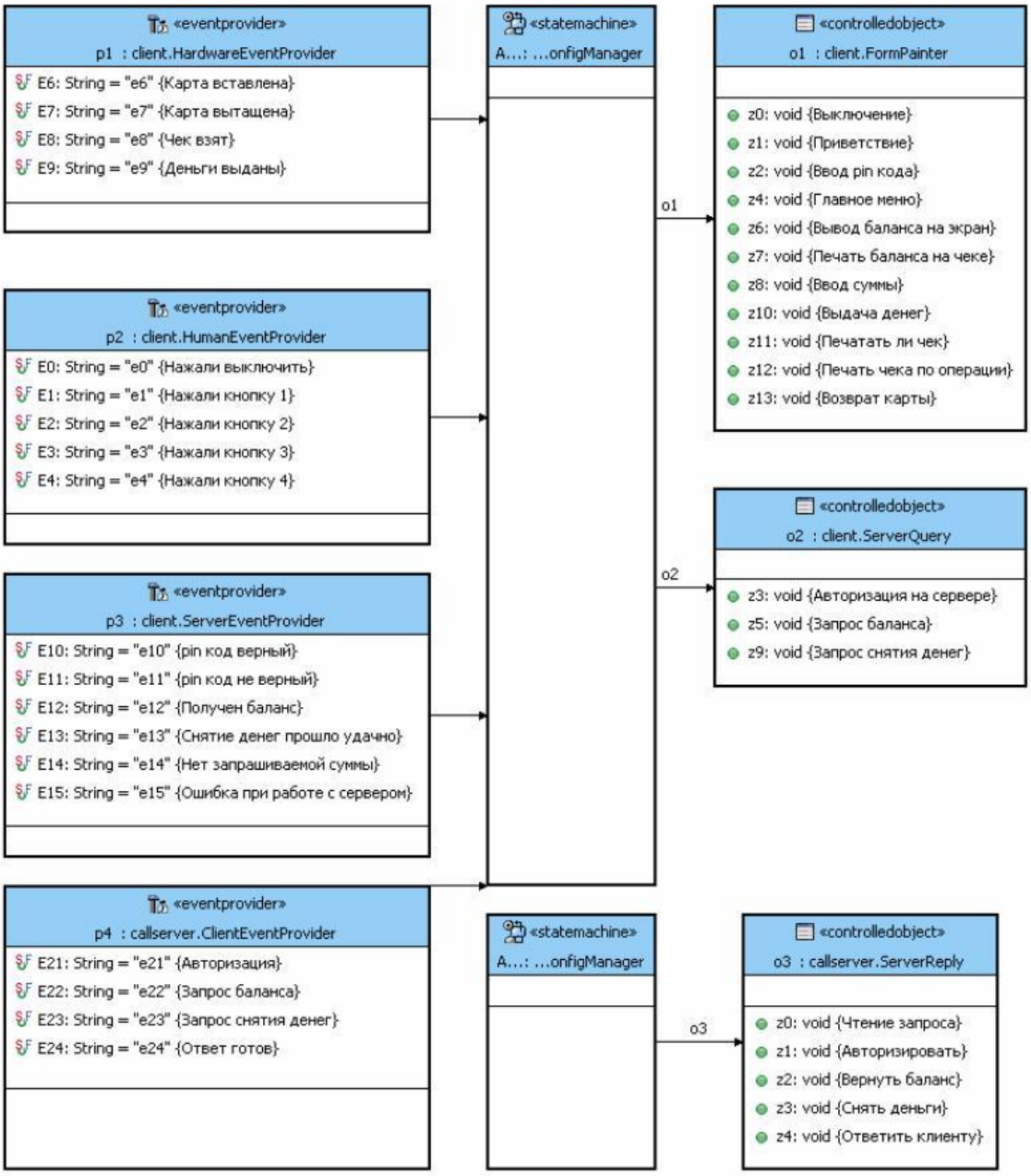

Рис. 3. Схема связей автоматов, источников событий и объектов управления в модели банкомата

На рис. 4 приведен граф переходов автомата `AClient`.

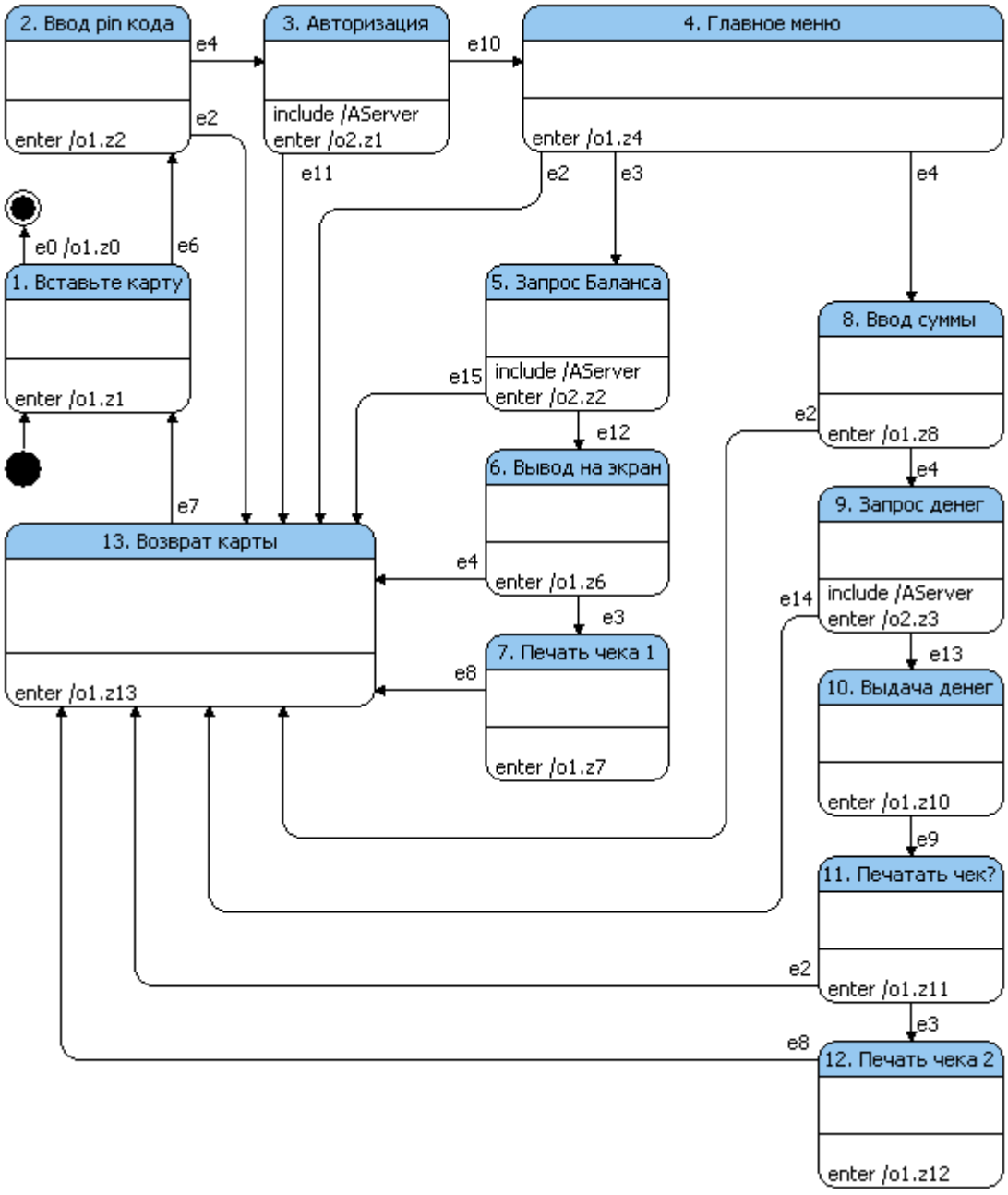

Рис. 4. Автомат `AClient`

На рис. 5 приведен граф переходов автомата `AServer`, посылающего запросы на сервер.

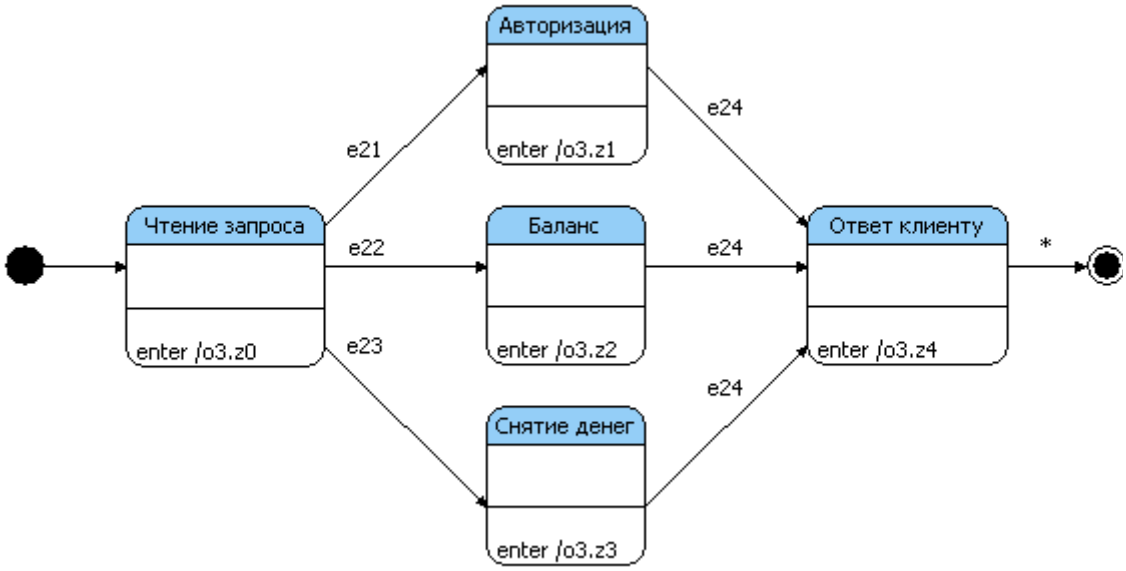

Рис. 5. Автомат `AServer`

1.3.2. Верификация свойств банкомата

Проверим свойство: «Пользователь банкомата не получит денег, пока не введет правильный *PIN*-код». Словесная формулировка свойства переводится в темпоральную логику *LTL* непосредственно:

```
[не выдадут деньги] U [введет правильный PIN-код]
```

Здесь `U` – темпоральный оператор `Until` («пока не»). Как показано на рис. 34, в *UniMod*-банкомате выдача денег происходит действием `o1.z10`, а правильно введенный *PIN*-код характеризуется событием `e10`. При этом формула для верификации принимает следующий вид:

```
!o1.z10 U e10
```

Здесь предикат `o1.z10` означает, что было выполнено действие `o1.z10`, а предикат `e10` – что произошло событие `e10`. Запишем эту *LTL*-формулу на входном языке верификатора *Bogor*:

```
LTL.temporalProperty(
  Property.createObservableDictionary(
    Property.createObservableKey("correct_pin",
        AutomataModel.wasEvent(model, "e10")),
    Property.createObservableKey("give_money",
        AutomataModel.wasAction(model, "o1.z10"))
  ),

  LTL.weakUntil(
```

```
            LTL.negation(LTL.prop("give_money")),
            LTL.prop("correct_pin")
        )
    );
```

Здесь предикат «было выполнено действие `o1.z10`» записан в виде `AutomataModel.wasAction(model, "o1.z10")` и сохранен под ключом «`give_money`». Аналогично, предикат «произошло событие `e10`» записан в виде `AutomataModel.wasEvent(model, "e10")` и сохранен под ключом «`correct_pin`». Эти ключи затем использованы для записи самой темпоральной формулы. Отметим, что вместо темпорального оператора `Until` здесь используется его модификация `weakUntil`. Отличие между ними состоит в том, что `p Until q` требует, чтобы `q` когда-нибудь выполнилось, в то время как `p weakUntil q` этого не требует. Это оправдано в данном случае, так как совершенно не обязательно, что пользователь когда-нибудь введет правильный *PIN*-код.

При верификации созданной формулы верификатор *UniMod.verifier* выдает следующий результат:

```
Transitions: 1, States: 1, Matched States: 0, Max Depth: 1, Errors found: 0,
Used Memory: 2MB
Transitions: 63, States: 41, Matched States: 22, Max Depth: 14, Errors found:
0, Used Memory: 1MB
Total memory before search: 765a008 bytes (0,73 Mb)
Total memory after search: 1a202a688 bytes (1,15 Mb)
Total search time: 703 ms (0:0:0)
States count: 41
Matched states count: 22
Max depth: 14
Done!
Verification successful!
```

Таким образом, требование, чтобы банкомат не выдавал деньги до введения правильного *PIN*-код, выполняется в системе автоматов, управляющих банкоматом.

Проверим еще одно свойство: «Пользователь не получит деньги, если он запросил больше, чем у него на счете». Утверждение можно формализовать следующим образом:

`[запрашивается больше денег, чем на счете]` → `[деньги не выдадутся]`

Здесь «→» – оператор импликации («тогда»).

При запросе суммы денег большей, чем на счете, происходит событие `e14`. Деньги выдаются действием `o1.z10`. Поэтому для рассматриваемого свойства можно записать следующую темпоральную формулу:

```
G (e14 → !o1.z10)
```

Это означает, что «всегда, если произошло событие `e14`, то не происходит действие `o1.z10`». Записав полученное утверждение во входной файл верификатора, получим:

```
LTL.temporalProperty (
      Property.createObservableDictionary (
            Property.createObservableKey("too_much",
AutomataModel.wasEvent(model, "e14")),
            Property.createObservableKey("give_money",
AutomataModel.wasAction(model, "o1.z10"))
      ),

      LTL.always(
            LTL.implication (
                  LTL.prop("too_much"),
                  LTL.negation(LTL.prop("give_money"))
            )
      )
);
```

Верификация этой формулы пройдет успешно:

```
Transitions: 1, States: 1, Matched States: 0, Max Depth: 1, Errors found: 0,
Used Memory: 2MB
Transitions: 7533, States: 3827, Matched States: 3706, Max Depth: 94, Errors
found: 0, Used Memory:
1MB
Total memory before search: 758a568 bytes (0,72 Mb)
Total memory after search: 1a700a144 bytes (1,62 Mb)
Total search time: 2704 ms (0:0:2)
States count: 3827
Matched states count: 3706
Max depth: 94
Done!
Verification successful!
```

Однако данная формула проверяет лишь то, что выдачи денег не произойдет сразу после события `e14`. Возможна ситуация, когда банкомат выдаст больше денег, чем можно выдать данному пользователю, однако это произойдет, например, через некоторое время или после нажатия какой-нибудь комбинации кнопок.

Модифицируем формулу, для того чтобы учесть такие варианты:

```
G (e14 → G !o1.z10)
```

Это означает, что «всегда, если произошло событие `e14`, то никогда не произойдет действия `o1.z10`». На языке *BIR* эта формула записывается в виде:

```
LTL.temporalProperty (
  Property.createObservableDictionary (
    Property.createObservableKey("too_much",
        AutomataModel.wasEvent(model, "e14")),
    Property.createObservableKey("give_money",
        AutomataModel.wasAction(model, "o1.z10"))
  ),

  LTL.always(
    LTL.implication (
      LTL.prop("too_much"),
      LTL.always(LTL.negation(LTL.prop("give_money")))
    )
  )
);
```

Верифицируем эту формулу. Верификатор выдает следующий результат:

```
Transitions: 1, States: 1, Matched States: 0, Max Depth: 1, Errors found: 0,
Used Memory: 2MB
Transitions: 108, States: 80, Matched States: 30, Max Depth: 62, Errors
found: 1, Used Memory: 1MB
[...]
Transitions: 374, States: 244, Matched States: 141, Max Depth: 101, Errors
found: 10, Used Memory: 1MB
Total memory before search: 761 256 bytes (0,73 Mb)
Total memory after search: 1 524 768 bytes (1,45 Mb)
Total search time: 6000 ms (0:0:6)
States count: 244
Matched states count: 141
Max depth: 101

Generating error trace 0...
[...]
Generating error trace 12...

Done!


13 traces were found.
Replaying the trace with least states (#12).

Replaying trace by key: 0
Stack of transitions leading to the error:
Model [ step [0] event [null] guards [null] transitions [null] actions [null]
states [null] ] fsaState [T1_init]
Model [ step [0] event [] guards [] transitions [] actions [] states
[(/AClient:9. Запрос денег/AServer) - (Top); (/AClient:5. Запрос
Баланса/AServer) - (Top); (/AClient:3. Авторизация/AServer) - (Top);
(/AClient) - (Top)] ] fsaState [T1_init]

Model [ step [1] event [*] guards [] transitions [s1#1. Вставьте
карту#*#true] actions [o1.z1] states [(/AClient:9. Запрос денег/AServer) -
(Top); (/AClient:5. Запрос Баланса/AServer) - (Top); (/AClient:3.
Авторизация/AServer) - (Top); (/AClient) - (1. Вставьте карту)] ] fsaState
[T1_init]
Model [ step [2] event [e6] guards [true->true] transitions [1. Вставьте
карту#2. Ввод pin кода#e6#true] actions [o1.z2] states [(/AClient:9. Запрос
денег/AServer) - (Чтение запроса); (/AClient:5. Запрос Баланса/AServer) -
(Снятие денег); (/AClient:3. Авторизация/AServer) - (Чтение запроса);
(/AClient) - (2. Ввод pin кода)] ] fsaState [T1_init]

Model [ step [3] event [e4] guards [true->true] transitions [2. Ввод pin
кода#3. Авторизация#e4#true] actions [o2.z3] states [(/AClient:9. Запрос
```

денег/AServer) - (Чтение запроса); (/AClient:5. Запрос Баланса/AServer) - (Снятие денег); (/AClient:3. Авторизация/AServer) - (Чтение запроса); (/AClient) - (3. Авторизация)]] fsaState [T1_init]

Model [step [4] event [e10] guards [true->true] transitions [3. Авторизация#4. Главное меню#e10#true] actions [o1.z4] states [(/AClient:9. Запрос денег/AServer) - (Чтение запроса); (/AClient:5. Запрос Баланса/AServer) - (Снятие денег); (/AClient:3. Авторизация/AServer) - (Чтение запроса); (/AClient) - (4. Главное меню)]] fsaState [T1_init]

Model [step [5] event [e4] guards [true->true] transitions [4. Главное меню#8. Ввод суммы#e4#true] actions [o1.z8] states [(/AClient:9. Запрос денег/AServer) - (Чтение запроса); (/AClient:5. Запрос Баланса/AServer) - (Снятие денег); (/AClient:3. Авторизация/AServer) - (Чтение запроса); (/AClient) - (8. Ввод суммы)]] fsaState [T1_init]

Model [step [6] event [e4] guards [true->true] transitions [8. Ввод суммы#9. Запрос денег#e4#true] actions [o2.z9] states [(/AClient:9. Запрос денег/AServer) - (Чтение запроса); (/AClient:5. Запрос Баланса/AServer) - (Снятие денег); (/AClient:3. Авторизация/AServer) - (Чтение запроса); (/AClient) - (9. Запрос денег)]] fsaState [T1_init]

Model [step [7] event [**e14**] guards [true->true] transitions [9. Запрос денег#13. Возврат карты#e14#true] actions [o1.z13] states [(/AClient:9. Запрос денег/AServer) - (Чтение запроса); (/AClient:5. Запрос Баланса/AServer) - (Снятие денег); (/AClient:3. Авторизация/AServer) - (Чтение запроса); (/AClient) - (13. Возврат карты)]] fsaState [T0_S2]

Model [step [8] event [e7] guards [true->true] transitions [13. Возврат карты#1. Вставьте карту#e7#true] actions [o1.z1] states [(/AClient:9. Запрос денег/AServer) - (Чтение запроса); (/AClient:5. Запрос Баланса/AServer) - (Снятие денег); (/AClient:3. Авторизация/AServer) - (Чтение запроса); (/AClient) - (1. Вставьте карту)]] fsaState [T0_S2]

Model [step [9] event [e6] guards [true->true] transitions [1. Вставьте карту#2. Ввод pin кода#e6#true] actions [o1.z2] states [(/AClient:9. Запрос денег/AServer) - (Чтение запроса); (/AClient:5. Запрос Баланса/AServer) - (s2); (/AClient:3. Авторизация/AServer) - (Чтение запроса); (/AClient) - (2. Ввод pin кода)]] fsaState [T0_S2]

Model [step [10] event [e4] guards [true->true] transitions [2. Ввод pin кода#3. Авторизация#e4#true] actions [o2.z3] states [(/AClient:9. Запрос денег/AServer) - (Чтение запроса); (/AClient:5. Запрос Баланса/AServer) - (s2); (/AClient:3. Авторизация/AServer) - (Чтение запроса); (/AClient) - (3. Авторизация)]] fsaState [T0_S2]

Model [step [11] event [e10] guards [true->true] transitions [3. Авторизация#4. Главное меню#e10#true] actions [o1.z4] states [(/AClient:9. Запрос денег/AServer) - (Чтение запроса); (/AClient:5. Запрос Баланса/AServer) - (s2); (/AClient:3. Авторизация/AServer) - (Чтение запроса); (/AClient) - (4. Главное меню)]] fsaState [T0_S2]

Model [step [12] event [e4] guards [true->true] transitions [4. Главное меню#8. Ввод суммы#e4#true] actions [o1.z8] states [(/AClient:9. Запрос денег/AServer) - (Чтение запроса); (/AClient:5. Запрос Баланса/AServer) - (s2); (/AClient:3. Авторизация/AServer) - (Чтение запроса); (/AClient) - (8. Ввод суммы)]] fsaState [T0_S2]

Model [step [13] event [e4] guards [true->true] transitions [8. Ввод суммы#9. Запрос денег#e4#true] actions [o2.z9] states [(/AClient:9. Запрос денег/AServer) - (Чтение запроса); (/AClient:5. Запрос Баланса/AServer) - (s2); (/AClient:3. Авторизация/AServer) - (Чтение запроса); (/AClient) - (9. Запрос денег)]] fsaState [T0_S2]

Model [step [14] event [e13] guards [true->true] transitions [9. Запрос денег#10. Выдача денег#e13#true] actions [**o1.z10**] states [(/AClient:9. Запрос денег/AServer) - (Чтение запроса); (/AClient:5. Запрос Баланса/AServer) - (s2); (/AClient:3. Авторизация/AServer) - (Чтение запроса); (/AClient) - (10. Выдача денег)]] fsaState [bad$accept_all]

Done!

Утверждение не выполняется, и верификатор выдал контрпример, показывающий историю работы программы, в которой нарушается утверждение. В контрпримере жирными линиями выделено, что, действительно, на шаге 7 произошло событие `e14`, а после этого на шаге 14 выполнилось действие `o1.z10`. Это нарушило верифицируемое условие. На рис. 6 изображено графическое представление контрпримера.

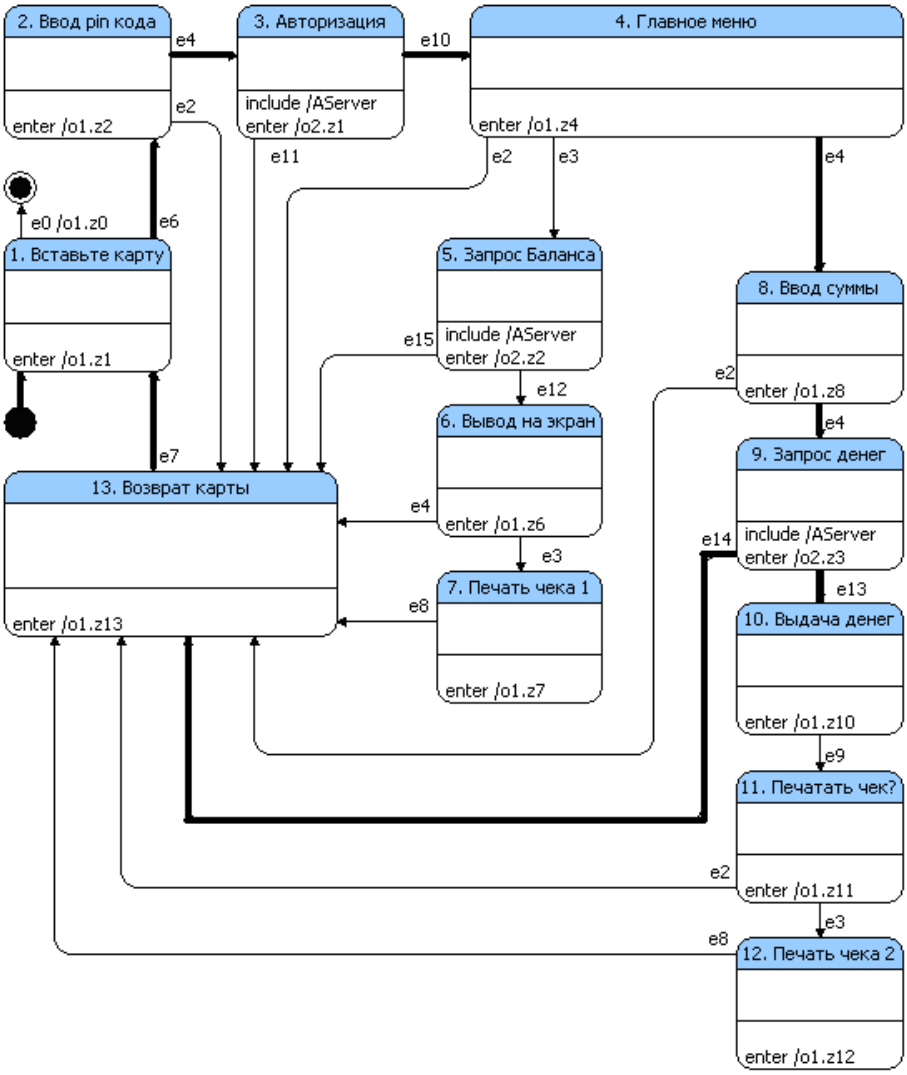

Рис. 6. Контрпример формулы «G (e14 → G !o1.z10)»

Этот рисунок отражает историю, когда пользователь запросил больше денег, чем было у него на карте, а банкомат вернул ему карту и не выдал деньги. После этого пользователь заново ввел карту и запросил на этот раз достаточную сумму денег (событие `e13`), и банкомат выдал деньги. Данная история является правильным поведением банкомата, однако нарушает

заданное требование. Таким образом, требование было сформулировано некорректно, поэтому и возникла ошибка верификации.

Исправим требование, добавив условие, что деньги не выдаются до тех пор, пока не будет запрошена сумма, которую можно снять:

```
G (e14 → (!o1.z10 U e13))
```

Запишем эту формулу на языке *BIR*:

```
LTL.temporalProperty (
      Property.createObservableDictionary (
            Property.createObservableKey("too_much",
AutomataModel.wasEvent(model, "e14")),
            Property.createObservableKey("give_money",
AutomataModel.wasAction(model, "o1.z10")),
            Property.createObservableKey("enough",
AutomataModel.wasEvent(model, "e13"))
      ),

      LTL.always(
            LTL.implication (
                  LTL.prop("too_much"),
                  LTL.weakUntil (
                        LTL.negation(LTL.prop("give_money")),
                        LTL.prop("enough")
                  )
            )
      )
);
```

Верификация этой формулы приведет к следующему результату:

```
Transitions: 1, States: 1, Matched States: 0, Max Depth: 1, Errors found: 0,
Used Memory: 2MB
Transitions: 9996, States: 5154, Matched States: 4846, Max Depth: 94, Errors
found: 0, Used Memory: 1MB
Transitions: 13233, States: 6743, Matched States: 6501, Max Depth: 94, Errors
found: 0, Used Memory: 1MB
Total memory before search: 765 808 bytes (0,73 Mb)
Total memory after search: 1 892 320 bytes (1,8 Mb)
Total search time: 4000 ms (0:0:4)
States count: 6743
Matched states count: 6501
Max depth: 94
Done!
Verification successful!
```

Таким образом, формула была корректно уточнена и верифицирована. Банкомат удовлетворяет предъявленному требованию.

Выводы по главе 1

В этой главе описан новый верификатор автоматных систем. Его преимущество по сравнению с другими верификаторами, состоит в том, что при верификации используется интерпретатор автоматных систем инструментального средства *UniMod*. Поэтому этот верификатор проверяет корректность работы не модели программы, а автоматной системы

исполняемой инструментальным средством *UniMod*. Это позволяет избежать несоответствия между верифицируемой и исполняющейся автоматной системой, а, следовательно, избежать дополнительных ошибок верификации.

Приведен пример верификации автоматной системы, моделирующей работу банкомата. Пример показал, что верификация осуществляется корректно, достаточно просто и удобно.

ГЛАВА 2. ОБЗОР СУЩЕСТВУЮЩИХ ВЕРИФИКАТОРОВ

В этой главе будут описаны инструментальные средства для верификации автоматных систем, которые разработаны на кафедре «Компьютерные технологии» СПбГУ ИТМО в ходе работ по государственному контракту [2 – 5]. В конце главы будет проведено сравнение функциональных возможностей этих средств.

2.1. Инструментальное средство *FSM Verifier*

2.1.1. *Общее описание*

Инструментальное средство *FSM Verifier* [14], разработанное Е.А. Курбацким, основывается на верификаторе *NuSMV*, который верифицирует модели, описанные на языке *SMV*. Схема работы верификатора состоит из следующих стандартных шагов.

1. Преобразовать автоматную систему в модель на языке *SMV*.
2. Преобразовать требования к системе в формулы темпоральной логики.
3. Запустить верификатор *NuSMV*.
4. Преобразовать контрпример к модели в контрпример к исходной системе автоматов.

Эта схема изображена на рис. 7.

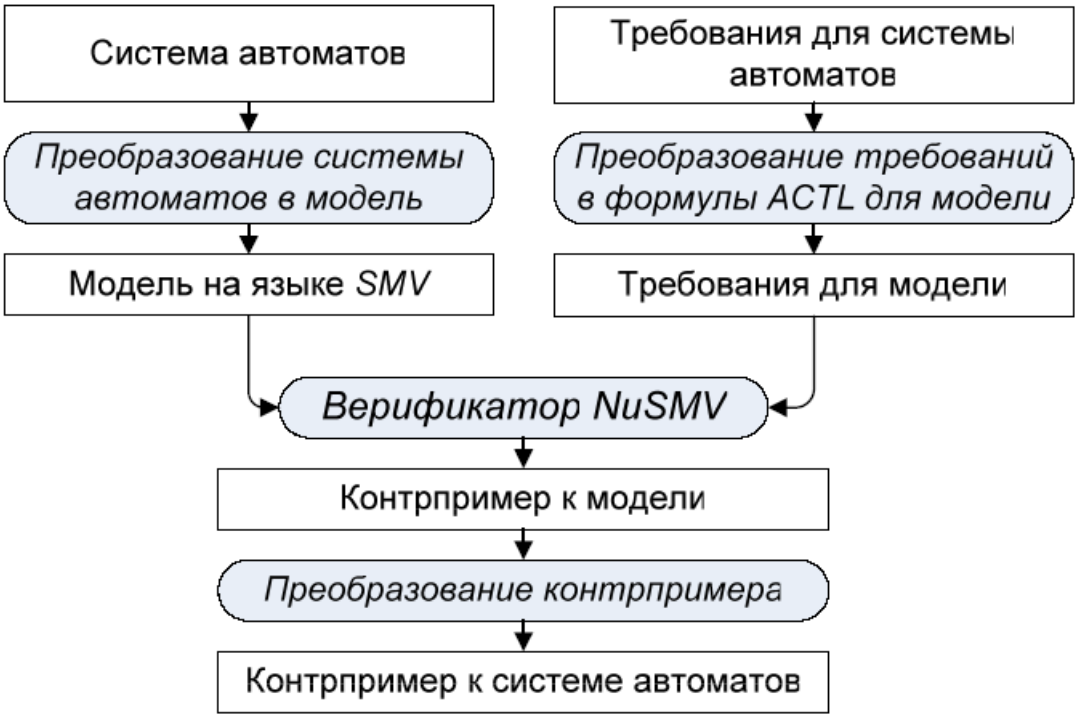

Рис. 7. Схема работы верификатора *FSM Verifier*

Опишем систему автоматов, которая верифицируется рассматриваемым верификатором. Имеется набор конечных детерминированных автоматов с несколькими выходными воздействиями на ребрах и действиями при входе в состояния. Для каждого автомата задается набор событий, которые он может обрабатывать. Для каждого события задается, может ли оно поступать от источника событий или только от автомата.

Каждый переход может содержать событие, при котором он активизируется, и условие, необходимое для активации перехода. Условие представляет собой логическую формулу, содержащую в качестве предикатов входные переменные (`x1`, `x2`) и выражения вида `Ai in sj` (оно выполняется, если автомат `Ai` находится в состоянии `sj`). Входные переменные могут возвращать только булевы значения. Также переход может содержать последовательность выходных воздействий вида `o.zi()` и передачу управления другим автоматам вида `Ai.ej()`.

2.1.2. Выделение атомарных состояний

В рассматриваемом верификаторе автоматах системы выделяются, кроме основных состояний, множество промежуточных состояний, в которые автомат попадает во время перехода из одного состояния в другое. Промежуточное состояние фиксируется каждый раз, когда автомат выполняет одно из следующих действий:

- вызывает выходное воздействие;
- вызывает другой автомат.

На рис. 8 приведен пример выделения промежуточных состояний для одного перехода.

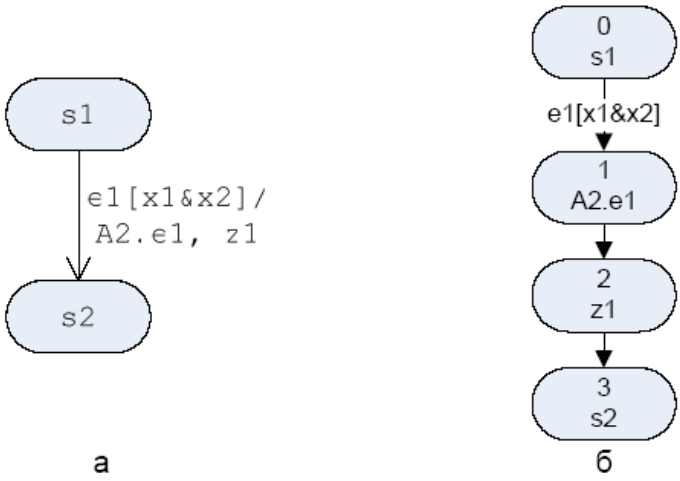

Рис. 8. Выделение промежуточных состояний на переходе. Исходный переход (а), преобразованный переход (б)

2.1.3. Верифицируемые свойства

В рассматриваемом средстве возможна верификация темпоральных формул с использованием следующих предикатов:

- автомат `Ak` находится в состоянии `sj`;
- выполнилось выходное воздействие `zi`;
- произошло событие `ei`.

Инструментальное средство *NuSMV*, а, следовательно, и *FSM Verifier*, позволяет верифицировать свойства, сформулированные в темпоральной

логике *CTL* [3]. В формулах, использующих эту логику, возможно применение следующих темпоральных операторов.

- `AF f` (`Future`) – формула верна для таких состояний, для которых на всех путях, следующих из них, существует состояние, где `f` выполняется.
- `AG f` (`Global`) – формула верна для таких состояний, для которых на всех путях, следующих из них, `f` всегда выполнится.
- `A[f U g]` (`Until`) – формула верна для таких состояний, для которых на всех путях, следующих из них, когда-нибудь выполнится `g`, а до этого всегда будет выполняться `f`.

Кроме того, в формулах рассматриваемого типа можно использовать стандартные логические операторы.

2.1.4. Преобразование контрпримера

Отчет об ошибке верификации, возвращаемый верификатором *NuSMV*, содержит контрпример в терминах построенной модели автоматной системы. Контрпример содержит последовательность элементарных состояний, приводящих к ошибке. Каждое элементарное состояние однозначно определяет состояние или переход в исходной автоматной системе, поэтому можно формально преобразовать контрпример в термины исходной системы автоматов. Подробнее о том, как это реализовано в верификаторе *FSM Verifier* изложено в работе [3].

2.1.5. Описание инструментального средства

Дистрибутив инструментального средства содержит два файла:

- `verifier.jar` – программа, преобразующая систему автоматов в модель на языке *SMV*;
- `counterexample.jar` – программа, преобразующая контрпример, выданный верификатором *NuSMV*, в контрпример для системы автоматов.

Кроме того, для работы инструментального средства необходимы следующие программы:

- верификатор *NuSMV*;
- *Java Runtime Environment*.

Эти программы могут работать в операционных системах типа *Windows* и *Linux*.

FSM Verifier принимает на вход автоматную систему, записанную в формате *XML*. Структура входного файла была разработана специально для *FSM Verifier* и не поддерживается другими программами. Структуру входного файла приведена в работе [3].

Формула для верификации записывается в тот же *XML*-файл в простом виде, как показано приведенном ниже примере:

```
<specification>
  <string>AG (A0.s1 -&gt; AF A1.s1)</string>
</specification>
```

При верификации должны быть выполнены три команды:

- из автоматной системы и формулы, которые записаны в файл `inputfile.fsm`, создается модель автоматной системы на языке *SMV* в файле `input.smv`:

  ```
  java –jar fsmverifier.jar inputfile.fsm > input.smv
  ```

- вызывается верификатор *NuSMV*:

  ```
  NuSMV input.smv > verifier.out
  ```

- контрпример преобразуется в термины исходной автоматной системы:

  ```
  java –jar counterexample.jar verifier.out inputfile.fsm
  ```

В результате программа выводит контрпример в виде таблицы в формате *HTML*. В каждой строке таблицы указаны:

- номер шага;
- имя активного автомата;
- обрабатываемое событие;
- имена состояний всех автоматов;

- выполняемое действие;
- значение входных воздействий.

Пример использования инструментального средства *FSM Verifier* для верификации автоматной системы, описывающей работу банкомата, описан в работе [4].

2.2. Инструментальное средство *CTLVerifier*

2.2.1. Общее описание

Инструментальное средство *CTLVerifier* [15], разработанное С.Э. Вельдером, не использует других верификаторов. В этом средстве реализован алгоритм верификации формул темпоральной логики *CTL*.

При использовании *CTLVerifier* нет необходимости преобразовывать автоматную систему во входной язык другого верификатора. Тем не менее, требуется преобразовывать автоматную систему в модель *Крипке*, верифицируемую инструментальным средством. Результат верификации также выражается в терминах состояний модели *Крипке*, что приводит к необходимости преобразования контрпримера.

2.2.2. Выделение атомарных состояний

В рассматриваемом подходе переходы в автоматах делятся на цепочки элементарных состояний, аналогично тому, как это сделано в *FSM Verifier*. Отличие состоит в том, что в методе *CTLVerifier* немного видоизменяется первое элементарное состояние, которое содержит событие и условия на переходе. Кроме условий, уже указанных на переходе, туда по два раза записываются все существующие в модели условия: один раз добавляется само условие (например, `x1`), а второй раз – его отрицание (`!x1`). Это требуется верификатору для того, чтобы в этом состоянии выполнились предикаты, описывающие неважные для перехода условия.

На рис. 9 изображен пример такого разбиения.

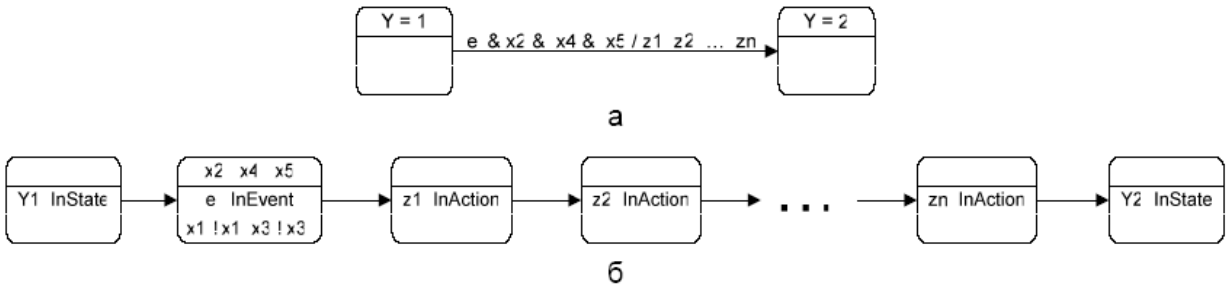

Рис. 9. Выделение промежуточных состояний на переходе. Исходный переход (а), преобразованный переход (б)

Отметим, что в данном инструменте, как и верификаторе *FSM Verifier*, на переходах могут присутствовать только булевы переменные. Кроме того, эти переменные могут соединяться только логическим оператором &.

Подробнее о выделении промежуточных состояний – в работе [4].

2.2.3. Верифицируемые свойства

Как следует из названия, инструментальное средство *CTLVerifier* позволяет верифицировать формулы, записанные в темпоральной логике *CTL*. В качестве предикатов можно использовать следующие условия:

- определенный автомат находится в определенном состоянии;
- обрабатывается определенное событие;
- верна или неверна определенная входная переменная;
- вызывается определенное выходное воздействие.

Метод *CTLVerifier* позволяет верифицировать формулы логики *CTL*, использующие только следующие темпоральные операторы: `EX`, `EG`, `EU`. Другие темпоральные операторы логики *CTL* могут быть выражены через эти по следующим правилам:

- `AX g = !EX !g`
- `EF g = 1 EU g`
- `AF g = !EG !g`
- `AG g = !EF !g = !(1 EU g)`
- `f AU g = !((!g EU (!f || g)) || EG !g)`

2.2.4. Преобразование контрпримера

Также как и в верификаторе *FSM Verifier*, в рассматриваемом средстве любое элементарное состояние – это либо состояние автомата, либо промежуточное состояние на переходе автомата. Поэтому каждое элементарное состояние однозначно определяет состояние или переход в исходной автоматной системе. Следовательно, достаточно преобразовать контрпример в терминах модели *Крипке* в термины исходной автоматной системы.

2.2.5. Описание инструментального средства

CTLVerifier принимает на вход систему автоматов, описанную в текстовом формате. Этот формат достаточно прост и разработан специально для рассматриваемого верификатора.

Во входном файле, кроме описания системы автоматов, приводится также формула для верификации, которая записывается в несколько шагов. При этом все промежуточные формулы тоже должны быть записаны. Например, формула `EG !e1` записывается следующим образом:

```
[Properties]
f1 = e1
f2 = !f1
f3 = $EG f2
```

Верификация производится одной командой:

```
CTLVerif.exe <входной файл> [ <выходной файл>
        [<выходная папка>] ]
```

Программа *CTLVerifier* (в версии на момент написания работы) действует только в операционных системах типа *Windows*.

В результате верификации выводится следующая информация:

- список предикатов, используемых в построенной модели *Крипке*;
- модель *Крипке*, полученная в результате преобразования исходной автоматной системы. Все элементарные состояния пронумерованы, определены переходы между ними и списки предикатов, выполняющихся в каждом состоянии;

- для каждой формулы список номеров состояний модели *Крипке*, в которых она выполняется. Если формула темпоральная и доказывается циклом состояний, то приводится цепочка номеров состояний, которая приводит к выполнению формулы. Например:
  `1 34 35 ( 3 38 39 5 109 110 8 91 92 15 85 86 20 82 83 )`
- если указана выходная папка, то для каждой верифицируемой формулы создается файл с контрпримером в терминах исходной автоматной системы.

Пример использования инструментального средства *CTLVerifier* для верификации автоматной системы, описывающей работу банкомата, приведен в работе [4].

2.3. Инструментальное средство *Converter*

2.3.1. Общее описание

Инструментальное средство *Converter* [16], разработанное М.А. Лукиным, использует, пожалуй, самый известный верификатор – *SPIN* [8]. Этот верификатор предназначен для верификации формул в темпоральной логике *LTL*. Модель для верификации записывается на входном языке *Promela*.

При использовании этого средства решаются следующие задачи: конвертация автоматной системы в язык *Promela*; определение предикатов для формализации свойств; преобразование контрпримера в термины исходной системы. Ниже описано, как эти задачи решаются в инструментальном средстве *Converter*.

2.3.2. Выделение атомарных состояний

Модель на языке *Promela*, полученная в результате конвертации исходной автоматной системы, содержит следующие переменные, которые хранят состояние модели:

- `int lastEvent;` – переменная, содержащая номер последнего обработанного события;
- `int stateAi;` – переменная, создаваемая для каждого автомата. Она хранит номер текущего состояния автомата `Ai`.

Поскольку в построенной модели на языке *Promela* других переменных нет, то поэтому элементарное состояние модели определяется набором значений этих переменных.

Другими словами, в данном подходе элементарное состояние автоматной системы – это набор текущих состояний всех автоматов, а также последнее обработанное событие. Переходы не разделяются на цепочки элементарных событий, как это делалось в инструментальных средствах *FSM Verifier* и *CTLVerifier*.

В модели на языке *Promela* для каждого автомата выделяется функция, которая недетерминированно выбирает переход из текущего состояния автомата, и в зависимости от выбранного перехода обновляет значения переменных `stateAi` и `lastEvent`. При этом в текущей версии метода входные переменные и выходные воздействия не записываются в модель на языке *Promela*, и поэтому никак не учитываются при верификации.

2.3.3. Верифицируемые свойства

Рассматриваемый метод верифицирует формулы в темпоральной логике *LTL*. Поддерживаются только два типа предикатов:

- `lastEvent = e1` – выполняется, когда последнее обработанное событие `e1`;
- `stateAi = 1` – выполняется, когда номер текущего состояния автомата `Ai` в модели на языке *Promela* равен единице. При этом отметим, что это не имя соответствующего состояния в исходной автоматной системе, и поэтому пользователю необходимо найти номер интересующего состояния в преобразованной модели самостоятельно.

2.3.4. Преобразование контрпримера

В случае, если верификатор *SPIN* находит ошибку в модели, имеется возможность «воспроизвести» контрпример на модели. Это означает, что этот верификатор будет выполнять модель на языке *Promela* как настоящую программу на языке, похожем на язык программирования *C*.

Инструментальное средство *Converter* использует эту возможность. При каждом переходе из одного состояния автомата в другое, в модели на языке *Promela* выводится сообщение о переходе и текущих состояниях автоматов. Таким образом, когда верификатор *SPIN* воспроизводит ошибку (контрпример), в стандартный вывод печатаются указанные сообщения, позволяя пользователю увидеть переходы в исходной автоматной системе, как это показано в следующем примере:

```
State Test 1 : init
Going to state Test 2 : s0
Event = e2
State Test 2 : s0
Going to state Test 33 : s1-1
Event = e3
State Test 33 : s1-1
```

2.3.5. Описание инструментального средства

Инструментальное средство *Converter* работает с автоматными системами, созданными при помощи инструментального средства *UniMod* [9].

Инструментальное средство *UniMod* позволяет сохранять автоматную систему в *XML*-формате. Файл в этом формате подается на вход верификатора *Converter*.

Дистрибутив инструментального средства *Converter* поставляется с необходимыми библиотеками и верификатором *SPIN*. Единственное, что требуется – установить компилятор `gcc` и прописать его в переменной `PATH`, для того чтобы можно было вызвать программу `gcc` без указания ее расположения в файловой системе компьютера. Программа в настоящее время может запускаться только в операционных системах типа *Windows*.

Для верификации требуется выполнить одну команду:

```
run.cmd <XML файл системы> <файл с отчетом> <LTL-
        формула>
```

Здесь:

- *XML*-файл системы – верифицируемая автоматная система, сохраняемая с помощью инструментального средства *UniMod* в *XML*-формате;
- файл с отчетом – имя файла, в который будет записан результат верификации;
- *LTL*-формула – верифицируемая формула, например: `"!(<>{lastEvent == e1})"`.

Для записи формул используются следующие операторы.:

- `[]` – `Globally` (всегда);
- `<>` – `Future` (когда-нибудь в будущем);
- `U` – `Until` (до тех пор пока);
- `V` – запись `p V q` эквивалентна `!(!p U !q)`;
- `!` – отрицание;
- `&&` – логическое И;
- `||` – логическое ИЛИ;
- `->` – следует;
- `<->` – эквивалентно.

Пример использования средства *Converter* для верификации автоматной системы, описывающей работу банкомата, описан в работе [4].

2.4. Инструментальное средство *Automata Verificator*

2.4.1. Общее описание

Верификатор автоматных систем *Automata Verificator* [12], разработанный К.В. Егоровым, не использует существующих верификаторов. В нем реализован алгоритм двойного поиска в глубину, позволяющий

проверять формулы, описанные с помощью автомата *Бюхи*, а, следовательно, и формулы темпоральной логики *LTL*.

Особенностью верификатора является то, что алгоритм верификации реализован с возможностью *многопоточного* исполнения. Таким образом, как было показано в работе [12], при выполнении на многоядерных процессорах алгоритм дает лучшие результаты по сравнению с однопоточным вариантом.

2.4.2. Выделение атомарных состояний

В рассматриваемом верификаторе атомарное состояние определяется набором текущих состояний автоматов, входящих в автоматную систему. Таким образом, не производится дробления переходов на элементарные состояния.

2.4.3. Верифицируемые свойства

Список предикатов, поддерживаемых верификатором *Automata Verificator*, схож с предикатами, используемыми для верификатора *UniMod.Verifier*. Поддерживаются следующие предикаты:

- `wasEvent`;
- `isInState`;
- `wasInState`;
- `cameToFinalState`;
- `wasAction`;
- `wasFirstAction`.

Смысл перечисленных предикатов такой же, как в верификаторе *UniMod.Verifier*. Удобно то, что в качестве аргументов предикатов можно использовать имена состояний, событий и действий исходной автоматной системы, даже если эти имена содержат пробелы и другие специальные символы.

Кроме того, в верификаторе *Automata Verificator* можно достаточно просто создавать новые предикаты. Для этого требуется разработать класс на языке *Java*, в котором необходимо создать метод с аннотацией «`@Predicate`». Например, можно создать предикат, позволяющий верифицировать свойства типа «действие `o1.z1` выполняется через одно после `o1.z2`».

В формулах, как отмечено выше, используется темпоральная логика *LTL*.

2.4.4. Преобразование контрпримера

Рассматриваемый верификатор не основан на других верификаторов, и алгоритм верификации работает напрямую с автоматной системой. Поэтому контрпример сразу выражен в терминах исходной автоматной системы, и нет необходимости его преобразовывать.

2.4.5. Описание инструментального средства

Верификатор *Automata Verificator* разработан в виде *Java*-классов и изначально не поставляется в виде готовой библиотеки, которую можно было бы запускать из командной строки. В рамках настоящей работы *автором* создан класс, позволяющий запускать верификацию из командной строки, и была собрана библиотека со всеми классами верификатора *Automata Verificator*.

Рассматриваемый верификатор принимает на вход автоматные системы, описанные в формате инструментального средства *UniMod*. Формат запуска верификации следующий:

```
java -jar verifier.jar A.xml A1 "F(wasEvent(p.e1))"
```

где

- `java` – команда запуска *Java*-машины. Должна быть установлена программа *Java Runtime Environment* версии не ниже шестой;
- `A.xml` – путь к файлу с описание автоматной системы в формате *UniMod;*

- `A1` – название корневого автомата;
- `F(wasEvent(p.e1))` – верифицируемая формула. В этом примере она означает следующее: «когда-нибудь произойдет событие `e1`, генерируемое источником событий `p`».

В результате работы такой команды сначала в консоль выводится автомат *Бюхи*, сгенерированный по *LTL*-формуле, а затем результат верификации. При удачной верификации выводится сообщение «`Verification successful`», а если найден контрпример, то вывод верификатора выглядит следующим образом:

```
LTL: F(isInState(AClient, AClient["10. Выдача денег"]))
initial 0
BuchiNode 0
      -->[!isInState(AClient, 10. Выдача денег)] 0
Accept set 0 [0]

DFS 2 stack:
-->["<"13. Возврат карты", "s1">", 0, 0]-->["<"1. Вставьте карту", "s1">", 0,
0]
DFS 1 stack:
-->["<"s1", "s1">", 0, 0]-->["<"1. Вставьте карту", "s1">", 0, 0]
-->["<"2. Ввод pin кода", "s1">", 0, 0]-->["<"3. Авторизация", "s1">", 0, 0]
-->["<"4. Главное меню", "s1">", 0, 0]-->["<"13. Возврат карты", "s1">", 0,
0]
```

Автомат *Бюхи* выводится как набор состояний. Также указывается его начальное состояние (`initial`) и наборы допускающих состояний (`Accept set`). В данном примере автомат *Бюхи* состоит из одного состояния под номером «`0`». Это же состояние начальное и допускающее.

Далее записывается контрпример в виде набора состояний в стеке главного обхода в глубину (`DFS 1`) и вложенного обхода в глубину (`DFS 2`) [6]. По сути контрпример получается путем продолжения последовательности `DFS 1` последовательностью `DFS 2`. Однако необходимо предварительно опустить первое состояние последовательности `DFS 2`, поскольку оно повторяет последнее состояние последовательности `DFS 1`.

Каждое состояние автоматной системы в контрпримере записывается внутри квадратных скобок. В них содержится три компоненты:

1. Набор текущих состояний автоматов автоматной системы.
2. Номер текущего состояния автомата *Бюхи*.
3. Множество допускающих состояний автомата *Бюхи*.

2.4.6. Пример использования

Приведем пример использования верификатора *Automata Verificator* на задаче верификации автоматной системы банкомата, описанной в разд. 1.3.1.

Первое верифицируемое свойство: «Пользователь не может получить деньги, если он не ввел правильный *Pin*-код». Оно должно выполняться в рассматриваемом банкомате, поэтому верификация этого свойства в корректной автоматной системе должна закончиться успешно. Переформулируем свойство: «Не может быть, чтобы пользователь не вводил правильный *Pin*-код, до того как получил деньги». Словесная формулировка свойства напрямую переводится в темпоральную логику *LTL*:

```
! (![введет правильный PIN-код] U [выдадут деньги])
```

Здесь `U` – темпоральный оператор `Until` – «пока не, до тех пор пока». Как показано на рис. 3, выдача денег происходит действием `o1.z10`. Поскольку ни один из рассматриваемых верификаторов не позволяет проверять выполнение выходных воздействий, а действие `o1.z10` вызывается только в состоянии *10. Выдача денег* автомата *AClient*, то будем использовать это состояние как показатель выдачи денег. Правильно введенный *Pin*-код характеризуется событием `e10`. Таким образом, формула для верификации принимает следующий вид:

```
! (!e10 U (AClient in "10. Выдача денег"))
```

Второе верифицируемое свойство записывается следующим образом: «Пользователь обязательно получит деньги». Это свойство должно не выполняться для банкомата с корректной автоматной системой, и поэтому верификатор должен отобразить контрпример этого свойства. Словесная формулировка этого свойства напрямую переводится в темпоральную логику *LTL*:

```
F [выдадут деньги]
```

Здесь `F` – темпоральный оператор `Future` («когда-нибудь»). Как отмечено выше, выдача денег происходит только в одном состоянии. Поэтому формула принимает следующий вид:

```
F (AClient in “10. Выдача денег”)
```

Верификатор *Automata Verificator* работает с автоматными системами в формате *UniMod*. Поэтому дополнительных преобразований исходной автоматной системы банкомата не требуется.

Верифицируем первое свойство. Утверждение, что произошло событие `e10`, записывается следующим образом: `wasEvent(p3.e10)`, где `p3` – источник событий, генерирующий событие `e10` (рис. 3). Утверждение, что автомат *AClient* находится в состоянии «*10. Выдача денег»*, записывается следующим образом: «`isInState(AClient, AClient[\"10. Выдача денег\"])`».

Приводимая ниже команда выполняет верификацию первого свойства банкомата:

```
java -jar verifier.jar Bankomat.xml AClient
        "!U(!wasEvent(p3.e10), isInState(AClient,
        AClient[\"10. Выдача денег\"]))"
```

Результат выполнения команды:

```
LTL: !U(!wasEvent(p3.e10), isInState(AClient, AClient["10. Выдача денег"]))
initial 1
BuchiNode 0
      -->[true] 0
BuchiNode 1
      -->[!wasEvent(e10)] 1
      -->[isInState(AClient, 10. Выдача денег)] 0
Accept set 0 [0]

Verification successful
```

Как видно, верификация завершена успешно, что правильно.

Верифицируем второе свойство:

```
java -jar verifier.jar Bankomat.xml AClient
        "F(isInState(AClient, AClient[\"10. Выдача
        денег\"]))"
```

Результат выполнения:

```
LTL: F(isInState(AClient, AClient["10. Выдача денег"]))
initial 0
BuchiNode 0
      -->[!isInState(AClient, 10. Выдача денег)] 0
Accept set 0 [0]

DFS 2 stack:
-->["<"13. Возврат карты", "s1">", 0, 0]-->["<"1. Вставьте карту", "s1">", 0,
0]
DFS 1 stack:
-->["<"s1", "s1">", 0, 0]-->["<"1. Вставьте карту", "s1">", 0, 0]
-->["<"2. Ввод pin кода", "s1">", 0, 0]-->["<"3. Авторизация", "s1">", 0, 0]
```

```
-->["<"4. Главное меню", "s1">", 0, 0]-->["<"13. Возврат карты", "s1">", 0,
0]
```

В ходе верификации найден контрпример. Сначала следует читать стек *DFS 1*, а затем – стек *DFS 2*. Во всех приведенных глобальных состояниях автоматной системы лишь состояние автомата *AClient* изменяется.

В соответствии с контрпримером автомат *AClient* начинает работу в начальном состоянии `s1`, затем переходит в состояние «*1. Вставьте карту»*, затем в состояние «*2. Ввод Pin-кода»*, а после этого в состояние «*3. Авторизация»*. Авторизация проходит успешно, и автомат переходит в состояние «*4. Главное меню»*. Видимо, в этот момент пользователь нажимает на кнопку *Отмена*, поскольку автомат переходит в состояние «*13. Возврат карты»*. После этого следует стек *DFS 2*, который завершает контрпример с переходом в уже посещенное состояние «*1. Вставьте карту»*.

Найденный верификатором *Automata Verificator* контрпример является корректным.

Выводы по главе 2

В этой главе было описано несколько существующих инструментальных средств для верификации управляющих систем, реализованных на основе автоматного подхода. Эти верификаторы можно сравнить по нескольким показателям.

Рассмотрим автоматные системы, с которыми может работать каждый метод.

По числу автоматов, каждое из описанных средств работает с автоматными системами, содержащими произвольное числом автоматов.

По типу автоматов, все средства работают как с автоматами Мура или Мили, так и со смешанными автоматами. Инструментальное средство *Converter* не учитывает выходные воздействия в автоматах. Для этого средства нет разницы в том, какой тип автоматов подается на вход верификатора.

Верификатор *UniMod.Verifier* напрямую использует *UniMod* для моделирования и интерпретации автоматных систем. Поэтому он позволяет верифицировать смешанные автоматы.

В инструментальном средстве *Converter* требуется, чтобы у каждого автомата было лишь одно конечное состояние, однако это ограничение не уменьшает возможностей верификатора, так как автомат с несколькими конечными состояниями можно преобразовать в требуемый вид, соединив все конечные состояния в одно.

На условия на переходах автоматов в рассматриваемых методах накладываются разные ограничения. В инструментальном средстве *Converter* условия на переходах никак не учитываются. Поэтому можно писать любые условия, но использовать их в верифицируемых формулах нельзя.

В верификаторе *CTLVerifier* требуется, чтобы условия на переходах имели вид `x1&!x2&x3` – соединялись только оператором конъюнкции, а каждая входная переменная возвращала булево значение.

В верификаторе *FSM Verifier* на переходах можно записывать условия, содержащие любые булевы формулы с использованием входных переменных и состояний других автоматов. Естественно, что входные переменные должны возвращать булево значение. Стоит отметить, что только в автоматах, верифицируемых *FSM Verifier*, можно на переходах записывать условия, зависящие от состояний других автоматов.

В верификаторах *UniMod.Verifier* и *Automata Verificator* на переходах верифицируемых автоматов можно записывать любые булевы выражения, содержащие значения булевых входных переменных, а также сравнения числовых входных переменных с константами или другими числовыми входными переменными, например, `x1 || (1 < x2 && x2 < 3)`. Однако в текущей версии инструментального средства *UniMod* нет возможности использовать в условиях текущие состояния других автоматов.

Сравним теперь описанные верификаторы *по выразительности свойств*, которые они позволяют верифицировать.

В текущей версии верификатор *Converter* позволяет записывать требования только на события на переходах и на текущие состояния автоматов. Таким образом, он не позволяет верифицировать какие-либо требования к вызову автоматной системой выходных воздействий на объектах управления.

Верификатор *Automata Verificator* позволяет записывать требования на события, состояния автоматов и вызываемые выходные воздействия. В настоящее время в верификаторе *Automata Verificator* не поддерживается проверка значений входных переменных.

Верификатор *UniMod.Verifier* позволяет формулировать практически любое условие для автоматной системы. Можно проверять обработанное событие, вызванные выходные воздействия, состояния автоматов и вычисленные условия на переходах. Однако при этом условия на переходах не разбираются и рассматриваются как одно целое. Например, если был сделан переход по условию «`!o1.x1 && o1.x2`», то предикат верифицируемой формулы `wasTrue("!o1.x1 && o1.x2")` будет выполняться, однако предикат `wasTrue("o1.x2")` не будет выполняться, хотя входная переменная `o1.x2` тоже имела значение `True`.

Кроме того, верификатор *UniMod.Verifier* имеет следующее ограничение на формализацию верифицируемых свойств. В нем переход автомата из одного состояния в другое не делится на элементарные состояния, и поэтому для верификатора переход происходит атомарно. Другими словами, для верификатора все, что происходит в пределах обработки одного события – происходит одновременно. Это накладывает ограничение, состоящее в том, что если для выполнения требований важен порядок возникновения некоторых условий (выполнения предикатов), то они не должны выполняться в пределах обработки одного события.

Например, пусть требование состоит в том, чтобы всегда после действия `z1` выполнялось действие `z2`. Тогда нельзя это требование представить формулой «**G**(`z1` → **X** `z2`)», поскольку оператор **X** означает

«при обработке следующего события», а не «следующим действием». Для обеспечения возможности формулировки таких требований для действий объектов управления введен предикат `getActionIndex`, с помощью которого описанное требование будет сформулировано следующим образом:

```
G(z1 → (getActionIndex(z1) + 1 = getActionIndex(z2))
```

Такая же проблема существует и в верификаторе *Automata Verificator*, поскольку в нем также переходы не делятся на элементарные состояния. В работе [12] предлагается создавать специальные предикаты в виде функций на языке *Java*, в случае если стандартные предикаты оказываются недостаточно выразительными.

Верификаторы *CTLVerifier* и *FSM Verifier* лишены описанного недостатка, поскольку дробят переходы на элементарные состояния. Таким образом, эти верификаторы учитывают порядок выполнения выходных воздействий, даже если они происходят в пределах одного перехода. Эти методы позволяют формулировать предикаты, задающие обрабатываемое событие, текущие состояния автоматов, вызываемые выходные воздействия и значения входных переменных.

Сравним теперь реализованные инструментальные средства с точки зрения *удобства и возможности использования*. Верификаторы *Converter*, *UniMod.Verifier* и *Automata Verificator* получают на вход автоматную систему в формате, используемом инструментальным средством *UniMod*. Это позволяет пользователю верифицировать автоматные системы, разработанные при помощи инструментального средства *UniMod*. Таким образом, при их использовании имеет место единый программный пакет для разработки, запуска и верификации автоматных систем.

Верификаторы *CTLVerifier* и *FSM Verifier* имеют оригинальные форматы входного файла. Поэтому пользователь должен конвертировать автоматную систему в необходимый формат.

В таблице приведены характеристики рассмотренных инструментальных средств для верификации автоматных систем.

Верификатор	Язык спецификации	Используемый верификатор	Тип автоматов	Условия на переходах	Дробление переходов	Предикаты	Поддержка моделей *UniMod*
FSM Verifier	*CTL*	*NuSMV*	Смешанные	x1, A1.s1, &&, ||, !	Да	A1 = s1 e1 z1 x1	Нет
CTLVerifier	*CTL*	–	Смешанные	x1, &&, !	Да	A1 = s1 e1 z1 x1	Нет
Converter	*LTL*	*SPIN*	– Выходные воздействия не учитываются	– Условия не учитываются	Нет	A1 = s1 e1	Да
UniMod.Verifier	*LTL*	*Bogor*	Смешанные	x1, x2 > 0, &&, ||, !	Нет	A1 = s1 e1 z1 x1 Но x1&x2 не разбирается	Да
Automata Verificator	*LTL*	–	Смешанные	– Условия не учитываются	Нет	A1 = s1 e1 z1	Да

ГЛАВА 3. СРАВНЕНИЕ ИНСТРУМЕНТАЛЬНЫХ СРЕДСТВ ВЕРИФИКАЦИИ АВТОМАТНЫХ СИСТЕМ

В этой главе проводится сравнение рассмотренных верификаторов автоматных систем на примерах. В работе [4] было проведено сравнение работы верификаторов на примере автоматной системы банкомата. В этой главе верификаторы будут тестироваться на автоматных системах большого размера. Цель тестирования – определить, какие верификаторы смогут верифицировать сложные автоматные системы, и могут быть применены в реальных задачах верификации, а какие верификаторы полезны скорее в учебных целях.

3.1. Верификация автомата с большим числом состояний

В данном разделе сравниваемые верификаторы тестируются на задаче верификации свойств автоматной системы, состоящей из одного автомата с большим числом состояний. Цель данного эксперимента – определить верхнюю границу числа состояний в одном автомате автоматной системы, которая может быть верифицирована тем или иным верификатором.

3.1.1. Описание эксперимента

Необходимо выбрать такую структуру автоматной системы и такое ее свойство, для того чтобы можно было свободно изменять число состояний системы, заранее зная, выполняется ли сформулированное свойство или нет. В качестве такой структуры было выбрано двоичное дерево, один из листьев которого замыкается на начальное состояние, а остальные листы ведут в конечное состояние. На рис. 10 изображена структура такого автомата.

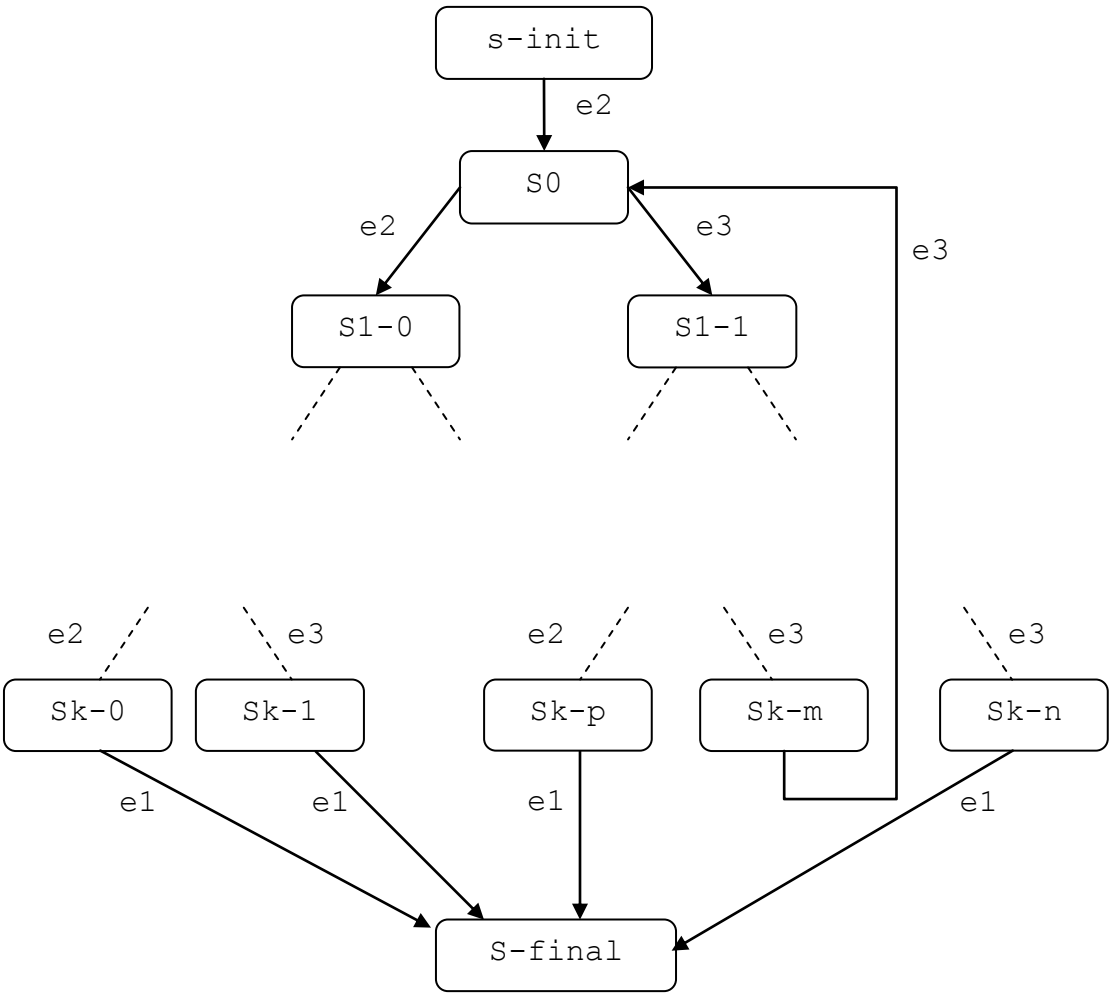

Рис. 10. Автомат с циклом

В качестве верифицируемого свойства использовано выражение:

```
F e1
```

Это свойство выполняется, если на любом пути из начального состояния существует переход по событию `e1`. Этим событием помечены только переходы в конечное состояние. Поэтому такое утверждение равносильно следующему: «когда-нибудь автомат попадет в конечное состояние».

Причина, по которой не используется более логичная формула `F S-final`, состоит в том, что необходимо создать формулу, которая будет единой для автоматов всех размеров и для всех рассмотренных верификаторов.

В верификаторе *Converter* в формуле необходимо указывать номер состояния в преобразованной модели. Этот номер может оказаться разным для тестируемых автоматных систем разного размера.

Контрпримером верифицируемого свойства будет путь вида:

```
S-init -> S0 -> S1-1 -> … -> Sk-m -> S0
```

Кроме того, каждый верификатор также запускается на примере автоматной системы, в которой верифицируемое свойство выполняется. Это связано с тем, что некоторые верификаторы тратят больше времени на доказательство выполнения формулы, чем на нахождение контрпримера. Общий вид такого автомата изображен на рис. 11. В нем переходы из всех листьев ведут в конечное состояние.

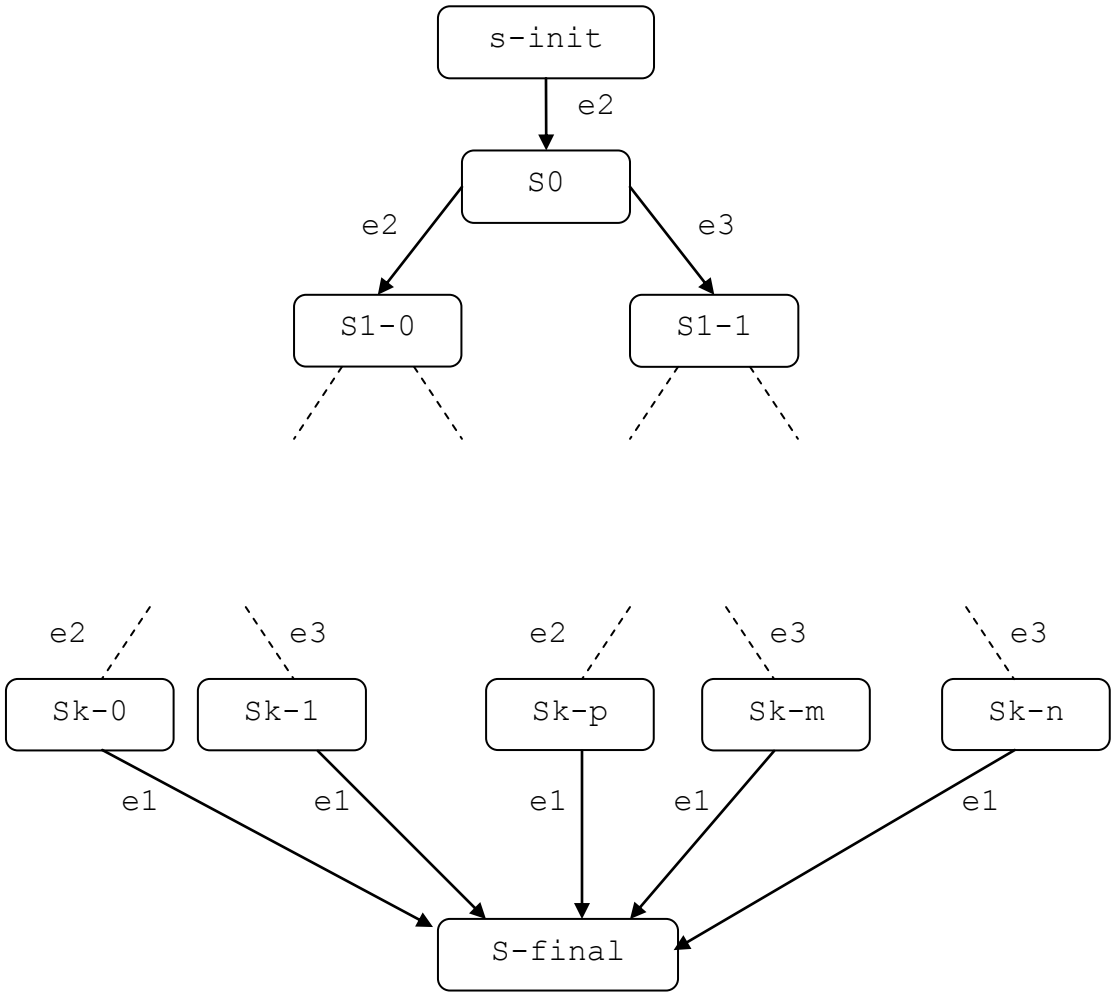

Рис. 11. Автомат без цикла

Для генерации описанных автоматных систем была создана специальная программа. В качестве аргументов она получает глубину дерева автомата, число переходов из каждого узла, а также булева переменная, показывающая будет или не будет автомат удовлетворять условию. Программа генерирует модель автоматной системы в формате *UniMod*. Затем используются программы для преобразования этого формата в форматы верификаторов *CTLVerifer* и *FSM Verifier*.

Кроме того, была написана программа, запускающая каждый верификатор и отсчитывающая число миллисекунд, затраченное на его работу. Существует известная погрешность в вычислениях, состоящая в том, что отсчитывается абсолютное время между стартом верификатора и его завершением, а не процессорное время, потраченное конкретно на работу верификатора. Однако в данном эксперименте такая погрешность допустима, поскольку исследователя скорее интересует максимальный размер автомата, который может быть верифицирован за приемлемое время. Приемлемое время исчисляется в минутах, что значительно превышает погрешность, вызванную в результате переключения процессора на другие задачи.

Эксперимент проводится на вычислительной машине со следующими параметрами:

- процессор *Intel Core2* 6300, 1.86 ГГц;
- объем оперативной памяти 2.00 Гб;
- объем свободной памяти на жестком диске больше 10 Гб;
- операционная система *Windows XP*.

Для верификаторов, использующих *Java Runtime Environment*, применяется версия 1.5.0_05, кроме верификатора *Automata Verificator*, для которого используется версия 1.6.0_13. *Java*-машина запускалась со следующими параметрами: `-Xms128m -Xmx512m -Xss96k -XX:MaxPermSize=128m`.

3.1.2. Результаты тестирования верификатора *FSM Verifier*

Верификатор *FSM Verifier* успешно верифицировал автоматные системы размером до 2048 состояний. Верификация автомата в 2048 состояний заняла около 36 с. Наблюдалась незначительная разница между временем верификации корректного автомата и некорректного автомата, но корректный автомат верифицируется быстрее. На рис. 12 изображен график с результатами эксперимента.

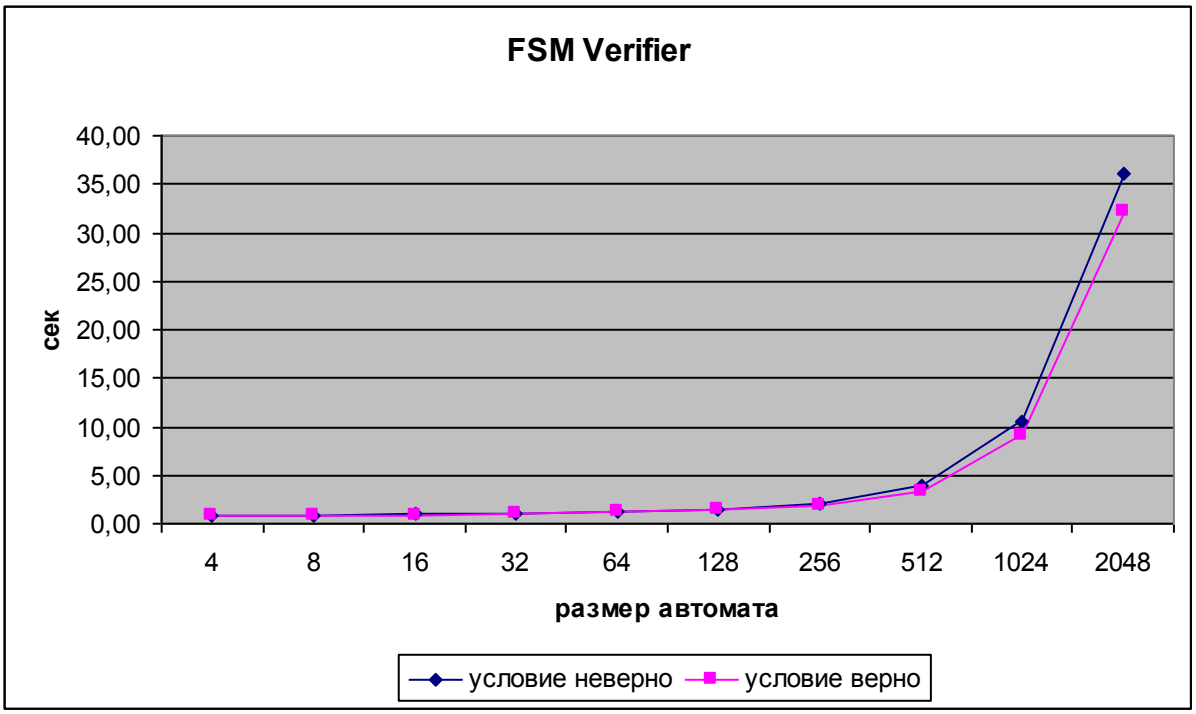

Рис. 12. Результаты тестирования верификатора *FSM Verifier*

Автоматы большего размера этим средством верифицировать не удается, так как верификатор *NuSMV* не справляется с задачей. При его запуске в консоль как обычно выводится общая информация о верификаторе *NuSMV*, и, спустя восемь секунд, команда завершается без какого-либо результата.

Возможно, это связано с использованием устаревшей версии указанного верификатора. Однако если следовать инструкции по использованию верификатора *FSM Verifier*, то верификация автомата размером 4096 состояний не выполняется.

3.1.3. Результаты тестирования верификатора *CTLVerifier*

Верификатор *CTLVerifier* успешно верифицировал автоматные системы размером до 2048 состояний. Верификация автомата размером 2048 состояний заняла 4.86 с. При этом время верификации корректного автомата и некорректного автомата практически одинаково. На рис. 13 изображен график с результатами эксперимента.

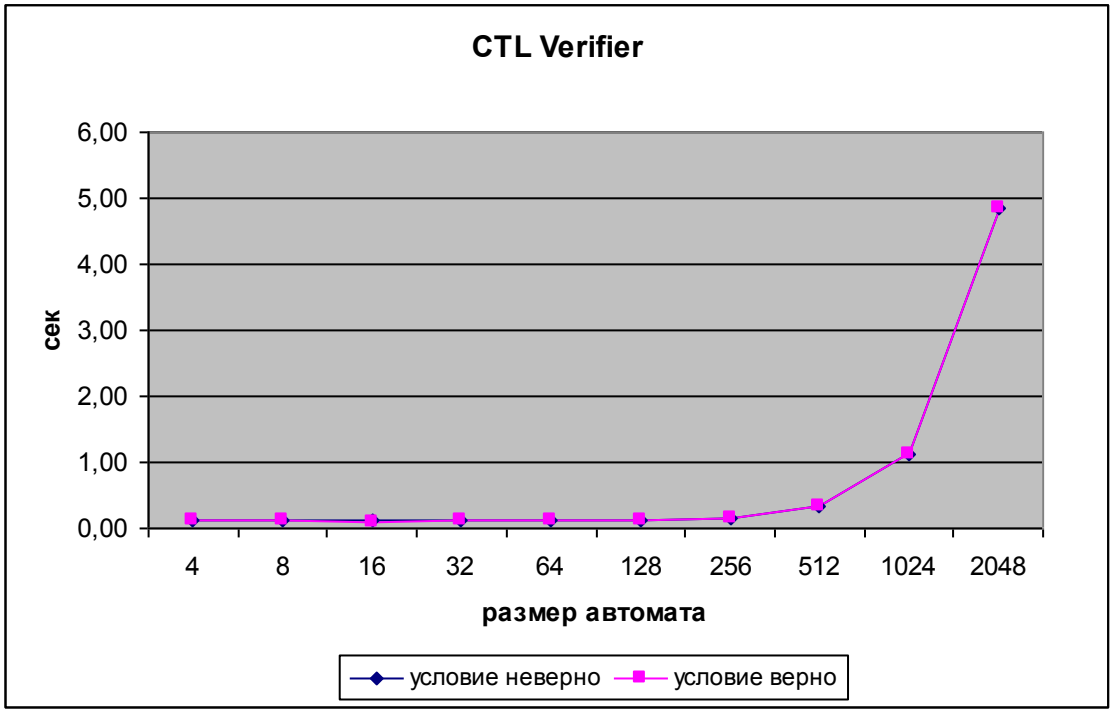

Рис. 13. Результаты тестирования верификатора *CTLVerifier*

При попытке верификации автомата размером 4096 состояний, через несколько секунд после запуска верификатора выдается сообщение, изображенное на рис. 14.

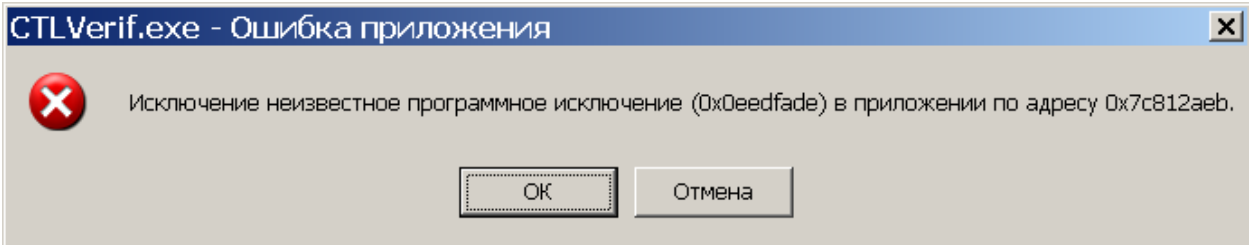

Рис. 14. Ошибка при верификации автоматной системы размером 4096 состояний

Видимо, это связано с тем, что у верификатора заканчивается выделенная ему память. Возможно что, если увеличить размер указанной памяти, то удастся верифицировать автоматы большего размера.

3.1.4. Результаты тестирования верификатора *Converter*

Верификатор *Converter* успешно верифицировал автоматы размером до 4096 состояний. Разница во времени верификации корректного и некорректного автомата незначительна. Результаты эксперимента изображены на рис. 15.

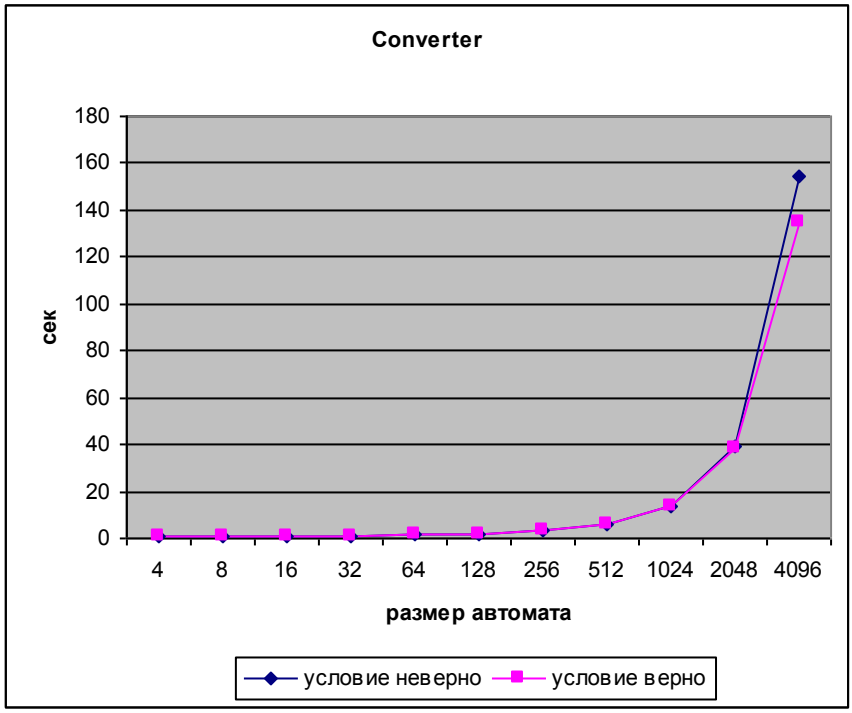

Рис. 15. Результаты тестирования верификатора *Converter*

При попытке верификации автомата размером 8192 состояния, верификатор *SPIN* не справляется с построенной моделью.

3.1.5. Результаты тестирования верификатора *UniMod.Verifier*

С помощью верификатора *UniMod.Verifier* удалось верифицировать автоматы размером до 1024 состояний. Наблюдается значительная разница во времени верификации: корректные автоматы верифицируются дольше.

Верификация автомата размером 1024 состояний заняла около двух минут в случае невыполнения формулы, и около 13 мин. в случае выполнения формулы. На рис. 16 изображены результаты эксперимента.

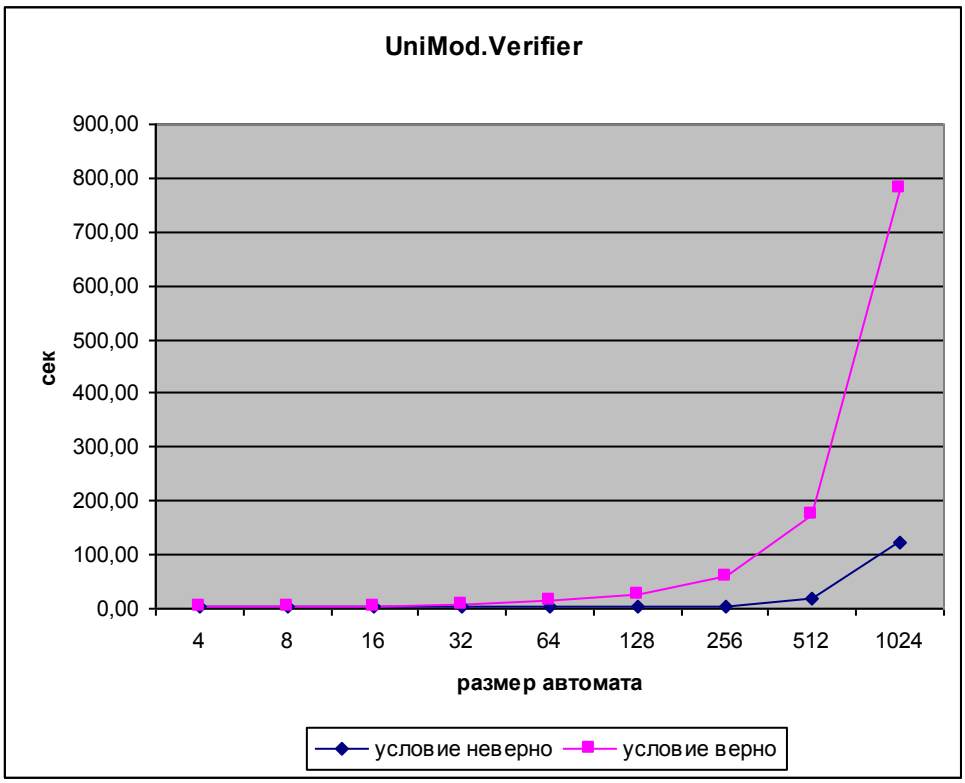

Рис. 16. Результаты тестирования верификатора *UniMod.Verifier*

Верификация автоматов размером 2048 состояний не была завершена в течение 20 минут. Это означает, что верификатор практически не пригоден для верификации больших автоматных систем.

3.1.6. Результаты тестирования верификатора *Automata Verificator*

С помощью верификатора *Automata Verificator* удалось верифицировать автоматы размером до 260 тысяч состояний. При этом наблюдается значительная разница между временем верификации в случае выполнения формулы и невыполнения формулы. Корректный автомат размером 262 144 состояний был верифицирован примерно за 2.4 мин. Некорректный автомат такого же размера был верифицирован за 28.5 с. На рис. 17 изображены результаты эксперимента.

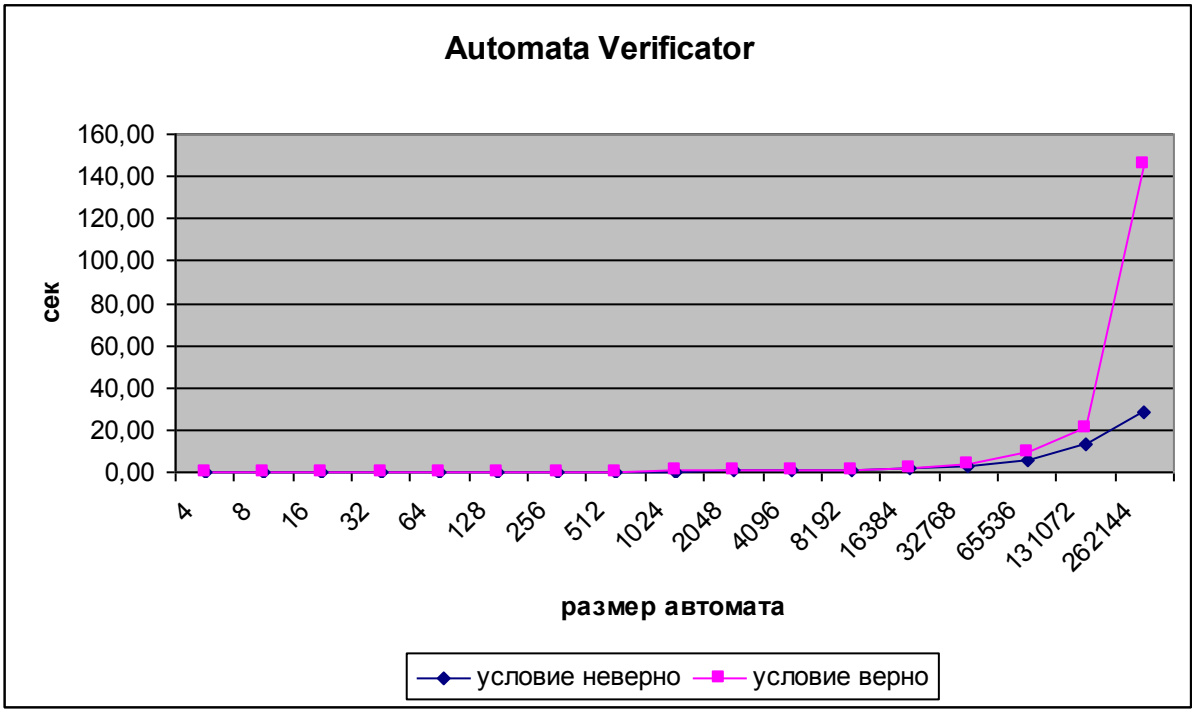

Рис. 17. Результаты тестирования верификатора *Automata Verificator*

Верификатор *Automata Verificator* успешно справился с верификацией всех автоматов, на которых проводился эксперимент. Программно не удалось сгенерировать автомат размером 524 тысячи состояний, однако на графике результатов видна тенденция резкого роста времени верификации корректного автомата после размера в 131 тысячу состояний. Поэтому предполагается, что верификатор не справится с корректным автоматом размером в миллион состояний за время меньшее 20 мин.

На рис. 18 изображены общие результаты проведенных экспериментов.

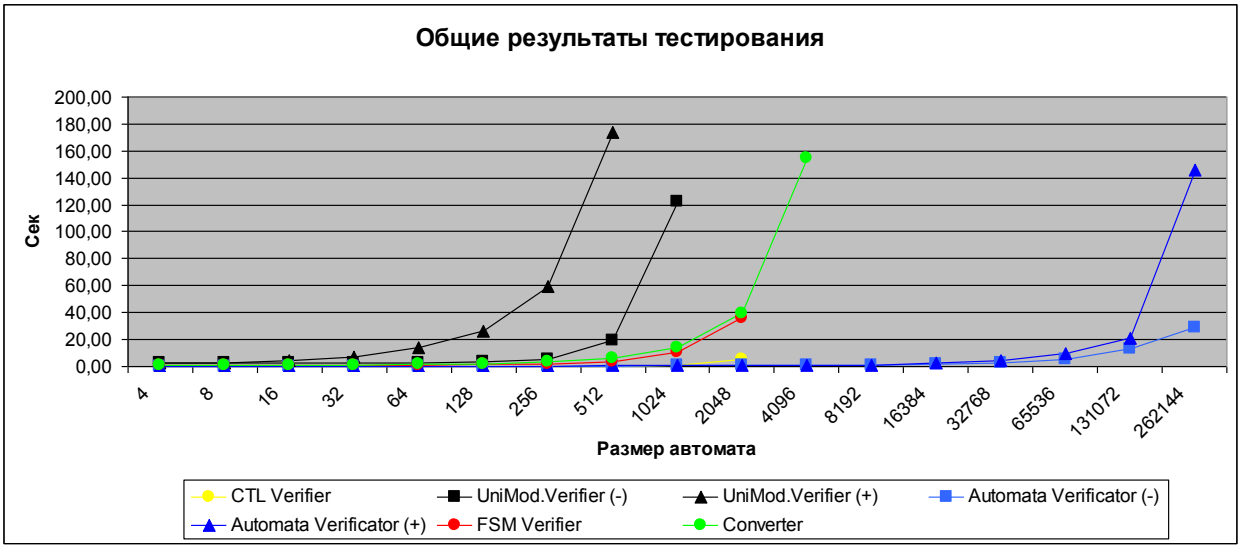

Рис. 18. Общие результаты тестирования. (-) означает верификацию некорректного автомата, (+) – корректного.

На графике видно, что верификатор *Automata Verificator* обладает явным преимуществом: максимальный размер верифицированного им автомата превышает в тысячи раз максимальные размеры автоматов других верификаторов. И, возможно, это еще не предел. Одно из объяснений этого факта – то, что верификатор *Automata Verificator* не разделяет переходы автомата на элементарные состояния, как это делает *FSM Verifier* и *CTLVerifier*. Поэтому он требует меньших ресурсов памяти и времени для работы. Однако, и выразительная способность верифицируемых формул при этом меньше.

О других верификаторах можно сказать следующее. Верификатор *CTLVerifier* обладает очень хорошей тенденцией малого увеличения времени с ростом числа состояний. Возможно, если решить проблему заканчивающейся памяти, то удастся верифицировать автоматы больших размеров.

Рост времени работы верификаторов *Converter* и *FSM Verifier* значительно выше, чем у верификатора *CTLVerifier*. При этом с помощью верификатора *Converter* удалось верифицировать модели в два раза большие, чем с помощью верификатора *FSM Verifier*.

У верификатора *UniMod.Verifier* хуже тенденция роста времени работы с ростом размера автомата. Поэтому он, видимо, не может использоваться для верификации автоматных систем большого размера. Это объясняется тем, что при верификации для совершения переходов напрямую используется интерпретатор инструментального средства *UniMod*. Это значительно сложнее и дольше, чем просто присвоить переменной состояния новое значение, как это делается в других верификаторах.

3.2. Верификация различных структур автоматов

3.2.1. Автоматная система с вложенными автоматами

Проверим, насколько хорошо сравниваемые верификаторы работают с системой автоматов. Создадим с помощью инструментального средства *UniMod* автоматную систему, состоящую из двух автоматов, На рис. 19 изображена схема связей построенной автоматной системы.

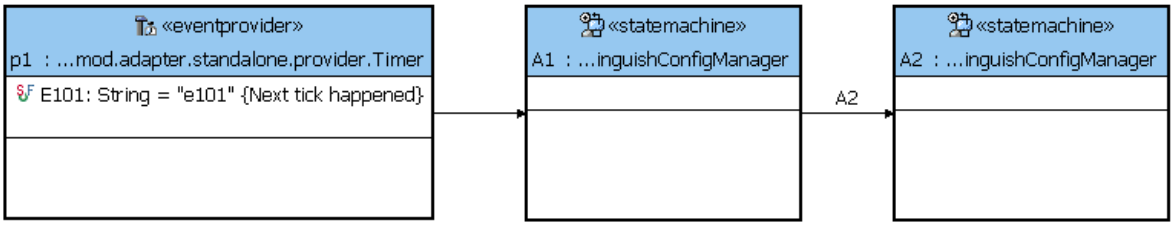

Рис. 19. Схема связей автомата

На рис. 20 изображена диаграмма состояний головного автомата *A1*.

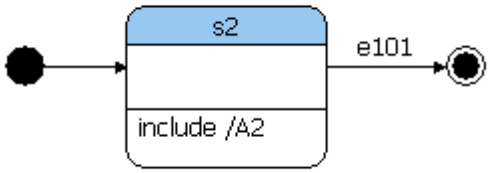

Рис. 20. Диаграмма состояний автомата *A1*

На рис. 21 изображена диаграмма состояний вложенного автомата *A2*.

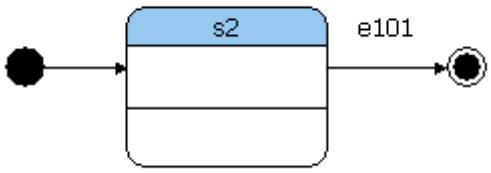

Рис. 21. Диаграмма состояний автомата *A2*

Событие `e101` происходит по таймеру – через секунду после запуска этой автоматной системы в инструментальном средстве *UniMod*. Если запустить эту программу, то в консоль выведется следующее:

```
[Run] Start event [e101] processing. In state [/A1:Top]
[Run] Transition to go found [s1#s2##true]
[Run] Try transition [s2#s3#e101#true]
[Run] Transition to go found [s2#s3#e101#true]
[Run] State machine came to final state [/A1:s3]
[Run] Finish event [e101] processing. In state [/A1:s3]
```

Таким образом, происходит следующее. При возникновении события `e101` сначала инициализируется головной автомат *A1*, переходя из начального состояния в состояние `s2`. Затем в этом же автомате находится переход по событию `e101`, который и осуществляется. В результате корневой автомат оказывается в конечном состоянии, и программа завершает работу. Вложенный автомат *A2* даже не инициализируется.

Проверим следующее утверждение с помощью рассматриваемых верификаторов: «Автомат *A2* никогда не попадет в состояние `s3`». Это утверждение формализуется следующим образом: (`G ! (A2 in state s3)`).

Проверка на верификаторе *UniMod.Verifier* дает успешный результат. Как и следовало ожидать, это утверждение выполняется.

Проверка на верификаторе *Converter* дает следующий результат:

```
            State A1 1 : s1
            Going to state A1 2 : s2
            State A1 2 : s2
            Calling automaton A2
                State A2 5 : s1
                Going to state A2 6 : s2
                State A2 6 : s2
Never claim moves to line 89 [((stateA2==7))]
                Going to state A2 7 : s3
Never claim moves to line 93 [(1)]
spin: trail ends after 41 steps
```

Таким образом, найден контрпример: автомат *A1* переходит из начального состояния в состояние `s2`; затем автомат *A2* получает управление и переходит из начального состояния в состояние `s2`, после этого по событию

e101 автомат *A2* переходит в свое конечное состояние. Строка «Event = e101» не выведена в контрпример поскольку она, видимо, выводится уже после того, как верифицируемое свойство нарушилось, и верификатор *SPIN* ее опустил. Это неправильный результат: верификатор нашел контрпример, хотя в реальной автоматной системе свойство выполняется.

Верифицируем то же свойство при помощи верификатора *Automata Verificator*. В результате получим контрпример:

```
DFS 2 stack:
-->["<"s3", "s3">", 0, 0]-->["<"s3", "s3">", 0, 0]
DFS 1 stack:
-->["<"s1", "s1">", 1, 0]-->["<"s2", "s1">", 1, 0]-->["<"s2", "s2">", 1, 0]
-->["<"s2", "s3">", 0, 0]-->["<"s3", "s3">", 0, 0]
```

Это такой же контрпример, как и найденный верификатором *Converter*.

В результате конвертации автоматной системы в формат верификатора *CTLVerifier* и верификации того же свойства, верификатор *CTLVerifier* выводит такой же контрпример. При этом состояния автомата *A1* были переименованы в s11, s12, s13, а состояния автомата *A2* в s21, s22, s23:

```
$  1: A1 InState s11
   9: * A1 InEvent
   2: A2 InState s21
   5: * A2 InEvent
   3: A2 InState s22
   6: A2 e101 InEvent
   4: A2 InState s23
```

Формат автоматных систем верификатора *FSM Verifier* не поддерживает автоматов, вложенных в состояния. Поэтому рассматриваемую автоматную систему не удалось конвертировать в формат верификатора *FSM Verifier*.

Таким образом, оказалось, что ни один верификатор, кроме *UniMod.Verifier*, не смог правильно верифицировать указанное свойство автоматной системы. Это связано с тем, что интерпретация автоматных систем в каждом из этих верификаторов производится не так, как в инструментальном средстве *UniMod*. Каждый верификатор в каждом состоянии находит все возможные переходы, и выбирает любой из них. При этом, видимо, не учитывается, что обработка события каждый раз начинается с головного автомата.

Этот пример является подтверждением того, что интерпретация автоматных систем верификаторами может отличаться от интерпретации ее инструментальным средством, выполняющим автоматную систему, что приводит к ошибкам верификации.

Верификатор *UniMod.Verifier* является в этом смысле самым надежным по отношению к инструментальному средству *UniMod*, поскольку интерпретация в случае верификации производится теми же модулями, что и при запуске.

3.2.2. Автоматная система с вложенными состояниями

Верифицируем свойство автоматной системы, автоматы которой содержат вложенные состояния. В качестве примера используется автоматная система, моделирующая работу автомобильной сигнализации [13]. На рис. 22 изображена диаграмма головного автомата. Схема связей и диаграмма вложенного автомата приведены в работе [13].

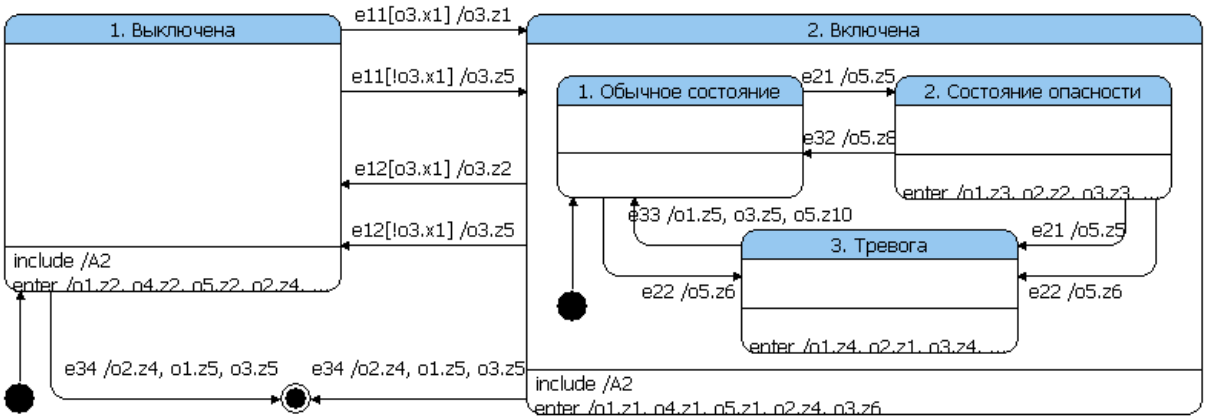

Рис. 22. Диаграмма состояний корневого автомата системы автомобильной сигнализации

Как видно из этой диаграммы, автомат содержит два основных состояния: состояние, в котором сигнализация включена, и состояние, в котором она выключена. При этом во включенном состоянии имеются три режима, моделируемых вложенными состояниями. При включении сигнализации (событие `e11`) автомат оказывается в первом вложенном

состоянии: «1. Обычное состояние». Если машину слегка качнули (событие `e21`), осуществляется переход во второе вложенное состояние «2. Состояние опасности», и сигнализация делает предупреждающий вызов сирены. Если же машину ударили (событие `e22`), то осуществляется переход в третье вложенное состояние режим «3. Тревога» и включается сирена. Сигнализация выключается по событию `e12`.

Проверяемое свойство сформулировано следующим образом: «если сигнализацию включили, то при сильных ударах она будет включать сирену, по крайней мере, до тех пор, пока ее не выключат». В логике *LTL* это утверждение выражается формулой:

```
G ( e11 -> ( (G (e22 -> A1 in "3. Тревога")) W e12) )
```

Верификатор *UniMod.Verifier* успешно верифицирует эту сложную формулу: свойство выполняется в спроектированной автоматной системе.

Другие верификаторы не поддерживают автоматы с вложенными состояниями. Поэтому не удалось верифицировать приведенное свойство с помощью других верификаторов.

В принципе, можно улучшить и эти верификаторы, для того чтобы они также работали с вложенными состояниями. Однако оригинальность верификатора *UniMod.Verifier* состоит в том, что возможность работы с вложенными состояниями не реализовывалась в нем специально: она была обеспечена автоматически за счет использования интерпретатора инструментального средства *UniMod*. Таким образом, верификатор *UniMod.Verifier* автоматически может работать с любыми автоматными системами, с которыми работает инструментальное средство *UniMod*. Если разработчики инструментального средства *UniMod* расширят функциональность и, например, добавят поддержку условий на переходах, зависящих от текущих состояний других автоматов, то автоматически и верификатор *UniMod.Verifer* сможет верифицировать такие системы. При этом другие верификаторы придется исправлять и дополнять при введении новой функциональности.

ЗАКЛЮЧЕНИЕ

Подведем итоги сравнения верификаторов автоматных систем.

Верификаторы отличаются типом темпоральной логики, которая используется при записи проверяемых требований. Поэтому нельзя выделить лишь один верификатор как самый лучший. Для каждого верификатора существует свое применение.

С точки зрения верификации задач большой размерности лучшим верификатором в настоящее время является верификатор *Automata Verificator*. Он позволяет верифицировать большие автоматные системы. Поэтому, скорее всего, он может быть применен для реальных сложных задач.

С точки зрения функциональности и корректности лучшим верификатором является *UniMod.Verifier*. Во-первых, он верифицирует правильно те автоматные системы, на которых другие верификаторы ошибаются. Во-вторых, он верифицирует не абстрактные, а конкретные автоматные системы, выполняющиеся инструментальным средством *UniMod*. В результате получается единый набор средств для разработки, запуска и верификации автоматных систем. Пользователь может быть уверен, что если утверждение успешно верифицировалось верификатором *UniMod.Verifier*, то оно будет также корректно работать в инструментальном средстве *UniMod*. Этого нельзя сказать о других верификаторах.

Подводя итог, можно утверждать, что в результате магистерской диссертации разработан и проверен верификатор автоматных систем *UniMod.Verifier*. Эксперименты показали, что во многих случаях этот верификатор является единственно применимым среди существующих верификаторов автоматных систем.

Результаты работы были использованы при выполнении государственного контракта по верификации автоматных программ [2 – 5]. По результатам работы опубликованы две статьи в издании из перечня ВАК:

1. *Гуров В. С., Яминов Б. Р.* Верификация автоматных программ при помощи верификатора UniMod.Verifier // Научно-технический вестник СПбГУ ИТМО. Вып. 53. Автоматное программирование. 2008, с. 162 – 176.
2. *Кочелаев Д. Ю., Лагунов И. А., Хасянзянов Б. З., Яминов Б. Р.* Инструментальное средство для поддержки автоматного программирования *UniMod 2*: проектирование, валидация, верификация, реализация // Научно-технический вестник СПбГУ ИТМО. Вып. 53. Автоматное программирование. 2008, с. 251– 257.

ИСТОЧНИКИ

1. *Шалыто А. А., Туккель Н. И.* SWITCH-технология – автоматный подход к созданию программного обеспечения «реактивных» систем // Программирование. 2001. № 5. http://is.ifmo.ru/works/switch/l/
2. Отчет по контракту о верификации автоматных программ. Первый этап. http://is.ifmo.ru/verification/_2007_01_report-verification.pdf
3. Отчет по контракту о верификации автоматных программ. Второй этап. http://is.ifmo.ru/verification/_2007_02_report-verification.pdf
4. Отчет по контракту о верификации автоматных программ. Третий этап. http://is.ifmo.ru/verification/_2007_03_report-verification.pdf
5. Отчет по контракту о верификации автоматных программ. Четвертый этап. http://is.ifmo.ru/verification/_2007_04_report-verification.pdf
6. *Кларк Э., Грамберг О., Пелед Д.* Верификация моделей программ: Model Checking. М.: МЦНМО, 2002.
7. *Шалыто А. А., Красс А. С.* Отчет о патентных исследованиях по теме «Разработка технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода. http://is.ifmo.ru/verification/_2007_01_patent-verification.pdf
8. *SPIN home page.* http://SPINroot.com
9. *Гуров В. С., Мазин М. А., Шалыто А. А. UniMod* – инструментальное средство для автоматного программирования // Научно-технический вестник СПбГУ ИТМО. Вып. 30. Фундаментальные и прикладные исследования информационных систем и технологий. 2006, с. 32 – 44. http://is.ifmo.ru/works/_instrsr.pdf
10. *Robby, Dwyer M., Hatcliff J.* Bogor: A Flexible Framework for Creating Software Model Checkers /TAIC PART 2006, pp. 3 – 22.
11. *Robby, Dwyer M., Hatcliff J.* Bogor: An Extensible and Highly-Modular Model Checking Framework /Proceedings of the Fourth Joint Meeting of the

European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003).

12. *Егоров К. В., Шалыто А. А.* Разработка верификатора автоматных программ // Научно-технический вестник СПбГУ ИТМО. Вып. 53. Автоматное программирование. 2008, с. 177 – 188. http://is.ifmo.ru/papers/egorov/automata_verificator.pdf
13. *Киракозов А. Х., Шалыто А. А., Яминов Б. Р.* Система управления автомобильной сигнализацией. http://is.ifmo.ru/unimod-projects/alarmsystem/
14. *Курбацкий Е. А.* Верификация программ, построенных на основе автоматного подхода с использованием программного средства *SMV* // Научно-технический вестник СПбГУ ИТМО. Вып. 53. Автоматное программирование. 2008, с. 137 – 144. http://books.ifmo.ru/ntv/ntv/53/ntv_53.pdf
15. *Вельдер С. Э., Шалыто А. А.* Методы верификации моделей автоматных программ // Научно-технический вестник СПбГУ ИТМО. Вып. 53. Автоматное программирование. 2008, с. 123 – 136. http://books.ifmo.ru/ntv/ntv/53/ntv_53.pdf
16. *Лукин М. А., Шалыто А. А.* Верификация автоматных программ использованием верификатора *SPIN* // Научно-технический вестник СПбГУ ИТМО. Вып. 53. Автоматное программирование. 2008, с. 145 – 161. http://books.ifmo.ru/ntv/ntv/53/ntv_53.pdf