Научно-технический вестник Санкт-Петербургского государственного университета информационных технологий, механики и оптики. Вып. 53. Автоматное программирование. 2008, с. 162–176.

УДК 004.4’242

ВЕРИФИКАЦИЯ АВТОМАТНЫХ ПРОГРАММ ПРИ ПОМОЩИ ВЕРИФИКАТОРА *UNIMOD.VERIFIER*

В. С. Гуров, Б. Р. Яминов

В статье описан разработанный авторами верификатор автоматных программ, созданных при помощи инструментального средства для поддержки автоматного программирования *UniMod*. Верификатор работает за счет интеграции инструментального средства *UniMod* и верификатора программ *Bogor*. При использовании разработанного верификатора отсутствует необходимость преобразовывать автоматную программу во входной язык верификатора. Требования к программе записываются на языке темпоральной логики *LTL*.

Введение

В настоящей работе описывается верификация автоматных программ при помощи верификатора *UniMod.verifier*, который разработан на основе интеграции инструментального средства *UniMod* [1, 2], предназначенного для создания и запуска автоматных программ, и верификатора программ *Bogor* [3, 4].

В начала описывается метод верификации, который используется в верификаторе *UniMod.verifier,* а затем приводится пример верификации автоматной программы.

Метод верификации

Постановка задачи

Предлагаемый метод предназначен для решения задачи верификации автоматных программ. Автоматная программа [5] может рассматриваться как реактивная система, состоящая из трех компонент:

- источники событий – создают события в системе;
- объекты управления – объекты, выполняющие действия;
- управляющая система – модуль, принимающий события от источников событий, и вызывающий действия объектов управления. В автоматной программе данный модуль реализован, как система иерархически связанных конечных автоматов.

Верификация автоматных программ состоит в проверке того, что управляющая система работает корректно (корректность реализации источников событий и объектов управления должна проводиться отдельно). Корректность автоматной программы определяется выполнением темпоральных утверждений вида «если произошло событие `e1`, то когда-нибудь будет вызвано действие `z1`» или «если `x1` всегда неверно (`x1` всегда `false`), то автомат никогда не попадет в состояние `s2`». Утверждения, которые требуется проверить, называют *требованиями*. В том случае, если система автоматов удовлетворяет требованиям (сформулированные утверждения выполняются для каждой истории работы системы), считается, что верификация завершена успешно. Если же утверждения не выполняются, и существует последовательность действий, которая приводит систему автоматов к нарушению сформулированных требований, то выводится этот набор действий. Его называют «*контрпример*».

Для решения задачи верификации автоматных программ было выполнено несколько работ, в том числе работа [7], в которой был предложен метод верификации автоматных программ с помощью верификатора *SPIN*. В настоящей работе предлагается оригинальный метод верификации автоматных программ с помощью *эмуляции* (или имитации) работы автоматной программы. Этот метод позволяет значительно снизить сложность преобразований исходной автоматной программы, необходимых для верификации, по сравнению с методом, описанным в работе [7].

Верификация программ на основе алгоритма двойного поиска в глубину

Верификация программ может выполняться с использованием алгоритма двойного поиска в глубину [8]. Этот алгоритм используется во многих верификаторах, в том числе в верификаторах *SPIN* и *Bogor*. Верификация с применением алгоритма двойного поиска в глубину выполняется следующим образом.

Сначала для верифицируемой программы строится модель *Крипке* [8] – граф элементарных (атомарных) состояний, в которых может находиться программа, и переходов между ними. Модель *Крипке* является подробной схемой работы программы, в которой в каждом состоянии четко определены элементарные свойства программы.

Требования к программе формулируются в виде формулы темпоральной логики. Такие формулы позволяют специфицировать работу программы *во времени*. Темпоральные формулы состоят из *предикатов* – элементарных утверждений о программе, логических операторов («не», «и», «или») и темпоральных операторов – операторов, описывающих выполнение утверждений во времени.

Для применения алгоритма двойного поиска в глубину используется темпоральная логика *LTL* (*Linear Temporal Logic*) [8]. В этой логике применяются следующие темпоральные операторы:

- **X** (neXt) – «*X p*» верно тогда, когда в следующий момент времени в программе будет выполняться предикат *p*;
- **G** (Globally) – «*G p*» верно, если во время работы программы всегда выполняется *p*;
- **F** (Future) – «*F p*» верно, если в будущем наступит момент, когда выполнится *p*;
- **U** (Until) – «*p U q*» верно, если в программе в каждый момент времени выполняется *p* до тех пор, пока не выполнится *q*. При этом *q* обязательно должно когда-либо выполниться;
- **R** (Release) – «*q R p*» верно, если *p* выполняется до тех пор, пока не станет выполняться *q* (включая момент, когда выполнится *q*), или всегда, если *q* не выполнится никогда.

Формула *LTL*, описывающая требования к программе, преобразуется в автомат *Бюхи* [8] – конечный автомат над бесконечными словами. Переходы автомата *Бюхи* помечены предикатами из исходной формулы *LTL*. Поскольку задача верификатора – найти контрпример, если он существует, автомат *Бюхи* строится для *отрицания* формулы *LTL*. Автомат *Бюхи* допускает любые последовательности значений предикатов, которые не удовлетворяют требованиям.

Далее модель *Крипке* преобразуется в автомат *Бюхи* для того, чтобы построить пересечение его с автоматом *Бюхи*, построенным по отрицанию формулы *LTL*. Оно также является автоматом *Бюхи*. Для построенного пересечения автоматов запускается алгоритм *двойного поиска в глубину* [8], который находит допускающую последовательность предикатов, если она существует. Если эта последовательность существует, то:

- она допускается автоматом *Бюхи*, построенным по модели *Крипке*. Следовательно, эта последовательность является историей работы исходной программы;
- последовательность допускается автоматом *Бюхи*, построенным из отрицания формулы *LTL*. Следовательно, эта последовательность является историей, нарушающей проверяемые требования.

Таким образом, найденная последовательность предикатов является контрпримером.

В верификаторе *Bogor* явно не строятся модель *Крипке*, автомат *Бюхи*, который строится по модели *Крипке,* и его пересечение с автоматом *Бюхи*. Верификатор получает на вход программу, написанную на входном языке верификатора. Это позволяет верификатору выделять в программе элементарные действия, поскольку в семантике языка верификатора определено, какие действия совершаются атомарно, как откатывать назад эти действия, возвращая систему в предыдущее состояние, как вычислять состояния объектов (переменных) и какие операторы порождают недетерминированность. Таким образом, верификатор может управлять исполнением программы, например, совершать элементарные шаги, «отменять» элементарные шаги (откатываться назад), вычислять глобальное состояние системы. Все это необходимо для работы алгоритма двойного поиска в глубину.

Такие возможности позволяют верификатору избежать явного построения пересечения автомата модели *Крипке* с автоматом *Бюхи*. Для того чтобы так работать, верификатору *Bogor* необходимо выполнять следующие действия.

1. Вычислять глобальное состояние программы. Оно должно однозначно определять поведение программы.
2. Совершать элементарный шаг работы программы. Такой шаг является переходом программы из одного глобального состояния в другое без посещения иных глобальных состояний.
3. Откатывать назад элементарный шаг работы программы. При этом программа возвращается в предыдущее состояние.
4. В каждом состоянии определять возможные элементарные шаги.
5. Определять значения набора предикатов программы, используемых в требованиях.

Верификация автоматных программ с использованием алгоритма двойного поиска в глубину

В методе эмуляции указанные в предыдущем разделе действия осуществляются следующим образом.

1. Глобальное состояние программы складывается из набора текущих состояний каждого автомата.

   Верификатор *UniMod.verifier* использует инструментальное средство *UniMod* для работы с автоматной программой. Это средство хранит текущие состояния каждого автомата, выполняет по команде разработанного верификатора обработку события и т.д. При обработке события информация о сделанных переходах в автоматах, о выполненных действиях и о новом глобальном состоянии передается из *UniMod* в *UniMod.verifier*. Теоретически для построения модели *Крипке* для системы автоматов необходимо было бы построить один автомат как пересечение всех автоматов системы. Однако это делается неявно в самом инструментальном средстве *UniMod*. При этом *UniMod.verifier* сразу получает информацию о глобальном состоянии системы автоматов, как набор состояний каждого автомата.
2. Элементарный шаг работы программы – это обработка системой автоматов событий. В результате обработки может смениться набор состояний автоматов.
3. Поведение автоматной программы (без учета объектов управления) определяется набором состояний автоматов. Таким образом, для отката назад достаточно вернуть автоматы в те состояния, в которых они находились до выполнения шага вперед.
4. Для автоматной программы строго определена схема работы системы автоматов, последовательность передачи управления между автоматами. Кроме того, система работает в одном потоке. Поэтому недетерминированность в работе системы возникает лишь в результате разных последовательностей входных событий, а также в результате различных возможных значений переменных, запрашиваемых у объектов управления.

Поэтому в методе эмуляции возможные элементарные шаги определяются следующим образом.

Перед совершением очередного элементарного шага определяется набор событий, которые система автоматов может обработать в текущем глобальном состоянии, и каждое из этих событий затем используется для создания одной из историй работы программы. Аналогично, при необходимости вычислить условие перехода, в котором участвуют переменные объектов управления, такое условие принимается равным `True` или `False`. Оба варианта используются для создания двух различных историй работы программы.

5. Для вычисления предикатов при совершении элементарного шага сохраняется следующая информация:
   - состояния автоматов до выполнения шага;
   - обрабатываемое событие;
   - список вычисленных значений условий на переходах;
   - список вызванных действий у объектов управления.

   Эта информация позволяет вычислять значения предикатов. В методе эмуляции поддерживается следующий набор предикатов:
   - `wasEvent(e)` – возвращает `True`, если в последнем шаге было выбрано для обработки событие `e`, и `False` – в противном случае;
   - `wasInState(sm, s)` – возвращает `True`, если перед последним шагом автомат `sm`, находился в состоянии `s`;
   - `isInState(sm, s)` – возвращает `True`, если после совершения последнего шага автомат `sm` находится в состоянии `s`;
   - `cameToState(sm, s)` – возвращает `True`, если после совершения последнего шага автомат `sm` сменил свое состояние на `s`. Это то же самое, что `(isInState(sm, s) && !wasInState(sm, s))`;
   - `cameToFinalState()` – возвращает `True`, если после совершения шага корневой автомат модели перешел в конечное состояние. Это означает, что автоматная программа завершила работу.
   - `wasAction(z)` – возвращает `True`, если в ходе выполнения шага было вызвано выходное воздействие `z`;
   - `wasFirstAction(z)` – возвращает `True`, если в ходе выполнения шага первым вызванным действием было `z`;
   - `wasLastAction(z)` – возвращает `True`, если в ходе выполнения шага последним вызванным действием было `z`;
   - `getActionIndex(z)` – возвращает 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)` – возвращает `True`, если в ходе выполнения последнего шага один из переходов был помечен условием `g`, и его значение было определено как `True`. `g` описывает целое условие, а не значение одной переменной. Например, `g = «!o1.x1 && o1.x2»`;
   - `wasFalse(g)` – возвращает `True`, если в ходе выполнения последнего шага на одном из переходов встречается условие `g`, и его значение было определено как `False`.

Сравнение метода эмуляции с известным методом

В работе [7] верификация системы автоматов производится следующим образом.

1. Система автоматов преобразуется в модель *Крипке*, записанную на входном языке верификатора *SPIN*.
2. Требования к системе автоматов переводятся в термины построенной модели.
3. Модель верифицируется верификатором *SPIN*. В случае ошибки выдается контрпример в терминах входного языка *SPIN*.
4. Контрпример переводится в термины исходной системы автоматов.

Эта схема верификации изображена на рис. 1.

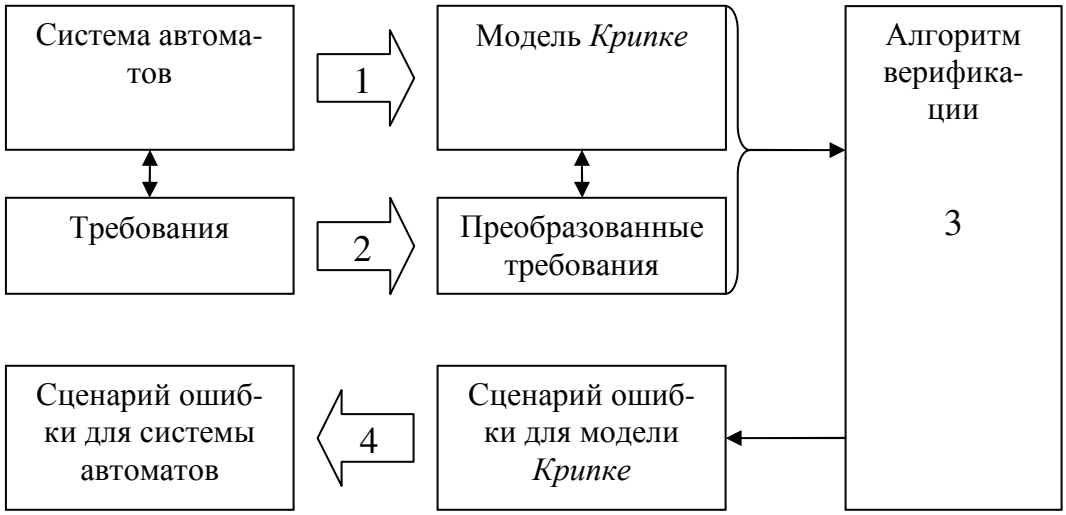

Рис. 1. Схема верификации с явным построением модели *Крипке*

Особенностью метода эмуляции является то, что он не требует дополнительных преобразований автоматной программы или контрпримера, сгенерированного верификатором. Модель *Крипке* не строится явно, а алгоритм верификации работает непосредственно с системой автоматов. В результате не требуются ни преобразование системы во входной язык верификатора, ни преобразование требований, ни преобразование сценария ошибки, поскольку сценарий сразу выдается в терминах состояний и переходов в системе автоматов. Схема верификации по методу эмуляции изображена на рис. 2.

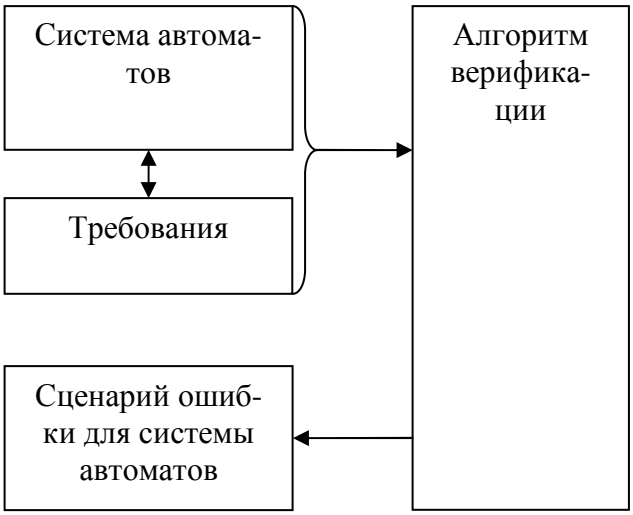

Рис. 2. Схема верификации методом эмуляции

Применение верификатора

Верификатор *UniMod.verifier* был применен для верификации автоматной программы, моделирующей работу банкомата. В следующих разделах описывается автоматная программа банкомата, а затем применение верификатора *UniMod.verifier* для верификации корректности работы банкомата.

Описание банкомата

Постановка задачи

Банкомат – это устройство, автоматизирующее операции по выдаче и переводу денег, хранящихся в банке, лицу, которому они принадлежат. Идентификация каждого клиента происходит с помощью имеющейся у него кредитной карты банка и соответствующего карте секретного *PIN*-кода. Поэтому человек может вне банка снимать деньги или оплачивать определенные услуги.

Сформулируем основные требования к устройству банкомата. Он должен:

- идентифицировать клиента;
- выполнять операции «Показать доступные средства» и «Снять определенную сумму денег»;
- уметь связываться с банком.

Клиентская программа запускается и предлагает пользователю выполнять различные операции с его личной картой.

Первое, что требуется от пользователя – вставить карту. Далее пользователь вводит свой личный *PIN*-код. Если на сервере не найдена запись о счете, с введенным номером и *PIN*-кодом, то работа с этой картой прекращается. Если же *PIN*-код и номер счета был введён правильно, то пользователю предлагается выполнить одну из следующих операций:

- «Забрать карту» – возврат карты. Все текущие операции отменяются, и карта возвращается на руки пользователю;
- «Баланс» – отображает текущий остаток на счете, выводя его на экран и предоставляя возможность распечатать;

- «Снятие денег» – производит операцию снятия денег с карты. Для этого пользователь должен ввести сумму, которую он хочет снять. Клиент пошлёт запрос на сервер о текущем балансе, и получит ответ. Если на карте есть достаточно денег, то операция на сервере завершится успешно, и банкомат выдаст требуемую сумму денег. Так же при нажатии на кнопку «Печать» будет напечатан чек по данной операции. Если обнаружится, что на карте недостаточно денег, то с карточки ничего не снимется, и клиент выводит соответствующее сообщение на экран.

После возврата карты, пользователь может вставить её снова либо уйти, нажав кнопку «Выход».

Проектирование

Программа банкомата состоит из двух частей – клиентской и серверной. В клиентской части реализован пользовательский интерфейс (*AClient*), а также интерфейс отправки запросов на сервер (*AServer*). Серверная часть производит операции со счетами.

Роль сервера исполняет класс *Server*, написанный и запускающийся отдельно. Поведение клиента моделируется автоматом *AClient* и вложенным в него автоматом *AServer*.

Поставщики событий и объекты управления

Поставщики событий:

1. *HardwareEventProvider* – системные события, генерируемые оборудованием;
2. *HumanEventProvider* – события, инициируемые пользователем;
3. *ServerEventProvider* – ответы на запросы, поступающие от сервера;
4. *ClientEventProvider* – запросы, поступающие на сервер.

Объекты управления:

5. *FormPainter* – визуализация работы;
6. *ServerQuery* – отправляет запросы на сервер;
7. *ServerReply* – отвечает на клиентские запросы.

Схема связей

На рис. 3 изображена схема связей автоматов *AClient* и *AServer*.

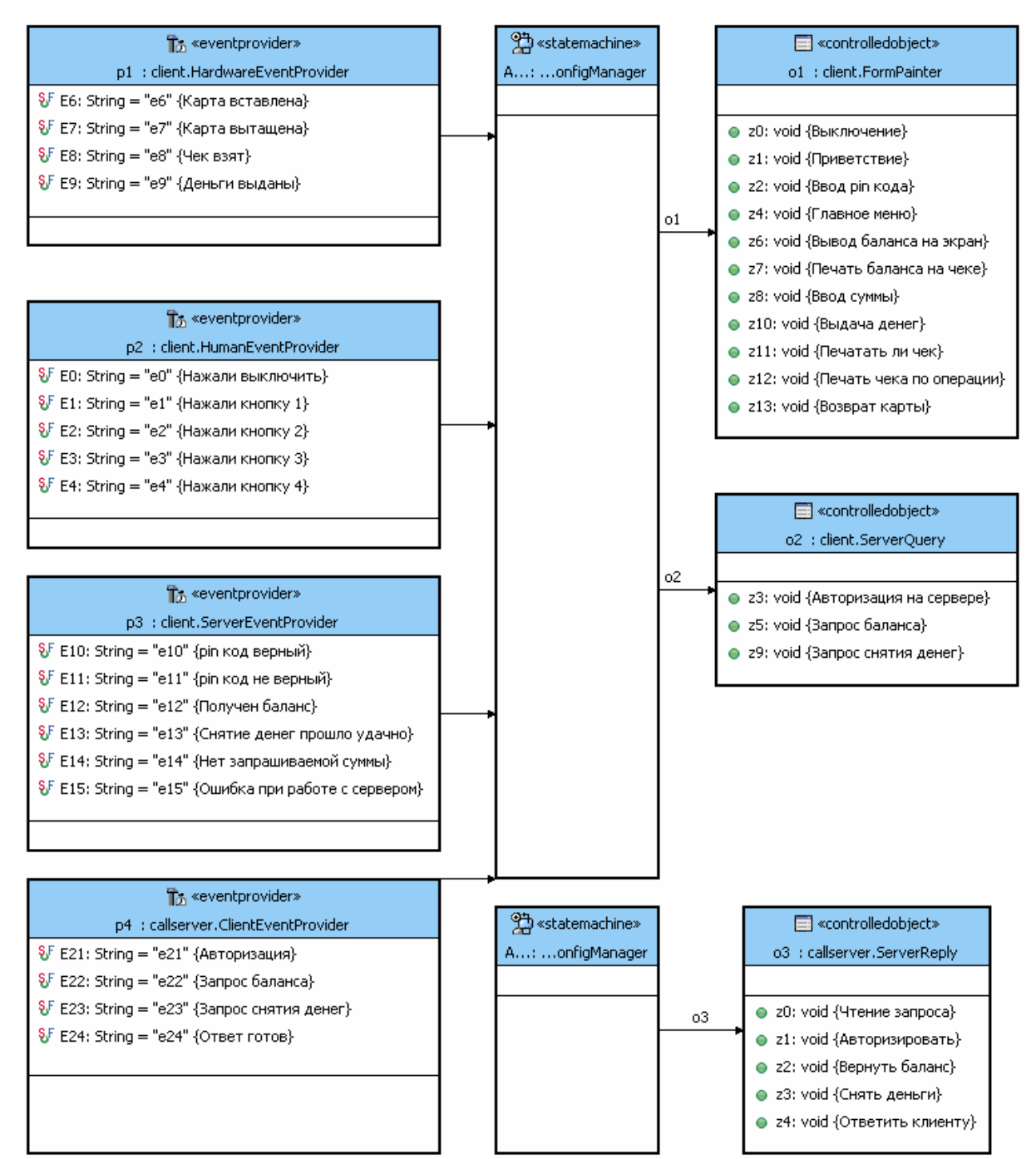

Рис. 3. Схема связей. Несмотря на отсутствие связи между автоматами, *AServer* вложен в *AClient*.

Автоматы

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

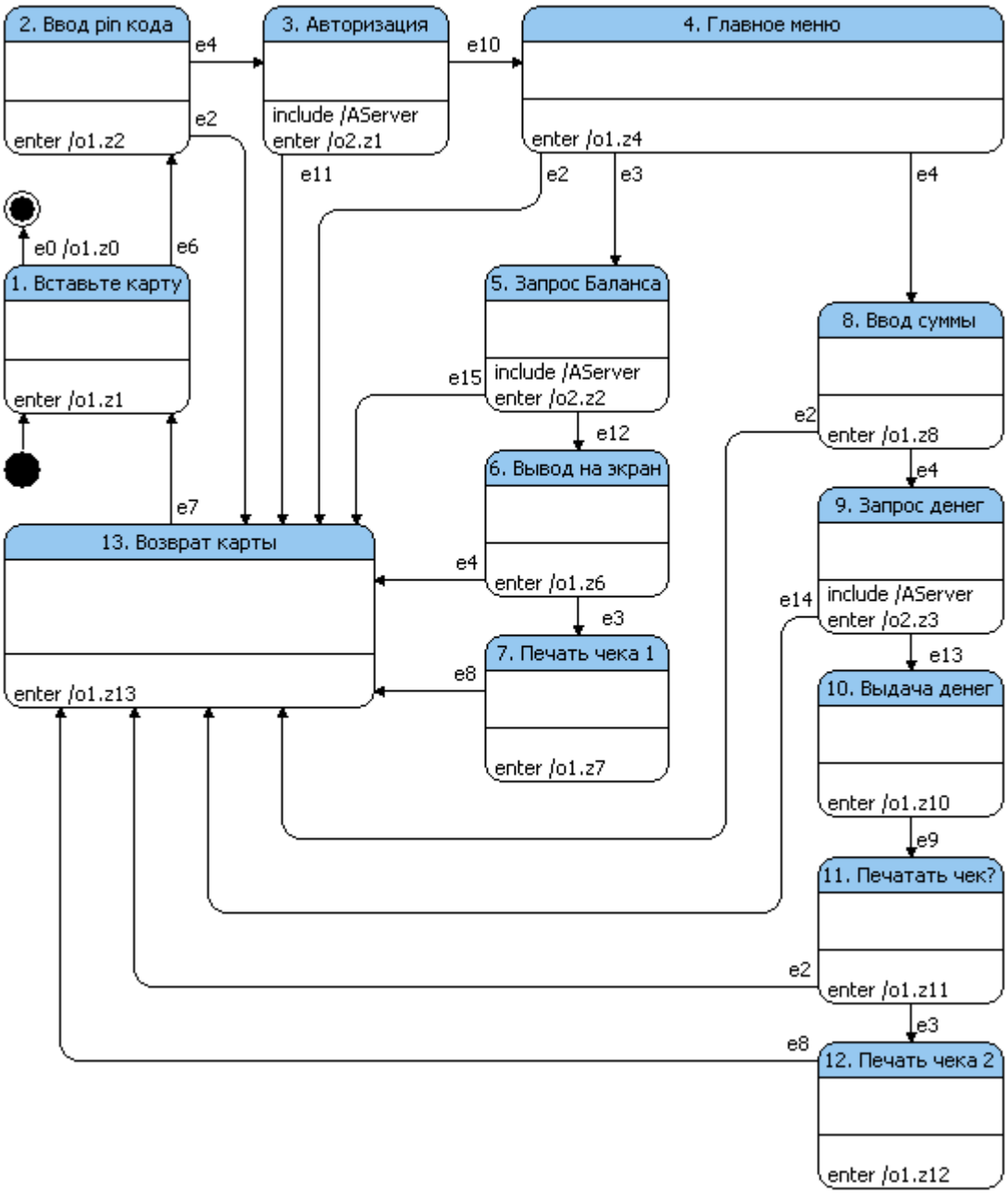

Рис. 4. Автомат *AClient*.

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

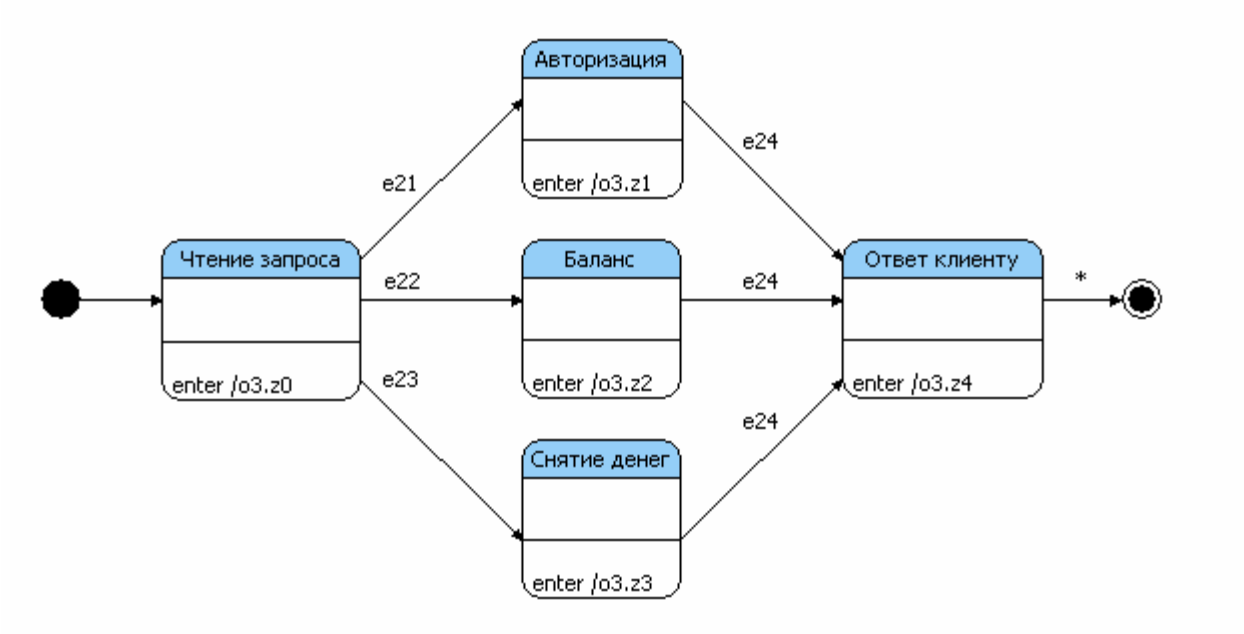

Рис. 5. Автомат *AServer*.

Верификация UniMod-банкомата

Используем верификатор *UniMod.verifier* для верификации системы управления банкоматом, описанной выше.

Банкомат выдает деньги только после авторизации

Проверим, что пользователь банкомата не получит денег, пока не введет правильный *PIN*-код. Словесная формулировка требования непосредственно переводится в темпоральную логику *LTL*:

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

где `U` – темпоральный оператор `Until` – «пока не». Как показано на рис. 3, в *UniMod*-банкомате выдача денег происходит действием `o1.z10`, а правильно введенный *PIN*-код характеризуется событием `e10`. Таким образом, формула для верификации принимает следующий вид:

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

где предикат `o1.z10` означает, что было выполнено действие `o1.z10`, а предикат `e10` означает, что произошло событие `e10`. Запишем эту *LTL*-формулу языке *BIR* — 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. Стоит заметить, что вместо темпорального оператора `Until` здесь используется его модификация `weakUntil`. Разница между ними в том, что `p Until q` требует, чтобы `q` когда-нибудь выполнилось, в то время как `p weakUntil q` этого не требует. Действительно, это оправдано в данном случае, потому что не гарантировано, 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*-кода, выполняется в системе автоматов, управляющих банкоматом.

Деньги не выдаются, пока не сделан соответствующий запрос

Проверим, что деньги не выдаются, пока не сделан соответствующий запрос. Переведем формулировку в *LTL*-формулу:

```
[не выдаются деньги] U [сделан запрос на выдачу денег]
```

Деньги выдаются с выполнением действия `o1.z10`, а запрос выдачи денег посылается на сервер автоматом *AServer* при событии `e23`, в соответствии со схемой автомата, изображенной на рис. 5. Тогда формула принимает следующий вид:

```
!o1.z10 U e23
```

На языке *BIR* эта формула запишется следующим образом:

```
LTL.temporalProperty (
  Property.createObservableDictionary (
    Property.createObservableKey("money_requested",
        AutomataModel.wasEvent(model, "e23")),
    Property.createObservableKey("give_money",
        AutomataModel.wasAction(model, "o1.z10"))
  ),

  LTL.weakUntil (
    LTL.negation(LTL.prop("give_money")),
    LTL.prop("money_requested")
```

```
        )
    );
```

Однако, хотя на первый взгляд формула кажется выполняющейся в автомате, верификатор выдает ошибку со следующим сценарием:

```
Model [ step [0] event [null] guards [null] transitions [null] actions [null] states [null] ] fsaState
[T0_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 [T0_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 [T0_init]
Model [ step [2] event [e6] guards [true->true] transitions [1. Вставьте карту#2. Ввод pin кода#e6#true] ac-
tions [o1.z2] states [(/AClient:9. Запрос денег/AServer) - (Чтение запроса); (/AClient:5. Запрос
Баланса/AServer) - (Авторизация); (/AClient:3. Авторизация/AServer) - (Чтение запроса); (/AClient) - (2. Ввод
pin кода)] ] fsaState [T0_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) -
(Авторизация); (/AClient:3. Авторизация/AServer) - (Чтение запроса); (/AClient) - (3. Авторизация)] ] fsaS-
tate [T0_init]
Model [ step [4] event [e10] guards [true->true] transitions [3. Авторизация#4. Главное меню#e10#true] ac-
tions [o1.z4] states [(/AClient:9. Запрос денег/AServer) - (Чтение запроса); (/AClient:5. Запрос
Баланса/AServer) - (Авторизация); (/AClient:3. Авторизация/AServer) - (Чтение запроса); (/AClient) - (4.
Главное меню)] ] fsaState [T0_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) -
(Авторизация); (/AClient:3. Авторизация/AServer) - (Чтение запроса); (/AClient) - (8. Ввод суммы)] ] fsaState
[T0_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) -
(Авторизация); (/AClient:3. Авторизация/AServer) - (Чтение запроса); (/AClient) - (9. Запрос денег)] ] fsaS-
tate [T0_init]
Model [ step [7] event [e13] guards [true->true] transitions [9. Запрос денег#10. Выдача денег#e13#true] ac-
tions [o1.z10] states [(/AClient:9. Запрос денег/AServer) - (Чтение запроса); (/AClient:5. Запрос
Баланса/AServer) - (Авторизация); (/AClient:3. Авторизация/AServer) - (Чтение запроса); (/AClient) - (10.
Выдача денег)] ] fsaState [bad$accept_all]
```

Как видно, на седьмом шаге выполнилось действие `o1.z10`, хотя за всю историю не происходило события `e23`. Посмотрим, что привело к выдаче денег. На шаге 6 автомат *AClient* попал в состояние «9. Запрос денег», в которое вложен автомат *AServer*. При этом выполнилось действие `o2.z3`, которое, как изображено на рис. 3, означает «Запрос снятия денег». Далее произошло событие `e13`, которое означает «Снятие денег прошло удачно». Возникает вопрос, каким образом произошел запрос снятия денег, если не происходило события `e23`. На самом деле, объект управления `o2` (*ServerQuery*) служит для того, чтобы создавать события генератора событий `p4` (*ClientEventProvider*). Объект управления `o2` реализован так, что при вызове действия `o2.z3` («Запрос снятия денег») генерируется событие `e23` («Запрос снятия денег»).

Верификатор *UniMod.verifier* не работает с объектами управления при верификации, поэтому он не может автоматически учитывать их логику при верификации. Таким образом, верификатор «не знает», что если выполнилось событие `o2.z3`, то обязательно произойдет событие `e23`. Для того чтобы все же верифицировать рассматриваемое свойство банкомата, добавим необходимую логику прямо в верифицируемое свойство, в виде «при условии, что выполняется необходимая логика, верифицируемое свойство тоже выполняется».

Итак, интересующая в данный момент логика объекта управления `o2` заключается в том, что если было выполнено действие `o2.z3`, то будет сгенерировано событие `e23`. Запишем это в логике *LTL*:

```
G (o2.z3 → X e23)
```

где `X` – *LTL*-оператор *Next*. Добавим теперь полученное условие в верифицируемую формулу, используя оператор следования, как было предложено выше:

```
( G (o2.z3 → X e23) ) → ( !o1.z10 U e23 )
```

Теперь запишем эту формулу на языке *BIR*:

```
LTL.temporalProperty (
  Property.createObservableDictionary (
    Property.createObservableKey("server_request_money",
        AutomataModel.wasAction(model, "o2.z3")),
    Property.createObservableKey("money_requested",
        AutomataModel.wasEvent(model, "e23")),
    Property.createObservableKey("give_money",
        AutomataModel.wasAction(model, "o1.z10"))
  ),

  LTL.implication(

    /* Ограничение: o2.z3 генерирует e23 */
    LTL.always (LTL.implication (
      LTL.prop("server_request_money"),
      LTL.next(
        LTL.prop("money_requested")
      )
    )),

    /* Свойство для проверки */
    LTL.weakUntil (
      LTL.negation(LTL.prop("give_money")),
      LTL.prop("money_requested")
    )
  )
);
```

Верификация данной формулы верификатором *UniMod.verifier* оказывается удачной, что означает, что управляющая система банкомата действительно выполняет выдачу денег лишь после того, как запросила наличие денег с сервера.

Если произойдет ошибка, то карта будет возвращена

Проверим, что если произойдет ошибка взаимодействия банкомата с сервером, то карта будет возвращена пользователю. В темпоральной логике такое свойство можно записать следующим образом:

```
G ([произошла ошибка] → F [карта будет возвращена])
```

где `G` – темпоральный оператор *Globally*, а `F` – темпоральный оператор *Future*. Событие «Ошибка при работе с сервером» кодируется в банкомате как `e15`, а возвращение карты – это действие `o1.z13`. Тогда формула принимает следующий вид:

```
G (e15 → F o1.z13)
```

На языке BIR формула выглядит следующим образом:

```
LTL.temporalProperty (
  Property.createObservableDictionary (
```

```
      Property.createObservableKey("error",
          AutomataModel.wasEvent(model, "e15")),
      Property.createObservableKey("card_returned",
          AutomataModel.wasAction(model, "o1.z13"))
    ),

    LTL.always (LTL.implication (
      LTL.prop("error"),
      LTL.eventually (LTL.prop ("card_returned"))
    ))
  );
```

Верификация данной формулы успешна.

Безусловная выдача денег

Верифицируем заведомо ложное свойство банкомата, чтобы проверить способность верификатора находить ошибки. Например, проверим, что «пользователь всегда получает деньги». В темпоральной логике оно запишется как

```
G F [пользователь получает деньги]
```

Деньги выдаются действием `o1.z10`, так что формула принимает следующий вид:

```
G F o1.z10
```

На языке *BIR*, соответственно, это записывается как

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

  LTL.always (LTL.eventually (LTL.prop ("give_money")))
);
```

В результате верификации этой формулы верификатор выдает следующий контрпример:

```
Model [ step [0] event [null] guards [null] transitions [null] actions [null] states [null] ] fsaState
[T0_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 [T0_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 [T0_init]
Model [ step [2] event [e0] guards [true->true] transitions [1. Вставьте карту#s2#e0#true] actions [o1.z0]
states [(/AClient:9. Запрос денег/AServer) - (Top); (/AClient:5. Запрос Баланса/AServer) - (Top);
(/AClient:3. Авторизация/AServer) - (Top); (/AClient) - (s2)] ] fsaState [T0_init]
Model [ step [2] event [e0] guards [true->true] transitions [1. Вставьте карту#s2#e0#true] actions [o1.z0]
states [(/AClient:9. Запрос денег/AServer) - (Top); (/AClient:5. Запрос Баланса/AServer) - (Top);
(/AClient:3. Авторизация/AServer) - (Top); (/AClient) - (s2)] ] fsaState [bad$accept_S2]
Model [ step [2] event [e0] guards [true->true] transitions [1. Вставьте карту#s2#e0#true] actions [o1.z0]
states [(/AClient:9. Запрос денег/AServer) - (Top); (/AClient:5. Запрос Баланса/AServer) - (Top);
(/AClient:3. Авторизация/AServer) - (Top); (/AClient) - (s2)] ] fsaState [bad$accept_S2]
```

В этом контрпримере автоматная система совершает лишь два шага: переход из начального состояния в состояние «1. Вставьте карту», и затем переход по нажатию кнопки «Выключить» (собы-

тие `e0`) в конечное состояние главного автомата *AClient*. Как и предполагалось, выдачи денег не происходит в этой истории работы банкомата.

Заметим, что в контрпримере строчка с шагом «2» повторяется три раза. Это связано с тем, что после совершения второго шага автоматная система перестает работать, и шаги совершает лишь автомат *Бюхи*: из состояния `T0_init` в состояние `bad$accept_S2`.

Заключение

Рассмотренные примеры показали, что верификатор *UniMod.verifier* адекватно верифицирует предложенные свойства банкомата. Поскольку модель банкомата достаточно простая, видно, какие свойства выполняются, а какие – нет. Это дает возможность проверить, совпадают ли результаты автоматической проверки верификатором с результатами «ручной» или «мысленной» проверки. Примеры показали, что верификатор *UniMod.verifier* успешно верифицирует свойства, которые кажутся выполняющимися, и генерирует контрпримеры для свойств, которые явно не выполняются. Это позволяет говорить о том, что данный верификатор работает корректно.

Кроме того, примеры показали, что анализ генерируемых верификатором контрпримеров прост и прозрачен. Контрпример позволяет достаточно легко выяснить причину невыполнения того или иного свойства.

Несмотря на то, что верификатор не анализирует логику работы объектов управления и генераторов событий, оказалось, что достаточно просто исправить этот недостаток при необходимости. Это было сделано в примере «Деньги не выдаются, пока не сделан соответствующий запрос», где небольшое исправление верифицируемого свойства позволило осуществить его успешную верификацию.

Все это позволяет говорить о том, что верификатор *UniMod.verifier* – надежный, корректный и легкий в использовании инструмент.

Литература

1. Гуров В.С., Мазин М.А., Шалыто А.А. UniMod — Инструментальное средство для автоматного программирования // Начуно-технический вестник. Вып. 30. Фундаментальные и прикладные исследования информационных систем и технологий. СПбГУ ИТМО. 2006, с. 32–44. http://is.ifmo.ru/works/_instrsr.pdf
2. Гуров В., Мазин М., Нарвский А., Шалыто А. UML. SWITCH-технология. Eclipse // Информационно-управляющие системы. 2004. № 6. http://is.ifmo.ru/works/uml-switch-eclipse/
3. Robby, Dwyer M., Hatcliff J. Bogor: A Flexible Framework for Creating Software Model Checkers. TAIC PART 2006, pp 3–22.
4. Robby, Dwyer M., Hatcliff J. Bogor: An Extensible and Highly-Modular Model Checking Framework, March 2003. In the 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). Technical Report, SAnToS-TR2003-3.
5. Шалыто А.А. Технология автоматного программирования. / Труды первой Всероссийской научной конференции «Методы и средства обработки информации». М.: МГУ. 2003, http://is.ifmo.ru/works/tech_aut_prog/
6. Deng W., Dwyer M., Hatcliff J., Jung G., Robby, Singh G. Model-checking Middleware-based Event-driven Real-time Embedded Software, March 2003. In the Proceedings of the First International Symposium on Formal Methods for Components and Objects (FMCO 2002). Technical Report, SAnToS-TR2003-2.

7. Васильева К. А., Кузьмин Е. В. Верификация автоматных программ с использованием LTL // Моделирование и анализ информационных систем. Ярославль: ЯрГУ. 2007. Т. 14, № 1, с. 3–14.
8. Кларк Э., Грамберг О., Пелед Д. Верификация моделей программ: Model Checking. М.: МЦНМО, 2002.