МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ
(МИНОБРНАУКИ)

ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ «САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ»
(СПБГУ ИТМО)

УДК 004.4’242
№ госрегистрации 0120.0 710294
Инв. №

УТВЕРЖДАЮ
Ректор СПбГУ ИТМО,
докт. техн. наук, профессор
В. Н. Васильев

«____» ___________ 2008 г.

Подп. и дата	
Инв. № дубл.	
Взам. инв. №	
Подп. и дата	
Инв. № подл.	

РАЗРАБОТКА ТЕХНОЛОГИИ ВЕРИФИКАЦИИ УПРАВЛЯЮЩИХ ПРОГРАММ СО СЛОЖНЫМ ПОВЕДЕНИЕМ, ПОСТРОЕННЫХ НА ОСНОВЕ АВТОМАТНОГО ПОДХОДА

ПРОМЕЖУТОЧНЫЙ ОТЧЕТ ПО III ЭТАПУ «ЭКСПЕРИМЕНТАЛЬНЫЕ ИССЛЕДОВАНИЯ ПОСТАВЛЕННЫХ ПЕРЕД НИР ЗАДАЧ»

ЛИСТОВ 108

Декан факультета «Информационные технологии и программирование»
докт. техн. наук, профессор
______________ В. Г. Парфенов

Руководитель темы
заведующий кафедрой «Технологии программирования»,
докт. техн. наук, профессор
_______________ А. А. Шалыто

Ответственный исполнитель
доцент кафедры «Компьютерные технологии», канд. техн. наук
_______________ Г. А. Корнеев

2008

СПИСОК ИСПОЛНИТЕЛЕЙ

Руководитель темы Заведующий кафедрой докт. техн. наук, профессор	А. А. Шалыто	Отчет в целом
Ответственный исполнитель Доцент, канд. техн. наук	Г. А. Корнеев	Отчет в целом
Заведующий кафедрой, докт. физ.-мат. наук, профессор	В. А. Соколов	Разделы 1 и 2.
Ведущий научный сотрудник, докт. техн. наук, профессор	В. В. Антипов	Разделы 2.1. и 2.2.
Ведущий научный сотрудник, канд. техн. наук	В. В. Киселев	Разделы 2.3. и 2.4.
Ведущий научный сотрудник, канд. техн. наук	Р. Н. Котляр	Разделы 2.1. и 2.4.
Ведущий научный сотрудник, канд. техн. наук	Ю. П. Московцев	Разделы 2.2. и 2.3.
Ведущий научный сотрудник, канд. техн. наук, доцент	В. А. Третьяков	Разделы 2.2. и 2.4.
Ведущий научный сотрудник, канд. техн. наук	Г. М. Файкин	Разделы 3.1. и 3.2.
Доцент, канд. физ.-мат. наук	Е. В. Кузьмин	Разделы 3.1. и 2.3.
Старший преподаватель, канд. физ.-мат. наук	Д. Ю. Чалый	Разделы 2.1. и 2.3.
Руководитель лаборатории	С. П. Жуков	Разделы 2.3. и 2.4.
Канд. техн. наук	С. П. Новиков	Разделы 2.1. и 2.4.
Ассистент	В. С. Гуров	Разделы 2.2. и 2.4.
Руководитель ВТК, ассистент кафедры КТ	А. П. Мельничук	Раздел 1.
Член ВТК, профессор кафедры ИС	Е. Ю. Михайлова	Раздел 1.2.
Член ВТК, доцент кафедры КТ	В. Д. Наумчик	Раздел 1.1.
Член ВТК, доцент кафедры КТ	М. Ю. Осипов	Раздел 1.3.
Член ВТК, доцент кафедры КТ	А. Н. Воробьев	Раздел 1.1.
Член ВТК, доцент кафедры КТ	Ю. А. Щупак	Раздел 1.2.
Член ВТК, доцент кафедры КТ	С. В. Чириков	Раздел 1.3.
Член ВТК, доцент кафедры КТ	А. С. Сегаль	Раздел 1.1.
Член ВТК, доцент кафедры КТ	Д. Г. Шопырин	Раздел 1.2.
Член ВТК, доцент кафедры ИС	Д. А. Зубок	Раздел 1.3.
Член ВТК, ассистент кафедры ИС	В. В. Повышев	Раздел 1.2.
Член ВТК, ассистент кафедры ИС	В. В. Ильин	Раздел 1.1.
Член ВТК, ассистент кафедры ИС	М. Г. Холин	Раздел 1.3.
Аспирант	Р. А. Виноградов	Разделы 1.1. и 1.2.
Магистрант	Б. Р. Яминов	Разделы 2.4. и 3.3.
Магистрант	С. Э. Вельдер	Разделы 2.2. и 3.2.
Магистрант	М. А. Лукин	Разделы 2.1. и 3.1.
Магистрант	К. А. Васильева	Разделы 1.1. и 1.3.
Студент	Е. А. Курбацкий	Раздел 2.3.
Студент	М. Э. Дворкин	Разделы 1.1. и 2.1.
Студент	Д. С. Архипов	Разделы 1.2. и 2.2.
Студент	М. В. Домрачев	Разделы 1.3. и 2.3.
Студент	А. В. Синотова	Разделы 1.2. и 3.1.
Студент	С. В. Кубасов	Разделы 1.3. и 3.2.

РЕФЕРАТ

Объектом исследования настоящей работы являются методы автоматизации верификации автоматных моделей управляющих программ.

Цель этапа — апробация и экспериментально исследование методов верификации автоматных моделей управляющих программ, разработанных на предыдущих этапах на примере верификации системы управления моделью банкомата.

В результате выполнения третьего этапа работ были разработаны две модели банкоматов, позволившие оценить предложенные методы. Модели банкоматов были проверифицированы инструментальными средствами, созданными по предложенным методам. В ходе экспериментальных исследований при помощи данных инструментальных средств, были формально доказаны истинные свойства моделей банкоматов и построены контпримеры для ложных свойств. При этом для некоторых свойств, полагавшихся верными, также были построены контрпримеры. Это показывает эффективность разработанных методов и средств их реализующих.

В первой главе отчета описаны модели банкоматов и предложен набор свойств, которые позволяют оценить эффективность исследуемых методов.

Во второй главе модели банкоматов верифицируются на наличие свойств. При этом рассматриваются четыре метода автоматизированной верификации и инструментальные средства, построенные на их основе.

В третьей главе приведена программная документация на разработанные инструментальные средства.

ОГЛАВЛЕНИЕ

СПИСОК ИСПОЛНИТЕЛЕЙ 2
РЕФЕРАТ....3
ОГЛАВЛЕНИЕ....4
ВВЕДЕНИЕ 7
ОСНОВНАЯ ЧАСТЬ 9
1. РАЗРАБОТКА МОДЕЛЕЙ БАНКОМАТОВ 9
1.1. Автоматный банкомат 9
1.1.1. Пользовательский интерфейс 9
1.1.2. Диаграмма классов....11
1.1.3. Класс *TBankomatForm* 11
1.1.4. Класс *TAutomats* 12
1.1.4.1. Словесное описание 12
1.1.4.2. Нумерация и перечень событий (e)....12
1.1.4.3. Нумерация и перечень входных переменных (x) 12
1.1.4.4. Нумерация и перечень выходных воздействий (z)....12
1.1.4.5. Автомат “Управление банкоматом” (A0)....13
1.1.4.6. Автомат “Местонахождение денег” (A1)....14
1.1.4.7. Автомат “Местонахождение карты” (A2)....15
1.2. *UniMod*-банкомат 16
1.2.1. Пользовательский интерфейс 16
1.2.2. Поставщики событий и объекты управления....18
1.2.3. Автоматы 18
1.3. Верифицируемые свойства....20
1.3.1. Истинные свойства 20
1.3.2. Ложные свойства....21
1.4. Выводы....21
2. ВЕРИФИКАЦИЯ МОДЕЛЕЙ БАНКОМАТОВ....22
2.1. Верификация при помощи *FSM Verifier* 22
2.1.1. Описание метода 22
2.1.1.1. Построение модели системы автоматов....22
2.1.1.2. Требования — формулы темпоральной логики на модели Крипке 23
2.1.1.3. Преобразование контрпримера в модели Крипке в контрпример в автоматной модели....23
2.1.2. Верифицируемая модель банкомата 24
2.1.3. Проверяемые свойства....27
2.1.3.1. Банкомат выдает деньги только после авторизации 27
2.1.3.2. Банкомат выдает деньги только при вставленной карте 27
2.1.3.3. Банкомат не захватывает карту. 28
2.1.3.4. Изъятие карты по таймеру....28
2.1.3.5. Банкомат выдает деньги только по требованию 28
2.1.3.6. Баланс выдается только после авторизации 29
2.1.3.7. После извлечения карты её можно будет вставить снова 29
2.1.3.8. Транзакция будет завершена в случае ошибки....29
2.1.3.9. Безусловная выдача денег....29
2.1.3.10. Снятие денег после запроса баланса....31
2.1.4. Выводы....31
2.2. Верификация при помощи *CTLVerifier*....32
2.2.1. Описание метода 32

2.2.1.1. Построение модели Крипке 32
2.2.1.2. Построение редуцированного графа переходов 33
2.2.1.3. Преобразование сценария для модели Крипке в сценарий для автомата 35
2.2.2. Верификация *CTL*-свойств 35
2.2.2.1. Банкомат выдает деньги только после авторизации 36
2.2.2.2. Банкомат выдает деньги только при вставленной карте 37
2.2.2.3. Баланс выдается только после авторизации 38
2.2.2.4. Банкомат не захватывает карту 40
2.2.2.5. Банкомат выдает карту по требованию 41
2.2.2.6. Выдаваемая сумма не превышает остаток на счете 43
2.2.2.7. Банкомат выдает деньги только при указании суммы 45
2.2.2.8. Возврат карты при ошибке 46
2.2.2.9. Безусловная выдача денег 47
2.2.2.10. Повторный запрос карты 49
2.2.2.11. Снятие денег после запроса балансов 50
2.2.2.12. Возвращение карты до ошибки 51
2.2.2.13. Безошибочные пути 53
2.2.2.14. Выдача достаточной суммы денег 55
2.2.2.15. Выход в главное меню 56
2.3. Верификация при помощи *Converter* 57
2.3.1. Описание метода 57
2.3.1.1. Построение модели на языке *Promela* по визуальной автоматной программе 57
2.3.1.2. Расширение нотации верификатора *SPIN* для языка *LTL* 59
2.3.1.3. Построение контрпримера 60
2.3.2. Верификация свойств 60
2.3.2.1. Банкомат выдает деньги только после авторизации 61
2.3.2.2. Банкомат выдает деньги только при вставленной карте 62
2.3.2.3. Чек не печатается до ввода правильного *PIN*-кода 63
2.3.2.4. Выдаваемая сумма не превышает остаток на счете 64
2.3.2.5. Выдача денег без запроса 71
2.3.2.6. Возврат карты при ошибке 72
2.3.2.7. Выдача денег 73
2.4. Верификация при помощи *UniMod.verifier* 77
2.4.1. Описание метода 77
2.4.1.1. Общее описание 77
2.4.1.2. Особенности преобразований 78
2.4.1.3. Связь с верификатором *Bogor* 80
2.4.2. Верификация *UniMod*-банкомата 82
2.4.2.1. Банкомат выдает деньги только после авторизации 82
2.4.2.2. Банкомат выдает деньги только при вставленной карте 83
2.4.2.3. Банкомат печатает чек только после авторизации 84
2.4.2.4. Выдаваемая сумма не превышает остаток на счете 85
2.4.2.5. Деньги не выдаются, пока пользователь не сделает соответствующий запрос 88
2.4.2.6. Если произойдет ошибка, то карта будет возвращена 90
2.4.2.7. Безусловная выдача денег 91
2.5. Выводы 92
3. ПРОГРАММНАЯ ДОКУМЕНТАЦИЯ 93
3.1. FSM Verifier 93
3.1.1. Общие сведения 93

3.1.2. Функциональное назначение 93
3.1.3. Описание логической структуры...93
3.1.4. Используемые технические средства...94
3.1.5. Вызов и загрузка 94
3.1.6. Входные данные...94
3.1.7. Выходные данные 95
3.2. *CTL*VERIFIER 95
3.3. *UNIMOD*.VERIFIER 101
3.3.1. Общие сведения 101
3.3.2. Функциональное назначение 102
3.3.3. Описание логической структуры...102
3.3.4. Используемые технические средства...103
3.3.5. Вызов и входные данные...104
3.3.6. Выходные данные 105
ЗАКЛЮЧЕНИЕ...106
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ...107

ВВЕДЕНИЕ

Технология верификации управляющих программ со сложным поведением разрабатывается в рамках проведения научно-исследовательской работы по *лоту «Разработка технологий верификации программного обеспечения» шифр «2007-4-1.4-18-02-041» по теме «Разработка технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода»*, выполняемой в рамках Федеральной целевой научно-технической программы «Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007−2002 годы» *по государственному контракту № 02.514.11.4048 от 18.05.2007*, заключенному между Федеральным агентством по науке и инновациям и Государственным образовательным учреждением высшего профессионального образования «Санкт-Петербургский государственный университет информационных технологий, механики и оптики» на основании решения Конкурсной комиссии Роснауки № 14 (протокол от 28.04.2007 г. № 15).

Целью настоящего этапа является апробация и экспериментальное исследование методов автоматизации верификации управляющих программ, построенных на основе автоматного программирования.

Задачами этапа являются:

1. Разработка тестовых моделей банкоматов и набора верифицируемых свойств.
2. Апробация методов автоматизированной верификации управляющих программ, построенных на основе автоматного программирования.
3. Экспериментальное исследование инструментальных средств, созданных на основе методов автоматизированной верификации управляющих программ.

В настоящее время для формальной верификации программного обеспечения применяются два основных подхода: дедуктивная верификация (верификация на основе логического вывода) и верификация на модели (*Model checking*).

Дедуктивная верификация трудоемка и требует высококвалифицированных специалистов в области доказательства теорем и логического вывода.

Верификация на модели состоит из четырех основных этапов.

4. Построение модели программы.
5. Задание требований в терминах выбранного типа темпоральной логики.
6. Верификация модели с целью проверки выполнения формализованных требований.
7. Анализ контрпримера в случае несоответствия программы требованиям.

Выполнение первого этапа верификации в общем случае достаточно трудоемко в связи с необходимостью построения модели, адекватной верифицируемой программе. При этом полученная модель должна иметь конечное число состояний, так как аппарат анализа моделей с бесконечным числом состояний разработан только для отдельных классов систем (например, для вполне структурированных систем помеченных переходов). Отметим, что для эффективной проверки модели число состояний в ней должно быть не слишком большим. Однако существующие методы построения моделей для программ, написанных традиционным способом, приводят к очень большому числу состояний, так как в традиционных программах обычно не разделяются управляющие и вычислительные состояния.

Затруднение вызывает также и выполнение второго этапа верификации, так как для верификации требования должны быть сформулированы в терминах модели. При этом также встает вопрос об адекватности формально записанных требований исходным.

По сравнению с первыми двумя этапами, третий этап верификации достаточно хорошо автоматизируется. Известны инструментальные средства для верификации моделей (верификаторы), в том числе свободные, например, *NuSMV*, *SPIN* и *Bogor*. На четвертом этапе для программ общего вида при нахождении ошибки в модели часто возникают проблемы при переносе контрпримера в верифицируемую программу.

Как показано на предыдущих этапах работы, в рамках автоматного подхода проблема адекватности модели программы решается за счет того, что набор взаимодействующих автоматов, описывающий логику работы программы, близок по структуре к модели Крипке, используемой обычно при верификации на модели, и допускает простой переход к ней. При этом число состояний в получаемой модели пропорционально числу состояний и пометок переходов в системе автоматов, и поэтому сравнительно невелико.

Структурная близость системы взаимодействующих автоматов и модели Крипке также позволяет решить проблему формулировки требований к модели, так как они могут быть сформулированы в терминах исходной системы автоматов. Аналогично решается проблема переноса контрпримера, построенного при нахождении ошибок в модели.

Таким образом, при применении к автоматным программам метода *Model checking* открывается возможность автоматического преобразования системы взаимодействующих автоматов в модель Крипке. Это позволило разработать более эффективные методы верификации программ указанного класса. Апробация указанных методов и экспериментальное исследование инструментальных средств, построенных на их основе, является целью настоящего этапа работ.

Дополнительные патентные исследования, проведенные в рамках третьего этапа работы (отчет о патентных исследованиях № 2008.05.30-1 входит в состав отчетной документации по этапу), позволяют утверждать, что в настоящее время отсутствуют патенты и иные охранные документы, которые могут препятствовать применению в Российской Федерации результатов научных исследований, проводимых по контракту.

Изложенное позволяет утверждать, что результаты выполнения научно-исследовательской работы соответствуют мировому уровню разработок в рассматриваемой области.

ОСНОВНАЯ ЧАСТЬ

1. РАЗРАБОТКА МОДЕЛЕЙ БАНКОМАТОВ

В рамках второго этапа работ были разработаны методы верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода. Для апробации этих методов и экспериментального исследования инструментальных средств, построенных на их основе, требуется разработать тестовые примеры управляющих программ со сложным поведением и определить набор верифицируемых свойств.

В качестве такого примера были выбраны модели банкоматов. Отметим, что задача верификации банкоматов имеет ценность не только с теоретической, но и с практической точки зрения.

В разд. 1.1 рассматривается банкомат [16], разработанный на основе процедурного программирования с явным выделением состояний [19]. Банкомат [9], реализованный на основе инструментального средства *UniMod* [5, 4] рассматривается в разд. 1.2. Верифицируемые свойства банкоматов описаны в разделе 1.3.

1.1. Автоматный банкомат

Автоматный банкомат [16] реализован на языке *Object Pascal* (реализация *Delphi*). Архитектура банкомата основана на процедурном программировании с явным выделением состояний [19]. При описании проекта применяется подход с созданием четырех документов, описанный в работе [28].

1.1.1. Пользовательский интерфейс

На панели (рис. 1) находятся десять кнопок с цифрами, кнопки “Сброс”, “Отмена”, “Ввод”, “Начало”, “Протокол”, а также восемь обезличенных кнопок. Для вывода информации на панели имеется дисплей. Карта вводится через приемник карт (CARD). Чек выдается устройством его выдачи (RECEIPT), а деньги —в правой нижней части панели.

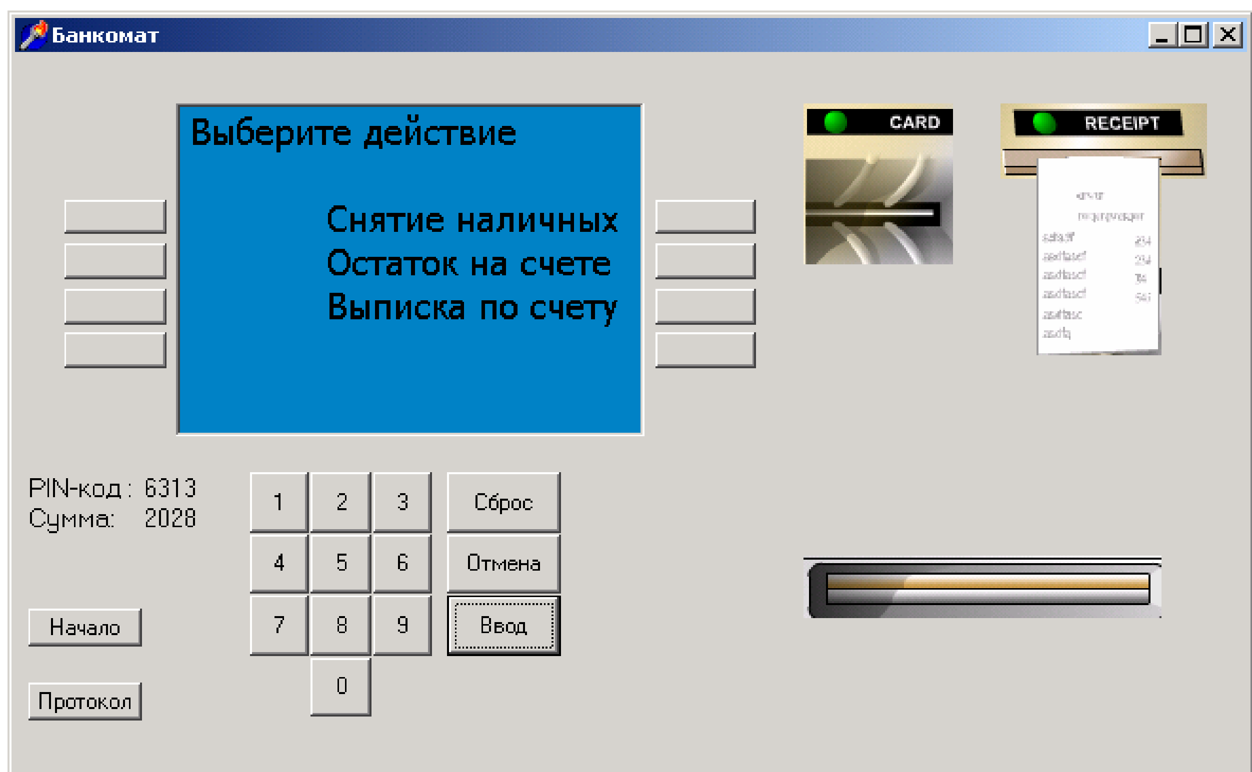


Рис. 1. Внешний вид визуализатора банкомата

PIN-код (персональный код) клиента генерируется при каждом обращении к банкомату. Также генерируется и сумма денег, находящаяся на счету клиента. Эта сумма находится в пределах от 100 до 5000.

Кнопка "Начало" нажимается, если в банкомате у клиента осталось, по его мнению, слишком мало денег, и он хочет начать "жить" сначала. При этом у него изменяется PIN-код и сумма, хранящаяся на счете.

При нажатии на кнопку "Протокол" на экране появляется окно, в котором осуществляется протоколирование работы программы (рис. 2).

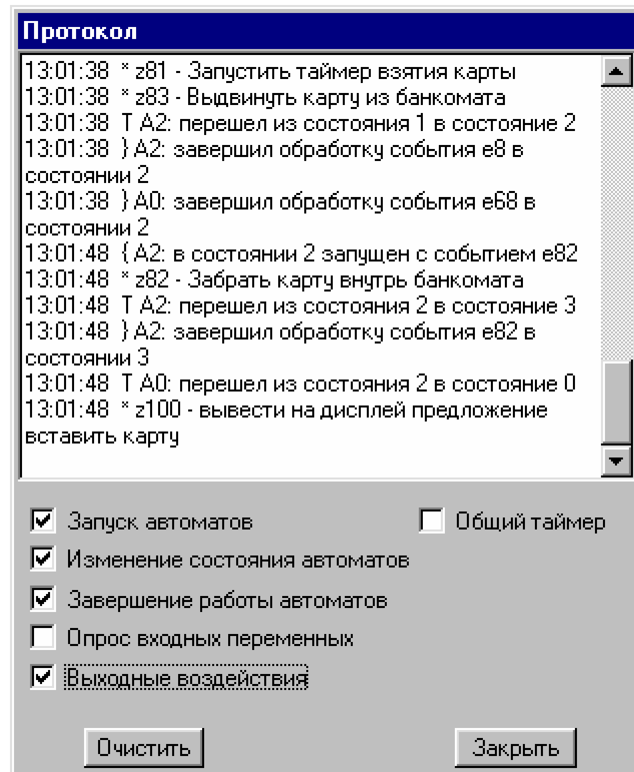


Рис. 2. Протоколирование

Процесс работы с банкоматом состоит в следующем.

- При запуске программы либо при нажатии кнопки "Начало" (если программа была запущена) на экране появляется *PIN*-код и сумма, находящаяся на счете пользователя. На дисплее формируется надпись "Вставьте карту".
- Щелчком левой кнопки мыши по изображению карты она перемещается в приемник карт. На дисплее появляется надпись "Введите ваш персональный код" (*PIN*).
- С помощью оцифрованных кнопок вводится *PIN*-код. При нажатии на каждую оцифрованную кнопку на дисплей выводится символ "X".
- Нажимается кнопка "Ввод". На дисплее появляется надпись "Выберите действие", а соответствующим кнопкам присваиваются имена "Снятие наличных", "Остаток на счете", "Выписка по счету".
- При нажатии кнопок, соответствующих остатку на счете или выписке по счету, банкомат выдает чек и возвращает карту. На дисплее появляется надпись "Возьмите, пожалуйста, карту". Если карту не удается забрать в течение 20 с, то она исчезает в приемнике карт, и ее можно получить, только заново нажав на кнопку "Начало".
- Для получения денег нажимается кнопка "Снятие наличных", а на дисплее появляется надпись "Печатать чек?" Соответствующим обезличенным кнопкам присваиваются имена "Да" и "Нет".

- При нажатии любой из этих кнопок появляется надпись “Выберите сумму”, а обезличенным кнопкам присваиваются наименования: “3000” “2000” “1500” “1000” “500” “200” “100” и “Другая”.
- При нажатии кнопки “Другая” на дисплее появляется надпись “Введите сумму”. По мере ее ввода, она отображается на дисплее и для ее получения необходимо нажать кнопку “Ввод”.
- Появляется карта, которая должна быть взята в течение 20 с, и чек, если он был затребован.
- После взятия карты банкомат выдает деньги, которые также необходимо забрать в течение двадцати секунд.

Кнопка “Сброс” прерывает текущую операцию и возвращает карту, а кнопка “Отмена” позволяет ввести заново неверно набранную информацию.

1.1.2. Диаграмма классов

В программе основными являются класс *TBankomatForm*, предназначенный для визуализации работы банкомата, и класс *TAutomats*, который реализует все автоматы и является системой управления банкоматом.

Кроме основных, программа содержит также два вспомогательных класса.

Класс *TLog* протоколирует логику работу приложения и связан с соответствующим окном, предназначенным для вывода в него протокола.

Класс *TCard* предназначен для хранения информации о карте, и содержит ряд методов для управления картой.

Диаграмма классов приведена на рис. 3.

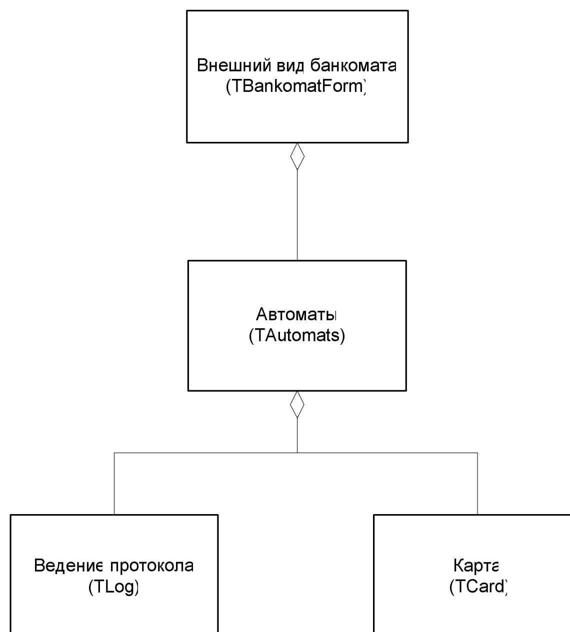


Рис. 3. Диаграмма классов

Ниже описывается каждый из перечисленных классов, причем наиболее подробно описан класс, содержащий автоматы.

1.1.3. Класс *TBankomatForm*

Класс является наследником встроенного в *Delphi* класса *TForm* и определяет окно визуализатора.

Класс *TBankomatForm* осуществляет визуализацию работы программы. При этом экземпляр этого класса получает внешние события и “препровождает” большинство из них экземпляру класса *TAutomats*, с помощью которого реализуется логика работы программы.

Рассматриваемый класс порождает форму, которая содержит дисплей (реализован в виде поля ввода типа *TMemo*), 23 кнопки (десять цифровых, три управляющих, восемь обезличенных и две дополнительные кнопки), а также изображения карты, денег и чека.

В обработчиках событий (нажатия на кнопки, на изображения и срабатывания таймера) вызываются соответствующие автоматы. Например, обработчик события *e21* “Нажатие кнопки 1” (кнопка типа *TButton* с названием *digit_1*) вызывает автомат *A0* с этим событием:

```
procedure TBankomatForm.digit_1Click(Sender: TObject);
begin
    aut.A0( 21 );
end;
```

По событиям, полученным от таймеров *MoneyTimer* и *CardTimer,* вызываются автоматы *A1* и *A2* с событиями *e72* и *e82* соответственно.

1.1.4. Класс *TAutomats*

1.1.4.1. Словесное описание

Класс реализует логику работы системы управления банкоматом. Конструктор принимает ссылку на поле дисплея (встроенный тип *TLines*).

В классе реализованы три автомата – методы вида *procedure An (e : integer)*, где *n* –номер автомата, *e* – событие. Состояние каждого автомата хранится в переменной *yn*, начальное состояние всех автоматов – нулевое.

1.1.4.2. Нумерация и перечень событий (e)

0. Сработал общий таймер
1. Вставлена карта
3. Нажатие кнопки “Сброс”
4. Нажатие кнопки “Отмена”
5. Нажатие кнопки “Ввод”
7. Извлечь деньги
8. Вернуть карту
20−29. Нажатие кнопок “0”−“9”
61−68. Нажатие первой−восьмой программируемых кнопок.
71. Получение денег клиентом
72. Сработал таймер извлечения денег
81. Получение карты клиентом
82. Сработал таймер извлечения карты

1.1.4.3. Нумерация и перечень входных переменных (x)

1. Введено меньше четырех цифр PIN-кода
2. PIN-код введен неверно
3. Превышен лимит на карте
4. Требуется печать чека

1.1.4.4. Нумерация и перечень выходных воздействий (z)

4. Очистить поле ввода PIN-кода
5. Напечатать чек
7. Извлечь деньги из банкомата
30−39. Добавление цифры “0”−“9” к PIN-коду
40−49. Добавление цифры “0”−“9” к сумме
61−67. Установить сумму равной 3000, 2000, 1500, 1000, 500, 200 и 100 соответственно.

68. Очистить поле ввода суммы
71. Запустить таймер взятия денег
72. Забрать деньги внутрь банкомата
81. Запустить таймер взятия карты
82. Забрать карту внутрь банкомата
83. Извлечь карту из банкомата
100. Вывести на дисплей предложение вставить карту
101. Вывести на дисплей предложение ввести PIN-код
102. Вывести на дисплей сообщение о завершении транзакции
103. Вывести на дисплей меню
104. Вывести на дисплей предложение о выборе суммы
105. Вывести на дисплей предложение о печати чека
106. Вывести на дисплей предложение о вводе суммы
107. Вывести на дисплей сообщение о неверном PIN-коде
108. Вывести на дисплей сообщение о превышении лимита на карте
109. Вывести на дисплей предложение о взятии карты
110. Вывести на дисплей предложение о взятии денег

1.1.4.5. Автомат “Управление банкоматом” (A0)

Автомат является основным для управления банкоматом. Он отвечает за взаимодействие с клиентом и осуществляет:

- вывод справочной информации и меню на дисплей;
- ввод данных от клиента.

Он вызывает другие автоматы с определенными событиями.

Схема связей этого автомата приведена на рис. 4, а граф переходов – на рис. 5.

Отметим, что событие *e0* не участвует в условиях переходов, но автомат запускается по этому событию.

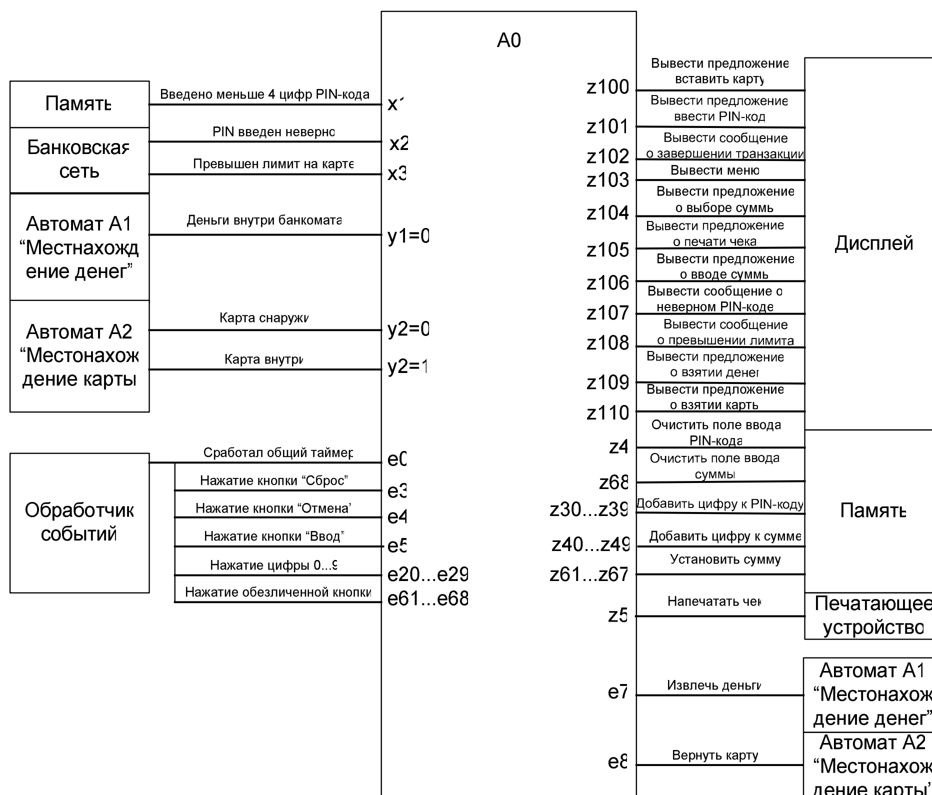


Рис. 4. Схема связей автомата “Управление банкоматом” (A0)

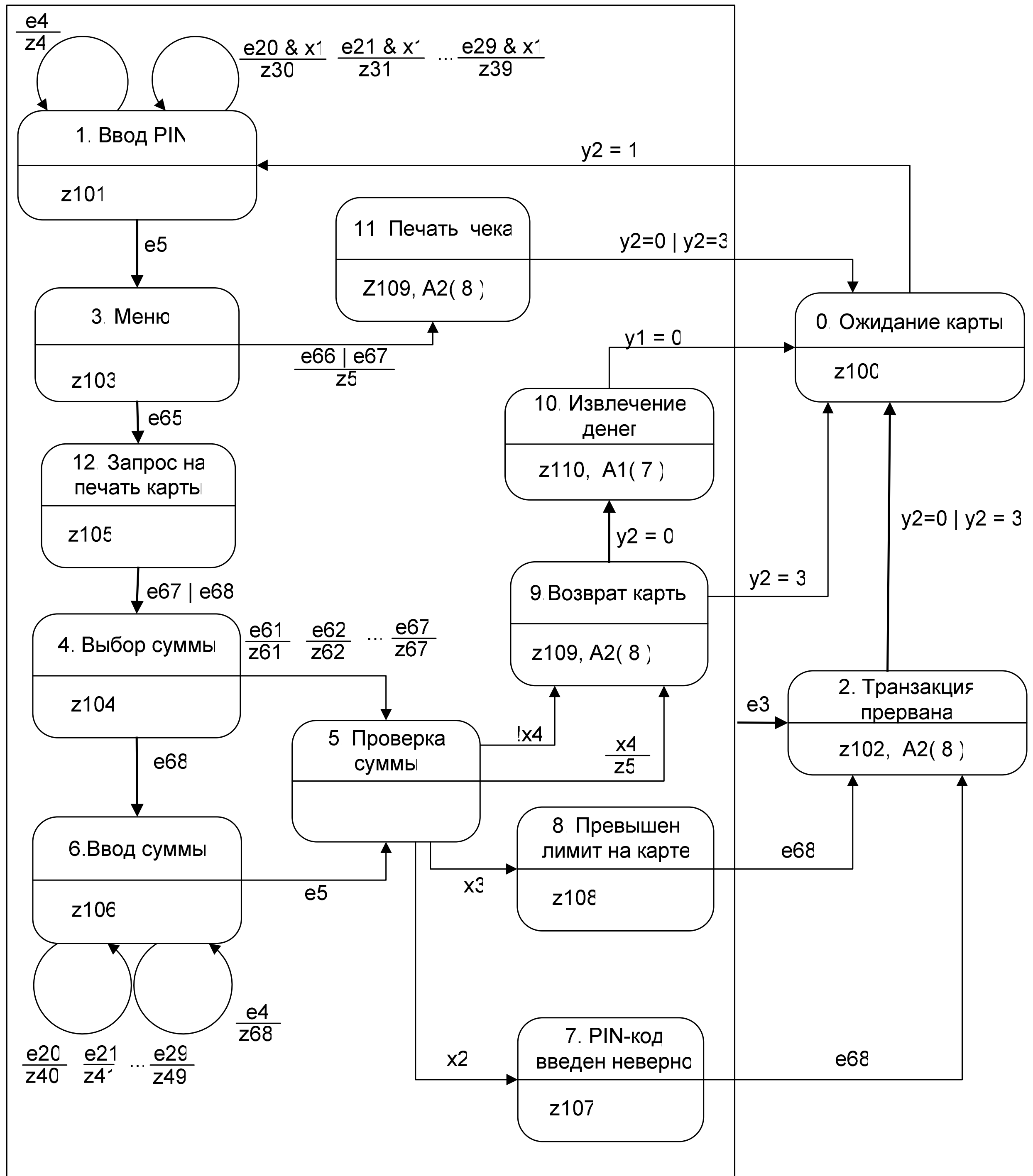


Рис. 5. Граф переходов автомата “Управление банкоматом” (A0)

1.1.4.6. Автомат “Местонахождение денег” (A1)

Автомат предназначен для управления местонахождением денег. При поступлении события *e7* “Извлечь деньги” банкомат должен обеспечить извлечение денег. Если деньги не будут взяты в течение времени, заданного таймером (двадцать секунд), то они будут втянуты внутрь банкомата.

Схема связей приведена на рис. 6, а граф переходов – на рис. 7.

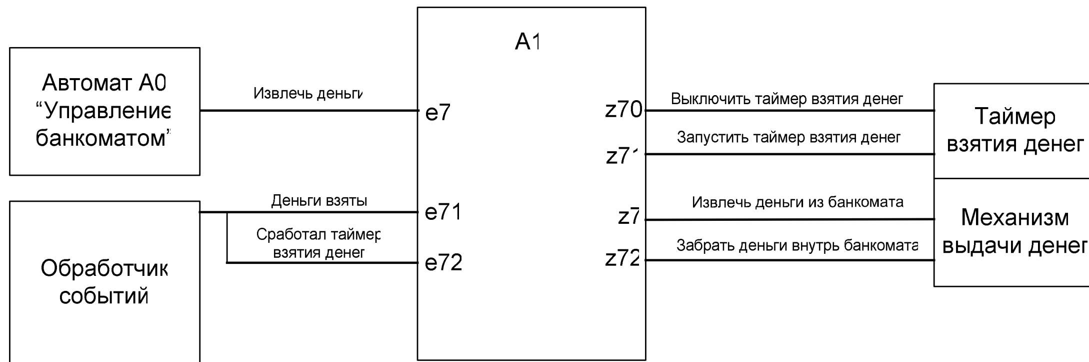


Рис. 6. Схема связей автомата "Местонахождение денег" (A1)

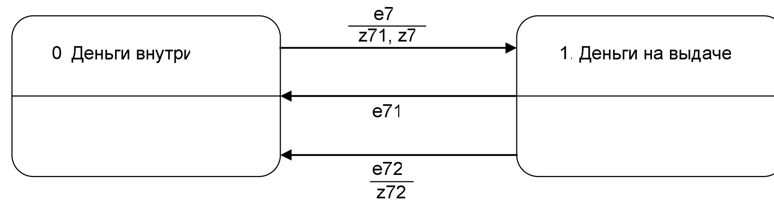


Рис. 7. Граф переходов автомата "Местонахождение денег" (A1)

1.1.4.7. Автомат "Местонахождение карты" (A2)

Автомат управляет местонахождением карты. При поступлении события *e8* "Вернуть карту" банкомат должен выдвинуть карту наружу и, если она не будет взята клиентом по истечении времени, заданного таймером (двадцать секунд), она должна быть втянута в банкомат.

Схема связи этого автомата изображена на рис. 8, а граф переходов – на рис. 9.

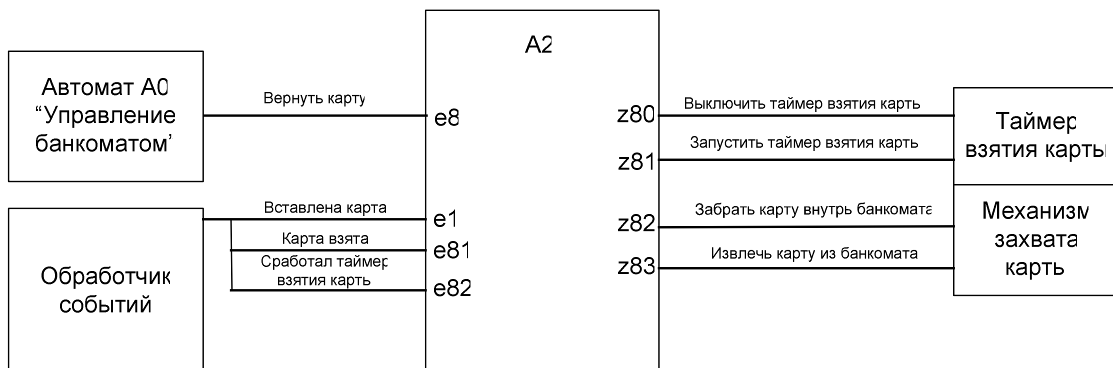


Рис. 8. переходов автомата "Местонахождение карты" (A2)

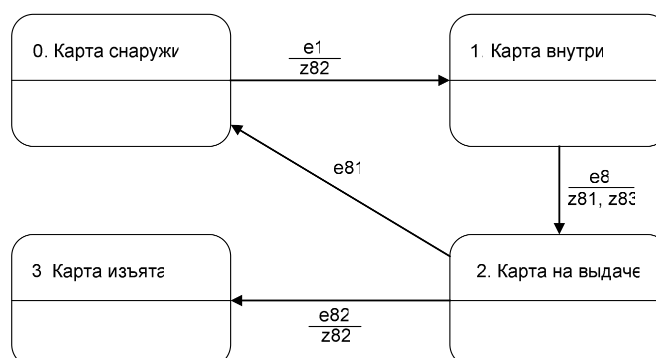


Рис. 9. Граф переходов автомата "Местонахождение карты" (A2)

1.2. *UNIMOD*-БАНКОМАТ

UniMod-банкомат реализован на языке *Java* с применением инструментального средства автоматного программирования *UniMod* [5, 4]. Архитектура банкомата основана на объектно-ориентированном программировании с явным выделением состояний [18].

Основные требования к банкомату:

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

1.2.1. Пользовательский интерфейс

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

Первое, что требуется от пользователя – вставить карту. В этом банкомате этот процесс эмулируется вводом номера карты. В реальном банкомате номер карты считывается с неё автоматически.

Далее пользователь вводит свой личный *pin*-код. Если на сервере не найдется 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, а карта возвращается на руки пользователю.
- "Баланс" – отображает текущий остаток на счете, выводя его на экран и предоставляя возможность распечатать на чеке.
- "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 карты ничего не снимется, и клиент выводит соответствующее сообщение на экран.

После возврата карты, пользователь может вставить её опять либо уйти, нажав кнопку "Выход".

Также как в реальном банкомате операции общения с сервером обеспечиваются по сети. Подавтомат *AServer* удаленно вызывает требуемые методы сервера. При возникновении события «Ошибка при работе с сервером» клиент осуществляет возврат карты и переходит в начальное состояние.

На рис. 10 и рис. 11 приведен внешний вид модели банкомата в некоторых диалогах.

Для управления операциями введем следующие кнопки:

- кнопка *«Выход»* – завершение работы с банкоматом;
- четыре кнопки управления по краям лицевой панели банкомата.

Кроме того, на панели могут появляться кнопки эмуляции действий:

- кнопка "*Вставить карту*" – эмулирует ввод карты пользователем;
- кнопка "*Забрать карту*" – эмулирует изъятие карты;
- кнопка "*Взять чек*" – эмулирует получение чека пользователем;
- кнопка "*Взять деньги*" – эмулирует получение денег из банкомата.

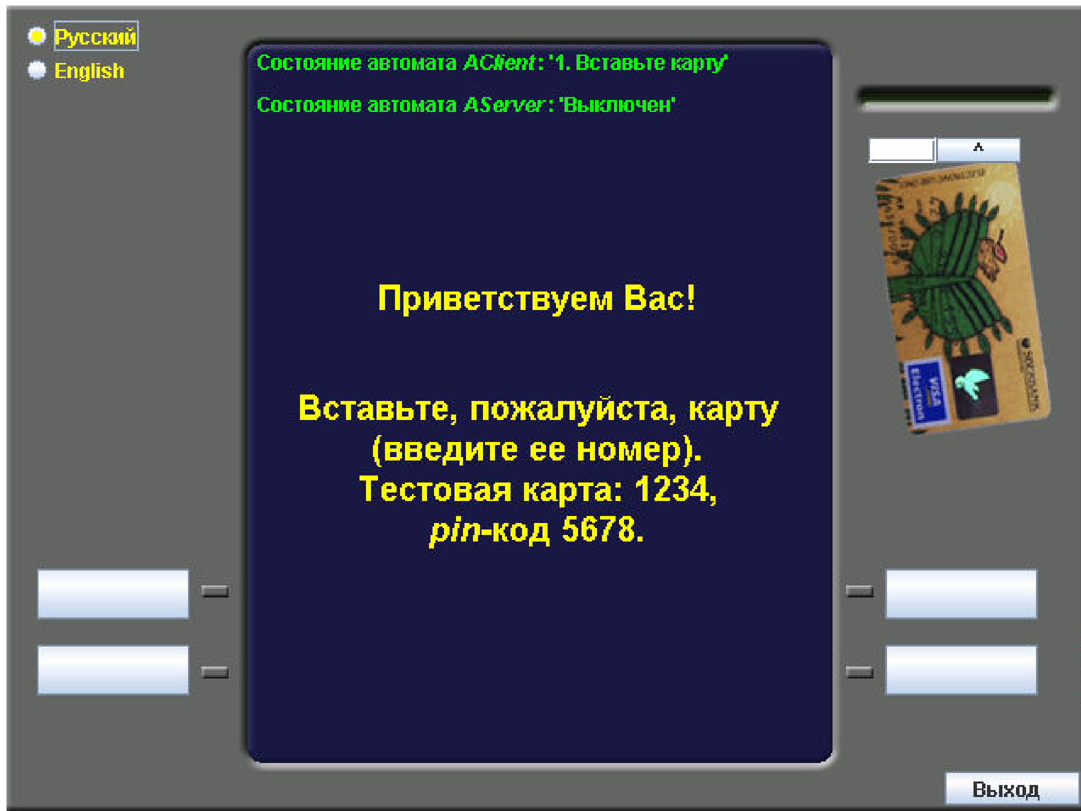


Рис. 10. Внешний вид банкомата. Начальный диалог

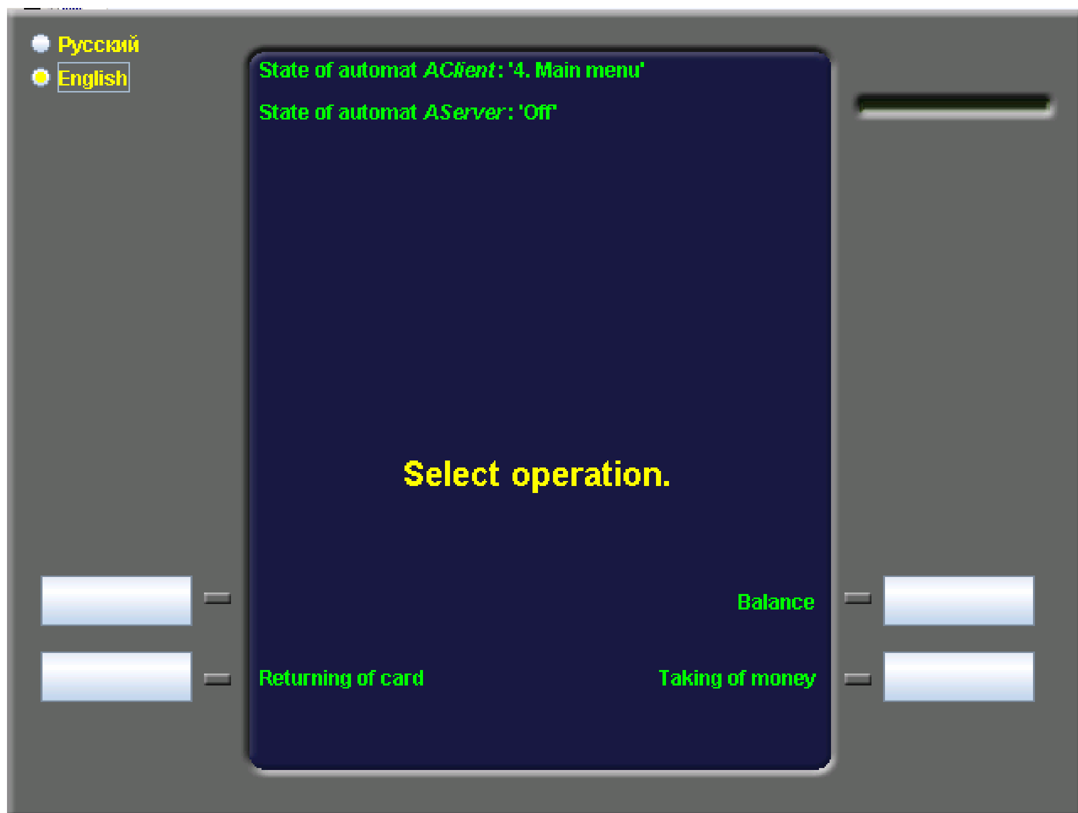


Рис. 11. Внешний вид банкомата. Диалог выбора операции над картой.

1.2.2. Поставщики событий и объекты управления

При формализации работы банкомата используются четыре поставщика событий:

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

В процессе работы банкомат взаимодействует с тремя объектами управления.

1. *FormPainter* – визуализация работы;
2. *ServerQuery* – отправляет запросы на сервер;
3. *ServerReply* – отвечает на клиентские запросы.

1.2.3. Автоматы

Данный проект 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*.

На рис. 12 изображена схема связей автоматов *AClient* и *AServer*.

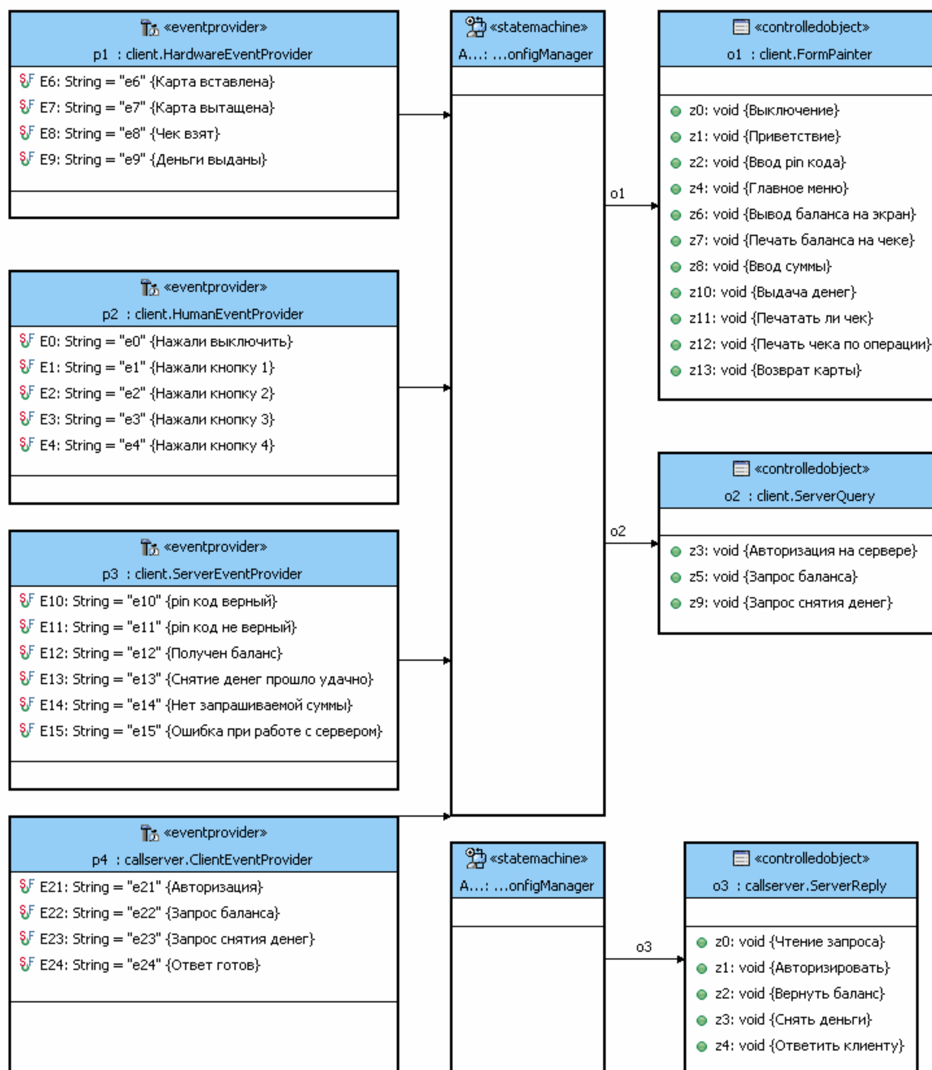


Рис. 12. Схема связей. Несмотря на отсутствие связи между автоматами, *AServer* вложен в *AClient*

На рис. 13 приведен граф переходов автомата *AClient*.

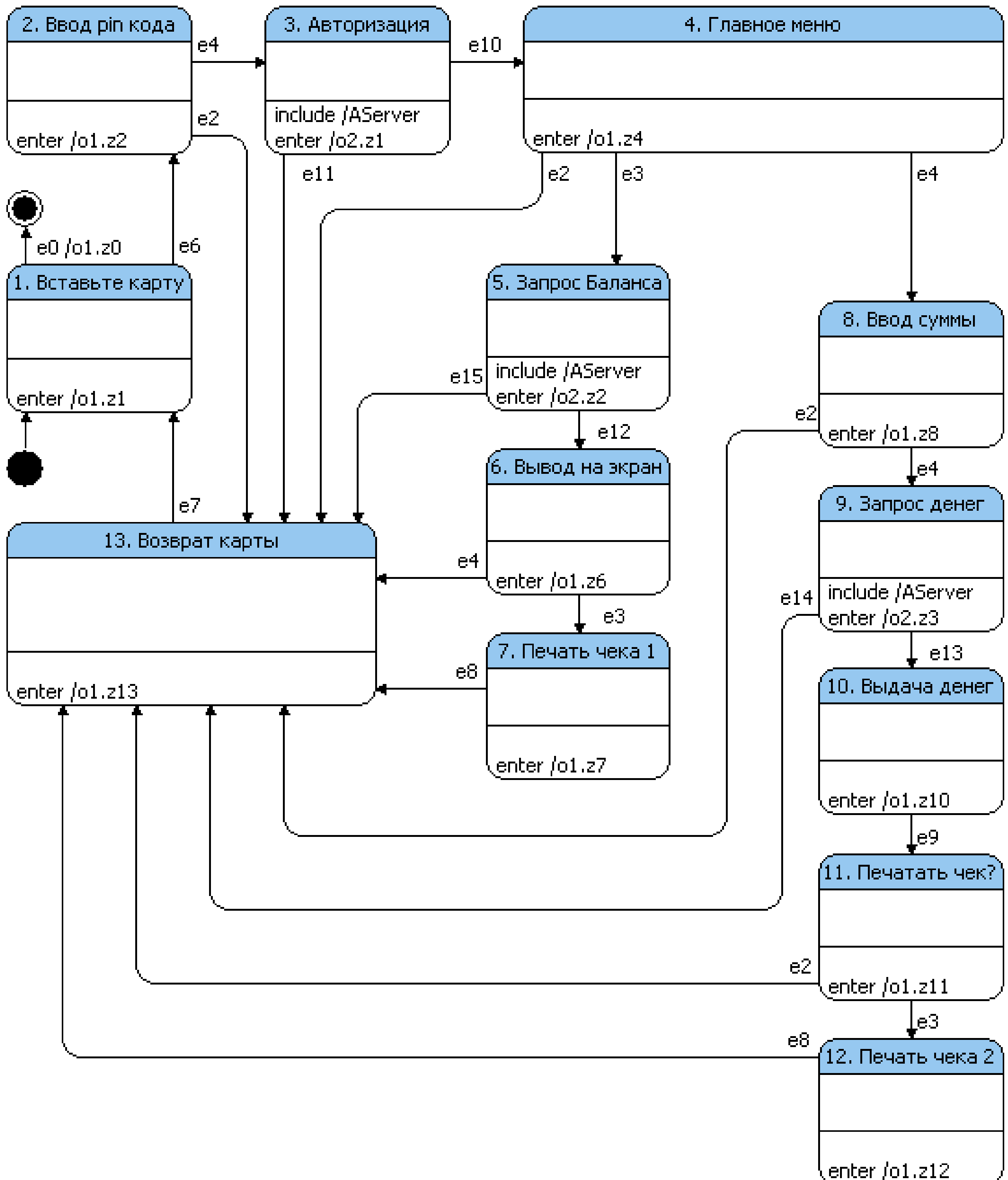


Рис. 13. Автомат *AClient*.

На рис. 14 приведен граф переходов автомата *AServer*, посылающего запросы на сервер.

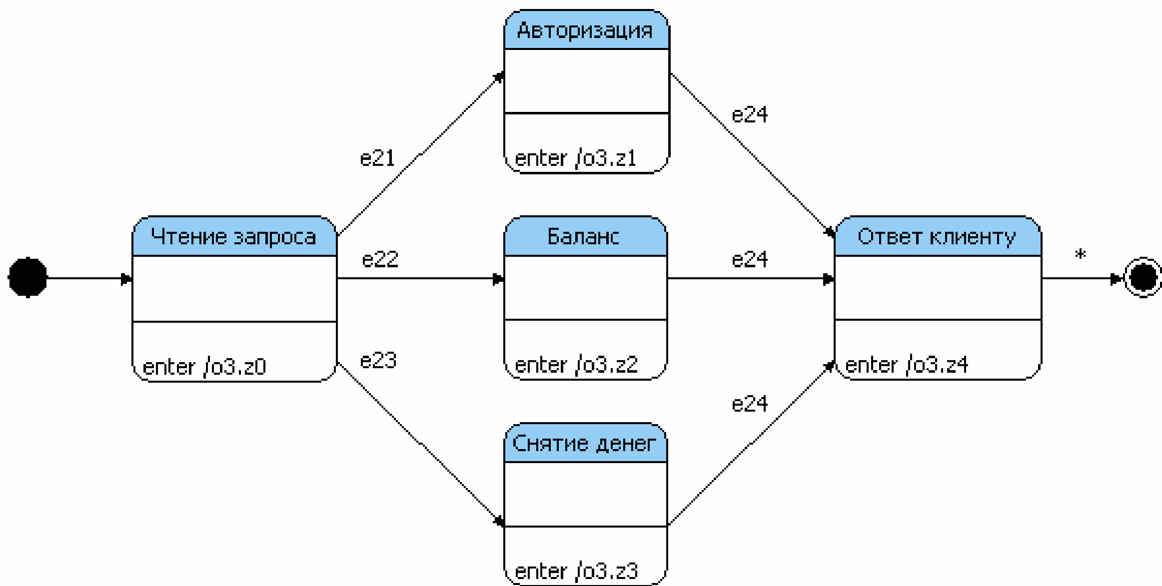


Рис. 14. Автомат *AServer*.

1.3. Верифицируемые свойства

Для апробации методов верификации автоматных программ требуется сформулировать свойства, которые будут верифицироваться для моделей банкоматов. При этом для полноты проверки необходимо подготовить свойства двух типов:

- истинные свойства – позволят выявить ошибки второго рода;
- ложные свойства – позволят выявить ошибки первого рода и проверить методы восстановления контрпримеров из модели Крипке в автоматную модель.

1.3.1. Истинные свойства

Данные свойства должны выполняться для обеих моделей банкоматов.

1. Банкомат выдает деньги только после авторизации. Проверяется отсутствие последовательности действий, при которой пользователь получает деньги до ввода правильного PIN-кода.

2. Банкомат выдает деньги только при вставленной карте. Проверяется отсутствие последовательности действий, при которой пользователь получает деньги до вставки карты в банкомат или после ее изъятия.

3. Банкомат выдает деньги только при указании суммы. Проверяется отсутствие последовательности действий, при которой пользователь получает деньги до указания снимаемой суммы.

4. Баланс выдается только после авторизации. Проверяется отсутствие последовательности действий, при которой пользователю выдается информация о балансе до ввода правильного PIN-кода.

5. Выдаваемая сумма не превышает остаток на счете. Проверяется отсутствие последовательности действий, при которой пользователь получает сумму денег большую, чем имеющаяся на счете.

6. Выдача достаточной суммы денег. Проверяется, что если пользователь вставил карту, набрал верный PIN-код и указал сумму денег, имеющуюся в наличие, то она будет выдана.

7. Банкомат не захватывает карту. Проверяется, что банкомат всегда возвращает карту после выдачи денег или баланса.

8. *Банкомат выдает карту по требованию.* Проверяется, что пользователь в любой момент может изъять вставленную карту из банкомата.
9. *Изъятие карты по таймеру.* Проверяется, что банкомат может изъять карту только, когда будет получен сигнал от таймера изъятия карты.

1.3.2. Ложные свойства

Данные свойства должны не выполняться для обеих моделей банкоматов.

10. *Безусловная выдача денег.* Проверяется, что при любой последовательности действий пользователю будут выданы деньги.
11. *Повторный запрос карты.* Проверяется, что банкомат может запросить карту, если она уже вставлена.
12. *Снятие денег после запроса баланса.* Проверяется, что банкомат может разрешить снять деньги после запроса баланса без повторной авторизации.
13. *Возвращение карты без ошибки.* Проверяется, что при любом поведении банкомата пользователь сможет получить свою карту назад до того, как произойдёт ошибка на сервере.
14. *Выход в главное меню.* Проверяется, что в течение всей работы автомата в любой момент есть возможность попасть в главное меню (нарушается при отсутствии связи с сервером).

1.4. Выводы

Разработаны две тестовые модели банкоматов, которые позволят провести апробацию методов верификации автоматных программ. Отметим, что автоматы спроектированы на основе разных подходов: автоматный банкомат — на основе процедурного программирования с явным выделением состояний, а *UniMod*-банкомат — на основе объектно-ориентированного программирования с явным выделением состояний. Кроме того, первый банкомат реализован на языке программирования *Delphi*, а второй — *Java*. Это позволит показать, что рассматриваемые методы имеют достаточную общность и применимы в различных ситуациях.

Для банкоматов были разработаны набор верифицируемых свойств. При этом выделены не только те свойства, которые должны выполняться, но и те, которые должны не выполняться. Это позволит проверить корректность методов верификации и восстановления контрпримеров.

2. ВЕРИФИКАЦИЯ МОДЕЛЕЙ БАНКОМАТОВ

Метод проверки модели (*Model checking*) – это метод автоматической верификации программных систем. При использовании этого подхода строится модель с конечным числом состояний. Свойства модели выражаются на языке темпоральной логики. Модель и свойства подаются на вход программе-верификатору. Верификатор автоматически проверяет свойства модели и в случае не выполнения свойства выдает контрпример, который указывает место, где нарушается свойство модели.

При существующем подходе построение модели осуществляется человеком. Подобный поход имеет недостатки:

- написание модели вручную требует больших затрат времени;
- при построении модели могут быть допущены ошибки, что снижает эффективность проверки.

Возможность проверки программы, а не абстрактной модели позволила бы избежать этих проблем.

2.1. ВЕРИФИКАЦИЯ ПРИ ПОМОЩИ *FSM VERIFIER*

2.1.1. Описание метода

2.1.1.1. Построение модели системы автоматов

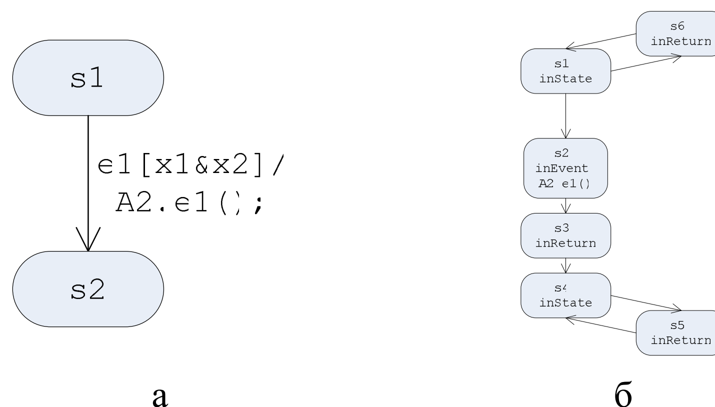
Задача построения модели системы автоматов можно разделить на следующие этапы:

- выделение дополнительных состояний в автомате;
- запись каждого автомата набором переменных и переходов между ними;
- объединение моделей автоматов в общую модель.

Рассмотрим произвольный автомат из системы автоматов. Выделим в автомате, помимо основных состояний, множество его промежуточных состояний, в которых автомат пребывает во время перехода из одного основного состояния в другое. Промежуточное состояние автомата фиксируется каждый раз, когда автомат совершит одно из следующих действий:

- проверит условие на переходе;
- произведет некоторое выходное воздействие;
- вызовет другой автомат;
- вернет управление в вызвавший его автомат.

Рассмотрим выделение промежуточных состояний на примере. На рис. 15, а изображен переход автомата из состояния `s1` в `s2` при возникновении события `e1` при условии, что входные переменные `x1` и `x2` равны единице. В процессе перехода выполняется вызов автомата `A2` с событием `e1`.



а б

Рис. 15. Выделение промежуточных состояний на переходе
Исходный переход (а), преобразованный переход (б)

На рис. 15, б показан переход автомата из состояния `s1` в состояние `s4`. В нем выделены следующие *промежуточные состояния*, расположенные вертикально:

- `s1` – автомат находится в состоянии `s1`;
- `s2` – в этом *состоянии модели* автомат вызывает автомат A2 с событием `e1`;
- `s3` – переходит в основное состояние `s2`;
- `s5` – автомат находится в состоянии `s2`;

Автомат может не удовлетворять условию полноты. Поэтому возможно, что никакое из условий на переходах не выполнится и автомат остается в текущем состоянии. Для моделирования этой ситуации введем для каждого состояния автомата *промежуточное состояние,* которое ведет в него и принадлежит к типу `inReturn`. На рис. 15, б такими состояниями являются `s5` и `s6`.

2.1.1.2. Требования — формулы темпоральной логики на модели Крипке

Для записи требований используются формулы темпоральной логики *ACTL*. Опишем, как записываются некоторые свойства автоматной модели в виде формул *ACTL.*

Для удобства в модель добавлены предикаты, соответствующие свойствам исходной модели. Введем формулы, которые будут описывать состояния автоматов:

- условие того, что автомат `Ak` находится в состоянии `sj` записывается как `yk.sj;`.
- условие того, что выполнилось выходное воздействие `z1`, записывается как `Action_z1;`
- условие того, что произошло событие `ei`, записывается как `A1.ei`.

Для записи формул, описывающих состояния автомата, также можно использовать логические операторы.

Помимо свойств текущего состояния, в условиях можно использовать темпоральные операторы: `AF`, `AG`, `A[U]`. Операторы `AX` не используются для записи свойств автоматов, так как в данной модели один переход автомата соответствует неопределенному числу переходов модели.

2.1.1.3. Преобразование контрпримера в модели Крипке в контрпример в автоматной модели

Если опровергаемая формула принадлежит логике *ACTL*, то получим контрпример в структуре Крипке. Любой контрпример для модели является либо конечным путем, либо путем с конечным началом и циклом. На рис. 16 показан общий вид контрпримера.

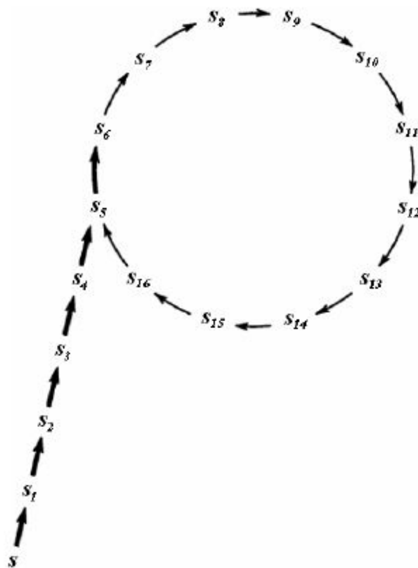


Рис. 16. Общий вид контрпримера

Каждое состояние является набором переменных State_k, OpType_k, Active_k (0 ≤ k ≤ n), xk(0 ≤ k < m). Контрпример к системе автоматов также представляется в виде последовательности состояний, только вместо значений переменных информация предоставляется в терминах состояний и переходов. Для каждого состояния контрпримера выведем информацию об автомате.

2.1.2. Верифицируемая модель банкомата

Произведем верификацию автоматного банкомата (разд. 1.1). Модель этого банкомата представляет собой систему автоматов, записанную в XML-файле. Так как для записи системы используется другая нотация, то необходимо внести следующие изменения в приведенную в указанном разделе систему автоматов:

- заменить названия состояний на s0, s1, s2 …;
- заменить дублирующиеся однотипные переходы одним:
  - из состояния s1 по событиям: e20, e21, e22, e23, e24, e25, e26, e27, e28, e29;
  - из состояния s4 в состояния s5 переходы по событиям: e61, e62, e63, e64, e65, e66, e67;
  - из состояния s6 по событиям: e20, e21, e22, e23, e24, e25, e26, e27, e28, e29;
- заменить в условиях перехода переменные y1, y2 на A1 и A2 соответственно;
- разделить переход из состояния s12 в состояние s4 на два;
- разделить переход из состояния s3 в состояние s11 на два;
- заметить все действия, производимые в состояниях, на действия, производимые на дугах.

При этом автомат на рис. 5 преобразуется в автомат, изображенный на рис. 17.

Эта модель записывается в *XML*-формате:

```
<class name="A0">
  <state name="s0" description="Ожидание карты" info="">
    <transition toState="s1" info="" event="" condition="A2.s1"
        action=""/>
  </state>
  <state name="s2" description="Транзакция прервана" info="">
    <transition toState="s0" info="" event="" condition="A2.s0 | A2.s3"
        action=""/>
  </state>
  <state name="s1" description="Ввод PIN" info="">
    <transition toState="s3" info="" event="e5" condition=""
        action=""/>
    <transition toState="s1" info="" event="e20" condition="x1"
        action="z30"/>
    <transition toState="s1" info="" event="e4" condition=""
        action="z4"/>
  </state>
  <state name="s3" description="Меню" info="">
    <transition toState="s12" info="" event="e65" condition=""
        action=""/>
    <transition toState="s11" info="" event="e66" condition=""
        action="z109, A2.e8"/>
    <transition toState="s11" info="" event="e67" condition=""
        action="z109, A2.e8"/>
  </state>
```

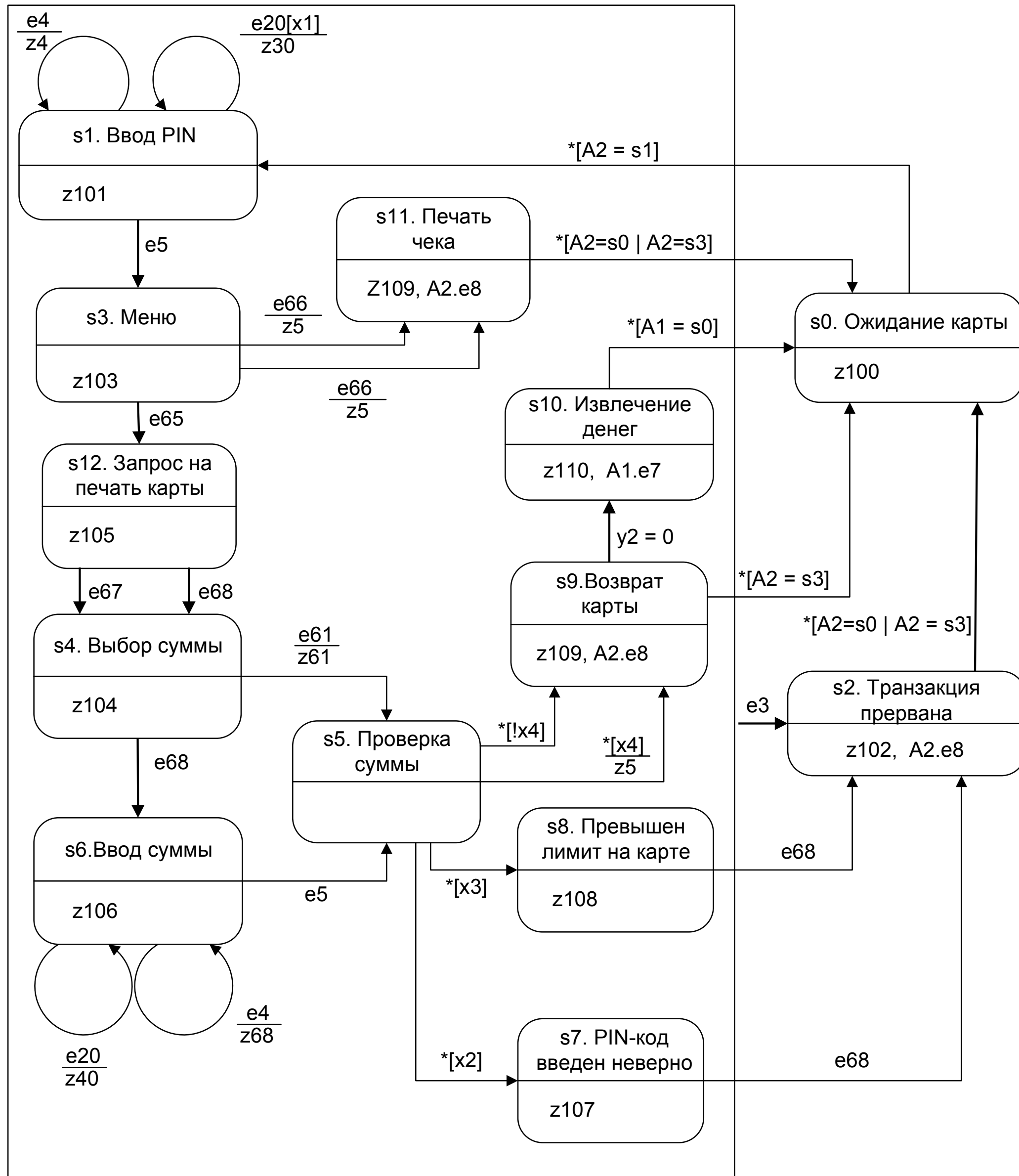


Рис. 17. Модель для автомата A0.

```
<state name="s4" description="Выбор суммы" info="">
  <transition toState="s6" info="" event="e68" condition=""
      action=""/>
  <transition toState="s5" info="" event="e61" condition=""
      action="z61"/>
</state>
<state name="s5" description="Проверка суммы" info="">
  <transition toState="s9" info="" event="" condition="!x4"
      action="z109, A2.e8"/>
  <transition toState="s9" info="" event="" condition="x4"
      action="z5, z109, A2.e8"/>
  <transition toState="s8" info="" event="" condition="x3"
      action=""/>
  <transition toState="s7" info="" event="" condition="x2"
      action=""/>
</state>
<state name="s6" description="Ввод суммы" info="">
  <transition toState="s5" info="" event="e5" condition=""
      action=""/>
```

```
    <transition toState="s6" info="" event="e4" condition=""
        action="z68"/>
    <transition toState="s6" info="" event="e20" condition=""
        action="z40"/>
  </state>
  <state name="s7" description="PIN-код введен неверно" info="">
    <transition toState="s2" info="" event="e68" condition=""
        action=""/>
  </state>
  <state name="s8" description="Превышен лимит на карте" info="">
    <transition toState="s2" info="" event="e68" condition=""
        action=""/>
  </state>
  <state name="s9" description="Возврат карты" info="">
    <transition toState="s10" info="" event="" condition="A2.s0"
        action="z110, A1.e7"/>
    <transition toState="s0" info="" event="" condition="A2.s3"
        action=""/>
  </state>
  <state name="s10" description="Извлечение денег" info="">
    <transition toState="s0" info="" event="" condition="A1.s0"
        action=""/>
  </state>
  <state name="s11" description="Печать чека" info="">
    <transition toState="s0" info="" event="" condition="A2.s0 | A2.s3"
        action=""/>
  </state>
  <state name="s12" description="Запрос на печать карты" info="">
    <transition toState="s4" info="" event="e67" condition=""
        action=""/>
    <transition toState="s4" info="" event="e68" condition=""
        action=""/>
  </state>
  <field name="A1" type="A1"/>
  <field name="A2" type="A2"/>
  <specs/>
</class>
```

Автоматы A1 и A2 можно оставить без изменений.

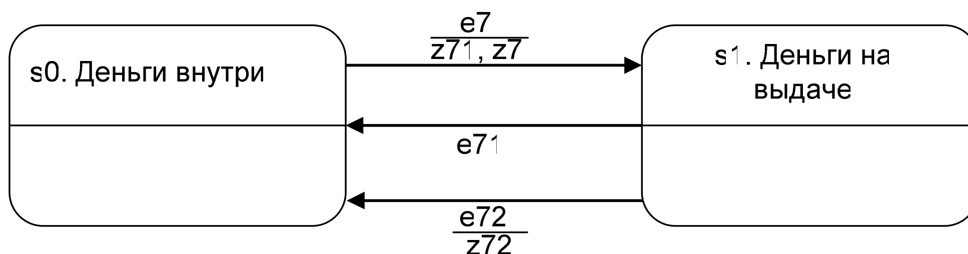


Рис. 18. Модель автомата A1

Модель автомата A1 записывается в XML-формате:

```
<class name="A1">
  <state name="s0" description="Деньги внутри" info="">
    <transition toState="s2" info="" event="e7" when="" action="z71,
         z7"/>
  </state>
  <state name="s1" description="Деньги на выдаче" info="">
    <transition toState="s1" info="" event="e71" when="" action=""/>
    <transition toState="s1" info="" event="e72" when="" action="z72"/>
  </state>
  <specs/>
</class>
```

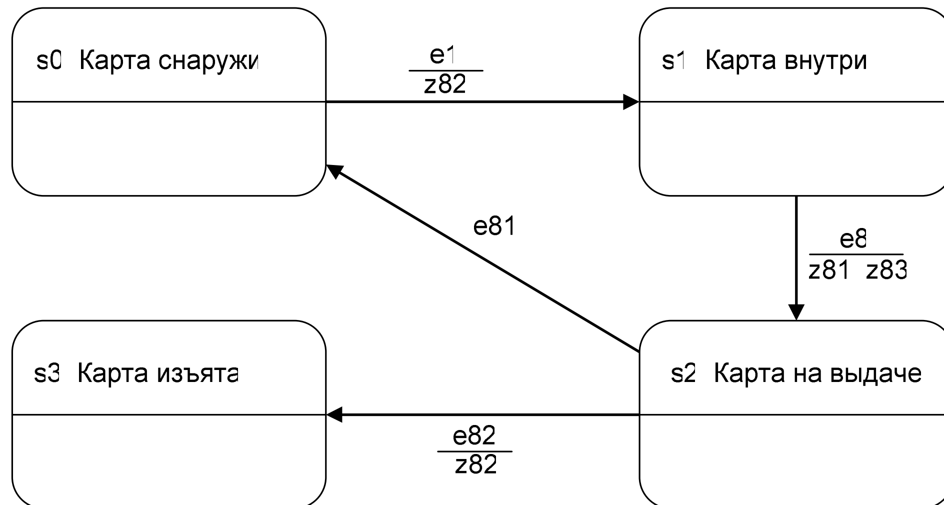


Рис. 19. Модель автомата `A2`

Модель автомата `A2` записывается в *XML*-формате:

```
<class name="A2">
  <state name="s0" description="Карта снаружи" info="">
    <transition toState="s1" info="" event="e1" condition=""
        action="z82"/>
  </state>
  <state name="s1" description="Карта внутри" info="">
    <transition toState="s2" info="" event="e8" condition=""
        action="z81, z83"/>
  </state>
  <state name="s2" description="Карта изъята" info="">
    <transition toState="s0" info="" event="e81" condition=""
        action=""/>
    <transition toState="s3" info="" event="e82" condition=""
        action="z82"/>
  </state>
  <state name="s3" description="Карта на выдаче" info=""/>
  <specs/>
</class>
```

2.1.3. Проверяемые свойства

2.1.3.1. Банкомат выдает деньги только после авторизации

Проверяется отсутствие последовательности действий, при которой пользователь получает деньги до ввода правильного *PIN*-кода (истинное свойство 1).

Сначала необходимо выразить атомарные свойства:

- банкомат выдает деньги, если автомат `A1` перешел в состояние «`1. Деньги на выдаче`». Это свойств выражается формулой $A1 = s1$;
- свойство, что *PIN* -код введен неверно, выражается формулой $x2 = 1$.

Необходимо чтобы пока *PIN* -код не будет введен правильно, $x2 = 0$ автомат `A1` не оказался в состоянии `s1`. Это свойство можно выразить при помощи темпорального оператора **AR** [7]: $\mathbf{A}[(x2 = 0)\mathbf{R}(A1 \neq s1)]$.

Формула для программного средства: `!(!(x2 = 0) EU !(A1!= s1))`.

2.1.3.2. Банкомат выдает деньги только при вставленной карте

Проверяется отсутствие последовательности действий, при которой пользователь получает деньги до введения карты в банкомат или после ее изъятия (истинное свойство 2).

Сначала необходимо выразить атомарные свойства:

- банкомат выдает деньги, если автомат `A1` перешел в состояние «`1. Деньги на выдаче`». Это свойств выражается формулой $A1 = s1$;
- свойство, что на счете недостаточно средств выражается формулой $x3 = 1$;
- свойство, что банкомат находится в начальном состоянии, выражается формулой $A0 = s0$.

Свойство что банкомат не выдаст денег, пока не попадет в начальное состояние, выражается формулой $\mathbf{A}[(A0 = s0)\ \mathbf{R}\ (A1 \mathrel{!=} s1)]$. Окончательно проверяемое свойство выражается формулой $\mathbf{AG}(x3 = 1 \rightarrow \mathbf{A}[(A0 = s0)\ \mathbf{R}\ (A1 \neq s1)])$.

Формула для программного средства: `AG((x3 = 1) ->!(!(A0 = s0) EU !(A1 != s1)))`.

2.1.3.3. Банкомат не захватывает карту.

Проверяется, что банкомат всегда возвращает карту после выдачи денег или баланса (истинное свойство 7).

Сначала необходимо выразить атомарные свойства:

- банкомат выдает деньги, если автомат `A1` перешел в состояние «`1. Деньги на выдаче`». Это свойств выражается формулой $A1 = s1$;
- банкомат печатает чек, если вызывается выходное воздействие $z5$. Это свойство выражается формулой $Action = z5$;
- банкомат вернет карту, если деньги автомат `A2` перейдет в состояние «`2. Карта на выдаче`». Это выражается формулой `A2 = s2`.

Окончательная формула будет иметь вид $\mathbf{AG}((A1 = s1 \vee Action = z5) \rightarrow \mathbf{AF}(A2 = s2))$.

Формула для программного средства: `AG((A1 = s1 | Action = z1) -> AF(A2 = s2))`.

2.1.3.4. Изъятие карты по таймеру

Проверяется, что банкомат может изъять карту только, когда будет получен сигнал от таймера изъятия карты (истинное свойство 9).

В терминах модели данное свойство формулируется следующим образом «до тех пор пока не придет событие `e82` от таймера банкомат не вызовет выходное воздействие `z82` для изъятия карты».

Сначала необходимо выразить атомарные свойства:

- свойство «Вызвано событие `e82`» выражается формулой: $Event = e82$;
- свойство «Произошло выходное воздействие `z82`» выражается формулой: $Action = z82$.

Необходимо чтобы предикат $Action = z82$ не выполнялся до тех пор, пока не выполнится $Event = e82$. Это свойство выражается оператором **AR**.

Окончательная формула $\mathbf{A}[(Event = e82)\ \mathbf{R}\ (Action = z82)]$.

Формула для программного средства: `!((Event!= e82) EU (A2 != s3)`.

2.1.3.5. Банкомат выдает деньги только по требованию

Проверяется отсутствие последовательности действий, при которой пользователь получает деньги до выбора в меню соответствующего пункта (аналог истинного свойства 3).

В терминах модели свойство формулируется следующим образом: Пока автомат A0 в состоянии «`3. Меню`» не будет вызвано событие `e65`, автомат `A1` не перейдет в состояние `s1`.

Сначала необходимо выразить атомарные свойства:

- свойство «Автомат `A0` находится в состоянии `s3`»,выражается формулой $A0 = s3$;
- свойство «Вызвано событие `e65`» выражается формулой: $Event = e65$;
- свойство «Автомат `A1` находится в состоянии `s1`» выражается формулой $A1 = s1$.

- свойство «Событие `e65` вызывается в состоянии «меню» выражается формулой $A0 = s3 \wedge Event = e65$.

Окончательное свойство выражается при помощи оператора **AR**.

Окончательная формула: $\mathbf{A}[(A0 = s3 \wedge Event = e65)\ \mathbf{R}\ (A1 = s1)]$.

Формула для программного средства: `!(!(A0=s3 & Event = e65) EU !(A1=s1))`.

2.1.3.6. Баланс выдается только после авторизации

Проверяется отсутствие последовательности действий, при которой пользователю выдается информация о балансе до ввода правильного *PIN*-кода (истинное свойство 4).

В терминах модели данное свойство выражается следующим образом «Не будет вызвано выходное воздействие `z2`, пока предикат `x2` не станет ложным». Сначала необходимо выразить атомарные свойства:

- свойство «Не будет вызвано выходное воздействие `z2`» выражается формулой $(Action \neq z2)$;
- свойство «Предикат `x2` станет ложным» выражается формулой $x2 = 0$.

Окончательная формула: $\mathbf{A}[(Action \neq z2)\ \mathbf{U}\ (x2 = 0)]$.

Формула для программного средства: `(Action != z2) AU (x2 = 0)`.

2.1.3.7. После извлечения карты её можно будет вставить снова

В терминах модели это свойство выражается следующим образом «Если банкомат перешел в состояние «`2. Карта на выдаче`», то он всегда перейдет в состояние «`0. Ожидание карты`». Сначала необходимо выразить атомарные свойства:

- свойство «Автомат `A2` находится в состоянии `s2`» выражается формулой $A2 = s2$;
- свойство «Автомат `A0` находится в состоянии `s0`» выражается формулой $A0 = s0$;
- свойство «Если автомат `A2` находится в состоянии `s2`, то автомат `A0` когда-нибудь перейдет в состояние `s0`» выражается формулой: $(A2 = s2) \to \mathbf{AF}(A0 = s0)$. Для того чтобы проверить данное свойство для всех достижимых состояний используется оператор **AG**.

Окончательная формула: $\mathbf{AG}((A2 = s2) \to \mathbf{AF}(A0 = s0))$.

Формула для программного средства: `AG ((A2 = s2) -> AF (A0 = s0))`.

2.1.3.8. Транзакция будет завершена в случае ошибки

В терминах модели это свойство выражается следующим образом «Если банкомат не находится в состояниях `s0`, и `s2` и при этом возникает событие `e2`, то банкомат перейдет в состояние `s2`». Сначала необходимо выразить атомарные свойства:

- свойство «Автомат `A0` не находится в состоянии `s0`» выражается формулой $A0 \neq s0$;
- свойство «Автомат `A0` не находится в состоянии `s2`» выражается формулой $A0 \neq s2$;
- свойство «Возникает событие `e2`» выражается формулой: $Event = e2$.

Для того чтобы указать, что автомат не находится ни в одном из состояний `s0`, `s2` записывается формула: $(A0 \neq s0) \wedge (A0 \neq s2)$. Свойство, что автомат `A0` перейдет в состояние `s2`, выражается формулой $\mathbf{AF}(A0 = s2)$. Для того, чтобы проверить формулу для каждого достижимого состояния используется оператор **AG**.

Окончательная формула: $\mathbf{AG}((A0 \neq s0) \wedge (A0 \neq s2) \wedge (Event = e2) \to \mathbf{AF}(A0 = s2))$.

Формула для программного средства: `AG((A0!=s0) & (A0!= s2) & (Event =e2) -> AF(A0=s2))`.

2.1.3.9. Безусловная выдача денег

Проверяется, что при любой последовательности действий пользователю будут выданы деньги (ложное свойство 1).

В терминах модели это свойство выражается следующим образом «Из любого состояния банкомат переходит в состояние «`2. Деньги на выдаче`». Свойство «Автомат `A1` находится в состоянии `s1`» выражается формулой $A1 = s1$.

Окончательная формула: $\mathbf{AG}(\mathbf{AF}(A1 = s1))$.

Формула для программного средства `AG(AF(A1=s1))`.

Контрпример выделен на рис. 20 красным цветом.

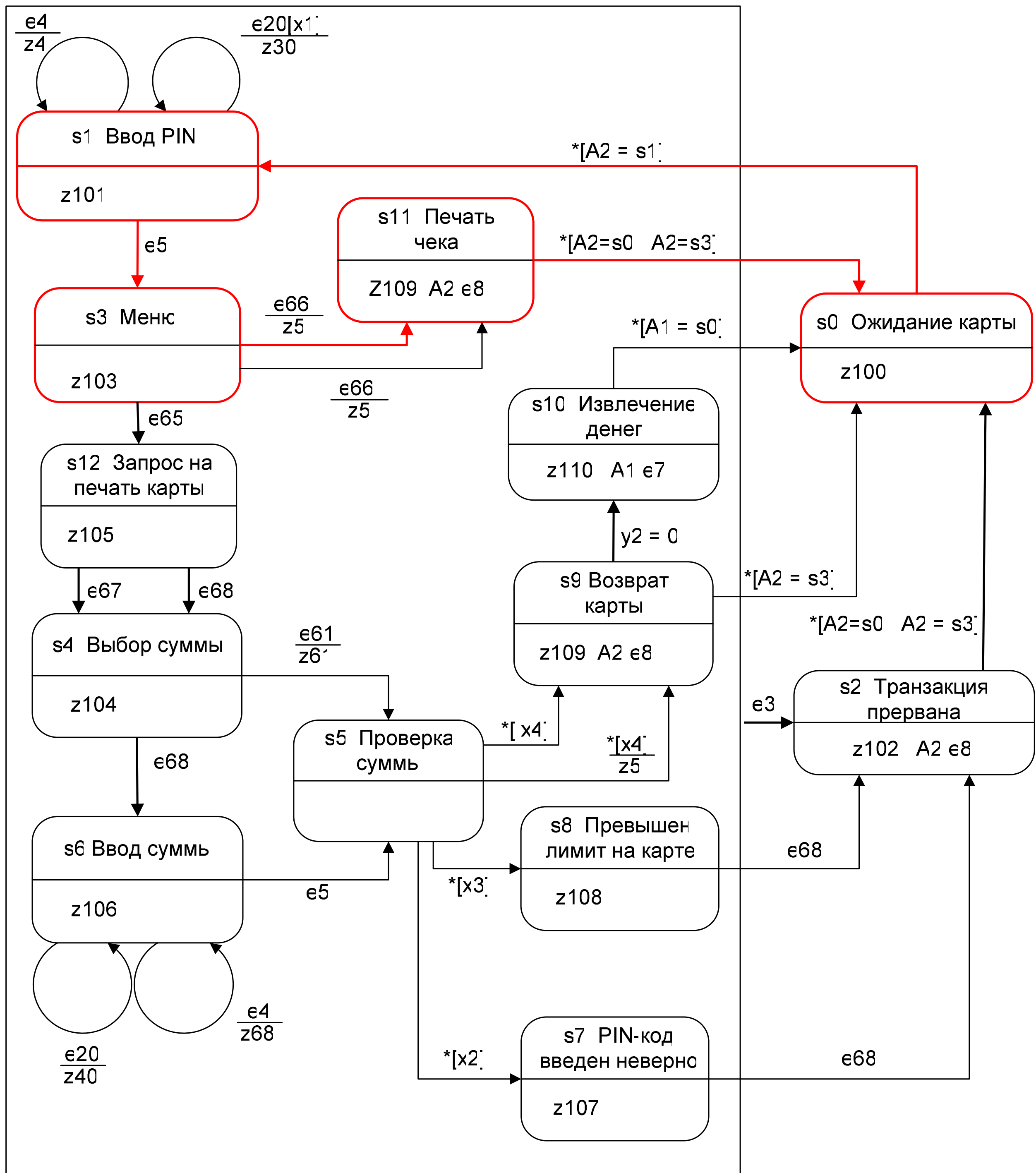


Рис. 20. Сценарий, при котором пользователь никогда не получает денег

Этот случай не является содержательным. Поэтому модифицируем условие: «Банкомат может оставаться в состоянии «`1. Ввод ПИН`», только если поступают события `e4` или `e20`».

Модифицированная формула должна учитывать все состояния, кроме тех, в которых происходят события `e4` или `e20`. Изменим формулу на `AG(AF(A1=s1 | A0=e4 | A0=e20))`

После этого верификатор возвращает контрпример, показанный на рис. 21. При этом изменяются так же состояния автомата `A2`.

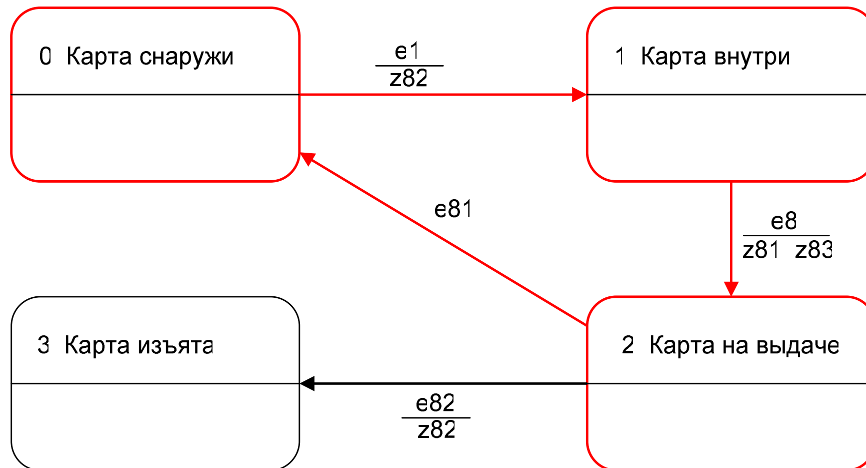


Рис. 21. Последовательность переходов автомата A2

Таким образом, был получен ожидаемый контрпример.

2.1.3.10. Снятие денег после запроса баланса

Проверяется, что банкомат может разрешить снять деньги после запроса баланса без повторной авторизации (ложное свойство 3).

В терминах модели это свойство выражается следующим образом «После того как автомат `A0` перешел в состояние «`11. Печать чека`», автомат `A1` может перейти в состояние `s1`, такми образом, чтобы выходное воздействие `z81` не выполнилось». Сначала необходимо выразить атомарные свойства:

- свойство «Автомат `A0` находится в состоянии `s11`» выражается формулой $A0 = s11$;
- свойство «Автомат `A0` находится в состоянии `s0`» выражается формулой $A0 = s0$;
- свойство «Автомат `A1` не находится в состоянии `s1`» выражается формулой $A1 \neq s1$;
- свойство «Вызывается выходное воздействие `z81`» выражается формулой $Action = z81$.

Окончательная формула: $\mathbf{AG}(A0 = s11 \to \mathbf{E}[A1 != \mathrm{s1}\ \mathbf{U}\ A0 = s0 \wedge Action = z81])$.

Формула для программного средства: `AG ((A0 = s11) -> (A1 != s1 EU A0 = s0 && Action = z81))`.

В результате верификации обнаруживается контрпример, изображенный на рис. 22. Он выделен красным цветом. При этом не показан контрпример для подформулы `E[!(A0 = s0) U A1 = s1]`, так как данная подформула означает существование пути в состояние «`1. Деньги на выдаче`» автомата A1, который не проходит через состояние s0. Можно убедиться, что такого пути не существует.

2.1.4. Выводы

При проверке были получены следующие результаты:

- не было ложных срабатываний. Все контрпримеры, которые возвращала программа, присутствовали в модели;
- при проверке было проверено несколько заранее неверных свойств. При этом ошибки, которые ожидались, были найдены;
- при проверке первого свойства построенная модель оказалась неточной. Этот недочет легко был обнаружен и откорректирован.

Из изложенного следует, что программа корректно реализует алгоритм верификации, а также то, что рассмотренный метод пригоден для проверки свойств автоматных программ.

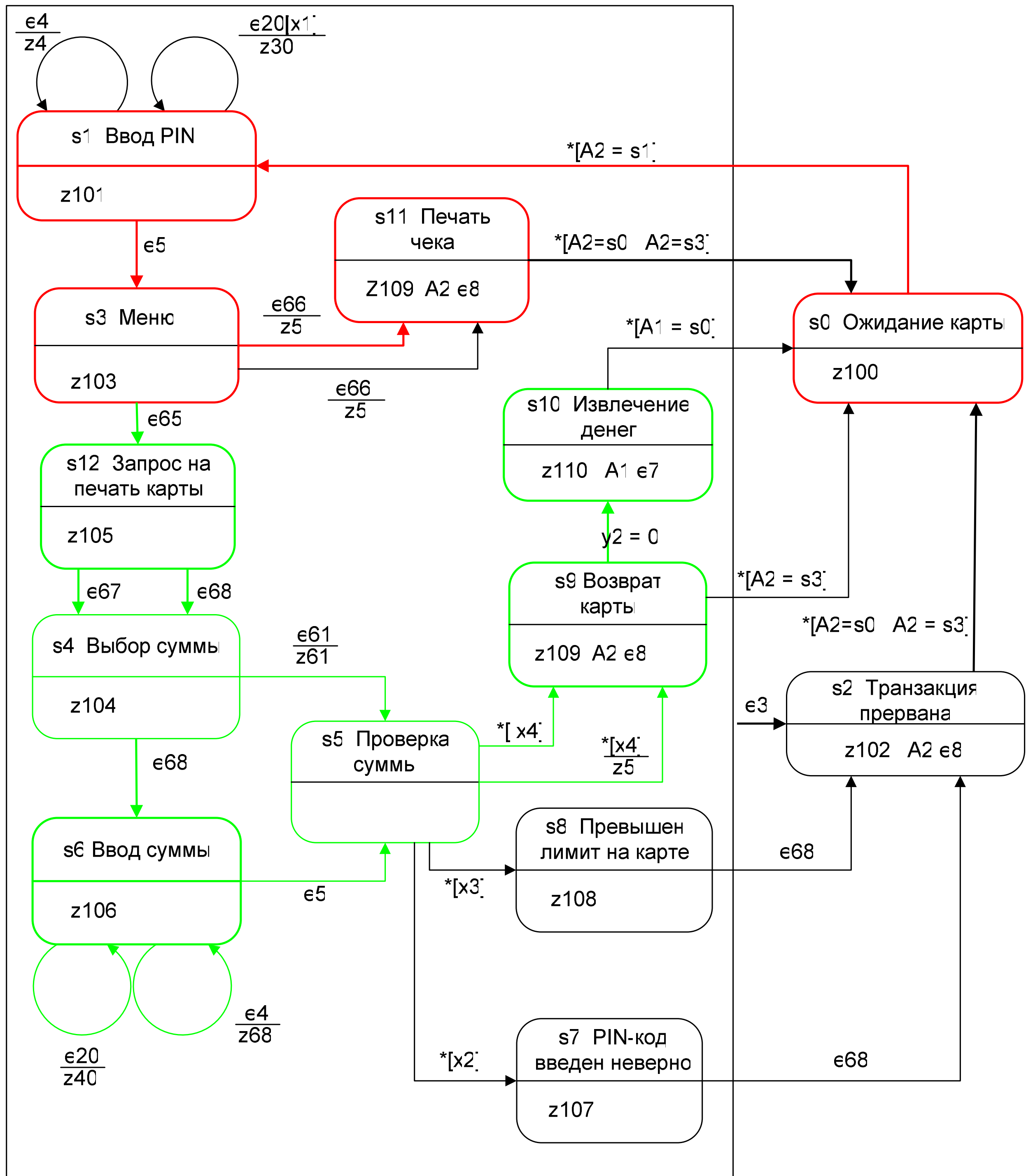


Рис. 22. Контрпример к автомату A0

2.2. Верификация при помощи *CTLVerifier*

2.2.1. Описание метода

2.2.1.1. Построение модели Крипке

В этом разделе описывается метод генерации множества атомарных предложений автоматной программы и преобразования автомата с булевыми входными переменными в модель

Крипке. Метод заключается в редукции полного графа переходов с внесением тесных отрицаний внутрь атомарной формулы. Приводится пример записи требований к программе. Требования выражаются в темпоральной логике *CTL*.

Моделью Крипке (также *CTL*-моделью) для данного множества атомарных предложений AP будем считать тройку $\mathcal{M} = (S, R, Label)$, где

- S — непустое множество состояний;
- $\rightarrow \subseteq S \times S$ — тотальное отношение на *S*, называемое отношением переходов. Для него должна выполняться формула $\forall s \in S\ \exists s' \in S \mid (s, s') \in \rightarrow$ (свойство тотальности). Оно сопоставляет каждому элементу (состоянию) $s \in S$ *непустое* множество его состояний-последователей.
- $Label \subseteq S \times AP$ — помечающее отношение, сопоставляющее каждому состоянию $s \in S$ множество атомарных предложений, истинных в s.

Иногда можно потребовать, чтобы в модели Крипке было задано непустое множество начальных состояний $S_0 \subseteq S$ или даже одно начальное состояние $s \in S$.

Кратко опишем преобразование автомата в модель Крипке по рассматриваемому методу. Альтернативные способы получения модели Крипке из автоматной модели описаны в работах [27–32].

Для автомата наиболее точной была бы модель, в которой для каждого события сформирована полная система переходов. Полученная модель имела бы большой размер, но в ней можно проверять любые темпоральные свойства.

В используемом методе (редукции) переходы будут раскладываться на атомарные блоки, которым соответствуют события и выходные воздействия, расположенные друг за другом, а входные воздействия будут специальным образом в них сохранены. Метод редукции строит компактные модели (линейного размера по отношению к размеру автоматной модели) и позволяет проверять практически все свойства, которые можно сформулировать относительно состояний, событий и входных и выходных воздействий. Он является наилучшим по соотношению эффективность-выразительность.

Для удобства в состояния модели Крипке добавляются три «управляющих» атомарных предложения: *InState*, *InEvent*, *InAction* – для состояний модели, построенных, соответственно, из *состояний*, *событий*, *выходных воздействий* исходного автомата. Это сделано для того, чтобы при записи формулы в темпоральной логике можно было различать тип исследуемого состояния.

Для того чтобы при работе с системой автоматов можно было различать, в каком из них находится соответствующая позиция рассматриваемого пути, каждое состояние снабжается дополнительным атомарным предложением, уникальным для каждого автомата.

Множество стартовых состояний модели Крипке состоит из одного элемента – стартового состояния исходного автомата.

Конвертация автомата в модель Крипке и формулировка проверяемых свойств демонстрируется на примере системы, эмулирующей работу банкомата.

2.2.1.2. Построение редуцированного графа переходов

В данном методе множество AP равно {`Y1`, `Y2`, …} ⋃ {`e1`, `e2`, …} ⋃ {`x1`, `x2`, …} ⋃ ⋃ {`!x1`, `!x2`, …} ⋃ {`z1`, `z2`, …} ⋃ {`InState`, `InEvent`, `InAction`} ⋃ *Names*. Здесь *Names* – множество имён всех автоматов системы.

На первом шаге положим множество S равным множеству состояний исходного автомата. Введем пустое исходно множество $Label$ и для каждого состояния `s` добавим в него две пометки: (`s`, `s`) и (`s`, `InState`).

После этого для каждого состояния `s` выполняем следующую операцию. Пусть `s` содержит выходные воздействия $z_{s[1]}$, …, $z_{s[n]}$, которые выполняются при входе в `s`. Добавим в модель n состояний {`r1`, …, `rn`} и n переходов `r1` → `r2`, …, `rn-1` → `rn`, `rn` → `s`. В отношение $Label$

добавим пометки (rk, $z_{s[k]}$), (rk, InAction) для всех k от 1 до n. Далее, когда будем добавлять рёбра в модель, каждому ребру, идущее в состояние s, будем сопоставлять ребро, которое ведет в состояние r1.

Пример такого разбиения проиллюстрирован на рис. 23.

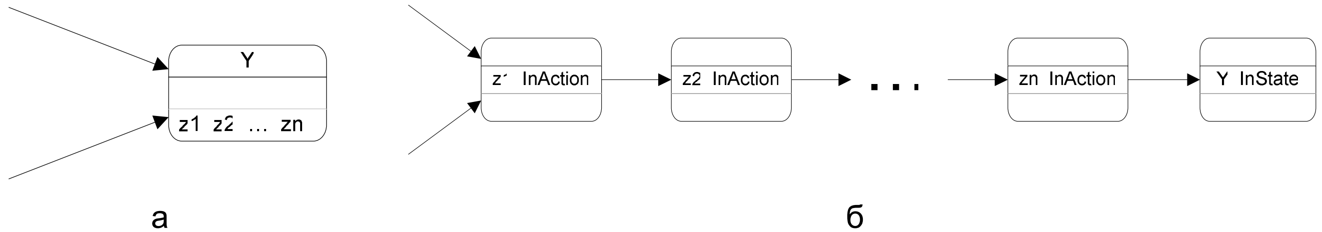


Рис. 23. Состояние с выходными воздействиями до преобразования (а) и после (б)

Пусть L – множество следующих символов: $L = \{$x1, !x1, x2, !x2, x3, !x3, …$\}$. Это множество всех литералов, составленных из входных переменных. Следует различать знаки «¬» и «!». Первый из них означает операцию взятия логического отрицания, а второй интерпретируется просто как символ (часть строки!xi). Пусть L' – некоторое (не обязательно совместное) множество литералов. Множество отрицаний этих литералов обозначим через $!\,L'$.

Тогда для каждого ребра r исходного автомата, ведущего из состояния p в состояние q с пометкой e_i & $y_{j[1]}$ & $y_{j[2]}$ & $y_{j[3]}$ & … $y_{j[m]}$ / $z_{i[1]}, \ldots, z_{i[n]}$, где либо $y_{j[j*]} = x_{j[j*]}$, либо $y_{j[j*]} = !x_{j[j*]}$ (это означает, что $y_{j[j*]}$ является либо входной переменной, либо ее отрицанием), добавим в модель $n+1$ состояние $\{$re, r1, …, rn$\}$, $n+2$ перехода: p $\to$ re, re $\to$ r1, r1 $\to$ r2, …, rn-1 $\to$ rn, rn $\to$ q, а в *Label* добавим пометки (re, e_i), (re, InEvent), (rk, $z_{i[k]}$), (rk, InAction) для всех k от 1 до n, а также пометки (re, u_k), где

$u_k \in L \setminus !\{y_{j[1]}, y_{j[2]}, y_{j[3]}, \ldots y_{j[m]}\} = L \setminus \{!y_{j[1]}, !y_{j[2]}, !\,y_{j[3]}, \ldots, !y_{j[m]}\}$.

Пример такого преобразования приведен на рис. 24.

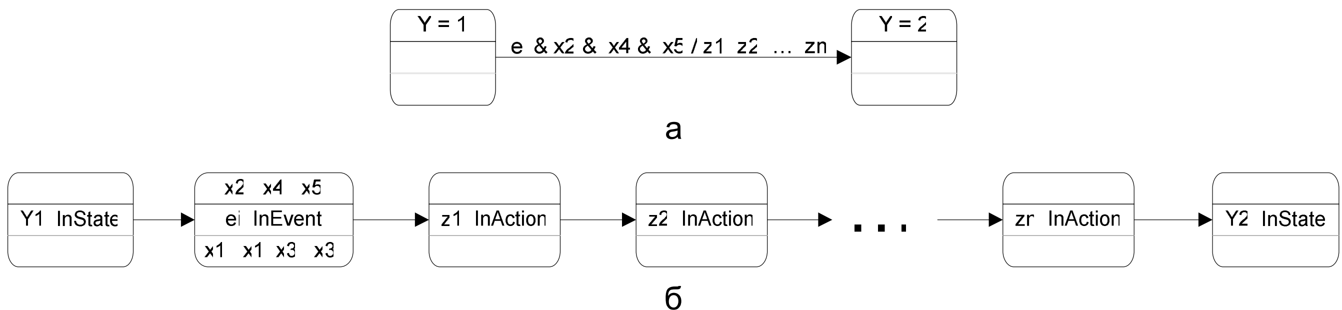


Рис. 24. Переход между состояниями
до преобразования по методу редукции (а) и после него (б)

Таким образом, в множество атомарных предложений на тех состояниях, которые были построены из событий добавлены входные переменные в том виде, в котором они записаны на переходах автомата (вместе с отрицаниями, если они есть). Атомарные предложения каждого такого состояния содержат все литералы, составленные из несущественных входных переменных для данного перехода. Несущественными будем называть те переменные исходного автомата, которые не обозначены на рассматриваемом переходе. Следовательно, на одно и то же состояние добавляются как несущественные переменные, так и их отрицания. С точки зрения синтаксиса и семантики темпоральной логики это допустимо: процесс обработки модели Крипке не предполагает совместность множества атомарных предложений состояния, так как интерпретирует эти предложения как строки. Причина такого обращения с несущественными переменными заключается в следующем: требуется обеспечить, чтобы любая ссылка на несущественную в данном событии переменную, упомянутую в *CTL*-формуле, давала заранее

определённый результат. На рис. 24 существенные переменные изображены в верхнем слое прямоугольника состояния, а несущественные — в нижнем.

Вложенные автоматы считаются частями исходного. Если в состояние s автомата A вложен автомат B, то все дуги, входящие в состояние s в модели A следует перенаправить в стартовое состояние модели B, а у всех дуг, исходящих из состояния s модели A, следует изменить начало на терминальное состояние модели B. После всех этих действий для каждого состояния полученной модели добавим в отношение *Label* атомарное предложение (элемент множества *Names*), соответствующее имени автомата, которому это состояние принадлежит.

Теперь рассмотрим построение и интерпретацию *CTL*-формул для редуцированных моделей.

CTL-семантика в данном методе будет немного отличаться от общепринятой: перед тем, как выполнять верификацию *CTL*-формулы, ее следует привести к определенному («каноническому») виду. Вначале необходимо удалить все парные отрицания (путем замены подформул вида $\neg\neg f$ на f). После этого все входные воздействия, входящие в формулу без отрицания, необходимо предварить двумя отрицаниями: одно из них синтаксическое, а другое логическое (требуется заменить атомарные предложения вида `xi` на ¬`!xi`). При этом только результирующая формула подлежит верификации методами, предназначенными для *CTL*-логики.

Такой метод можно рассматривать как упрощение метода полной системы переходов, описанного в начале раздела, при котором отождествляются двоичные наборы значений несущественных переменных. Использование такой схемы подходит для многих формул.

2.2.1.3. Преобразование сценария для модели Крипке в сценарий для автомата

При использовании нестандартных методов моделирования интерпретировать результаты разработчику приходится самому – он проводит проводить анализ путей прямо на модели *Крипке*, которую он сам (вручную) и построил. При интерактивном моделировании совместно с исполнением и визуализацией автомата [7] процесс представления путей в модели Крипке путями в автомате желательно автоматизировать.

После того, как отработала программа-верификатор, необходимо определить выполнимость формул спецификации на определенных участках автомата. Среди этих участков могут быть состояния, события, выходные воздействия. Сценарий для любой подформулы спецификации представляет собой путь в модели Крипке, иллюстрирующий справедливость или ошибочность данной подформулы. Задача состоит в том, чтобы сценарий, предъявленный программой, был представлен в исходном автомате.

Для описанной в этой главе схемы операция переноса путей из модели Крипке в автомат выполняется однозначно. Действительно, состояния модели, содержащие атомарное предложение `Y=…` или вспомогательное атомарное предложение *InState*, однозначно преобразуются в соответствующие им состояния автомата. Путь же между любыми двумя соседними состояниями всегда представляет собой "змейку" из события и выходных воздействий. Любое из этих "промежуточных" состояний однозначно определяет то главное состояние автомата, из которого эта "змейка" исходит. Из атомарных предложений, которыми помечены её состояния, однозначно восстанавливаются события. Значения существенных входных переменных (тех, которые записаны на переходе) и список несущественных переменных определяются оттуда же. Последовательным проходом по полученному пути восстанавливается информация о выполнении выходных воздействий и их очередности, а также о том, как попасть в данное состояние.

2.2.2. Верификация *CTL*-свойств

В данном разделе описываются примеры свойств, сформулированных для автоматных моделей управляющих программ, которые можно верифицировать методом, описанным в разд. 2.2.1, и результаты верификации для этих свойств. Под результатом верификации понимается множество состояний, в которых формула выполняется, а также набор подтверждающих или

опровергающих трасс для формул и состояний модели. Трасса – это последовательность позиций модели, следующих друг за другом, а позиция – это состояние, событие или выходное воздействие автомата (можно также сказать, что позиция – это состояние модели Крипке). Подтверждающие трассы иллюстрируют справедливость формул, начинающихся с квантора существования, в тех состояниях, в которых они выполняются, а опровергающие трассы иллюстрируют ошибочность формул, начинающихся с квантора всеобщности, в тех состояниях, где они не выполняются.

Рассмотрим пример автоматной модели для системы, эмулирующей работу *UniMod*-банкомата. Система состоит из двух автоматов `AClient` и `AServer`, причём автомат `AServer` вложен в `AClient`. Схема связей для соответствующей автоматной модели изображена на рис. 12, а графы переходов автоматов `AServer` и `AClient` – соответственно на рис. 13 и рис. 14 соответственно.

2.2.2.1. Банкомат выдает деньги только после авторизации

Проверяется отсутствие последовательности действий, при которой пользователь получает деньги до ввода правильного *PIN*-кода (истинное свойство 1). Это свойство можно переформулировать на более низком уровне: «Не существует пути, который начинается в текущей позиции, содержит действие `o1.z10` и до того, как `o1.z10` появится впервые, не содержит события `e10`». В такой формулировке это свойство легко переводится на язык *CTL*: $\neg\mathbf{E}[\neg e10\ \mathbf{U}\ o1.z10]$. Поскольку формула не содержит входных воздействий, её семантику в соответствии с разд. 2.2.1 изменять не требуется.

Можно видеть, что формула верна во всех позициях, затенённых на рис. 25. В незатенённых позициях она не выполняется. Эти позиции формируют контрпример – опровергающую трассу для рассматриваемой формулы. Можно сказать, что эта трасса подтверждает обратную формулу $\mathbf{E}[\neg e10\ \mathbf{U}\ o1.z10]$. Она представляет собой начальный отрезок пути, существование которого утверждается этой формулой. Данная последовательность является контрпримером для позиции, выделенной на рисунке рис. 25 жирным шрифтом.

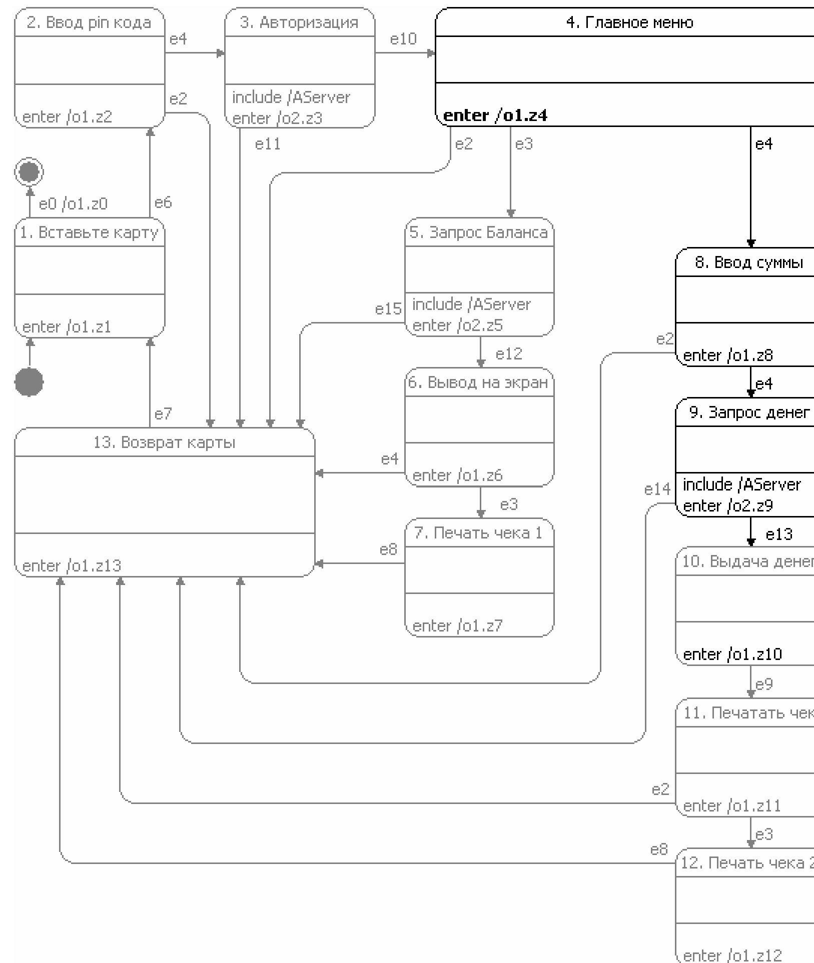


Рис. 25. Невыполняющие позиции для истинного свойства 1

2.2.2.2. Банкомат выдает деньги только при вставленной карте

Проверяется отсутствие последовательности действий, при которой пользователь получает деньги до вставки карты в банкомат или после ее изъятия (истинное свойство 2). Переформулируем его на более низком уровне: «Не существует пути, который содержит действие `o1.z10` и до того, как `o1.z10` появится впервые, не содержит события `e6`». Это же свойство на языке *CTL* имеет вид: $\neg\mathbf{E}[\neg e6\ \mathbf{U}\ o1.z10]$. Проверим эту формулу в состоянии 1 и в позиции `o1.z13` (выходное воздействие, помещённое в состояние 13). Легко видеть, что в обеих позициях она выполняется. Действительно, из состояния 1 любой путь ведёт либо сразу в терминальное состояние (и тогда позиция `o1.z10` никогда не будет достигнута), либо в позицию `e6`, а из позиции `o1.z13` любой путь через состояние 13 и позиции `e7` и `o1.z1` ведёт в состояние 1. Более того, существует только один простой (не проходящий через одно и то же состояние дважды) путь, который ведёт из позиции `o1.z13` в позицию `o1.z10` (он выделен на рис. 26). Этот путь проходит через `e6`.

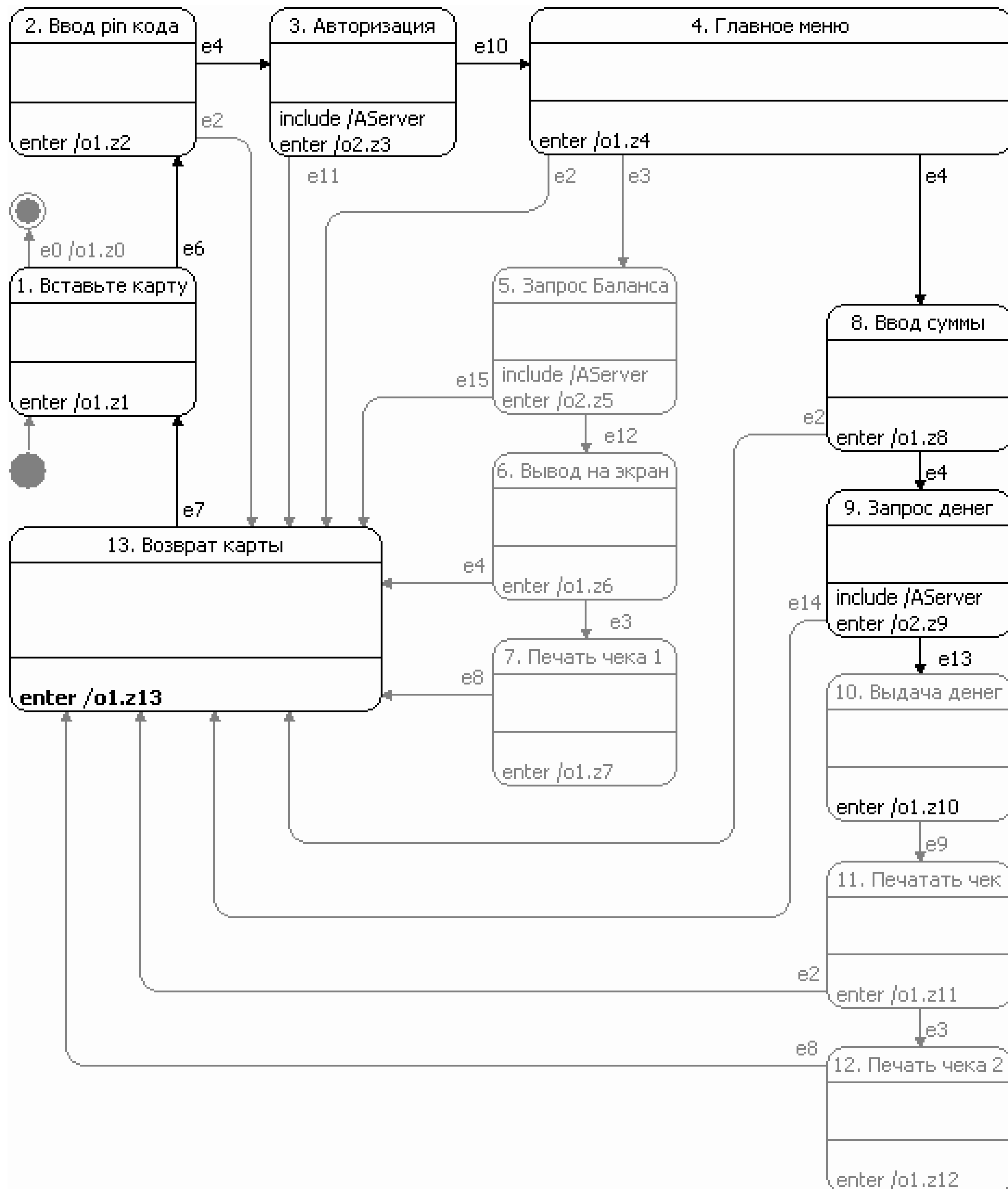


Рис. 26. Путь из позиции `o1.z13` в позицию `o1.z10`

2.2.2.3. Баланс выдается только после авторизации

Проверяется отсутствие последовательности действий, при которой пользователю выдается информация о балансе до ввода правильного *PIN*-кода (истинное свойство 4). Более точно: «Не существует пути, который содержит действие `o1.z7`, и до того, как `o1.z7` появится впервые, не содержит события `e10`». Это же свойство на языке *CTL*: $\neg\mathbf{E}[\neg e10\ \mathbf{U}\ o1.z7]$. Формула верна во всех позициях, затенённых на рис. 27, и только в них. Последовательность позиций, в которых она неверна, образует опровергающую трассу, начинающуюся в позиции, выделенной на рис. 27 жирным шрифтом. Проверим теперь формулу в тех же позициях, что и в п. 2.2.2.2: в состоянии 1 и в позиции `o1.z13`. Формула выполняется. Существует только один простой путь, который ведёт из позиции `o1.z13` в позицию `o1.z7` (он выделен на рис. 28). Этот путь проходит через `e10`.

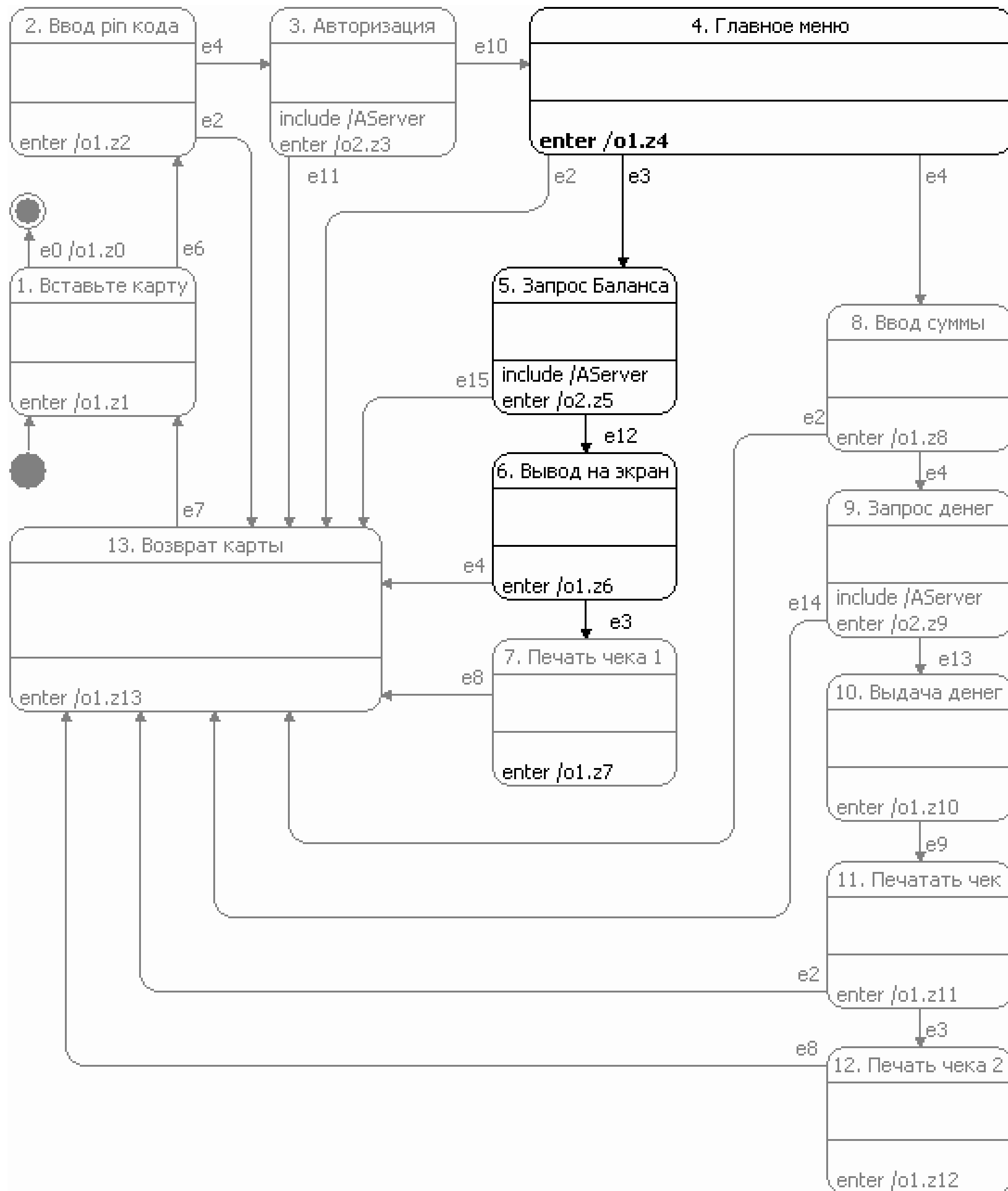


Рис. 27. Невыполняющие позиции для истинного свойства 4

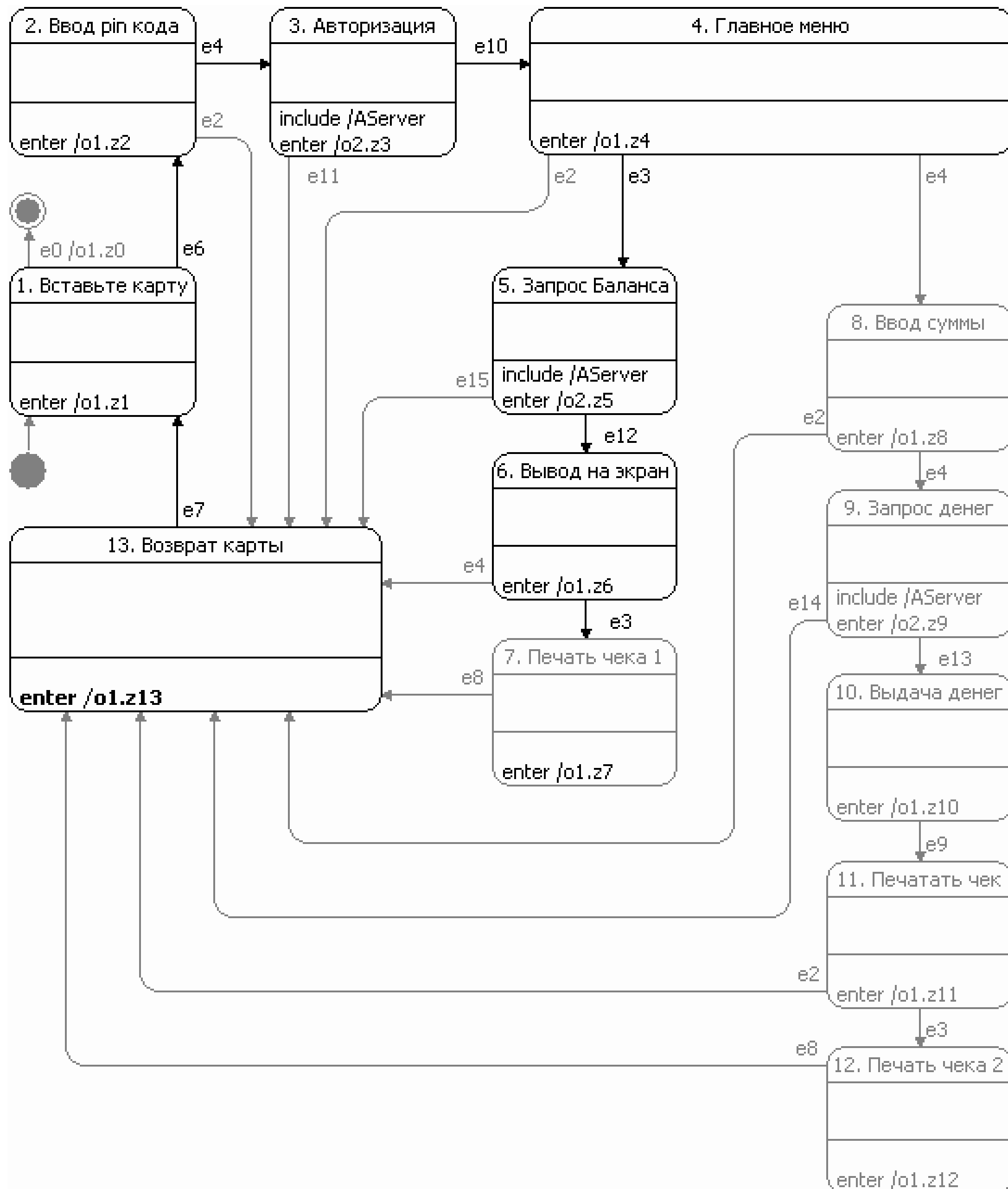


Рис. 28. Путь из позиции `o1.z13` в позицию `o1.z7`

2.2.2.4. Банкомат не захватывает карту

Проверяется, что банкомат всегда возвращает карту после выдачи денег или баланса (истинное свойство 7). Более точно: "Из любой позиции можно попасть в ситуацию, когда пользователю возвращают карту". Это означает, что для каждой фиксированной позиции требуется проверить: существует путь из этой позиции в состояние 13 и в позицию `e7`. На языке *CTL* это свойство записывается следующим образом (для фиксированной позиции): $\mathbf{EF}\,y13 \wedge \mathbf{EF}\,e7$. Проверяем формулу во всех позициях. Формула выполняется (за исключением терминального состояния и перехода в него).

Попробуем теперь изменить модель. Переход из состояния 9 по событию `e14` перенаправим в состояние 9, и переход из состояния 10 по событию `e9` перенаправим в состояние 9. В новой модели будут существовать состояния, из которых уже нельзя будет получить карту из банкомата. Верификатор найдёт позиции, опровергающие формулу, из которых любой путь не достигнет ни состояния 13, ни позиции `e7`. Они выделены на рис. 29.

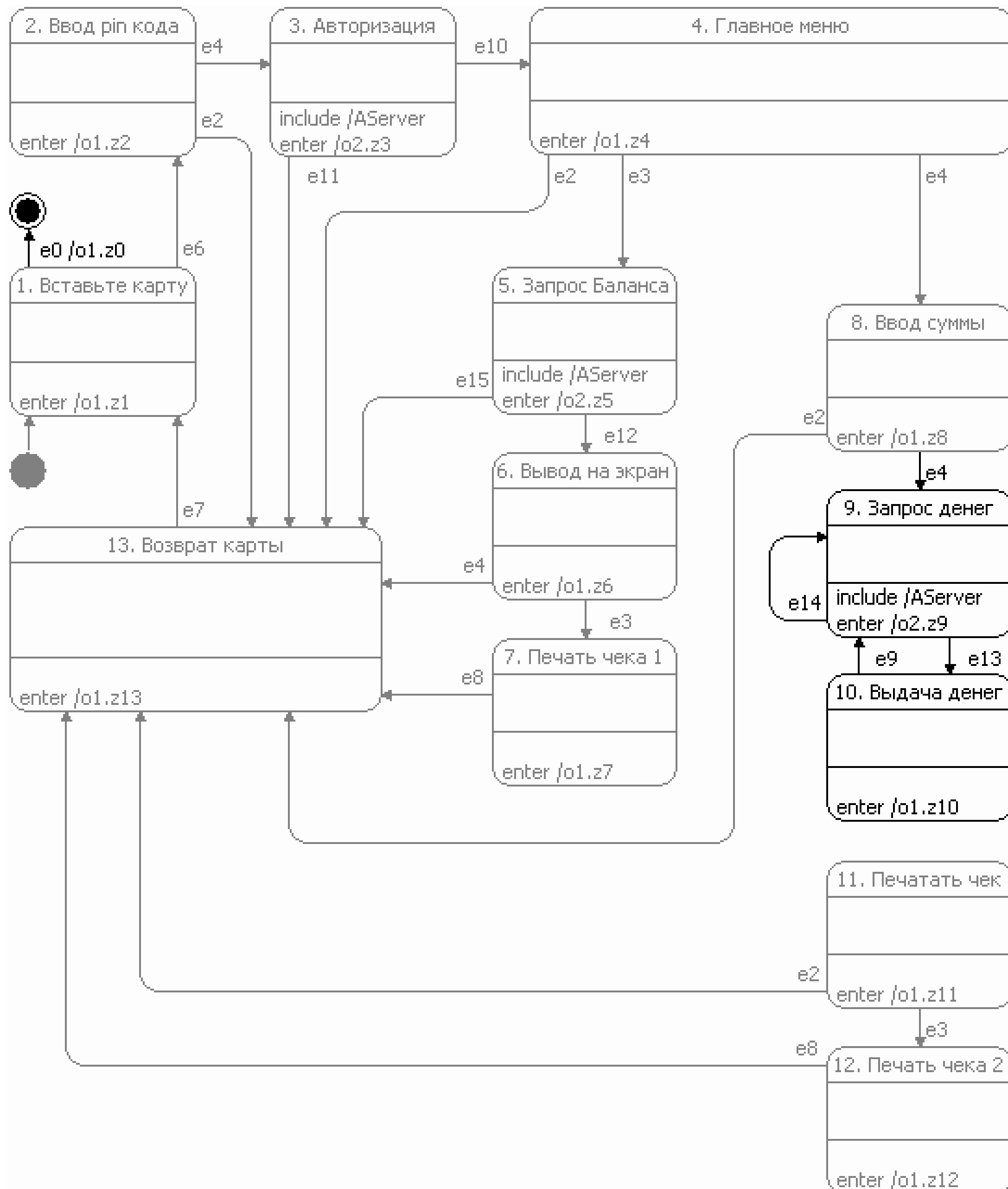


Рис. 29. Невыполняющие позиции в изменённой модели для истинного свойства 7

2.2.2.5. Банкомат выдает карту по требованию

Проверяется, что пользователь в любой момент может изъять вставленную карту из банкомата (истинное свойство 9). Более точно: "Всегда возможно, сделав только один переход между состояниями в автомате `AClient`, попасть в состояние 13". Будем строить соответствующую *CTL*-формулу по шагам. Важно учесть, что один переход между состояниями автомата может содержать много позиций. Однако эти позиции построены не из состояний исходного автомата, а из событий и выходных воздействий. Следовательно, они не помечены атомарным предложением `InState`. Также следует учесть, что в пределах перехода между состояниями автомата `AClient` может выполняться вложенный автомат `AServer`, причём участок модели, соответствующий автомату `AServer`, содержит как события и выходные воздействия, так и состояния исходного автомата `AServer`, которые помечены предложением `InState`. Так как переходы между этими состояниями исполняет сервер и они незаметны для пользователя, целесообразно не учитывать переходы автомата `AServer` в качестве того единственного перехода,

который требуется описать. Рассмотрим следующую формулу пути: $\neg(AClient \wedge InState)\,\mathbf{U}\,y13$. Для фиксированного пути она означает следующее: позиция y=13 обязательно наступит, но до того, как она наступит впервые, никогда не удастся попасть в позицию автомата AClient, которая соответствует его состоянию. Это означает, что первая позиция автомата AClient, которая будет соответствовать его состоянию (а не событию и не выходному воздействию), является состоянием с номером 13. Существование такого пути утверждается в формуле $\mathbf{E}[\neg(AClient \wedge InState)\,\mathbf{U}\,y13]$. Множество позиций, выполняющих эту формулу, выделено на рис. 30. Заметим, что все состояния (которые не являются событиями и выходными воздействиями) автомата AClient, кроме 13-го, не выполняют эту формулу. Причина этого явления в том, что они уже помечены атомарными предложениями AClient и InState. Следовательно, формулой не учитывается тот факт, что требуется сделать ещё один шаг. Усовершенствуем формулу: $\mathbf{EX}\,\mathbf{E}[\neg(AClient \wedge InState)\,\mathbf{U}\,y13]$. Можно видеть (рис. 31), что эта формула выполняется почти во всех состояниях.

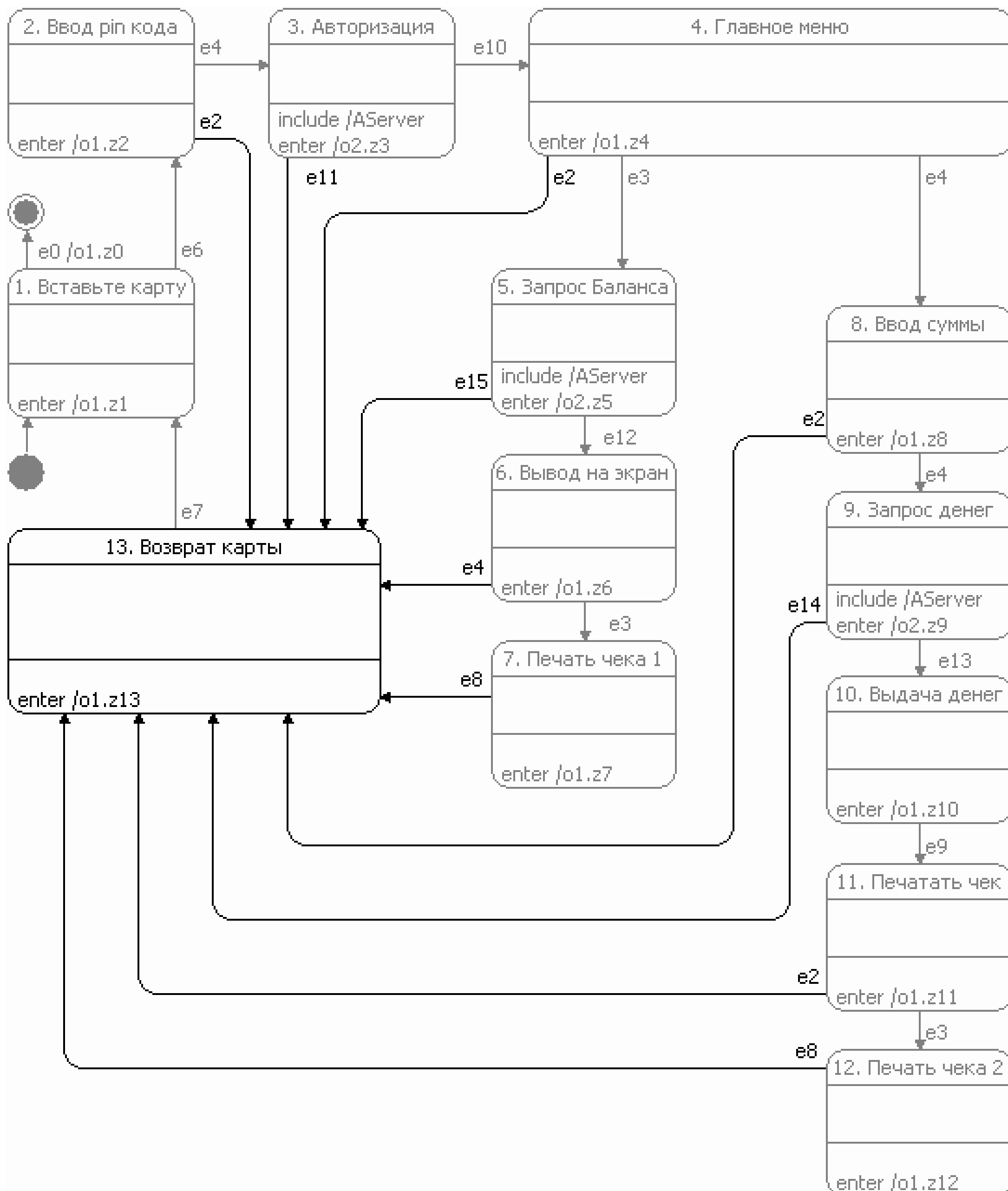


Рис. 30. Выполняющие позиции для первоначальной формулы

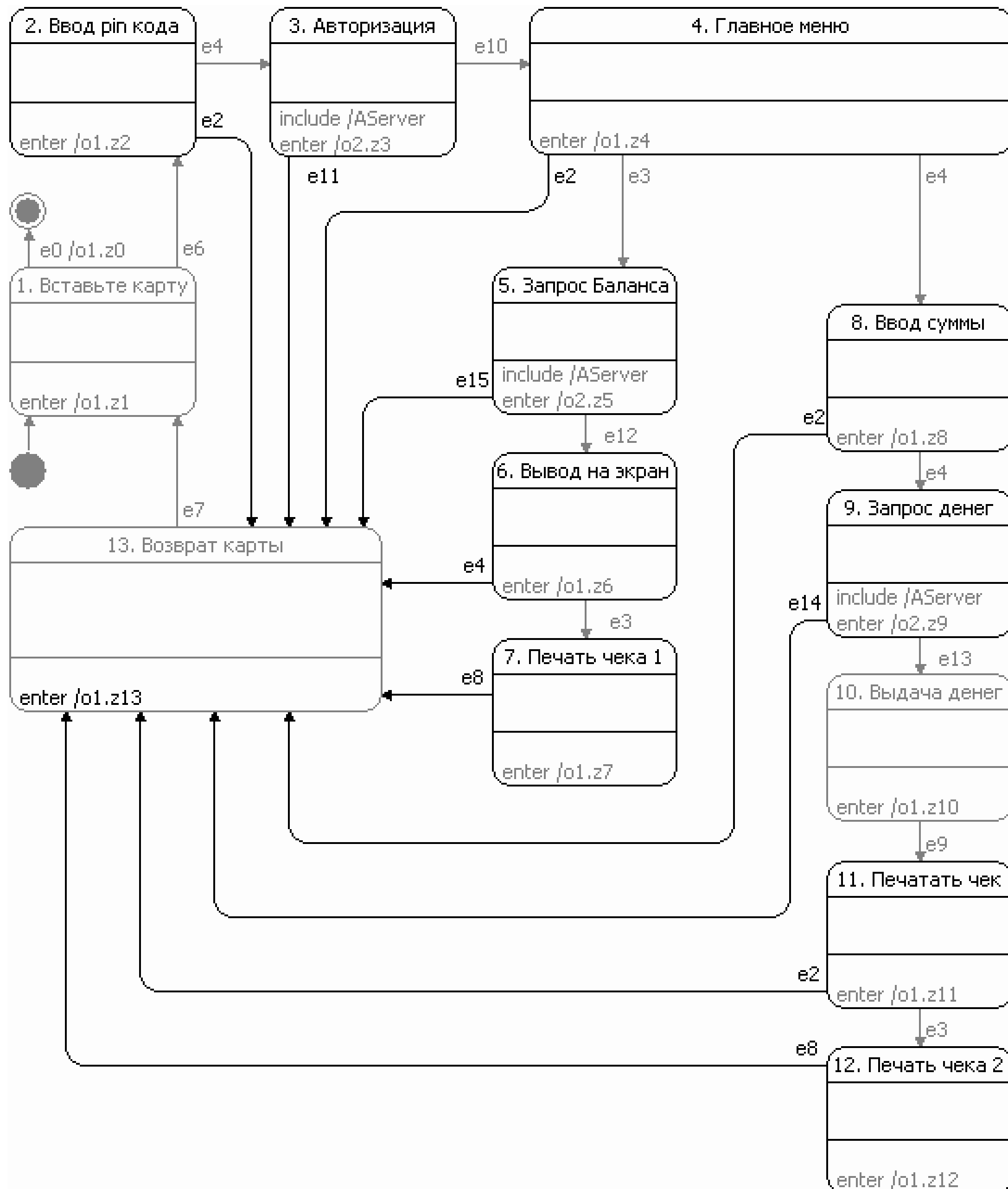


Рис. 31. Выполняющие позиции для правильной формулы

2.2.2.6. Выдаваемая сумма не превышает остаток на счете

Проверяется отсутствие последовательности действий, при которой пользователь получает большую сумму денег, чем имеющаяся на счете (истинное свойство 5). Более точно: «Если произошло событие `e14`, то невозможно попасть в состояние 10, минуя позицию `o3.z0` (новый запрос к серверу)». В этом случае *CTL*-формула имеет вид: $e14 \rightarrow \neg\mathbf{E}[\neg o3.z0\ \mathbf{U}\ y10]$. Она верна во всех позициях модели. Все пути из позиции e14 в состояние 10 через позицию `o3.z0` дважды «проходят» через автомат `AServer`: в первый раз – попав в состояние 3, а второй – попав в состояние 9 автомата `AClient`. Пример такого пути (не проходящего дважды через одно состояние одного и того же экземпляра автомата) выделен на рис. 32 и рис. 33.

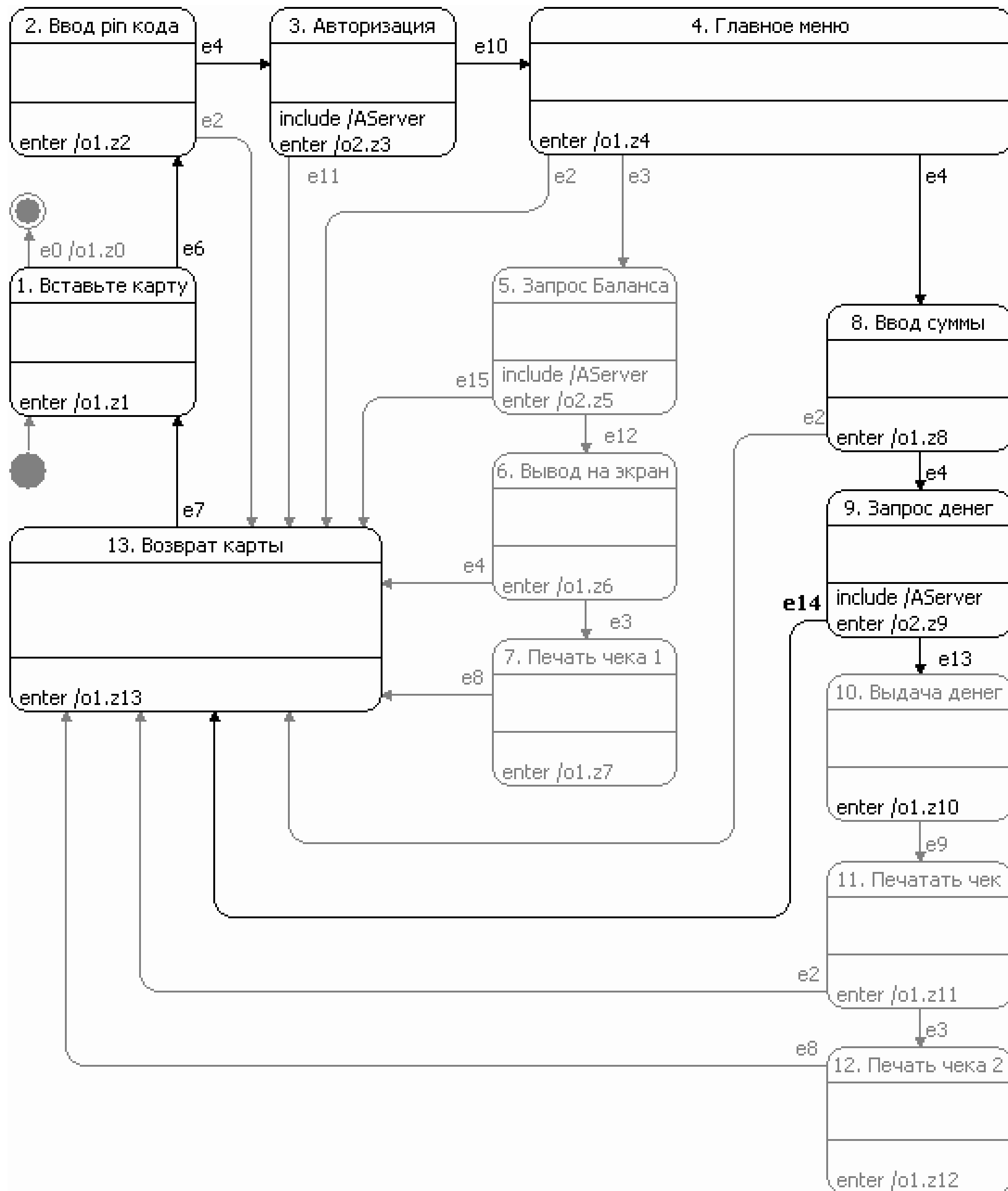


Рис. 32. Участок простого пути в автомате `AClient`

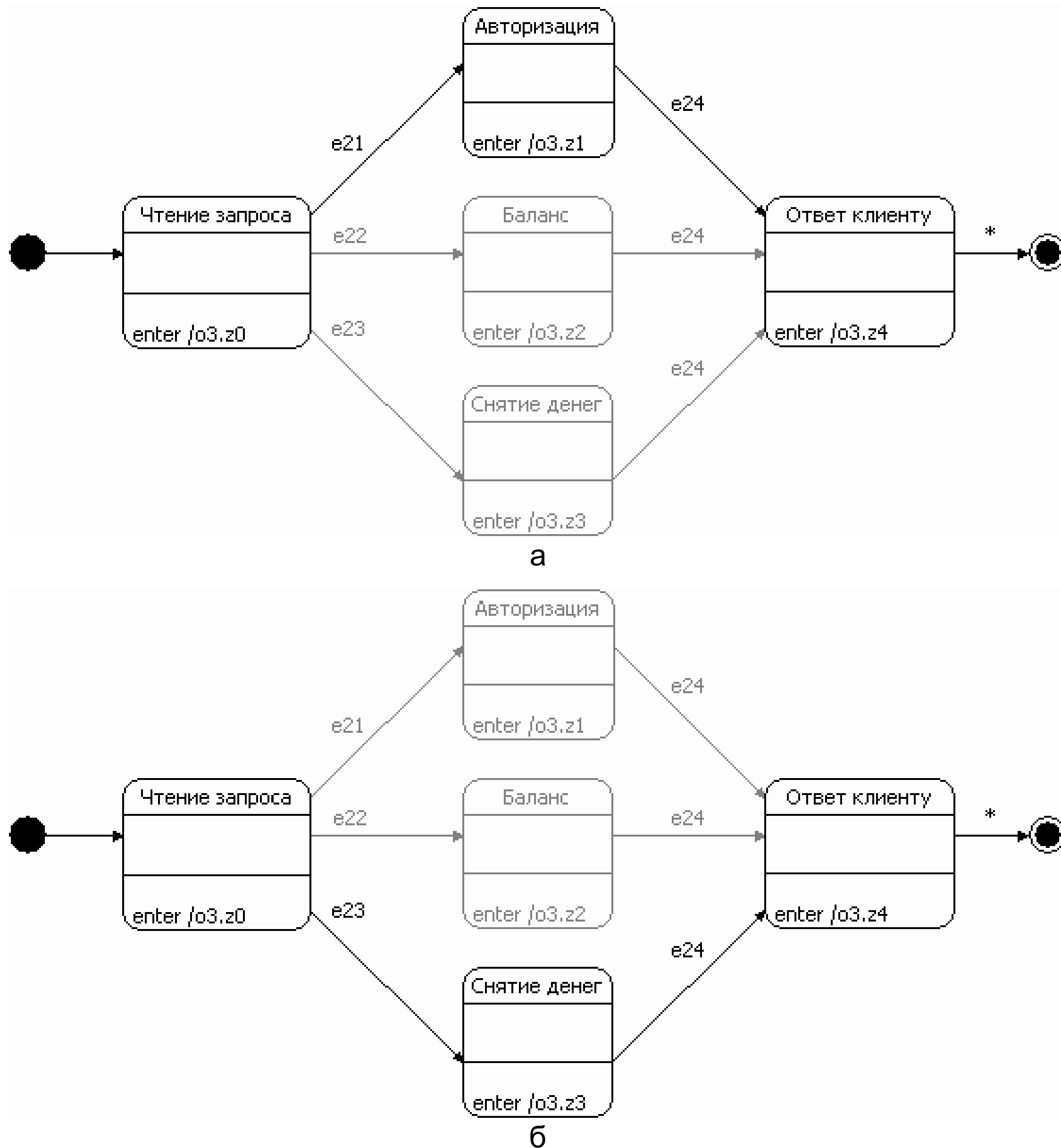


Рис. 33. Участок простого пути в автомате `AServer` при первом запуске (а) и при втором (б)

2.2.2.7. Банкомат выдает деньги только при указании суммы

Проверяется отсутствие последовательности действий, при которой пользователь получает деньги до указания снимаемой суммы (истинное свойство 3). Более точно "Не существует пути, который начинается в текущей позиции, содержит действие `o1.z10` и до того, как `o1.z10` появится впервые, не проходит через состояние 8". При этом CTL-формула имеет вид: $\neg\mathbf{E}[\neg y8\ \mathbf{U}\ o1.z10]$. Проверим эту формулу для позиции, выделенной жирным шрифтом на рис. 25 (выходное воздействие `o1.z4` в состоянии 4 «главное меню»). Формула выполняется. На рис. 25 выделен единственный простой путь из этой позиции в позицию `o1.z10`. Этот путь проходит через состояние 8.

Попробуем теперь изменить модель. В автомат `AClient` добавим переход из состояния 4 в состояние 9 по событию `e1` (нажатие кнопки 1). В позиции `o1.z4` измененной модели формула $\neg\mathbf{E}[\neg y8\ \mathbf{U}\ o1.z10]$ не выполняется. Подтверждающая трасса выделена на рис. 34.

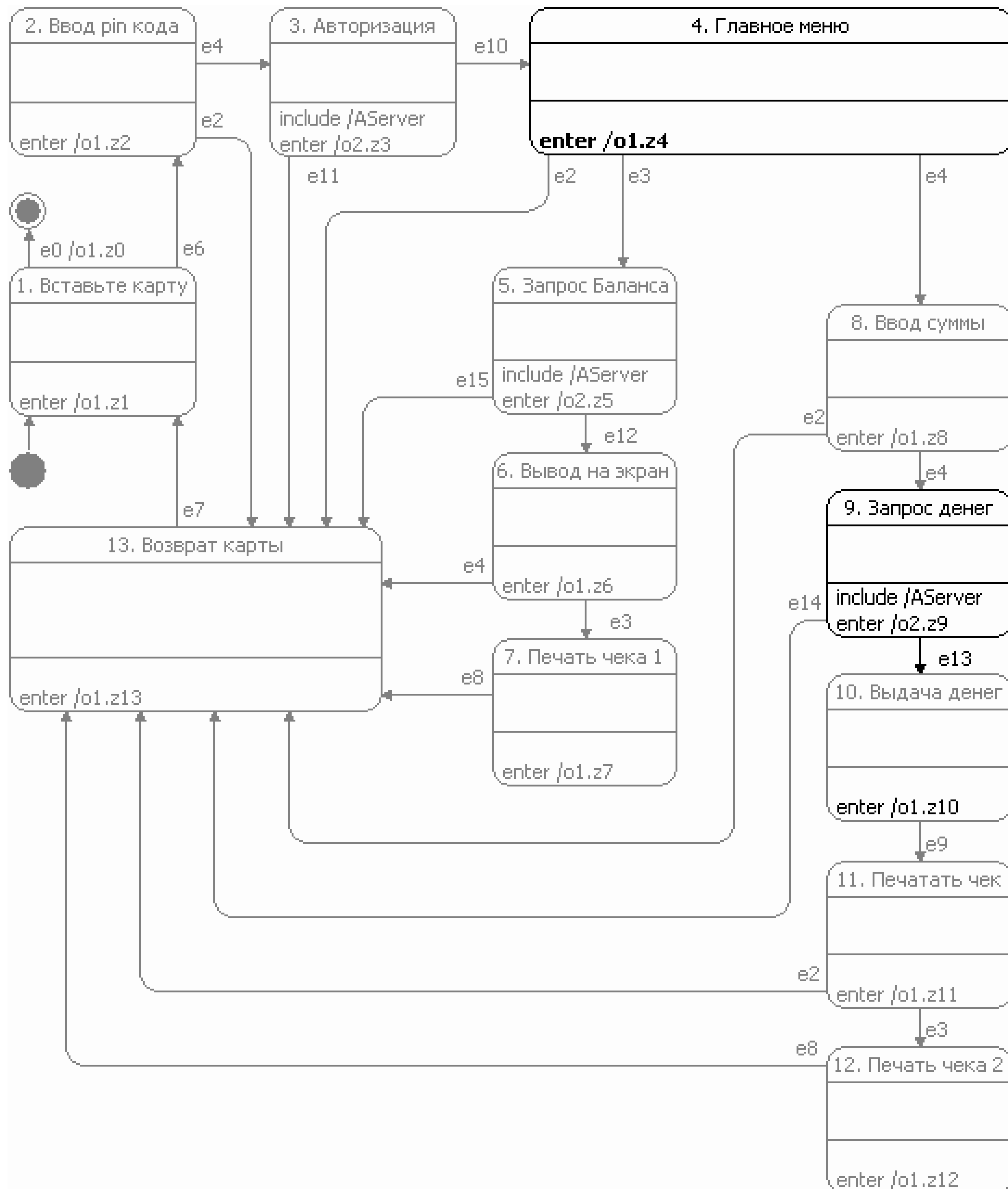


Рис. 34. Контрпример в изменённой модели для истинного свойства 3

2.2.2.8. Возврат карты при ошибке

Проверяется, что если происходит ошибка, то карта в любом случае будет возвращена. Более точно: "Любой путь, начинающийся в позиции `e15` (ошибка на сервере) проходит через состояние 13". При этом *CTL*-формула имеет вид: $e15 \rightarrow \mathbf{AF}\, y13$. Формула выполняется во всей модели.

Теперь изменим модель. Добавим переход из состояния 9 (запрос денег) в состояние 4 (главное меню) по событию `e15`. В этой позиции `e15` существует циклический путь, выделенный на рис. 35, который является контрпримером – опровергающей трассой для *CTL*-формулы $\mathbf{AF}\, y13$.

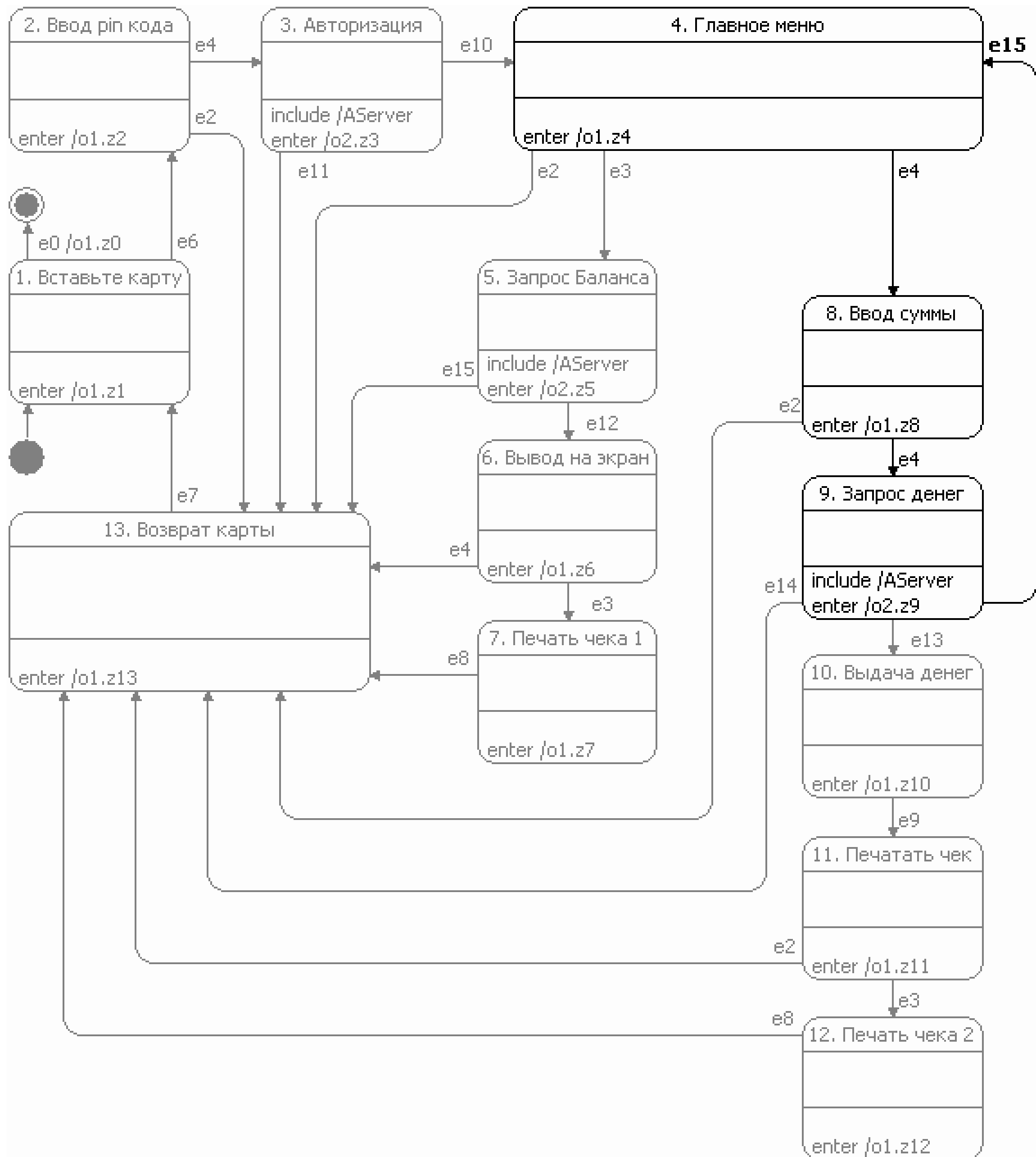


Рис. 35. Контрпример в изменённой модели

2.2.2.9. Безусловная выдача денег

Проверяется, что при любой последовательности действий пользователю будут выданы деньги (ложное свойство 1). Более точно: "Любой путь, начинающийся в текущей позиции, проходит через позицию `e9` (деньги выданы) ". Соответствующая формула языка *CTL* имеет вид: $\mathbf{AF}\, e9$. Это свойство практически всегда неверно. Действительно, формула почти во всех позициях не выполняется. Выполняется она только в четырёх позициях, выделенных на рис. 36 (события `e13` и `e9`, состояние 10, выходное воздействие `o1.z10`). В любую из этих позиций автомат `AClient` попадает только при успешном запросе на выдачу денег. Они описывают логику процесса выдачи денег в автоматной модели. Уже в состоянии 9 эта формула не выполняется (при запросе денег на счету может не оказаться запрашиваемой суммы. В этом случае произойдёт событие `e14`). Опровергающая трасса (одна из возможных) для состояния 9 приведена на рис. 37. Эта трасса, начиная с позиции `o1.z13` в состоянии 13, является периодической.

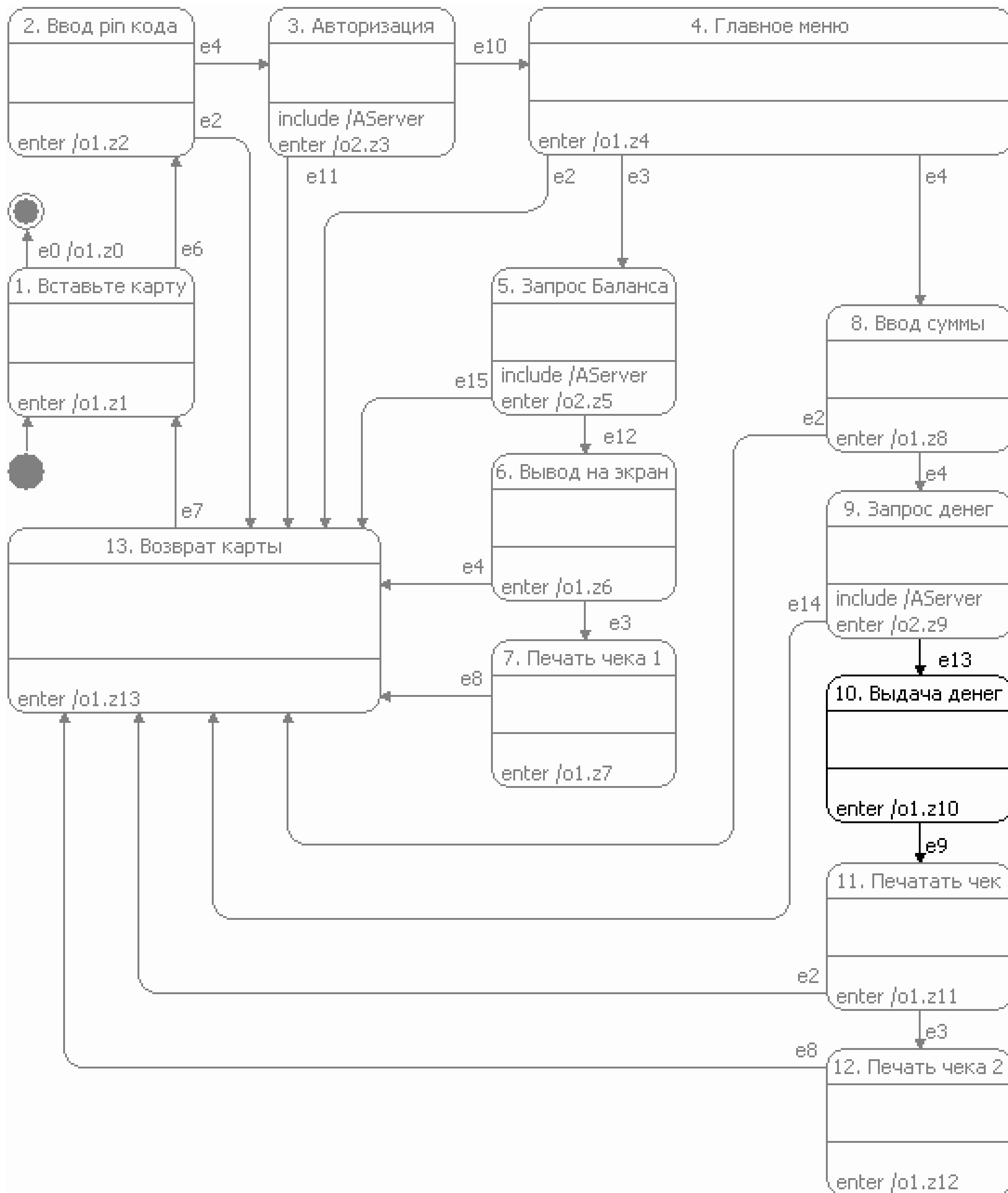


Рис. 36. Выполняющие позиции для ложного свойства 1

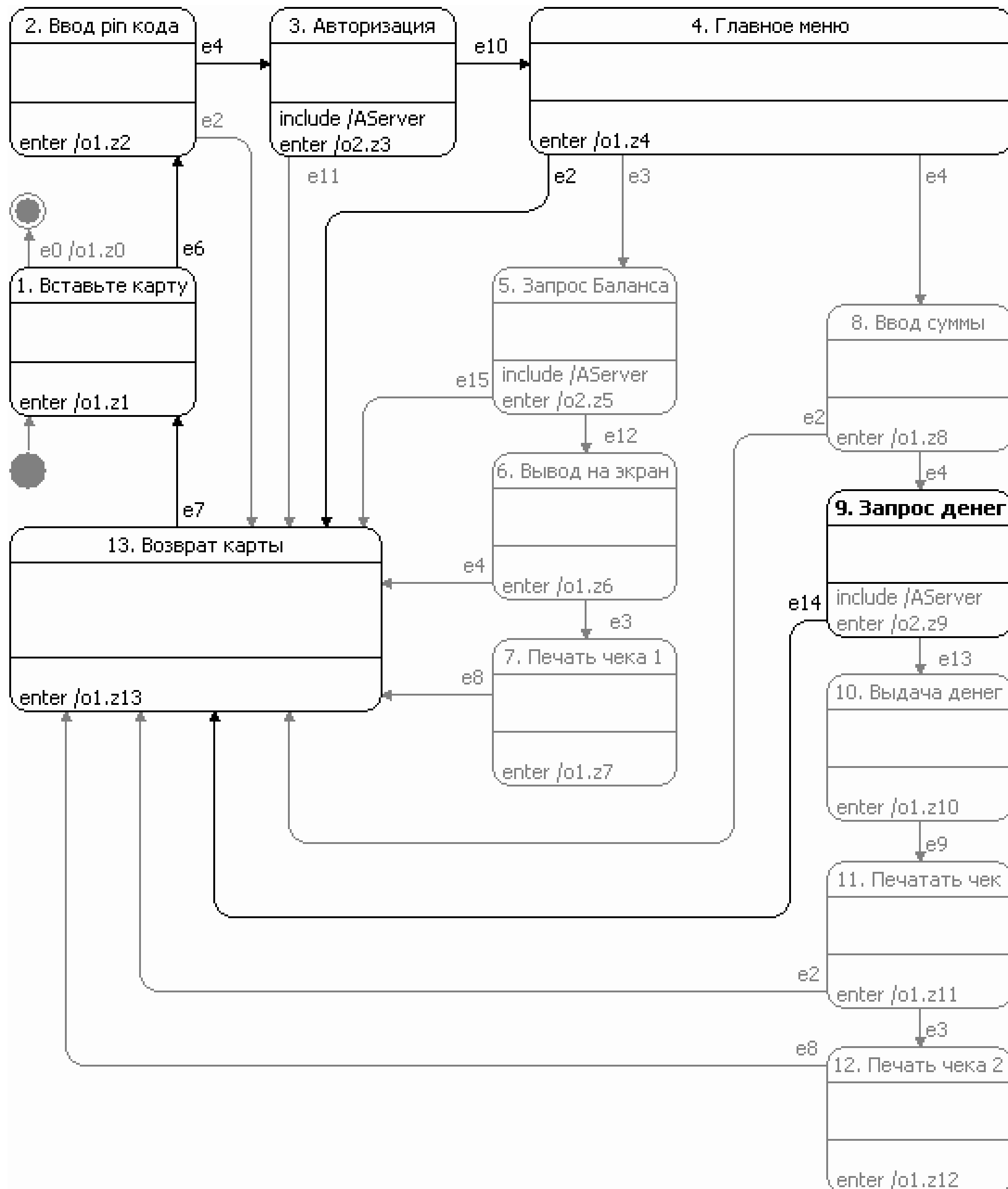


Рис. 37. Контрпример для ложного свойства 1

2.2.2.10. Повторный запрос карты

Проверяется, что банкомат может запросить карту, если она уже вставлена (ложное свойство 2). Более точно: "Существует путь из текущей позиции в состояние 1, не проходящий через событие `e7` (карта вытащена)". Соответствующая *CTL*-формула имеет вид: $\mathbf{E}[\neg e7\ \mathbf{U}\ y1]$. Это свойство не выполняется всегда, кроме исключительных позиций, выделенных на рис. 38. Это именно те позиции, при которых, соответственно, карта не вставлена (или уже вынута).

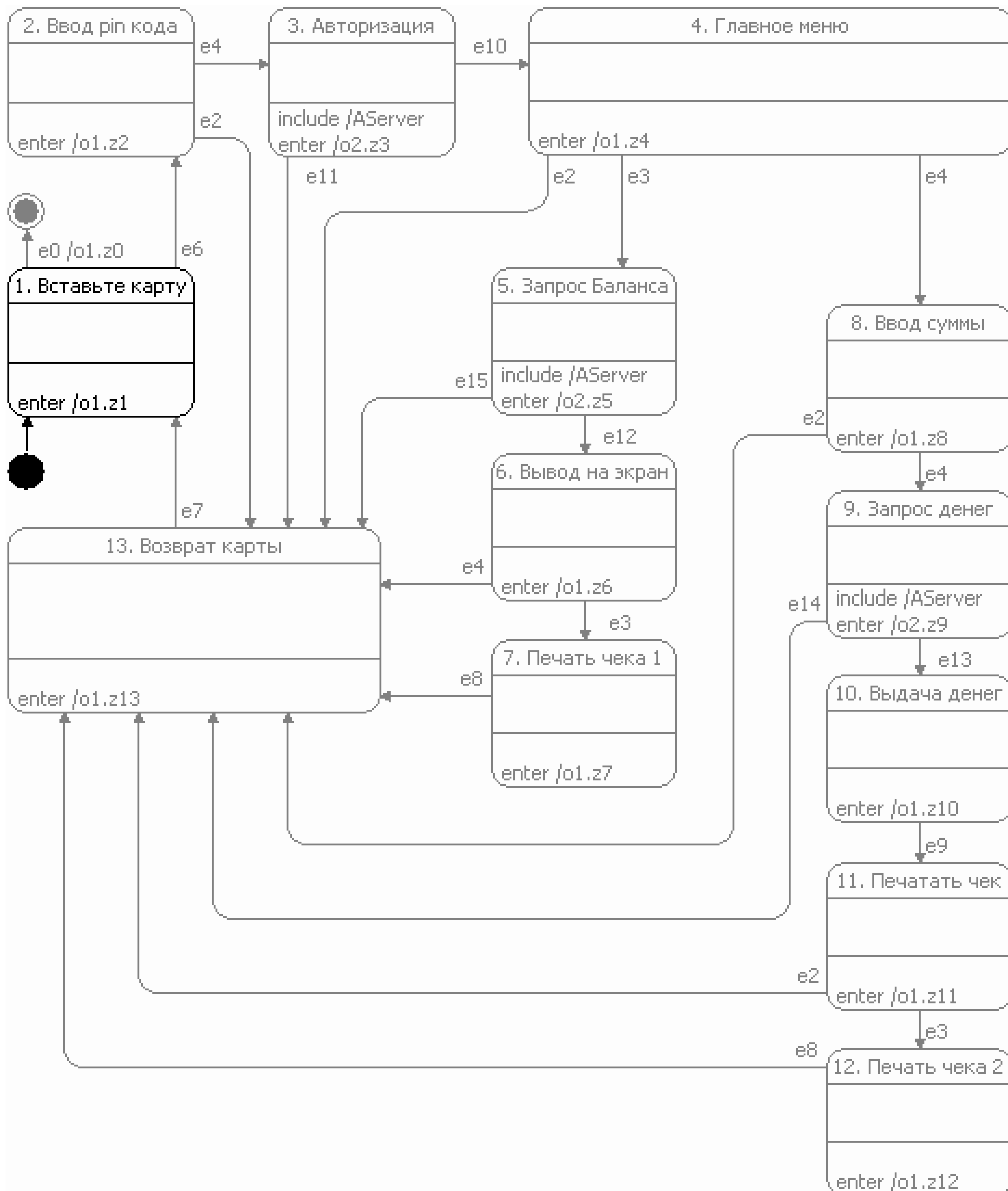


Рис. 38. Выполняющие позиции для ложного свойства 2

2.2.2.11. Снятие денег после запроса балансов

Проверяется, что банкомат может разрешить снять деньги после запроса баланса без повторной авторизации (ложное свойство 3). Более точно "Если автомат `AClient` находится в состоянии 6, то существует путь в состояние 8 (ввод суммы), который не проходит через состояние 13 (возврат карты)". Соответствующая *CTL*-формула: $y6 \rightarrow \mathbf{E}[\neg y13 \; \mathbf{U} \; y8]$. Это свойство неверно: оно не выполняется в состоянии 6. Действительно, единственный простой путь из состояния 6 в состояние 8 проходит через состояние 13 (рис. 39). Формула $y6 \rightarrow \mathbf{E}[\neg y13 \; \mathbf{U} \; y8]$ выполняется во всех позициях (кроме состояния 6), так как все они не помечены соответствующим атомарным предложением (`Y=6`).

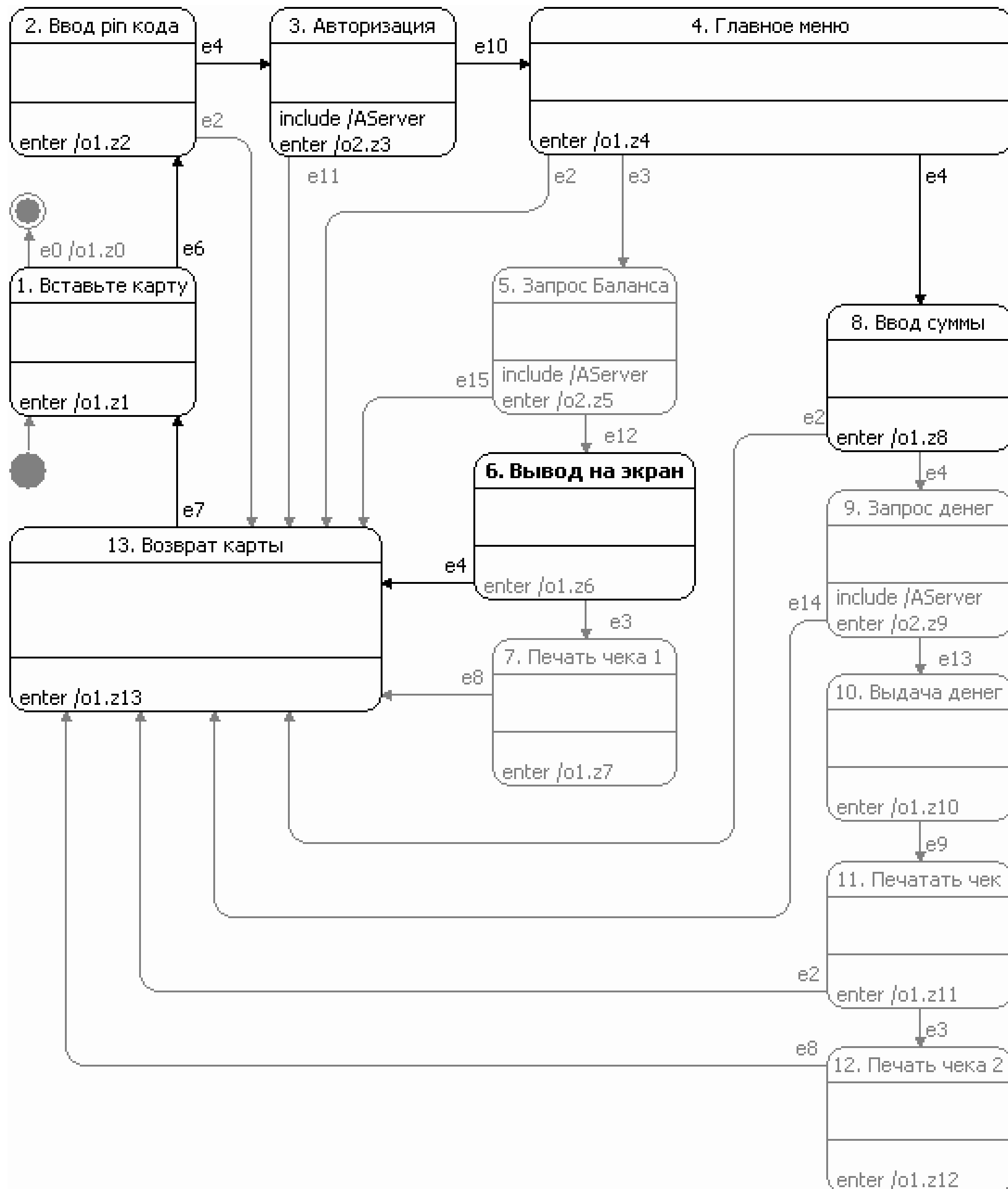


Рис. 39. Простой путь из состояния 6 в состояние 8

2.2.2.12. Возвращение карты до ошибки

Проверяется, что при любом поведении банкомата пользователь сможет получить свою карту назад до того, как произойдёт ошибка на сервере. Более точно "В любом протоколе работы автомата появится состояние 13, и до первого его появления никогда не возникнет событие e15". Соответствующая *CTL*-формула: $\mathbf{A}[\neg e15\ \mathbf{U}\ y13]$. Позиции, в которых это свойство выполняется, выделены на рис. 40. Пример позиции, в которой эта формула не выполняется – состояние 1. Опровергающая трасса для этой позиции (путь из состояния 1 в состояние 13, проходящий через позицию e15), выделена на рис. 41.

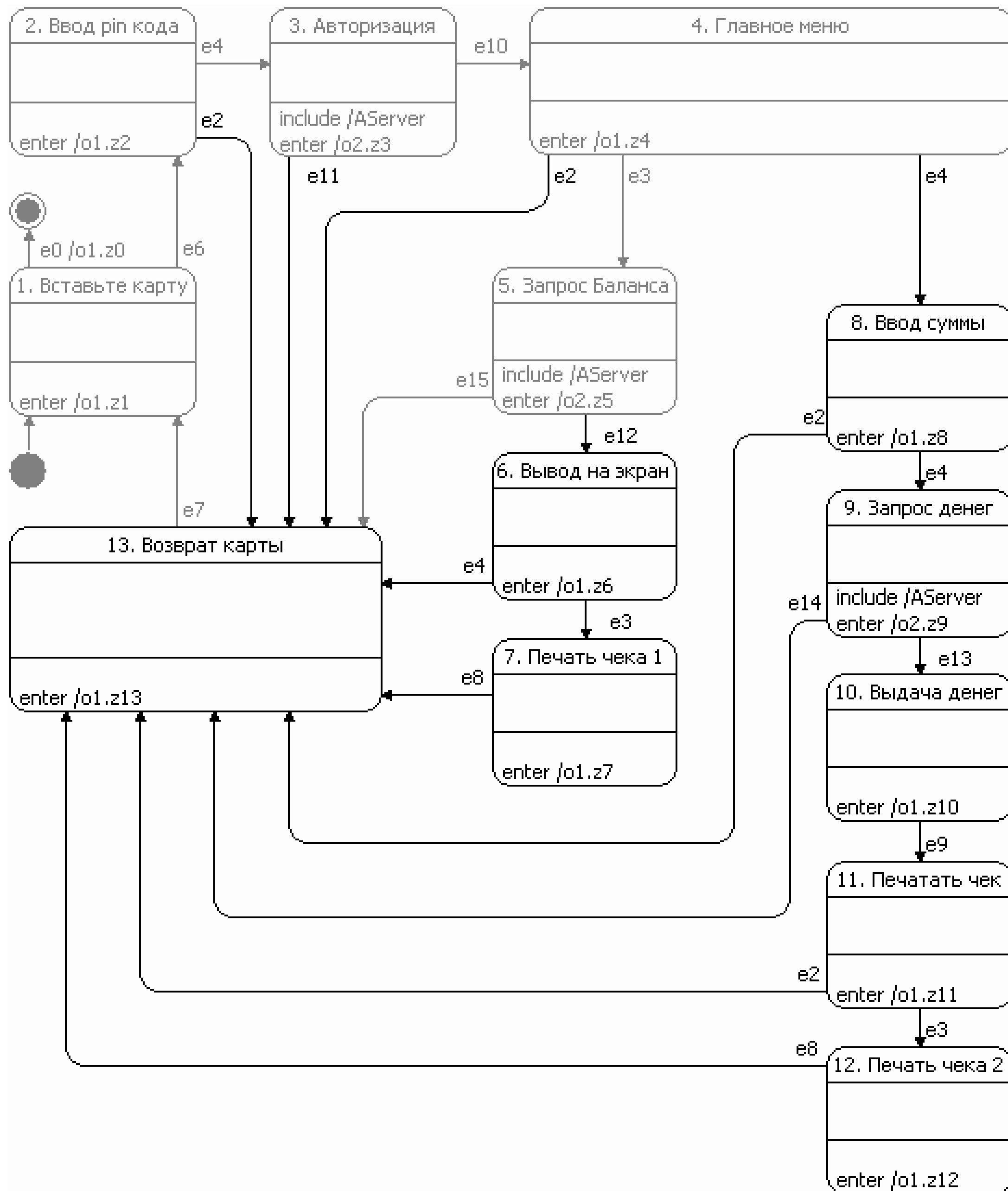


Рис. 40. Выполняющее множество для возвращения карты до ошибки

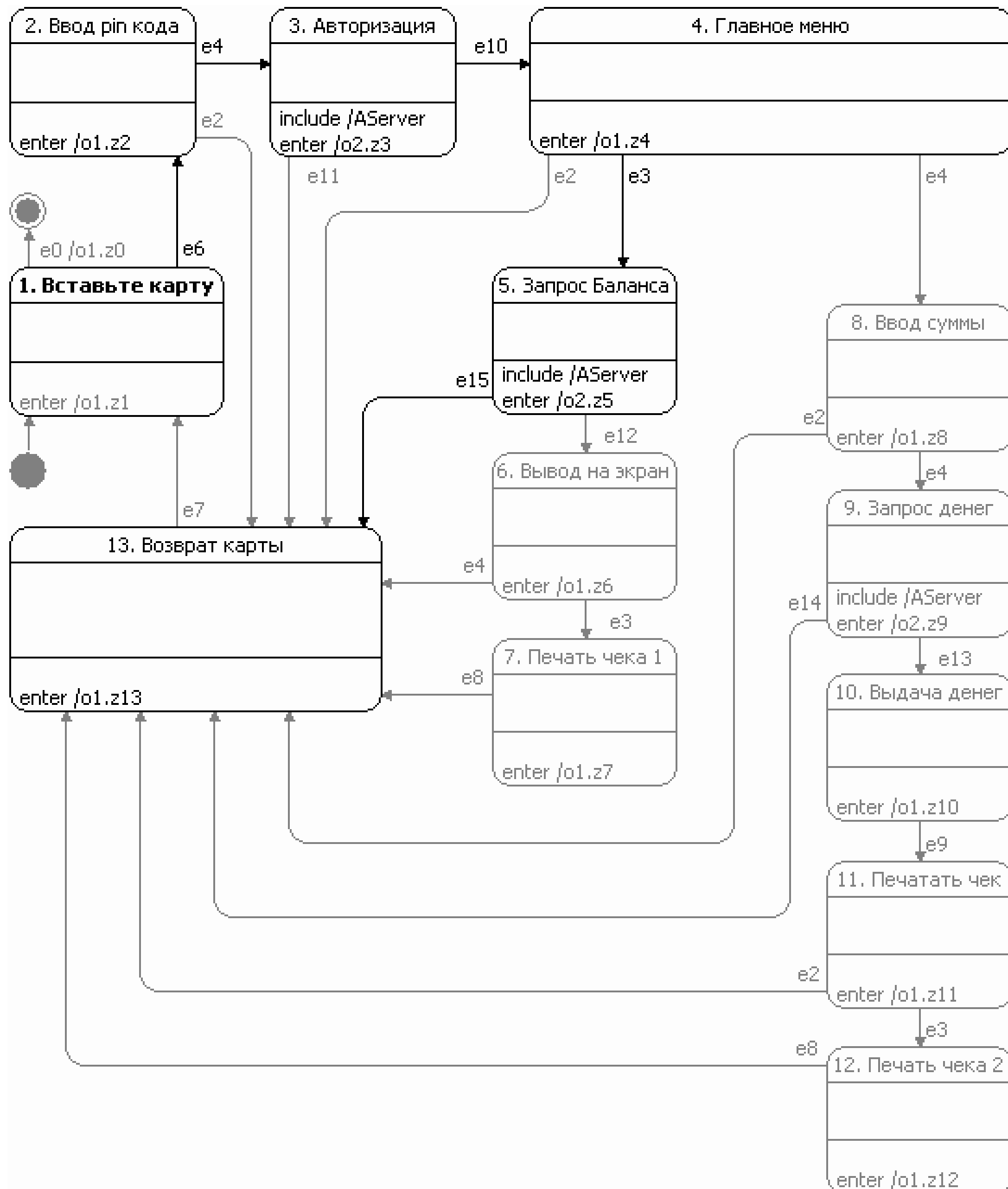


Рис. 41. Контрпример для состояния 1

2.2.2.13. Безошибочные пути

Допустим, требуется проверить, не забыл ли проектировщик установить в автомате переходы по событиям, при которых происходят ошибки, и переходы, при которых ошибок не происходит. Будем проверять свойство "Существует путь, не содержащий ошибок (событий `e15`), но не любой путь удовлетворяет этому условию". Соответствующая *CTL*-формула: $(\mathbf{EG}\neg e15)\wedge(\neg\mathbf{AG}\neg e15)$. Эта формула выполняется во всех позициях модели, кроме нескольких исключений: перехода в терминальное состояние и перехода по событию `e15`. Подтверждающие трассы для первого и второго операндов конъюнкции выделены на рис. 42 и рис. 43 соответственно. На рис. 42 стартовая позиция — выходное воздействие `o1.z12`, размещенное в состоянии 12. На рис. 43 любую позицию выделенного цикла можно обозначить стартовой.

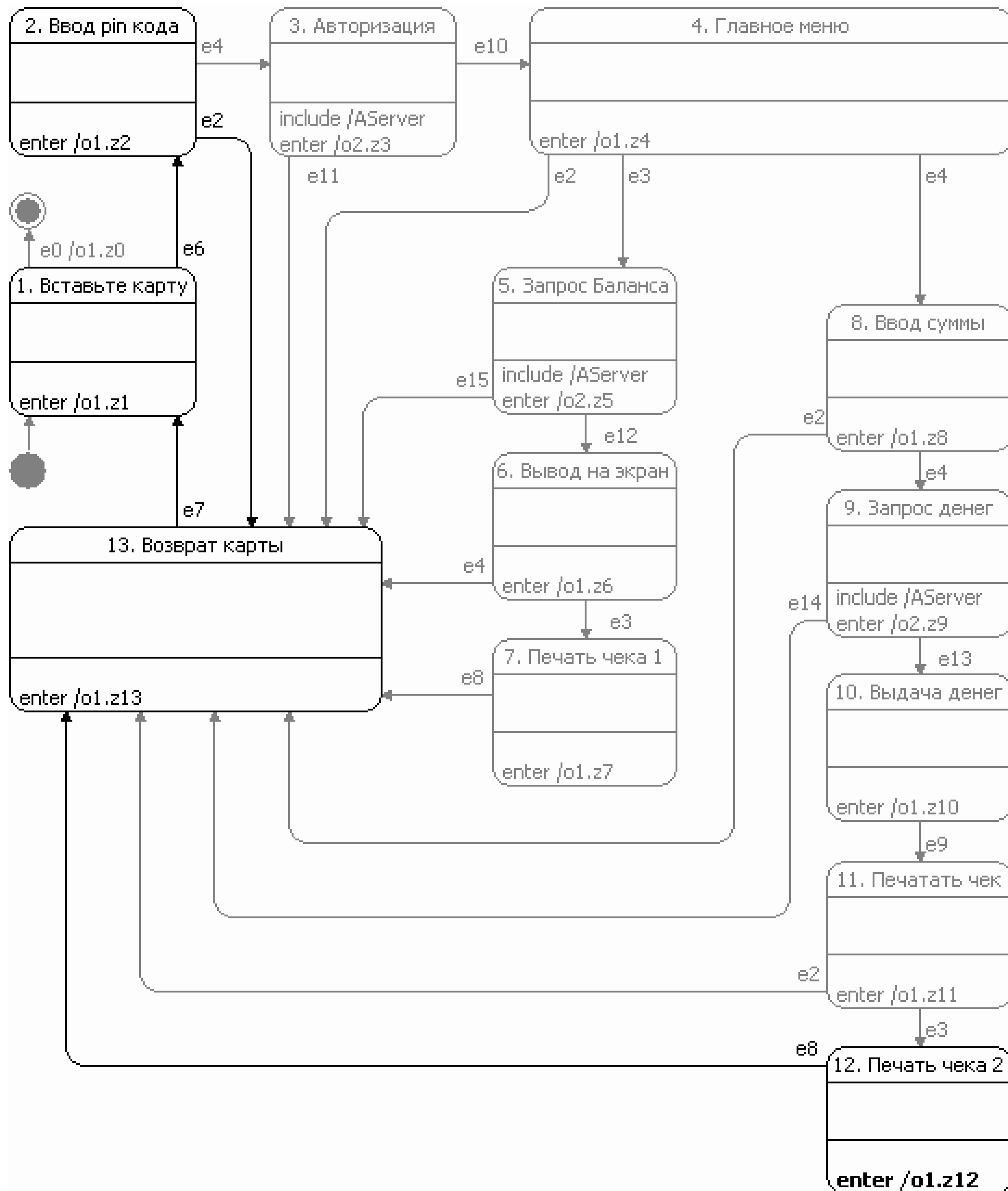


Рис. 42. Пример пути для первого операнда

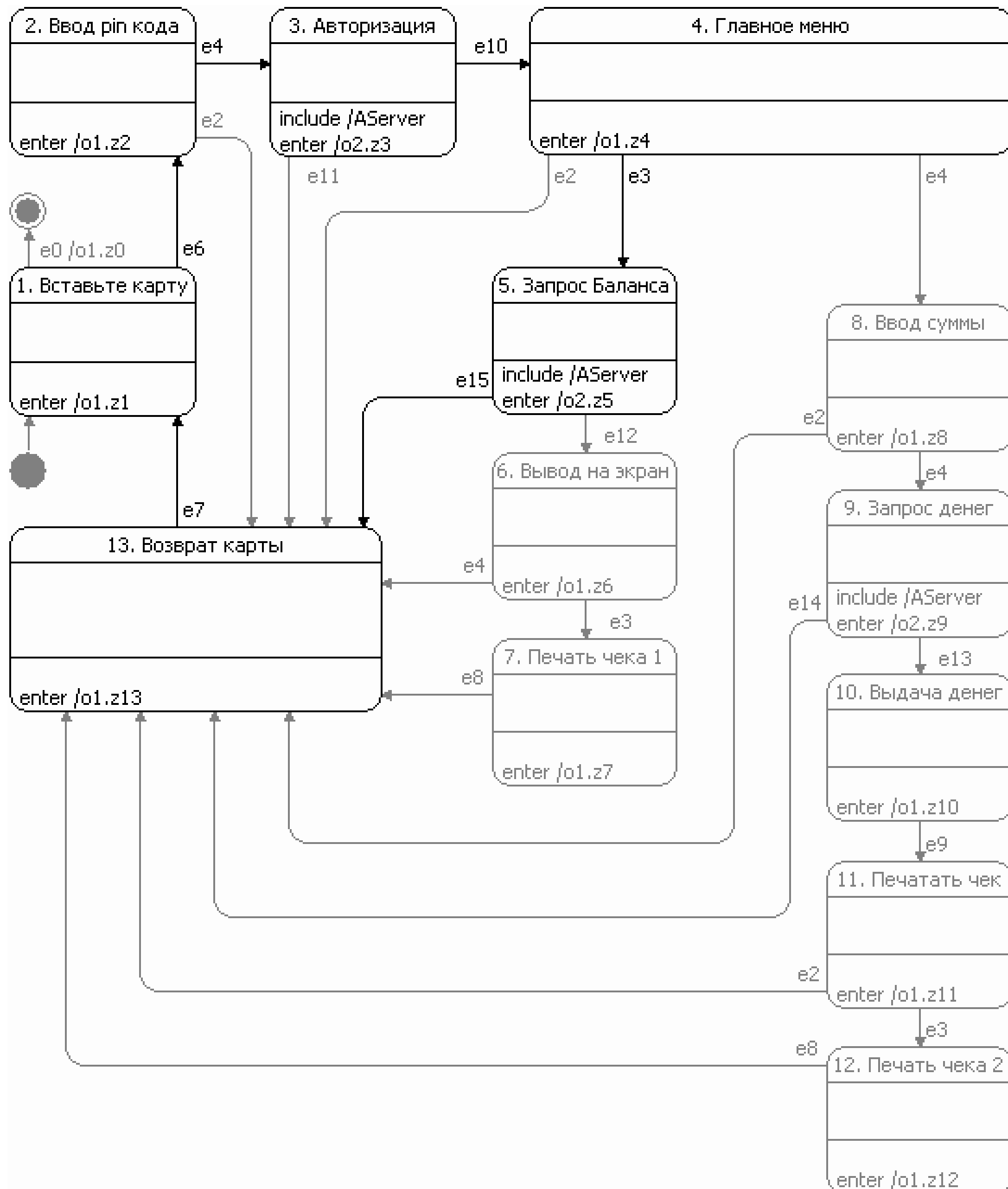


Рис. 43. Пример циклического пути для второго операнда

2.2.2.14. Выдача достаточной суммы денег

Проверяем свойство: "Если при запросе на снятие денег суммы на карте оказалось достаточно, то пользователь получит деньги". Более точно: "Если выполняется действие `o2.z9`, то в любом протоколе работы автомата появится позиция, которая является если не событием `e14`, то действием `o1.z10`". Соответствующая *CTL*-формула: $o2.z9 \to \mathbf{AF}(\neg e14 \to o1.z10)$. Эта формула выполняется во всех позициях автомата. Во всех позициях, кроме `o2.z9`, она выполняется потому, что посылка импликации ложна. В позиции `o2.z9` она выполняется потому, что каждый из путей, стартующих в этой позиции, в конечном счёте проходит либо через позицию `e14`, либо через позицию `o1.z10`.

2.2.2.15. Выход в главное меню

Проверяется свойство "В течение всей работы автомата в любой момент есть возможность попасть в главное меню". Более точно: "В любом протоколе работы автомата из любой позиции можно достичь состояния 4". Соответствующая *CTL*-формула: $\mathbf{AG\,EF}\,y14$.

Эта формула не выполняется ни в одной позиции автомата. Действительно, из каждой позиции существует путь в терминальное состояние, из которого не существует пути в состояние 4 (пример со стартовым состоянием 8 приведён на рис. 44).

Попробуем теперь изменить модель. Уничтожим терминальное состояние и переход в него из состояния 1. Формула сразу стала выполняться во всех состояниях. Действительно, в первоначальной модели наличие перехода из состояния 1 в терминальное состояние «блокировало» выполнимость темпоральной операции $\mathbf{AG}$. Теперь этот «лишний» переход из состояния 1 отсутствует, а из него существует путь в состояние 4 (рис. 45).

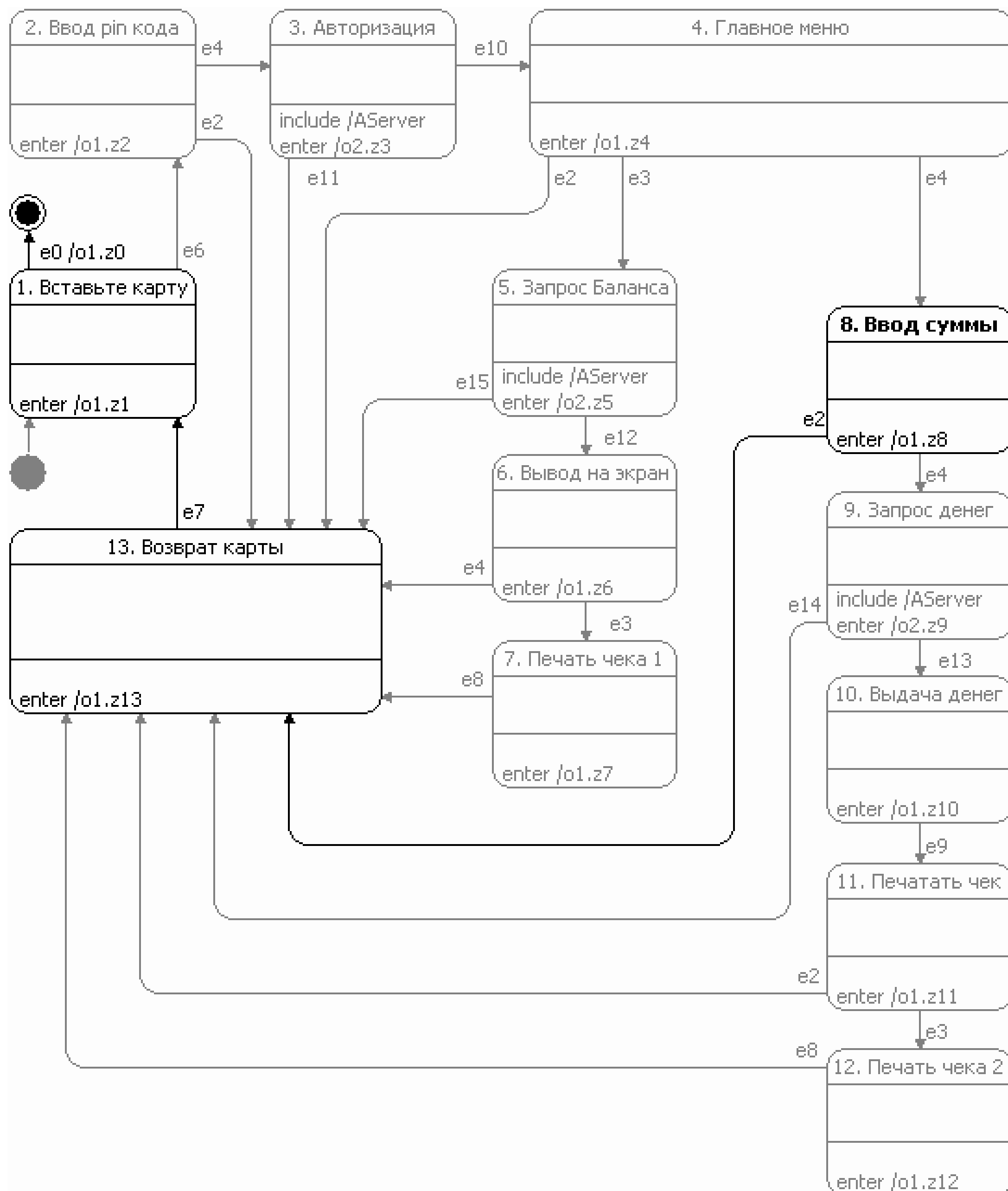


Рис. 44. Опровергающий пример для состояния 8

s

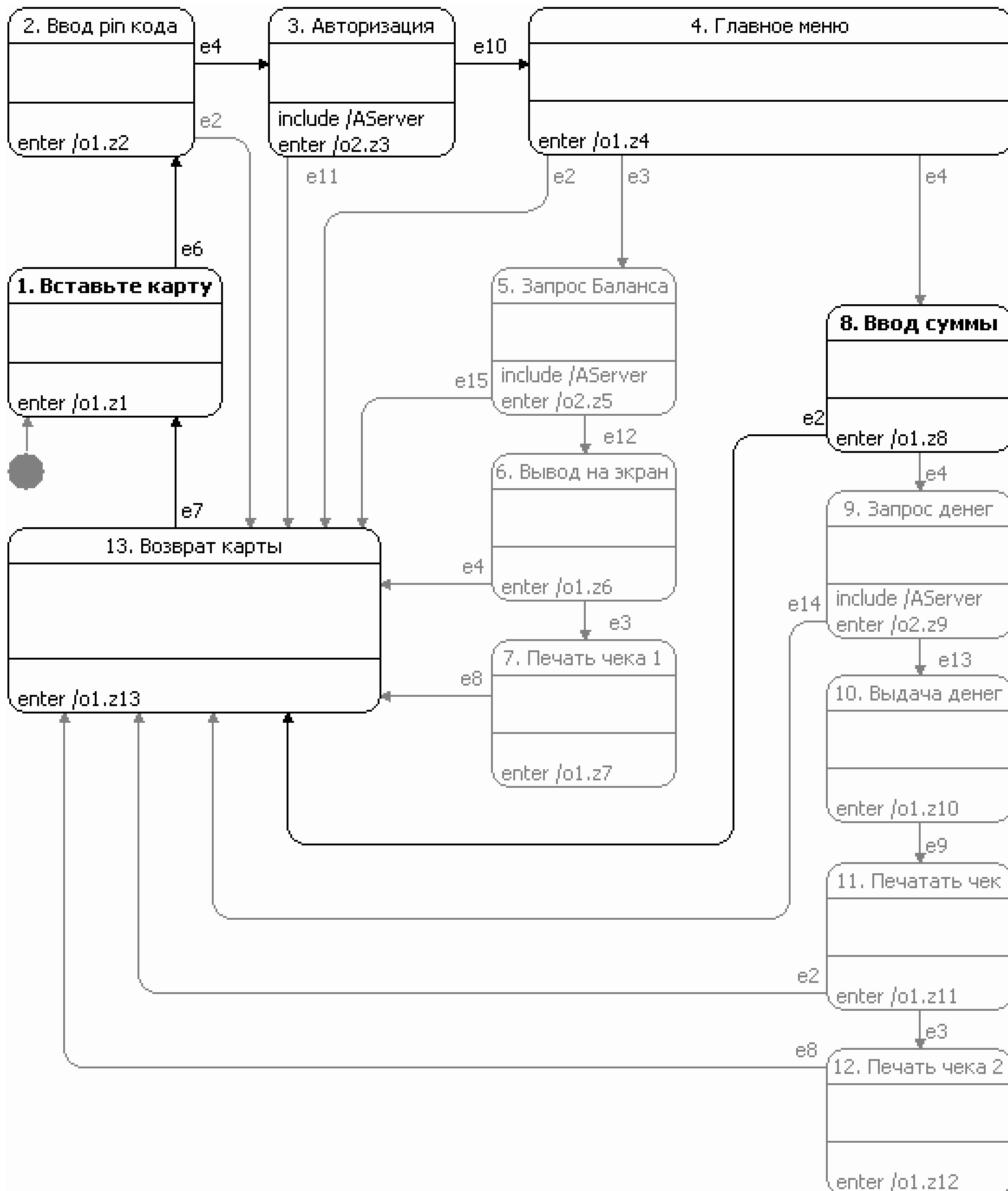


Рис. 45. Перестроенная трасса, ставшая подтверждающей

2.3. Верификация при помощи *Converter*

2.3.1. Описание метода

2.3.1.1. Построение модели на языке *Promela* по визуальной автоматной программе

При построении модели используются:

- графы переходов автоматов;
- взаимодействие автоматов по вложенности;
- события на переходах.

Построенная модель абстрагируется от:

- входных переменных, обозначаемых как *x* с соответствующими индексами;
- выходных воздействий, обозначаемых как *z* с соответствующими индексами;

Такая абстракция позволяет генерировать более простые модели, обеспечивая возможность верификации программ большей размерности. Вместе с тем, более подробная модель позволяет более точно верифицировать программы. Генерация более подробной модели на языке *Promela* – дело будущего.

Опишем метод построения модели.

1. Для каждого автомата `Ai` заведем переменную `stateAi`, в которой будет храниться номер текущего состояния. На языке *Promela* это описывается следующим образом:

```
int stateAi;
```

2. Введем переменную для событий:

```
int lastEvent;
```

3. Каждому состоянию присвоим уникальный номер, используя сквозную нумерацию для всех автоматов. Переход в новое состояние с номером `k` будет осуществляться присвоением переменной `stateAi` числа `k`:

```
stateAi = k;
```

4. Событие exx (xx — номер события) на переходе между состояниями описывается в модели следующим образом:

```
lastEvent = xx;
```

5. Для каждого автомата `Ai` создадим функцию. На языке *Promela* это записывается следующим образом:

```
inline Ai() {
  /* тело функции */
}
```

6. Для каждого автомата `Ai` в теле созданной функции выполнить шаги 2.3 – 2.9.
7. Определить начальное (стартовое) состояние `s`. Присвоить:

```
stateAi = s;
```

8. Построить цикл:

```
do
::(stateAi == s1) ->
  printf("State 1 : Имя состояния\n");
::(stateAi == s2) ->
  printf("State 2 : Имя состояния\n");
…
::(stateAi == sk) ->
  printf("State k : Имя состояния\n");
od;
```

Здесь `s1, …, sk` — номера состояний автомата `Ai`.
Инструкция `printf` аналогична соответствующей инструкции из языка *C*. Пометка используется в дальнейшем для восстановления контрпримера.

9. Если в некоторое состояние `sj` вложен автомат `Am`, то дописать в условие `(stateAi == sj)` вызов функции этого автомата:

```
Am();
```

10. Для каждого состояния `sj` найти все возможные переходы `(sj, sl)` из него. К условию `(stateAi == sj)` дописать конструкцию `if`, а для каждого перехода `(sj, sl)` дописать в конструкции `if` следующее присваивание:

```
::stateAi = sl;
```

Это обозначает, что для присваивания `stateAi = sl` необходимое условие выполнено всегда. В результате получится конструкция вида:

```
if
::stateAi = s1
```

```
::stateAi = s2
…
::stateAi = sk
fi;
```

Эта конструкция обозначает, что присваивание нового номера состояния происходит недетерминировано. Таким образом, верификатор проверит все варианты переходов в новое состояние.

11. Если переход помечен событием `exx`, то дописать выражение:

```
lastEvent = xx;
```

Таким образом, в результате получается конструкция вида:

```
if
…
::stateAi = sl;
  lastEvent = xx;
…
fi;
```

12. Если состояние `st` – конечное, то в условие `(stateAi == st)` дописать инструкцию завершения цикла:

```
break;
```

13. После построения функций для всех автоматов определить стартовый автомат `Ai` и создать процесс, его запускающий. На языке *Promela* это записывается следующим образом:

```
proctype Model() {
  Ai();
}

init {
run Model();
}
```

14. Дописать в модель требования (проверяемые свойства), преобразованные с помощью верификатора *SPIN* с языка *LTL* на язык *Promela*.

2.3.1.2. Расширение нотации верификатора *SPIN* для языка *LTL*

Нотация верификатора *SPIN* для языка *LTL* предполагает, что все элементарные высказывания записаны в виде некоторых идентификаторов, которые расшифровываются отдельно [4]. Излагаемый метод расширяет указанную нотацию *SPIN*, добавляя возможность записи элементарных высказываний в формуле. В качестве элементарного высказывания может использоваться любое выражение на языке *Promela*. Элементарное высказывание записывается в фигурных скобках.

Опишем алгоритм преобразования формулы, записанной в расширенной нотации, к формуле в нотации верификатора *SPIN*.

1. Алгоритму на вход подается строка с формулой.
2. Вводится счетчик элементарных высказываний.
3. Пока не кончилась строка, идем по строке в поиске открывающих фигурных скобок.
4. Если нашли открывающую фигурную скобку, то увеличиваем счетчик на единицу и идем по строке дальше в поиске закрывающей фигурной скобки.
5. Если закрывающая фигурная скобка не была найдена до конца строки или до момента появления новой открывающей фигурной скобки, то *LTL*-формула неправильная. Прекратить работу.
6. Если закрывающая фигурная скобка была найдена, то заменим элементарное высказывание, находящееся между фигурными скобками на идентификатор `pk`,

который состоит из символа `p` и числа `k`, которое показывает счетчик элементарных высказываний, и запишем в модель следующий код:

```
#define pk (элементарное высказывание)
```

Переходим к шагу 3.

7. Если строка закончилась, то завершить работу.

В таком виде *LTL*-формула может быть обработана верификатором *SPIN* (при условии, что темпоральные операторы записаны в нотации верификатора *SPIN* [4]).

Заметим, что данный алгоритм не проверяет правильность написания элементарных высказываний, оставляя эту проверку верификатору *SPIN*.

2.3.1.3. Построение контрпримера

Верификатор *SPIN* в качестве внутреннего представления модели строит *модель Крипке* и производит верификацию построенной модели. В случае несоответствия модели проверяемым свойствам верификатор выводит контрпример как путь в модели на языке *Promela*, поданной ему на вход. В модель на языке *Promela* встроены пометки, в которых содержится текущее состояние автомата. Таким образом, в отчете, генерируемом верификатором, содержится вся информация о контрпримере. На настоящее время требуется отчет верификатора анализировать вручную, однако в будущем предполагается разработать алгоритмы, упрощающие процесс анализа.

2.3.2. Верификация свойств

В качестве тестовой модели будет рассмотрен *UniMod*-банкомат (разд. 1.2). На рис. 46 и рис. 47 показаны номера, которые *Converter* присвоил вершинам системы автоматов.

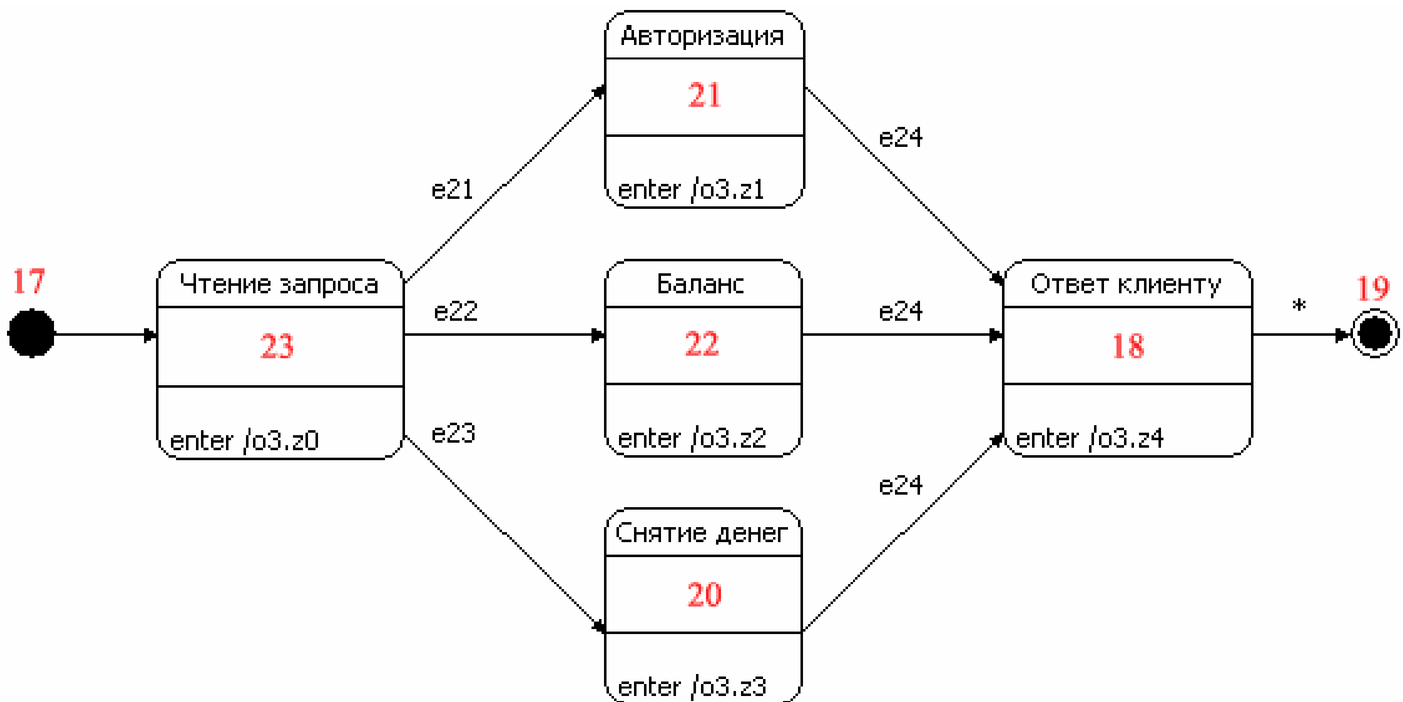


Рис. 46. номера состояний автомата AServer

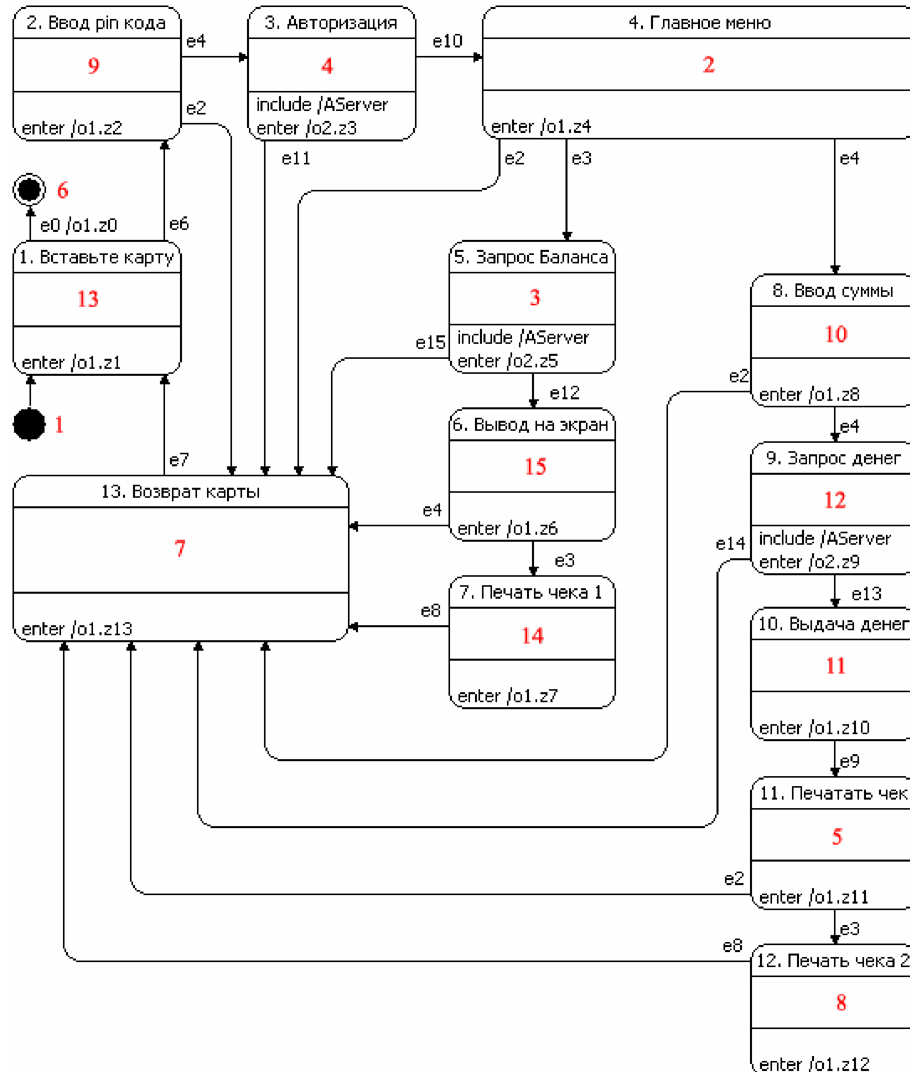


Рис. 47. Номера состояний автомата AClient

2.3.2.1. Банкомат выдает деньги только после авторизации

Проверяется отсутствие последовательности действий, при которой пользователь получает деньги до ввода правильного *PIN*-кода.

Напомним, что на вход верификатору *SPIN* требуется подавать отрицания проверяемых свойств. Запишем отрицание этого свойства: "Пользователь не ввел правильный *PIN*-код, но деньги получил". Для построения формулы на языке *LTL*, соответствующей данному свойству, введем два элементарных высказывания:

`lastEvent == e10` и `stateAClient == 11`.

Первое высказывание означает, что произошло событие *e10* ("*PIN*-код верный") – пользователь ввел правильный *PIN*-код и был авторизован.

Второе высказывание означает, что автомат *AClient* попадет в вершину с номером 11 "*10. Выдача денег*" и выдаст деньги.

Перефразируем данное свойство (точнее, его отрицание) через введенные элементарные высказывания: "В автомате *AClient* не происходило событие *e10* и автомат *AClient* перешел в состояние "*10. Выдача денег*"". Следовательно, была верна формула

```
!{lastEvent == e10},
```

а после этого сразу стала верна формула

```
{stateAClient == 11}.
```

Следовательно, *LTL*-формула для этого свойства будет иметь вид:

```
((!{lastEvent == e10}) U ({stateAClient == 11})).
```

Ожидаемый результат: в отчете о проведенной верификации должно быть сказано, что ошибок нет.

Проведем верификацию. Инструментальное средство *Converter* выдаст следующий отчет:

```
Converter>spin -a models/aclient.ltl

Converter>gcc pan.c -o pan.exe

Converter>pan -n
warning: for p.o. reduction to be valid the never claim must be stutter-
         invariant
(never claims generated from LTL formulae are stutter-invariant)
(Spin Version 4.2.8 -- 6 January 2007)
  + Partial Order Reduction

Full statespace search for:
  never claim          +
  assertion violations  + (if within scope of claim)
  acceptance   cycles   - (not selected)
  invalid end states - (disabled by never claim)

State-vector 32 byte, depth reached 99, errors: 0
      88 states, stored
       8 states, matched
      96 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)

2.622  memory usage (Mbyte)


Converter>spin -t -p models/aclient.ltl
Starting :init: with pid 0
spin: cannot find trail file
```

Строка отчета

```
State-vector 28 byte, depth reached 33, errors: 0
```

говорит о том, что ошибок нет, как и ожидалось.

2.3.2.2. Банкомат выдает деньги только при вставленной карте

Проверяется отсутствие последовательности действий, при которой пользователь получает деньги до введения карты в банкомат (истинное свойство 2).

Запишем отрицание этого свойства: "Пользователь не вставил карту, но получил деньги". Когда пользователь вставляет карту в банкомат, возникает событие *e6*. Таким образом, для формирования *LTL*-формулы требуется элементарное высказывание:

```
lastEvent == e6,
```

которое обозначает, что произошло событие *e6* и элементарное высказывание

```
stateAClient == 11,
```

которое обозначает, что произошла выдача денег.

Таким образом, *LTL*-формула для этого свойства будет иметь вид:

```
((!{lastEvent == e6}) U ({stateAClient == 11})).
```

Ожидаемый результат: в отчете о проведенной верификации должно быть сказано, что ошибок нет.

Проведем верификацию. Инструментальное средство *Converter* выдаст следующий отчет:

```
Converter>spin -a models/aclient.ltl

Converter>gcc pan.c -o pan.exe

Converter>pan -n
warning: for p.o. reduction to be valid the never claim must be stutter-
         invariant
(never claims generated from LTL formulae are stutter-invariant)
(Spin Version 4.2.8 -- 6 January 2007)
  + Partial Order Reduction

Full statespace search for:
  never claim          +
  assertion violations  + (if within scope of claim)
  acceptance   cycles   - (not selected)
  invalid end states - (disabled by never claim)

State-vector 32 byte, depth reached 27, errors: 0
      16 states, stored
       1 states, matched
      17 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)

2.622  memory usage (Mbyte)


Converter>spin -t -p models/aclient.ltl
Starting :init: with pid 0
spin: cannot find trail file
```

Строка отчета

```
State-vector 28 byte, depth reached 33, errors: 0
```

говорит о том, что ошибок нет.

2.3.2.3. Чек не печатается до ввода правильного *PIN*-кода

Проверяется отсутствие последовательности действий, при которой пользователю выдается информация о балансе (печатается чек) до ввода правильного PIN-кода (истинное свойство 4).

Запишем отрицание этого свойства: "Пользователь ввел неправильный *PIN*-код, но чек был напечатан". Так как у банкомата есть два состояния, в которых он печатает чек, то для составления *LTL*-формулы требуется три элементарных высказывания. Первое высказывание "Пользователь ввел правильный *PIN*-код":

```
lastEvent == e10.
```

Второе элементарное высказывание: "Автомат *AClient* попадет в вершину с номером 14 "*7. Печать чека 1*"". Оно записывается следующим образом:

```
stateAClient == 14.
```

Третье элементарное высказывание: "Автомат *AClient* попадает в вершину с номером 8 "*11. Печать чека 2*"". Оно записывается следующим образом:

```
stateAClient == 8.
```

Запишем отрицание данного свойства через элементарные высказывания: "Не произошло события *e10*, но автомат *AClient* попал в состояние 8 или в состояние 14". Тогда *LTL*-формула будет иметь вид:

```
((!{lastEvent == e10}) U ({stateAClient == 14} || {stateAClient == 8})).
```

Ожидаемый результат: в отчете о проведенной верификации должно быть сказано, что ошибок нет.

Проведем верификацию. Инструментальное средство *Converter* выдаст следующий отчет:

```
Converter>spin -a models/aclient.ltl

Converter>gcc pan.c -o pan.exe

Converter>pan -n
warning: for p.o. reduction to be valid the never claim must be stutter-
         invariant
(never claims generated from LTL formulae are stutter-invariant)
(Spin Version 4.2.8 -- 6 January 2007)
  + Partial Order Reduction

Full statespace search for:
  never claim          +
  assertion violations  + (if within scope of claim)
  acceptance   cycles   - (not selected)
  invalid end states - (disabled by never claim)

State-vector 32 byte, depth reached 99, errors: 0
      88 states, stored
       8 states, matched
      96 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)

2.622  memory usage (Mbyte)


Converter>spin -t -p models/aclient.ltl
Starting :init: with pid 0
spin: cannot find trail file
```

Строка отчета

```
State-vector 32 byte, depth reached 99, errors: 0
```

говорит, что ошибок нет.

2.3.2.4. Выдаваемая сумма не превышает остаток на счете

Проверяется отсутствие последовательности действий, при которой пользователь получает большую сумму денег, чем имеющаяся на счете (истинное свойство 5).

Запишем отрицание этого свойства: "Пользователь попытался снять больше денег, чем доступно по карте и произошла выдача денег". Когда пользователь пытается снять больше денег, чем доступно по карте, возникает событие *e14* ("Нет запрашиваемой суммы"). Для *LTL*-формулы потребуются два элементарных высказывания:

`lastEvent == e14` и `stateAClient == 11`.

LTL-формула для данного свойства выглядит следующим образом:

`<>(({lastEvent == e14}) && ({lastEvent == e14} U ({stateAClient == 11}))).`

Эта формула означает следующее: когда-нибудь произойдет событие *e14* и последнее произошедшее событие будет *e14* до тех пор, пока не произойдет выдача денег. В подформуле

`({lastEvent == e14}) && ({lastEvent == e14} U ({stateAClient == 11}))`

требуется конструкция

`({lastEvent == e14}) && ( … )`

для того чтобы событие *e14* обязательно произошло, так как выражение *pUq* будет верно и в том случае, когда элементарное высказывание *p* в пути в модели *Крипке* не встретилось ни разу, а сразу встретилось высказывание *q*, что будет подтверждено ниже.

Ожидаемый результат: в отчете о проведенной верификации должно быть сказано, что ошибок нет.

Проведем верификацию. Инструментальное средство *Converter* выдаст следующий отчет:

```
Converter>spin -a models/aclient.ltl

Converter>gcc pan.c -o pan.exe

Converter>pan -n
warning: for p.o. reduction to be valid the never claim must be stutter-
         invariant
(never claims generated from LTL formulae are stutter-invariant)
(Spin Version 4.2.8 -- 6 January 2007)
  + Partial Order Reduction

Full statespace search for:
  never claim          +
  assertion violations  + (if within scope of claim)
  acceptance   cycles   - (not selected)
  invalid end states - (disabled by never claim)

State-vector 32 byte, depth reached 149, errors: 0
     204 states, stored
      23 states, matched
     227 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)

2.622  memory usage (Mbyte)


Converter>spin -t -p models/aclient.ltl
Starting :init: with pid 0
spin: cannot find trail file
```

Строка отчета

```
State-vector 32 byte, depth reached 149, errors: 0
```

говорит о том, что ошибок нет.

Проведем теперь верификацию банкомата с формулой

```
<>({lastEvent == e14} U {stateAClient == 11}).
```

В отчете о проведенной верификации должно быть сообщение об ошибке и выведен контрпример: трасса пути по автомату *AClient,* как показано на рис. 48.

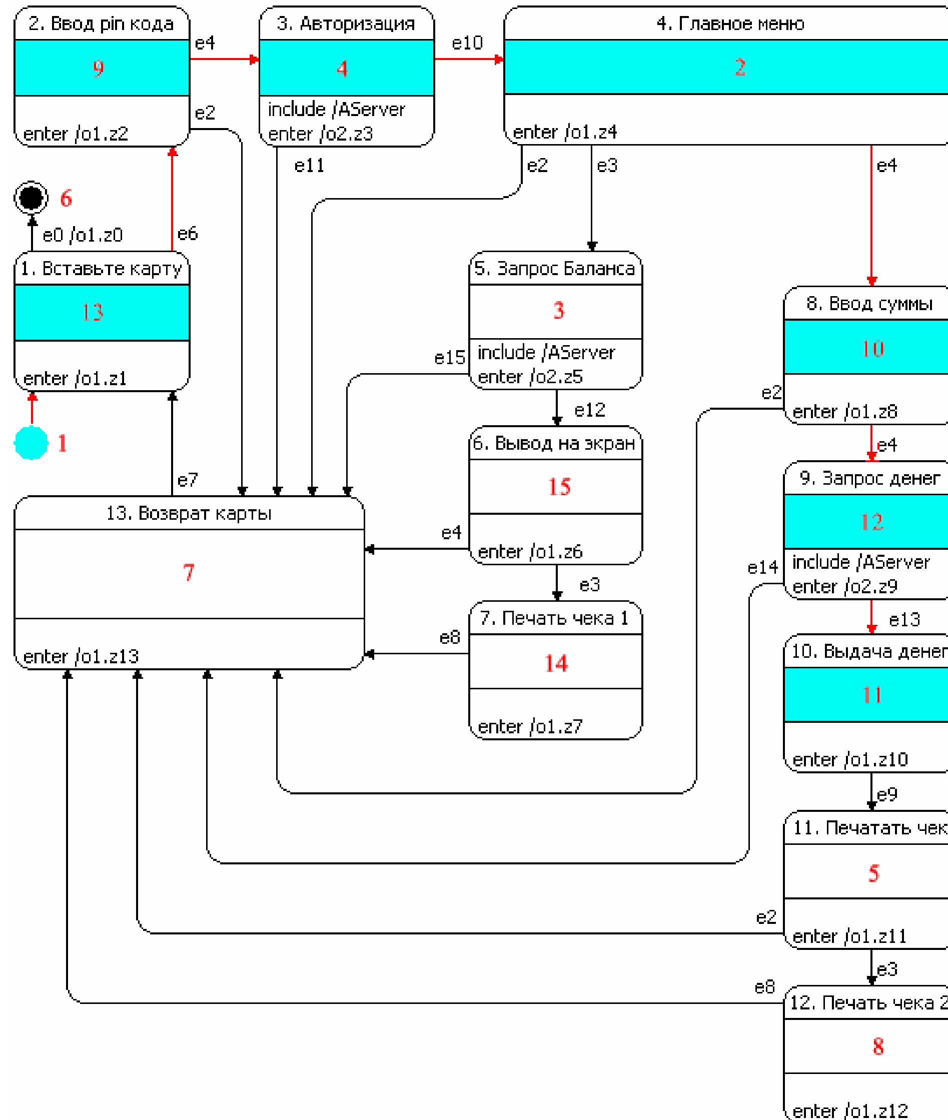


Рис. 48. Трасса ошибки к формуле `<>({lastEvent == e14} U {stateAClient == 11})`

Проведем верификацию. Инструментальное средство *Converter* выдаст следующий отчет:

```
Converter>spin -a models/aclient.ltl

Converter>gcc pan.c -o pan.exe

Converter>pan -n
warning: for p.o. reduction to be valid the never claim must be stutter-
           invariant
(never claims generated from LTL formulae are stutter-invariant)
pan: claim violated! (at depth 127)
pan: wrote models/aclient.ltl.trail
(Spin Version 4.2.8 -- 6 January 2007)
Warning: Search not completed
   + Partial Order Reduction

Full statespace search for:
   never claim           +
   assertion violations  + (if within scope of claim)
   acceptance   cycles   - (not selected)
   invalid end states - (disabled by never claim)
```

```
State-vector 28 byte, depth reached 133, errors: 1
     135 states, stored
       9 states, matched
     144 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)

2.622  memory usage (Mbyte)


Converter>spin -t -p models/aclient.ltl
Starting :init: with pid 0
Starting :never: with pid 1
Never claim moves to line 204  [(1)]
Starting Model with pid 2
  2: proc  0 (:init:) line 198 "models/aclient.ltl" (state 1)      [(run
         Model())]
  4: proc  1 (Model) line  24 "models/aclient.ltl" (state 1) [stateAClient
         = 1]
  6: proc  1 (Model) line  26 "models/aclient.ltl" (state 2)
            [((stateAClient==1))]
              State 1 : s1
  8: proc  1 (Model) line  27 "models/aclient.ltl" (state 3)
            [printf('State 1 : s1\\n')]
 10: proc  1 (Model) line  29 "models/aclient.ltl" (state 4) [stateAClient
         = 13]
 12: proc  1 (Model) line 113 "models/aclient.ltl" (state 224)
            [((stateAClient==13))]
              State 13 : 1. Вставьте карту
 14: proc  1 (Model) line 114 "models/aclient.ltl" (state 225)
            [printf('State 13 : 1. Вставьте карту\\n')]
 16: proc  1 (Model) line 118 "models/aclient.ltl" (state 228)
            [stateAClient = 9]
 18: proc  1 (Model) line 119 "models/aclient.ltl" (state 229)
            [lastEvent = 6]
 20: proc  1 (Model) line  82 "models/aclient.ltl" (state 148)
            [((stateAClient==9))]
              State 9 : 2. Ввод pin кода
 22: proc  1 (Model) line  83 "models/aclient.ltl" (state 149)
            [printf('State 9 : 2. Ввод pin кода\\n')]
 24: proc  1 (Model) line  85 "models/aclient.ltl" (state 150)
            [stateAClient = 4]
 26: proc  1 (Model) line  86 "models/aclient.ltl" (state 151)
            [lastEvent = 4]
 28: proc  1 (Model) line  50 "models/aclient.ltl" (state 71)
            [((stateAClient==4))]
              State 4 : 3. Авторизация
 30: proc  1 (Model) line  51 "models/aclient.ltl" (state 72)
            [printf('State 4 : 3. Авторизация\\n')]
 32: proc  1 (Model) line 148 "models/aclient.ltl" (state 73)
            [stateAServer = 17]
 34: proc  1 (Model) line 150 "models/aclient.ltl" (state 74)
            [((stateAServer==17))]
              State 17 : s1
 36: proc  1 (Model) line 151 "models/aclient.ltl" (state 75)
            [printf('State 17 : s1\\n')]
 38: proc  1 (Model) line 153 "models/aclient.ltl" (state 76)
            [stateAServer = 23]
 40: proc  1 (Model) line 181 "models/aclient.ltl" (state 105)
            [((stateAServer==23))]
              State 23 : Чтение запроса
```

```
42: proc  1 (Model) line 182 "models/aclient.ltl" (state 106)
          [printf('State 23 : Чтение запроса\\n')]
44: proc  1 (Model) line 184 "models/aclient.ltl" (state 107)
          [stateAServer = 20]
46: proc  1 (Model) line 185 "models/aclient.ltl" (state 108)
          [lastEvent = 23]
48: proc  1 (Model) line 163 "models/aclient.ltl" (state 87)
          [((stateAServer==20))]
            State 20 : Снятие денег
50: proc  1 (Model) line 164 "models/aclient.ltl" (state 88)
          [printf('State 20 : Снятие денег\\n')]
52: proc  1 (Model) line 166 "models/aclient.ltl" (state 89)
          [stateAServer = 18]
54: proc  1 (Model) line 167 "models/aclient.ltl" (state 90)
          [lastEvent = 24]
56: proc  1 (Model) line 155 "models/aclient.ltl" (state 79)
          [((stateAServer==18))]
            State 18 : Ответ клиенту
58: proc  1 (Model) line 156 "models/aclient.ltl" (state 80)
          [printf('State 18 : Ответ клиенту\\n')]
60: proc  1 (Model) line 158 "models/aclient.ltl" (state 81)
          [stateAServer = 19]
62: proc  1 (Model) line 160 "models/aclient.ltl" (state 84)
          [((stateAServer==19))]
            State 19 : s2
64: proc  1 (Model) line 161 "models/aclient.ltl" (state 85)
          [printf('State 19 : s2\\n')]
66: proc  1 (Model) line  54 "models/aclient.ltl" (state 119)
          [stateAClient = 2]
68: proc  1 (Model) line  55 "models/aclient.ltl" (state 120)
          [lastEvent = 10]
70: proc  1 (Model) line  31 "models/aclient.ltl" (state 7)
          [((stateAClient==2))]
            State 2 : 4. Главное меню
72: proc  1 (Model) line  32 "models/aclient.ltl" (state 8)
          [printf('State 2 : 4. Главное меню\\n')]
74: proc  1 (Model) line  38 "models/aclient.ltl" (state 13)
          [stateAClient = 10]
76: proc  1 (Model) line  39 "models/aclient.ltl" (state 14)
          [lastEvent = 4]
78: proc  1 (Model) line  90 "models/aclient.ltl" (state 156)
          [((stateAClient==10))]
            State 10 : 8. Ввод суммы
80: proc  1 (Model) line  91 "models/aclient.ltl" (state 157)
          [printf('State 10 : 8. Ввод суммы\\n')]
82: proc  1 (Model) line  93 "models/aclient.ltl" (state 158)
          [stateAClient = 12]
84: proc  1 (Model) line  94 "models/aclient.ltl" (state 159)
          [lastEvent = 4]
86: proc  1 (Model) line 104 "models/aclient.ltl" (state 170)
          [((stateAClient==12))]
            State 12 : 9. Запрос денег
88: proc  1 (Model) line 105 "models/aclient.ltl" (state 171)
          [printf('State 12 : 9. Запрос денег\\n')]
90: proc  1 (Model) line 148 "models/aclient.ltl" (state 172)
          [stateAServer = 17]
92: proc  1 (Model) line 150 "models/aclient.ltl" (state 173)
          [((stateAServer==17))]
            State 17 : s1
94: proc  1 (Model) line 151 "models/aclient.ltl" (state 174)
          [printf('State 17 : s1\\n')]
96: proc  1 (Model) line 153 "models/aclient.ltl" (state 175)
          [stateAServer = 23]
```

```
 98: proc  1 (Model) line 181 "models/aclient.ltl" (state 204)
              [((stateAServer==23))]
                State 23 : Чтение запроса
100: proc  1 (Model) line 182 "models/aclient.ltl" (state 205)
              [printf('State 23 : Чтение запроса\\n')]
102: proc  1 (Model) line 184 "models/aclient.ltl" (state 206)
              [stateAServer = 20]
104: proc  1 (Model) line 185 "models/aclient.ltl" (state 207)
              [lastEvent = 23]
106: proc  1 (Model) line 163 "models/aclient.ltl" (state 186)
              [((stateAServer==20))]
                State 20 : Снятие денег
108: proc  1 (Model) line 164 "models/aclient.ltl" (state 187)
              [printf('State 20 : Снятие денег\\n')]
110: proc  1 (Model) line 166 "models/aclient.ltl" (state 188)
              [stateAServer = 18]
112: proc  1 (Model) line 167 "models/aclient.ltl" (state 189)
              [lastEvent = 24]
114: proc  1 (Model) line 155 "models/aclient.ltl" (state 178)
              [((stateAServer==18))]
                State 18 : Ответ клиенту
116: proc  1 (Model) line 156 "models/aclient.ltl" (state 179)
              [printf('State 18 : Ответ клиенту\\n')]
118: proc  1 (Model) line 158 "models/aclient.ltl" (state 180)
              [stateAServer = 19]
120: proc  1 (Model) line 160 "models/aclient.ltl" (state 183)
              [((stateAServer==19))]
                State 19 : s2
122: proc  1 (Model) line 161 "models/aclient.ltl" (state 184)
              [printf('State 19 : s2\\n')]
124: proc  1 (Model) line 108 "models/aclient.ltl" (state 218)
              [stateAClient = 11]
Never claim moves to line 203 [((stateAClient==11))]
126: proc  1 (Model) line 109 "models/aclient.ltl" (state 219)
              [lastEvent = 13]
Never claim moves to line 207 [(1)]
spin: trail ends after 128 steps
#processes: 2
     lastEvent = 13
     stateAClient = 11
     stateAServer = 19
128: proc  1 (Model) line  25 "models/aclient.ltl" (state 246)
128: proc  0 (:init:) line 199 "models/aclient.ltl" (state 2) <valid end
         state>
128: proc  - (:never:) line 208 "models/aclient.ltl" (state 8) <valid end
         state>
2 processes created
```

Строка отчета

```
State-vector 28 byte, depth reached 133, errors: 1
```

говорит о том, что была найдена ошибка.

Начиная со строки

```
Starting :init: with pid 0,
```

приводится контрпример. Выпишем контрпример в явном виде:

Автомат *Aclient* перешел в состояние *s1* (начальное состояние этого автомата).

Автомат *Aclient* перешел в состояние *1. Вставьте карту*.

Произошло событие *e6.*

Автомат *Aclient* перешел в состояние *2. Ввод Pin кода*.

Произошло событие *e4.*

Автомат *Aclient* перешел в состояние *3. Авторизация*.

Автомат *AServer* перешел в состояние *s1* (начальное состояние этого автомата).
Автомат *AServer* перешел в состояние *Чтение запроса*.
Произошло событие *e23.*
Автомат *AServer* перешел в состояние *Снятие денег*.
Произошло событие *e24.*
Автомат *AServer* перешел в состояние *Ответ клиенту*.
Автомат *AServer* перешел в состояние *s2* (конечное состояние этого автомата).
Произошло событие *e10.*
Автомат *Aclient* перешел в состояние *4. Главное меню*.
Произошло событие *e4.*
Автомат *Aclient* перешел в состояние *8. Ввод суммы*.
Произошло событие *e4.*
Автомат *Aclient* перешел в состояние *9. Запрос денег.*
Автомат *AServer* перешел в состояние *s1* (конечное состояние этого автомата).
Автомат *AServer* перешел в состояние *Чтение запроса*.
Произошло событие *e23.*
Автомат *AServer* перешел в состояние *Снятие денег*.
Произошло событие *e24.*
Автомат *AServer* перешел в состояние *Ответ клиенту*.
Автомат *AServer* перешел в состояние *s2* (конечное состояние этого автомата).
Произошло событие *e13.*
Автомат *Aclient* перешел в состояние *10. Выдача денег.*

Таким образом, путь в автомате *AClient* соответствует рис. 48. Заметим, что когда автомат *AClient* перешел в состояние "*3. Авторизация*", путь в автомате *AServer* должен был быть таким, как показано на рис. 49.

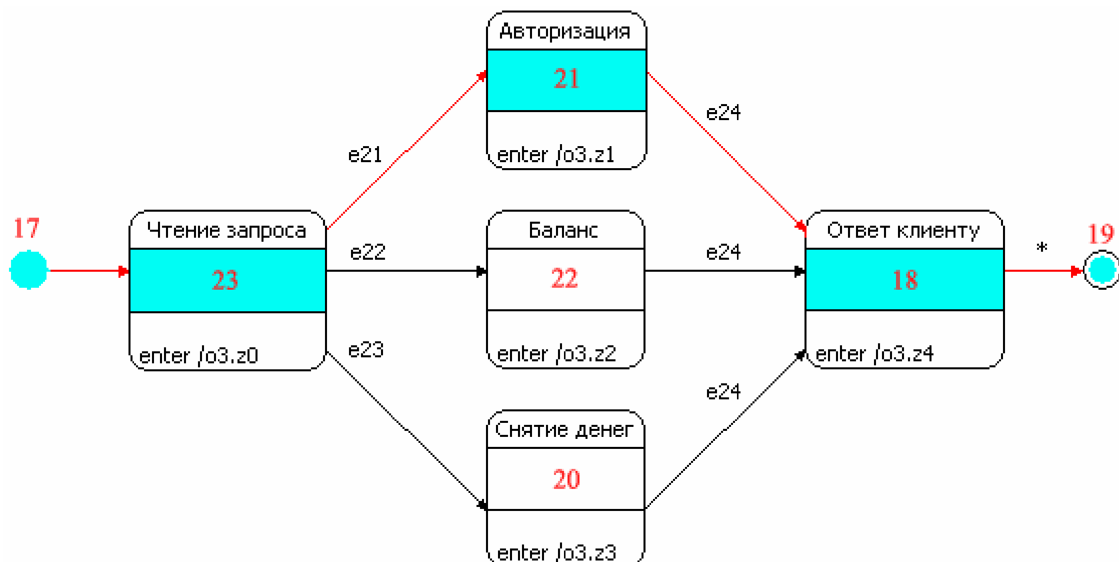


Рис. 49. Правильный путь в автомате AServer

Однако, в контрпримере путь такой, как показано на рис. 50.

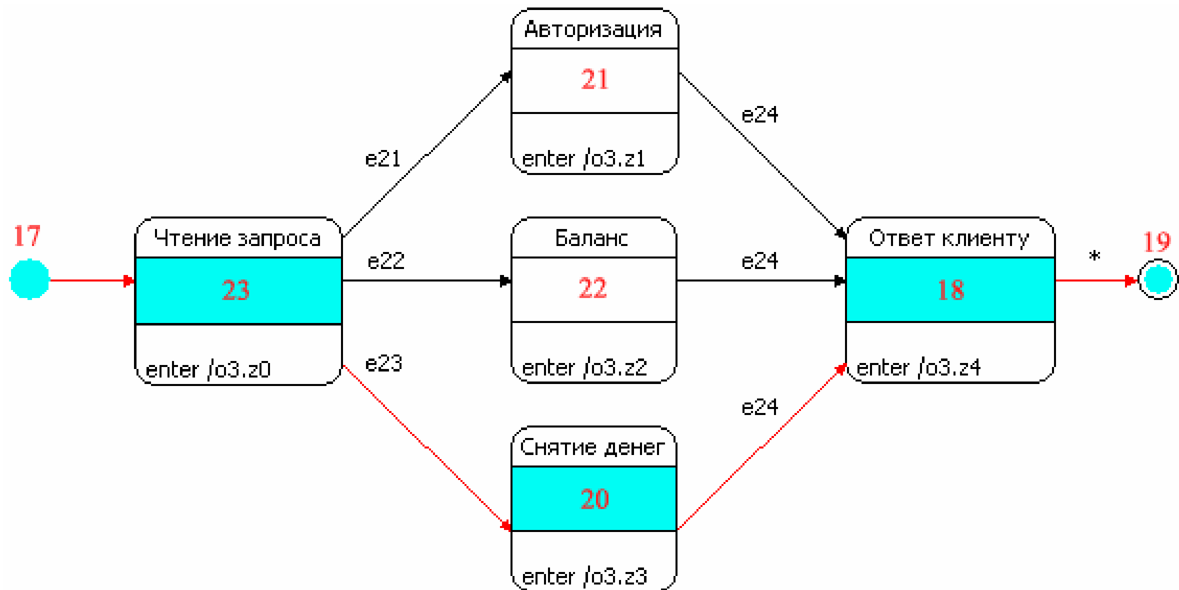


Рис. 50. Реальный путь в автомате AServer

Это связано с тем, что в построенной модели не содержится логика, которая выбирает, какое событие должно произойти в состоянии "*Чтение запроса*" автомата *AServer*.

2.3.2.5. Выдача денег без запроса

Проверяется, что деньги не выдаются, пока пользователь не сделает соответствующий запрос (аналог ложного свойства 2).

Запишем отрицание этого свойства: "Пользователь не делал запроса на выдачу денег, но банкомат стал выдавать деньги". Пользователь сделал запрос на выдачу денег, когда в состоянии "*8. Ввод суммы*" автомата *AClient* произошло событие *e4* ("Нажали кнопку 4"). Тогда формула, соответствующая запросу на выдачу денег будет выглядеть следующим образом:

```
{stateAClient == 10} && {lastEvent == e4}.
```

Кроме того, для построения *LTL*-формулы требуется элементарное высказывание, соответствующее выдаче денег банкоматом:

```
stateAClient == 11.
```

Получаем следующую *LTL*-формулу:

```
(!({stateAClient == 10} && {lastEvent == e4})) U {stateAClient == 11}.
```

Ожидаемый результат: в отчете о проведенной верификации должно быть сказано, что ошибок нет.

Проведем верификацию. Инструментальное средство *Converter* выдаст следующий отчет:

```
Converter>spin -a models/aclient.ltl

Converter>gcc pan.c -o pan.exe

Converter>pan -n
warning: for p.o. reduction to be valid the never claim must be stutter-
         invariant
(never claims generated from LTL formulae are stutter-invariant)
(Spin Version 4.2.8 -- 6 January 2007)
   + Partial Order Reduction

Full statespace search for:
   never claim          +
```

```
  assertion violations  + (if within scope of claim)
  acceptance   cycles   - (not selected)
  invalid end states - (disabled by never claim)

State-vector 32 byte, depth reached 141, errors: 0
     147 states, stored
      14 states, matched
     161 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)

2.622  memory usage (Mbyte)


Converter>spin -t -p models/aclient.ltl
Starting :init: with pid 0
spin: cannot find trail file
```

Строка отчета

```
State-vector 28 byte, depth reached 33, errors: 0
```

говорит о том, что ошибок нет.

2.3.2.6. Возврат карты при ошибке

Верифицируемое свойство: "Если произойдёт ошибка, карта будет возвращена". Запишем отрицание этого свойства: "Если произойдет ошибка, карта не будет возвращена". Если происходит ошибка, вызывается событие *e15* ("Ошибка при работе с сервером"). Следовательно, элементарное высказывание, соответствующее возникновению ошибки, записывается следующим образом:

```
lastEvent == e15.
```

Банкомат возвращает карту в состоянии "*13.Возврат карты*". Следовательно, элементарное высказывание, соответствующее возврату карты, записывается следующим образом:

```
stateAClient == 7.
```

"Карта не будет возвращена" записывается на языке *LTL* следующим образом:

```
[](!{stateAClient == 7}).
```

Таким образом, получаем следующую *LTL*-формулу:

```
<>({lastEvent == e15} && ({lastEvent == e15} U ([](!{stateAClient == 7})))).
```

Ожидаемый результат: в отчете о проведенной верификации должно быть сказано, что ошибок нет.

Проведем верификацию. Инструментальное средство *Converter* выдаст следующий отчет:

```
Converter>spin -a models/aclient.ltl

Converter>gcc pan.c -o pan.exe

Converter>pan -n
warning: for p.o. reduction to be valid the never claim must be stutter-
          invariant
(never claims generated from LTL formulae are stutter-invariant)
(Spin Version 4.2.8 -- 6 January 2007)
  + Partial Order Reduction

Full statespace search for:
  never claim          +
  assertion violations  + (if within scope of claim)
  acceptance   cycles   - (not selected)
  invalid end states - (disabled by never claim)
```

```
State-vector 32 byte, depth reached 264, errors: 0
     348 states, stored
      32 states, matched
     380 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)

2.622  memory usage (Mbyte)


Converter>spin -t -p models/aclient.ltl
Starting :init: with pid 0
spin: cannot find trail file
```

Строка отчета

```
State-vector 32 byte, depth reached 264, errors: 0
```

говорит о том, что ошибок нет.

2.3.2.7. Выдача денег

Проверяется, что банкомат не выдает деньги (ложное свойство).

Запишем отрицание этого свойства: "Банкомат когда-нибудь выдаст деньги". Сформулируем его на языке *LTL*:

```
<>({stateAClient == 11}).
```

Ожидаемый результат: в отчете о проведенной верификации должно быть сообщение об ошибке и выведен контрпример. В этом контрпримере должен быть путь по состояниям автомата *AClient* до состояния "*10. Выдача денег*".

Проведем верификацию. Инструментальное средство *Converter* выдаст следующий отчет:

```
Converter>spin -a models/aclient.ltl

Converter>gcc pan.c -o pan.exe

Converter>pan -n
warning: for p.o. reduction to be valid the never claim must be stutter-
         invariant
(never claims generated from LTL formulae are stutter-invariant)
pan: claim violated! (at depth 127)
pan: wrote models/aclient.ltl.trail
(Spin Version 4.2.8 -- 6 January 2007)
Warning: Search not completed
   + Partial Order Reduction

Full statespace search for:
   never claim          +
   assertion violations  + (if within scope of claim)
   acceptance   cycles   - (not selected)
   invalid end states - (disabled by never claim)

State-vector 28 byte, depth reached 133, errors: 1
     135 states, stored
       9 states, matched
     144 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)

2.622  memory usage (Mbyte)
```

```
Converter>spin -t -p models/aclient.ltl
Starting :init: with pid 0
Starting :never: with pid 1
Never claim moves to line 203 [(1)]
Starting Model with pid 2
  2: proc  0 (:init:) line 197 "models/aclient.ltl" (state 1)      [(run
         Model())]
  4: proc  1 (Model) line  23 "models/aclient.ltl" (state 1) [stateAClient
         = 1]
  6: proc  1 (Model) line  25 "models/aclient.ltl" (state 2)
           [((stateAClient==1))]
             State 1 : s1
  8: proc  1 (Model) line  26 "models/aclient.ltl" (state 3)
           [printf('State 1 : s1\\n')]
 10: proc  1 (Model) line  28 "models/aclient.ltl" (state 4) [stateAClient
         = 13]
 12: proc  1 (Model) line 112 "models/aclient.ltl" (state 224)
           [((stateAClient==13))]
             State 13 : 1. Вставьте карту
 14: proc  1 (Model) line 113 "models/aclient.ltl" (state 225)
           [printf('State 13 : 1. Вставьте карту\\n')]
 16: proc  1 (Model) line 117 "models/aclient.ltl" (state 228)
           [stateAClient = 9]
 18: proc  1 (Model) line 118 "models/aclient.ltl" (state 229)
           [lastEvent = 6]
 20: proc  1 (Model) line  81 "models/aclient.ltl" (state 148)
           [((stateAClient==9))]
             State 9 : 2. Ввод pin кода
 22: proc  1 (Model) line  82 "models/aclient.ltl" (state 149)
           [printf('State 9 : 2. Ввод pin кода\\n')]
 24: proc  1 (Model) line  84 "models/aclient.ltl" (state 150)
           [stateAClient = 4]
 26: proc  1 (Model) line  85 "models/aclient.ltl" (state 151)
           [lastEvent = 4]
 28: proc  1 (Model) line  49 "models/aclient.ltl" (state 71)
           [((stateAClient==4))]
             State 4 : 3. Авторизация
 30: proc  1 (Model) line  50 "models/aclient.ltl" (state 72)
           [printf('State 4 : 3. Авторизация\\n')]
 32: proc  1 (Model) line 147 "models/aclient.ltl" (state 73)
           [stateAServer = 17]
 34: proc  1 (Model) line 149 "models/aclient.ltl" (state 74)
           [((stateAServer==17))]
             State 17 : s1
 36: proc  1 (Model) line 150 "models/aclient.ltl" (state 75)
           [printf('State 17 : s1\\n')]
 38: proc  1 (Model) line 152 "models/aclient.ltl" (state 76)
           [stateAServer = 23]
 40: proc  1 (Model) line 180 "models/aclient.ltl" (state 105)
           [((stateAServer==23))]
             State 23 : Чтение запроса
 42: proc  1 (Model) line 181 "models/aclient.ltl" (state 106)
           [printf('State 23 : Чтение запроса\\n')]
 44: proc  1 (Model) line 183 "models/aclient.ltl" (state 107)
           [stateAServer = 20]
 46: proc  1 (Model) line 184 "models/aclient.ltl" (state 108)
           [lastEvent = 23]
 48: proc  1 (Model) line 162 "models/aclient.ltl" (state 87)
           [((stateAServer==20))]
             State 20 : Снятие денег
 50: proc  1 (Model) line 163 "models/aclient.ltl" (state 88)
           [printf('State 20 : Снятие денег\\n')]
```

```
 52: proc  1 (Model) line 165 "models/aclient.ltl" (state 89)
           [stateAServer = 18]
 54: proc  1 (Model) line 166 "models/aclient.ltl" (state 90)
           [lastEvent = 24]
 56: proc  1 (Model) line 154 "models/aclient.ltl" (state 79)
           [((stateAServer==18))]
             State 18 : Ответ клиенту
 58: proc  1 (Model) line 155 "models/aclient.ltl" (state 80)
           [printf('State 18 : Ответ клиенту\\n')]
 60: proc  1 (Model) line 157 "models/aclient.ltl" (state 81)
           [stateAServer = 19]
 62: proc  1 (Model) line 159 "models/aclient.ltl" (state 84)
           [((stateAServer==19))]
             State 19 : s2
 64: proc  1 (Model) line 160 "models/aclient.ltl" (state 85)
           [printf('State 19 : s2\\n')]
 66: proc  1 (Model) line  53 "models/aclient.ltl" (state 119)
           [stateAClient = 2]
 68: proc  1 (Model) line  54 "models/aclient.ltl" (state 120)
           [lastEvent = 10]
 70: proc  1 (Model) line  30 "models/aclient.ltl" (state 7)
           [((stateAClient==2))]
             State 2 : 4. Главное меню
 72: proc  1 (Model) line  31 "models/aclient.ltl" (state 8)
           [printf('State 2 : 4. Главное меню\\n')]
 74: proc  1 (Model) line  37 "models/aclient.ltl" (state 13)
           [stateAClient = 10]
 76: proc  1 (Model) line  38 "models/aclient.ltl" (state 14)
           [lastEvent = 4]
 78: proc  1 (Model) line  89 "models/aclient.ltl" (state 156)
           [((stateAClient==10))]
             State 10 : 8. Ввод суммы
 80: proc  1 (Model) line  90 "models/aclient.ltl" (state 157)
           [printf('State 10 : 8. Ввод суммы\\n')]
 82: proc  1 (Model) line  92 "models/aclient.ltl" (state 158)
           [stateAClient = 12]
 84: proc  1 (Model) line  93 "models/aclient.ltl" (state 159)
           [lastEvent = 4]
 86: proc  1 (Model) line 103 "models/aclient.ltl" (state 170)
           [((stateAClient==12))]
             State 12 : 9. Запрос денег
 88: proc  1 (Model) line 104 "models/aclient.ltl" (state 171)
           [printf('State 12 : 9. Запрос денег\\n')]
 90: proc  1 (Model) line 147 "models/aclient.ltl" (state 172)
           [stateAServer = 17]
 92: proc  1 (Model) line 149 "models/aclient.ltl" (state 173)
           [((stateAServer==17))]
             State 17 : s1
 94: proc  1 (Model) line 150 "models/aclient.ltl" (state 174)
           [printf('State 17 : s1\\n')]
 96: proc  1 (Model) line 152 "models/aclient.ltl" (state 175)
           [stateAServer = 23]
 98: proc  1 (Model) line 180 "models/aclient.ltl" (state 204)
           [((stateAServer==23))]
             State 23 : Чтение запроса
100: proc  1 (Model) line 181 "models/aclient.ltl" (state 205)
           [printf('State 23 : Чтение запроса\\n')]
102: proc  1 (Model) line 183 "models/aclient.ltl" (state 206)
           [stateAServer = 20]
104: proc  1 (Model) line 184 "models/aclient.ltl" (state 207)
           [lastEvent = 23]
106: proc  1 (Model) line 162 "models/aclient.ltl" (state 186)
           [((stateAServer==20))]
```

```
                State 20 : Снятие денег
108: proc  1 (Model) line 163 "models/aclient.ltl" (state 187)
               [printf('State 20 : Снятие денег\\n')]
110: proc  1 (Model) line 165 "models/aclient.ltl" (state 188)
               [stateAServer = 18]
112: proc  1 (Model) line 166 "models/aclient.ltl" (state 189)
               [lastEvent = 24]
114: proc  1 (Model) line 154 "models/aclient.ltl" (state 178)
               [((stateAServer==18))]
                State 18 : Ответ клиенту
116: proc  1 (Model) line 155 "models/aclient.ltl" (state 179)
               [printf('State 18 : Ответ клиенту\\n')]
118: proc  1 (Model) line 157 "models/aclient.ltl" (state 180)
               [stateAServer = 19]
120: proc  1 (Model) line 159 "models/aclient.ltl" (state 183)
               [((stateAServer==19))]
                State 19 : s2
122: proc  1 (Model) line 160 "models/aclient.ltl" (state 184)
               [printf('State 19 : s2\\n')]
124: proc  1 (Model) line 107 "models/aclient.ltl" (state 218)
               [stateAClient = 11]
Never claim moves to line 202 [((stateAClient==11))]
126: proc  1 (Model) line 108 "models/aclient.ltl" (state 219)
               [lastEvent = 13]
Never claim moves to line 206 [(1)]
spin: trail ends after 128 steps
#processes: 2
      lastEvent = 13
      stateAClient = 11
      stateAServer = 19
128: proc  1 (Model) line  24 "models/aclient.ltl" (state 246)
128: proc  0 (:init:) line 198 "models/aclient.ltl" (state 2) <valid end
          state>
128: proc  - (:never:) line 207 "models/aclient.ltl" (state 8) <valid end
          state>
2 processes created
```

Строка отчета

```
State-vector 28 byte, depth reached 133, errors: 1
```

говорит о том, что была найдена ошибка. Выпишем контрпример в явном виде:

Автомат *Aclient* перешел в состояние *s1*.
Автомат *Aclient* перешел в состояние *1. Вставьте карту*.
Произошло событие *e6*.
Автомат *Aclient* перешел в состояние *2. Ввод Pin кода*.
Произошло событие *e4*.
Автомат *Aclient* перешел в состояние *3. Авторизация*.
Автомат *AServer* перешел в состояние *s1*.
Автомат *AServer* перешел в состояние *Чтение запроса*.
Произошло событие *e23*.
Автомат *AServer* перешел в состояние *Снятие денег*.
Произошло событие *e24*.
Автомат *AServer* перешел в состояние *Ответ клиенту*.
Автомат *AServer* перешел в состояние *s2*.
Произошло событие *e10*.
Автомат *Aclient* перешел в состояние *4. Главное меню*.
Произошло событие *e4*.
Автомат *Aclient* перешел в состояние *8. Ввод суммы*.
Произошло событие *e4*.

Автомат *Aclient* перешел в состояние *9. Запрос денег*.
Автомат *AServer* перешел в состояние *s1*.
Автомат *AServer* перешел в состояние *Чтение запроса*.
Произошло событие *e23*.
Автомат *AServer* перешел в состояние *Снятие денег*.
Произошло событие *e24*.
Автомат *AServer* перешел в состояние *Ответ клиенту*.
Автомат *AServer* перешел в состояние *s2*.
Произошло событие *e13*.
Автомат *Aclient* перешел в состояние *10. Выдача денег*.

Как и ожидалось, контрпример содержит путь по состояниям автомата *AClient* до состояния "*10. Выдача денег*".

2.4. Верификация при помощи *UniMod.verifier*

2.4.1. Описание метода

2.4.1.1. Общее описание

Инструментальное средство *UniMod.verifier* предназначен для верификации автоматных программ. В ходе верификации программы возникает необходимость осуществлять следующие действия.

1. Вычислять глобальное состояние программы. Глобальное состояние должно однозначно определять поведение программы
2. Совершать элементарный шаг программы. Элементарный шаг является переходом программы из одного глобального состояния в другое без посещения иных глобальных состояний.
3. Откатывать элементарный шаг 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.

Инструментальное средство *UniMod.verifier* использует *метод эмуляции* [1] ля выполнения перечисленных действий. При его использовании верификация совершается путем управляемого эмулирования работы автоматной программы с одновременным наблюдением за верифицируемыми свойствами программы. Такой подход позволяет напрямую работать с верифицируемой программой, без дополнительных преобразований над ней.

В методе эмуляции пять действий, необходимых при верификации, осуществляются следующим образом.

Глобальное состояние программы складывается из набора текущих состояний каждого автомата.

Элементарный шаг работы программы – это обработка системой автоматов событий. В результате обработки может смениться набор состояний автоматов.

Поведение автоматной программы определяется набором состояний автоматов, поэтому для отката достаточно вернуть автоматы в те состояния, в которых они выполнялись до выполнения шага.

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 выражении которого участвуют переменные объектов управления, такое условие принимается равным `True` или `False`, и оба они используются для создания различных историй работы программы.

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`. Тоже, что `(isInState(sm, s) && !wasInState(sm, s))`;
- `cameToFinalState()` – возвращает `True`, если после совершения шага корневой автомат модели вошел в свое 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 было встречено условие `g`, и его значение было определено как `False`.

2.4.1.2. Особенности преобразований

Особенностью метода эмуляции является то, что он не требует дополнительных преобразований над автоматной программой или над контрпримером, сгенерированным верификатором. Например, в работе [1] верификация системы автоматов производится с использованием преобразований.

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 сценарий ошибки в терминах входного языка *SPIN*.
4. Сценарий ошибки переводится в термины исходной системы автоматов.

Эта схема верификации изображена на рис. 51.

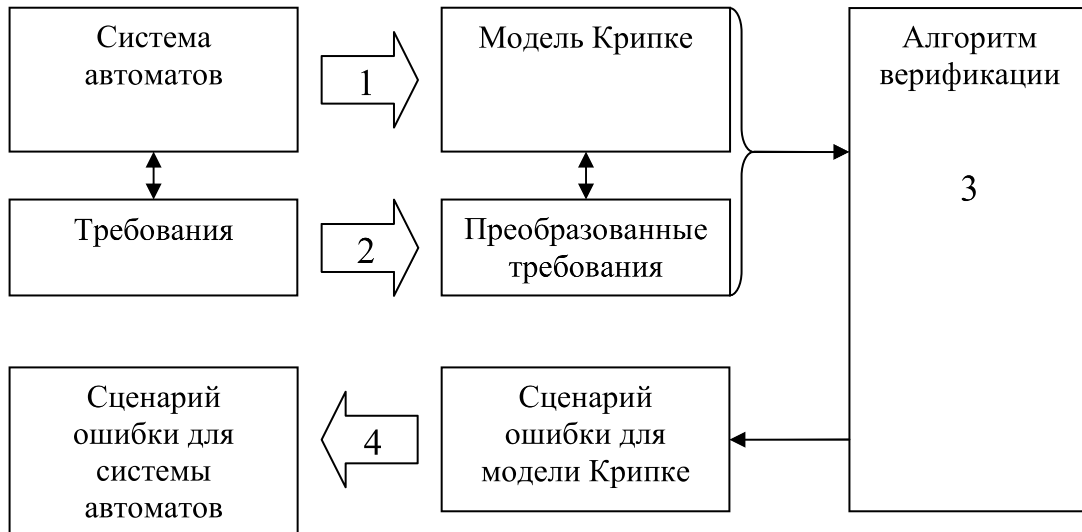


Рис. 51. Схема верификации с явным построением модели Крипке

В методе же эмуляции не строится явно модель Крипке, и 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. 52.

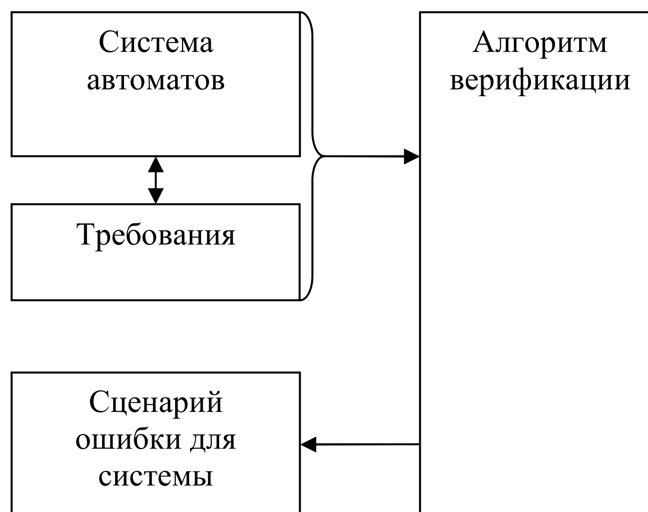


Рис. 52. Схема верификации методом эмуляции

Поскольку в методе эмуляции обработка событий в системе автоматов происходит атомарно с точки зрения верификации, возникают некоторое ограничение относительно темпоральной формулы, формулирующей требования к системе. Ограничение заключается в том, что если в требовании важен порядок выполнения предикатов, то их значения не должны изменяться одновременно в пределах одной обработки события.

2.4.1.3. Связь с верификатором *Bogor*

Инструментальное средство *UniMod.verifier* основано на верификаторе *Bogor* [30, 31] – верификаторе с модульной структурой и расширяемым входным языком, который называется *BIR* — *Bogor Input Language*. Благодаря расширяемости входного языка верификатора *Bogor* появилась возможность интеграции верификатора с инструментом для создания и запуска автоматных программ *UniMod*. Таким образом, было создано единое средство для создания, запуска и верификации автоматных программ.

Входной язык верификатора *Bogor* был расширен новым типом – моделью системы автоматов. Над этой моделью можно совершить лишь одно действие («step» – «совершить один элементарный шаг работы системы автоматов»), то есть выбрать очередное событие и обработать его в системе автоматов. Также у модели можно запросить многочисленные свойства, соответствующие предикатам, как например, wasEvent, isInState и т.д. Созданный тип был назван AutomataModel. Его объявление на языке *BIR* выглядит следующим образом:

```
extension AutomataModel for
    com.UniMod.verifier.Bogorextension.AutomataModelModule
{
    typedef type;

    expdef AutomataModel.type create();

    expdef boolean wasEvent(AutomataModel.type model, string event);
    expdef boolean wasInState(AutomataModel.type model, string
         stateMachine, string state);
    expdef boolean isInState(AutomataModel.type model, string
         stateMachine, string state);
    expdef boolean cameToState(AutomataModel.type model, string
         stateMachine, string state);
    expdef boolean cameToFinalState(AutomataModel.type model);
    expdef boolean wasAction(AutomataModel.type model, string action);
    expdef boolean wasFirstAction(AutomataModel.type model, string
         action);
    expdef boolean wasLastAction(AutomataModel.type model, string
         action);
    expdef int getActionIndex(AutomataModel.type model, string action);
    expdef boolean wasTrue(AutomataModel.type model, string guard);
    expdef boolean wasFalse(AutomataModel.type model, string guard);

    actiondef step(AutomataModel.type model);
}
```

Во многих предикатах участвует имя автомата. Поскольку в системе автоматов один автомат может быть вложен в несколько состояний своего родителя, то для идентификации автомата вводится понятие *путь автомата*, который строится по следующим правилам:

- путь корневого автомата имеет формат "/<название корневого автомата>";
- путь вложенных автоматов имеет формат "<путь родительского автомата>:<состояние родительского автомата>/<название вложенного автомата>".

Верификатор *Bogor* верифицирует программы, написанные на языке *BIR*. За счет создания специального типа, программа для верификации системы автоматов весьма проста:

```
AutomataModel.type model;

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

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

Программа состоит из двух состояний. В первом из них (`init`) создается новая система автоматов. Реально при этом из указанного отдельно файла создается *UniMod*-модель автоматной программы. Второе состояние (`loop`) бесконечный цикл, в котором постоянно выполняется шаг работы системы автоматов. Стоит заметить, что хотя цикл бесконечный и не имеет выхода. Это не значит, что программа зависнет. Программа для верификатора – это не реально исполняющаяся программа, а модель, которая подвергается верификации. Когда автоматная модель в бесконечном цикле попадет в уже посещенное состояние, верификатор остановит цикл.

Требования к системе автоматов формулируются на языке *BIR* в темпоральной логике *LTL*. Для этого в языке *BIR* есть специальное расширение. Например, свойство "автомат `A` никогда не попадет в состояние `Error`" записывается на языке *BIR* следующим образом:

```
LTL.temporalProperty(
  Property.createObservableDictionary(
  Property.createObservableKey(
    "is_Error", AutomataModel.isInState(model, "/A", "Error"))
  ),

  LTL.always(
    LTL.negation(LTL.prop("is_Error"))
  )
);
```

Сначала создается пропозиционная формула "i`s_Error`", которой соответствует вызов требуемого свойства системы автоматов. Затем записывается темпоральная формула в виде вложенных темпоральных операторов. В расширении *LTL* языка *BIR* поддерживаются операторы, указанные в таблице 1 (операторы записаны, как они объявляются на языке *BIR*).

С помощью этих операторов можно задать любую *LTL*-формулу для записи требований к системе автоматов. Такие формулы во время верификации преобразуются в автомат Бюхи, как этого требует алгоритм двойного обхода в глубину [7]. Например, *LTL*-формула `G¬(is_Error)` будет преобразована в автомат следующего вида:

```
function generated$FSA()
{
    loc T0_init:
        when true do
        {
        }
        goto T0_init;
        when AutomataModel.isInState(model, "/A", "Error") do
        {
        }
        goto bad$accept_all;
    loc bad$accept_all:
        when true do
        {
        }
        goto bad$accept_all;
}
```

Состояния сгенерированного автомата, названия которых начинаются с «`bad$`», являются допускающими.

Таблица 1. Операторы языка *BIR*

expdef LTL.Formula *always* (LTL.Formula);	**G** (Globally, всегда)
expdef LTL.Formula *eventually* (LTL.Formula);	**F** (Future, когда-нибудь в будущем)
expdef LTL.Formula *negation* (LTL.Formula);	¬ (отрицание)
expdef LTL.Formula *next* (LTL.Formula);	**X** (neXt, в следующий момент времени)
expdef LTL.Formula *until* (LTL.Formula, LTL.Formula);	**U** (Until, до тех пор, пока)
expdef LTL.Formula *weakUntil* (LTL.Formula, LTL.Formula);	**W** (Weak until). Этот оператор был добавлен в *LTL*-расширение языка *BIR* для удобства. $p\ W\ q = (p\ U\ q)\ \vert\ G\ (p \cap \neg q)$. Это – то же самое, что $p\ U\ q$, однако q не должно когда-либо выполниться.
expdef LTL.Formula *release* (LTL.Formula, LTL.Formula);	**R** (Release, освобождение)
expdef LTL.Formula *equivalence* (LTL.Formula, LTL.Formula);	↔ (Эквивалентно). $p \leftrightarrow q = (p \rightarrow q)\ \&\ (q \rightarrow p)$
expdef LTL.Formula *implication* (LTL.Formula, LTL.Formula);	→ (Следует)
expdef LTL.Formula *conjunction* (LTL.Formula, LTL.Formula);	**&** (И)
expdef LTL.Formula *disjunction* (LTL.Formula, LTL.Formula);	**|** (Или)

2.4.2. Верификация *UniMod*-банкомата

Используем инструментальное средство *UniMod.verifier* для верификации системы управления банкоматом, описанным в разд. 1.2.

2.4.2.1. Банкомат выдает деньги только после авторизации

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

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

здесь `U` – темпоральный оператор `Until` – "пока не". Как показано на рис. 12, в *UniMod*-банкомате выдача денег обеспечивается 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 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 совершенно не обязательно, что пользователь когда-нибудь введет правильный *PIN*-код.

При верификации созданной формулы инструментальным средством *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.

2.4.2.2. Банкомат выдает деньги только при вставленной карте

Проверим, что пользователь банкомата не получит денег до того, как вставит банковскую карту в банкомат. Аналогично предыдущему утверждению, словесная формулировка легко переводится в *LTL*-формулу:

```
[не выдадут деньги] U [вставит карту]
```

На рис. 12 изображено, что вставленная карта характеризуется событием `e6`, а выдача денег происходит с действием `o1.z10`. Тогда *LTL*-формула принимает вид:

```
!o1.z10 U e6
```

Аналогично предыдущей формуле, на языке *BIR* это записывается следующим образом:

```
LTL.temporalProperty (
  Property.createObservableDictionary (
    Property.createObservableKey("card_inserted",
        AutomataModel.wasEvent(model, "e6")),
    Property.createObservableKey("give_money",
        AutomataModel.wasAction(model, "o1.z10"))
  ),

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

Здесь используется описанный ранее предикат «`give_money`», а также объявляется новый – «`card_inserted`», который расшифровывается как `AutomataModel.wasEvent(model, "e6")`. Кроме того, как и в предыдущем примере, вместо темпорального оператора `Until` используется оператор `weakUntil`.

Результат верификации этой формулы инструментальным средством *UniMod.verifier* имеет вид:

```
Transitions: 1, States: 1, Matched States: 0, Max Depth: 1, Errors found: 0, Used Memory: 2MB
Transitions: 5, States: 4, Matched States: 1, Max Depth: 4, Errors found: 0, Used Memory: 1MB
Total memory before search: 759a920 bytes (0,72 Mb)
Total memory after search: 1a193a216 bytes (1,14 Mb)
Total search time: 1594 ms (0:0:1)
States count: 4
Matched states count: 1
Max depth: 4
Done!
Verification successful!
```

Таким образом, требование, чтобы деньги выдавались только при условии, что была вставлена карта, выполняется в *UniMod*-банкомате.

2.4.2.3. Банкомат печатает чек только после авторизации

Проверим, что банкомат не напечатает чек, если не был введен правильный *PIN*-код. Словесная формулировка преобразуется в темпоральную формулу:

```
[не будет напечатан чек] U [введен правильный PIN-код]
```

В *UniMod*-банкомате предусмотрено два случая, когда печатается чек: печать баланса и печать отчета по снятию денег. Учтем оба варианта:

```
[не будет напечатан чек баланса И не будет напечатан чек по снятию денег]
        U [введен правильный PIN-код]
```

Перефразируем формулу, заменив "не … и не …" на "не (… или …)":

```
[не будет напечатан чек (баланса ИЛИ по снятию денег)] U [введен
        правильный PIN-код]
```

Введение правильного PIN-кода в *UniMod*-банкомате описывается событием `e10`. Печать баланса на чеке происходит с действием `o1.z7`, а печать чека по операции выдачи денег происходит с действием `o1.z12`. Тогда формула принимает следующий вид:

```
!(o1.z7 || o1.z12) U e10
```

Запишем ее на языке *BIR*:

```
LTL.temporalProperty (
  Property.createObservableDictionary (
    Property.createObservableKey("check1",
        AutomataModel.wasAction(model, "o1.z7")),
    Property.createObservableKey("check2",
        AutomataModel.wasAction(model, "o1.z12")),
    Property.createObservableKey("correct_pin",
        AutomataModel.wasEvent(model, "e10"))
  ),

  LTL.weakUntil (
    LTL.negation (LTL.disjunction (
      LTL.prop("check1"),
      LTL.prop("check2")
    )),
    LTL.prop("correct_pin")
  )
);
```

Верификация этой формулы дает результат (формат описан в разд. 3.3.6):

```
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: 762a048 bytes (0,73 Mb)
Total memory after search: 1a199a928 bytes (1,14 Mb)
Total search time: 688 ms (0:0:0)
States count: 41
Matched states count: 22
```

```
Max depth: 14
Done!
Verification successful!
```

Таким образом, верифицируемая формула выполняется.

2.4.2.4. Выдаваемая сумма не превышает остаток на счете

Проверим, что пользователь не получит денег, если он запросил больше, чем есть у него на счете. Утверждение можно формализовать следующим образом:

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

где «→» – оператор импликации,.

При запросе суммы денег большей, чем есть на счете, происходит событие `e14`. Деньги выдаются событием `o1.z10`. Можно записать утверждение следующим образом:

```
G (e14 → !o1.z10),
```

где `G` — темпоральный оператор *Globally.* Таким образом, "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, 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, чтобы учитывать такие варианты:

```
G (e14 → G !o1.z10).
```

Таким образом, «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!
```

Свойство не выполняется. При этом выдан контрпример, показывающий историю работы программы, в которой свойство нарушается. В контрпримере жирным выделено, что, действительно, на шаге "7" произошло событие `e14`. После этого на шаге «14» выполнилось действие `o1.z10`, что нарушило верифицируемое условие. На рис. 53 изображено графическое представление контрпримера.

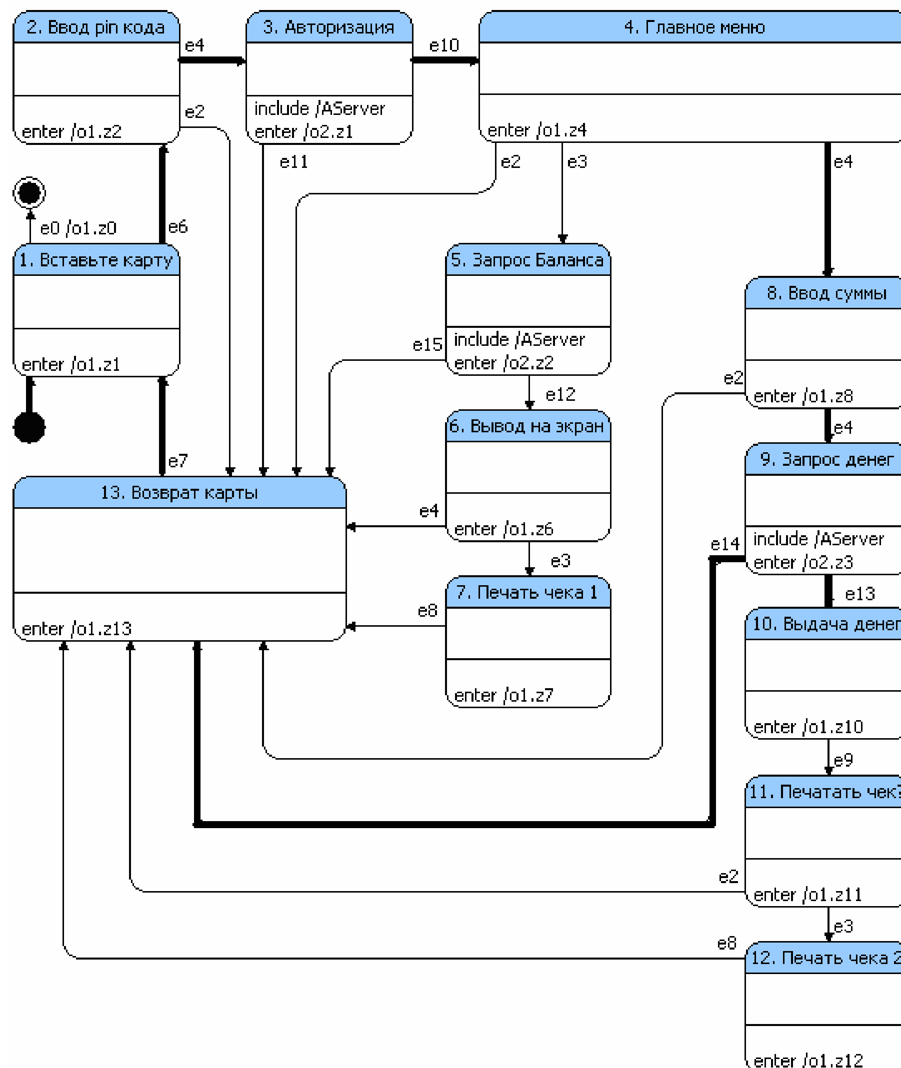


Рис. 53. Контрпример формулы «G (e14 → G !o1.z10)»

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

сформулированное требование. Таким образом, требование было сформулировано некорректно, отчего и возникла ошибка верификации.

Исправим требование, добавив условие, что деньги не выдаются, но до тех пор, пока не будет запрошена сумма денег, возможная к снятию:

```
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 выдает результат:

```
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 требуемым образом уточнена и верифицирована. Банкомат удовлетворяет предъявленному требованию.

2.4.2.5. Деньги не выдаются, пока пользователь не сделает соответствующий запрос

Проверим, что деньги не выдаются, пока пользователь не сделает 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. 14. 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]
actions [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)] ] fsaState [T0_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) - (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)] ] fsaState [T0_init]
Model [ step [7] 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) - (Авторизация); (/AClient:3. Авторизация/AServer) - (Чтение запроса); (/AClient) - (10.
Выдача денег)] ] fsaState [bad$accept_all]
```

Из протокола видно, что 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. 12, означает "Запрос снятия денег". Далее произошло событие `e13`, которое, как изображено на рис. 12, 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").

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

```
G (o2.z3 → X e23)
```

где `X` – *LTL*-оператор *Next*. Добавим теперь полученное условие в верифицируемую формулу через 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 инструментальным средством *UniMod.verifier* оказывается удачной. Это означает, что управляющая система банкомата действительно выполняет выдачу денег лишь после того, как запросила наличие денег с сервера.

2.4.2.6. 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 `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 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.disjunction(
          LTL.prop ("card_returned"),
          LTL.eventually (LTL.prop ("card_returned"))
        )
      ))
    );
```

Верификация данной формулы успешна.

2.4.2.7. Безусловная выдача денег

Верифицируем заведомо ложное свойство банкомата, чтобы проверить возможности верификатора: "Пользователь всегда получает деньги" (ложное свойство 3). В темпоральной логике оно запишется следующим образом:

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

Деньги выдаются действием `o1.z10`. При этом формула принимает вид:

```
G F o1.z10
```

На языке *BIR* она записывается

```
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", а затем переход по нажатию кнопки "Выключить" (событие `e0`) в конечное состояние главного автомата *AClient*. Как и предполагалось, в этом случае, выдачи денег не происходит.

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`.

Разобранные примеры показали, что инструментальное средство *UniMod.verifier* адекватно верифицирует предложенные свойства банкомата. Поскольку модель банкомата достаточно проста, из протокола 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 рассмотренный верификатор 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 при необходимости этот недостаток достаточно просто исправить. Это было сделано в примере «Деньги не выдаются, пока пользователь не сделает соответствующий запрос», где небольшое добавление в верифицируемое свойство позволило решить проблему наведенной ошибки.

Все это позволяет говорить о том, что верификатор *UniMod.verifier* – надежная, корректная и легкая в использовании программа.

2.5. Выводы

Инструментальные средства *FSM Verifier*, *CTLVerifier*, *Converter* и *UniMod.verifier*, разработанные на основе методов верификации автоматных программ, предложенных на втором этапе работ, показали свою эффективность при верификации тестовых примеров. При этом были подтверждены ожидаемые свойства банкоматов и найдены ожидаемые ошибки.

Из изложенного следует, что методы, разработанные в рамках работы позволяют эффективно верифицировать автоматные программы.

3. ПРОГРАММНАЯ ДОКУМЕНТАЦИЯ

3.1. FSM VERIFIER

3.1.1. Общие сведения

Инструментальное средство *FSM Verifier* предназначено для верификации автоматных программ. Для описании модели в нем используется язык *SMV*. Для проверки модели подойдет любое средство, использующее язык *SMV*. Верификация банкомата проводилась с использованием верификатора *NuSMV* [18] (http://nusmv.irst.itc.it/).

Программное средство разработано на языке *Java*, и требует для своего запуска *JRE 5* компании «*Sun Microsystems*», которую можно скачать с сайта компании: (http://java.sun.com/javase/downloads/index.jsp)

3.1.2. Функциональное назначение

Инструментальное средство позволяет проверять систему автоматов Мили. Автоматы могут взаимодействовать между собой следующим образом:

- они могут узнавать о номерах состояний других автоматов;
- один автомат может передавать управление другому автомату, предавая ему событие.

Для задания спецификации системы используется формулы логики *CTL*. При верификации можно задавать следующие свойства состояний системы:

- в каком состоянии будет находиться автомат или несколько автоматов;
- на какой дуге будет находиться вызываемый автомат;
- какими будут значения входных переменных при выполнении перехода.

Программное средство позволяет восстанавливать контрпример по выводу верификатора, заменяя последовательность значений переменных описаниями в терминах автоматной модели

3.1.3. Описание логической структуры

Инструментальное средство *FSM Verifier* реализует алгоритм, описанный в разд. 1.1 отчета за второй этап.

Основные модули программы изображены на рис. 54.

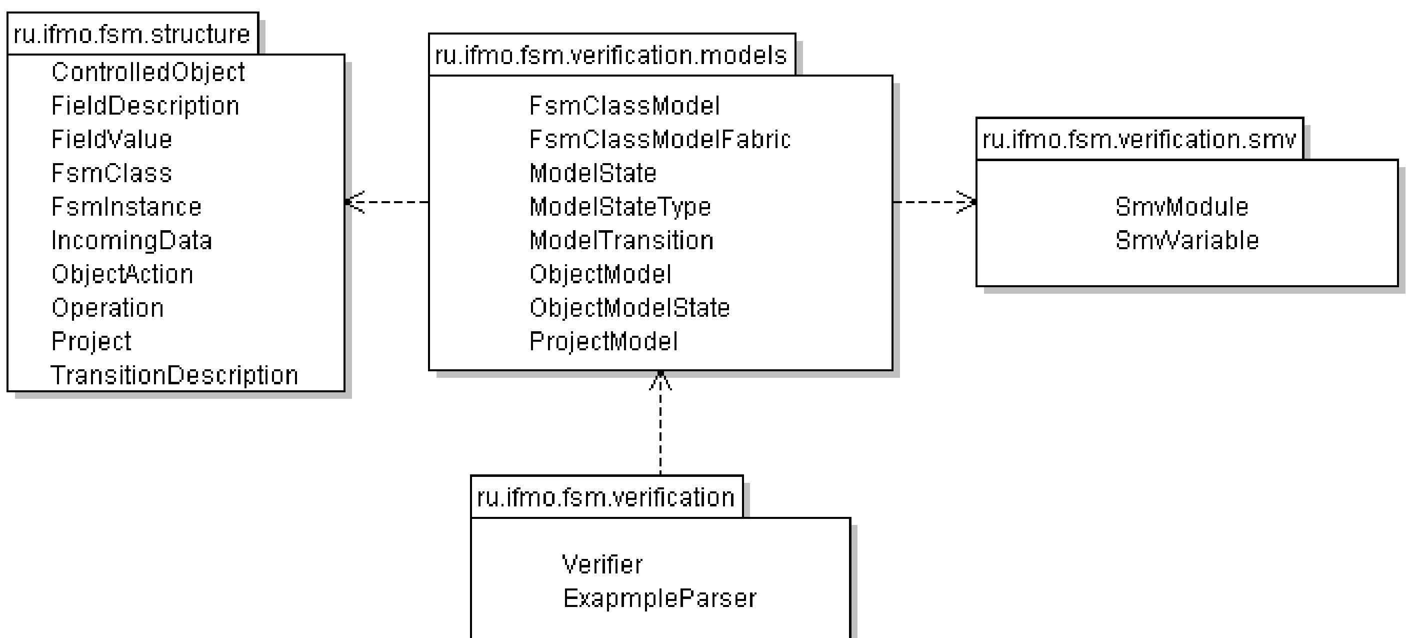


Рис. 54. Пакеты программы *FSM Verifier*.

Опишем назначение модулей:

- `ru.ifmo.fsm.verification` – содержит классы с точками входа;
- `ru.ifmo.fsm.structure` – содержит набор классов, описывающих структуру автоматной программы;
- `ru.ifmo.fsm.verification.models` – содержит классы, описывающие модель автомата.
- `ru.ifmo.fsm.verification.smv` – содержит классы, описывающие модель на языке *SMV*.

3.1.4. Используемые технические средства

Так как программа *FSM Verifier* использует для своей работы виртуальную машину *Java 5,*. то для нормального функционирования этой программы необходимо, чтобы аппаратное обеспечение персональной ЭВМ удовлетворяло следующим требованиям:

- процессор *Intel Pentium VI* или совместимый;
- тактовая частота процессора 2000 МГц, не менее;
- оперативная память 128 MB, не менее;
- дисковый накопитель объемом 1 GB, не менее.

3.1.5. Вызов и загрузка

Вызов программы осуществляется следующим образом:

```
java -jar fsmverifier.jar inputfile.fsm STACK_SIZE >output.smv
```

Здесь `inputfile.fsm` – файл с описанием автоматной модели, `STACK_SIZE` – размер стека для автоматной модели.

Далее вызывается программа верификатор:

Для *NuSMV*:

```
NuSMV output.smv
```

Для *SMV*:

```
smv output.smv
```

Далее, для получения контрпримера запускается преобразование контрпримера:

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

Здесь `output.out` – файл с выводом верификатора.

3.1.6. Входные данные

Входные данные задаются в файле, указанном в соответствующем параметре. В формате *XML*.

Корневой элемент *XML*-файла называется `project` и имеет единственный атрибут `name`, в котором содержится имя проекта. Корневой элемент содержит произвольное число элементов `<class>` и `<instance>`, а также один элемент `<controlledObject>`. Ниже подробно рассматривается каждый из этих элементов.

- Элемент `<class>` описывает класс автоматов с одинаковым набором состояний и переходов. Элемент `<class>` может, содержать в себе элементы: `<state>`, `<field>`. Этот элемент имеет следующие атрибуты:
  1. `name` – имя классов автоматов;
  2. `description` – описание классов автоматов.
- Элемент `<state>` описывает состояние автомата. Этот элемент может содержать в себе элементы `<transition>`, может содержать следующие атрибуты:
  3. `name` – имя состояния;
  4. `description` – описание состояния.

- Элемент `<transition>` содержит описание одного перехода автомата. Он имеет следующие атрибуты:
    5. `toState` – имя состояния, в которое переходит автомат по этому переходу;
    6. `event` – событие, при котором срабатывает переход;
    7. `condition` – условие, при котором срабатывает переход;
    8. `action` – действия, которые выполняются при переходе.
- Элемент `<field>` содержит описание ссылки на автомат, с которым взаимодействует (вызывает или проверяет состояния) текущий автомат. Данный элемент имеет следующие обязательные атрибуты:
    9. `name` – имя ссылки;
    10. `type` – тип ссылки.
- Элемент `<controlledObject>` содержит описание объекта управления. Может содержать элементы `<x>` и `<z>`, которые описывают входные и выходные воздействия соответственно. Они содержат атрибуты:
    11. `name` – имя входного или выходного воздействия
    12. `description` – описание входного или выходного воздействия.
- Элемент `<instance>` содержит описание конкретного экземпляра автомата.
    13. `type` – имя класса, к которому относится данный экземпляр;
    14. `name` – имя автомата.
- Элемент `<fieldValue>` содержит описание конкретного экземпляра автомата:
    15. `fieldname` – имя поля;
    16. `autoNamename` – имя автомата.
- Элемент `<specification>` содержит формулу для проверки. Содержит атрибуты:
    17. `name` – название формулы
    18. `data` – текст *CTL* формулы.

3.1.7. Выходные данные

При построении модели программа в стандартный вывод выдает модель на языке *SMV*. При преобразовании контрпримера программа в стандартный вывод пишет контрпример в виде:

```
Шаг: 1
  Автомат A1 в состоянии s1
  Автомат A2 в состоянии s1
Шаг: 2
  Получено событие: e1
  Активный автомат: A1
  Находится в состоянии s1
…
Шаг: N
  Автомат A1 в состоянии s2
  Автомат A2 в состоянии s1
```

3.2. *CTL*VERIFIER

Инструментальное *CTLVerifier* средство выполняет следующие функции. Оно получает на вход смешанные автоматы, взаимодействующие по вложенности, и строит для них модель Крипке. После этого средство на этой модели выполняет верификацию формул, поданных на вход. Поскольку верификация выполняется индуктивно по подформулам и верификатор в процессе работы проверяет, кроме основной формулы, все её подформулы, каждая из этих подформул имеет свой идентификатор, и для каждой из них генерируется свой результат. Верификатор возвращает на выход модель Крипке. Кроме того, для каждой подформулы верификатор возвращает множество позиций модели, в которых эта подформула выполняется. Для тех позиций, в которых соответствующий результат может быть подтверждён некоторой трассой, выводится произвольная подтверждающая трасса.

Инструментальное средство имеет следующий формат входных данных (далее будет приведён пример для модели банкомата): данные содержатся в одном файле, который разделён на секции. Регистр символов не имеет значения. Строки, начинающиеся с символа ‘;’ (точка с запятой), считаются комментарием и игнорируются. Секции идут в строго определённом порядке. Каждая секция начинается с заголовка, который представляет собой имя этой секции, помещённое в квадратные скобки: `[NameOfSection]`.

Все используемые во входном файле имена могут содержать любые символы, кроме управляющих, таких как: ‘$’, ‘%’, ‘<’, ‘-’, ‘>’, ‘:’, ‘;’, ‘/’, ‘&’, ‘|’, ‘^’, ‘!’, ‘=’, разделители.

Первая секция называется `Automata`. В ней перечислены названия всех автоматов, содержащихся в модели, по одному названию в строке. Перед именем основного автомата стоит символ ‘$’. Пример:

```
[Automata]
$AClient
AServer
```

Вторая секция называется `Includes` и содержит информацию о вложенности одних автоматов в другие. Если автомат `B` вложен в некоторое состояние автомата `A`, то эта информация на отдельной строке обозначается следующим образом: `A<-B`. Отношение вложенности не должно содержать циклов. Пример:

```
[Includes]
AClient<-AServer
```

Третья секция называется `Events` и содержит список всех событий, на которые может реагировать проверяемая автоматная система. Название каждого такого события записывается на отдельной строке. Пример:

```
[Events]
e0
e1
e2
```

Четвёртая секция называется `Actions` и содержит список всех действий (выходных переменных), которые могут генерироваться автоматами. Название каждого такого действия записывается на отдельной строке. Пример:

```
[Actions]
o1.z0
o1.z1
o2.z3
o2.z5
```

Пятая секция называется `Inputs` и содержит список входных переменных, которые в качестве условий могут присутствовать на переходах автоматов. Название каждой такой переменной, как и раньше, записывается на отдельной строке. Примеры верификации *CTL*-формул рассматривались для *UniMod*-модели банкомата, которая не содержит входных переменных. Поэтому для неё эта секция будет пустой:

```
[Inputs]
```

Далее идут секции, описывающие автоматные модели. Каждая из секций соответствует одному автомату. Имя секции совпадает с именем автомата. Секция состоит из двух частей, которые разделяются строкой, начинающейся со знака ‘-’ (минус). Первая часть содержит информацию о состояниях, а вторая – о переходах автомата.

В первой части информация о каждом состоянии содержится на отдельной строке. Строка начинается с имени состояния. Имена всех состояний во всех автоматах должны быть различны. Перед именем стартового состояния ставится символ ‘$’, перед именем терминального состояния ставится символ ‘%’ (знак процента). После этого идёт двоеточие, и далее через запятую по порядку перечисляются все выходные воздействия, которые должны быть выполнены автоматом

при входе в это состояние. Потом ставится символ ‘;’ и через запятую по порядку перечисляются автоматы, вызываемые при входе в это состояние.

Пример информации о состояниях для автомата AClient (терминальное состояние обозначено через Y0):

```
$Y1 : o1.z1;
%Y0 :;
Y2  : o1.z2;
Y3  : o2.z3;  AServer
Y4  : o1.z4;
Y5  : o2.z5;  AServer
Y6  : o1.z6;
```

Во второй части секции для соответствующего автомата информация о каждом переходе содержится на отдельной строке. Строка начинается с описания самого перехода: указывается состояние, из которого он исходит, и состояние, в которое он входит. Эти состояния разделены конструкцией “->” (например, “Y1->Y2”). Далее идёт двоеточие, после этого указывается событие, по которому произошёл переход и условие перехода (если оно есть). Событие и условие разделены символом ‘&’ (амперсанд). Условие представляет собой конъюнкцию литералов, составленных из входных переменных — список литералов, разделённых символом ‘&’. Каждый литерал представляет собой имя переменной и знак отрицания ‘!’ при необходимости. Подразумевается, что если знак отрицания отсутствует, то переход может быть выполнен только при истинном значении соответствующей переменной. Если же он присутствует, то – только при ложном значении. После условия на переходе ставится символ ‘/’, а за ним через запятую по порядку перечисляются все выходные воздействия, которые должны быть выполнены автоматом на этом переходе. Пример записи информации об одном переходе:

‘Y5->Y8:e7&x1&!x2/z3,z4’.

Пример информации о переходах для автомата AClient:

```
Y1  -> Y0  : e0  / o1.z0
Y1  -> Y2  : e6  /
Y2  -> Y3  : e4  /
Y2  -> Y13 : e2  /
Y3  -> Y4  : e10 /
Y3  -> Y13 : e11 /
Y4  -> Y13 : e2  /
Y4  -> Y5  : e3  /
Y4  -> Y8  : e4  /
Y5  -> Y6  : e12 /
Y5  -> Y13 : e15 /
Y6  -> Y7  : e3  /
Y6  -> Y13 : e4  /
Y7  -> Y13 : e8  /
Y8  -> Y13 : e2  /
Y8  -> Y9  : e4  /
Y9  -> Y13 : e14 /
Y9  -> Y10 : e13 /
Y10 -> Y11 : e19 /
Y11 -> Y13 : e2  /
Y11 -> Y12 : e3  /
Y12 -> Y13 : e8  /
```

Таким образом, секция автомата AClient имеет следующий вид:

```
[AClient]
$Y1 : o1.z1;
%Y0 :;
Y2  : o1.z2;
Y3  : o2.z3;  AServer
```

```
Y4  : o1.z4;
Y5  : o2.z5;  AServer
Y6  : o1.z6;
Y7  : o1.z7;
Y8  : o1.z8;
Y9  : o2.z9;  AServer
Y10 : o1.z10;
Y11 : o1.z11;
Y12 : o1.z12;
Y13 : o1.z13;
------------------
Y1  -> Y0  : e0  / o1.z0
Y1  -> Y2  : e6  /
Y2  -> Y3  : e4  /
Y2  -> Y13 : e2  /
Y3  -> Y4  : e10 /
Y3  -> Y13 : e11 /
Y4  -> Y13 : e2  /
Y4  -> Y5  : e3  /
Y4  -> Y8  : e4  /
Y5  -> Y6  : e12 /
Y5  -> Y13 : e15 /
Y6  -> Y7  : e3  /
Y6  -> Y13 : e4  /
Y7  -> Y13 : e8  /
Y8  -> Y13 : e2  /
Y8  -> Y9  : e4  /
Y9  -> Y13 : e14 /
Y9  -> Y10 : e13 /
Y10 -> Y11 : e19 /
Y11 -> Y13 : e2  /
Y11 -> Y12 : e3  /
Y12 -> Y13 : e8  /
```

Соответствующая секция для автомата `AServer`:

```
[AServer]
$Read     : o3.z0;
Authority : o3.z1;
Balance   : o3.z2;
Withdraw  : o3.z3;
%Answer   : o3.z4;
------------------
Read->Authority   : e21 /
Read->Balance     : e22 /
Read->Withdraw    : e23 /
Authority->Answer : e24 /
Balance  ->Answer : e24 /
Withdraw ->Answer : e24 /
```

После всех секций, описывающих автоматы, идёт заключительная секция, которая содержит проверяемые формулы. Она называется `Properties`. Каждая строка содержит одно свойство. Свойства формируются индуктивно по построению. Строка начинается с имени свойства, далее идёт знак ‘=’, а за ним – либо элементарное предложение, либо элементарная операция над определёнными ранее свойствами. Более точно, опишем синтаксис свойства в нотации Бэкуса-Наура:

```
<свойство> ::= <Имя> '='
          (  '0' | '1' |
           | <атомарное предложение> |
           | '!' <имя> |
           | <имя> '&' <имя> | <имя> '|' <имя> | <имя> '^' <имя> |
```

```
          | '$EX' <имя> | '$EG' <имя> | '$EU[' <имя> ',' <имя> ']'
         )

<атомарное предложение> ::= <имя автомата>  |
                          | <имя состояния> |
                          | <имя события>   |
                          | ['!'] <имя входной переменной> |
                          | <имя выходной переменной>      |
                          | 'InState' | 'InEvent' | 'InAction'
```

`<Имя>` (с заглавной буквы) означает имя текущего свойства.

`<имя>` (с маленькой буквы) означает имя некоторого свойства, определённого ранее (на одной из предыдущих строк).

'`&`' означает конъюнкцию, '`|`' – дизъюнкцию, '`^`' – сложение по модулю 2, '`!`' – отрицание.

Формулы языка *CTL* выражаются через операции **EX**, **EG** и **EU** следующим образом:

$$\mathbf{AX}\, g = \neg \mathbf{EX} \neg g$$
$$\mathbf{EF}\, g = \mathbf{EU}[1, g]$$
$$\mathbf{AF}\, g = \neg \mathbf{EG} \neg g$$
$$\mathbf{AG}\, g = \neg \mathbf{EF} \neg g = \neg \mathbf{EU}[1, \neg g]$$
$$\mathbf{AU}[f, g] = \neg(\mathbf{EU}[\neg g, \neg(f \vee g)] \vee \mathbf{EG} \neg g)$$

Пусть требуется проверить формулу $f = \mathbf{EX}\, \mathbf{E}[\neg(AClient \wedge InState)\, \mathbf{U}\, y13]$, описанную в разд. 2.2.2.5 (и все её подформулы). Тогда секция `Properties` во входном файле будет выглядеть следующим образом:

```
[Properties]
f1 = AClient
f2 = InState
f3 = f1 | f2
f4 = !f3
f5 = Y13
f6 = $EU [f4, f5]
f7 = $EX f6
```

В этом случае формулам `f1`, …, `f7` соответствуют следующие подформулы:

$f_1 = AClient$

$f_2 = InState$

$f_3 = AClient \wedge InState$

$f_4 = \neg(AClient \wedge InState)$

$f_5 = y13$

$f_6 = \mathbf{E}[\neg(AClient \wedge InState)\, \mathbf{U}\, y13]$

$f_7 = f = \mathbf{EX}\, \mathbf{E}[\neg(AClient \wedge InState)\, \mathbf{U}\, y13]$

Пусть требуется проверить формулу $\neg \mathbf{E}[\neg e10\, \mathbf{U}\, o1.z7]$. При этом входной файл имеет вид:

```
[Automata]
$AClient
AServer

[Includes]
AClient<-AServer
```

```
[Events]
e0
e1
e2
e3
e4
e6
e7
e8
e9
e10
e11
e12
e13
e14
e15
e21
e22
e23
e24

[Actions]
o1.z0
o1.z1
o1.z2
o1.z4
o1.z6
o1.z7
o1.z8
o1.z10
o1.z11
o1.z12
o1.z13
o2.z3
o2.z5
o2.z9
o3.z0
o3.z1
o3.z2
o3.z3
o3.z4

[Inputs]


[AClient]
$Y1 : o1.z1;
%Y0 :;
Y2  : o1.z2;
Y3  : o2.z3;  AServer
Y4  : o1.z4;
Y5  : o2.z5;  AServer
Y6  : o1.z6;
Y7  : o1.z7;
Y8  : o1.z8;
Y9  : o2.z9;  AServer
Y10 : o1.z10;
Y11 : o1.z11;
Y12 : o1.z12;
Y13 : o1.z13;

------------------
```

```
Y1  -> Y0  : e0  / o1.z0
Y1  -> Y2  : e6  /
Y2  -> Y3  : e4  /
Y2  -> Y13 : e2  /
Y3  -> Y4  : e10 /
Y3  -> Y13 : e11 /
Y4  -> Y13 : e2  /
Y4  -> Y5  : e3  /
Y4  -> Y8  : e4  /
Y5  -> Y6  : e12 /
Y5  -> Y13 : e15 /
Y6  -> Y7  : e3  /
Y6  -> Y13 : e4  /
Y7  -> Y13 : e8  /
Y8  -> Y13 : e2  /
Y8  -> Y9  : e4  /
Y9  -> Y13 : e14 /
Y9  -> Y10 : e13 /
Y10 -> Y11 : e19 /
Y11 -> Y13 : e2  /
Y11 -> Y12 : e3  /
Y12 -> Y13 : e8  /

[AServer]
$Read     : o3.z0;
Authority : o3.z1;
Balance   : o3.z2;
Withdraw  : o3.z3;
%Answer   : o3.z4;
------------------
Read->Authority   : e21 /
Read->Balance     : e22 /
Read->Withdraw    : e23 /
Authority->Answer : e24 /
Balance  ->Answer : e24 /
Withdraw ->Answer : e24 /

[Properties]
f1  = e10
f2  = !f1
f3  = o1.z7
f4  = $EU [f2, f3]
f5  = !f4
```

Синтаксис запуска инструментального средства:

```
build.exe <имя входного файла> [<имя выходного файла>]
```

Если имя выходного файла не указано, результат выводится на консоль.

3.3. *UniMod*.verifier

3.3.1. Общие сведения

Инструментальное средство *UniMod.verifier* – это верификатор, предназначенный для верификации автоматных программ, созданных с помощью инструментального средства *UniMod*.

Для функционирования верификатора необходимы следующие библиотеки.

1. Библиотеки инструментального средства *UniMod*, версия 2. Данная версия на момент написания отчета еще не была официально опубликована, однако, исходные коды программы доступны через *SVN* по адресу https://*UniMod*.svn.sourceforge.net/svnroot/*UniMod*/*UniMod*-2. В верификаторе

UniMod.verifier необходимые классы были скомпилированы и собраны в две библиотеки:

- core.jar – библиотека, содержащая классы, описывающие автоматную модель,
- runtime.jar – библиотека, содержащая классы, необходимые для интерпретации и запуска автоматной модели.

Кроме этого, необходимо еще несколько стандартных библиотек для работы *UniMod* 2.

2. Библиотеки верификатора *Bogor*, доступны по адресу http://projects.cis.ksu.edu/plugin/ksuafrs/showfiles.php?group_id=8. Основная библиотека называется *Bogor.jar*. Кроме того, необходим набор стандартных библиотек.
3. Библиотека *UniMod.verifier*. Она называется *verifier.jar* и содержит классы, реализующие работу верификатора.

Инструментальное средство *UniMod* 2 и верификатор *Bogor* написаны на языке *Java*, что дало возможность их интеграции в инструментальное средство *UniMod.verifier*, которое также написано на языке *Java*.

3.3.2. Функциональное назначение

Инструментальное средство *UniMod.verifier* характеризуется следующими особенностями:

- оно позволяет верифицировать любые автоматные модели, реализованные при помощи инструментального средства *UniMod;*
- инструментальное средство *UniMod*, в свою очередь, позволяет создавать автоматные программы, представляющие собой реактивную систему, с управляющей компонентой в виде иерархически связанной системы автоматов;
- ограничений на число автоматов или степень вложенности нет;
- каждый автомат может быть как автоматом Мура или Мили, так и смешанным автоматом;
- ограничений на число состояний автоматов нет;
- *UniMod.verifier* позволяет задавать верифицируемые свойства автоматной модели, сформулированные в виде формул темпоральной логики *LTL;*
- в верифицируемой формуле может использоваться заданное множество предикатов, описывающих аспекты работы автоматной программы. Список этих предикатов перечислен в разд. 2.4.1.1;
- каждый предикат вычисляется один раз при обработке каждого события. Формулировать более мелкие предикаты невозможно;
- верифицируется только управляющая система автоматов. Источники событий и объекты управления не участвуют в верификации. Таким образом, возможно возникновение наведенных ошибок в случае, если логика выполнения программы хранится в объектах управления. Для решения проблемы можно внести такую логику в верифицируемую формулу, как показано в примере разд. 2.4.2.5.

3.3.3. Описание логической структуры

Инструментальное средство *UniMod.verifier* использует верификатор *Bogor* для верификации, и инструментальное средство *UniMod*, как интерпретатор автоматной программы. Схема компонентов системы изображена на рис. 55.

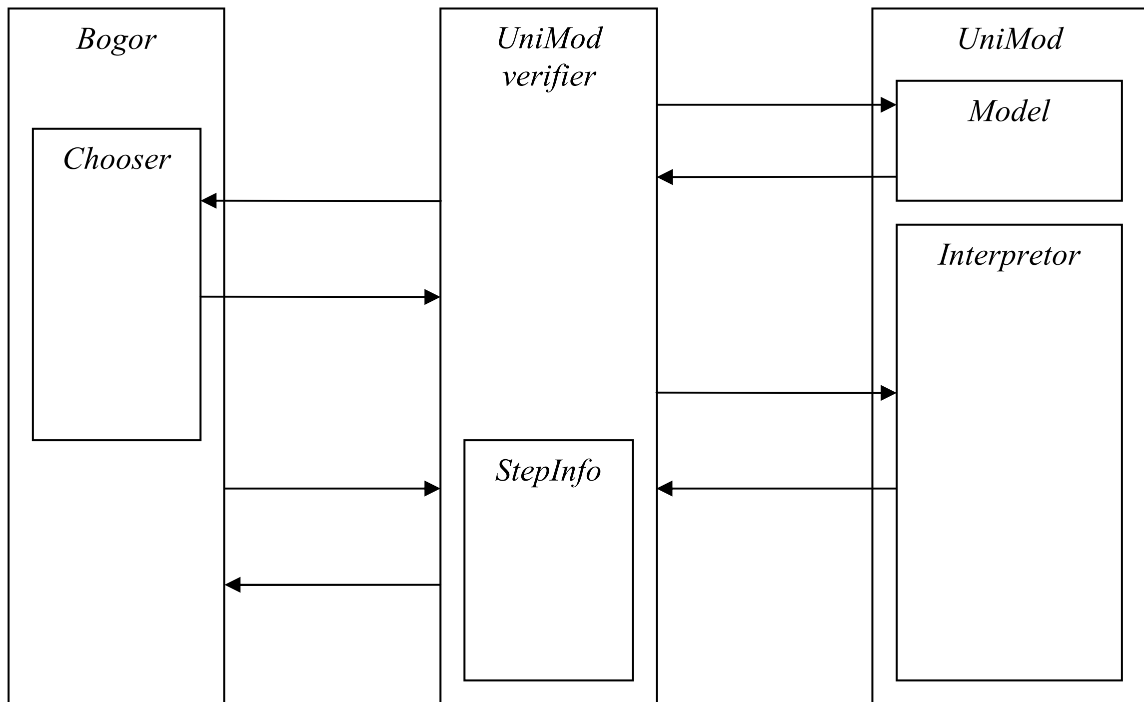


Рис. 55. Схема взаимодействия *Bogor*, *UniMod* и *UniMod.verifier*

Опишем подробнее компоненты системы.

- *Bogor* – это верификатор, предоставляющий возможность расширения его входного языка, что дает возможность ему работать со сложными объектами, в данном случае – с автоматными программами.
- *Chooser* – это компонента верификатора *Bogor*, которая осуществляет выбор при возникновении недетерминированности действия верифицируемого объекта. В данном случае этот компонент выбирает, какое произойдет событие в автоматной программе, а также какое значение примет условие на переходе.
- *UniMod.verifier* – рассматриваемое инструментальное средство.
- *StepInfo* – компонента, хранящая информацию о совершенном шаге обработки события. Такая компонента создается для каждого шага, и выполнимость предикатов вычисляется с использованием данных, хранящихся в ней. Например, там хранится имя обработанного события, вызванные действия объектов управления, состояния автоматов и т.д.
- *UniMod* – инструментальное средство *UniMod*.
- *Model* – модель автоматной программы, хранящаяся в инструментальном средстве *UniMod*, которая сохраняется в виде *XML*-файла.
- *Interpretator* – компонента инструментального средства *UniMod*, выполняющая автоматную программу по ее модели.

Верификатор *Bogor* осуществляет верификацию по алгоритму двойного поиска в глубину [7]. Инструментальное средство *UniMod.verifier* применяет *Bogor* для верификации автоматных программ по методу, описанному в разд. 2.4.1.1 и подробно в разд. 1.3 отчета за второй этап.

3.3.4. Используемые технические средства

Для нормального функционирования инструментального средства *UniMod.verifier* на персональной ЭВМ необходимо, чтобы аппаратное обеспечение персональной ЭВМ удовлетворяло следующим требованиям:

- процессор *Intel Pentium IV* или совместимый;
- тактовая частота процессора не менее 2000 МГц;
- оперативная память не менее 128 Мб;
- дисковый накопитель не менее 3 Гб;
- отображающее устройство (монитор) любого разрешения;
- устройство ввода (клавиатура).

Кроме того, необходимо, чтобы на ЭВМ было установлено следующее программное обеспечение:

- операционная система *Windows XP*;
- *Java Runtime Environment*, версия 5.

3.3.5. Вызов и входные данные

Для верификации автоматной модели с помощью инструментального средства *UniMod.verifier*, необходимо сделать следующее:

- скопировать каталог средства с прилагаемого диска на локальную машину;
- отредактировать файл *verifier.bat* в скопированном каталоге, указав на *.connectivity* файл, содержащий описание автоматной модели в формате инструментального средства *UniMod* 2. Например, если описание хранится в файле *Bank.connectivity* в том же каталоге, что и *verifier.bat*, то последняя строчка файла *verifier.bat* будет выглядеть следующим образом:

```
java com.UniMod.verifier.Main Bank.connectivity UniMod.bir %1
```

- описать свойство для верификации в файле *UniMod.bir*. Свойство объявляется в виде функции с именем. Это имя затем используется при запуске верификатора, для того чтобы указать, какой свойство верифицировать. Например, свойство банкомата "Банкомат выдает деньги только после авторизации" выглядит в файле *UniMod.bir* следующим образом:

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

- запустить верификацию. Для этого необходимо запустить файл *verifier.bat* с параметром – именем свойства для верификации. Например, для верификации приведенного выше свойства, необходимо в командной строке операционной системы *Windows* написать следующее:

```
verifier.bat NoPinNoMoney
```

В результате запуска верификатора в командную строку выведется результат верификации, сгенерированный *UniMod.verifier*. Для того, чтобы вывести то же самое в файл, необходимо в конце команды добавить «`> имя_файла`», например:

```
verifier.bat NoPinNoMoney > result.txt
```

Все исполнимые файлы работают только в операционной системе *Windows*. Инструментальное средство и используемые им библиотеки написаны на платформо-независимом языке *Java*. поэтому верификатор может работать и в других операционных системах, на которых можно установить *Java Runtime Environment* версии 1.5.0. Для запуска верификатора в других операционных системах необходимо переписать исполнимый файл *verifier.bat*. Кроме того, необходимо заменить программу *ltl2ba.exe* на аналогичную программу, работающую в соответствующей операционной системе. По адресу https://robby.user.cis.ksu.edu/Bogor/property-extensions/ltl/ltl2ba/ можно найти версии для операционных систем *Linux*, *Mac OS X*, *Windows 2000*, *Windows 2003*, *Windows XP*. Данная программа используется верификатором *Bogor* и *SPIN* для конвертации темпоральной формулы *LTL* в автомат Бюхи.

3.3.6. Выходные данные

В процессе работы верификатора генерируется отчет о его работе. Если был найден контрпример, то в стандартный поток вывода записывается контрпример. В отчете содержится различная служебная информация, как, например, число пройденных состояний, максимальная длина истории и т.д. Например, в результате приведенного выше примера запуска верификатора, выводится следующая информация:

```
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: 758a032 bytes (0,72 Mb)
Total memory after search: 1a195a872 bytes (1,14 Mb)
Total search time: 688 ms (0:0:0)
States count: 41
Matched states count: 22
Max depth: 14
Done!
Verification successful!
```

Если были найдены контрпримеры, то выводится один из них. Контрпример состоит из последовательного набора строк, каждая из которых описывает шаг работы автоматной модели. Этот набор шагов приводит к нарушению верифицируемого свойства. Например, строка описания шага может выглядеть следующим образом:

```
Model [ step [34] 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) - (Авторизация); (/AClient:3. Авторизация/AServer) - (Чтение запроса); (/AClient) - (10.
Выдача денег)] ] fsaState [bad$accept_all]
```

В этой строке указывается следующая информация:

- `step` – номер шага в контрпримере;
- `event` – обработанное событие;
- `guards` – вычисленные в ходе шага условия на переходах с их значениями;
- `transitions` – совершенные переходы;
- `actions` – вызванные действия объектов управления;
- `states` – состояния автоматов в результате шага;
- `fsaState` – состояние автомата Бюхи, сгенерированного по *LTL*-формуле, в результате перехода.

ЗАКЛЮЧЕНИЕ

В результате исследований, выполненных на третьем этапе работ по контракту, была проведена апробация и экспериментальное исследование методов верификации управляющих программ со сложным поведением, разработанных на втором этапе. Была доказана применимость этих методов на практике и их эффективность с точки зрения обнаружения дефектов программного обеспечения.

В первой главе были разработаны тестовые модели управляющей программы — модели банкоматов. Так же был разработан набор свойств для верификации тестовых моделей. При этом, рассматривались не только выполняющиеся свойства, но и заведомо ложны свойства, что позволило обнаруживать ошибки, как первого, так и второго рода.

Во второй главе приведены результаты экспериментальных исследований верификаторов автоматных программ, созданных на основе методов, разработанных на втором этапе работы. Исследования показали высокое качество, как самих верификаторов, так и реализованных методов.

В третьей главе приведена программная документация на разработанные верификаторы.

Таким образом, были решены все задачи, поставленные в техническом задании на проведение третьего этапа работы.

Результаты выполненных работ, а также патентных исследований, позволяют утверждать, что научно-технический уровень исследований соответствует уровню исследований в рассматриваемой области, проводимых в лучших исследовательских центрах мира.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *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.
2. *Васильева К. А., Кузьмин Е. В., Соколов В. А.* Верификация автоматных программ с использованием логики LTL. http://is.ifmo.ru/verification/_ltl_aut_ver_1.pdf
3. *Вельдер С. Э., Шалыто А. А.* О верификации автоматных программ на основе метода Model Checking // Информационно-управляющие системы. 2007. № 3, с. 27–38.
4. *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/
5. *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
6. *Кларк Э., Грамберг О., Пелед Д.* Верификация моделей программ: Model Checking. М.: МЦНМО, 2002.
7. *Хоффман Л.* Разговоры о Model Checking. http://is.ifmo.ru/verification/_model_checking.pdf
8. *Классификация абстрактных автоматов.* http://ru.wikipedia.org/wiki/Классификация_абстрактных_автоматов/
9. *Козлов В. А., Комалёва О. А.* Моделирование работы банкомата // http://is.ifmo.ru/*UniMod*-projects/bankomat/
10. *Лифшиц Ю.* Верификация программ и темпоральные логики. Лекция № 3 курса «Современные задачи теоретической информатики». http://download.yandex.ru/class/lifshits/lecture-note03.pdf/
11. *Хоффман Л.* В поисках надежного кода. http://is.ifmo.ru/verification/_v_poiskax_nadejnogo_koda.pdf
12. *Лифшиц Ю.* Верификация программ и темпоральные логики. Лекция № 3 курса «Современные задачи теоретической информатики». СПбГУ ИТМО, 2005. http://logic.pdmi.ras.ru/~yura/modern/03modernnote.pdf
13. *Лифшиц Ю.* Символьная верификация программ. Лекция № 4 курса «Современные задачи теоретической информатики». СПбГУ ИТМО, 2005. http://logic.pdmi.ras.ru/~yura/modern/04modernnote.pdf
14. *Новиков Ф. А.* Визуальное конструирование программ. // Информационно-управляющие системы. 2005. № 6, с. 9–22. http://is.ifmo.ru/works/visualcons/
15. *Описание Промелы и некоторых примеров программ.* http://dcn.infos.ru/~karpov/Course%20work/Spin%20%20Manual.pdf
16. *Первушин Е. В., Шалыто А. А.* Моделирование банкомата // http://is.ifmo.ru/projects/bankomat/
17. *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/
18. *Шалыто А. А., Туккель Н. И.* Объектно-ориентированное программирование с явным выделением состояний / Материалы Международной научно-технической конференции «Искусственный интеллект». Т.1. Таганрогский радиотехнический университет, Донецкий гос. институт искусственного интеллекта. 2002.
19. *Шалыто А. А., Туккель Н. И.* Программирование с явным выделением состояний // Мир ПК. 2001. № 8, 9. http://is.ifmo.ru/works/mirk/
20. *Яковлев А. В., Лукин М. А., Шалыто А. А.* Реализация классической игры "Ним" на основе автоматного подхода. СПбГУ ИТМО, 2005. http://is.ifmo.ru/UniMod-projects/knim/
21. *Büchi automaton.* http://en.wikipedia.org/wiki/Büchi_automaton/

22. *Clarke E. M., Emerson E. A.* Synthesis of synchronisation skeletons for branching time logic /Logic of Programs, LNCS 131, pp. 52–71, 1981.
23. *Emerson E. A., Clarke E. M.* Using branching time temporal logic to synthesize synchronisation skeletons // Science of Computer Programming 2: 241–266, 1982.
24. Finite state machine. http://en.wikipedia.org/wiki/Finite_state_machine
25. *Linear temporal logic.* http://en.wikipedia.org/wiki/Linear_temporal_logic/
26. Mealy machine. http://en.wikipedia.org/wiki/Mealy_machine
27. Moore machine. http://en.wikipedia.org/wiki/Moore_machine
28. *New Symbolic Model Verifier.* http://nusmv.irst.itc.it/
29. *Promela reference – ltl.* http://www.spinroot.com/spin/Man/ltl.html/
30. *Robby, Dwyer M., Hatcliff J. Bogor*: A Flexible Framework for Creating Software Model Checkers. TAIC PART 2006, pp 3–22.
31. *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.
32. *Roux C., Encrenaz E. CTL* May Be Ambiguous when Model Checking Moore Machines. UPMC – LIP6 – ASIM, CHARME, 2003. http://sed.free.fr/cr/charme2003-presentation.pdf
33. *SPIN home page.* http://SPINroot.com
34. *UniMod home page.* http://*UniMod*.sourceforge.net/