FИНИСТЕРСТВО 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. № 370094.4

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

ЗАКЛЮЧИТЕЛЬНЫЙ ОТЧЕТ ПО IV ЭТАПУ «ОБОБЩЕНИЕ И ОЦЕНКА РЕЗУЛЬТАТОВ ИССЛЕДОВАНИЙ»

ЛИСТОВ 99

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
Нормоконтролер	Г. Р. Туктарова	Отчет в целом
Заведующий кафедрой, докт. физ.-мат. наук, профессор	В. А. Соколов	Раздел 1.
Ведущий научный сотрудник, докт. техн. наук, профессор	В. В. Антипов	Раздел 2.
Ведущий научный сотрудник, канд. техн. наук	В. В. Киселев	Раздел 1.1.
Ведущий научный сотрудник, канд. техн. наук	Р. Н. Котляр	Раздел 1.2
Ведущий научный сотрудник, канд. техн. наук	Ю. П. Московцев	Раздел 1.3.
Ведущий научный сотрудник, канд. техн. наук, доцент	В. А. Третьяков	Раздел 1.4.
Ведущий научный сотрудник, канд. техн. наук	Г. М. Файкин	Раздел 1.5.
Старший преподаватель, канд. физ.-мат. наук	Д. Ю. Чалый	Раздел 2.1.
Руководитель лаборатории	С. П. Жуков	Раздел 2.2.
Канд. техн. наук	С. П. Новиков	Раздел 2.3.
Ассистент	В. С. Гуров	Раздел 2.4.
Руководитель ВТК, ассистент кафедры КТ	А. П. Мельничук	Раздел 1.2.1.
Член ВТК, профессор кафедры ИС	Е. Ю. Михайлова	Раздел 1.2.2.
Член ВТК, доцент кафедры КТ	В. Д. Наумчик	Раздел 1.3.1.
Член ВТК, доцент кафедры КТ	М. Ю. Осипов	Раздел 1.3.2.
Член ВТК, доцент кафедры КТ	А. Н. Воробьев	Раздел 1.3.3.
Член ВТК, доцент кафедры КТ	Ю. А. Щупак	Раздел 1.4.1.
Член ВТК, доцент кафедры КТ	С. В. Чириков	Раздел 1.4.2.
Член ВТК, доцент кафедры КТ	А. С. Сегаль	Раздел 1.4.3
Член ВТК, доцент кафедры КТ	Д. Г. Шопырин	Раздел 1.5.1.
Член ВТК, доцент кафедры ИС	Д. А. Зубок	Раздел 1.5.2.
Член ВТК, ассистент кафедры ИС	В. В. Повышев	Раздел 1.2.1.
Член ВТК, ассистент кафедры ИС	В. В. Ильин	Раздел 2.1.2.
Член ВТК, ассистент кафедры ИС	М. Г. Холин	Раздел 2.1.2.
Магистрант	Б. Р. Яминов	Раздел 2.3.
Магистрант	С. Э. Вельдер	Раздел 2.2.
Магистрант	М. А. Лукин	Раздел 2.4.
Магистрант	К. А. Васильева	Раздел 2.4.1.
Студент	Е. А. Курбацкий	Раздел. 2.1.
Студент	М. Э. Дворкин	Раздел 2.4.2.

РЕФЕРАТ

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

Цель этапа – разработка предложений и рекомендаций по использованию технологии верификации автоматных программ со сложным поведением, предложенной на предыдущих этапах выполнения контракта.

В результате выполнения четвертого, заключительного этапа работ сформулированы рекомендации по применению методов и технологии верификации автоматных программ, разработанных на предыдущем этапе. Также были разработаны методические рекомендации по применению созданных прототипов инструментальных средств для верификации автоматных программ.

В первой главе отчета описаны предложения и рекомендации по верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода.

Вторая глава содержит методические рекомендации и указания по верификации автоматных моделей систем со сложным поведением на основе разработанных методов и технологии верификации программ.

ОГЛАВЛЕНИЕ

СПИСОК ИСПОЛНИТЕЛЕЙ 2
РЕФЕРАТ 3
ОГЛАВЛЕНИЕ 4
ВВЕДЕНИЕ 6
ОСНОВНАЯ ЧАСТЬ 8
1. ПРЕДЛОЖЕНИЯ И РЕКОМЕНДАЦИИ ПО ВЕРИФИКАЦИИ УПРАВЛЯЮЩИХ ПРОГРАММ СО СЛОЖНЫМ ПОВЕДЕНИЕМ, ПОСТРОЕННЫХ НА ОСНОВЕ АВТОМАТНОГО ПОДХОДА 8
1.1. Методические рекомендации по выбору метода верификации 8
1.2. Метод верификации автоматных программ с использованием верификатора *NuSMV* 10
1.2.1. Описание метода 10
1.2.1.1. Преобразование системы автоматов в модель 11
1.2.1.2. Преобразование системы автоматов в модель на языке *SMV* 13
1.2.1.3. Запись требований 15
1.2.1.4. Преобразование контрпримера 16
1.2.2. Инструментальное средство *FSMVerifier* 16
1.3. Метод верификации визуальных автоматных моделей 16
1.3.1. Описание метода 17
1.3.2. Инструментальное средство *Converter* 17
1.3.3. Методические рекомендации по применению инструментального средства *Converter* 18
1.4. Метод верификации на основе эмуляции 19
1.4.1. Описание метода 19
1.4.2. Инструментальное средство *UniMod.Verifier* 22
1.4.3. Методические рекомендации по применению инструментального средства *UniMod.Verifier* 24
1.5. Метод верификации на основе темпоральной логики *CTL* 26
1.5.1. Описание метода 26
1.5.2. Рекомендации по применению инструментального средства *CTLVerifier* 28
1.6. Выводы 32
2. МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ И УКАЗАНИЯ ПО ВЕРИФИКАЦИИ УПРАВЛЯЮЩИХ ПРОГРАММ СИСТЕМ СО СЛОЖНЫМ ПОВЕДЕНИЕМ, ПОСТРОЕННЫХ НА ОСНОВЕ АВТОМАТНОГО ПОДХОДА 34
2.1. Инструментальное средство *FSM Verifier* 34
2.1.1. Краткое описание 34
2.1.1.1. Вызов и загрузка 34
2.1.1.2. Входные данные 34
2.1.1.3. Выходные данные 35
2.1.1.4. Технические требования 36
2.1.2. Установка и настройка 36
2.1.3. Пример верификации 37
2.1.3.1. Верифицируемая модель банкомата 37
2.1.3.2. Проверка истинного свойства 41
2.1.3.3. Проверка ложного свойства 41

2.1.3.4. Модель банкомата на языке *SMV* 45
2.2. Инструментальное средство *Converter* 49
2.2.1. Установка и настройка 49
2.2.2. Пример применения 51
2.3. Инструментальное средство *UniMod.Verifier* 62
2.3.1. Установка и настройка 62
2.3.2. Пример применения 63
2.3.2.1. Подготовка *UniMod*-модели 63
2.3.2.2. Задание формул для верификации 65
2.3.2.3. Верификация свойств 68
2.4. Инструментальное средство *CTLVerifier* 74
2.4.1. Установка и настройка 74
2.4.1.1. Общие сведения 74
2.4.1.2. Функциональное назначение 75
2.4.1.3. Описание логической структуры 75
2.4.1.4. Используемые технические средства 80
2.4.1.5. Вызов и загрузка 80
2.4.1.6. Входные данные 80
2.4.2. Пример применения 85
2.4.2.1. Примеры входных данных 85
2.4.2.2. Примеры выходных данных: модель Крипке 87
2.4.2.3. Примеры выходных данных: результаты верификации 93
2.5. Выводы 96
ЗАКЛЮЧЕНИЕ 97
СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 98

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 разработка предложений и рекомендаций по использованию технологии верификации автоматных программ со сложным поведением, предложенных на предыдущих этапах выполнения контракта.

Задачами этапа являются:

1. Анализ разработанных методов и прототипов инструментальных средств, созданных на их основе.
2. Разработка предложений и методических рекомендаций по применению предложенных методов и прототипов инструментальных средств.

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.

1. Построение модели программы.
2. Задание требований в терминах выбранного типа темпоральной логики.
3. Верификация модели с целью проверки выполнения формализованных требований.
4. 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 таких программах обычно не разделяются управляющие и вычислительные состояния.

Для традиционных программ трудности 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 эффективные методы верификации программ указанного класса. Апробация указанных методов и экспериментальное исследование инструментальных средств, построенных на их основе, как отмечалось выше, является целью настоящего этапа работ.

Метрологическое обеспечение НИР не предусмотрено техническим заданием.

Дополнительные патентные исследования, проведенные в рамках четвертого этапа работы (отчет о патентных исследованиях № 2008.09.30-2 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.

В рамках работ было подготовлено три промежуточных отчета:

- Отчет по I этапу «Выбор направления исследований и базовых методов», инвентарный № 370095.1.
- Отчет по II этапу «Теоретические исследования поставленных перед НИР задач», инвентарный № 370095.2.
- Отчет по III этапу «Экспериментальные исследования поставленных перед НИР задач», инвентарный № 370095.3.

ОСНОВНАЯ ЧАСТЬ

1. ПРЕДЛОЖЕНИЯ И РЕКОМЕНДАЦИИ ПО ВЕРИФИКАЦИИ УПРАВЛЯЮЩИХ ПРОГРАММ СО СЛОЖНЫМ ПОВЕДЕНИЕМ, ПОСТРОЕННЫХ НА ОСНОВЕ АВТОМАТНОГО ПОДХОДА

В рамках второго этапа работ были разработаны методы верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода. Апробация этих подходов была осуществлена на примере моделей банкоматов в рамках третьего этапа работ. В процессе апробации были выявлены особенности разработанных методов, что позволяет сформулировать предложения и рекомендации по верификации управляющих программ сложным поведением на основе автоматного подхода.

В разд. 1.1 приводятся методические рекомендации по выбору метода верификации в зависимости от решаемой задачи. В разд. 1.2−1.5 рассматриваются предложения и рекомендации по верификации программ с применением разработанных методов.

1.1. Методические рекомендации по выбору метода верификации

Как показали эксперименты над прототипами дополнительных модулей верификаторов, методы, разработанные в данной работе, позволяют эффективно верифицировать автоматные модели управляющих программ. Каждый из разработанных методов имеет свою область применения.

Метод верификации, использующий верификатор *NuSMV*, позволяет верифицировать автоматные программы на соответствие спецификации, записанной в терминах темпоральной логики *CTL*. При использовании этого метода контрпример в виде пути в модели может быть непосредственно отображен в путь в системе конечных автоматов. Однако логика *CTL* не позволяет описывать условия на отдельные варианты выполнения программы, ограничивая верификацию только утверждениями, которые должны быть справедливы для всех путей выполнения программы. В то же время данный метод является наиболее простым из разработанных в настоящей работе.

Метод верификации визуальных автоматных моделей строит модель Крипке неявно. Поэтому для преобразования контрпримера в термины автоматной модели требуется дополнительная обработка протоколов работы верификатора. При этом из протоколов извлекается информация об изменении переменных состояний, по которым восстанавливается цепочка переходов, составляющая контрпример. Метод верификации визуальных автоматных моделей позволяет также верифицировать автоматные программы с объектами управления, имеющими состояния. В этом случае изменение состояния объектов управления описывается в терминах темпоральной логики. Аналогичным образом могут быть учтены зависимости между событиями.

Метод верификации на основе эмуляции вместо модели Крипке строит автомат Крипке, который, в свою очередь, может быть пересечен с автоматом Бюхи, построенным по проверяемым требованиям. При этом верификация может проводиться двумя способами: пошаговая эмуляция программы или формальное построение пересечения автоматов. В обоих случаях контпримером является путь в пересечении автоматов Крипке и Бюхи, по которому восстанавливается путь в автомате Крипке. За счет изоморфности автомата Крипке декартову произведению автоматов исходной модели путь в нем может быть преобразован в путь в исходной модели. Отметим, что в случае эмуляции специального описания возможных последовательностей событий и состояний объектов управления не требуется. Таким образом, этот метод является наиболее сложным из рассмотренных, но при этом и наиболее мощным.

При использовании метода верификации на основе темпоральной логики *CTL* модель Крипке строится в явном виде. Построенная таким образом модель содержит все состояния исходной автоматной модели, а также дополнительные состояния, получаемые в процессе преобразования. Таким образом, для восстановления трассы, соответствующей контрпримеру, требуется из трассы конпримера для модели Крипке удалить вершины, соответсвующие дополнительным состояниям.

Сравнительные характеристики разработанных методов приведены в табл. 1.

Таблица 1. Сравнительные характеристики разработанных методов верификации

Метод	Язык специфи-кации	Используемый верификатор	Постро-ение модели Крипке	Поддержка состояния объекта управления	Поддержка зависимостей между событиями
1. Метод верификации автоматных программ с использованием верификатора *NuSMV*	CTL	NuSMV	Неявная	Нет	Нет
2. Метод вери-фикации визу-альных авто-матных моделей	LTL	SPIN	Неявная	При специальном описании	При специальном описании
3. Метод вери-фикации на ос-нове эмуляции	LTL	Bogor	Автомат Крипке	Да	Да
4. Метод вери-фикации на ос-нове темпо-ральной логики CTL	CTL	—	Явная	Да	Нет

Из рассмотрения таблицы следует, что если требования могут быть сформулированы на языке *CTL* и объекты управления не имеют состояний, то следует использовать первый метод как наиболее простой. В случае наличия состояний у объектов управления, если условия на них могут быть выражены в термиминах языка *CTL*, то целесообразно использовать четвертый вариант.

В случае отсутствия состояний у объектов управления или возможности описания этих состояний в терминах *LTL*, рекомендуется применять метод верификации визуальных автоматных моделей. В противном случае, рекомендуется использовать метод верификации на основе эмуляции.

В общем виде технология верификации автоматных моделей управляющих программ может быть сформулирована в виде последовательности этапов:

1. Описание требований в терминах темпоральной логики для состояний автоматной модели.
2. Выбор метода верификации.
3. Описание изменения состояний объектов управления (при необходимости).
4. Запуск верификатора.

5. Если верификация завершилась неуспешно:
   а. Анализ контрпримера.
   б. Модификация автоматной модели.
   в. Переход к этапу 4.

Отметим, что в отличие от технологии верификации программ общего вида, при верификации автоматных программ на первом этапе требования записываются в терминах исходной автоматной модели. При этом не возникают ошибки перевода требований из терминов исходной программы в термины модели Крипке.

На втором этапе проводится выбор одного из разработанных методов верификации.

Третий этап необходим в случае применения методов верификации визуальных автоматных моделей и верификации на основе эмуляции.

В результате запуска верификатора на четвертом этапе либо подтверждается соответствие автоматной модели требованиям (в этом случае, процесс верификации оканчивается успешно), либо строится контрпример в терминах автоматной модели. Так как контрпример формулируется в терминах автоматной модели, а не в терминах модели Крипке, то и ошибки при восстановлении контрпримера из модели также отсутствуют.

При необходимости на пятом этапе производится модификация автоматной программы с целью устранения обнаруженной ошибки. После ее устранения процесс верификации повторяется.

Применение технологии верификации автоматных моделей, основанной на разработанных методах, должно позволить верифицировать автоматные модели управляющих программ, что чрезвычайно важно для ответственных систем со сложным поведением.

1.2. Метод верификации автоматных программ с использованием верификатора *NuSMV*

В разд. 1.1.3 отчета по второму этапу работ по контракту описывается метод верификации автоматных программ с использованием языка *SMV*. Далее приводится краткое описание данного метода с изменениями, внесенными после практического применения данного метода, описанного в отчете по третьему этапу. В настоящем разделе описывается методические рекомендации по применению данного метода.

1.2.1. Описание метода

При использовании метода верификации автоматных программ с использованием верификатора *NuSMV* [1] получает на вход систему автоматов и свойства на языке темпоральной логики. Метод обеспечивает проверку соответствия модели заданным свойствам и возвращает контрпример, если свойства системы нарушены. При этом задача верификации сводится к следующим подзадачам:

- преобразовать программу в модель на языке *SMV*;
- преобразовать требования к системе в формулы темпоральной логики;
- запустить программу-верификатор *NuSMV;*
- преобразовать контрпример к модели в контрпример в автоматной программе.

На рис. 1 изображена схема предлагаемого подхода.

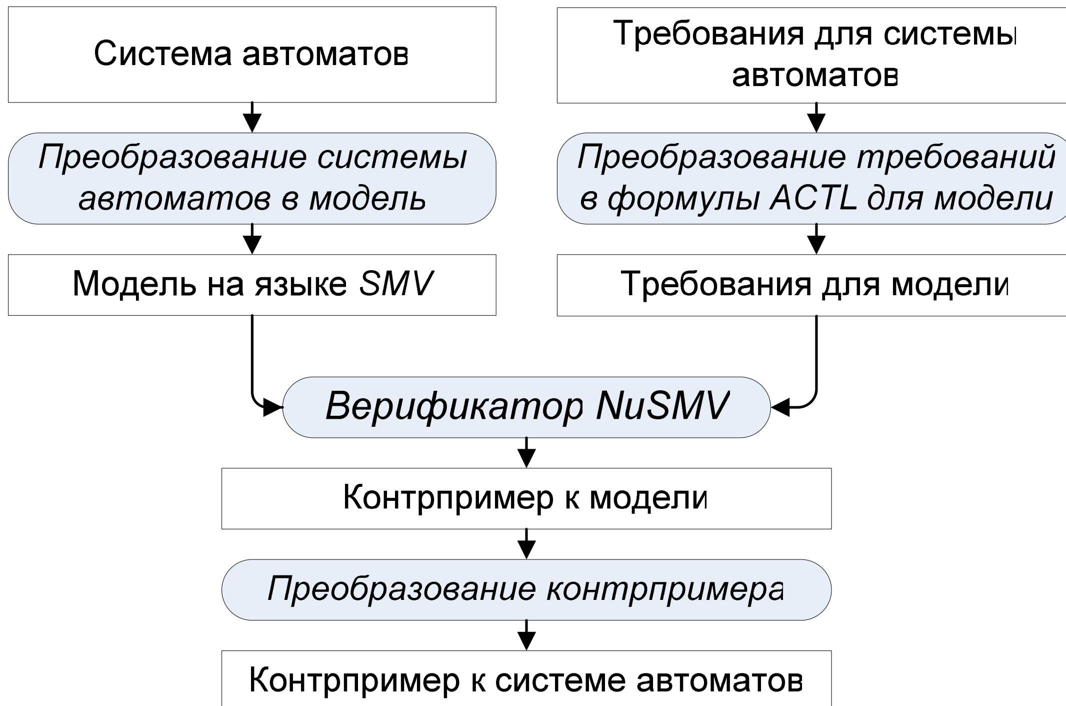

Рис. 1. Предлагаемый подход

1.2.1.1. Преобразование системы автоматов в модель

Построение модели выполняется в два этапа.

- так как система переходов допускает пометки только в вершинах, то для каждого автомата строится *модель автомата*. Вводятся дополнительные состояния на переходах, которые соответствуют действиям, выполняемым автоматом на переходах;
- полученные модели автомата объединяются в одну систему переходов и записываются как переменные и правила переходов между ними на языке *SMV*.

Рассмотрим задачу построения модели автомата. В автомате выделяются, помимо основных состояний, множество его промежуточных состояний, в которых автомат пребывает во время перехода из одного основного состояния в другое. В промежуточных состояниях будет храниться информация о том, какое действие автомат совершает. Промежуточное состояние автомата фиксируется каждый раз, когда автомат совершит одно из следующих действий:

- вызовет выходное воздействие. При этом в состоянии указывается, какое выходное воздействие вызывается;
- вызовет другой автомат. При этом в состоянии указывается, какой автомат и с каким событием вызывается.

В рамках используемого подхода состояния, в которых автомат возвращает управление, не выделяются. Выделяются дополнительные состояния на переходах. На переходах, на которых не выполняются действий, они используются для выделения перехода.

Состояниями модели автомата являются состояния исходного автомата и промежуточные состояния. Рассмотрим переход t из состояния s. Условие и событие, при котором переход t выполняется, записываются на переходе модели из состояния s в первое промежуточное состояние, выделенное на переходе t. Промежуточные состояния нумеруются так, чтобы начальное состояние автомата имело номер ноль. На рис. 2 приведен пример выделения промежуточных состояний для одного перехода.

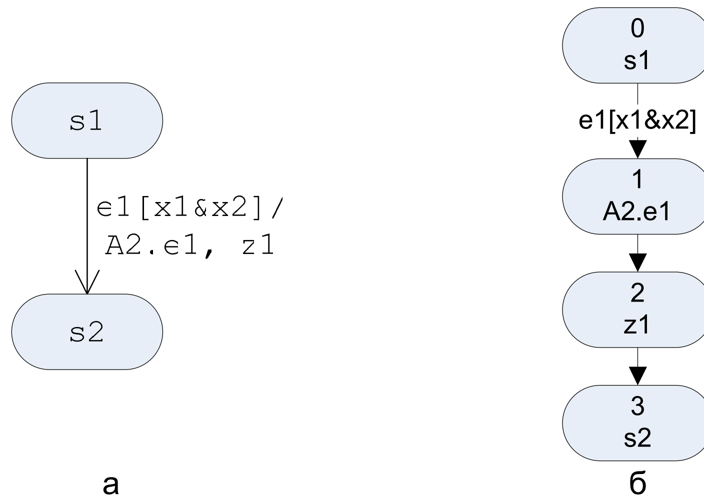

Рис. 2. Выделение промежуточных состояний на переходе.
Исходный переход (а), преобразованный переход (б)

Рассмотрим задачу преобразования нескольких моделей автоматов в систему переходов. Состояние модели будет описываться набором переменных. Каждому автомату A_k сопоставим переменную $State_k$. Для всей системы автоматов в целом введем переменные $Acite$, $Event$, xk $(k < 0 \leq m)$, где m – число входных переменных. Введенные переменные имеют следующий смысл:

- переменная $State_k$ содержит состояние модели, в котором может находиться каждая из n моделей. Каждая переменная может принимать значения от нуля до числа состояний модели.
- переменная *Active* содержит номер автомата, активного в данный момент или ноль, если никакой автомат не активен.
- переменная $Event$ содержит имя события, которое передано автомату в данный момент. Событие может быть передано от внешнего источника или от другого автомата в результате вызова. Если в данный момент никакое событие не передается, то переменная $Event$ принимает значение ноль.
- переменные x_1, x_2, x_m ... соответствуют входным воздействиям. Каждая переменная может принимать значения ноль (ложь) или один (истина).

Каждое состояние модели представляет собой объединение переменных $State_k$ $(1 \leq k \leq n)$, *Active*, $Event$, x_k $(k < 0 \leq m)$. Так как состояние задается набором переменных, то можно не строить систему переходов в явном виде. Таким образом, можно проверять модели с большим числом состояний.

Для того чтобы задать систему переходов, требуется выразить отношение между текущими и следующими значениями переменных. Зададим начальное значение всех переменных:

- переменные $State_k$ для всех k содержат состояния модели, соответствующие начальным состояниям всех автоматов;
- переменная *Active* – содержит ноль;
- переменная $Event$ – содержит ноль;
- переменные x_1, x_2, ... x_m – содержат входные воздействия, которые могут принимать значения ноль или единица.

Теперь зададим правила переходов. Для переменной $State_k$ запишем следующие правила переходов:

- если автомат активен *(Active = k)*, то изменение значения переменной $State_k$ выполняется в соответствии с переходами модели автомата *k*. Переменная $State_k$ может изменить свое значение на любое значение, в которое есть переход из текущего состояния в модели автомата, если условие, записанное на переходе, выполняется;
- если никакой переход невозможен, то значение переменной $State_k$ не изменяется;
- если автомат не активен, то значение переменной остается прежним.

Опишем изменение значений переменной *Active*. Возможно несколько вариантов:

- если переменная *Active* равна нулю. Это означает, что никакой автомат в данный момент времени не активен, а, следовательно, требуется выбрать автомат, который будет выполняться. Этот выбор происходит недетерминировано из множества всех номеров автоматов. После того, как автомат для выполнения выбран, его номер записывается в переменную *Active*;
- иначе переменная *Active* равна *k*. Это означает, что активен автомат *k*. Для определения значения переменной *Active* необходимо рассмотреть следующее состояние модели *k*. Это состояние содержится в переменной $State_k$:
  - если состояние $State_k$ – промежуточное состояние, и в нем вызывается автомат *l*, то, он будет активен на следующем шаге. Таким образом, переменной *Active* необходимо присвоить значение *l*;
  - если $State_k$ является состоянием исходного автомата, то, автомат *k* уже закончил переход. Проверим, нет ли автомата *l*, который вызвал автомат *k:*
    - если автомат *l* был вызван каким-то другим автоматом *m*, то управление требуется вернуть автомату *m*. Следующее значение переменной *Active* будет равно *m*;
    - иначе следующее значение переменной *Active* равно нулю.
  - иначе значение переменной *Active* не изменяется.

Переменная *Event* принимает следующие значения:

- если переменная *Active* равна *k* и следующее значение переменной $State_k$ соответствует вызову автомата с событием *e*, то следующее значение переменной *Event* равно *e*;
- иначе следующее значение *Event* равно нулю.

1.2.1.2. Преобразование системы автоматов в модель на языке *SMV*

Опишем преобразование моделей автоматов в модель на язык *SMV*. Каждая модель автомата размещается в отельном модуле с именем, соответствующим имени автомата. Каждый модуль будет иметь следующие параметры:

- `Active` – является ли данный модуль активным;
- `Event` – событие, переданное автомату;
- `Ap1, Ap2,...` – состояния экземпляров автоматов, с которыми данный модуль взаимодействует.
- `x1, x2,... -` входные воздействия.
- Определение модуля имеет следующий вид:
  `MODULE name(Active, Event, Ap1, Ap2, …, x1, x2, … xm)`

Каждый модуль должен хранить номер текущего состояния модели в переменной *State*. Она описывается как *State: 0..p;*. Здесь *p* – число состояний модели данного автомата.

Далее требуется указать правила, по которым будут изменяться значения переменных. Эти правила описываются в секции `ASSIGN`.

В начале значение переменной *State* равно нулю. Это записывается как `init(State) = 0;` Для каждого состояния модели указывается в какие состояния модель может переходить. Это делается при помощи ключевого слова `TRANS`. После него записывается предикат, истинность которого означает наличие перехода. Раннее предлагалось использовать оператор `ASSIGN`, однако он не удобен для реализации недетерминированных переходов. Для того чтобы составить такой предикат запишем условие для каждого перехода, объединив их операцией *логическое или*. Для каждого перехода из состояния s_i в состояние s_j, который происходит по событию e_{ij} при условии c_{ij}, записывается выражение:

```
Active & State = si & next(State) = sj & Event = eij & cij
```

Для каждого состояния модели s_i запишем условие того, что ни одни переход из данного состояния невозможен:

```
Active & State = si & next(State) = si & !(Event = ei1 & ci1) &
        !(Event = ei2 & ci2) & ...
```

Если автомат не активен, то состояние не изменяется:

```
(!Active & State = next(State)
```

Далее требуется выразить основные состояния автоматов. Это записывается в секции `DEFINE`:

```
DEFINE
  s1 := State = n1;
  s2 := State = n2;
. . .
```

Далее выразим свойство, что модель находится в состоянии автомата:

```
inState := State = n1 | State = n2 | …;
```

Здесь s_1, s_2, ... – имена состояний автомата, а n_1, n_2, ... – соответствующие им номера состояний модели.

Переменные, общие для всей системы, записываются в модуле *main* после ключевого слова `VAR`:

- переменные x_k, описывающие входные воздействия, записываются как `xk: {0, 1};`
- переменные, описывающие экземпляры автоматов, записываются как `name: name(Active = k, Event, Ap1, Ap2, …, x1, x2, … xm)`, где `name` – имя автомата `k`; `Ap1`, `Ap2` – имена автоматов, с которыми он взаимодействует; `x1`, `x2`, ..., `xm` – входные воздействия;
- переменная *Event* описывается следующим образом: `Event: {0, e1, e2, e3, …};`

Зададим значения переменных. В частности, начальное значение переменной *Active*:

```
init(Active) := 0;
```

Определим следующее значение переменной *Active*:

```
next(Active) := case
```

Если все автоматы не активны, то выберем любой автомат:

```
Active = 0: 1..n;
```

Для всех состояний модели s_k и всех моделей A_k, в которых вызывается автомат:

```
(Active = k & next(Ak.State) = sk): l;
(Active != k & Ak.State = sk & Al.inState): k;
```

Для всех автоматов запишем условие, что он вернет управление, когда закончит переход:

```
(Active = k & next(Ak.inState)): 0;
```

Иначе значение переменной *Active* не изменяется:

```
  1: Active;
esac
```

Далее записывается описание переменной *Event*. Начальное значение переменной *Event*:

```
init(Event) := 0;
next(Event) := case
```

Для всех вызовов автоматов с событием e_k записывается:

```
Active = k & next(Ak.State) = sk: ek;
```

Иначе значение переменной $Event = 0$:

```
  1: 0;
esac;
```

Для каждого выходного воздействия z_k запишем:

```
DEFINE
  Zk = (Active = k1 & A1.State = s1) |
    (Active = k2 & A2.State = s2) | ...
```

1.2.1.3. Запись требований

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

Введем обозначения, которые будут компоненты состояния автоматов:

- условие, что автомат A_k находится в состоянии s_j, записывается как `Ak.sj`;
- условие, что выполнилось выходное воздействие z_1, записывается как `z1`;
- условие, что произошло событие e_i, записывается как `Event = ei`;

Для записи формул, описывающих состояния автомата, можно использовать логические операторы. Если *f* и *g* – формулы состояния, то формулами состояния являются:

- `f & g` – одновременно выполняются `f` и `g`;
- `f | g` – выполняется либо `f` либо `g`;
- `f xor g` – выполняется либо `f` либо `g`, но не одновременно;
- `!f` – не выполняется `f`;
- `f -> g` – если выполняется `f`, то выполняется `g`;
- `f <-> g` – тоже что и `(f -> g) & (g -> f)`.

Помимо свойств текущего состояния в условиях можно использовать темпоральные операторы: *AF*, *AF*, *AG*. Оператор *AX* не используются для записи свойств автоматов, так как в данной модели один переход автомата соответствует неопределенному числу переходов модели. Опишем используемые операторы:

- `AF f` (*Future*) – оператор означает, что на всех путях из текущего состояния существует состояние, когда формула `f` выполнится.

- `AG f` (*Global*) – оператор означает, что данная формула `f` будет выполняться на каждом пути из текущего состояния в каждом состоянии: формула `f` будет выполняться в каждом состоянии, достижимом из текущего состояния.
- `A[f U g]` (*Until*) – оператор истинен, если на каждом пути когда-нибудь выполнится формула *g*, а до этого момента всегда будет выполняться формула *f*.

1.2.1.4. Преобразование контрпримера

Если опровергаемая формула принадлежит *ACTL*, то при её невыполнении получим контрпример в системе переходов. Любой контрпример для модели является либо конечным путем, либо путем с конечным началом и циклом.

Каждое состояние является набором переменных $Event$, $State_k$, *Active*, *xk* ($0 \le k < m$). Приведем алгоритм для преобразования контрпримера к системе переходов в контрпример к системе автоматов. Контрпример к системе автоматов также представляется в виде последовательности состояний, только вместо значений переменных информация представляется в терминах состояний и действий. Для каждого состояния контрпримера выведем следующую информацию о системе автоматов.

Если *Active* = 0, то никакой автомат не активен. Данное состояние не учитывается в контпримере для автомата.

Если *Active != 0*, то выводится название активного автомата. Также выводится действие активного автомата. Выводятся состояния всех автоматов, полученные из переменных $State_k$. Аналогично выводятся значения всех входных воздействий, записанных в переменные x_k, и значение переменной $Event$.

1.2.2. Инструментальное средство *FSMVerifier*

Метод построения модели Крипке по автоматной модели позволяет проверять систему автоматов со следующими свойствами. Каждый автомат является автоматом Мили. Автоматы могут взаимодействовать следующим образом:

- они могут получать информацию о состояниях других автоматов;
- один автомат может передавать управление другому автомату, предавая ему событие.

На переходе автомата может указываться:

- событие, при котором он происходит;
- условие, при котором он происходит. В условии в качестве атомарных условий можно использовать входные переменные *x1*, *x2*, ... и выражения вида *Ai.sij* (оно истинно, если автомат *Ai* находится в состоянии *sij*);
- последовательность действий. Она может содержать выходные воздействия *zi* и передачу управления другим автоматам *Ai.ej*.

Достоинством метода верификации автоматных программ с использованием инструментального средства *FSM Verifier*, использующего верификатор *NuSMV*, является линейный рост описания модели на языке *SMV* относительно сложности системы автоматов. Сложностью системы автоматов будем называть сумму числа состояний всех автоматов, их переходов и информации, которая записана на каждом из переходов. Число переменных в модели возрастает линейно от числа автоматов.

Более подробно данное инструментальное средство описано в разд. 2.1.

1.3. Метод верификации визуальных автоматных моделей

Метод предназначен для верификации визуальных [8] автоматных программ [9, 10] с применением верификатора *SPIN* [11].

1.3.1. Описание метода

В качестве формализма для проверяемых свойств используется язык линейной темпоральной логики *LTL* [12, 13]. Метод позволяет верифицировать системы автоматов Мура или Мили, а также смешанного типа.

При использовании метода модель Крипке строится неявно. Поэтому для преобразования контрпримера в термины автоматной модели требуется дополнительная обработка протоколов работы верификатора. При этом из протоколов извлекается информация об изменении переменных состояний, по которым восстанавливается цепочка переходов, составляющая контрпример. Метод верификации визуальных автоматных моделей позволяет также верифицировать автоматные программы с объектами управления, имеющими состояния. В этом случае изменение состояния объектов управления описывается в терминах темпоральной логики. Аналогичным образом могут быть учтены зависимости между событиями.

В данном методе по визуальной автоматной программе строится модель, которую верифицирует инструментальное средство *SPIN*.

Для отображения контрпримера в удобном для человека виде в методе предусматривается внесение в модель пометок.

1. По разработанному алгоритму или программе вручную строится модель на языке *Promela* – входном языке верификатора *SPIN*.
2. Разрабатываются и записываются на языке *LTL* [13] требования (спецификация) к модели, которые верификатор преобразовывает в *автомат Бюхи* [1], записанный на языке *Promela*.
3. Проводится автоматическая верификация построенной модели с помощью верификатора *SPIN*. Верификация построенной модели проходит в несколько этапов:
   - Запускается верификатор с ключом `-a`. В качестве параметра верификатор принимает файл с построенной моделью на языке *Promela*. Верификатор строит программу *pan*.
   - Запускается программа *pan*. Программа преобразует модель на языке *Promela* в *модель Крипке* [5, 11] и верифицирует ее.
   - Запускается верификатор *SPIN* с ключом `-t` для вывода контрпримера в случае несоответствия модели требованиям. Ключ `-p` может применяться для вывода полной информации о контпримере.
4. Если модель не соответствует требованию, то проводится анализ контрпримеров. Возможен случай возникновения «ложных опровержений» – ошибка находится не в алгоритме, а в модели. В этом случае требуется изменить модель. Кроме того, модель может быть слишком большой и верификатор не справится с ее верификацией. В этом случае требуется уменьшить модель.

1.3.2. Инструментальное средство *Converter*

Инструментальное средство *Converter* – это верификатор визуальных автоматных программ, использующий для верификации инструментальное средство *SPIN*. Визуальные автоматные программы должны быть разработаны при помощи инструментального средства *UniMod* [25].

В табл. 2 приведен список условных и темпоральных операторов, которые можно использовать при записи свойств на языке *LTL* и их соответствие стандартным обозначениям.

Таблица 2. Операторы языка *SPIN*

Оператор	Описание
`[]`	Globally, всегда
`<>`	Future, когда-нибудь в будущем
`!`	отрицание
`U`	Until, до тех пор, пока. Запись *p U q* означает, что *p* будет верно до тех пор, пока не выполнится *q*. При этом *q* обязано когда-либо выполниться.
`V`	Запись *p V q* эквивалентно *!(!p U !q)*
`&&`	Логическое И
`||`	Логическое ИЛИ
`->`	Следует
`<->`	Эквивалентно

Также в *LTL*-формулах можно использовать следующие переменные:

- *lastEvent* для обозначения последнего на данный момент произошедшего события;
- для каждого автомата *A* переменную *stateA* для обозначения текущего состояния автомата *A*.

1.3.3. Методические рекомендации по применению инструментального средства *Converter*

Инструментальное средство *Converter* характеризуется следующими особенностями.

- инструментальное средство *Converter* 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, и с источниками событий и объектами управления в виде *Java*-классов;
- каждый автомат может быть как автоматом Мура или Мили, так и смешанным автоматом;
- проверяемые свойства формализуются на языке *LTL*;
- инструментальное средство *Converter* позволяет верифицировать только управляющую систему автоматов, абстрагируясь от источников событий и объектов управления. Иначе говоря, ошибки, содержащиеся в источниках событий и объектах управления, не будут найдены. Таким образом, рекомендуется сократить логику до минимума в объектах управления и источниках событий.

Более подробно это инструментальное средство описано в разд. 2.2.

1.4. Метод верификации на основе эмуляции

Метод эмуляции предложен в ходе выполнения работ по контракту и подробно описан в отчете за второй этап в разд. 1.3. Этот метод предназначен для верификации темпоральных формул, выраженных в темпоральной логике *LTL* [5] .

1.4.1. Описание метода

Основой метода является алгоритм двойного обхода в глубину [5]. Данный алгоритм позволяет верифицировать *LTL*-формулы для любых программ, для которых возможно выполнение следующих действий:

1. Вычисление глобального состояния программы. Это состояние однозначно определяет поведение программы.
2. Совершение элементарного шага работы программы. Элементарный шаг является переходом программы из одного глобального состояния в другое без посещения иных глобальных состояний.
3. Откат элементарного шага работы программы. При этом программа возвращается в предыдущее глобальное состояние.
4. Определение в каждом состоянии возможных элементарных шагов.
5. Определение значения набора предикатов программы, используемых в темпоральной формуле.

Как отмечено выше, метод эмуляции использует алгоритм двойного обхода в глубину для верификации. При этом необходимые для алгоритма действия, перечисленные выше, вызываются у интерпретатора автоматной программы. Таким образом, метод эмуляции работает не с автоматной программой, а с ее интерпретатором. Поэтому этот метод называется методом «эмуляции», поскольку он «эмулирует» работу интерпретатора с автоматной программой и при этом осуществляет верификацию.

Подробно опишем решение пяти задач, перечисленных выше, решаемых при верификации автоматной программы.

Выделим верифицируемую программу и внешнюю среду. Внешняя среда неуправляема программой, а ею управляет верификатор. Его цель – найти такие внешние условия для программы, при которых верифицируемые требования не будут выполнены. Задача программы, в свою очередь, удовлетворять требованиям при любых условиях.

Выделим верифицируемую систему и внешнюю среду для автоматной программы. Поскольку ставится задача верификации лишь автоматной системы управления, то объекты управления и источники событий выносятся во внешнюю среду. Автоматная система получает из внешней среды события и значения входных переменных. В свою очередь, во внешнюю среду она выдает команды объектам управления. Данная идея иллюстрируется рис. 3.

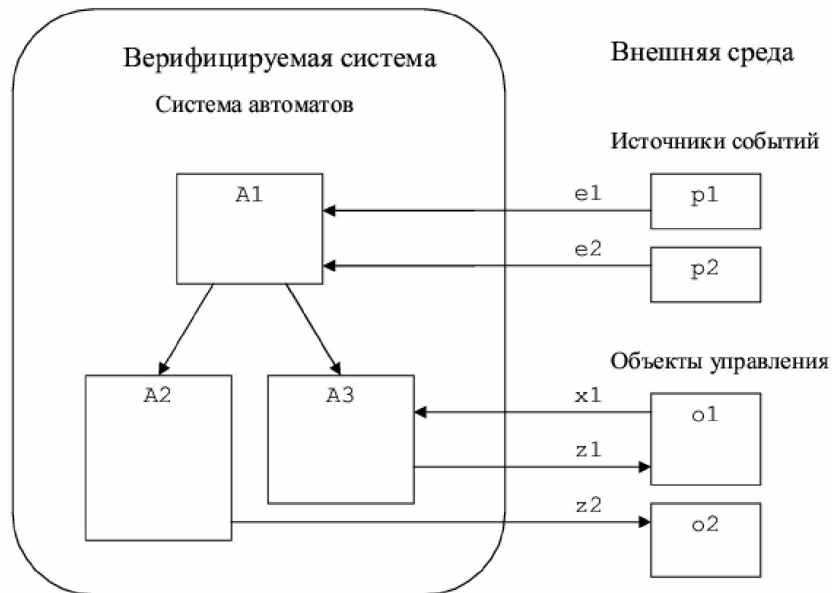

Рис. 3. Общая схема верифицируемой системы

Из такого определения глобального состояния программы непосредственно вытекает определение элементарного шага работы автоматной системы – это обработка события, поскольку в результате нее может измениться набор состояний автоматов. Вообще говоря, можно было бы выполнить более мелкое дробление: например, считать за элементарный шаг обработку события одним автоматом. Или еще более мелко: принять за элементарный шаг мелкое действие при переходе, такое как запрос значения переменной объекта управления, вызов действия или передача управления дочернему автомату. Однако при более мелком дроблении увеличилось бы число состояний верифицируемой программы, а вместе с тем, и время верификации.

Откат элементарного шага работы автоматной программы реализуется достаточно просто: поскольку поведение автоматной программы определяется набором состояний автоматов, то для отката достаточно вернуть автоматы в те состояния, в которых они находились до выполнения шага.

Метод эмуляции предназначен для работы с интерпретатором, в котором для автоматной программы строго определена схема работы системы автоматов, определена последовательность передачи управления между автоматами, и интерпретатор работает в одном потоке. Поэтому недетерминированность в интерпретации автоматной программы возникает только в результате неопределенности сигналов, поступающих из внешней среды, изображенной на рис. 3. Другими словами, недетерминированность порождается неопределенностью последовательности событий и значений условий на переходах автоматов в различные моменты времени. Тогда возможные элементарные шаги автоматной программы определяются набором возможных событий и набором возможных значений переменных, запрашиваемых у объектов управления.

В методе эмуляции недетерминированность в работе интерпретатора учитывается следующим образом. Перед совершением очередного элементарного шага у интерпретатора запрашивается набор событий, которые автоматная программа обрабатывает в текущем глобальном состоянии. Этот набор складывается из событий, которыми помечены переходы из текущего состояния главного автомата, событий, которыми помечены переходы из текущих состояний автоматов, вложенных в текущее состояние главного автомата, и т.д. Полученный список определяет недетерминированность по событию: при возникновении разных событий из этого списка система поведет себя по-разному. Для того чтобы перебрать все возможные варианты, сначала выбирается первое событие из списка и интерпретатор выполняет его, как будто оно возникло в ходе выполнения автоматной программы. При этом

делается пометка, что в текущем глобальном состоянии было выполнено лишь первое из списка возможных событий. При откате обратно в это глобальное состояние метка будет прочитана, и второе событие из списка будет подано на обработку интерпретатором, как будто оно возникло в ходе интерпретации автоматной программы. Таким образом, будут проверифицированы уже две возможные истории развития событий. Так будет повторяться до тех пор, пока не будут обработаны все возможные события для данного глобального состояния.

В ходе обработки события может возникнуть необходимость вычислить значение условия на одном из переходов, в котором может участвовать переменная объекта управления. Поскольку объекты управления находятся во внешней среде, это условие недетерминированно принимается равным `True` или `False`. Опять же, при откате в текущее глобальное состояние, верификатор вновь запустит обработку того же события, но с новым значением условия на переходе (это приведет к выбору автоматом другого перехода).

При выборе значений для условий контролируется, чтобы они не противоречили друг другу. Например, если из текущего состояния автомата есть два перехода, помеченных условиями `[x]` и `[!x]`, то невозможно, чтобы оба эти условия были приняты за `False`, поскольку они заданы одной переменной.

В методе эмуляции при совершении интерпретатором шага автоматной программы, вся информация о происходящих действиях записывается и используется затем для вычисления значений предикатов – элементарных утверждений, записанных в требованиях. Предикаты применяются для формирования требований к автоматной программе. Поэтому для того, чтобы подробно специфицировать работу системы, требуются предикаты, описывающие возникающие события, вызываемые действия объектов управления, значения переменных объектов управления и состояния автоматов – все, что характеризует работу автоматной программы. Для этого при совершении шага записывается следующая информация:

- состояния автоматов до выполнения шага;
- обрабатываемое событие;
- список вычисленных значений условий на переходах;
- список вызванных действий у объектов управления;
- список совершенных переходов (это, кроме того, позволяет получить состояния автоматов после выполнения шага).

Сохраняемая информация позволяет вычислять значения предикатов. В методе эмуляции поддерживается следующий набор предикатов:

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

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

Перечисленные предикаты позволяют формулировать требования к автоматной программе. Например, требование «если произошло событие `e1` и при этом выполнялось условие `x1`, то будет выполнено действие `z1`» может быть записано в виде:

```
wasEvent(e1) && wasTrue(x1) → wasAction(z1)
```

Повторим еще раз, как метод эмуляции позволяет решать пять задач, возникающих при верификации автоматной программы с помощью алгоритма двойного обхода в глубину.

1. Глобальное состояние автоматной программы задается набором состояний ее автоматов.
2. Элементарным шагом программы является обработка автоматной программой одного события.
3. Элементарный шаг откатывается путем установки автоматов в исходные состояния.
4. Перебор возможных историй работы программы происходит за счет выбора всех возможных событий и выбора всех возможных значений условий на переходах.
5. Предикаты описывают события, действия и значения переменных, полученные в последнем шаге.

1.4.2. Инструментальное средство *UniMod.Verifier*

Программное средство *UniMod.Verifier* – это верификатор автоматных программ, использующий метод эмуляции. В качестве интерпретатора автоматных программ *UniMod.Verifier* использует инструментальное средство *UniMod* [2, 3], которое позволяет создавать и запускать автоматные программы, спроектированные в виде набора *UML*-диаграмм. Исходные коды инструментального средства *UniMod* написаны на языке *Java* и открыты для свободного доступа, что позволяет интегрировать его с другими программами. Для работы инструментального средства *UniMod.Verifier* было создано дополнение к средству *UniMod*, позволяющее управлять работой его интерпретатора. С помощью этого дополнения стало возможным запрашивать у интерпретатора текущее глобальное состояние автоматной программы, совершать обработку заданного события, откатываться в предыдущее глобальное состояние, запрашивать свойства автоматной программы для вычисления значений предикатов и т.д. Таким образом, созданное дополнение к инструментальному средству *UniMod* позволяет делать с его интерпретатором все необходимое для работы метода эмуляции.

В верификаторе *UniMod.Verifier* не был реализован алгоритм двойного обхода в глубину. Вместо этого был использован верификатор *Bogor* [21, 22], который использует этот алгоритм для верификации программ. *Bogor* – верификатор с модульной структурой и расширяемым входным языком *BIR – Bogor Input Language*. Благодаря расширяемости

входного языка верификатора *Bogor* появилась возможность интеграции верификатора с дополнением к инструментальному средству *UniMod*. Таким образом, алгоритм двойного обхода в глубину, реализованный в верификаторе *Bogor*, через дополнение работает с интерпретатором инструментального средства *UniMod*, для того, чтобы верифицировать автоматную программу. Все это вместе является верификатором *UniMod.Verifier* (рис. 4).

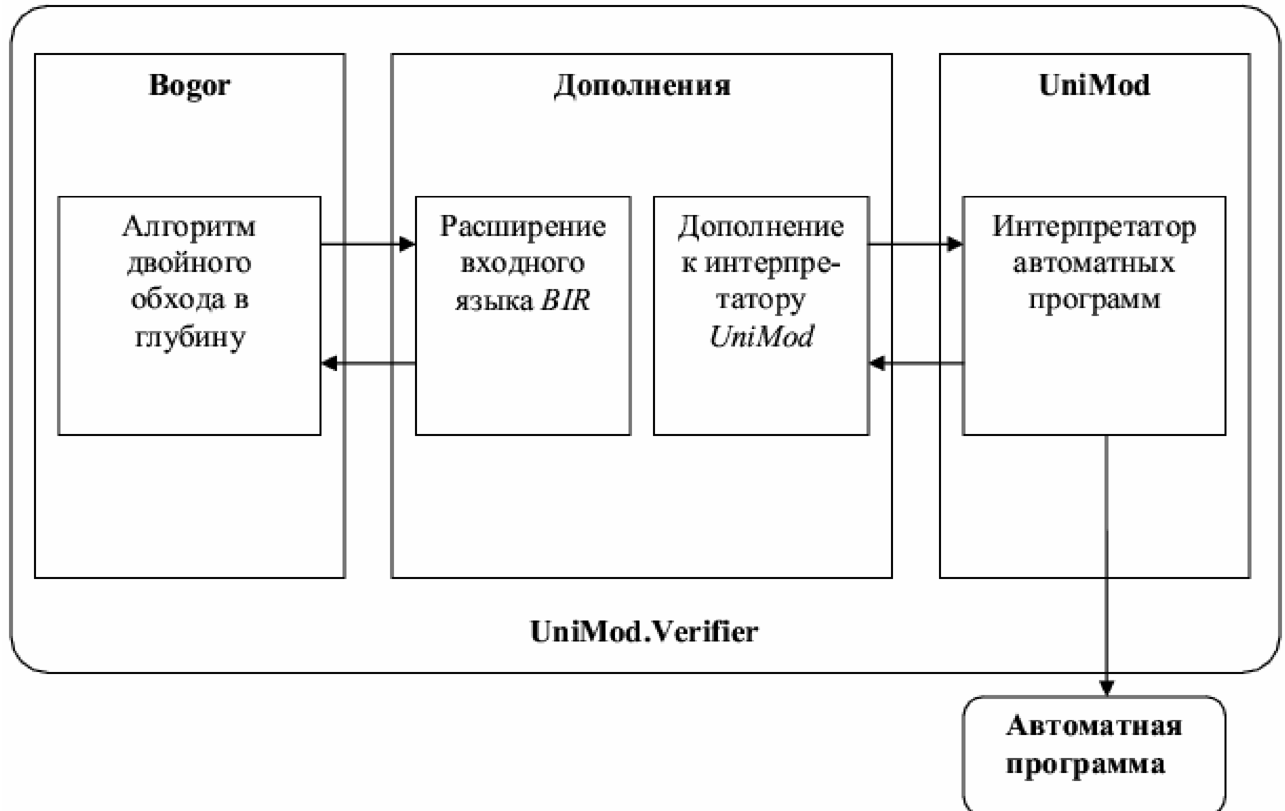

Рис. 4. Схема взаимодействий компонент инструментального средства *UniMod.Verifier*

Входной язык верификатора *Bogor* был расширен новым типом – моделью системы автоматов. Над этой моделью можно совершить лишь одно действие, называемое `step` – «совершить один элементарный шаг работы системы автоматов». При его выполнении выбирается очередное событие и происходит его обработка в системе автоматов. Также у модели можно запросить значения предикатов, определенных в методе эмуляции, как, например, `wasEvent`, `isInState` и т.д. Созданный тип был назван `AutomataModel`.

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

```
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: состояния `init`, в котором 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. Отметим, 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* существует 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"))
  )
);
```

Такие формулы во время верификации преобразуются в автомат Бюхи, как этого требует алгоритм двойного обхода в глубину [5]. Например, приведенная выше *LTL*-формула 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 Бюхи, названия которых начинаются с «`bad$`», являются допускающими [5].

1.4.3. Методические рекомендации по применению инструментального средства *UniMod.Verifier*

Инструментальное средство *UniMod.Verifier* характеризуется следующими особенностями:

- *п*озволяет верифицировать любые автоматные модели, реализованные при помощи инструментального средства *UniMod*;
- инструментальное средство *UniMod*, в свою очередь, позволяет создавать автоматные программы, представляющие собой реактивные системы, с управляющей компонентой в виде иерархически связанной системы автоматов,

и с источниками событий и объектами управления, реализованными в виде *Java*-классов;

- 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;
- позволяет задавать верифицируемые свойства автоматной модели, сформулированные в виде формулы темпоральной логики *LTL*;
- в верифицируемой формуле может использоваться заданное множество предикатов, описывающих аспекты работы автоматной программы, например, происходящие события, вызываемые действия объектов управления и состояния автоматов. Полный список этих предикатов перечислен в разд. 1.4;
- каждый предикат вычисляется только один раз после обработки каждого события. Формулировать точные утверждения о происходящем внутри перехода невозможно. Например, если требуется проверить, что переход `t1` одного автомата всегда происходит сразу после перехода `t2` другого автомата, и оба перехода могут произойти в пределах обработки одного события, то верификация такого утверждения может оказаться некорректной. Существует, однако, способ проверять последовательность выполнения действий даже в пределах обработки одного события, используя специальный предикат `getActionIndex`(...);
- предикаты `wasTrue` и `wasFalse` описывают вычисленные значения *условий* на переходах, а не переменных. Таким образом, если условие `!o1.x1` было вычислено как `True`, то предикат `wasTrue(!o1.x1)` будет верным, однако предикат `wasFalse(o1.x1)` не будет верным;
- 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 содержится в объектах управления.

Поясним подробнее последнее утверждение. При использовании рассматриваемого подхода верифицируется изолированная автоматная модель: источники событий и объекты управления рассматриваются как внешняя среда. При этом считается, что внешняя среда непредсказуема, а, следовательно, всегда возможно возникновение любого события или любого значения переменной объекта управления. Однако часто в объектах управления и источниках событий имеется логика, и в некоторые моменты времени невозможно возникновение некоторых событий или некоторых значений переменных.

Например, объект управления может являться источником событий — пусть, действие `z1` создает событие `e1`. В этом случае верификация формулы

```
G (z1 -> X e1)
```

(всегда после действия `z1` происходит событие `e1`) может завершиться неудачей, поскольку верификатору неизвестно о зависимости между этим действием и событием. Полученный сценарий ошибки называется «наведенной» ошибкой, поскольку реально ее не существует в рабочей автоматной программе, однако верификатор ее находит.

Тем не менее, можно бороться с наведенными ошибками путем внесения зависимостей прямо в *LTL*-формулу. Предположим, что верифицируется некая формула, и ее верификация может оказаться неточной из-за зависимости, приведенной выше. Тогда можно заменить *LTL*-формулу на следующую:

```
(G (z1 -> X e1)) -> <исходная формула>
```

Эту формулу можно истолковать, как «если выполняется зависимость, то верна и формула». Тогда любые сценарии, неудовлетворяющие первой половине формулы, будут удовлетворять всей формуле. Таким образом, можно избавиться от всех наведенных ошибок.

Более подробно Инструментальное средство *UniMod.Verifier* описано в разд. 2.3.

1.5. Метод верификации на основе темпоральной логики *CTL*

Метод верификации на основе темпоральной логики *CTL* разработан в ходе работ по контракту и подробно описан в разд. 1.1 отчета по второму этапу.

1.5.1. Описание метода

Рассмотрим программу, модель которой задаётся системой автоматов, взаимодействующих по вложенности. Необходимо преобразовать эту модель в единую модель Крипке, целиком описывающую поведение заданной системы. Прежде всего, для множества автоматов выполняется топологическая сортировка по отношению вложенности. Данное отношение не должно содержать циклов, иначе полученная модель имела бы бесконечный размер. Модель Крипке строится индуктивно для каждого автомата системы, причём автоматы обрабатываются в порядке, который был сформирован топологической сортировкой. Такой порядок означает, что перед обработкой внешних автоматов вложенные в них автоматы уже будут обработаны (для них будет построена модель Крипке).

Опишем алгоритм построения модели Крипке для индивидуального автомата в предположении, что модели Крипке для всех вложенных автоматов уже построены. Будем пользоваться терминологией, введённой в разд. 2.2.1.1 отчёта по третьему этапу.

В методе редукции графа переходов множество *AP* равно $\{\mathtt{Y}_1, \mathtt{Y}_2, \ldots\} \cup \{\mathtt{e}_1, \mathtt{e}_2, \ldots\} \cup \{\mathtt{x}_1, \mathtt{x}_2, \ldots\} \cup \{!\mathtt{x}_1, !\mathtt{x}_2, \ldots\} \cup \{\mathtt{z}_1, \mathtt{z}_2, \ldots\} \cup \{\mathtt{InState}, \mathtt{InEvent}, \mathtt{InAction}\} \cup Names$.

Здесь $\{\mathtt{Y}_1, \mathtt{Y}_2, \ldots\}$ означает множество наименований всех состояний автоматов, $\{\mathtt{e}_1, \mathtt{e}_2, \ldots\}$ – событий, $\{\mathtt{x}_1, \mathtt{x}_2, \ldots\}$ – входных воздействий, а $\{\mathtt{z}_1, \mathtt{z}_2, \ldots\}$ – выходных воздействий. *Names* – это множество наименований самих автоматов, а `InState`, `InEvent` и `InAction` – управляющие автомарные предложения, предназначенные для того, чтобы было удобно различать позиции построенные из состояний, событий и выходных воздействий.

Модель Крипке будем строить по частям: вначале построим те части, которые соответствуют состояниям автомата (будет необходимо обработать выходные воздействия и автоматы, вложенные в эти состояния), а потом добавим туда информацию о переходах. На первом шаге положим множество *S* равным множеству состояний исходного автомата, и для каждого состояния `s` добавим в отношение *Label* две пометки: (`s`, `s`) и (`s`, `InState`).

После этого для каждого состояния `s` выполним следующую операцию. Пусть `s` содержит выходные воздействия $\mathtt{z}_{\mathtt{s[1]}}, \ldots, \mathtt{z}_{\mathtt{s[u]}}$, которые выполняются при входе в `s`. Добавим в модель *u* состояний $\{\mathtt{r}_1, \ldots, \mathtt{r}_\mathtt{u}\}$ и *u* переходов $\mathtt{r}_1 \to \mathtt{r}_2, \ldots, \mathtt{r}_{\mathtt{u-1}} \to \mathtt{r}_\mathtt{u}, \mathtt{r}_\mathtt{u} \to \mathtt{s}$. В отношение *Label* добавим пометки $(\mathtt{r}_\mathtt{k}, \mathtt{z}_{\mathtt{s[k]}})$, $(\mathtt{r}_\mathtt{k}, \mathtt{InAction})$ для всех *k* от 1 до *u*. При добавлении рёбер в модель на следующих этапах, каждое ребро, идущее в `s`, будем перенаправлять в $\mathtt{r}_1$.

Пример такого преобразования приведен на рис. 5.

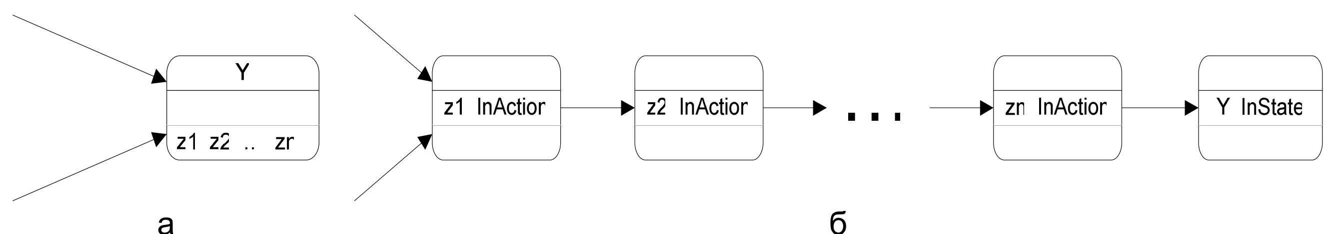

Рис. 5. Состояние с выходными воздействиями до (а) и после (б) преобразования

Эту операцию назовём разделением выходных переменных и состояний. Теперь следует отделить состояния от вложенных в них автоматов.

Пусть в состояние Y внешнего автомата вложены автоматы $A_{Y,1}$, $A_{Y,2}$, …, $A_{Y,v}$ (для них модели Крипке уже построены по индуктивному предположению). В модели Крипке для внешнего автомата (которая ещё строится) уже присутствует позиция, соответствующая состоянию Y. Для каждого из автоматов $A_{Y,1}$, $A_{Y,2}$, …, $A_{Y,v}$ добавим его модель Крипке к строящейся модели Крипке внешнего автомата. Под добавлением понимается, что необходимо скопировать во внешнюю модель Крипке все состояния, переходы и пометки внутренней модели Крипке. Для каждого состояния внутренней модели Крипке создадим уникальное, соответствующее только ему, состояние внешней модели Крипке. Во внешнее отношение переходов $\to$ добавим переходы между теми состояниями, которые соответствовали всем переходам между состояниями внутренней модели. Аналогично поступим и с помечающим отношением *Label*.

После того, как внутренние модели Крипке будут скопированы во внешнюю модель Крипке, добавим в неё также *v* переходов: для каждого *i* от 1 до $v-1$ добавим в отношение $\to$ переход из терминальной позиции модели Крипке для автомата $A_{Y,i}$ в стартовую позицию модели Крипке для автомата $A_{Y,i+1}$, и один переход из терминальной позиции модели Крипке для автомата $A_{Y,v}$ в позицию, соответствующую состоянию Y. Если до этого в позицию Y вели какие-либо рёбра, то все они перенаправляются в стартовую позицию модели Крипке для автомата $A_{Y,1}$.

Пример такого преобразования приведен на рис. 6.

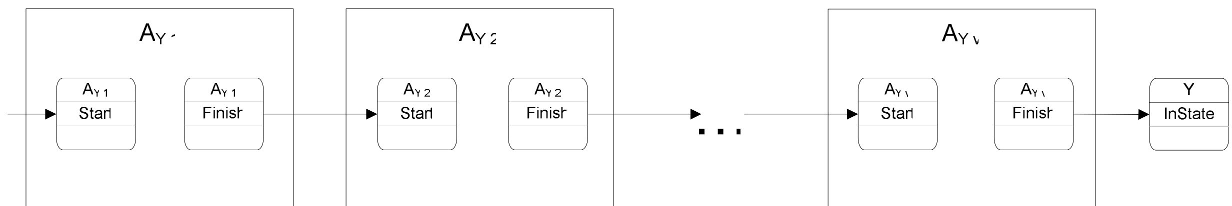

Рис. 6. Обработка автоматов, вложенных в состояние

Обратим внимание, что в различные состояния автомата *A* могут быть вложены различные «копии» одного и того же автомата *B*. В этом случае для каждой копии создаётся своя модель Крипке (все эти модели изоморфны друг другу), и перенаправление рёбер выполняется для неё. Размер полученной модели Крипке (число состояний в ней) будет ограничен сверху произведением размеров моделей Крипке для каждого автомата в отдельности (без учёта вложенных).

На данном этапе закончена обработка состояний автомата. Поэтому перейдём к описанию того, как обрабатываются переходы.

Рассмотрим множество следующих символов: $\{x_1, !x_1; x_2, !x_2; x_3, !x_3; \ldots\}$. Это множество всех литералов, составленных из входных переменных. Следует различать смысл знаков $\neg$ и !. Первый из них означает выполнение операции логического отрицания, а второй интерпретируется как символ (часть строки $!x_i$).

Тогда для каждого ребра r исходного автомата, который ведет из состояния p в состояние q с пометкой e_i & $h_{j[1]}$ & $h_{j[2]}$ & $h_{j[3]}$ & … $h_{j[m]}$ / $z_{i[1]}$, …, $z_{i[n]}$, где либо $h_{j[j*]} = x_{j[j*]}$, либо $h_{j[j*]} = !x_{j[j*]}$ ($h_{j[j*]}$ либо входная переменная, либо ее отрицание), добавим в модель $n+1$ состояние $\{r_e, r_1, \ldots, r_n\}$, $n+2$ перехода: $p \to r_e$, $r_e \to r_1$, $r_1 \to r_2$, …, $r_{n-1} \to r_n$, $r_n \to q$. В отношение *Label* добавим пометки (r_e, e_i), (r_e, InEvent), $(r_k, z_{i[k]})$, (r_k, InAction) для всех *k* от 1 до *n*, а также пометки $(r_e, h_{j[1]})$, $(r_e, h_{j[2]})$, …, $(r_e, h_{j[m]})$.

Пример такого преобразования приведен на рис. 7.

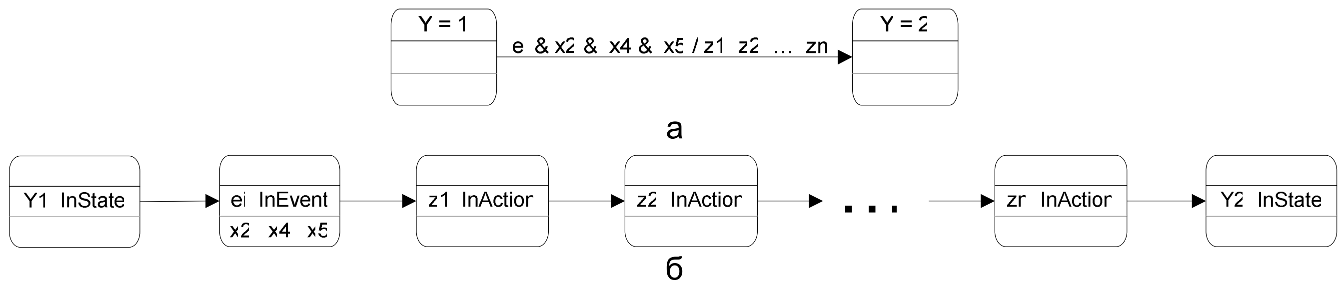

Рис. 7. Переход между состояниями
до преобразования по методу редукции (а) и после него (б)

Из этого рисунка видно, что для тех состояний, которые были построены из событий, в множество атомарных предложений были добавлены входные переменные в том виде, в котором они записаны на переходах автомата (вместе с отрицаниями, если они существуют).

Далее для каждой внешней позиции полученной модели (внешней будем называть любую позицию, за исключением тех, которые были скопированы при обработке вложенных автоматов) добавим в отношение *Label* пометку с атомарным предложением (элементом множества *Names*), соответствующим имени обрабатываемого автомата (того, для которого строится модель и которому это состояние принадлежит). Эти действия предназначены для того, чтобы в формулах темпоральной логики можно было различать, какой именно из автоматов системы выполняется на данном участке пути.

Для системы, эмулирующей работу банкомата (разд. 2.2 отчёта по третьему этапу), модель Крипке имеет большой размер. Поэтому она приведена на рис. 8 в упрощённом виде: позиции модели Крипке не выделяются каждая в свой блок, но видны особенности обработки вложенных автоматов.

1.5.2. Рекомендации по применению инструментального средства *CTLVerifier*

Разберём 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, её следует привести к определённому («каноническому») виду. Вначале в ней необходимо удалить все парные отрицания (путём замены подформул вида $\neg\neg f$ на f). После этого все входные воздействия, которые присутствуют в формуле без отрицания, необходимо предварить двумя отрицаниями: одно из них синтаксическое, другое логическое (это значит, что следует заменить литералы вида x_i на формулы $\neg !x_i$). Только после этих модификаций результирующую формулу можно верифицировать методами, предназначенными для языка *CTL*. Причина такого обращения с литералами заключается в следующем: требуется обеспечить, чтобы любая ссылка на несущественную переменную, которая упомянута в *CTL*-формуле, давала истинный результат (несущественными переменными на данном переходе называются те входные переменные исходного автомата, значение которых не проверяется на этом переходе).

Рассмотрим пример для модели банкомата. Проверим *CTL*-формулу: $e14 \rightarrow \neg\mathbf{E}[\neg o3.z0 \ \mathbf{U}\ y10]$. Она означает, что если произошло событие `e14`, то невозможно попасть в состояние `10`, минуя позицию `o3.z0`. Эта формула верна во всех позициях модели. Все пути из позиции `e14` в состояние `10` 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. 9. Позиция `e14`, являющаяся стартовой для данного пути, выделена на рисунке жирным шрифтом.

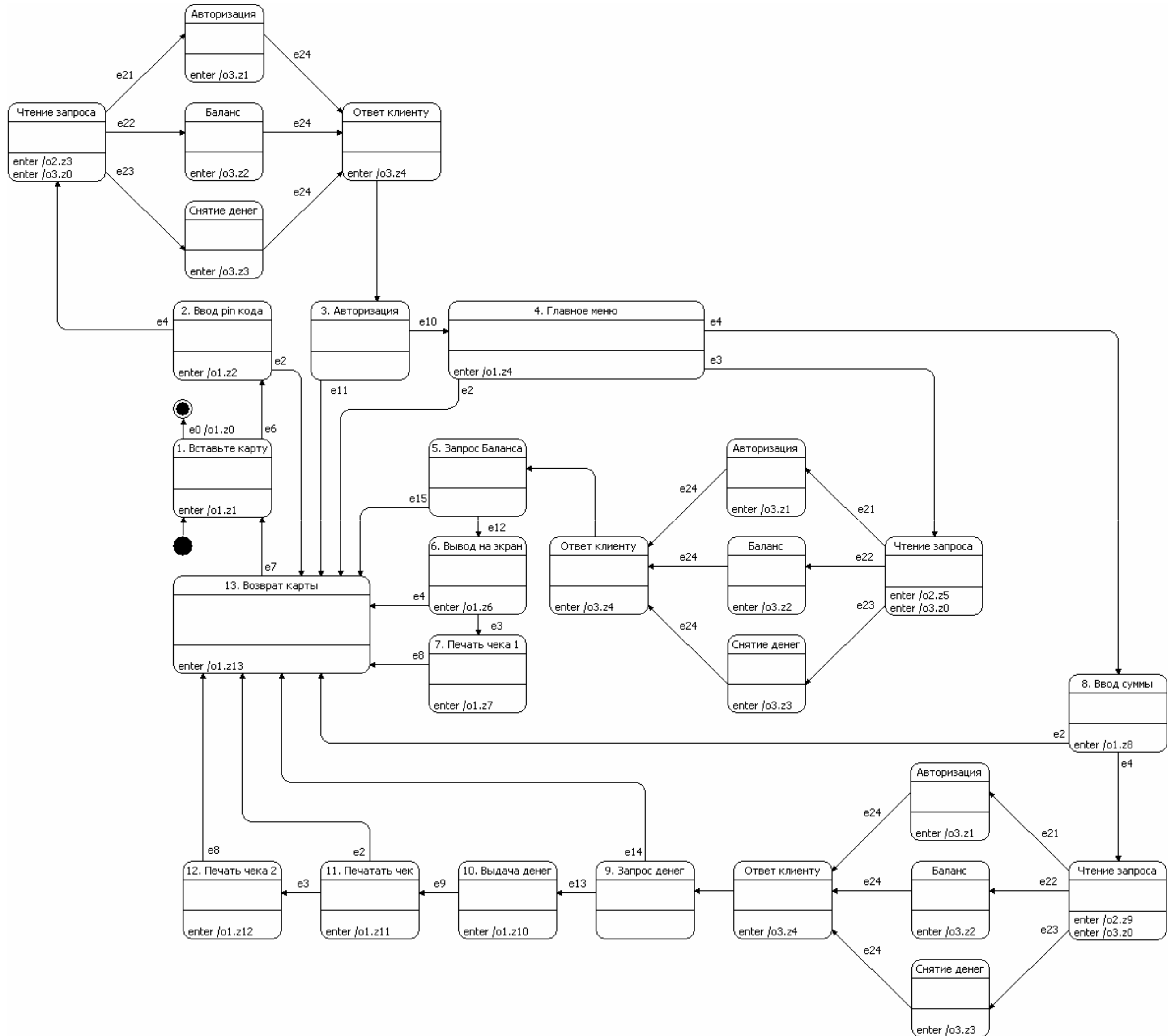

Рис. 8. Редукция графа переходов для модели банкомата

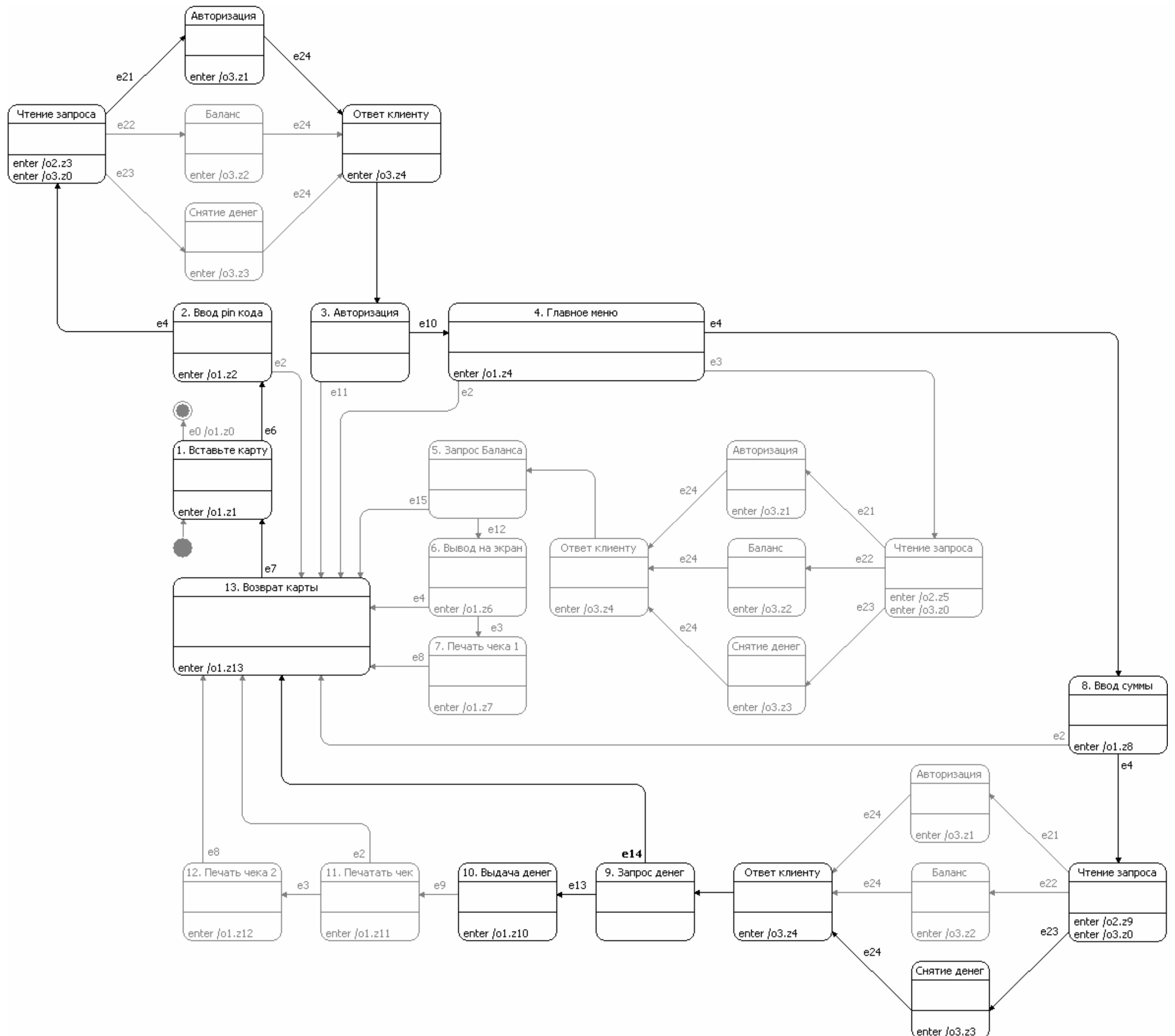

Рис. 9. Пример пути в модели Крипке для банкомата

Таким образом, в методе редукции графа переходов была видоизменена семантика *CTL*. Рассмотренная схема преобразовывала исходную формулу, построенную для новой семантики *CTL*, в новую формулу, для которой применима общепринятая семантика языка *CTL*. Использование такой схемы подходит для многих формул.

Путь в модели Крипке для банкомата, отображённый на рис. 9, можно показать и в терминах исходной системы автоматов (рис. 10, рис. 11).

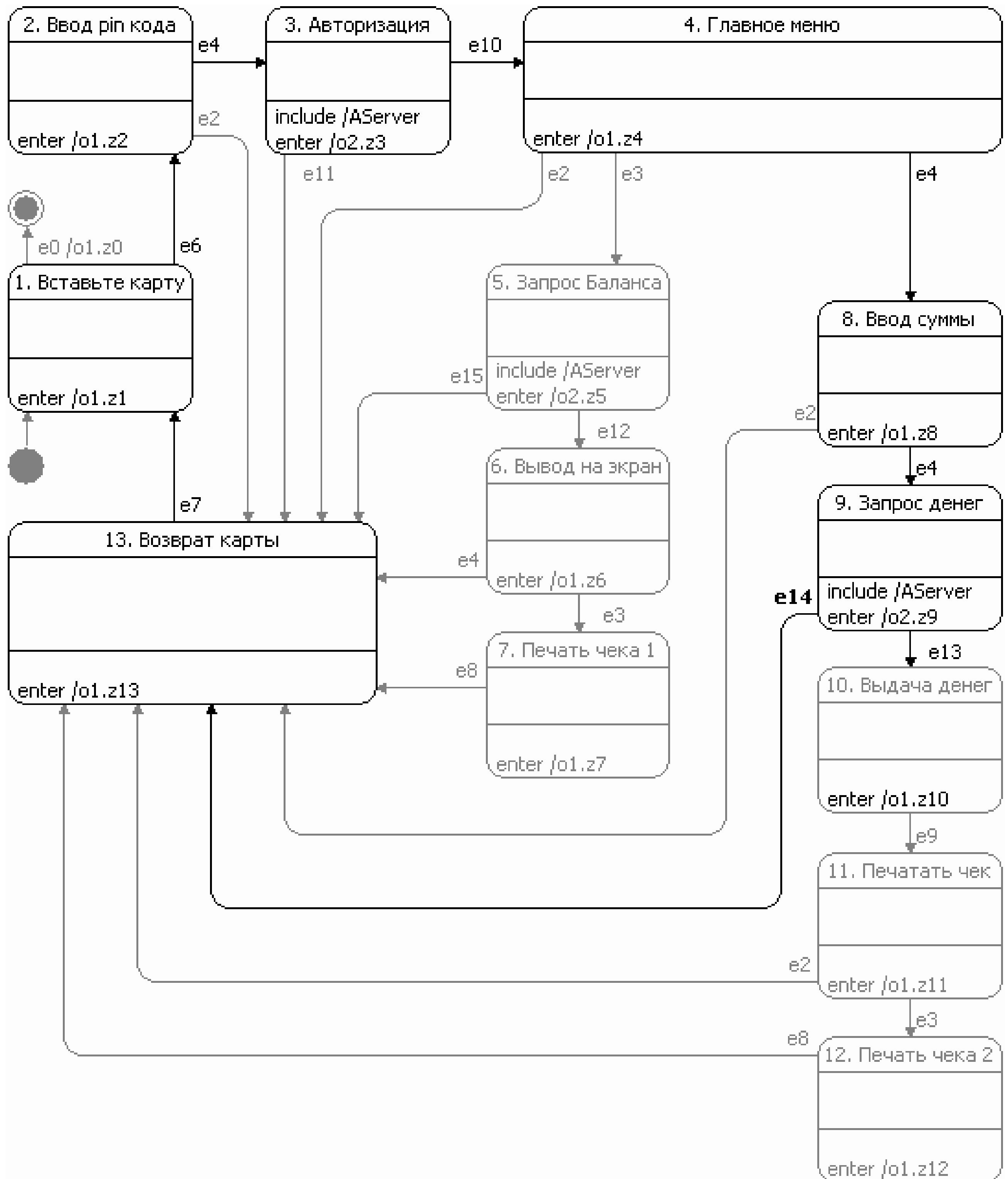

Рис. 10. Участок пути в автомате `AClient`

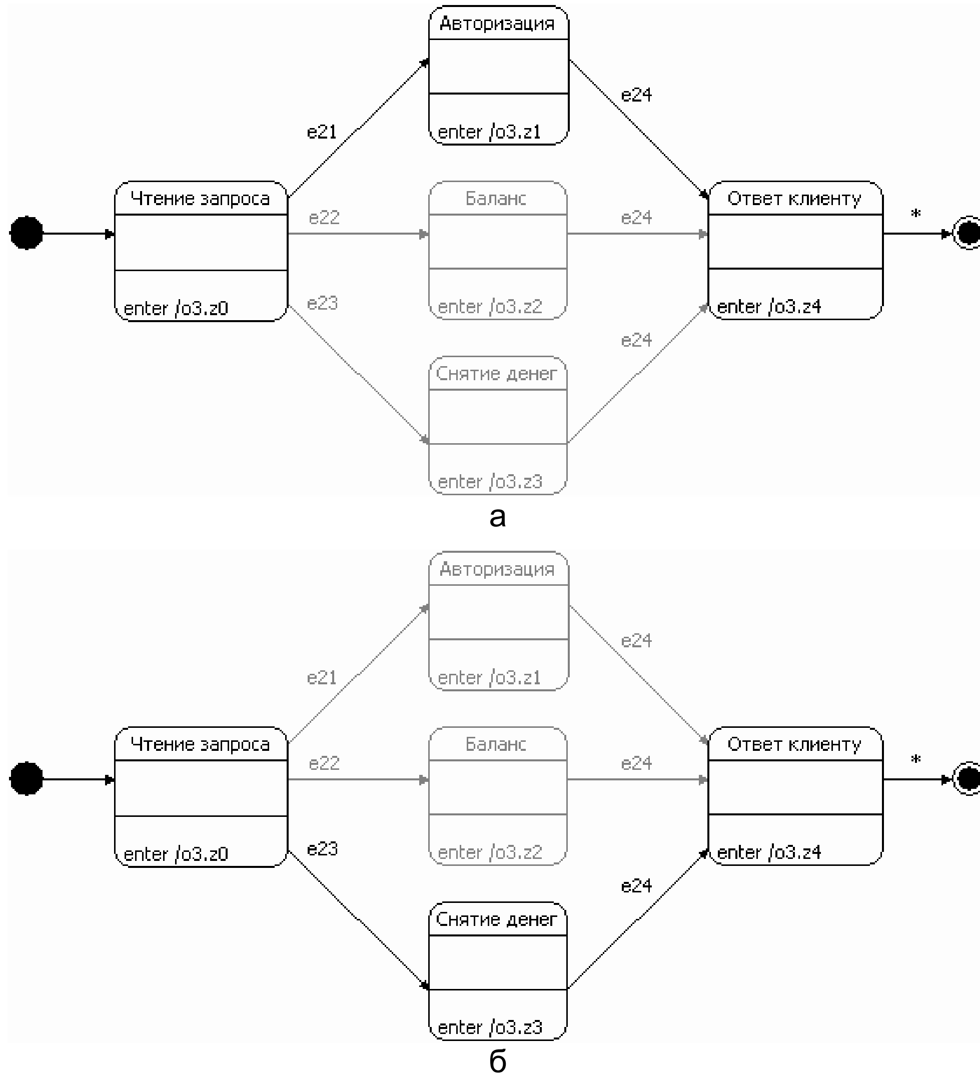

Рис. 11. Участок пути в автомате `AServer` при первом прохождении (а) и при втором (б)

Выше был рассмотрен метод редукции графа переходов для представления системы автоматов структурами Крипке. Этот метод был реализован программно. Также были приведены примеры темпоральных формул, которые можно верифицировать данным методом. Программа позволяет строить трассы, которые подтверждают заданные формулы, начинающиеся с квантора существования пути (или опровергают отрицания этих формул). Рассмотренные примеры показывают, что разработанное инструментально средство позволяет убеждаться в корректности модели, находить ошибки в случае некорректных формул и ошибки в неверной модели (ряд тестов проводился для намеренно изменённой, «испорченной» модели, для того чтобы продемонстрировать работу алгоритма в соответствующих ситуациях).

Более подробно инструментально средство *CTLVerifier* описано в разд. 2.4.

1.6. Выводы

Разработаны предложения и рекомендации по верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода, состоящее из двух частей:

- предложения и рекомендации по выбору метода верификации;

- предложения и рекомендации по применению разработанных методов верификации автоматных программ.

2. МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ И УКАЗАНИЯ ПО ВЕРИФИКАЦИИ УПРАВЛЯЮЩИХ ПРОГРАММ СИСТЕМ СО СЛОЖНЫМ ПОВЕДЕНИЕМ, ПОСТРОЕННЫХ НА ОСНОВЕ АВТОМАТНОГО ПОДХОДА

На основе методов, разработанных в рамках работ по контракту, были созданы прототипы инструментальных средств их реализующих. В данной главе для каждого прототипа даны методические рекомендации и указания по установке, настройке и запуску. Применение прототипов проиллюстрировано на примерах.

Инструментальное средство *UniMod.Verifier* является наиболее удобным инструментальным средство за счет интеграции с программным комплексом автоматного программирования *UniMod*. В связи с этим для инструментального средства *UniMod.Verifier* был создан пакет программной документации, дополненный по сравнению с документацией, представленной в отчете за третий этап работ.

2.1. Инструментальное средство *FSM Verifier*

Инструментальное средство *FSM Verifier* создано на основе метода верификации автоматных программ с использованием верификатора *NuSMV*. Оно полностью реализует указанный метод.

2.1.1. Краткое описание

В настоящем разделе приводятся указания по запуску инструментального средства *FSM Verifier*, а также описываются форматы входных и выходных данных для него.

2.1.1.1. Вызов и загрузка

Для преобразования программы автоматной модели в *SMV*-формат требуется вызвать программу `fsmverifier.jar`, которой в качестве параметра передается файл с автоматной моделью. Модель на языке *SMV* выводится в стандартный поток вывода (`stdout`). Таким образом, чтобы преобразовать системы автоматов в модель требуется выполнить следующую команду:

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

Здесь `inputfile.fsm` – файл с описанием автоматной модели.

Далее вызывается верификатор *NuSMV*. В качестве параметра ей передается модель на языке *SMV*, полученная на предыдущем шаге. В стандартный вывод выдается контрпример:

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

Для получения контрпримера запускается программа преобразования контрпримера. В качестве параметров ей передается контрпример, возвращенный верификатором *NuSMV,* и автоматная модель:

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

Здесь `inputfile.fsm` файл с описанием банкомата.

2.1.1.2. Входные данные

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

- Элемент `<fsm>` описывает класс автоматов с одинаковым набором состояний и переходов. Элемент `<fsm>` может содержать в себе элементы: `<state>`, `<field>`. Элемент `<fsm>` имеет следующие атрибуты:
  - `name` – имя классов автоматов;
  - `description` – описание классов автоматов.
- Элемент `<state>` описывает состояние автомата. Этот элемент может содержать в себе элементы `<transition>`. Он содержать следующие атрибуты:
  - `name` – имя состояния;
  - `description` – описание состояния.
- Элемент `<transition>` содержит описание одного перехода автомата. Он имеет следующие атрибуты:
  - `toState` – имя состояния, в которое переходит автомат по этому переходу;
  - `event` – событие, при котором срабатывает переход;
  - `condition` – условие, при котором срабатывает переход;
  - `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:
  - `name` – имя ссылки;
  - `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:
  - `name` – имя воздействия;
  - `description` – описание воздействия.
- Элемент `<specification>` содержит формулу для проверки. Он содержит атрибут:
  - `data` – текст *CTL*-формулы.

2.1.1.3. Выходные данные

При преобразовании контрпримера программа в стандартный вывод пишет контрпример в *HTML*-формате в виде таблицы. Табл. 3 демонстрирует общий вид контрпримера к системе автоматов.

Таблица 3. Пример контрпримера

Step	Active	Event	A0	...	An	Action	x1	...	xm
1	$Active_1$	e_1	s_{11}	...	s_{n1}	Action1	x_{11}	...	x_{m1}
2	$Active_2$	e_2	s_{12}	...	s_{n2}	Action2	x_{12}	...	x_{m2}
3	$Active_3$	e_3	s_{13}	...	s_{n3}	Action3	x_{13}	...	x_{m3}
...	...	...	...	...	...	...	...	...	...
p	$Active_p$	e_p	s_{1p}	...	s_{np}	Actionp	x_{1p}	...	x_{mp}

В таблице указаны следующие значения:

- номер шага;
- имя активного в данный момент автомата;
- событие, которое передается в данный момент;
- имена состояний всех автоматов;

- действие, выполняемое в данный момент;
- значение всех входных воздействий.

2.1.1.4. Технические требования

Для нормального функционирования программы *FSM Verifier* на персональной ЭВМ, необходимо, чтобы аппаратное обеспечение удовлетворяло следующим требованиям:

- процессор *Intel Pentium IV* и совместимые с ним;
- тактовая частота процессора не менее 1000МГц;
- оперативная память не менее 128 *MB*;
- дисковый накопитель объемом не менее 1 *GB*.

2.1.2. Установка и настройка

Дистрибутив инструментального средства содержит два файла:

- `verifier.jar` – преобразователь системы автоматов в модель на языке *SMV*;
- `counterexample.jar` – преобразователь контрпримера, выданного *NuSMV*, в контрпример для системы автоматов.

Для работы инструментального средства необходимы следующие программы:

- *Java Runtime Environment* (*JRE*) http://www.java.com/en/download/manual.jsp
- Верификатор *NuSMV* http://nusmv.fbk.eu/bin/bin_download2-v2.cgi
  http://nusmv.fbk.eu/distrib/NuSMV-2.4.3-i586-pc-mingw32msvc.exe

Приведем пример инсталляции программы. Сначала требуется установить *JRE,* последнюю версию которой можно скачать на сайте указанном выше. Пусть она установлена в директорию: `C:\Program Files\Java\jre\`. После установки требуется добавить путь к *Java*-машине в переменную окружения `PATH`. Для этого открываем диалог: «`Мой компьютер -> свойства -> Дополнительно -> Переменные среды`» (рис. 12). В переменную окружения `PATH` добавляется путь `C:\Program Files\Java\jre\`.

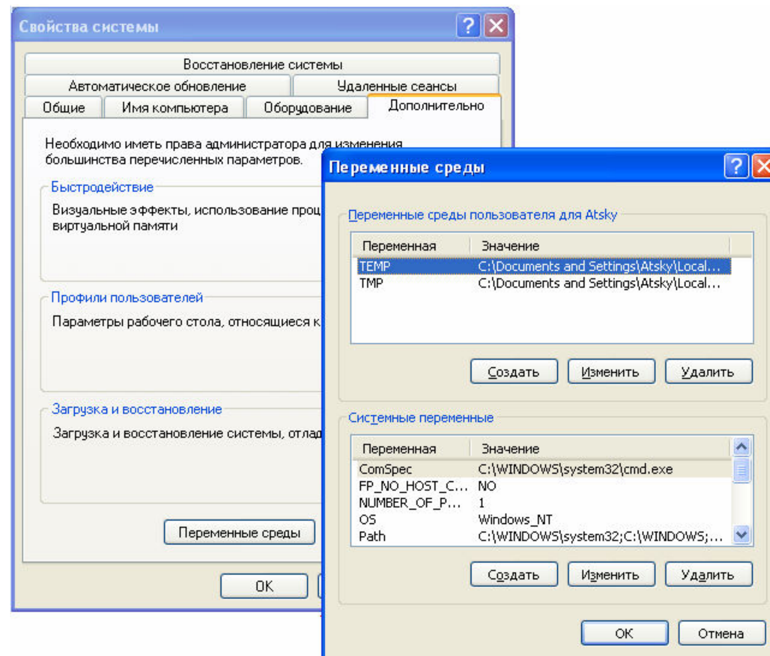

Рис. 12. Диалог настройки переменных среды

Верификатор *NuSMV*, скачанный с сайта, указанного выше, устанавливается в каталог `C:\NuSMV` (рис. 13).

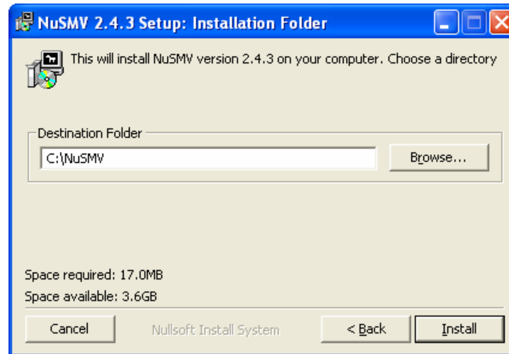

Рис. 13. Диалог выбора директории для установки верификатора *NuSMV*

Верификатор *FSM Verifier* распаковывается из архива `FSMVerifier.zip` командой

```
jar -xvf FSMVerifier.zip
```

2.1.3. Пример верификации

Рассмотрим пример применения инструментального средства *FSMVerifier* для верификации модели банкомата, описанного в работе [9].

2.1.3.1. Верифицируемая модель банкомата

Модель банкомата представляет собой систему автоматов, записанную в *XML*-файле. Так как для записи системы используется другая нотация, то необходимо внести следующие изменения в заданную систему автоматов:

- заменить названия состояний на `s0, s1, s2` …;
- заменить 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`;
- разделить переход из стояния `s12` в состояние `s4` на два;
- разделить переход из стояния `s3` в состояние `s11` на два;
- заметить все действия, производимые в состояниях, на действия, производимые на ребрах.

Рассмотрим автомат `A0`, заданный графом переходов, который изображен на рис. 14.

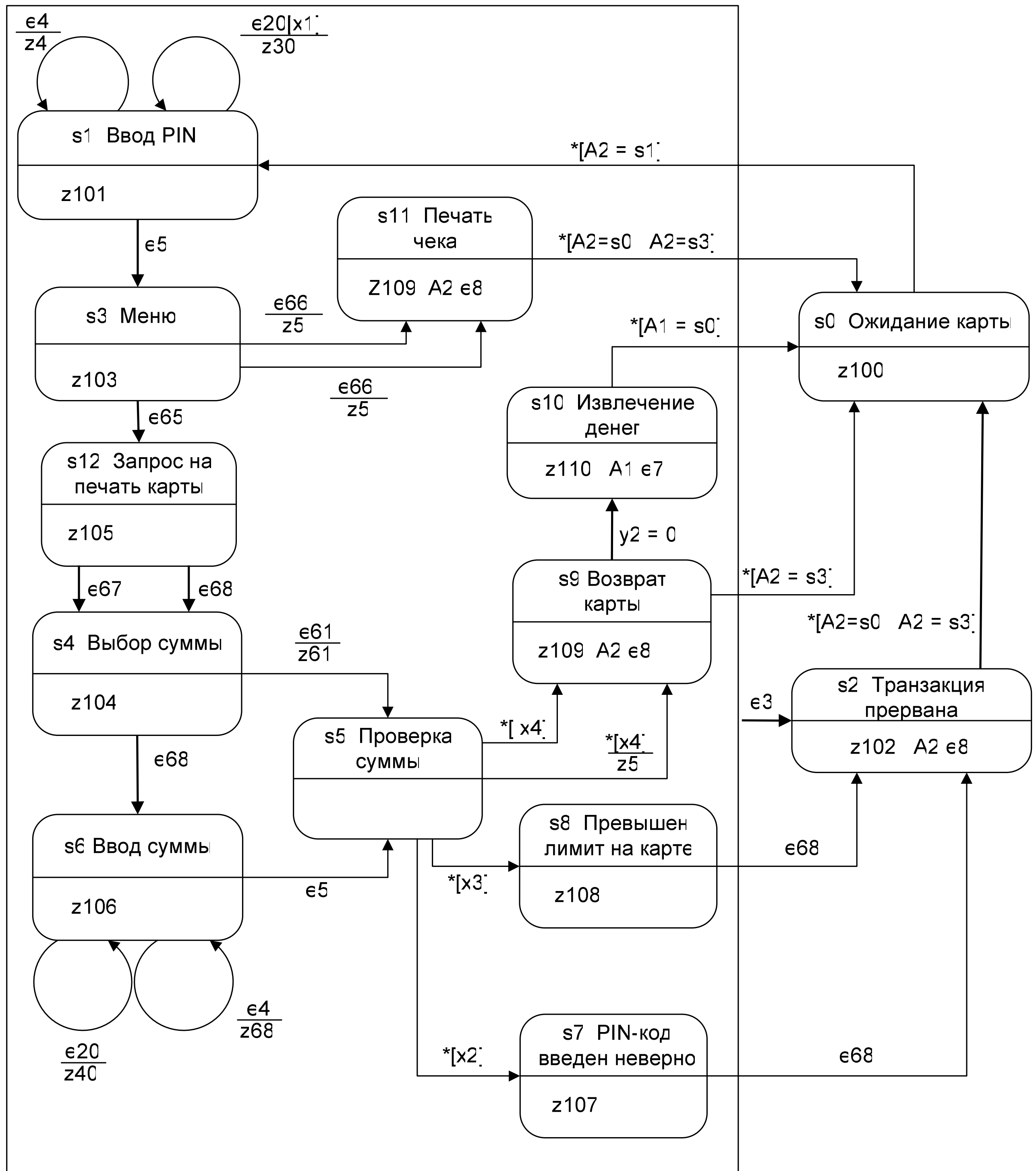

Рис. 14. Модель для автомата A0

Опишем модель в *XML*-формате. При этом автомат A0 (рис. 14) описывается следующим образом:

```
<fsm name="A0">
    <state name="s0" description="Ожидание карты" info=""
         onEnter="z100" onExit="">
      <transition toState="s1" info="" event="" condition="A2.s1"
         action=""/>
    </state>
    <state name="s2" description="Транзакция прервана" info=""
         onEnter="z102" onExit="A2.e8">
```

```
  <transition toState="s0" info="" event="" condition="A2.s0 |
      A2.s3" action=""/>
</state>
<state name="s1" description="Ввод PIN" info="" onEnter="z101"
      onExit="">
  <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="" onEnter="z103"
      onExit="">
  <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>
<state name="s4" description="Выбор суммы" info=""
      onEnter="z104" onExit="">
  <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="" onEnter=""
      onExit="">
  <transition toState="s8" info="" event="" condition="x3"
      action=""/>
  <transition toState="s7" info="" event="" condition="x2"
      action=""/>
  <transition toState="s9" info="" event="" condition="!x4"
      action=""/>
  <transition toState="s9" info="" event="" condition="x4"
      action="z5"/>
</state>
<state name="s6" description="Ввод суммы" info="" onEnter="z106"
      onExit="">
  <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=""
      onEnter="z107" onExit="">
  <transition toState="s2" info="" event="e68" condition=""
      action=""/>
</state>
<state name="s8" description="Превышен лимит на карте" info=""
      onEnter="z108" onExit="">
  <transition toState="s2" info="" event="e68" condition=""
      action=""/>
</state>
<state name="s9" description="Возврат карты" info=""
      onEnter="z109, A2.e8" onExit="">
  <transition toState="s10" info="" event="" condition="A2.s0"
      action=""/>
```

```
      <transition toState="s0" info="" event="" condition="A2.s3"
          action=""/>
    </state>
    <state name="s10" description="Извлечение денег" info=""
          onEnter="z110, A2.e7" onExit="">
      <transition toState="s0" info="" event="" condition="A1.s0"
          action=""/>
    </state>
    <state name="s11" description="Печать чека" info=""
          onEnter="z109, A2.e8" onExit="">
      <transition toState="s0" info="" event="" condition="A2.s0 |
          A2.s3" action=""/>
    </state>
    <state name="s12" description="Запрос на печать карты" info=""
          onEnter="z105" onExit="">
      <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/>
  </fsm>
```

Модели автоматов `A1` (рис. 15) и `A2` (рис. 16) можно оставить без изменений.

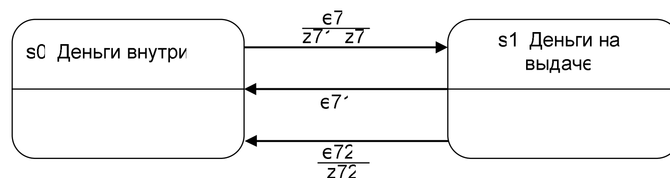

Рис. 15. Модель автомата `A1`

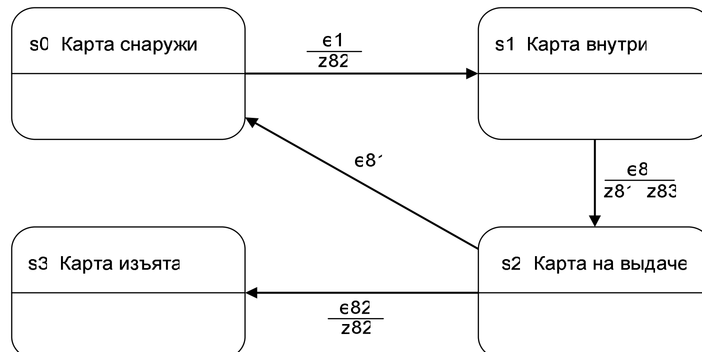

Рис. 16. Модель автомата `A2`

Модель автомата `A1` записывается в *XML*-формате следующим образом:

```
  <fsm name="A1">
    <state name="s0" description="Деньги внутри" info="" onEnter=""
          onExit="">
      <transition toState="s1" info="" event="e7" condition=""
          action="z71, z7"/>
    </state>
    <state name="s1" description="Деньги на выдаче" info=""
          onEnter="" onExit="">
      <transition toState="s0" info="" event="e71" condition=""
          action=""/>
      <transition toState="s0" info="" event="e72" condition=""
          action="z72"/>
```

```
  </state>
  <specs/>
</fsm>
```

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

```
<fsm name="A2">
  <state name="s0" description="Карта снаружи" info="" onEnter=""
       onExit="">
    <transition toState="s1" info="" event="e1" condition=""
       action="z82"/>
  </state>
  <state name="s1" description="Карта внутри" info="" onEnter=""
       onExit="">
    <transition toState="s2" info="" event="e8" condition=""
       action="z81, z83"/>
  </state>
  <state name="s2" description="Карта изъята" info="" onEnter=""
       onExit="">
    <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=""
       onEnter="" onExit=""/>
  <specs/>
</fsm>
```

Далее полученный *XML*-файл преобразуется в модель на языке *SMV*. Пример такой модели приведен в разд. 2.1.4.4.

2.1.3.2. Проверка истинного свойства

Банкомат выдает деньги только по требованию. Проверяется отсутствие последовательности действий, при которой пользователь получает деньги до выбора в меню соответствующего пункта. В терминах модели свойство формулируется следующим образом: «Пока автомат `A0` находится в состоянии «`3. Меню`» не будет вызвано событие `e65`, автомат `A1` не перейдет в состояние `s1`».

Сначала необходимо выразить атомарные свойства:

- свойство «вызвано событие `e65`» выражается формулой: $Event = e65$;
- свойство «автомат `A1` находится в стоянии s1» выражается формулой $A1 = s1$.

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

$$\mathbf{A}[Event = e65 \mathbf{R}\ (A1 != s1)].$$

Формула для программного средства имеет вид: `!E [ !A0.e65 U A1.s1 ]`.

Верификатор возвращает следующий результат:

```
-- specification !E [ !A0.e65 U A1.s1 ]   is true
```

2.1.3.3. Проверка ложного свойства

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

В терминах модели это 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 имеет вид: **A** [!*z5* **U** *x2* = 0].
Вывод верификатора *NuSMV*, после проверки модели:

```
-- specification A [ !z5 U x2 = 0 ]   is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  x1 = 0
  x2 = 1
  x3 = 0
  x4 = 0
  A0.State = 0
  A1.State = 0
  A2.State = 0
  Active = 0
  Event = 0
  z83 = 0
  z82 = 0
  z81 = 0
  z72 = 0
  z71 = 0
  z7 = 0
  z68 = 0
  z61 = 0
  z5 = 0
  z40 = 0
  z4 = 0
  z30 = 0
  z110 = 0
  z109 = 0
  z108 = 0
  z107 = 0
  z106 = 0
  z105 = 0
  z104 = 0
  z103 = 0
  z102 = 0
  z101 = 0
  z100 = 0
  A0.s9 = 0
  A0.s8 = 0
  A0.s7 = 0
  A0.s6 = 0
  A0.s5 = 0
  A0.s4 = 0
  A0.s3 = 0
  A0.s2 = 0
  A0.s12 = 0
  A0.s11 = 0
  A0.s10 = 0
  A0.s1 = 0
  A0.s0 = 1
  A0.inState = 1
  A0.e68 = 0
  A0.e67 = 0
  A0.e66 = 0
  A0.e65 = 0
  A0.e61 = 0
  A0.e5 = 0
```

```
  A0.e4 = 0
  A0.e20 = 0
  A1.s1 = 0
  A1.s0 = 1
  A1.inState = 1
  A1.e72 = 0
  A1.e71 = 0
  A1.e7 = 0
  A2.s3 = 0
  A2.s2 = 0
  A2.s1 = 0
  A2.s0 = 1
  A2.inState = 1
  A2.e82 = 0
  A2.e81 = 0
  A2.e8 = 0
  A2.e1 = 0
-> Input: 1.2 <-
-> State: 1.2 <-
  Active = 3
-> Input: 1.3 <-
-> State: 1.3 <-
  A2.State = 4
  z82 = 1
  A2.s0 = 0
  A2.inState = 0
  A2.e1 = 1
-> Input: 1.4 <-
-> State: 1.4 <-
  A2.State = 1
  Active = 0
  z82 = 0
  A2.s1 = 1
  A2.inState = 1
  A2.e1 = 0
-> Input: 1.5 <-
-> State: 1.5 <-
  Active = 1
-> Input: 1.6 <-
-> State: 1.6 <-
  A0.State = 13
  z101 = 1
  A0.s0 = 0
  A0.inState = 0
-> Input: 1.7 <-
-> State: 1.7 <-
  A0.State = 2
  Active = 0
  z101 = 0
  A0.s1 = 1
  A0.inState = 1
-> Input: 1.8 <-
-> State: 1.8 <-
  Active = 1
-> Input: 1.9 <-
-> State: 1.9 <-
  A0.State = 16
  z103 = 1
  A0.s1 = 0
  A0.inState = 0
  A0.e5 = 1
-> Input: 1.10 <-
```

```
-> State: 1.10 <-
  A0.State = 3
  Active = 0
  z103 = 0
  A0.s3 = 1
  A0.inState = 1
  A0.e5 = 0
-> Input: 1.11 <-
-> State: 1.11 <-
  Active = 1
-> Input: 1.12 <-
-> State: 1.12 <-
  A0.State = 21
  z105 = 1
  A0.s3 = 0
  A0.inState = 0
  A0.e65 = 1
-> Input: 1.13 <-
-> State: 1.13 <-
  A0.State = 12
  Active = 0
  z105 = 0
  A0.s12 = 1
  A0.inState = 1
  A0.e65 = 0
-> Input: 1.14 <-
-> State: 1.14 <-
  Active = 1
-> Input: 1.15 <-
-> State: 1.15 <-
  A0.State = 51
  z104 = 1
  A0.s12 = 0
  A0.inState = 0
  A0.e67 = 1
-> Input: 1.16 <-
-> State: 1.16 <-
  A0.State = 4
  Active = 0
  z104 = 0
  A0.s4 = 1
  A0.inState = 1
  A0.e67 = 0
-> Input: 1.17 <-
-> State: 1.17 <-
  Active = 1
-> Input: 1.18 <-
-> State: 1.18 <-
  A0.State = 31
  z61 = 1
  A0.s4 = 0
  A0.inState = 0
  A0.e61 = 1
-> Input: 1.19 <-
-> State: 1.19 <-
  A0.State = 5
  Active = 0
  z61 = 0
  A0.s5 = 1
  A0.inState = 1
  A0.e61 = 0
-> Input: 1.20 <-
```

```
-> State: 1.20 <-
  x4 = 1
  Active = 1
-> Input: 1.21 <-
-> State: 1.21 <-
  x2 = 0
  x4 = 0
  A0.State = 36
  z5 = 1
  A0.s5 = 0
  A0.inState = 0
```

Таблица 4. Контрпример к системе автоматов

Step	Active	Event	A0	A1	A2	Action	x1	x2	x3	x4
1	A2	-	s0	s0	s0	-	0	1	0	0
2	A2	-	s0	s0	s0	z82	0	1	0	0
3	A0	-	s0	s0	s1	-	0	1	0	0
4	A0	-	s0	s0	s1	z101	0	1	0	0
5	A0	-	s1	s0	s1	-	0	1	0	0
6	A0	-	s1	s0	s1	z103	0	1	0	0
7	A0	-	s3	s0	s1	-	0	1	0	0
8	A0	-	s3	s0	s1	z105	0	1	0	0
9	A0	-	s12	s0	s1	-	0	1	0	0
10	A0	-	s12	s0	s1	z104	0	1	0	0
11	A0	-	s4	s0	s1	-	0	1	0	0
12	A0	-	s4	s0	s1	z61	0	1	0	0
13	A0	-	s5	s0	s1	-	0	1	0	1
14	A0	-	s5	s0	s1	z5	0	0	0	0

2.1.3.4. Модель банкомата на языке *SMV*

```
MODULE A0(Event, Active, x1, x2, x3, x4, A1, A2)
VAR
   State: 0..52;
ASSIGN
   init(State) := 0;
TRANS
   (Active & State = 0 & next(State) = 13 & (A2.s1)) |
   (Active & State = 0 & next(State) = 0 & 1 & !(A2.s1)) | -- Return
          to this state.
   (Active & State = 1 & next(State) = 14 & (A2.s0 | A2.s3)) |
   (Active & State = 1 & next(State) = 1 & 1 & !(A2.s0 | A2.s3)) | --
          Return to this state.
   (Active & State = 2 & next(State) = 16) |
   (Active & State = 2 & next(State) = 17 & (x1)) |
   (Active & State = 2 & next(State) = 19) |
   (Active & State = 2 & next(State) = 2 & 1 & !(x1)) | -- Return to
          this state.
   (Active & State = 3 & next(State) = 21) |
   (Active & State = 3 & next(State) = 22) |
```

```
(Active & State = 3 & next(State) = 26) |
(Active & State = 4 & next(State) = 30) |
(Active & State = 4 & next(State) = 31) |
(Active & State = 5 & next(State) = 32 & (x3)) |
(Active & State = 5 & next(State) = 33 & (x2)) |
(Active & State = 5 & next(State) = 34 & (!x4)) |
(Active & State = 5 & next(State) = 36 & (x4)) |
(Active & State = 5 & next(State) = 5 & 1 & !(x3) & !(x2) & !(!x4)
       & !(x4)) | -- Return to this state.
(Active & State = 6 & next(State) = 39) |
(Active & State = 6 & next(State) = 40) |
(Active & State = 6 & next(State) = 42) |
(Active & State = 7 & next(State) = 44) |
(Active & State = 8 & next(State) = 45) |
(Active & State = 9 & next(State) = 46 & (A2.s0)) |
(Active & State = 9 & next(State) = 48 & (A2.s3)) |
(Active & State = 9 & next(State) = 9 & 1 & !(A2.s0) & !(A2.s3)) |
       -- Return to this state.
(Active & State = 10 & next(State) = 49 & (A1.s0)) |
(Active & State = 10 & next(State) = 10 & 1 & !(A1.s0)) | --
       Return to this state.
(Active & State = 11 & next(State) = 50 & (A2.s0 | A2.s3)) |
(Active & State = 11 & next(State) = 11 & 1 & !(A2.s0 | A2.s3)) |
       -- Return to this state.
(Active & State = 12 & next(State) = 51) |
(Active & State = 12 & next(State) = 52) |
(Active & State = 13 & next(State) = 2) |
(Active & State = 14 & next(State) = 15) |
(Active & State = 15 & next(State) = 0) |
(Active & State = 16 & next(State) = 3) |
(Active & State = 17 & next(State) = 18) |
(Active & State = 18 & next(State) = 2) |
(Active & State = 19 & next(State) = 20) |
(Active & State = 20 & next(State) = 2) |
(Active & State = 21 & next(State) = 12) |
(Active & State = 22 & next(State) = 23) |
(Active & State = 23 & next(State) = 24) |
(Active & State = 24 & next(State) = 25) |
(Active & State = 25 & next(State) = 11) |
(Active & State = 26 & next(State) = 27) |
(Active & State = 27 & next(State) = 28) |
(Active & State = 28 & next(State) = 29) |
(Active & State = 29 & next(State) = 11) |
(Active & State = 30 & next(State) = 6) |
(Active & State = 31 & next(State) = 5) |
(Active & State = 32 & next(State) = 8) |
(Active & State = 33 & next(State) = 7) |
(Active & State = 34 & next(State) = 35) |
(Active & State = 35 & next(State) = 9) |
(Active & State = 36 & next(State) = 37) |
(Active & State = 37 & next(State) = 38) |
(Active & State = 38 & next(State) = 9) |
(Active & State = 39 & next(State) = 5) |
(Active & State = 40 & next(State) = 41) |
(Active & State = 41 & next(State) = 6) |
(Active & State = 42 & next(State) = 43) |
(Active & State = 43 & next(State) = 6) |
(Active & State = 44 & next(State) = 1) |
(Active & State = 45 & next(State) = 1) |
(Active & State = 46 & next(State) = 47) |
(Active & State = 47 & next(State) = 10) |
(Active & State = 48 & next(State) = 0) |
```

```
   (Active & State = 49 & next(State) = 0) |
   (Active & State = 50 & next(State) = 0) |
   (Active & State = 51 & next(State) = 4) |
   (Active & State = 52 & next(State) = 4) |
   (!Active & next(State) = State)
DEFINE
  e20 := State = 17 | State = 42;
  e4 := State = 19 | State = 40;
  e5 := State = 16 | State = 39;
  e61 := State = 31;
  e65 := State = 21;
  e66 := State = 22;
  e67 := State = 26 | State = 51;
  e68 := State = 30 | State = 44 | State = 45 | State = 52;
  inState := State = 0 | State = 1 | State = 2 | State = 3 | State =
         4 | State = 5 | State = 6 | State = 7 | State = 8 | State =
         9 | State = 10 | State = 11 | State = 12;
  s0 := State = 0;
  s1 := State = 2;
  s10 := State = 10;
  s11 := State = 11;
  s12 := State = 12;
  s2 := State = 1;
  s3 := State = 3;
  s4 := State = 4;
  s5 := State = 5;
  s6 := State = 6;
  s7 := State = 7;
  s8 := State = 8;
  s9 := State = 9;
MODULE A1(Event, Active, x1, x2, x3, x4)
VAR
  State: 0..5;
ASSIGN
  init(State) := 0;
TRANS
   (Active & State = 0 & next(State) = 2 & (Event = e7)) |
   (Active & State = 0 & next(State) = 0 & 1 & !(Event = e7)) | --
         Return to this state.
   (Active & State = 1 & next(State) = 4) |
   (Active & State = 1 & next(State) = 5) |
   (Active & State = 2 & next(State) = 3) |
   (Active & State = 3 & next(State) = 1) |
   (Active & State = 4 & next(State) = 0) |
   (Active & State = 5 & next(State) = 0) |
   (!Active & next(State) = State)
DEFINE
  e7 := State = 2;
  e71 := State = 4;
  e72 := State = 5;
  inState := State = 0 | State = 1;
  s0 := State = 0;
  s1 := State = 1;
MODULE A2(Event, Active, x1, x2, x3, x4)
VAR
  State: 0..8;
ASSIGN
  init(State) := 0;
TRANS
   (Active & State = 0 & next(State) = 4) |
   (Active & State = 1 & next(State) = 5 & (Event = e8)) |
```

```
    (Active & State = 1 & next(State) = 1 & 1 & !(Event = e8)) | --
           Return to this state.
    (Active & State = 2 & next(State) = 7) |
    (Active & State = 2 & next(State) = 8) |
    (Active & State = 4 & next(State) = 1) |
    (Active & State = 5 & next(State) = 6) |
    (Active & State = 6 & next(State) = 2) |
    (Active & State = 7 & next(State) = 0) |
    (Active & State = 8 & next(State) = 3) |
    (!Active & next(State) = State)
DEFINE
  e1 := State = 4;
  e8 := State = 5;
  e81 := State = 7;
  e82 := State = 8;
  inState := State = 0 | State = 1 | State = 2 | State = 3;
  s0 := State = 0;
  s1 := State = 1;
  s2 := State = 2;
  s3 := State = 3;
MODULE main()
VAR
  x1: {0, 1};
  x2: {0, 1};
  x3: {0, 1};
  x4: {0, 1};
  A0: A0(Event, Active = 1, x1, x2, x3, x4, A1, A2);
  A1: A1(Event, Active = 2, x1, x2, x3, x4);
  A2: A2(Event, Active = 3, x1, x2, x3, x4);
  Active: 0..3;
  Event: {0, e8, e7};
ASSIGN
  init(Active) := 0;
  next(Active) := case
    Active = 0: 1..3;
    Active = 1 & next(A0.State) = 14: 3;
    Active != 1 & A0.State = 14 & A2.inState: 1;
    Active = 1 & next(A0.State) = 23: 3;
    Active != 1 & A0.State = 23 & A2.inState: 1;
    Active = 1 & next(A0.State) = 25: 3;
    Active != 1 & A0.State = 25 & A2.inState: 1;
    Active = 1 & next(A0.State) = 27: 3;
    Active != 1 & A0.State = 27 & A2.inState: 1;
    Active = 1 & next(A0.State) = 29: 3;
    Active != 1 & A0.State = 29 & A2.inState: 1;
    Active = 1 & next(A0.State) = 35: 3;
    Active != 1 & A0.State = 35 & A2.inState: 1;
    Active = 1 & next(A0.State) = 38: 3;
    Active != 1 & A0.State = 38 & A2.inState: 1;
    Active = 1 & next(A0.State) = 47: 3;
    Active != 1 & A0.State = 47 & A2.inState: 1;
    Active = 1 & next(A0.inState): 0;
    Active = 2 & next(A1.inState): 0;
    Active = 3 & next(A2.inState): 0;
    1: Active;
  esac;
  init(Event) := 0;
  next(Event) := case
    Active = 1 & next(A0.State) = 14: e8;
    Active = 1 & next(A0.State) = 23: e8;
    Active = 1 & next(A0.State) = 25: e8;
    Active = 1 & next(A0.State) = 27: e8;
```

```
        Active = 1 & next(A0.State) = 29: e8;
        Active = 1 & next(A0.State) = 35: e8;
        Active = 1 & next(A0.State) = 38: e8;
        Active = 1 & next(A0.State) = 47: e7;
        1: Event;
      esac;
   DEFINE
      z100 := (Active = 1 & A0.State= 15) | (Active = 1 & A0.State= 48)
             | (Active = 1 & A0.State= 49) | (Active = 1 & A0.State=
             50);
      z101 := (Active = 1 & A0.State= 13) | (Active = 1 & A0.State= 18)
             | (Active = 1 & A0.State= 20);
      z102 := (Active = 1 & A0.State= 44) | (Active = 1 & A0.State= 45);
      z103 := (Active = 1 & A0.State= 16);
      z104 := (Active = 1 & A0.State= 51) | (Active = 1 & A0.State= 52);
      z105 := (Active = 1 & A0.State= 21);
      z106 := (Active = 1 & A0.State= 30) | (Active = 1 & A0.State= 41)
             | (Active = 1 & A0.State= 43);
      z107 := (Active = 1 & A0.State= 33);
      z108 := (Active = 1 & A0.State= 32);
      z109 := (Active = 1 & A0.State= 22) | (Active = 1 & A0.State= 24)
             | (Active = 1 & A0.State= 26) | (Active = 1 & A0.State= 28)
             | (Active = 1 & A0.State= 34) | (Active = 1 & A0.State=
             37);
      z110 := (Active = 1 & A0.State= 46);
      z30 := (Active = 1 & A0.State= 17);
      z4 := (Active = 1 & A0.State= 19);
      z40 := (Active = 1 & A0.State= 42);
      z5 := (Active = 1 & A0.State= 36);
      z61 := (Active = 1 & A0.State= 31);
      z68 := (Active = 1 & A0.State= 40);
      z7 := (Active = 2 & A1.State= 3);
      z71 := (Active = 2 & A1.State= 2);
      z72 := (Active = 2 & A1.State= 5);
      z81 := (Active = 3 & A2.State= 5);
      z82 := (Active = 3 & A2.State= 4) | (Active = 3 & A2.State= 8);
      z83 := (Active = 3 & A2.State= 6);
```

2.2. Инструментальное средство *Converter*

Инструментальное средство Converter построено на основе метода верификации визуальных автоматных моделей.

2.2.1. Установка и настройка

Рабочая папка инструментального средства *Converter* имеет структуру, представленную в табл.5.

Таблица 5. Рабочая папка инструментального средства *Converter*

`example`	Папка, содержащая пример верификации банкомата.
`lib`	Папка, содержащая библиотеки, необходимые для работы инструментального средства.
`run.cmd`	Скрипт, запускающий инструментальное средство. Скрипт предназначен для запуска в среде ОС Windows.
`spin.exe`	Верификатор *SPIN*.
`sources`	Папка, содержащая исходные коды инструментального средства.

Инструментальное средство *Converter* состоит из двух *Java*-классов: *MainClass* и *Worker*. Для того, чтобы собрать средство, требуются:

1. Интегрированная среда разработки *Eclipse* версии 3.1.
2. Библиотека инструментального средства *UniMod*.
3. Верификатор *SPIN*.

Для того чтобы собрать инструментальное средство *Converter*, требуется выполнить следующие действия:

1. В среде разработки *Eclipse* создать новый *Java*-проект (*File* → New → *Project*).
2. В созданный проект включить классы *MainClass* и *Worker*.
3. Добавить к проекту (Project|Properties) следующие библиотеки инструмента *UniMod* (рис. 17):
   - `antlr.jar`
   - `common-01.01.022.jar`
   - `commons-beanutils.jar`
   - `commons-collections-3.1.jar`
   - `commons-digester-1.7.jar`
   - `commons-lang-2.0.jar`
   - `commons-logging.jar`
   - `log4j-1.2.8.jar`
   - `unimod-adapter-standalone.jar`
   - `unimod-core.jar`
   - `velocity-1.4.jar`

Библиотеки средства *UniMod* (если оно установлено) находятся по адресам: `C:\Eclipse\plugins\com.evelopers.unimod.core_1.3.1.35` и `C:\Eclipse\plugins\com.evelopers.unimod.adapter.standalone_1.3.1.35` (при условии, что *Eclipse* установлен в директорию `C:\Eclipse`, и *UniMod* имеет версию 1.3.1.35).

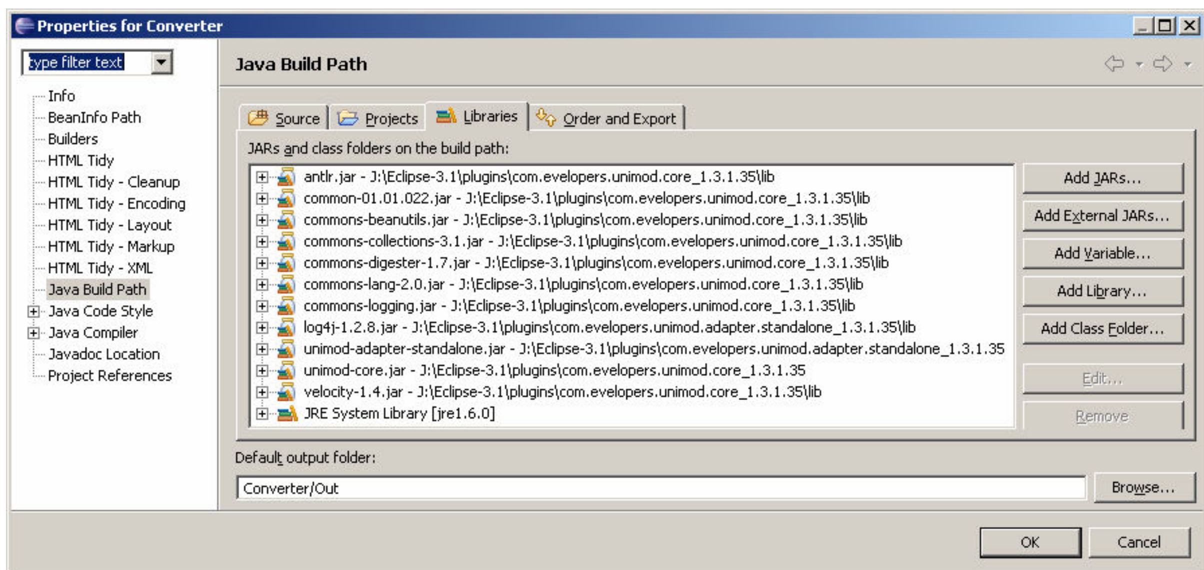

Рис. 17. Добавление библиотек инструмента *UniMod*

Запуск инструментального средства осуществляется скриптом `run.cmd`. Это скрипт принимает на вход три параметра:

1. Путь к *XML*-описанию визуальной автоматной программы.
2. Полное имя файла, в который будет записан отчет.
3. Проверяемое свойство в виде *LTL*-формулы.

2.2.2. Пример применения

Общий порядок верификации.

1. Экспортировать визуальную автоматную программу, разработанную при помощи инструментального средства *Unimod* в *XML*-файл, как показано на рис. 18.
2. Каждое проверяемое свойство (требование) сформулировать на языке *LTL*. Каждое свойство в инструментальном средстве *Converter* проверяется отдельно.
3. Для каждого свойства *Converter* генерирует отчет, в котором будет содержаться контрпример либо запись о том, что визуальная автоматная программа удовлетворяет проверяемому свойству.

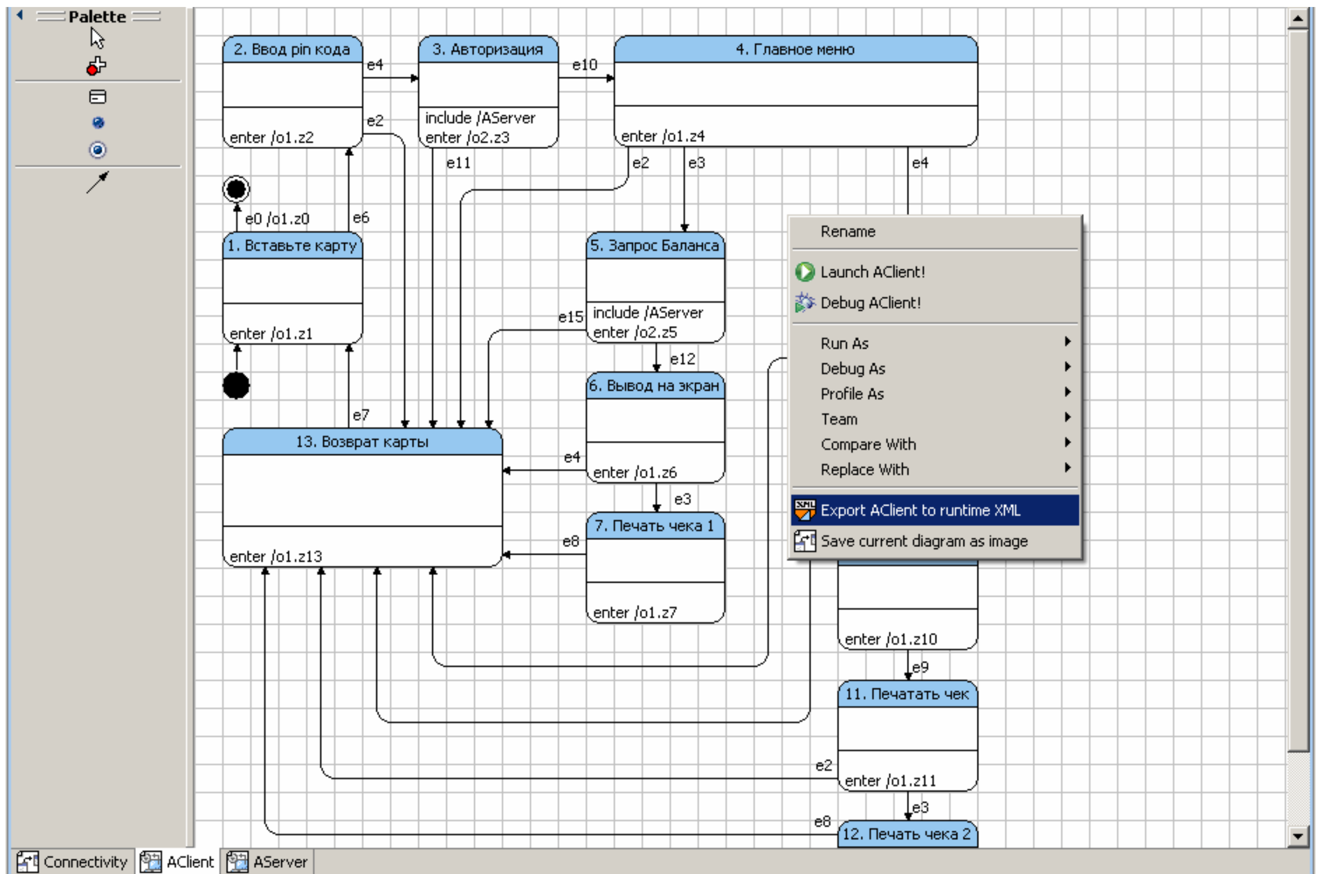

Рис. 18. Экспорт автоматной программы в *XML*

Проведем верификацию банкомата с использованием инструментального средства *Converter*. Запустим его с пустым свойством. При этом будет построена следующая модель:

```
int lastEvent;
#define STATE_0   0 /*Top*/
#define STATE_1   1 /*s1*/
#define STATE_2   2 /*4. Главное меню*/
#define STATE_3   3 /*5. Запрос Баланса*/
#define STATE_4   4 /*3. Авторизация*/
#define STATE_5   5 /*11. Печатать чек*/
#define STATE_6   6 /*s2*/
#define STATE_7   7 /*13. Возврат карты*/
#define STATE_8   8 /*12. Печать чека 2*/
#define STATE_9   9 /*2. Ввод pin кода*/
#define STATE_10  10 /*8. Ввод суммы*/
#define STATE_11  11 /*10. Выдача денег*/
#define STATE_12  12 /*9. Запрос денег*/
#define STATE_13  13 /*1. Вставьте карту*/
#define STATE_14  14 /*7. Печать чека 1*/
```

```
#define STATE_15  15 /*6. Вывод на экран*/

int stateAClient;
inline AClient() {
  stateAClient = STATE_1;
  do
  ::(stateAClient == STATE_1) ->
    printf("State AClient 1 : s1\n");
    if
    ::stateAClient = STATE_13;
      printf("Going to state AClient 13 : 1. Вставьте карту\n");
    fi;
  ::(stateAClient == STATE_2) ->
    printf("State AClient 2 : 4. Главное меню\n");
    if
    ::stateAClient = STATE_7;
      printf("Going to state AClient 7 : 13. Возврат карты\n");
      lastEvent = 2;
      printf("Event = e2\n");
    ::stateAClient = STATE_3;
      printf("Going to state AClient 3 : 5. Запрос Баланса\n");
      lastEvent = 3;
      printf("Event = e3\n");
    ::stateAClient = STATE_10;
      printf("Going to state AClient 10 : 8. Ввод суммы\n");
      lastEvent = 4;
      printf("Event = e4\n");
    fi;
  ::(stateAClient == STATE_3) ->
    printf("State AClient 3 : 5. Запрос Баланса\n");
    printf("Calling automaton AServer\n");
    AServer();
    if
    ::stateAClient = STATE_15;
      printf("Going to state AClient 15 : 6. Вывод на экран\n");
      lastEvent = 12;
      printf("Event = e12\n");
    ::stateAClient = STATE_7;
      printf("Going to state AClient 7 : 13. Возврат карты\n");
      lastEvent = 15;
      printf("Event = e15\n");
    fi;
  ::(stateAClient == STATE_4) ->
    printf("State AClient 4 : 3. Авторизация\n");
    printf("Calling automaton AServer\n");
    AServer();
    if
    ::stateAClient = STATE_2;
      printf("Going to state AClient 2 : 4. Главное меню\n");
      lastEvent = 10;
      printf("Event = e10\n");
    ::stateAClient = STATE_7;
      printf("Going to state AClient 7 : 13. Возврат карты\n");
      lastEvent = 11;
      printf("Event = e11\n");
    fi;
  ::(stateAClient == STATE_5) ->
    printf("State AClient 5 : 11. Печатать чек\n");
    if
    ::stateAClient = STATE_8;
      printf("Going to state AClient 8 : 12. Печать чека 2\n");
      lastEvent = 3;
```

```
        printf("Event = e3\n");
      ::stateAClient = STATE_7;
        printf("Going to state AClient 7 : 13. Возврат карты\n");
        lastEvent = 2;
        printf("Event = e2\n");
      fi;
    ::(stateAClient == STATE_6) ->
      printf("State AClient 6 : s2\n");
      break;
    ::(stateAClient == STATE_7) ->
      printf("State AClient 7 : 13. Возврат карты\n");
      if
      ::stateAClient = STATE_13;
        printf("Going to state AClient 13 : 1. Вставьте карту\n");
        lastEvent = 7;
        printf("Event = e7\n");
      fi;
    ::(stateAClient == STATE_8) ->
      printf("State AClient 8 : 12. Печать чека 2\n");
      if
      ::stateAClient = STATE_7;
        printf("Going to state AClient 7 : 13. Возврат карты\n");
        lastEvent = 8;
        printf("Event = e8\n");
      fi;
    ::(stateAClient == STATE_9) ->
      printf("State AClient 9 : 2. Ввод pin кода\n");
      if
      ::stateAClient = STATE_4;
        printf("Going to state AClient 4 : 3. Авторизация\n");
        lastEvent = 4;
        printf("Event = e4\n");
      ::stateAClient = STATE_7;
        printf("Going to state AClient 7 : 13. Возврат карты\n");
        lastEvent = 2;
        printf("Event = e2\n");
      fi;
    ::(stateAClient == STATE_10) ->
      printf("State AClient 10 : 8. Ввод суммы\n");
      if
      ::stateAClient = STATE_12;
        printf("Going to state AClient 12 : 9. Запрос денег\n");
        lastEvent = 4;
        printf("Event = e4\n");
      ::stateAClient = STATE_7;
        printf("Going to state AClient 7 : 13. Возврат карты\n");
        lastEvent = 2;
        printf("Event = e2\n");
      fi;
    ::(stateAClient == STATE_11) ->
      printf("State AClient 11 : 10. Выдача денег\n");
      if
      ::stateAClient = STATE_5;
        printf("Going to state AClient 5 : 11. Печатать чек\n");
        lastEvent = 9;
        printf("Event = e9\n");
      fi;
    ::(stateAClient == STATE_12) ->
      printf("State AClient 12 : 9. Запрос денег\n");
      printf("Calling automaton AServer\n");
      AServer();
      if
```

```
        ::stateAClient = STATE_11;
          printf("Going to state AClient 11 : 10. Выдача денег\n");
          lastEvent = 13;
          printf("Event = e13\n");
        ::stateAClient = STATE_7;
          printf("Going to state AClient 7 : 13. Возврат карты\n");
          lastEvent = 14;
          printf("Event = e14\n");
        fi;
      ::(stateAClient == STATE_13) ->
        printf("State AClient 13 : 1. Вставьте карту\n");
        if
        ::stateAClient = STATE_6;
          printf("Going to state AClient 6 : s2\n");
          lastEvent = 0;
          printf("Event = e0\n");
        ::stateAClient = STATE_9;
          printf("Going to state AClient 9 : 2. Ввод pin кода\n");
          lastEvent = 6;
          printf("Event = e6\n");
        fi;
      ::(stateAClient == STATE_14) ->
        printf("State AClient 14 : 7. Печать чека 1\n");
        if
        ::stateAClient = STATE_7;
          printf("Going to state AClient 7 : 13. Возврат карты\n");
          lastEvent = 8;
          printf("Event = e8\n");
        fi;
      ::(stateAClient == STATE_15) ->
        printf("State AClient 15 : 6. Вывод на экран\n");
        if
        ::stateAClient = STATE_14;
          printf("Going to state AClient 14 : 7. Печать чека 1\n");
          lastEvent = 3;
          printf("Event = e3\n");
        ::stateAClient = STATE_7;
          printf("Going to state AClient 7 : 13. Возврат карты\n");
          lastEvent = 4;
          printf("Event = e4\n");
        fi;
      od;
    }
    #define STATE_16  16 /*Top*/
    #define STATE_17  17 /*s1*/
    #define STATE_18  18 /*Ответ клиенту*/
    #define STATE_19  19 /*s2*/
    #define STATE_20  20 /*Снятие денег*/
    #define STATE_21  21 /*Авторизация*/
    #define STATE_22  22 /*Баланс*/
    #define STATE_23  23 /*Чтение запроса*/

    int stateAServer;
    inline AServer() {
      stateAServer = STATE_17;
      do
      ::(stateAServer == STATE_17) ->
        printf("State AServer 17 : s1\n");
        if
        ::stateAServer = STATE_23;
          printf("Going to state AServer 23 : Чтение запроса\n");
        fi;
```

```
    ::(stateAServer == STATE_18) ->
      printf("State AServer 18 : Ответ клиенту\n");
      if
      ::stateAServer = STATE_19;
        printf("Going to state AServer 19 : s2\n");
      fi;
    ::(stateAServer == STATE_19) ->
      printf("State AServer 19 : s2\n");
      break;
    ::(stateAServer == STATE_20) ->
      printf("State AServer 20 : Снятие денег\n");
      if
      ::stateAServer = STATE_18;
        printf("Going to state AServer 18 : Ответ клиенту\n");
        lastEvent = 24;
        printf("Event = e24\n");
      fi;
    ::(stateAServer == STATE_21) ->
      printf("State AServer 21 : Авторизация\n");
      if
      ::stateAServer = STATE_18;
        printf("Going to state AServer 18 : Ответ клиенту\n");
        lastEvent = 24;
        printf("Event = e24\n");
      fi;
    ::(stateAServer == STATE_22) ->
      printf("State AServer 22 : Баланс\n");
      if
      ::stateAServer = STATE_18;
        printf("Going to state AServer 18 : Ответ клиенту\n");
        lastEvent = 24;
        printf("Event = e24\n");
      fi;
    ::(stateAServer == STATE_23) ->
      printf("State AServer 23 : Чтение запроса\n");
      if
      ::stateAServer = STATE_20;
        printf("Going to state AServer 20 : Снятие денег\n");
        lastEvent = 23;
        printf("Event = e23\n");
      ::stateAServer = STATE_21;
        printf("Going to state AServer 21 : Авторизация\n");
        lastEvent = 21;
        printf("Event = e21\n");
      ::stateAServer = STATE_22;
        printf("Going to state AServer 22 : Баланс\n");
        lastEvent = 22;
        printf("Event = e22\n");
      fi;
  od;
}
proctype Model() {
  AClient();
}

init {
run Model();
}
```

Проведем верификацию банкомата с использованием инструментального средства *Converter*. На рис. 19, рис. 20 показаны номера, которые *Converter* присвоил состояниям системы автоматов.

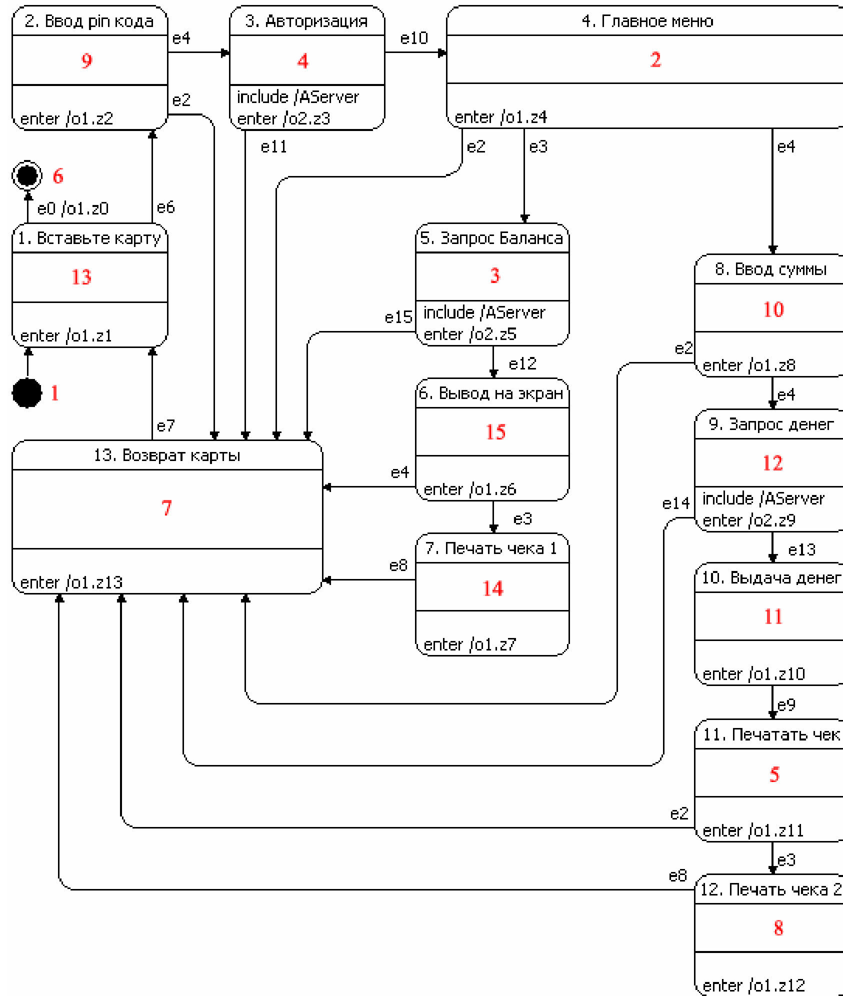

Рис. 19. Автомат *AClient*

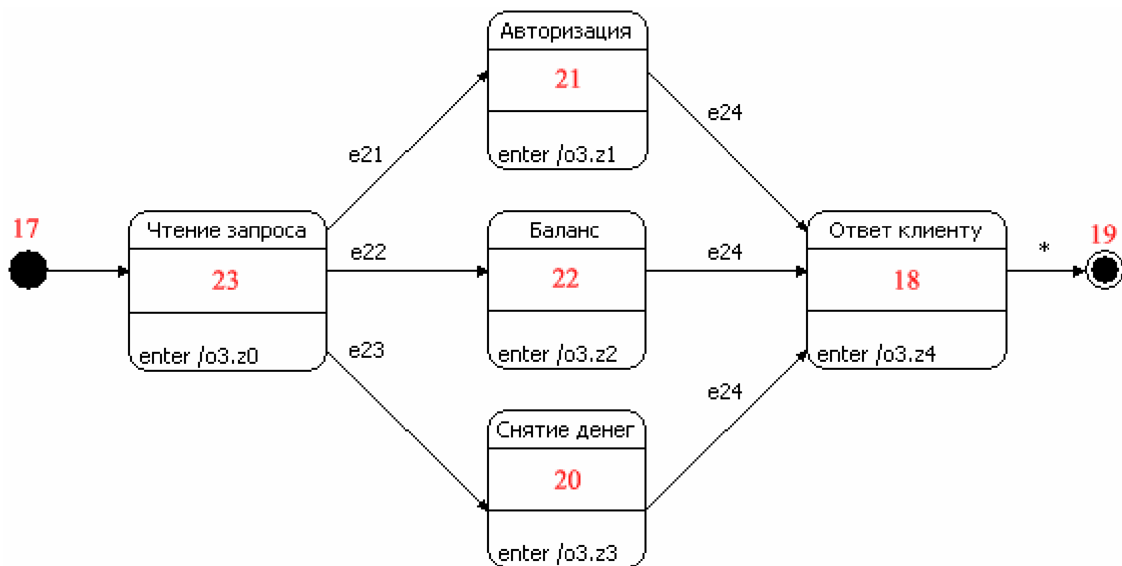

Рис. 20. Автомат *AServer*

Будем проверять следующие свойства:

- если пользователь пытается снять больше денег, чем доступно по карте, выдачи денег не произойдёт;
- банкомат не дает денег.

Рассмотрим свойство «Если пользователь пытается снять больше денег, чем доступно по карте, выдачи денег не произойдёт». Запишем отрицание этого свойства на русском языке: «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 построения *LTL*-формулы введем два элементарных высказывания:

```
lastEvent == e14
```

и

```
stateAClient == 11.
```

Первое элементарное высказывание означает, что произошло событие *e14* («Нет запрашиваемой суммы»). Второе элементарное высказывание означает, что автомат *AClient* попадет в состояние №11 «*10. Выдача денег*» и выдаст деньги.

Перефразируем данное свойство (точнее, его отрицание) через введенные элементарные высказывания: «когда-нибудь произойдет событие *e14* и последнее произошедшее событие будет *e14* до тех пор, пока не произойдет выдача денег». Подформула «последнее произошедшее событие будет *e14* до тех пор, пока не произойдет выдача денег» записывается следующим образом:

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

Для того, чтобы событие *e14* обязательно произошло, требуется конструкция

```
({lastEvent == e14}) && ( … ),
```

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 *q*.

В результате получим следующую *LTL*-формулу для данного свойства:

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

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

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

```
unimod\Converter>spin -a models/aclient.ltl
unimod\Converter>gcc pan.c -o pan.exe

unimod\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)


unimod\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
```

свидетельствует о том, что ошибок нет.

Перейдем к рассмотрению свойства «Банкомат не дает денег». Запишем отрицание этого свойства на русском языке: «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:

```
unimod\Converter>spin -a models/aclient.ltl
unimod\Converter>gcc pan.c -o pan.exe

unimod\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)


unimod\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
```

свидетельствует 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*».

2.3. Инструментальное средство *UniMod.Verifier*

Инструментальное средство *Unimod.Verifier* реализует метод верификации на основе эмуляции.

2.3.1. Установка и настройка

Рабочий каталог инструментального средства *UniMod.Verifier* имеет следующую структуру (табл.6).

Таблица 6. Рабочий каталог инструментального средства *UniMod.Verifier*

`Converter`	Папка, содержащая программу для конвертации автоматной модели, разработанной в инструментальном средстве *UniMod*, в модель инструментального средства *UniMod2*.
`Example`	Папка, содержащая пример верификации банкомата.
`Lib`	Папка, содержащая библиотеки, необходимые для работы инструментального средства.
`ltl.bogor-conf`	Файл, содержащий настройки верификатора *Bogor*.
`ltl2ba.exe`	Вспомогательная программа, предназначенная для преобразования *LTL*-формулы в автомат Бюхи.
`Unimod.bir`	Файл, куда записываются *LTL* -формулы для верификации. Кроме того, в нем содержится служебная информация.
`verifier.bat`	Скрипт, запускающий инструментальное средство. Этот скрипт предназначен для запуска в среде ОС *Windows*.

Для установки инструментального средства достаточно скопировать рабочий каталог *UniMod.Verifier* на локальный компьютер.

Верификатор и используемые им библиотеки написаны на платформо-независимом языке *Java*. Поэтому для работы с верификатором необходимо, чтобы на компьютере была установлена программа *Java Runtime Environment* версии 1.5.0. Верификатор запускается с помощью скриптов, работающих в операционной системе *Microsoft Window*. Инструментальное средство *UniMod.Verifier* может также работать и в других операционных системах, например, в операционных системах типа *Linux*. Для работы в них необходимо переписать скрипты, содержащиеся в папке `UniMod.Verifier`. Кроме того, необходимо заменить версию программы *ltl2ba.exe* на версию, работающую в соответствующей операционной системе.

2.3.2. Пример применения

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

2.3.2.1. Подготовка *UniMod*-модели

Инструментальное средство *UniMod.Verifier* предназначено для работы с автоматными программами, разработанными с использованием инструментального средства *UniMod* версии 2. Данная версия на момент написания отчета еще не была официально опубликована, однако, исходные коды программы доступны по адресу

https://unimod.svn.sourceforge.net/svnroot/unimod/unimod-2. Существует также возможность верифицировать автоматные программы, разработанные в инструментальном средстве *UniMod*, с помощью конвертации модели автоматной программы в новый формат *UniMod 2*. Подробно опишем, что для этого необходимо сделать.

Сначала разрабатывается автоматная программа в среде *UniMod*. Подробная инструкция содержится на сайте http://unimod.sourceforge.net.

Допустим, что автоматная программа в среде *UniMod* уже разработана и требуется ее верифицировать. Сначала необходимо создать *XML*-описание модели. Для этого на диаграмме одного из автоматов необходимо нажать правой кнопкой мыши, и в контекстном меню выбрать *Export to runtime XML*. Это действие изображено на рис. 21.

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

Далее необходимо преобразовать *XML*-описание в новый формат *UniMod 2*. Для этого требуется зайти в каталог `converter` в рабочей папке `UniMod.Verifier`, и выполнить скрипт `convert.bat`, указав первым параметром путь к *XML*-описанию автоматной программы, а вторым – название нового *XML*-описания в формате *UniMod 2*. На рис. 22 изображен пример запуска скрипта.

В результате конвертации в папке `converter` будут созданы несколько файлов, описывающих ту же автоматную модель, но в формате *UniMod 2*.

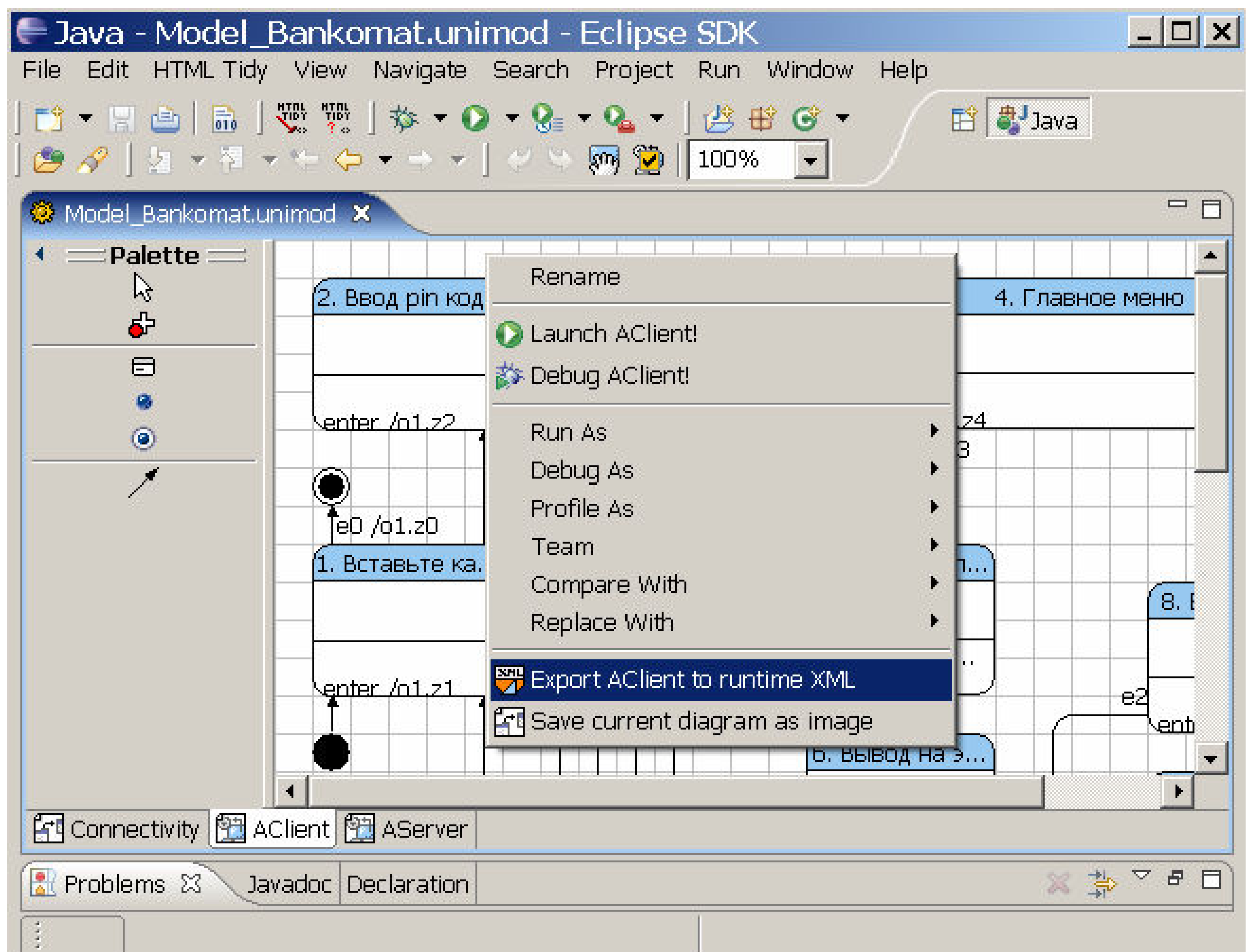

Рис. 21. Создание *XML*-описания автоматной программы

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Версия 5.1.2600]
(C) Корпорация Майкрософт, 1985-2001.

C:\UniMod.Verifier\converter>convert.bat AClient.xml NewBank
Done converting old model into new model: [AClient.xml] -> [NewBank].

C:\UniMod.Verifier\converter>
```

Рис. 22. Конвертация *XML*-описания в новый формат *UniMod 2*

Например, для модели банкомата, описанной в разд 1.1 отчета по третьему этапу, будут созданы файлы, перечисленные в табл. 7.

Таблица 7. Созданные файлы

`NewBank.connectivity`	Описание связей между автоматами, объектами управления и источниками событий в автоматной программе.
`NewBank_AClient.statechart`	Описание автомата *AClient*.
`NewBank_AServer.statechart`	Описание автомата *AServer*.

Теперь необходимо отредактировать скрипт `verifier.bat` в папке `UniMod.Verifier`, указав на созданное описание автоматной программы. Путь к описанию указывается в последней строчке скрипта после слов `com.unimod.verifier.Main`. Необходимо указать путь к файлу с расширением `.connectivity`. Например,

```
java com.unimod.verifier.Main example/NewBank.connectivity
   Unimod.bir %1
```

Таким образом, верификатор настроен для верификации разработанной в инструментальном средстве *UniMod* автоматной программы.

2.3.2.2. Задание формул для верификации

Теперь необходимо в файле *Unimod.bir* записать *LTL*-формулы для верификации. Формула объявляется в виде функции. Функцию следует объявлять в конце файла, но до последней закрывающей фигурной скобки «}». Опишем на примере банкомата, как создавать формулы.

Предположим, что верифицируется автоматная программа, описывающая работу банкомата. Также предположим, что требуется проверить, что банкомат выдает деньги только после авторизации. Таким образом, начальное словесное описание свойства имеет вид: «Банкомат выдает деньги только после авторизации». Перефразируем описание следующим образом: «Банкомат не выдает деньги, пока пользователь не введет правильный *PIN*-код». Надо записать это свойство в виде *LTL*-формулы. Для этого в словесном описании выделяются предикаты – логические утверждения, и условные и временные зависимости. В приведенном примере временная зависимость – это фраза «пока не», а предикаты – это «банкомат выдает деньги» и «пользователь вводит правильный *PIN*-код». Таким образом, получаем следующую *LTL*-формулу:

```
[банкомат не выдает денег] <пока не> [пользователь ввел правильный
         PIN-код]
```

В табл. 8 приведен полный список условных и темпоральных операторов, которые можно использовать в формулах в верификаторе *Bogor*, а, следовательно, и в верификаторе *UniMod.Verifier*. В правой колонке – описание оператора, а в левой – его формальное определение в файле `Unimod.bir`.

Таблица 8. Список операторов 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);`	$\neg$ (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). Оператор $p\ U\ q$ означает, что p будет верно до тех пор, пока не выполнится q. При этом q обязано когда-либо выполниться.
`expdef LTL.Formula weakUntil (LTL.Formula, LTL.Formula);`	W (Weak until, до тех пор пока или всегда). Этот оператор был добавлен для удобства. $p\ W\ q = (p\ U\ q)\ \mid G\ (p \cap \neg q)$. $p\ W\ q$ – это то же самое, что $p\ U\ q$, однако q не обязано когда-либо выполниться.
`expdef LTL.Formula release (LTL.Formula, LTL.Formula);`	R (Release, освобождение)
`expdef LTL.Formula equivalence (LTL.Formula, LTL.Formula);`	$\leftrightarrow$ (Эквивалентно). $p \leftrightarrow q = (p \rightarrow q)\ \&\ (q \rightarrow p)$
`expdef LTL.Formula implication (LTL.Formula, LTL.Formula);`	$\rightarrow$ (Следует)
`expdef LTL.Formula conjunction (LTL.Formula, LTL.Formula);`	& (И)
`expdef LTL.Formula disjunction (LTL.Formula, LTL.Formula);`	| (Или)

Таким образом, в данном примере «пока не» будет записано оператором W. Оператор U не подходит, поскольку в верифицируемом свойстве не требуется, чтобы пользователь когда-либо ввел правильный *PIN*-код. Требуется лишь, чтобы деньги могли быть выданы лишь в том случае, если он это сделает.

Далее необходимо записать предикаты в терминах автоматной программы. Предикаты задаются как действия объектов управления, события или состояния автоматов. Поэтому надо достаточно хорошо знать верифицируемые автоматы, чтобы точно описать предикат. В данном примере банкомата существует объект управления `o1`, отвечающий за основные действия банкомата, и в нем указано действие `z10` – «выдача денег». Таким образом, фраза «банкомат не выдает денег» может быть записана как `!o1.z10`. Правильно введенный *PIN*-код характеризуется в банкомате событием `e10`. Тогда *LTL*-формула из словесного вида преобразуется в формулу:

```
!o1.z10 W e10
```

Теперь можно записать эту формулу в виде функции в файле `Unimod.bir`:

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

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

Подробно разберем структуру этой функции. В первой строке объявляется функция и задается ее имя: `NoPinNoMoney`. Это имя затем будет использоваться для указания верификатору какую формулу верифицировать. Далее внутри функции `LTL.temporalProperty(…)` задается сама *LTL*-формула.

Описание *LTL*-формулы состоит из двух частей: описание предикатов и описание зависимостей. Предикаты записываются внутри функции `Property.createObservableDictionary(…)`, где через запятую перечисляются предикаты в виде функций `Property.createObservableKey(…)`. Первый аргумент объявления предиката – это его краткое уникальное название. Например, предикату «`e10`» было дано название «`correct_pin`». Второй аргумент – это функция, которая вызывается для вычисления значения предиката. Например, функция `AutomataModel.wasEvent(model, "e10")` возвращает истину, если последним обработанным автоматной программой событием было событие с названием «`e10`». Полный список таких функций объявляется в файле `Unimod.bir`. Описание функций приведено в табл. 9.

Таблица 9.Описание функций автомата

`expdef Boolean wasEvent(AutomataModel.type model, string event);`	Возвращает `True`, если в последнем шаге было выбрано для обработки событие `event`, и `False` в противном случае.
`expdef boolean wasInState(AutomataModel.type model, string stateMachine, string state);`	Возвращает `True`, если перед последним шагом автомат `stateMachine` находился в состоянии `state`.
`expdef boolean isInState(AutomataModel.type model, string stateMachine, string state);`	Возвращает `True`, если после выполнения последнего шага автомат `stateMachine`, находится в состоянии `state`.
`expdef boolean cameToState(AutomataModel.type model, string stateMachine, string state);`	возвращает `True`, если после выполнения последнего шага автомат `stateMachine` сменил свое состояние на `state`. Тоже, что `(isInState(stateMachine, state) && !wasInState(stateMachine, state))`.
`expdef boolean cameToFinalState(AutomataModel.type model);`	Возвращает `True`, если после выполнения шага корневой автомат модели вошел в свое конечное состояние. Это означает, что автоматная программа завершила работу.
`expdef boolean wasAction(AutomataModel.type model, string action);`	Возвращает `True`, если в ходе выполнения шага было вызвано выходное воздействие `action`.
`expdef boolean wasFirstAction(AutomataModel.type model, string action);`	Возвращает `True`, если в ходе выполнения шага первым вызванным действием было `action`.
`expdef boolean wasLastAction(AutomataModel.type model, string action);`	Возвращает `True`, если в ходе выполнения шага последним вызванным действием было `action`.
`expdef int getActionIndex(AutomataModel.type model, string action);`	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.
`expdef boolean wasTrue(AutomataModel.type model, string guard);`	Возвращает `True`, если в ходе выполнения последнего шага один из переходов был помечен условием `guard`, и его значение было определено как `True`. Условие `guard` описывает целое условие, а не значение одной переменной. Например, `guard = «!o1.x1 && o1.x2»`.
`expdef boolean wasFalse(AutomataModel.type model, string guard);`	Возвращает `True`, если в ходе выполнения последнего шага на одном из переходов было встречено условие `guard`, и его значение было определено как `False`.

В некоторых перечисленных функциях участвует имя автомата. Поскольку в системе автоматов один автомат может быть вложен в несколько состояний своего родителя, 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>».

Во второй части объявления *LTL*-формулы описываются временные и условные зависимости между предикатами. Эти зависимости являются вложенными функциями. Например, функция `LTL.weakUntil` описывает оператор *weakUntil*, У этой функции два аргумента – левая и правая часть оператора *weakUntil*. В левой части выражение `!o1.z10` – это отрицание предиката `o1.z10`. Отрицание описывается функцией `LTL.negation`. Сам предикат объявляется с помощью функции `LTL.prop`, аргументом которой является уникальное имя предиката, данное ему в первой части объявления функции `NoPinNoMoney`. В данном случае это «`give_money`».

2.3.2.3. Верификация свойств

Теперь, когда формула записана в файл `Unimod.bir`, все готово для верификации. Для того, чтобы запустить верификатор необходимо выполнить скрипт `verifier.bat` с параметром – именем функции, описывающей *LTL*-формулу. Верификатор проверит формулу и выведет результат в консоль. Например, верификация полученной выше формулы на автоматной программе банкомата приведет к результату, изображенному на рис. 23.

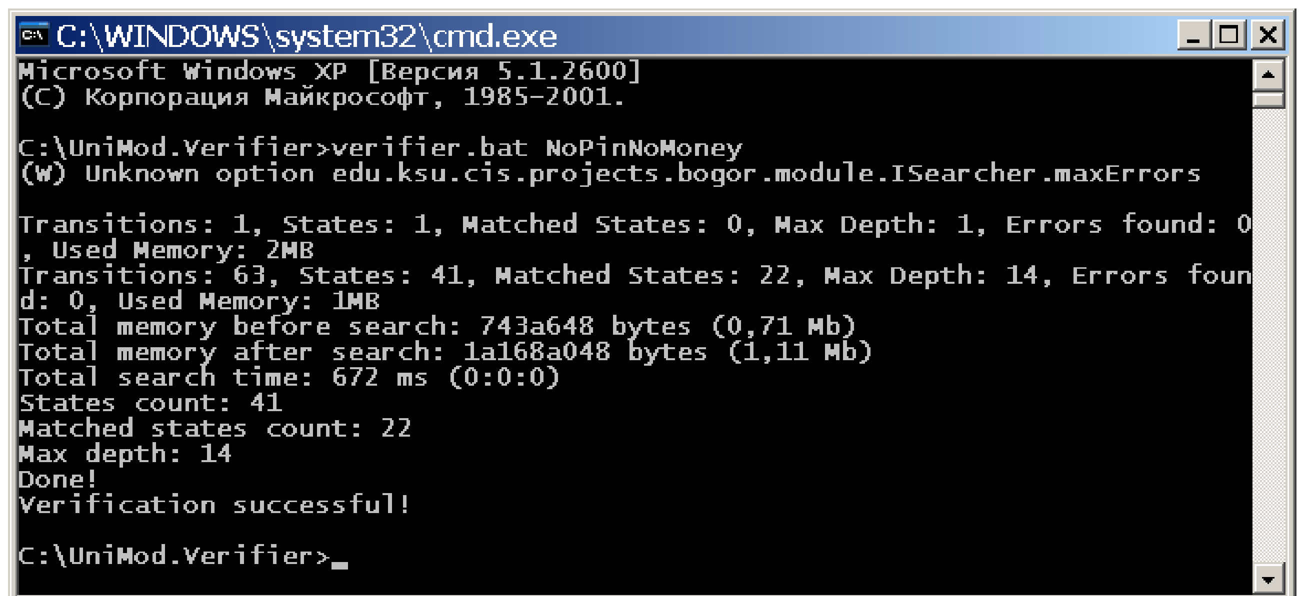

Рис. 23. Результат верификации формулы *NoPinNoMoney*

Верификатор выводит различную служебную информацию, например количество памяти, занимаемой *Java*-машиной, число состояний, пройденное верификатором. В последней строчке выводится результат: «`Verification successful!`». Это означает, что формула выполняется в автоматной программе, а значит верно, что спроектированный банкомат никогда не выдает денег до того, как пользователь введет правильный PIN-код.

Верифицируем теперь неверное свойство банкомата, для того чтобы показать работу со сценариями ошибок. Проверим заведомо ложное свойство: «Когда-нибудь банкомат обязательно выдаст деньги». Это неверно, например, поскольку если никогда никто не введет правильный PIN-код, то банкомат никогда и не выдаст денег. Запишем это свойство в виде *LTL*-формулы:

```
<когда-нибудь> [банкомат выдает деньги]
```

Темпоральное свойство «когда-нибудь» задается темпоральным оператором *F* («*Future*»), а выдача денег происходит с действием `o1.z10`, как и в предыдущем примере. Тогда формула принимает следующий вид:

```
F o1.z10
```

Запишем это в виде функции «`AlwaysMoney`» в файле `Unimod.bir`:

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

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

Для вычисления значения предиката используется функция `AutomataModel.wasAction(…)`, а функция `LTL.eventually(…)` соответствует темпоральному оператору *F*.

Запустим верификацию записанной формулы. Для удобства выведем результат верификации в файл. Для этого после команды надо добавить «`> имя_файла`»:

```
verifier.bat AlwaysMoney > result.txt
```

После завершения выполнения команды, в папке, где команда была запущена, будет создан файл `result.txt`. Его содержимое следующее:

```
(W) Unknown option
        edu.ksu.cis.projects.bogor.module.ISearcher.maxErrors

Transitions: 1, States: 1, Matched States: 0, Max Depth: 1, Errors
        found: 0, Used Memory: 2MB
Transitions: 1, States: 1, Matched States: 0, Max Depth: 1, Errors
        found: 0, Used Memory: 1MB
Transitions: 1, States: 1, Matched States: 0, Max Depth: 1, Errors
        found: 0, Used Memory: 1MB
Transitions: 1, States: 1, Matched States: 0, Max Depth: 1, Errors
        found: 0, Used Memory: 1MB
Transitions: 1, States: 1, Matched States: 0, Max Depth: 1, Errors
        found: 0, Used Memory: 2MB
Transitions: 1, States: 1, Matched States: 0, Max Depth: 1, Errors
        found: 0, Used Memory: 1MB
Transitions: 2, States: 2, Matched States: 0, Max Depth: 2, Errors
        found: 1, Used Memory: 1MB
Transitions: 2, States: 2, Matched States: 0, Max Depth: 2, Errors
        found: 1, Used Memory: 1MB
Total memory before search: 725 944 bytes (0,69 Mb)
Total memory after search: 1 160 544 bytes (1,11 Mb)
Total search time: 4032 ms (0:0:4)
States count: 2
Matched states count: 0
Max depth: 2

Generating error trace 0...

Done!

1 traces were found.
Replaying the trace with least states (#0).

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 [bad$accept_init]
Model [ step [0] event [] guards [] transitions [] actions [] states
        [(/AClient:9. Запрос денег/AServer) - (Top); (/AClient:5.
        Запрос Баланса/AServer) - (Top); (/AClient:3.
        Авторизация/AServer) - (Top); (/AClient) - (Top)] ]
        fsaState [bad$accept_init]
Model [ step [1] event [*] guards [] transitions [s1#1. Вставьте
        карту#*#true] actions [o1.z1] states [(/AClient:9. Запрос
        денег/AServer) - (Top); (/AClient:5. Запрос
        Баланса/AServer) - (Top); (/AClient:3. Авторизация/AServer)
        - (Top); (/AClient) - (1. Вставьте карту)] ] fsaState
        [bad$accept_init]
Model [ step [2] event [e6] guards [true->true] transitions [1.
        Вставьте карту#2. Ввод pin кода#e6#true] actions [o1.z2]
        states [(/AClient:9. Запрос денег/AServer) - (Top);
        (/AClient:5. Запрос Баланса/AServer) - (Top); (/AClient:3.
        Авторизация/AServer) - (Top); (/AClient) - (2. Ввод pin
        кода)] ] fsaState [bad$accept_init]
Model [ step [3] event [e2] guards [true->true] transitions [2. Ввод
        pin кода#13. Возврат карты#e2#true] actions [o1.z13] states
        [(/AClient:9. Запрос денег/AServer) - (Top); (/AClient:5.
        Запрос Баланса/AServer) - (Top); (/AClient:3.
        Авторизация/AServer) - (Top); (/AClient) - (13. Возврат
        карты)] ] fsaState [bad$accept_init]
Model [ step [4] event [e7] guards [true->true] transitions [13.
        Возврат карты#1. Вставьте карту#e7#true] actions [o1.z1]
        states [(/AClient:9. Запрос денег/AServer) - (Top);
        (/AClient:5. Запрос Баланса/AServer) - (Top); (/AClient:3.
        Авторизация/AServer) - (Top); (/AClient) - (1. Вставьте
        карту)] ] fsaState [bad$accept_init]

Done!
```

В начале файла приводится служебная информация. Строки вида:

```
Transitions: 1, States: 1, Matched States: 0, Max Depth: 1, Errors
        found: 0, Used Memory: 2MB
```

выводятся периодически в процессе верификации, для того чтобы показывать прогресс работы инструментального средства.

Далее следует фраза

```
1 traces were found.
```

Это означает, что была найдена одна ошибка – один сценарий работы автоматной программы, нарушающий проверяемую формулу. Далее приводится этот сценарий в виде последовательности строк вида

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

Каждая строка – это набор параметров и их значений. Перепишем строку в виде таблицы, описывая значение каждого параметра (табл. 10).

Таблица 10. Описание выводимых значений

Step	3	Номер шага в сценарии ошибки, или (номер момента времени).
Event	e2	Название обработанного в шаге события.
guards	true->true	Список вычисленных в ходе обработки события условий на переходах. True -> true означает, что был только один переход с условием «true» (безусловный переход), и это условие было вычислено как true.
transitions	2. Ввод pin кода#13. Возврат карты#e2#true	Список выполненных переходов. Переход записывается в формате: <стартовое состояние># <конечное состояние># <событие над переходом># <условие над переходом>.
actions	o1.z13	Вызванные в ходе выполнения шага действия.
states	(/AClient:9. Запрос денег/AServer) - (Top); (/AClient:5. Запрос Баланса/AServer) - (Top); (/AClient:3. Авторизация/AServer) - (Top); (/AClient) - (13. Возврат карты)	Состояния автоматов после выполнения шага. Задается путь к автомату и его состояние. Состояние Top означает, что автомат еще не был инициализирован.
fsaState	bad$accept_init	Состояние автомата Бюхи после выполнения шага. В данном случае автомат Бюхи уже находится в допускающем состоянии, поскольку название состояния начинается с «bad$».

Для того чтобы разобраться в сценарии ошибки, выведенном верификатором, следует посмотреть на диаграммы автоматов. В данном случае хватит одной диаграммы автомата AClient банкомата. Она изображена на рис. 24.

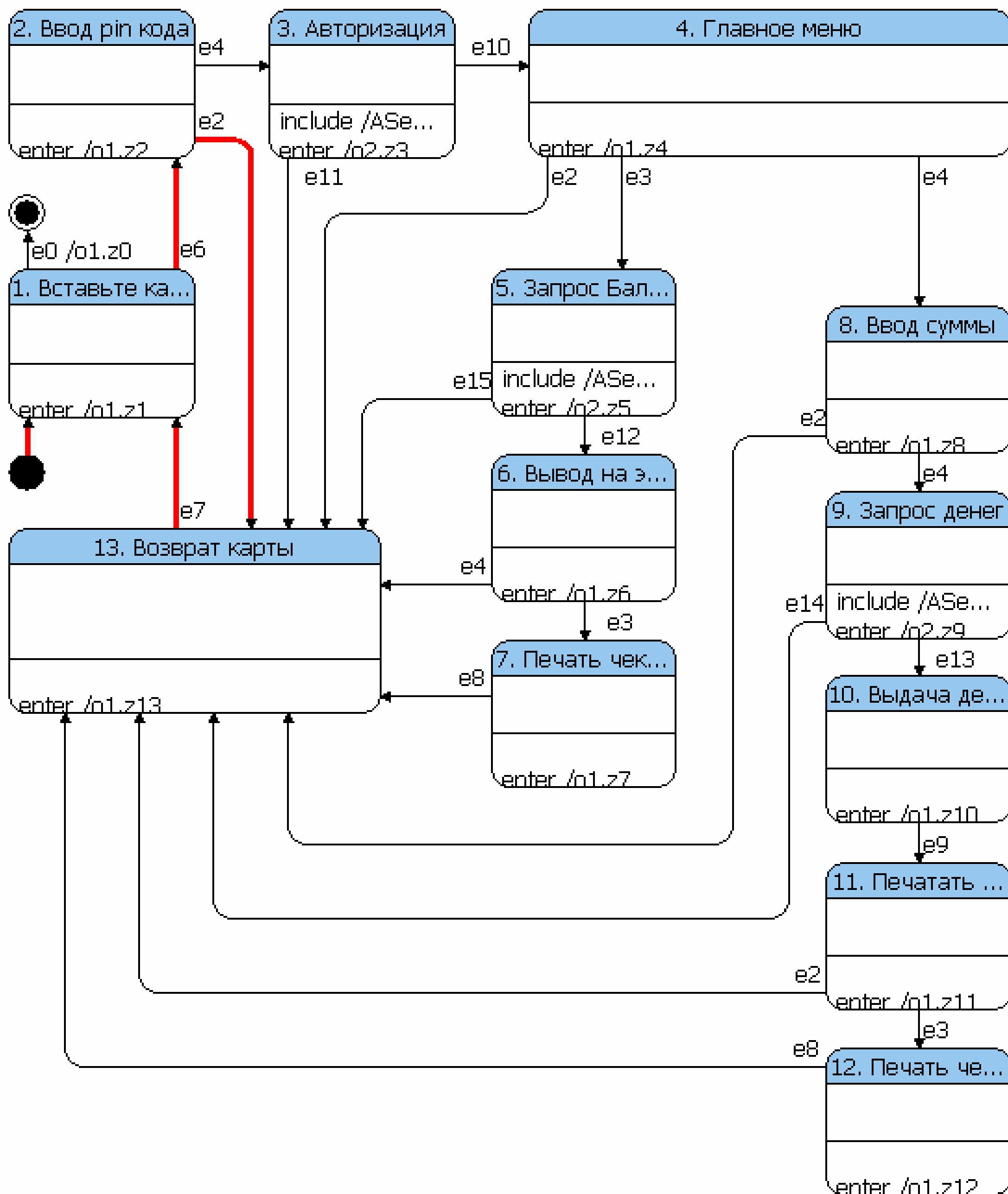

Рис. 24. Контрпример формулы *AlwaysMoney*

Разберем шаги сценария ошибки.

Нулевой шаг:

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

Две первые записи помечены номером «0». Они не важны для воспроизведения ошибки. В первой записи верификатор инициализирует автоматную программу, а во второй

– автоматная программа инициализирована, но ни одной обработки события еще не произошло. Поэтому все автоматы находятся в состояниях `Top`.

Первый шаг:

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

На первом шаге произошла обработка события «`*`» – это специальное событие инструментального средства автоматного программирования *UniMod*, которое означает, что переход необходимо сделать сразу, без ожидания события. На диаграмме оно не отмечено, однако автоматически добавляется к тем переходам, которые не помечены ни одним событием. В ходе обработки произошел один переход: из состояния «`s1`» в состояние «`1. Вставьте карту`». На рис. 24 все переходы, совершенные в этом сценарии, помечены жирными линиями.

Автомат Бюхи с самого начала находится в допускающем состоянии. Хотя структура сгенерированного автомата Бюхи явно не выводится, ее можно посмотреть следующим образом. Когда верификатор находит ошибки, он генерирует в папке `UniMod.Verifier` специальный файл `Unimod.bir.bogor-trails`. Этот файл является *ZIP*-архивом. Если открыть его как архив, внутри кроме прочего будет файл `Unimod.bir`. Его содержимое совпадает с содержимым файла `Unimod.bir` в папке `UniMod.Verifier`, однако верифицируемая *LTL*-формула заменена на описание автомата Бюхи. В данном примере функция `AlwaysMoney` была заменена на автомат Бюхи:

```
function generated$FSA()
{

    loc bad$accept_init:
        when !(AutomataModel.wasAction(model, "o1.z10")) do
        {
        }
        goto bad$accept_init;
}
```

Он состоит из единственного допускающего состояния. Из этого состояния есть петля при условии, что не выполнилось действие `o1.z10`. В противном случае никакого перехода не определено. Это означает, что если выполнится действие `o1.z10`, то не будет найдено возможного перехода в автомате Бюхи. Это автоматически будет означать, что исходная *LTL*-формула выполнилась. Если же `o1.z10` никогда не выполнится, то автомат Бюхи останется в допускающем состоянии. Это означает, что исходная *LTL*-формула не выполнилась.

Второй шаг:

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

Выполнилось событие «`e6`». При этом был совершен переход из состояния «`1. Вставьте карту`» в состояние «`2. Ввод pin кода`».

Третий шаг:

```
Model [ step [3] event [e2] guards [true->true] transitions [2. Ввод
        pin кода#13. Возврат карты#e2#true] actions [o1.z13] states
```

```
          [(/AClient:9. Запрос денег/AServer) - (Top); (/AClient:5.
          Запрос Баланса/AServer) - (Top); (/AClient:3.
          Авторизация/AServer) - (Top); (/AClient) - (13. Возврат
          карты)] ] fsaState [bad$accept_init]
```

Выполнилось событие «e2» (пользователь нажал кнопку «Отмена»). При этом был совершен переход из состояния «2. Ввод pin кода» в состояние «13. Возврат карты».

Четвертый шаг:

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

Произошло событие «e7» (карта была вынута), и автомат AClient перешел в состояние «1. Вставьте карту». Сравним наборы состояний автоматов (в том числе автомата Бюхи) на шаге «1» и «4». Они одинаковы. При этом автомат Бюхи находится в допускающем состоянии. Таким образом, получен цикл: автоматная программа может далее работать по тому же сценарию, как между шагами «1» и «4». При этом никогда не выдаст денег, поскольку не будет выполнено действие o1.z10. Таким образом, выданный верификатором сценарий действительно является контрпримером к верифицируемой *LTL*-формуле. Физически это означает, что пользователь может постоянно вставлять карту, нажимать «Отмена» и доставать карту, и никогда не получит денег.

2.4. Инструментальное средство *CTLVerifier*

Инструментальное средство *CTLVerifier* реализует метод верификации на основе темпоральной логики *CTL*.

2.4.1. Установка и настройка

В настоящем разделе приведено краткое описание инструментального средства *CTLVerifier*.

2.4.1.1. Общие сведения

Программное средство называется *CTLVerifier* и имеет наименование проекта *CTLVerif*. Проект состоит из исходных кодов, исполняемого файла и демонстрационных примеров.

Программа реализована на языке программирования *Delphi 7.0*. Исходные файлы проекта состоят из текста программы (CTLVerif.dpr) и конфигурации компилятора (CTLVerif.cfg, CTLVerif.dof).

Исполняемый файл называется CTLVerif.exe. Программа работает под *Microsoft Windows XP* и не требует дополнительного программного обеспечения для своего функционирования.

Демонстрационные примеры предназначены для иллюстрации возможностей инструментального средства и включают в себя модели и свойства, описанные в разд. 2.2.2 отчёта по третьему этапу. Каждый пример состоит из файла с расширением dat, который содержит входные данные для программного средства (автоматную модель и проверяемые свойства), и файла с расширением out, который содержит выходные данные программного средства (модель Крипке и списки состояний, в которых соответствующие свойства выполняются, вместе с подтверждающими трассами). Имена примеров, рассмотренных в разд. 2.2.2 отчёта по третьему этапу, начинаются с “Ex”, имена примеров с изменённой моделью заканчиваются на “alt”.

2.4.1.2. Функциональное назначение

Программное средство выполняет следующие функции. Оно получает на вход смешанные автоматные модели (взаимодействующие по вложенности) и строит для них модель Крипке, которая описывает поведение этих автоматных моделей. После этого инструментальное средство позволяет выполняет верификацию формул, поданных на вход. Поскольку верификация выполняется индуктивно по подформулам, и верификатор в процессе работы проверяет, кроме основной формулы, все её подформулы, то каждая из этих подформул имеет свой идентификатор, и для каждой из них генерируется свой результат. Верификатор возвращает на выходе 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, эта трасса выводится.

Ограничения на размер модели задаются в файле с исходным текстом программы. Константа `maxAuto` обозначает максимальное число обрабатываемых автоматов, а константа `maxVar` – максимальное число позиций в модели. По умолчанию значения этих констант следующие: `maxAuto` = 100, `maxVar` = 10000. При желании их можно изменить, перекомпилировав программу. Число автоматов и позиций ограничивается только размером доступной оперативной памяти.

2.4.1.3. Описание логической структуры

Процесс работы программного средства разбит на следующие этапы:

- чтение служебных входных данных для инструментального средства;
- чтение списка состояний и переходов для каждого автомата;
- построение модели Крипке по автоматным моделям;
- для каждого темпорального свойства проверка его на модели;
- построение множества состояний, выполняющих данное свойство;
- при необходимости для каждого из этих состояний генерация пути в модели, который позволяет убедиться в правильности формулы.

Модель Крипке строится по методу редукции – состояния и переходы разбиваются на последовательности *позиций* – элементарных состояний. Автоматы взаимодействуют по вложенности, для обработки такого взаимодействия программа выполняет топологическую сортировку графа вложенности автоматов.

Верификация выполняется индуктивно по подформулам. Для каждой подформулы программа определяет список состояний модели Крипке, в которых выполняется эта подформула. Проверка пропозициональных операций достаточно проста. Проверка темпоральных операций использует дополнительные структуры данных. Для каждого состояния строится список входящих и исходящих рёбер. Для проверки формулы $\mathbf{EX}g$ требуется сделать «один шаг назад» от позиций, в которых выполняется формула g. Для проверки формулы $\mathbf{E}[f\,\mathbf{U}\,g]$ строятся деревья обратных путей от вершин, выполняющих формулу g, вдоль вершин, выполняющих формулу f.

Проверка формулы $\mathbf{EG}\,g$ происходит в три этапа:

- исключение из обработки всех состояний, не выполняющих формулу g;
- поиск сильно связных компонент в оставшемся графе (применяется алгоритм Тарьяна, который использует линейное время относительно размера модели);
- построение деревьев обратных путей от этих компонент в оставшемся графе.

Для каждой формулы построение множества выполняющих состояний занимает время, линейное от размера модели.

Все описанные алгоритмы работают с применением алгоритма поиска в глубину в графе.

Для формулы $\mathbf{EX}g$ предоставляется подтверждение правильности: номер состояния, достижимого за один переход из текущего состояния, в котором выполняется формула *g*.

Для формулы $\mathbf{E}[f\,\mathbf{U}\,g]$ в качестве подтверждения предоставляется путь из текущего состояния в состояние, выполняющее формулу *g*, который проходит вдоль состояний, выполняющих формулу *f*.

Для формулы $\mathbf{EG}g$ в качестве подтверждения предоставляется путь, который, начиная с некоторого состояния (возможно, с первого), является периодическим (рис. 16 отчёта по третьему этапу). Выделение предпериода и периода производится алгоритмом «кролика и черепахи», суть которого состоит в том, что по рассматриваемому пути в графе «бегут» два индекса. За каждую итерацию алгоритма первый индекс делает один шаг вперёд, а второй – два. Так как на каждой итерации расстояние между первым и вторым индексами изменяется на единицу, рано или поздно наступит момент, когда они сравняются. Значение, которому они будут равны, обязательно содержится в цикле. После этого можно сделать ещё один проход по циклу, помечая его вершины и, тем самым, отделяя его от части, которая находится перед ним. Этот алгоритм работает за время, пропорциональное сумме длин предпериода и периода.

Программа является автономной и не требует внешнего верификатора для своей работы.

Основной компонентой программы является класс `TAutomaton`.

```
TAutomaton = class
   n: integer;

   posNum: integer;
   positions: array [1 .. maxVar] of record
     apNum: integer;
     ap: array [1 .. maxAuto] of string;
   end;
   digits: integer;

   edgeNum: integer;
   edges: array [1 .. maxVar] of record
     p: integer;
     q: integer;
   end;

   stateNum: integer;
   stateNames: array [1 .. maxAuto] of string;
   stateInfo: array [1 .. maxAuto] of record
      enterInModel: integer;
       exitInModel: integer;
            actNum: integer;
           actions: array [1 .. maxAuto] of integer;
           nestNum: integer;
              nest: array [1 .. maxAuto] of integer;
     hasTransition: boolean;
   end;

    start: integer;
   finish: integer;

   transNum: integer;
   transInfo: array [1 .. maxAuto] of record
     from, into: integer;
          event: integer;
         inpNum: integer;
         inputs: array [1 .. maxAuto] of integer;
         actNum: integer;
```

```
         actions: array [1 .. maxAuto] of integer;
    end;

    // Service routines
    constructor Init(num: integer);
    procedure ReadStateInfo;
    procedure ExtractStateInfo;
    procedure ReadTransitions(var s: string);
    procedure AddState(i: integer);
    procedure AddTransition(i: integer);
    procedure RenderKripke;

    // Model checking routines
    procedure CheckFalse;
    procedure CheckTrue;
    procedure CheckAtomic(s: string);
    procedure CheckNotAtomic(s: string);
    procedure CheckNot(x: integer);
    procedure CheckAnd(x, y: integer);
    procedure CheckOr(x, y: integer);
    procedure CheckXOr(x, y: integer);
    procedure CheckEX(x: integer);
    procedure CheckEU(x, y: integer);
    procedure CheckEG(x: integer);
end;
```

В качестве полей класс содержит идентификационный номер, информацию о состояниях и переходах фиксированного автомата и информацию о состояниях и переходах модели Крипке, построенной по этому автомату. Класс предоставляет две группы методов. Служебная группа содержит процедуры для чтения данных об автомате в специальные поля (методы `ReadStateInfo`, `ExtractStateInfo` и `ReadTransitions`) и формирования модели Крипке (в других полях) на основе этих данных (методы `AddState`, `AddTransition` и `RenderKripke`). Метод `AddState` формирует информацию о состоянии (воздействия, которые выполняются при входе в это состояние, и вызываемые из него автоматы). Метод `AddTransition` формирует информацию о переходе (события и входные условия, при которых выполняется переход, а также выходные воздействия на переходе). Метод `RenderKripke` выводит список состояний и переходов модели Крипке и строит граф её переходов.

В качестве примера приведем реализацию метода, который для каждого состояния модели Крипке на основе списка всех рёбер модели строит список входящих и исходящих рёбер:

```
for i := 1 to edgeNum do with edges[i] do begin
  writeln(edges[i].p : digits, ' -> ', edges[i].q);
  with model[p] do begin
    inc(outNum);
    _out[outNum] := q;
  end;
  with model[q] do begin
    inc(inNum);
    _in[inNum] := p;
  end;
end;
```

При построении модели важно знать, что вложенные автоматы обрабатываются раньше тех, в которые они вложены. Для построения этого порядка используется топологическая сортировка графа вложенности:

```
procedure TopoSort;
var i, n: integer;
```

```
  procedure DFS(i: integer);
  var j: integer;
  begin  with nestGraph[i] do begin
    scanned := 1;
    for j := 1 to outNum do case nestGraph[ _out[j] ].scanned of
      0: DFS(_out[j]);
      1: Error('Automata nesting cannot be recursive.');
    end;
    scanned := 2;
    order[n] := i;
    dec(n);
  end  end;

begin
  n := autoNum;
  for i := 1 to autoNum do
    if (nestGraph[i].scanned = 0) then DFS(i);
end;
```

Порядок указывается в массиве `order`.

Атомарные предложения хранятся в отсортированном массиве, и доступ к ним выполняется бинарным поиском за логарифмическое время:

```
function BSearch(var a: array of string;
                 n: integer;
                 s: string) : integer;
var c, l, m, r: integer;
begin
  l := 0;
  r := n - 1;
  while l <= r do begin
    m := (l + r) div 2;
    c := compareText(a[m], s);
    if (c < 0) then l := m + 1 else
    if (c > 0) then r := m - 1 else
    begin
      BSearch := m + 1;
      exit;
    end;
  end;
  BSearch := 0;
end;
```

В классе `TAutomaton` содержатся методы, предназначенные для проверки модели. Каждый из этих методов строит множество состояний, в которых выполняется формула, соответствующая названию метода, и выводит списки этих состояний, добавляя к ним доказательства (для тех формул, для которых это возможно). Например, метод `CheckEU` использует обход графа в глубину для построения дерева обратных путей, которое требуется при проверке формулы вида $\mathbf{E}[\,f\,\mathbf{U}\,g\,]$:

```
procedure DFS(i: integer);
var j: integer;
begin  if not satisfy[propNum, i] then with model[i] do begin
  satisfy[propNum, i] := true;
  for j := 1 to inNum do if satisfy[ x, _in[j] ] then begin
    DFS(_in[j]);
    model[ _in[j] ].trNext := i;
  end;
end  end;
```

Метод `CheckEG` использует алгоритм Тарьяна для обхода сильно связных компонент графа за линейное время относительно размера графа:

```
procedure TarjanVisit(p: integer);
var i, q: integer;
begin
  inc(nL);
  L[nL] := p;
  DFSNum[p] := n;
  lowest[p] := n;
  inc(n);
  for i := 1 to model[p].inNum do begin
    q := model[p]._in[i];
    if satisfy[x, q] and present[q] then if DFSNum[q] = 0 then begin
      TarjanVisit(q);
      if lowest[q] < lowest[p] then lowest[p] := lowest[q];
    end else if DFSNum[q] < lowest[p] then lowest[p] := DFSNum[q];
  end;
  if (lowest[p] = DFSNum[p]) then
    if (L[nL] = p) and noLoop[p] then begin
      present[ L[nL] ] := false;
      dec(nL);
    end else
    repeat
      satisfy[ propNum, L[nL] ] := true;
      present[ L[nL] ] := false;
      dec(nL);
    until (L[nL + 1] = p);
end;
```

Напомним, что отношение сильной связности для моделей Крипке не рефлексивно: вершина тогда и только тогда является сильно связной с самой собой, если из неё можно добраться в неё же, сделав *как минимум один* переход.

Поиск и вывод циклической трассы выполняется с использованием двух индексов `m` и `n`:

```
m := i;
n := i;
repeat
  m := model[m].trNext;
  n := model[n].trNext;
  n := model[n].trNext;
until (m = n);
repeat
  n := model[n].trNext;
  L[n] := 1
until (m = n);
n := i;
while (L[n] = 0) do begin
  write(n, ' ');
  n := model[n].trNext;
end;
write('(');
m := n;
repeat
  write(' ', n);
  n := model[n].trNext;
until (m = n);
writeln(' )');
```

2.4.1.4. Используемые технические средства

Для нормального функционирования программы *CTLVerifier* на персональном компьютере необходимо, чтобы аппаратное обеспечение удовлетворяло следующим требованиям:

- процессор *Intel Pentium IV* или совместимый с ними;
- тактовая частота процессора не менее 1000 МГц;
- оперативная память 256 MB;
- устройства ввода и вывода (клавиатура, монитор, дисковый накопитель).

Размер памяти, необходимой для работы программы, можно изменять путём установки значений констант `maxAuto` и `maxVar` в исходном тексте. Значения, установленные по умолчанию (`maxAuto` = 100, `maxVar` = 10000), используют память в объёме по 4MB на каждую автоматную модель.

2.4.1.5. Вызов и загрузка

Для просмотра исходных кодов программы следует зайти в каталог `Project` и открыть в нём файл `CTLVerif.dpr`. Файл `CTLVerif.dof` позволяет настраивать параметры компиляции. Исходный текст программы занимает 40 KB.

Для запуска программы следует зайти в каталог `Project\Release` и запустить в нём файл `CTLVerif.exe`. Версия *Release* программы занимает 60 KB, версия *Debug* (отладочная, находится в каталоге `Project\Debug`) – 70 KB. Программное средство имеет следующий синтаксис запуска:

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

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

В каталоге `Examples` находятся демонстрационные примеры для программы (в том числе из разд. 2.2.2 отчёта по третьему этапу). Примеры снабжены краткими комментариями, их можно использовать для обучения построению входного и интерпретации выходного файлов.

Приведем пример запуска программы (предполагается, что файл `AUnimod.dat` находится в одном каталоге с файлом `CTLVerif.exe`):

```
CTLVerif.exe  AUnimod.dat  AUnimod.out
```

2.4.1.6. Входные данные

Программное 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, заключенное в квадратные скобки: `[NameOfSection]`.

Все используемые во входном файле имена могут содержать любые символы, кроме управляющих: “`$`”, “`%`”, “`<`”, “`–`”, “`>`”, “`[`”, “`]`”, “`:`”, “`;`”, “`/`”, “`&`”, “`|`”, “`^`”, “`!`”, “`=`” и разделителей. Имена всех идентификаторов (автоматов, состояний, событий, входных и выходных воздействий, атомарных предложений, свойств) должны быть различны.

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
e3
e4
e6
e7
e8
e9
e10
e11
e12
e13
e14
e15
e21
e22
e23
e24
```

Четвертая 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
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
```

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 в отчете по третьему этапу работ для *UniMod*-модели банкомата, которая не содержит входных переменных. Поэтому для неё эта секция будет пустой:

```
[Inputs]
```

Далее идут секции, описывающие автоматные модели. Каждая секция соответствует одному автомату. Имя секции совпадает с именем автомата. Секция состоит из двух частей, которые разделяются строкой, начинающейся со знака минуса '-' (строка-разделитель). 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 имени идёт двоеточие, 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. Знаки ":" ";" всегда должны присутствовать. Лишних запятых быть не должно, каждый автомат должен иметь стартовое и терминальное состояния.

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;
Y7  : o1.z7;
Y8  : o1.z8;
Y9  : o2.z9;  AServer
Y10 : o1.z10;
Y11 : o1.z11;
Y12 : o1.z12;
Y13 : o1.z13;
```

Во второй части секции для соответствующего автомата приводится информация о каждом переходе, содержащаяся 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 которого указывается событие, инициирующее переход и условие перехода (если оно задано). Событие и условие разделены символом '&' (амперсанд). Условие представляет собой конъюнкцию литералов, составленных из входных переменных – список литералов, разделённых символом '&'. Каждый литерал состоит из имени переменной, перед которой по необходимости знаком отрицания '!'. Предполагается, что если знак отрицания отсутствует, то переход может быть выполнен только при истинном значении соответствующей переменной. Если же указанный знак присутствует, то переход может быть выполнен 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
```

Данная запись означает: переход из состояния Y5 в состояние Y8 по событию e7 при условии истинности x1 и ложности x2 с последовательным выполнением действий z3 и z4.

Знаки "->", ":", "/" всегда должны присутствовать, лишних амперсандов и запятых быть не должно.

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 : e9  /
Y11 -> Y13 : e2  /
Y11 -> Y12 : e3  /
Y12 -> Y13 : e8  /
Y13 -> Y1  : e7  /
```

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 : e9  /
Y11 -> Y13 : e2  /
```

```
Y11 -> Y12 : e3  /
Y12 -> Y13 : e8  /
Y13 -> Y1  : e7  /
```

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> | <имя> '$EU' <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 $\mathbf{EX}$, $\mathbf{EG}$ и $\mathbf{EU}$ следующим образом:

$$\mathbf{AX}\, g = \neg \mathbf{EX} \neg g$$

$$\mathbf{EF}\, g = 1\, \mathbf{EU}\, g$$

$$\mathbf{AF}\, g = \neg \mathbf{EG} \neg g$$

$$\mathbf{AG}\, g = \neg \mathbf{EF} \neg g = \neg(1\, \mathbf{EU}\, g)$$

$$f\, \mathbf{AU}\, g = \neg((\neg g\, \mathbf{EU} \neg(f \vee g)) \vee \mathbf{EG} \neg g)$$

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

```
[Properties]
```

```
f1 = AClient
f2 = InState
f3 = f1 & f2
f4 = !f3
f5 = Y13
f6 = f4 $EU f5
f7 = $EX f6
```

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

$$f_1 = AClient$$
$$f_2 = InState$$
$$f_3 = AClient \wedge InState$$
$$f_4 = \neg(AClient \wedge InState)$$
$$f_5 = y13$$
$$f_6 = \mathbf{E}[\neg(AClient \wedge InState)\,\mathbf{U}\,y13]$$
$$f_7 = f = \mathbf{EX}\,\mathbf{E}[\neg(AClient \wedge InState)\,\mathbf{U}\,y13]$$

2.4.2. Пример применения

2.4.2.1. Примеры входных данных

Приведем пример входного файла, в котором требуется проверить следующие формулы из отчёта по третьему этапу: формулу из разд. 2.2.2.3, первый элемент конъюнкции в примере из разд. 2.2.2.13 и формулу из разд. 2.2.2.5:

```
[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 : e9  /
Y11 -> Y13 : e2  /
Y11 -> Y12 : e3  /
Y12 -> Y13 : e8  /
Y13 -> Y1  : e7  /

[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  = f2 $EU f3
f5  = !f4

f6  = e15
f7  = !f6
f8  = $EG f7

f9  = AClient
f10 = InState
f11 = f9 & f10
f12 = !f11
f13 = Y13
f14 = f12 $EU f13
f15 = $EX f14
```

Из приведённого примера можно следует, что разработанное средство позволяет проверять сразу несколько свойств в одном файле. Например, в указанной выше секции `[Properties]` переменная `f5` соответствует формуле $\neg\mathbf{E}[\neg e10\ \mathbf{U}\ o1.z7]$, переменная `f8` – формуле $\mathbf{EG}\ \neg e15$, а переменная `f15` – формуле $\mathbf{EX}\ \mathbf{E}[\neg(AClient \wedge InState)\mathbf{U}\ y13]$.

2.4.2.2. Примеры выходных данных: модель Крипке

Программное средство имеет следующий формат выходных данных: данные содержатся в одном файле, который, как и входной файл, разделён на секции. Секции идут в строго определённом порядке. Каждая секция начинается с заголовка, который представляет собой имя этой секции, помещённое в квадратные скобки: `[NameOfSection]`.

Первая секция называется `Atomic Propositions`. В ней перечислены названия всех атомарных предложений, содержащихся в модели, по одному названию в строке. Среди них есть имена автоматов, состояний, событий, входных воздействий, входных воздействий с отрицаниями, выходных воздействий, служебные атомарные предложения (`InState`, `InEvent`, `InAction`).

Приведем пример секции атомарных предложений для автомата, эмулирующего работу лифта (отчёт по первому этапу, разд. 1.1.1):

```
[Atomic Propositions]
!x1
AElevator
Closed
Closing
e1
e2
e3
e4
e5
```

```
End
Error
InAction
InEvent
InState
Opened
Opening
x1
z1
z2
z3
```

Приведем приме секции атомарных предложений для рассмотренной модели банкомата:

```
[Atomic Propositions]
AClient
Answer
AServer
Authority
Balance
e0
e1
e10
e11
e12
e13
e14
e15
e2
e21
e22
e23
e24
e3
e4
e6
e7
e8
e9
InAction
InEvent
InState
o1.z0
o1.z1
o1.z10
o1.z11
o1.z12
o1.z13
o1.z2
o1.z4
o1.z6
o1.z7
o1.z8
o2.z3
o2.z5
o2.z9
o3.z0
o3.z1
o3.z2
o3.z3
o3.z4
Read
```

```
Withdraw
Y0
Y1
Y10
Y11
Y12
Y13
Y2
Y3
Y4
Y5
Y6
Y7
Y8
Y9
```

Вторая секция называется `Model Positions` и содержит информацию о состояниях (позициях) модели Крипке: для каждого состояния указан список атомарных предложений, выполненных в этом состоянии. Атомарные предложения перечисляются в лексикографическом порядке. Информация о каждом состоянии указана на отдельной строке. Формат строки следующий:

```
[<пометка состояния>] <порядковый номер состояния> ':'
          {<атомарное предложение>}
```

Номер состояния помечен спереди знаком '`$`', если это состояние является стартовым, и знаком '`%`' – если оно является терминальным.

Приведем пример секции для модели банкомата:

```
[Model Positions]
%1: AClient InState Y0
$2: AClient InAction o1.z1
 3: AClient InState Y1
 4: AClient InAction o1.z10
 5: AClient InState Y10
 6: AClient InAction o1.z11
 7: AClient InState Y11
 8: AClient InAction o1.z12
 9: AClient InState Y12
10: AClient InAction o1.z13
11: AClient InState Y13
12: AClient InAction o1.z2
13: AClient InState Y2
14: AClient InAction o2.z3
15: AServer InAction o3.z4
16: Answer AServer InState
17: AServer InAction o3.z1
18: AServer Authority InState
19: AServer InAction o3.z2
20: AServer Balance InState
21: AServer InAction o3.z0
22: AServer InState Read
23: AServer InAction o3.z3
24: AServer InState Withdraw
25: AServer e21 InEvent
26: AServer e22 InEvent
27: AServer e23 InEvent
28: AServer e24 InEvent
29: AServer e24 InEvent
30: AServer e24 InEvent
31: AClient InState Y3
32: AClient InAction o1.z4
33: AClient InState Y4
```

```
34: AClient InAction o2.z5
35: AServer InAction o3.z4
36: Answer AServer InState
37: AServer InAction o3.z1
38: AServer Authority InState
39: AServer InAction o3.z2
40: AServer Balance InState
41: AServer InAction o3.z0
42: AServer InState Read
43: AServer InAction o3.z3
44: AServer InState Withdraw
45: AServer e21 InEvent
46: AServer e22 InEvent
47: AServer e23 InEvent
48: AServer e24 InEvent
49: AServer e24 InEvent
50: AServer e24 InEvent
51: AClient InState Y5
52: AClient InAction o1.z6
53: AClient InState Y6
54: AClient InAction o1.z7
55: AClient InState Y7
56: AClient InAction o1.z8
57: AClient InState Y8
58: AClient InAction o2.z9
59: AServer InAction o3.z4
60: Answer AServer InState
61: AServer InAction o3.z1
62: AServer Authority InState
63: AServer InAction o3.z2
64: AServer Balance InState
65: AServer InAction o3.z0
66: AServer InState Read
67: AServer InAction o3.z3
68: AServer InState Withdraw
69: AServer e21 InEvent
70: AServer e22 InEvent
71: AServer e23 InEvent
72: AServer e24 InEvent
73: AServer e24 InEvent
74: AServer e24 InEvent
75: AClient InState Y9
76: AClient e0 InEvent
77: AClient InAction o1.z0
78: AClient e6 InEvent
79: AClient e4 InEvent
80: AClient e2 InEvent
81: AClient e10 InEvent
82: AClient e11 InEvent
83: AClient e2 InEvent
84: AClient e3 InEvent
85: AClient e4 InEvent
86: AClient e12 InEvent
87: AClient e15 InEvent
88: AClient e3 InEvent
89: AClient e4 InEvent
90: AClient e8 InEvent
91: AClient e2 InEvent
92: AClient e4 InEvent
93: AClient e14 InEvent
94: AClient e13 InEvent
95: AClient e9 InEvent
```

```
96: AClient e2 InEvent
97: AClient e3 InEvent
98: AClient e8 InEvent
99: AClient e7 InEvent
```

Третья секция называется `Model Edges` и содержит список всех переходов в модели Крипке. Каждый переход описывается на отдельной строке. Синтаксис описания:

```
<номер состояния, откуда ведёт ребро> '->'
         <номер состояния, куда ведёт ребро>
```

Приведем пример секции для модели банкомата:

```
[Model Edges]
 2 -> 3
 4 -> 5
 6 -> 7
 8 -> 9
10 -> 11
12 -> 13
15 -> 16
17 -> 18
19 -> 20
21 -> 22
23 -> 24
22->25
25->17
22->26
26->19
22->27
27->23
18->28
28->15
20->29
29->15
24->30
30->15
14->21
16->31
32->33
35->36
37->38
39->40
41->42
43->44
42->45
45->37
42->46
46->39
42->47
47->43
38->48
48->35
40->49
49->35
44->50
50->35
34->41
36->51
52->53
54->55
56->57
59->60
61->62
```

```
63->64
65->66
67->68
66->69
69->61
66->70
70->63
66->71
71->67
62->72
72->59
64->73
73->59
68->74
74->59
58->65
60->75
 3->76
76->77
77->1
 3->78
78->12
13->79
79->14
13->80
80->10
31->81
81->32
31->82
82->10
33->83
83->10
33->84
84->34
33->85
85->56
51->86
86->52
51->87
87->10
53->88
88->54
53->89
89->10
55->90
90->10
57->91
91->10
57->92
92->58
75->93
93->10
75->94
94->4
 5->95
95->6
 7->96
96->10
 7->97
97->8
 9->98
98->10
```

```
11->99
99->2
 1->1
```

Далее идут секции, описывающие проверенные свойства. Формат этих секций определён в следующем разделе.

2.4.2.3. Примеры выходных данных: результаты верификации

После секций, описывающих модель Крипке, в выходном файле размещены секции, описывающие проверенные свойства. Каждая секция соответствует одному свойству. Секции следуют друг за другом в том порядке, в котором свойства следовали во входном файле. Имя секции совпадает с именем формулы. Секция содержит список состояний, в которых данная формула выполняется. В каждой строке размещается по одному состоянию. Если секция пуста, то формула не выполняется ни в одном состоянии. Для формулы вида “`1`” выполняющими являются все состояния. Поэтому для неё вместо списка состояний печатается сообщение.

Для формул, которые получены пропозициональной операцией, в каждой строке записывается число – номер очередного состояния, выполняющего данную формулу. Приведем пример секции для формулы `f11 = AClient & InState`, рассмотренной в примере входных данных для банкомата (разд. 2.4.2.1):

```
[f11]
1
3
5
7
9
11
13
31
33
51
53
55
57
75
```

Для формул, которые получены темпоральной операцией, каждое выполняющее состояние снабжается доказательством.

Для формулы вида “`$EX f`” в каждой строке указывается номер выполняющего состояния `x`, за ним следует двоеточие ‘`:`’, после которого указан номер некоторого состояния, выполняющего формулу, в которое можно перейти из состояния `x` за один переход.

Например, формуле $f_{15} = \mathbf{EX}\,\mathbf{E}[\neg(AClient \wedge InState)\,\mathbf{U}\,y13]$ соответствует секция:

```
[f15]
 7: 96
 9: 98
10: 11
13: 80
31: 82
33: 83
51: 87
53: 89
55: 90
57: 91
75: 93
80: 10
82: 10
```

```
83: 10
87: 10
89: 10
90: 10
91: 10
93: 10
96: 10
98: 10
```

Для формулы вида “`f $EU g`” в каждой строке указывается список выполняющих состояний, который проходит через состояния, выполняющие формулу `f`, и заканчивается в состоянии, выполняющем формулу `g`. Этот список образует подтверждающую трассу для формулы “`f $EU g`”.

Приведем пример для формулы $f_{14} = \mathbf{E}[\neg(AClient \wedge InState)\ \mathbf{U}\ y13]$:

```
[f14]
10 11
11
80 10 11
82 10 11
83 10 11
87 10 11
89 10 11
90 10 11
91 10 11
93 10 11
96 10 11
98 10 11
```

Для формулы вида “`$EG f`” в каждой строке сначала указывается путь, начинающийся в соответствующем (выполняющем формулу `f`) состоянии, который с некоторого момента (возможно, начального) становится периодическим. В строке вначале через пробел записан список состояний из предпериода, а затем в скобках – список состояний из периода (по аналогии с периодическими числовыми дробями).

Рассмотрим формулу $f_8 = \mathbf{EG}\ \neg e15$. Проверим, выполняется ли она в позиции номер 6 (эта позиция выделена жирным шрифтом на рис. 25). Можно убедиться, что это так. Верификатор выдал подтверждающую трассу “`6 7 96 10 11 99 2 3 76 77 ( 1 )`”, которая изображена на рис. 25 (цикл состоит из одного терминального состояния).

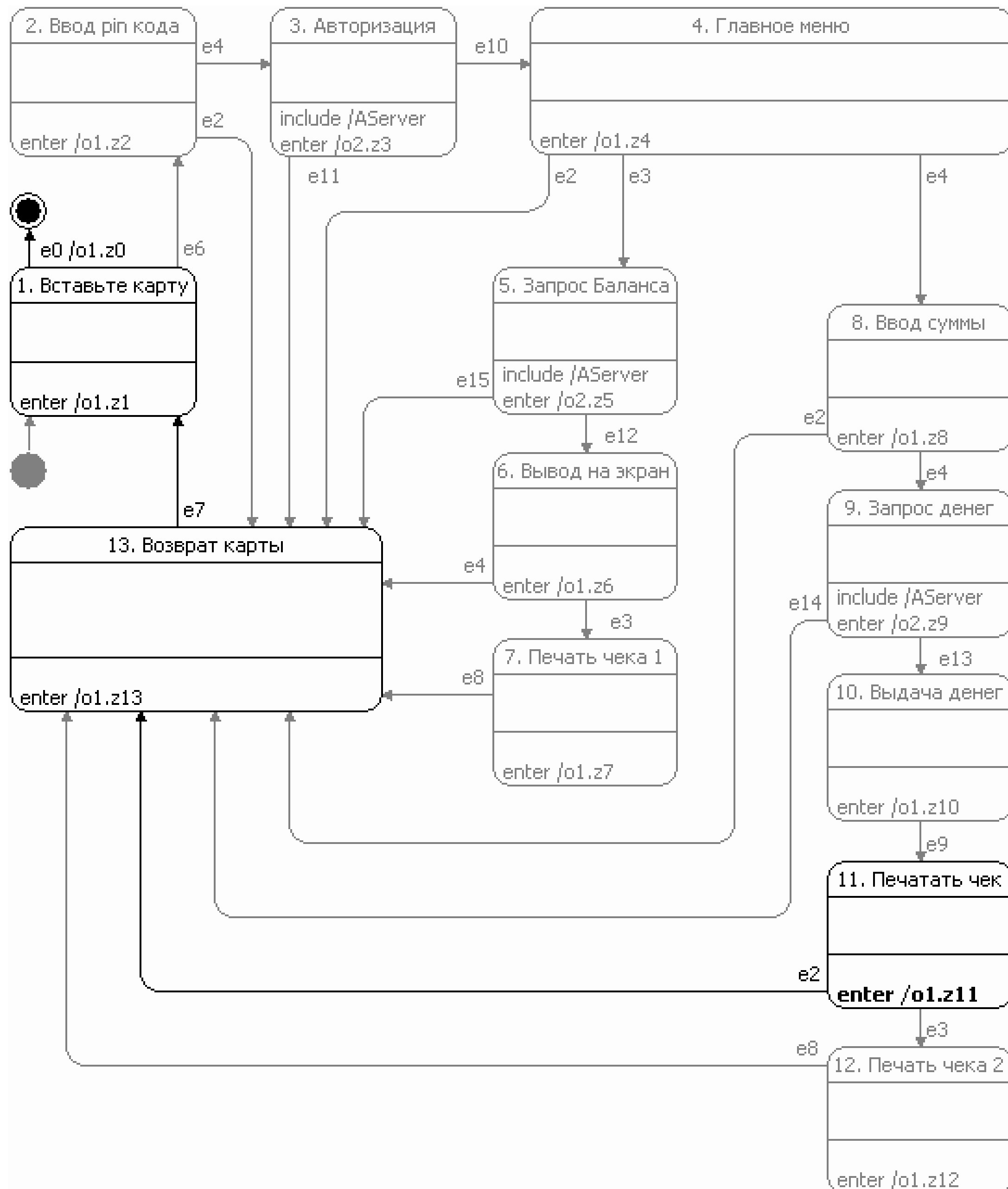

Рис. 25. Подтверждающая трасса для формулы `f8`

Укажем способ, который позволяет по трассе, записанной в терминах верификатора, получить осмысленный путь в терминах автоматной модели. Каждый элемент подтверждающей трассы является позицией модели Крипке. При этом атомарные предложения, которыми помечена каждая позиция определяют смысл данной позиции и элемент автомата, который ей соответствует. Например, рассмотренная трасса начинается с позиции `6`. В секции `[Model Positions]` указано, что этой позиции соответствуют атомарные предложения `AClient`, `InAction`, `o1.z11`. Это означает, что данная позиция представляет собой выходное воздействие `o1.z11`, сгенерированное во время выполнения автомата `AClient`. Такая позиция в автомате одна (она выделена на рис. 25 жирным шрифтом), но даже если бы их было несколько, требуемая трасса определялась бы однозначно. Действительно, следующая позиция в трассе имеет номер `7`, секция `[Model Positions]` однозначно определяет, что это состояние `Y=11` автомата `AClient`. Поэтому позиция `6` означает выходное воздействие, имеющее переход именно в состояние `Y=11`. Также однозначно определяются позиции, соответствующие событиям, значения всех

существенных входных переменных, проверяемых в случае, если произошли эти события, и название автомата, которые в этот момент выполнялся.

2.5. Выводы

Описаны четыре прототипа инструментальных средств для верификации управляющих программ со сложным поведением, основанных на автоматном подходе. Для каждого метода даны рекомендации по его настройке и запуску, а также приведен пример его применения.

ЗАКЛЮЧЕНИЕ

В результате исследований, выполненных на четвертом, заключительном 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 даны методические указания и рекомендации по применению методов, разработанных в рамках работ по контракту. При этом даны рекомендации по выбору метода верификации и указания по применению отдельных методов.

Во второй главе приведены описания прототипов инструментальных средств, созданных на основе разработанных методов и даны методические рекомендации по их применению.

Разработан дополненный пакет программной документации для инструментального средства *UniMod.Verifier*, как наиболее удобного в использовании.

Таким образом, были решены все задачи, поставленные в техническом задании на проведение четвертого этапа работ.

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 ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. *Вельдер С. Э., Шалыто А. А.* О верификации автоматных программ на основе метода Model Checking // Информационно-управляющие системы. 2007. № 3, с. 27–38.
2. *Гуров В., Мазин М., Нарвский А., Шалыто А*. UML. SWITCH-технология. Eclipse // Информационно-управляющие системы. 2004. № 6. http://is.ifmo.ru/works/uml-switch-eclipse/
3. *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
4. *Кларк Э., Грамберг О., Пелед Д.* Верификация моделей программ: Model Checking. – М.: МЦНМО, 2002.
5. *Хоффман Л.* Разговоры о Model Checking. http://is.ifmo.ru/verification/_model_checking.pdf
6. *Классификация абстрактных автоматов.* http://ru.wikipedia.org/wiki/Классификация_абстрактных_автоматов
7. *Козлов В. А., Комалёва О. А.* Моделирование работы банкомата // http://is.ifmo.ru/UniMod-projects/bankomat/
8. *Новиков Ф. А.* Визуальное конструирование программ. // Информационно-управляющие системы. 2005. № 6, с. 9–22. http://is.ifmo.ru/works/visualcons/
9. *Первушин Е. В., Шалыто А. А.* Моделирование банкомата. http://is.ifmo.ru/projects/bankomat/
10. *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/
11. *Шалыто А. А., Туккель Н. И.* Объектно-ориентированное программирование с явным выделением состояний / Материалы Международной научно-технической конференции «Искусственный интеллект». Т.1. Таганрогский гос. радиотехнический университет, Донецкий гос. институт искусственного интеллекта. 2002.
12. *Шалыто А. А., Туккель Н. И.* Программирование с явным выделением состояний // Мир ПК. 2001. № 8, 9. http://is.ifmo.ru/works/mirk/
13. *Яковлев А. В., Лукин М. А., Шалыто А. А.* Реализация классической игры "Ним" на основе автоматного подхода. СПБГУИТМО, 2005. http://is.ifmo.ru/UniMod-projects/knim/
14. *Büchi automaton.* http://en.wikipedia.org/wiki/Büchi_automaton
15. *Clarke E. M., Emerson E. A.* Synthesis of synchronisation skeletons for branching time logic /Logic of Programs, LNCS 131, pp. 52–71, 1981.
16. *Emerson E. A., Clarke E. M.* Using branching time temporal logic to synthesize synchronisation skeletons // Science of Computer Programming 2: 241–266, 1982.
17. Finite state machine. http://en.wikipedia.org/wiki/Finite_state_machine
18. *Linear temporal logic.* http://en.wikipedia.org/wiki/Linear_temporal_logic
19. *New Symbolic Model Verifier.* http://nusmv.irst.itc.it/
20. *Promela reference – ltl.* http://www.spinroot.com/spin/Man/ltl.html
21. *Robby, Dwyer M., Hatcliff J. Bogor*: A Flexible Framework for Creating Software Model Checkers. TAIC PART 2006, pp 3–22.
22. *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.

23. *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
24. *SPIN home page.* http://SPINroot.com
25. *UniMod home page.* http://*UniMod*.sourceforge.net/