МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ
(МИНОБРНАУКИ)

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

УДК 004.4’242
№ госрегистрации 0120.0 710294
Инв. №

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

«____» ___________ 2007 г.

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

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

ПРОМЕЖУТОЧНЫЙ ОТЧЕТ ПО II ЭТАПУ «ТЕОРЕТИЧЕСКИЕ ИССЛЕДОВАНИЯ ПОСТАВЛЕННЫХ ПЕРЕД НИР ЗАДАЧ»

ЛИСТОВ 105

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

2007

СПИСОК ИСПОЛНИТЕЛЕЙ

Руководитель темы Заведующий кафедрой докт. техн. наук, профессор	А. А. Шалыто	Отчет в целом
Ответственный исполнитель Доцент, канд. техн. наук	Г. А. Корнеев	Отчет в целом
Заведующий кафедрой, докт. физ.-мат. наук, профессор	В. А. Соколов	Разделы 1.1, 2.1 и 3.1.
Ведущий научный сотрудник, докт. техн. наук, профессор	В. В. Антипов	Разделы 2.2 и 3.2.
Ведущий научный сотрудник, канд. техн. наук	В. В. Киселев	Разделы 2.1 и 3.1.
Ведущий научный сотрудник, канд. техн. наук	Р. Н. Котляр	Разделы 2.3 и 3.3.
Ведущий научный сотрудник, канд. техн. наук	Ю. П. Московцев	Разделы 3.1 и 3.4.
Ведущий научный сотрудник, канд. техн. наук, доцент	В. А. Третьяков	Разделы 3.2 и 3.3.
Ведущий научный сотрудник, канд. техн. наук	Г. М. Файкин	Разделы 2.2 и 2.3.
Доцент, канд. физ.-мат. наук	Е. В. Кузьмин	Разделы 2.1 и 2.4.
Ассистент	В. С. Гуров	Отчет в целом
Руководитель лаборатории	С. П. Жуков	Разделы 3.1 и 3.4
Канд. тех. наук	С. П. Новиков	Разделы 1.1 и 1.2.
Руководитель ВТК, ассистент кафедры КТ	А. П. Мельничук	Раздел 2.1.
Член ВТК, профессор кафедры ИС	Е. Ю. Михайлова	Раздел 2.1.1.
Член ВТК, доцент кафедры КТ	В. Д. Наумчик	Раздел 2.1.4.
Член ВТК, доцент кафедры КТ	М. Ю. Осипов	Раздел 2.1.3.
Член ВТК, доцент кафедры КТ	А. Н. Воробьев	Раздел 2.1.1.
Член ВТК, доцент кафедры КТ	Ю. А. Щупак	Раздел 2.1.3.
Член ВТК, доцент кафедры КТ	С. В. Чириков	Раздел 2.1.4.
Член ВТК, доцент кафедры КТ	А. С. Сегаль	Раздел 2.1.2.
Член ВТК, доцент кафедры КТ	Д. Г. Шопырин	Раздел 2.1.1.
Член ВТК, доцент кафедры ИС	Д. А. Зубок	Раздел 2.1.4.
Член ВТК, ассистент кафедры ИС	В. В. Повышев	Раздел 2.1.2.
Член ВТК, ассистент кафедры ИС	В. В. Ильин	Раздел 2.1.1.
Член ВТК, ассистент кафедры ИС	М. Г. Холин	Раздел 2.1.3.
Магистрант	Б. Р. Яминов	Разделы 1.3, 2.1.3, 2.2.3, 2.3.3 и 3.3.
Магистрант	С. Э. Вельдер	Разделы 2.2 и 2.3.
Магистрант	М. А. Лукин	Разделы 1.2, 2.1.2, 2.2.2, 2.3.2 и 3.2.
Студент	М. Э. Дворкин	Разделы 2.1 и 2.4.
Студент	Е. А. Курбацкий	Разделы 1.1, 2.1.1, 2.2.1, 2.3.1 и 3.1.
Студент	Ю. А. Беляева	Разделы 1.1, 2.1.1, 2.2.1, 2.3.1 и 3.1.

РЕФЕРАТ

Излагаются методы автоматизации верификации автоматных моделей управляющих программ, разработанные при проведении второго этапа 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).

Предложен набор методов, позволяющих автоматизировать верификацию автоматных моделей управляющих программ. Для разработанных методов приведены их функциональные особенности и характеристики. Описываются созданные прототипы автоматизированных верификаторов автоматных моделей управляющих программ.

ОГЛАВЛЕНИЕ

СПИСОК ИСПОЛНИТЕЛЕЙ ... 2
РЕФЕРАТ ... 3
ОГЛАВЛЕНИЕ ... 4
ВВЕДЕНИЕ ... 6
1. МЕТОДЫ ВЕРИФИКАЦИИ АВТОМАТНЫХ МОДЕЛЕЙ УПРАВЛЯЮЩИХ ПРОГРАММ ... 8
1.1. Метод верификации автоматной мождели с использованием верификатора NuSMV ... 9
1.1.1. Постановка задачи ... 9
1.1.2. Система автоматов ... 10
1.1.3. Описание метода ... 12
1.1.3.1. Общая структура метода ... 12
1.1.3.2. Построение модели автомата ... 13
1.1.3.3. Преобразование автоматной системы в модель Крипке ... 14
1.1.3.4. Требования — формулы темпоральной логики на модели Крипке ... 16
1.1.3.5. Преобразование контрпримера в модели Крипке в контрпример в автоматной модели ... 17
1.1.4. Применение верификатора *NuSMV* ... 18
1.1.4.1. Описание языка *SMV* ... 18
1.1.4.2. Построение по системе автоматов модели на языке *SMV* ... 19
1.1.5. Применение метода верификации автоматных программ и использованием верификатора NuSMV ... 21
1.1.5.1. Описание примера ... 21
1.1.5.2. Построение модели ... 22
1.1.5.3. Построение модели на языке *SMV* ... 24
1.1.5.4. Проверка требований к модели ... 27
1.2. Метод верификации визуальных автоматных моделей ... 31
1.2.1. Постановка задачи ... 31
1.2.2. Построение по визуальной автоматной программе модели на языке *Promela* ... 32
1.2.3. Расширение нотации верификатора *SPIN* для языка линейной темпоральной логики ... 37
1.2.4. Типовые спецификации к автоматным моделям ... 38
1.2.5. Пример использования предлагаемого метода ... 39
1.2.5.1. Постановка задачи ... 39
1.2.5.2. Верификация ... 40
1.2.5.3. Анализ контрпримера ... 44
1.3. Метод верификации автоматных моделей на основе их эмуляции ... 46
1.3.1. Постановка задачи ... 46
1.3.2. Верификация с использованием алгоритма двойного поиска в глубину ... 49
1.3.3. Общее описание метода ... 54
1.3.4. Пример использования ... 57
1.3.5. Модель Крипке ... 62
1.4. Выводы ... 64

2. ФУНКЦИОНАЛЬНЫЕ ОСОБЕННОСТИ И ХАРАКТЕРИСТИКИ МЕТОДОВ ВЕРИФИКАЦИИ АВТОМАТНЫХ МОДЕЛЕЙ 65
2.1. Функциональные особенности методов верификации автоматных моделей 65
2.1.1. Метод построения модели Крипке по автоматной модели 65
2.1.2. Метод верификации визуальных автоматных моделей 65
2.1.3. Метод верификации автоматных моделей на основе эмуляции 65
2.2. Функциональные особенности методов преобразования контрпримеров 67
2.2.1. Метод верификации автоматных программ с использованеим верификатора NuSMV 67
2.2.2. Метод верификации визуальных автоматных моделей 68
2.2.3. Метод верификации на основе эмуляции 68
2.3. Характеристики методов верификации автоматных моделей 68
2.3.1. Метод верификации автоматных программ с использованием верификатора *NuSMV* 68
2.3.2. Метод верификации визуальных автоматных моделей 69
2.3.3. Метод верификации на основе эмуляции 70
2.4. Условия и ограничения функционирования методов верификации автоматных моделей 71
2.4.1. Метод построения модели Крипке по автоматной модели 71
2.4.2. Метод верификации визуальных автоматных моделей 71
2.4.3. Метод верификации на основе эмуляции 71
2.5. Выводы 73
3. ТЕХНОЛОГИЯ ВЕРИФИКАЦИИ АВТОМАТНЫХ МОДЕЛЕЙ УПРАВЛЯЮЩИХ ПРОГРАММ 74
3.1. SMVVerifier 74
3.1.1. Модель программного средства 74
3.1.2. Прототип программного средства 75
3.2. UniMod.verifier 77
3.2.1. Модель программного средства 77
3.2.2. Прототип программного средства 83
3.3. Converter 87
3.3.1. Модель программного средства 87
3.3.1.1. Общее описание инструментального средства *Converter* 87
3.3.1.2. Описание работы инструментального средства 88
3.3.1.3. Диаграмма классов инструмента 90
3.3.2. Прототип программного средства 90
3.4. Методические рекомендации и указания по верификации автоматных моделей управляющих программ 101
ЗАКЛЮЧЕНИЕ 103
ИСТОЧНИКИ 104

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).

Целями настоящего этапа является разработка эффективных методов верификации автоматных моделей управляющих программ.

Задачами этапа являются:

1. Разработка новых методов верификации автоматных моделей.
2. Определение функциональных особенностей и характеристик разработанных методов.
3. Разработка прототипов программных средств верификации автоматных моделей, основанных на предложенных методах.

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 традиционных программах обычно не разделяются управляющие и вычислительные состояния.

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

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

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.

Из изложенного следует, что при применении к автоматным программам метода *Model checking* открывается возможность автоматического преобразования системы взаимодействующих автоматов в модель Крипке. Это существенно упрощает процесс верификации рассматриваемого класса программ. Разработка методов, обеспечивающих такое упрощение, является задачей настоящего этапа работ.

Дополнительные патентные исследования, проведенные в рамках второго этапа работы (отчет о патентных исследованиях № 2007.12.15-02 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 будут соответствовать мировому уровню разработок в рассматриваемой области.

1. МЕТОДЫ ВЕРИФИКАЦИИ АВТОМАТНЫХ МОДЕЛЕЙ УПРАВЛЯЮЩИХ ПРОГРАММ

В данной главе описываются разработанные методы верификации автоматных моделей управляющих программ.

Метод проверки модели (*Model checking*) – это метод автоматизированной верификации программных систем. При использовании этого подхода строится модель с конечным числом состояний. Свойства модели выражаются на языке темпоральной логики. Модель и свойства подаются на вход программе верификатору. Верификатор автоматически проверяет свойства модели и в случае не выполнения свойства выдает контрпример, который указывает место, где нарушается свойство модели. На рис. 1 показана схема работы верификатора.

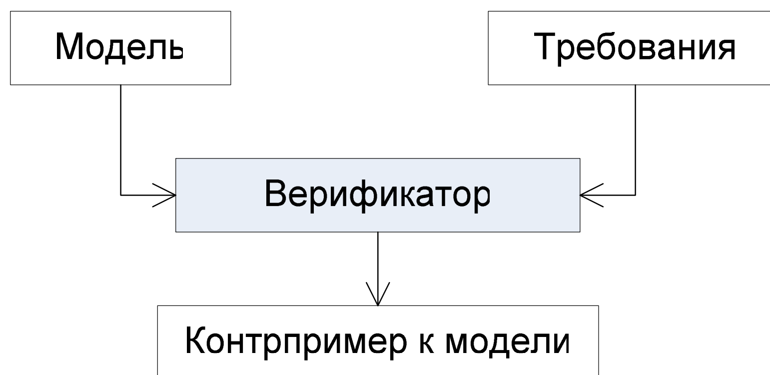

Рис. 1. Схема верификатора

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

- Написание модели вручную требует больших затрат времени.
- При построении модели могут быть допущены ошибки, что снижает эффективность проверки.

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

В разд. 1.1 предлагается метод построения модели Крипке по автоматной модели управляющей программы. Построенная модель может быть верифицирована существующими верификаторами.

Метод верификации визуальных автоматных моделей (разд. 1.2) предназначен для верификации автоматных моделей, представленных графами переходов автоматов Мили. За счет учета специфики автоматных программ этот метод позволяет строить модели Крипке с относительно небольшим числом состояний.

Метод верификации автоматных моделей на основе их эмуляции (далее – метод верификации на основе эмуляции, разд. 1.3) верифицирует автоматные программы, обходя граф достижимых состояний при помощи модифицированного алгоритма двойного поиска в глубину. Для применения этого метода строится абстрактный тип данных «Система автоматов», который поддерживает операции, необходимые для реализации алгоритма поиска в глубину.

Применение этих методов позволяет повысить уровень автоматизации при проведении верификации автоматных программ.

1.1. Метод верификации автоматной мождели с использованием верификатора NuSMV

1.1.1. Постановка задачи

Рассматривается задача проверки свойств системы, построенной на основе автоматного подхода [20]. В работе [21] описан способ проверки одного автомата. Реальные системы, как правило, слишком сложны, для того, что бы представлять их в виде одного автомата. Возникает задача проверки систем автоматов.

Предлагаемый метод проверяет систему автоматов со следующими свойствами. Каждый автомат является автоматом Мили [22]. Автоматы могут взаимодействовать следующим образом:

- Автоматы могут узнавать о состояниях других автоматов.
- Один автомат может передавать управление другому автомату, предавая ему событие.

Необходимо построить программу, которой на вход поступала бы система автоматов и требования на языке темпоральной логики, а она возвращала бы контрпример, если требования к системе нарушаются. Существует большое число программ для верификации модели. Поэтому можно не реализовать алгоритмы непосредственно для верификации, а можно воспользоваться одним из существующих средств. Таким образом, задача сводиться к следующим подзадачам:

- Преобразовать программу в модель.
- Преобразовать требования к системе в требования к модели.
- Запустить программу-верификатор.
- Преобразовать полученный контрпример к модели в контрпример в системе.

На рис. 2 изображена схема предлагаемого подхода.

Рис. 2. Предлагаемый подход

В качестве языка для описания модели будем использовать язык *SMV* [2, 23]. Этот язык применяется в символьном верификаторе моделей *SMV* (аббревиатура «*Symbolic Model Verifier*»). Это инструментальное средство предназначено для проверки того, что система переходов с конечным числом состояний удовлетворяет требованиям, заданным на языке темпоральной логики (*CTL)*. В нем применяется описанный в работе [23] символьный алгоритм верификации моделей, основанный на упорядоченных двоичных диаграммах решений (*OBDD)*. Отметим некоторые наиболее важные особенности языка *SMV*:

- Модули. Пользователь может разбить описание сложной системы на отдельные модули. Одни и те же модули могут иметь несколько экземпляров. Модули могут содержать ссылки на переменные, объявленные в других модулях. В проектируемых системах с иерархической структурой для именования переменных используются стандартные правила видимости. Допустима параметризация модулей, причем в качестве параметров могут выступать компоненты состояний, выражения или другие модули.
- Недетерминированные переходы. В моделях могут быть как детерминированные, так и недетерминированные переходы. Недетерминизм может отражать реальный выбор в действиях моделируемой системы, но может быть использован и для сокрытия некоторых подробностей при описании более абстрактной модели. Во многих языках описания аппаратуры отсутствует возможность описания недетерминизма, однако, она играет ключевую роль при построении моделей высокого уровня абстракции.
- Отношение переходов. Отношения переходов модулей могут быть заданы либо явно в терминах булевых отношений, зависящих от переменных состояния в текущий и последующий моменты времени, либо неявно в виде множества операторов параллельного присваивания. Операторы параллельного присваивания определяют значения переменных в следующем состоянии в зависимости от их значений в текущем состоянии.

Так как с 1998 года средство *SMV* не развивается, в настоящей работе для верификации используется средство *NuSMV* [24], использующее то же язык и метод верификации.

1.1.2. Система автоматов

Опишем систему взаимодействующих автоматов. Имеется набор конечных детерминированных автоматов Мили с несколькими выходными воздействиями на ребрах. У автомата задается набор событий, с которыми автомат может вызываться. У каждого события задается может ли это событие поступать от источника событий или только от автомата.

Каждый переход может содержать:

- Событие, при котором он происходит.
- Условие, при котором он происходит. В условии в качестве атомарных можно использовать входные переменные `x1`, `x2`, ... и выражения вида `Ai in sij` (оно истинно, если автомат `Ai` находится в состоянии `sij`).
- Последовательность действий. Она может содержать выходные воздействия `o.zi()` и передачу управления другим автоматам `Ai.ej()`.

На рис. 3 показан пример перехода автомата.

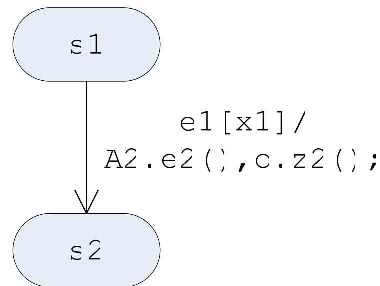

Рис. 3. Пример перехода автомата

События обрабатываются последовательно, одно событие за один раз.

В качестве входных переменных используются булевы переменные или состояния другого автомата. Переменные других типов необходимо преобразовать в булевы.

Автоматы могут взаимодействовать следующим образом:

- Автоматы могут узнавать о состояниях других автоматов. При записи условий на переходах можно задавать условия вида "автомат `Ai` находится в состоянии `sij`".
- Один автомат при переходе может передавать управление другому автомату. Это происходит следующим образом. На переходе автомата может быть написана последовательность выходных воздействий. Это либо сообщения, посылаемые наружу, либо сообщения другим автоматам. Посылка сообщения автомату – это передача управления этому автомату. Автомат, которому передали управление, совершает переход по дуге, соответствующей посланному сообщению, а затем возвращает управление. Пример такого взаимодействия показан на рис. 4.

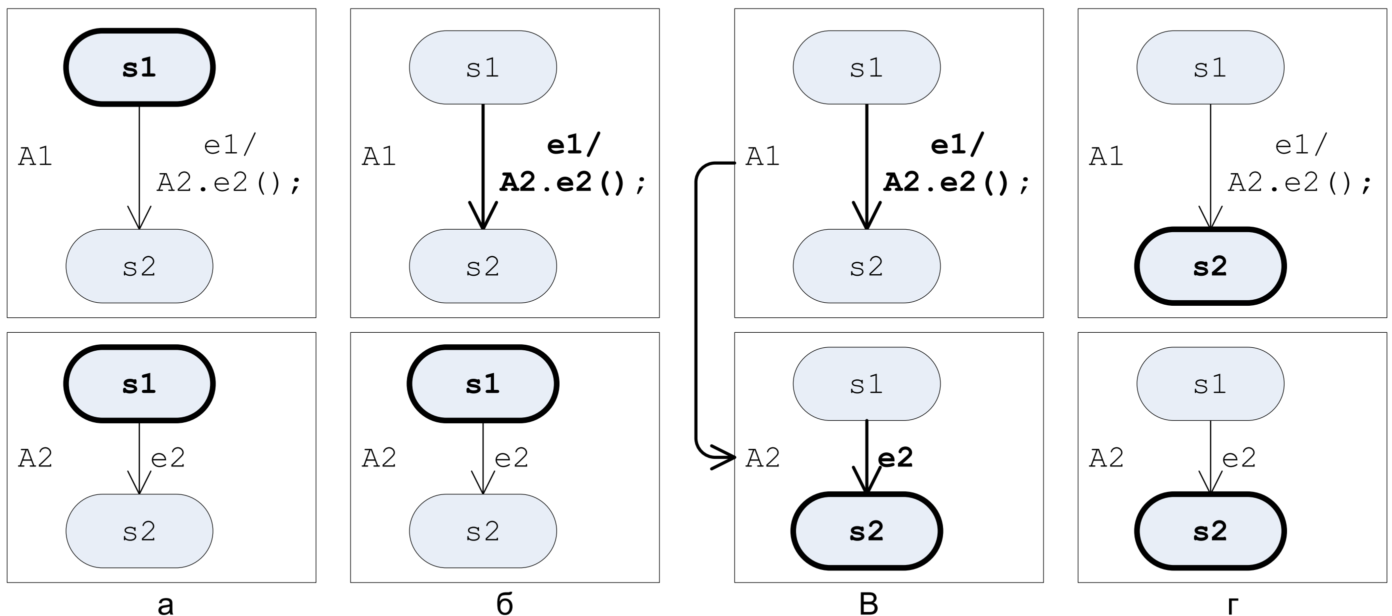

Рис. 4. Пример вызова автомата

Начальное состояние (а), автомат `A1` вызывается с событием `e1` (б), автомат `A1` вызывает автомат `A2` с событием `e2` (в), автомат `A1` переходит в состояние `s2` (г).

Автомат A1 совершает переход из состояния s1 в s2 (рис. 4, б), в процессе перехода он передает управление автомату A2 (рис. 4, в), который переходит в состояние s2. После этого управление возвращается к автомату A1 и он переходит в состояние s2 (рис. 4, г).

Описанное взаимодействие между автоматами подчиняется следующим правилам: каждый автомат в системе может вызвать любой другой автомат; автомат нельзя вызвать, если он в этот момент совершает переход. Последнее правило требует пояснения. На рис. 5 показана схема ошибочного вызова.

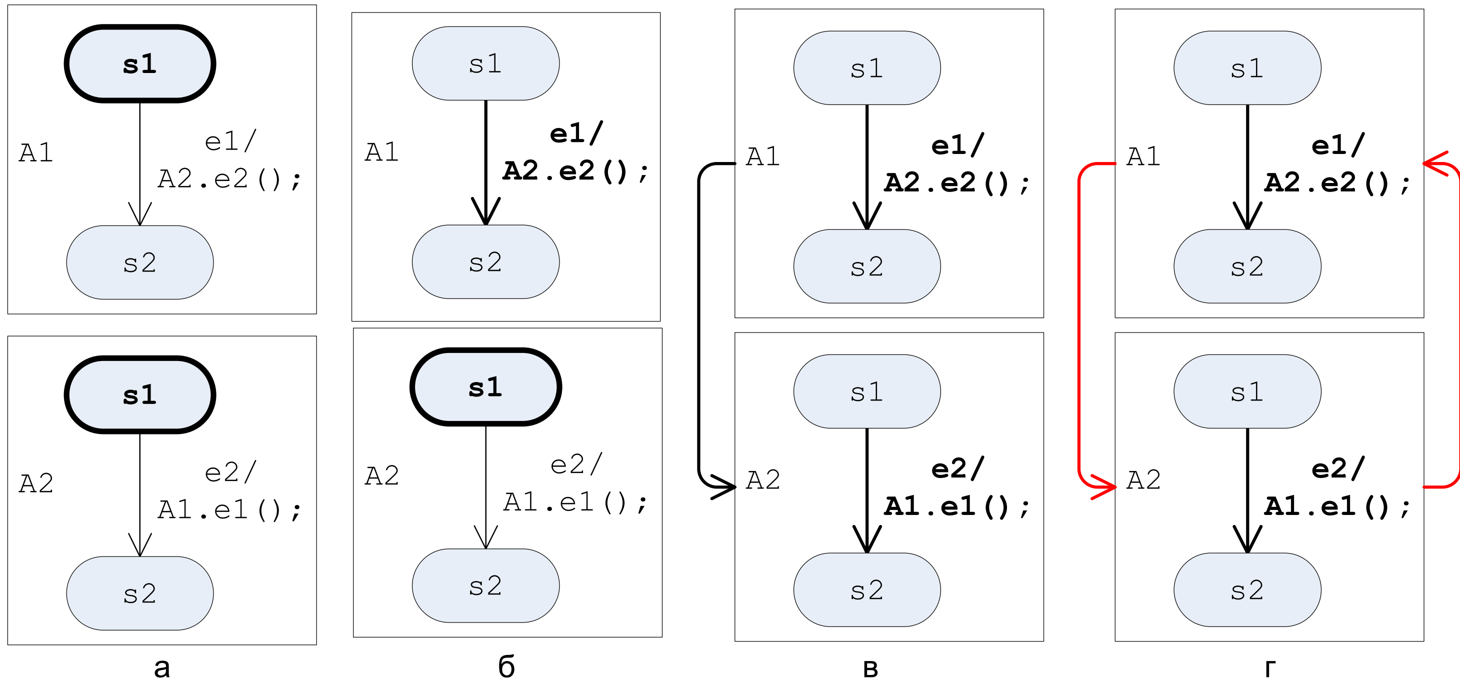

Рис. 5. Пример вызова автомата с ошибкой

Начальное состояние (а), автомат a1 вызывается с событием e1 (б), автомат a1 вызывает автомат a2 с событием e2 (в), автомат a1 вызывает a2 с событием e1(г).

Допустим, автомат A1 во время перехода вызывает автомат A2, посылая ему событие e2 (рис. 5,б). Автомату A2 передается управление, и он совершает переход по e2 (рис. 5,в). Если на этом переходе написано выходное воздействие, относящееся к автомату A1 (допустим, оно называется e1), произойдет ошибка, так как к моменту вызова e1 автомат A1 будет все еще находиться на переходе (рис. 5,г).

1.1.3. Описание метода

1.1.3.1. Общая структура метода

Построение модели выполняется в два этапа:

- Для каждого автомата строится его модель – система переходов с пометками в вершинах. Для этого вводятся дополнительные состояния, которые выражают действия, выполняемые автоматом на переходах.
- Из полученных моделей автоматов составляется модель системы, которая описывает совместное поведение автоматов.

1.1.3.2. Построение модели автомата

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

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

Рассмотрим построение модели автомата. *Состояниями модели* автомата будут состояния автомата и *промежуточные состояния*. *Промежуточные состояния* обозначаются `si.j`, где `si` – основное состояние автомата, из которого ведет переход, и на котором выделено данное промежуточное состояние, а `j` – номер *промежуточного состояния*. Состояния модели, соответствующие состояниям автомата обозначаются `si.0`. Каждое состояние модели принадлежит к одному из четырех типов:

- если автомат находится в основном состоянии, то тип состояния `inState`;
- если автомат вызывает другой автомат, то тип состояния `inEvent`;
- если автомат вызывает выходное воздействие, то тип состояния `inAction`;
- если автомат возвращает управление, то тип состояния `inReturn`.

Рассмотрим выделение промежуточных состояний на примере. На рис. 6,а изображен переход автомата из состояния `s1` в `s2`, при возникновении события `e1` при условии, что входные переменные `x1` и `x2` равны единице. В процессе перехода выполняется вызов автомата A2 с событием `e1`.

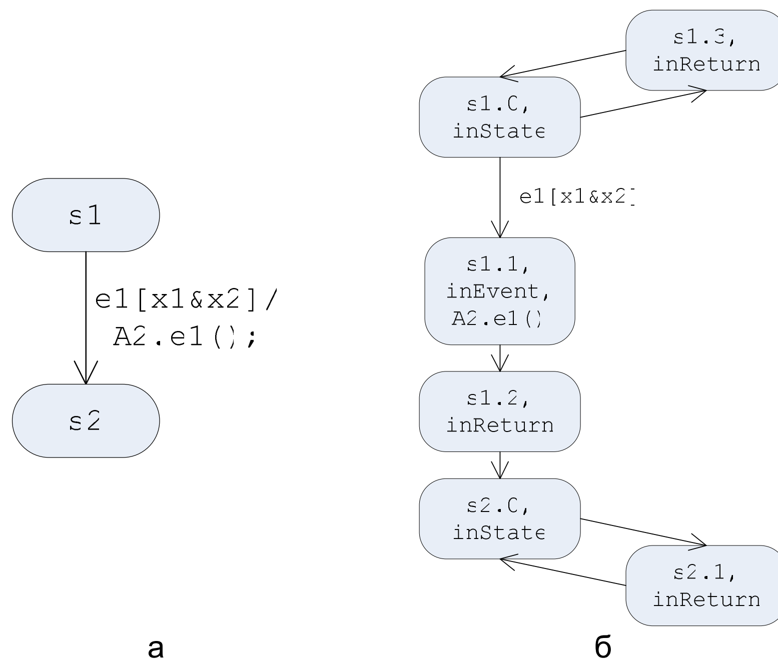

Рис. 6. Выделение промежуточных состояний на переходе:
исходный переход (а), преобразованный переход (б)

На рис. 6,б показан переход автомата из состояния s1.0 в состояние s2.0. В нем выделены следующие *промежуточные состояния*, расположенные вертикально:

- s1.0 – автомат находится в состоянии s1;
- s1.1 – в этом *состоянии модели* автомат вызывает автомат A2 с сообщением e1;
- s1.2 – переходит в основное состояние s2;
- s2.0 – автомат находится в состоянии s2;

Автомат может не удовлетворять условию полноты, поэтому возможно, что никакое из условий на переходах не выполнится и автомат остается в текущем состоянии. Для моделирования этой ситуации введем для каждого состояния автомата *промежуточное состояние,* которое ведет в него и принадлежит к типу inReturn. На рис. 6,б такими состояниями являются s1.3 и s2.1.

1.1.3.3. Преобразование автоматной системы в модель Крипке

Опишем, как будет представляться модель системы автоматов. Каждому автомату Ak сопоставим переменные yk $(0 \le k \le n)$, Operation_k, OpType_k, xk $(0 \le k < m)$, где m – число входных переменных. Для всей системы автоматов в целом введем переменные Active_k $(0 \le k \le n)$, Event, где n – число автоматов. Веденные переменные имеют следующий смысл:

- 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 y0, y1, ...yk содержат основные состояния, в которых могут находиться каждый из (k+1) автоматов. Каждая переменная может принимать значения из множества состояний соответствующего автомата.
- Переменные x1, x2, xm ... соответствуют входным переменным. Каждая переменная может принимать значения ноль (ложь) или один (истина). Если в автомате используются переменные иных типов, то необходимо преобразовать их в булевы переменные.
- Переменные Active_0, Active_1, Active_2 ..., содержат последовательность, в которой вызывались автоматы. Active_0 – содержит номер активного в данный момент автомата. Active_1 – содержит номер вызвавшего его автомата и т. д. Переменные Active_k могут содержать либо номер автомата, либо ноль. Они образуют стек вызовов системы автоматов. Глубина этого стека зависит от конкретной системы, но так как рекурсивные вызовы в автоматах невозможны, то глубина стека всегда будет меньше, чем число автоматов.
- Переменная Operation_k содержит номер *промежуточного состояния*, в котором находится автомат Ak. *Промежуточные состояния* обозначаются как si.j. Так как переменная yk уже хранит состояние si, остается сохранить только номер j, который хранит переменная Operation_k. Если автомат находится в основном состоянии, то переменная Operation_k = 0, в противном случае присваивается номер соответствующего промежуточного состояния. Таким образом, состояние модели автомата полностью определяется парой (yk, Operation_k). Диапазон значений переменной Operation_k определяется при добавлении дополнительных состояний к автомату.

- Переменная OpType_k содержит тип текущего *состояния модели* соответствующего автомата. Она может принимать значения из множества {inState, inEvent, inAction, inReturn}:
  - если автомат находится в основном состоянии, то OpType_k = inState;
  - если автомат вызывает другой автомат, то OpType_k = inEvent;
  - если автомат вызывает выходное воздействие, то OpType_k = inAction;
  - если автомат возвращает управление, то OpType_k = inReturn.

Каждое состояние модели представляет собой объединение переменных yk(0 ≤ k ≤ n), Operation_k, Active_k (0 ≤ k ≤ n), Event. Указанные переменные и переменную OpType_k можно непосредственно не хранить, а их можно выразить через переменные yk(0 ≤ k ≤ n), Operation_k.

Так как состояние задается набором переменных, то можно не строить модель Крипке непосредственно, а лишь указать правила переходов. Таким образом, можно проверять модели с большим числом состояний.

Для того чтобы задать модель Крипке, требуется указать начальное состояние и правила перехода в модели. Зададим начальное значение всех переменных:

- переменные yk для всех k содержат начальные состояния всех автоматов;
- переменные Active_k, Operation_k – содержат ноль;
- переменная Event – содержит ноль;
- переменные x1, x2, ... xm – содержат входные переменные, которые могут принимать значения ноль или единица.

Теперь зададим правила переходов. Для этого требуется определить следующие значения переменных через текущие значения. Для переменной a следующее значение обозначается как next(a).

Если Active_0 = 0 никакой автомат не выполняется. Необходимо выбрать следующее событие Event и набор входных переменных: x1,.x2, ..., xn. Переменную next(Event) недетерминировано выбираем из множества событий, которые могут произойти. Переменную next(Active_0) недетерминировано выбираем из множества номеров всех автоматов.

Если Active_0 = k и Operation_k = 0, то автомат Ak активен и находится в основном состоянии. Переменная Event содержит событие, с которым был вызван автомат. В этом состоянии требуется определить, какой переход должен быть выполнен. Для всех переходов из данного состояния проверим:

- Соответствует ли событие, при котором происходит переход, значению переменной Event.
- Выполняется ли условие перехода. При этом в формуле, описывающей условие, выражения вида Ak in skj заменим условиями вида yk = skj.

Если какой-либо переход соответствует этим условиям, то он выполняется. Для этого в переменную next(Operation_k) записывается промежуточное состояние, которое соответствует началу перехода. Если никакой переход не выполняется, то автомат остается в текущем состоянии и возвращает управление. Для этого в переменную next(Operation_k) записывается *состояние модели*, которое вернет управление вызвавшему автомату.

Если значение переменной Operation_k не равно нулю, то автомат выполняет какой-то переход. В текущем *промежуточном состоянии* могут выполняться следующие действия:

- вызов другого автомата;

- возвращение управления родительскому автомату и изменение основного состояния автомата Ak;

Рассмотрим каждое из этих действий более подробно. Когда автомат передает управление другому автомату, требуется выполнить следующие действия:

- Установить значение переменной next(Operation_k) в следующее состояние, для того чтобы, текущий автомат, получив управление, мог продолжить выполнение.
- В переменную Event записать событие, которое передается автомату.
- Для того чтобы автомат мог вернуть управление, необходимо сохранить текущее состояние системы в стеке. Стек образуется набором переменным Active_0, Active_1, … Здесь переменная Active_0 – верхушка стека, а переменная Active_n – его дно. Требуется для всех 0 < k <= n выполнить присваивание next(Active_k) := Active_(k-1);
- Для того чтобы передать управление автомату Ai, требуется записать в переменную next(Active_0) номер автомата i, которому передается управление.

Когда автомат возвращает управление вызвавшему его автомату, требуется:

- Достать из стека номер вызвавшего автомата. Для этого необходимо для всех 0<k<=n выполнить присваивание: next(Active_k-1) := Active_k. Значение переменной next(Active_n) будет равно нулю.
- Перейти в новое состояние. Для этого в переменную next(Operation_k) запишем ноль. В переменную next(yk) запишем соответствующее основное состояние.

Осталось определить значение переменных OpType_k. Их значения определяются в зависимости от того, к какому типу принадлежит состояние модели, заданное парой (yk, Operation_k).

1.1.3.4. Требования — формулы темпоральной логики на модели Крипке

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

Введем формулы, которые будут описывать состояния автоматов:

Для записи условия, что автомат Ak находится в состоянии sj недостаточно записать yk = sj, так же требуется указать, что состояние модели соответствует моменту, когда автомат находится в основном состоянии. Это свойство описыается Ak.inState. Тогда общая формула имеет вид Ak.inState & yk = sj.

Условие того, что выполнилось выходное воздействие z1, записывается как Action = z1.

Для записи условия, что произошло событие ei, достаточно записать Event = ei.

Для записи формул, описывающих состояния автомата, также 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, EF, AG, EG, A[U], E[U]. Операторы EX, AX не используются для записи свойств автоматов, так как в данной модели один переход автомата соответствует неопределенному числу переходов модели. Опишем эти операторы:

- AF f (*Future*) — оператор будущего. 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: f будет выполняться в каждом состоянии, достижимом из текущего состояния.
- A[f U g] (*Until*) — оператор истинен, только если на каждом пути когда-нибудь выполнится формула *g*, а до этого момента всегда будет выполняться формула f.

Операторы EF, EG, E[U] аналогичны предыдущим, только вместо выполнения условия на каждом пути требуется существования пути, на котором выполнится условие.

Использование логики *CTL* позволяет строить достаточно сложные условия, но имеет один серьезный недостаток. У некоторых формул из логики *CTL*, построенных по такой схеме, может не существовать контрпримера. Например, для формулы EF f нельзя построить контрпример, так как контрпримером для нее служат все состояния, достижимые из начального состояния с указанием того, что в них не выполняется f.

Существует более узкая логика *ACTL*. Она обладает свойством, что для каждой её формулы можно построить простой контрпример. Множество формул логики *ACTL* является подмножеством *CTL*, в котором разрешается использовать только операторы AX, AF, AG, A[U]. Причем отрицание можно использовать только под темпоральными операторами.

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

Если опровергаемая формула принадлежит *ACTL*, то получим контрпример в структуре Крипке. Любой контрпример для модели является либо конечным путем, либо путем с конечным началом и циклом. На рис. 7 показан общий вид контрпримера.

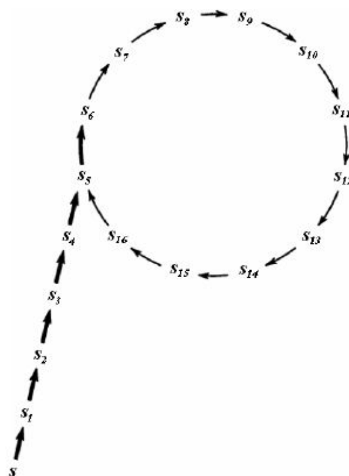

Рис. 7. Общий вид контрпримера

Каждое состояние является набором переменных `yk`, `Operation_k`, `OpType_k`, `Active_k (0 ≤ k ≤ n)`, `xk(0 ≤ k < m)`. Предлагается следующий алгоритм для преобразования контрпримера к модели Крипке в контрпример к системе автоматов. Контрпример к системе автоматов также представляется в виде последовательности состояний, только вместо значений переменных информация предоставляется в терминах состояний и переходов. Для каждого состояния контрпримера выведем информацию об автомате. При этом `yk`, `Operation_k`, `OpType_k`, `Active_k`, `xk` означают значения соответствующих переменных на текущем шаге:

- Если `Active_0 = 0`, то никакой автомат не активен. Выводятся состояния всех автоматов. Для всех `k` выводится имя автомата с номером `k` и значения переменной `yk`.
- Если `Event != 0`, то текущему автомату пересылается какое-то событие. Выводится его название.
- Если `Active_0 != 0`, то выводится название активного автомата. Его номер записан в переменную `Active_0`.
- Если `Operation_k != 0`, то текущий автомат находится на переходе. По паре `(yk, Oparation_k)` определяется текущее *состояние модели*, а по нему определяется на каком переходе находится автомат. Выводится этот переход. Также выводится тип текущего *состояния модели*.
- Если в текущем *состоянии модели* вызывается другой автомат, то выводится название этого автомата и событие, которое ему передается.

1.1.4. Применение верификатора *NuSMV*

1.1.4.1. Описание языка *SMV*

Для верификации модели системы автоматов используется средство *NuSMV*, которое принимает на вход модель на языке *SMV*. Модель на языке *SMV* содержит набор перечислимых переменных и правила переходов. Эта модель может быть разделена на модули. Каждый модуль может содержать в себе набор переменных, правила переходов и требования. Одни и те же модули, определенные один раз, можно использовать несколько раз, если это необходимо. Допустима параметризация модулей, причем в качестве параметров могут выступать компоненты состояний, выражения или другие модули. При этом каждый экземпляр модуля может содержать ссылки на переменные, объявленные в других модулях. Модули могут содержать в себе экземпляры других модулей.

В модуле может содержаться описание начального состояния системы и правила, по которым осуществляются переходы. Правила переходов записываются в виде:

новое значение переменной := некоторая формула от старых и новых значений.

Для записи модели требуется непосредственно определить правила, по которым значения переменных в новом состоянии будет выражаться через переменные в текущем состоянии и уже определенные переменные в новом состоянии.

Приведем описание синтаксиса входного файла *SMV*.

Описание модели разделено на модули. Каждый модуль начинается с заголовка `MODULE «имя модуля»`. В модели обязательно должен был модуль `main`. Каждый модуль может иметь секции:

- `VAR` – в данной секции описываются переменные. В ней указываются имя и тип переменных в виде: `«имя переменной»: «тип переменной»;`.

- ASSIGN – в данной секции задаются начальные значения переменных и правила перехода. Выражение init(a):= выражение; задает начальное значение переменной a. Выражение next(a):= выражение; задает следующее значение переменной a.
- DEFINE – в данной секции описываются определения. Определение задается следующим образом: «имя»:= «выражение»;. Определения можно использовать как обычные переменные, только для них не выделяется память в модели, а подставляется выражение.

Приведем пример простой модели на языке *SMV*:

```
MODULE main
VAR
  request : boolean;
  state : ready, busy;
ASSIGN
  init(state) := ready;
  next(state) := case
    state = ready & request = 1 : busy;
    1 : ready, busy;
  esac;
```

Модель состоит из одного модуля main. В секции VAR определены переменные request типа boolean и state перечислимого типа. Переменная request может принимать значения ноль или единица. Переменная state может принимать значения ready или busy. В секции ASSIGN описываются начальные значения переменных и правила перехода. Переменной state задается начальное значение ready. Следующее значение переменной state определяется по следующему правилу: если переменная state имеет значение ready и переменная request равна единице, то переменная state принимает на следующем шаге значение busy, иначе переменная state недетерминировано принимает значения ready или busy.

1.1.4.2. Построение по системе автоматов модели на языке *SMV*

Каждый автомат разместим в отельном модуле. Каждый модуль будет иметь следующие параметры:

- Active — является ли данный модуль активным;
- Event — входящее событие;
- yk — состояния экземпляров автоматов, с которыми данный модуль взаимодействует.

Каждый модуль будет иметь следующие переменные:

- Переменная State содержит номер состояния автомата, и заменяет переменную yk. Она описывается State: {s1, s2, …, sn};
- Переменная Operation принимает целые значения из некоторого интервала, она описывается Operation: 0..k;
- Переменные xk описываются xk: {0, 1};

Далее требуется указать правила, по которым будут изменяться значения переменных.

Правило для нахождения значения переменной next(State): если Active и (State, Operation) соответствуют *состоянию модели,* в котором возвращается управление, то значение переменной next(State) равно соответствующему основному состоянию автомата, иначе значение переменной next(State) не изменяется.

Опишем правила для нахождения значений переменной next(Operation):

- Если Active = 1 и Operation = 0, то автомат находится в основном состоянии. Проверим значения переменных Event и x1, x2, … на соответствие какому-либо переходу и запишем в переменную Operation соответствующее значение. Если никакой переход не выполнился, запишем в переменную Operation номер состояния, которое вернет управление вызвавшему автомату;
- Если Active = 1, Operation != 0 и Operation – не возвращает значения автомату, то запишем в переменную next(Operation) номер следующего элементарного состояния;
- Если Active и Operation – возвращает значение автомату, то присвоим next(Operation) := 0, иначе значение переменной next(Operation) равно текущему значению.

Вместо переменной OpType_k будем использовать четыре булевых переменных: inState, inEvent, inAction, inReturn. Эти переменные будут принимать значение равное единице, если текущее *состояние модели* имеет соответствующий тип. Значения этих переменных выражаются через переменные Operation и State следующим образом:

- inState := (Operation = 0);
- inReturn описывается набором скобок разделенных оператором |, в каждой скобке записывается условие (State = si & Operation = j), где (si, j) – *состояние модели* типа inReturn;
- inEvent записывается аналогично inReturn;
- inOperation := !inState & !inReturn & 1!inEvent;

Переменные, общие для всей системы, разместим в модуле main. Там же разместим экземпляры всех модулей, описывающих автоматы. Опишем переменные модуля main:

- yk хранит экземпляр модуля, описывающий автомат k;
- Переменная Active_k хранит номер активного автомата или 0. Она описывается Active_k: 0..n; где n – число автоматов в системе;
- Переменная Event содержит сообщение, передаваемое автомату. Она описывается следующим образом Event: {0, e1, e2, e3, …};

Зададим значения переменных. Начальные значения переменных уже описаны выше. Зададим правила для переходов.

Правила для нахождения значений переменной next(Event): если Active_0 = 0, то необходимо выбрать значение Event из множества событий, которые могут поступать от источника событий, иначе требуется для всех автоматов перебрать все элементарные состояния, в которых выполняется вызов другого автомата, и записать условие: если Active_0 = Ak и yk = skj и Operation_k = okj, то next(Event) := em, где k – номер автомата, а em – событие, которое передается автомату в состоянии skj.okj.

Правила для нахождения значений переменной next(Active_k):

- Если Active_0 = 0, то номер автомата выбирается недетерминировано из всех возможных номеров автоматов.
- Если Active_0 = Ak и OpType_k = inReturn, то автомат Ak возвращает управление вызвавшему его автомату. Номер автомата, которому требуется вернуть управление, содержится в стеке, реализованном на Active_0, Active_1… . Поэтому для того чтобы передать управление, достаточно извлечь значение из стека. Это можно

выполнить следующим способом. Для `k < n`: `Active_k := Active_(k+1)`, а для `k = n` присвоим `Active_k := 0`.

- Если `Active_0 = Ak` и `OpType_k = inEvent`, то автомат `Ak` передает управление другому автомату. Требуется положить в стек номер автомата, которому передается управление. Присвоим `Active_0` номер автомата, которому передается управление. Этот номер можно определить из текущего элементарного состояния. Так же необходимо сместить весь стек, для этого для `k > 0` присваивается переменной `Active_k := Active_(k-1)`.

1.1.5. Применение метода верификации автоматных программ и использованием верификатора NuSMV

1.1.5.1. Описание примера

Используем предлагаемый метод на примере лифта. Поведение лифта описывается двумя автоматами `doors` и `main`, которые будут управлять дверьми лифта и его движением соответственно.

На рис. 8 показан автомат `doors`.

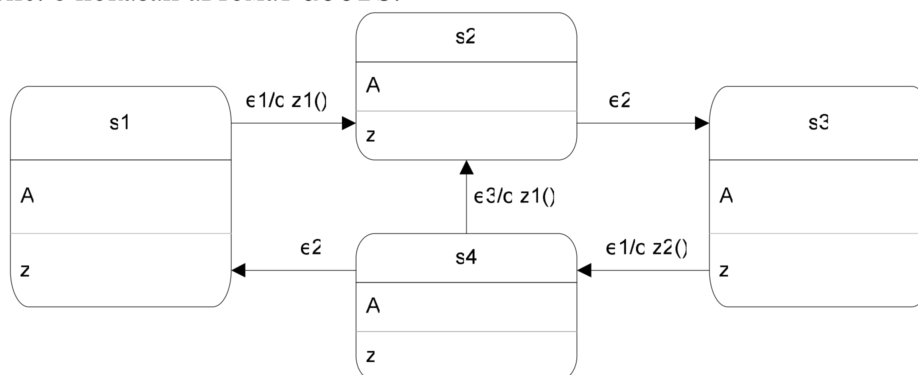

Рис. 8. Модель автомата, управляющего дверьми лифта `doors`

Автомат `doors` может находиться в следующих состояниях:

- `s1` – «Двери закрыты»;
- `s2` – «Двери открываются»;
- `s3` – «Двери открыты»;
- `s4` – «Двери закрываются».

В этом автомате использованы следующие обозначения:

- `e1` – событие, происходящее при открытии или закрытии дверей;
- `e2` – двери открылись или закрылись;
- `e3` – дверям что-то мешает закрыться;
- `z1` – открыть двери;
- `z2` – закрыть двери;

В начале двери лифта закрыты. Автомат находится в состоянии `s1`. Когда поступает событие `e1` «Открыть двери», формируется выходное воздействие z1, которое отрывает двери и автомат переходит в состояние `s2` «Двери открываются». Когда двери откроются, автомату посылается событие `e2`, означающее что двери открылись. В результате автомат переходит в состояние `s3`

«Двери открыты». Из этого состояния автомат может перейти в состояние `s4` «Двери закрываются». Из состояния `s4` можно перейти либо по событию `e2` в состояние `s1`, либо в состояние `s2`, если возникнет событие e3, означающее, что дверям, что-то мешает закрыться.

Теперь опишем автомат, управляющий движением лифта. На рис. 9 приведена диаграмма переходов автомата `main`.

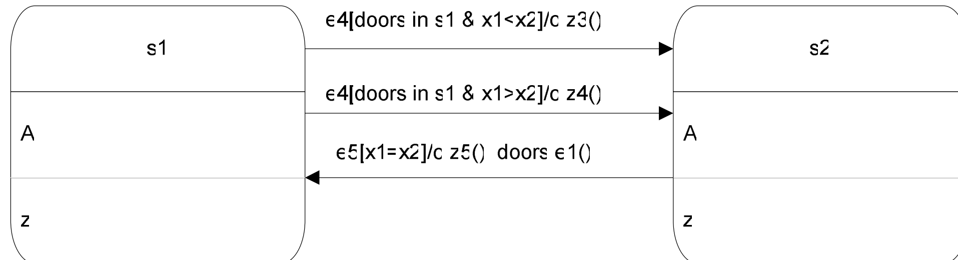

Рис. 9. Модель автомата, управляющего движением лифта main

Автомат `main` имеет следующие состояния:

- `s1` – «Лифт стоит на месте»;
- `s2` – «Лифт движется».

В автомате `main` использованы следующие обозначения:

- `doors.e1()` – вызов автомата `doors` с событием `e1`;
- `doors in s1` – условие, которое выполняется, если автомат `doors` находится в состоянии `s1`;
- `e4` – событие, происходящее когда нажата кнопка этажа назначения лифта;
- `e5` – событие, происходящее когда лифт достигает очередного этажа;
- `x1` – входное воздействие, возвращающее номер этажа нажатой кнопки;
- `x2` – входное воздействие, возвращающее текущий номер этажа, на котором находится лифт;
- `z3` – выходное воздействие, начинающее движение лифта вниз;
- `z4` – выходное воздействие, начинающее движение лифта вверх;
- `z5` – выходное воздействие, останавливающее движение лифта.

Сначала лифт стоит на месте и автомат `main` находится в состоянии `s1`. Для того чтобы лифт начал движение требуется, чтобы произошло событие `e4` «Вызов лифта», и выполнилось условие: двери закрыты и вызываемый этаж не равен текущему этажу. После этого вызывает соответствующее выходное воздействие `z3` или `z4`, и автомат переходит в состояние `s2`. Из состояния `s2` автомат может перейти в состояние `s1`, когда лифт достигнет очередного этажа, и текущий этаж равен этажу, который вызывали.

Данная система управления лифтом является сильно упрощенной. В ней не учитывается возможность поломки лифта или возможность указывать несколько этажей. Однако даже ее модель получается достаточно большой. Этой системы вполне достаточно для демонстрации проверки свойств.

1.1.5.2. Построение модели

Введем дополнительные состояния в модель автомата `doors` управления дверьми лифта (рис. 10).

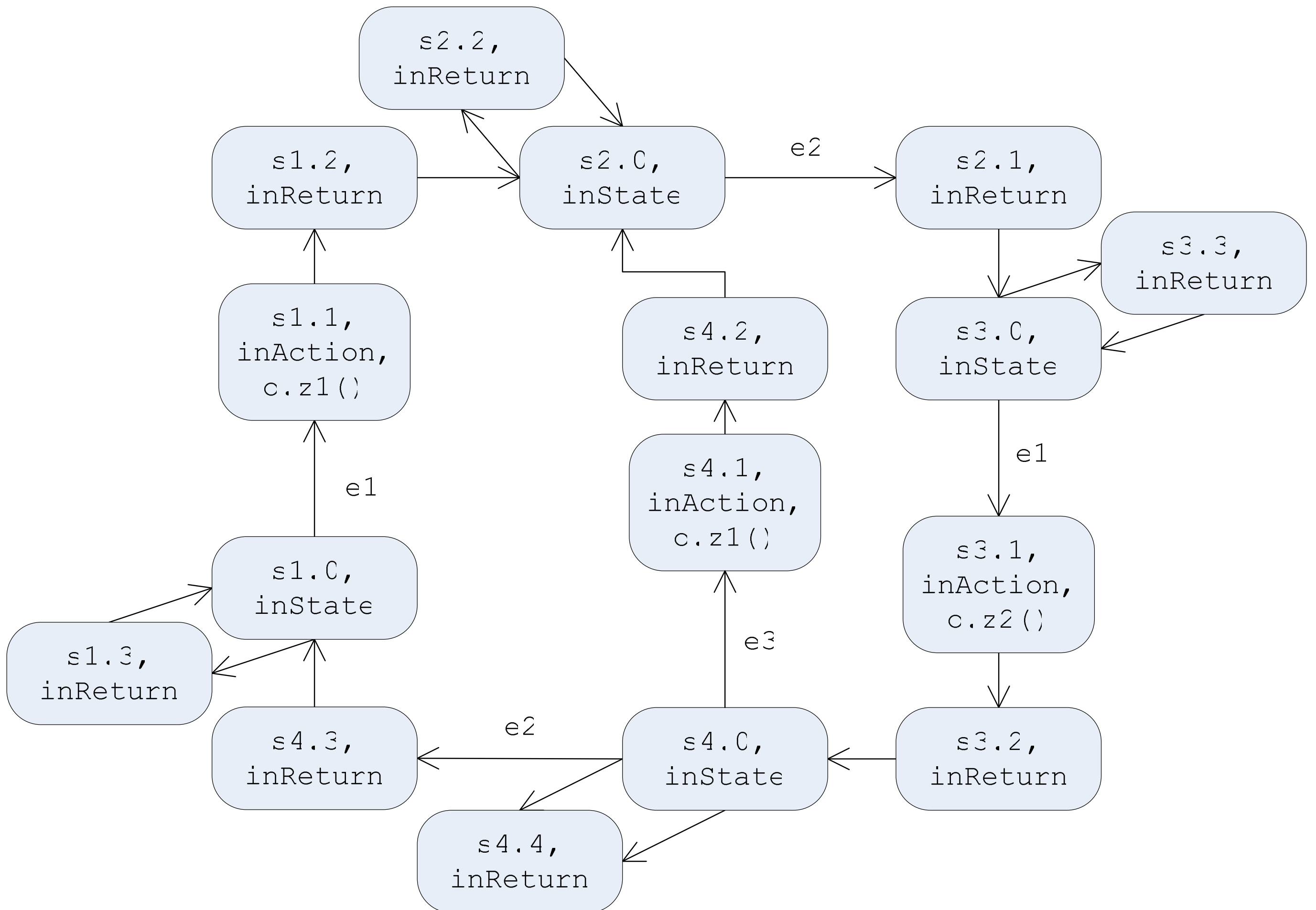

Рис. 10. Модель автомата, управляющего движением лифта `doors`

В вершинах графа указываются основное и промежуточное состояния, а также какому действию соответствует данное промежуточное состояние. Аналогично добавим промежуточные состояния к автомату `main`. В этом автомате используются переменные `x1` и `x2`, которые не являются булевыми. Для построения модели используем следующую абстракцию, введем две переменные булевых переменных `x1 = x1 < x2` и `x2 = x1 > x2`. Выражение `x1 = x2` заменим `x1`. Это можно сделать, так как условие `x1 < x2` никогда не будет проверяться вместе с условием `x1 = x2`. Модель автомата `main` приведена на рис. 11.

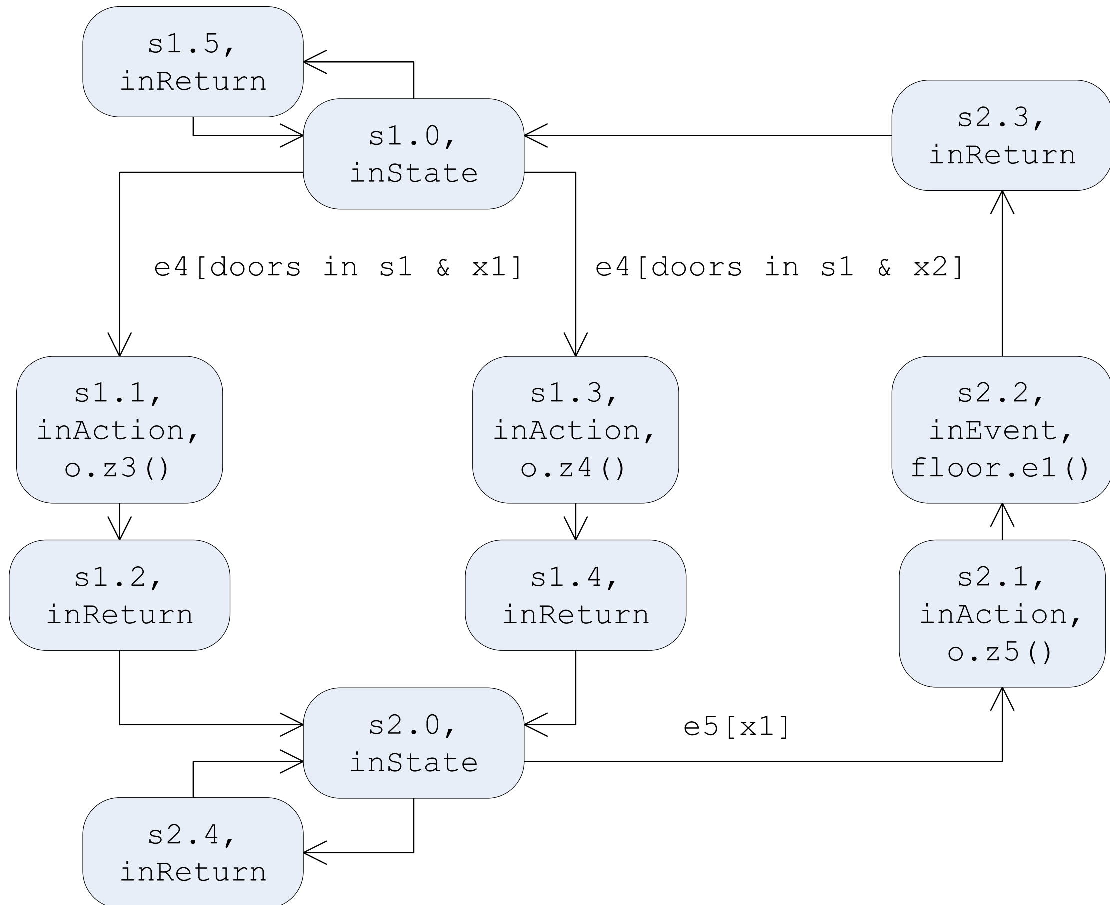

Рис. 11. Модель автомата, управляющего движением лифта `main`

Дополнительные состояния `s1.5`, `s2.4` добавлены, так как что автомат не является полным. В эти состояния модель будет переходить, если переменные не будут удовлетворять никакому условию на дуге.

1.1.5.3. Построение модели на языке *SMV*

Преобразуем автомат управления дверями лифта в модель на языке *SMV*.

У модуля есть два параметра: `Active, Event`.

```
MODULE DoorsController(Active, Event)
```

Далее следует секция описания переменных:

```
VAR
  State: {s1, s2, s3, s4};
  Operation: 0..4;
```

По рис. 10 описываются правила переходов в модели.

```
ASSIGN
  init(State) := s1;
  next(State) := case
    Active & State = s1 & Operation = 2: s2;
    Active & State = s1 & Operation = 3: s1;
```

```
      Active & State = s2 & Operation = 1: s3;
      Active & State = s2 & Operation = 2: s2;
      Active & State = s3 & Operation = 2: s4;
      Active & State = s3 & Operation = 3: s3;
      Active & State = s4 & Operation = 1: s1;
      Active & State = s4 & Operation = 3: s2;
      Active & State = s4 & Operation = 4: s4;
      1: State;
    esac;
    init(Operation) := 0;
    next(Operation) := case
      Active & State = s1 & Operation = 0 & Event = e1: 1;
      Active & State = s2 & Operation = 0 & Event = e2: 1;
      Active & State = s3 & Operation = 0 & Event = e1: 1;
      Active & State = s4 & Operation = 0 & Event = e2: 1;
      Active & State = s4 & Operation = 0 & Event = e3: 2;
      Active & State = s1 & Operation = 0: 3;
      Active & State = s2 & Operation = 0: 2;
      Active & State = s3 & Operation = 0: 3;
      Active & State = s4 & Operation = 0: 4;
      Active & State = s1 & Operation = 1: 2;
      Active & State = s1 & Operation = 2: 0;
      Active & State = s1 & Operation = 3: 0;
      Active & State = s2 & Operation = 1: 0;
      Active & State = s2 & Operation = 2: 0;
      Active & State = s3 & Operation = 1: 2;
      Active & State = s3 & Operation = 2: 0;
      Active & State = s3 & Operation = 3: 0;
      Active & State = s4 & Operation = 1: 0;
      Active & State = s4 & Operation = 2: 3;
      Active & State = s4 & Operation = 3: 0;
      Active & State = s4 & Operation = 4: 0;
      1: Operation;
    esac;
```

Определяются типы операций:

```
  DEFINE
    inState := (Operation = 0);
    inReturn :=
      (State = s1 & Operation = 2) |
      (State = s1 & Operation = 3) |
      (State = s2 & Operation = 1) |
      (State = s2 & Operation = 2) |
      (State = s3 & Operation = 2) |
      (State = s3 & Operation = 3) |
      (State = s4 & Operation = 1) |
      (State = s4 & Operation = 3) |
      (State = s4 & Operation = 4) |
      0;
```

```
  inEvent :=
    0;
  inAction := !inState & !inReturn & !inEvent;
```

Перейдем к описанию автомата, управляющего движением лифта `main`.

```
MODULE LiftController(Active, Event, doors)
VAR
  State: {s1, s2};
  Operation: 0..5;
  x1: {0,1};
  x2: {0,1};
ASSIGN
  init(State) := s1;
  next(State) := case
    Active & State = s1 & Operation = 2: s2;
    Active & State = s1 & Operation = 4: s2;
    Active & State = s1 & Operation = 5: s1;
    Active & State = s2 & Operation = 3: s1;
    Active & State = s2 & Operation = 4: s2;
    1: State;
  esac;
  init(Operation) := 0;
  next(Operation) := case
    Active & State = s1 & Operation = 0 & Event = e4 & (x1 &
        doors = s1): 1;
    Active & State = s1 & Operation = 0 & Event = e4 & (x2 &
        doors = s1): 3;
    Active & State = s2 & Operation = 0 & Event = e5 & x1: 1;
    Active & State = s1 & Operation = 0: 5;
    Active & State = s2 & Operation = 0: 4;
    Active & State = s1 & Operation = 1: 2;
    Active & State = s1 & Operation = 2: 0;
    Active & State = s1 & Operation = 3: 4;
    Active & State = s1 & Operation = 4: 0;
    Active & State = s1 & Operation = 5: 0;
    Active & State = s2 & Operation = 1: 2;
    Active & State = s2 & Operation = 2: 3;
    Active & State = s2 & Operation = 3: 0;
    Active & State = s2 & Operation = 4: 0;
    1: Operation;
  esac;
DEFINE
  inState := (Operation = 0);
  inReturn :=
    (State = s1 & Operation = 2) |
    (State = s1 & Operation = 4) |
    (State = s1 & Operation = 5) |
    (State = s2 & Operation = 3) |
    (State = s2 & Operation = 4) |
```

```
      0;
    inEvent :=
      (State = s2 & Operation = 2) |
      0;
    inAction := !inState & !inReturn & !inEvent;
```

Перейдем к описанию главного модуля модели.

```
MODULE main()
VAR
  Event: {0, e4, e2, e5, e1, e3};
  y1: DoorsController(Active_0 = 1, Event);
  y2: LiftController(Active_0 = 2, Event, y1.State);
  Active_0: 0..2;
  Active_1: 0..2;
ASSIGN
  init(Active_0) := 0;
  next(Active_0) := case
    Active_0 = 0: 1..2;
    Active_0 = 2 & y2.State = s2 & y2.Operation = 2: 1;
    Active_0 = 1 & y1.inReturn: Active_1;
    Active_0 = 2 & y2.inReturn: Active_1;
    1: Active_0;
  esac;
  init(Active_1) := 0;
  next(Active_1) := case
    Active_0 = 1 & y1.inReturn: 0;
    Active_0 = 2 & y2.inReturn: 0;
    Active_0 = 1 & y1.inEvent: Active_0;
    Active_0 = 2 & y2.inEvent: Active_0;
    1: Active_1;
  esac;
  init(Event) := 0;
  next(Event) := case
    Active_0 = 0: {e4, e2, e5, e3};
    Active_0 = 2 & y2.State = s2 & y2.Operation = 2: e1;
    1: 0;
  esac;
```

1.1.5.4. Проверка требований к модели

Проверим следующие требования к модели:

- Всегда ли, когда лифт находится в движении, его двери закрыты?
- Всегда ли автомат `main` может перейти в состояние `s2`?

Запишем эти требования в формулах темпоральной логики.

- Первое требование можно записать, как если автомат `main` находится в состоянии `s2`, то автомат `main.doors` находится в состоянии `s1`. В формуле темпоральной логики это выражается следующим образом `AG((State = s2 & Operation = 0) -> doors = s1);`
- Второе требование можно записать как `AG(AF(State = s2));`

При проверке второго свойства программа нашла контрпример.

```
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  Event = 0
  y1.State = s1
  y1.Operation = 0
  y2.State = s1
  y2.Operation = 0
  y2.x1 = 0
  y2.x2 = 0
  Active_0 = 0
  Active_1 = 0
  y1.inAction = 0
  y1.inEvent = 0
  y1.inReturn = 0
  y1.inState = 1
  y2.inAction = 0
  y2.inEvent = 0
  y2.inReturn = 0
  y2.inState = 1
-> Input: 1.2 <-
-> State: 1.2 <-
  Event = e4
  y2.x1 = 1
  Active_0 = 2
-> Input: 1.3 <-
-> State: 1.3 <-
  Event = 0
  y2.Operation = 1
  y2.x1 = 0
  y2.inAction = 1
  y2.inState = 0
-> Input: 1.4 <-
-> State: 1.4 <-
  y2.Operation = 2
  y2.inAction = 0
  y2.inReturn = 1
-> Input: 1.5 <-
-> State: 1.5 <-
  y2.State = s2
  y2.Operation = 0
  Active_0 = 0
  y2.inReturn = 0
  y2.inState = 1
-> Input: 1.6 <-
-> State: 1.6 <-
  Event = e5
```

```
    y2.x1 = 1
    Active_0 = 2
  -> Input: 1.7 <-
  -> State: 1.7 <-
    Event = 0
    y2.Operation = 1
    y2.x1 = 0
    y2.inAction = 1
    y2.inState = 0
  -> Input: 1.8 <-
  -> State: 1.8 <-
    y2.Operation = 2
    y2.inAction = 0
    y2.inEvent = 1
  -> Input: 1.9 <-
  -> State: 1.9 <-
    Event = e1
    y2.Operation = 3
    Active_0 = 1
    Active_1 = 2
    y2.inEvent = 0
    y2.inReturn = 1
  -> Input: 1.10 <-
  -> State: 1.10 <-
    Event = 0
    y1.Operation = 1
    y1.inAction = 1
    y1.inState = 0
  -> Input: 1.11 <-
  -> State: 1.11 <-
    y1.Operation = 2
    y1.inAction = 0
    y1.inReturn = 1
  -> Input: 1.12 <-
  -> State: 1.12 <-
    y1.State = s2
    y1.Operation = 0
    Active_0 = 2
    Active_1 = 0
    y1.inReturn = 0
    y1.inState = 1
  -> Input: 1.13 <-
  -> State: 1.13 <-
    y2.State = s1
    y2.Operation = 0
    Active_0 = 0
    y2.inReturn = 0
    y2.inState = 1
```

После преобразования его в контрпример к системе автоматов, получается следующая последовательность состояний.

```
Шаг: 1
  Автомат doors в состоянии s1
  Автомат main в состоянии s1
Шаг: 2
  Получено событие: e4
  Активный автомат: main
  Находиться в состоянии s1
Шаг: 3
  Активный автомат: main
  Выполняет переход: s1 -> e4[x1 < x2 & doors in s1]/o.z3(); ->
        s2
Шаг: 4
  Активный автомат: main
  Выполняет переход: s1 -> e4[x1 < x2 & doors in s1]/o.z3(); ->
        s2
  Конец перехода.
Шаг: 5
  Автомат doors в состоянии s1
  Автомат main в состоянии s2
Шаг: 6
  Получено событие: e5
  Активный автомат: main
  Находиться в состоянии s2
Шаг: 7
  Активный автомат: main
  Выполняет переход: s2 -> e5[x1 = x2]/o.z5(),doors.e1(); -> s1
Шаг: 8
  Активный автомат: main
  Выполняет переход: s2 -> e5[x1 = x2]/o.z5(),doors.e1(); -> s1
  Вызывает автомат doors с событием e1
Шаг: 9
    Получено событие: e1
    Активный автомат: doors
    Находиться в состоянии s1
Шаг: 10
    Активный автомат: doors
    Выполняет переход: s1 -> e1/o.z1(); -> s2
Шаг: 11
    Активный автомат: doors
    Выполняет переход: s1 -> e1/o.z1(); -> s2
    Конец перехода.
Шаг: 12
  Активный автомат: main
  Выполняет переход: s2 -> e5[x1 = x2]/o.z5(),doors.e1(); -> s1
  Конец перехода.
Шаг: 13
```

```
Автомат doors в состоянии s2
Автомат main в состоянии s1
```

Разберем контрпример более подробно. Ставилась задача проверить свойство, что лифт всегда может перейти в состояние `s2` (находится в движении). В контрпримере указывается что если, автомат `main` находится в состоянии `s1`, а автомат `doors` в состоянии `s2`, то автомат `main` не сможет перейти в состояние `s2`.

1.2. Метод верификации визуальных автоматных моделей

1.2.1. Постановка задачи

Требуется создать метод верификации визуальных [8] автоматных моделей [9] с более высоким уровнем автоматизации, по сравнению с известными методами [2, 14].

Существует большое число верификаторов, в том числе и с открытым кодом. Один из наиболее популярных верификаторов носит название *SPIN* [10]. Традиционно верификация программ с помощью верификатора *SPIN,* происходит следующим образом [2, 11]:

1. По разработанному алгоритму или программе вручную строится модель на языке *Promela* – входном языке верификатора *SPIN*.
2. Разрабатываются и записываются на языке *LTL* [12] требования (спецификация) к модели, которые верификатор преобразовывает в *автомат Бюхи* [13], 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* [2, 11] и верифицирует ее.
    - Запускается верификатор *SPIN* с ключами `–t` (и при необходимости `–p`, с этим ключом выводится более полная информация о контрпримере) для вывода контрпримера случае несоответствия модели требованиям.
4. Если модель не соответствует требованиям, 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.

Такой подход имеет следующие недостатки:

- Не гарантируется отсутствие ошибок в программе.
- Модель приходится строить вручную, а это трудоемкий процесс.

Однако для визуальных автоматных программ модель может быть автоматически построена по самой программе. Таким образом, в этом случае верификация гарантирует правильность работы программы. В этой области уже проводились исследования [14], однако и в этой работе предложенные методы предполагают ручное построение модели на языке *Promela*.

Таким образом, возникает следующая задача: разработать метод, позволяющий повысить уровень автоматизации автоматных программ. Для решения этой задачи и был выбран верификатор *SPIN*.

Предлагаемый метод состоит из следующих этапов:

1. Автоматизированное построение модели программы на языке *Promela* по визуальной автоматной программе.
2. Автоматическое преобразование записанных вручную на языке *Promela* проверяемых требований, соответствующих *LTL*-формулам,
3. Автоматическая верификация сгенерированной модели с использованием требований в нотации верификатора *SPIN*.
4. Автоматическое построение контрпримера по модели.
5. Преобразование (возможно, автоматическое) полученного контрпримера в контрпример в автоматной программе.

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

При построении модели используются:

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

Построенная модель абстрагируется от:

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

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

Опишем метод построения модели.

1. Подготовка исходных данных.

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

```
int stateAi;
```

1.2. Введем переменную для событий:

```
int lastEvent;
```

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

```
stateAi = k;
```

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

```
lastEvent = xx;
```

2. Построение

2.1. Для каждого автомата `Ai` создадим функцию. На языке *Promela* это записывается так:

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

2.2. Для каждого автомата `Ai` в теле созданной функции выполнить шаги 2.3 – 2.9.

2.3. Определить начальное (стартовое) состояние `s`. Присвоить:

```
stateAi = s;
```

2.4. Построить цикл:

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

Здесь `s1, …, sk` – номера состояний автомата `Ai`.

Инструкция `printf` аналогична соответствующей инструкции из языка *C*. Пометка используется в дальнейшем для восстановления контрпримера.

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

```
Am();
```

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

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

Это означает, что необходимое условие для присваивания `stateAi = sl` выполнено всегда.

В результате строится конструкция следующего вида:

```
if
::stateAi = s1
::stateAi = s2
…
::stateAi = sk
fi;
```

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

2.7. Если переход помечен событием `exx`, то дописать выражение

```
lastEvent = xx;
```

Таким образом, в результате получается конструкция вида:

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

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

```
break;
```

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

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

init {
run Model();
}
```

2.10. Дописать в модель требования (проверяемые свойства), преобразованные с помощью верификатора *SPIN* из формул на языке *Promela*, соответствующих *LTL*-формулам, в нотацию верификатора *SPIN*.

Пример. Рассмотрим автомат из программы [15] (автомат *A3*, рис. 12) и его модель на языке *Promela*, сгенерированную по автомату на основе предлагаемого метода построения модели. В пример не входит построение требований. В данном примере строится модель на языке *Promela* для автомата как для целой программы для того, чтобы продемонстрировать выполнение шага 2.9 алгоритма.

В автомате *A3* три состояния: `Начальное`, `Конечное` и `Главное интерфейсное состояние`. Перенумеруем их как показано на рис. 12. Основным текстом приведены комментарии к модели. Также в комментариях написано, по какому шагу алгоритма был сгенерирован код.

Заведем переменную для автомата `A3`. Шаг 1.1 алгоритма.

```
int stateA3;
```

Заведем переменную для событий. Шаг 1.2 алгоритма.

```
int lastEvent;
```

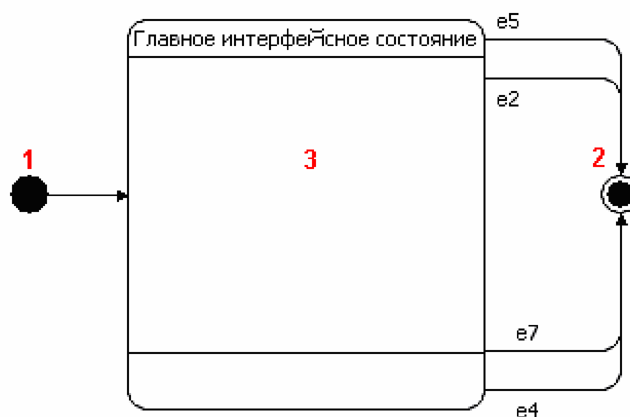

Рис. 12. Автомат *A3*

Заведем функцию для автомата `A3`. Шаг 2.1 алгоритма.

```
inline A3() {
```

Присвоим переменной `stateA3` номер начального состояния. Шаг 2.3 алгоритма.

```
stateA3 = 1;
```

Цикл, в котором происходит работа автомата. Шаг 2.4 алгоритма.

```
do
```

Вошли в `Стартовое состояние`. Шаг 2.4 алгоритма.

```
        ::(stateA3 == 1) ->
```

Пометка для отчета. Шаг 2.4 алгоритма.

```
          printf("State 1 : Начальное состояние\n");
```

Переход из `Начального состояния`. Шаг 2.6 алгоритма.

```
          if
            ::stateA3 = 3;
          fi;
```

Вошли в `Конечное состояние`. Шаг 2.4 алгоритма.

```
        ::(stateA3 == 2) ->
```

Пометка для отчета. Шаг 2.4 алгоритма.

```
          printf("State 2 : Конечное состояние\n");
```

Выход из цикла. Шаг 2.8 алгоритма.

```
          break;
```

Вошли в Главное интерфейсное состояние. Шаг 2.4 алгоритма.

```
        ::(stateA3 == 3) ->
```

Пометка для отчета. Шаг 2.4 алгоритма.

```
          printf("State 3 : Главное интерфейсное состояние\n");
```

Переходы из `Главного интерфейсного состояния`. Шаг 2.6 алгоритма.

```
          if
```

Переход в `Конечное состояние`. Шаг 2.6 алгоритма.

```
            ::stateA3 = 2;
```

Произошло событие `e1`. Шаг 2.7 алгоритма.

```
              lastEvent = 1;
```

Переход в `Конечное состояние`. Шаг 2.6 алгоритма.

```
            ::stateA3 = 2;
```

Произошло событие `e2`. Шаг 2.7 алгоритма.

```
              lastEvent = 2;
```

Переход в `Конечное состояние`. Шаг 2.6 алгоритма.

```
            ::stateA3 = 2;
```

Произошло событие `e4`. Шаг 2.7 алгоритма.

```
              lastEvent = 4;
```

Переход в `Конечное состояние`. Шаг 2.6 алгоритма.

```
            ::stateA3 = 2;
```

Произошло событие `e5`. Шаг 2.7 алгоритма.

```
              lastEvent = 5;
          fi;
```

Конец цикла. Шаг 2.4 алгоритма.

```
        od;
```

Конец функции `A3`. Шаг 2.1 алгоритма.

```
}
```

Создание процесса, в котором запускается автомат `A3`. Шаг 2.9 алгоритма.

```
proctype Model() {
```

Запуск функции `A3`. Шаг 2.9 алгоритма.

```
  A3();
}
```

Запуск процесса `Model`в главном процессе `init`. Шаг 2.9 алгоритма.

```
init {
run Model();
}
```

Таким образом, построена модель вида:

```
int stateA3;
int lastEvent;
inline A3() {
  stateA3 = 1;
  do
  ::(stateA3 == 1) ->
    printf("State 1 : Начальное состояние\n");
    if
      ::stateA3 = 3;
    fi;
  ::(stateA3 == 2) ->
    printf("State 2 : Конечное состояние\n");
    break;
  ::(stateA3 == 3) ->
    printf("State 3 : Главное интерфейсное состояние\n");
    if
      ::stateA3 = 2;
        lastEvent = 1;
      ::stateA3 = 2;
        lastEvent = 2;
      ::stateA3 = 2;
        lastEvent = 4;
      ::stateA3 = 2;
        lastEvent = 5;
    fi;
  od;
}
proctype Model() {
  A3();
}
init {
run Model();
}
```

Отметим, что в автомате `A3` четыре перехода из `Главного интерфейсного состояния` в `Конечное состояние`. При ручном создании модели соответствующий фрагмент кода можно было бы записать в виде:

```
if
  ::stateA3 = 2 ->
    if
      ::lastEvent = 1;
      ::lastEvent = 2;
      ::lastEvent = 4;
      ::lastEvent = 5;
    fi;
fi;
```

Этот код короче и в нем нет дублирования. Однако при автоматическом построении модели на языке *Promela* такой способ генерации кода менее удобен, так как алгоритм его генерации более сложен и его труднее расширить для построения более подробной модели.

1.2.3. Расширение нотации верификатора *SPIN* для языка линейной темпоральной логики

Нотация верификатора *SPIN* для языка *LTL* предполагает, что все элементарные высказывания записаны в виде некоторых идентификаторов, которые расшифровываются отдельно [16]. Излагаемый метод расширяет указанную нотацию *SPIN*, добавляя возможность записи элементарных высказываний в формуле. В качестве элементарного высказывания может использоваться любое выражение на языке *Promela*. Элементарное высказывание требуется записывать в фигурных скобках.

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

1. Алгоритму на вход подается строка с формулой.
2. Вводится счетчик элементарных высказываний.
3. Пока не кончилась строка, идем по строке в поиске открывающих фигурных скобок.
4. Если нашли открывающую фигурную скобку, то увеличиваем счетчик на единицу и пойдем по строке дальше в поиске закрывающей фигурной скобки.
5. Если закрывающая фигурная скобка не была найдена до конца строки или до момента появления новой открывающей фигурной скобки, то *LTL*-формула неправильная. Прекратить работу.
6. Если закрывающая фигурная скобка была найдена, то заменим элементарное высказывание, находящееся между фигурными скобками на идентификатор `pk`, который состоит из символа `p` и числа `k`, которое показывает счетчик элементарных высказываний, и запишем в модель следующий код:
   ```
   #define pk (элементарное высказывание)
   ```
   Переходим к шагу 3.
7. Если строка закончилась, то завершить работу.

В преобразованном виде *LTL*-формула может быть обработана верификатором *SPIN* (При условии, что темпоральные операторы записаны в нотации верификатора *SPIN* [16]).

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

Приведем пример работы предложенного алгоритма. Для этого рассмотрим формулу:

```
!({stateA1 != 4} U {stateA1 == 5})
```

Алгоритм работает следующим образом:

- Вводим счетчик элементарных высказываний *c* = 0 (шаг 2 алгоритма).
- Идем по строке с формулой (шаг 3 алгоритма).
- Дойдя до третьего символа, находим открывающую фигурную скобку. Увеличиваем *c* на единицу. Теперь *c* = 1. Продолжаем идти по строке (шаг 4 алгоритма).
- Находим закрывающую фигурную скобку (16 символ). Заменяем выражение (элементарное высказывание) `{stateA1 != 4}` на идентификатор `p1`. В результате, формула приобретает вид:
  ```
  !(p1 U {stateA1 == 5})
  ```
  При этом дополнительно в модель записываем следующий код:
  ```
  #define p1 (stateA1 != 4)
  ```
  Продолжаем идти по строке (шаг 6 алгоритма).
- Находим открывающую фигурную скобку (20 символ первоначальной формулы). Увеличиваем *c* на единицу. Теперь *c* = 2. Продолжаем идти по строке (шаг 4 алгоритма).
- Находим закрывающую фигурную скобку (33 символ исходной формулы). Заменяем выражение `{stateA1 == 5}` на идентификатор `p2`. В результате формула приобретает вид:
  ```
  !(p1 U p2)
  ```
  При этом дополнительно в модель записываем следующий код:
  ```
  #define (p2 stateA2 == 5)
  ```
  Продолжаем идти по строке (шаг 3 алгоритма).
- Дошли до конца строки. Завершение работы (шаг 7 алгоритма).

1.2.4. Типовые спецификации к автоматным моделям

Рассмотрим типовые требования, которые можно применять ко всем автоматным моделям [**17**].

1. Завершение работы. В рамках автоматной программы это требование может быть записано на языке темпоральной логики следующим образом:
   ```
   F(state == end_state),
   ```
   где `state` — состояние главного автомата, `end_state` — конечное состояние. Эта формула означает, что главный автомат когда-нибудь попадет в конечное состояние.
2. Прогресс. Требование, что в любой момент выполнения программы автомат может прийти в состояние `s` или произойдет событие `exx`. На языке линейной темпоральной логики это требование записывается следующим образом:
   ```
   GF(state == s) или GF(lastEvent == exx).
   ```
3. Постоянство. Требование о том, что некоторое свойство `p` с некоторого момента выполняется всегда. Это свойство применимо не только к автоматным программам, а к любой модели. Его запись на языке *LTL*:
   ```
   FG(p).
   ```

В данном разделе не описаны такие требования как, например, корректное завершение работы, так как темпоральная формула для них зависит от конкретной модели.

1.2.5. Пример использования предлагаемого метода

1.2.5.1. Постановка задачи

Рассмотрим автоматную реализацию игры *Ним* [15]. *Ним* – это игра для двух игроков, каждый из которых по очереди делает ход. Перед игроками располагается поле с фишками. Известны различные варианты игры Ним. В данном проекте правила игры таковы:

- фишки раскладываются в несколько рядов;
- игроки по очереди забирают фишки из любого ряда;
- не разрешается за один ход брать фишки из нескольких рядов;
- за один ход игрок должен взять хотя бы одну фишку;
- выигрывает тот, кто возьмет последнюю фишку (фишки).

В данном примере использована одна из первых версий реализации игры с одним автоматом *A1* (рис. 13). Цифрами указаны номера состояний, а символ «*» соответствует любому событию.

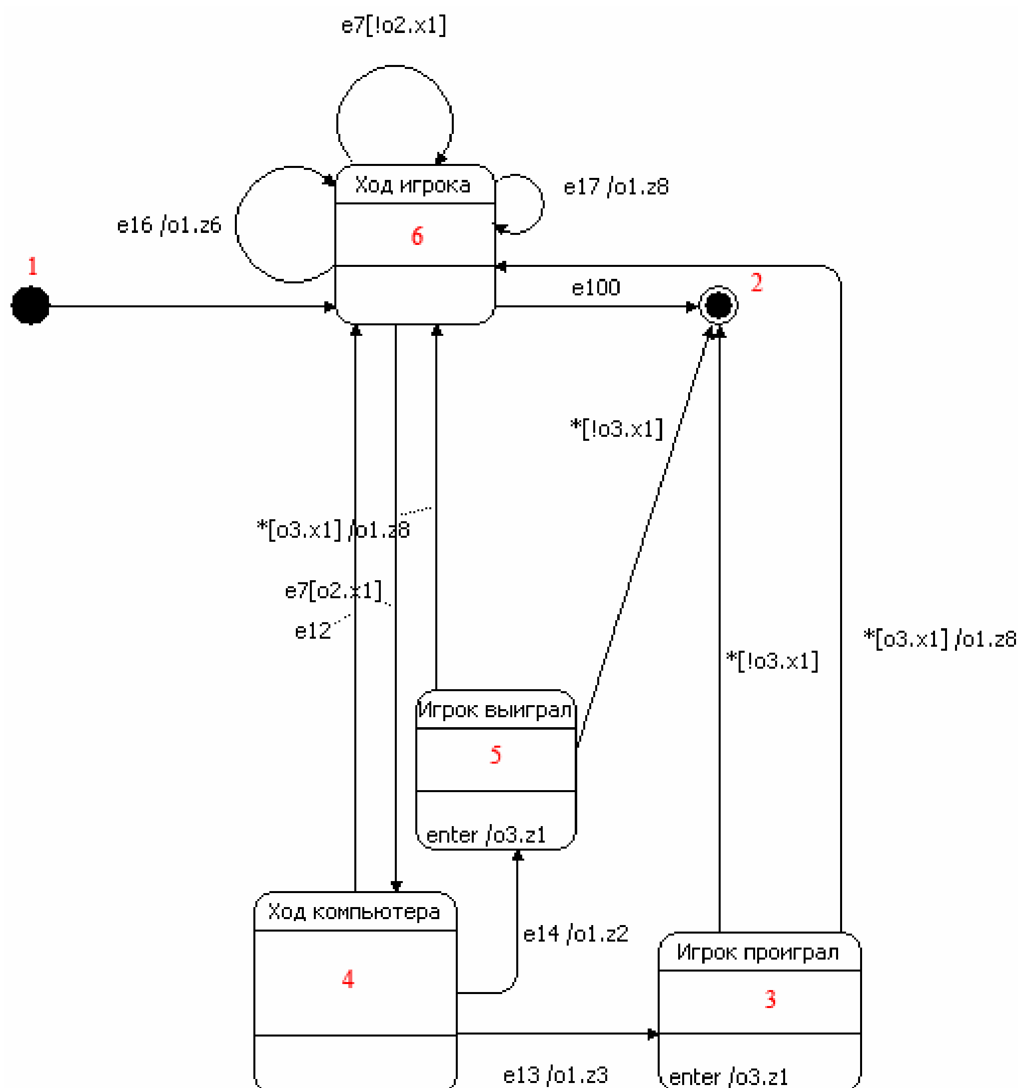

Рис. 13. Визуальная автоматная программа для игры *Ним*

Допустим, в ней была допущена ошибка (рис. 14). На этом рисунке лишний переход выделен жирно.

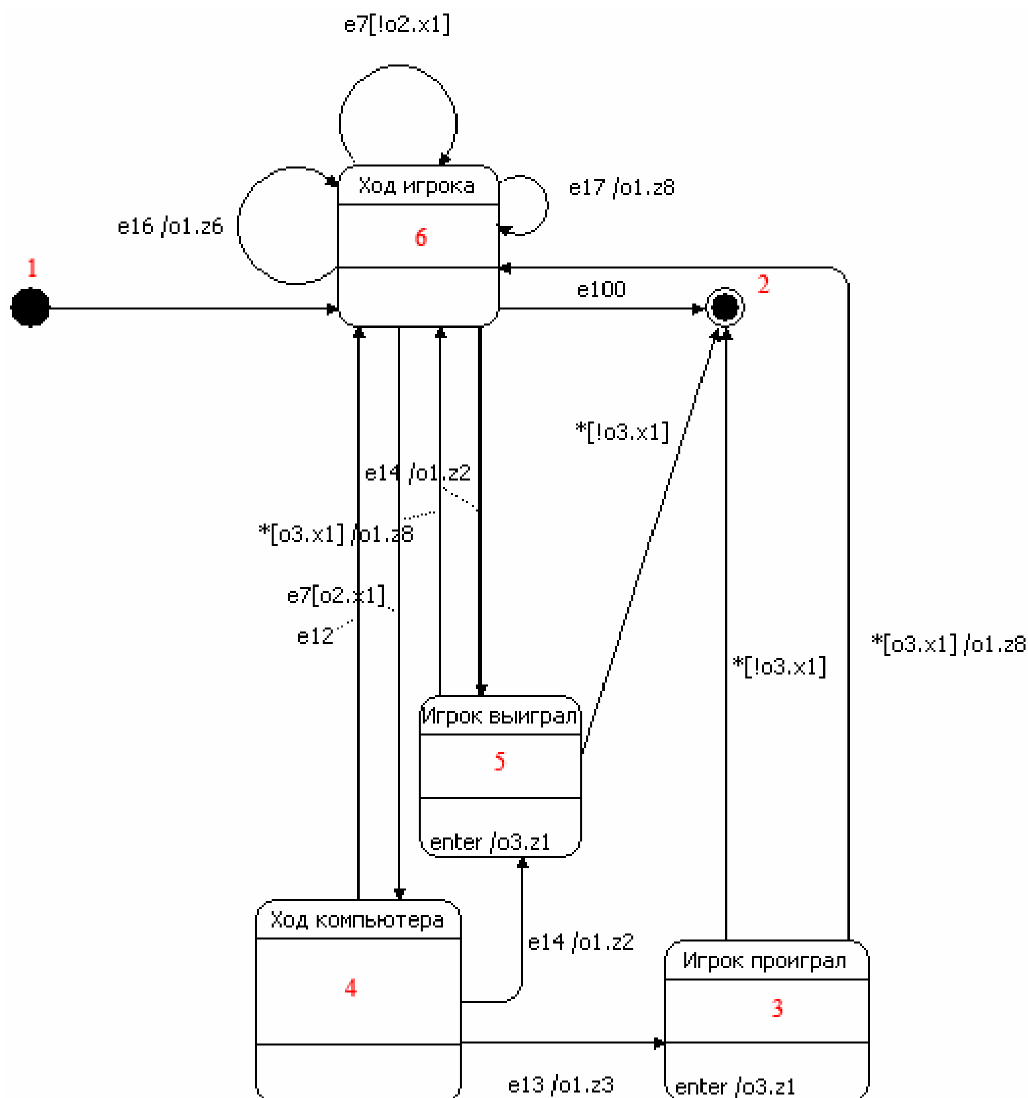

Рис. 14. Реализация с ошибкой

1.2.5.2. Верификация

В данной реализации решение о том, кто выиграл, принималось в состоянии «`Ход компьютера`». Таким образом, можно сформулировать требование: в состояние «`Игрок выиграл`» можно попасть только из состояния «`Ход компьютера`». Запишем это требование на языке *LTL*:

```
¬ ((stateA1 != 4) U (stateA1 == 5)).
```

Преобразуем его в нотацию верификатора *SPIN*. Для этого утверждение `stateA1 != 4` обозначим как `p1`, а утверждение `stateA1 == 5` обозначается как `p2`. При этом рассматриваемое требование записывается в виде:

```
#define p1  (stateA1 != 4)
#define p2  (stateA1 == 5)
```

Тогда требование в нотации верификатора S*PIN* будет выглядеть следующим образом:

```
!(p1 U p2).
```

Подадим это требование на вход верификатора *SPIN*. Это делается с помощью команды `spin -f "!(p1 U p2)"`. Верификатор *SPIN* выдаст следующий код:

```
never {    /* p1 U p2 */
T0_init:
  if
  :: ((p2)) -> goto accept_all
  :: ((p1)) -> goto T0_init
  fi;
accept_all:
  skip
}
```

Эта конструкция в терминологии верификатора *SPIN* называется *never claim* и является *автоматом Бюхи*, записанным на языке *Promela*.

Построим модель на языке *Promela* с помощью метода, изложенного выше, и добавим в нее код, сгенерированный верификатором *SPIN* по проверяемым требованиям (конструкцию *never claim*):

Жирным шрифтом выделен неправильный переход. Файл с моделью назовем `incorr.ltl`. В качестве основного текста используются комментарии к модели.

Необходимые для конструкции *never claim* определения.

```
#define p1   (stateA1 != 4)
#define p2   (stateA1 == 5)
```

Модель автоматной программы.

```
int lastEvent;

int stateA1;
inline A1() {
  stateA1 = 1;
  do
  ::(stateA1 == 1) ->
    printf("State 1 : s2\n");
    if
      ::stateA1 = 6;
    fi;
  ::(stateA1 == 2) ->
    printf("State 2 : s3\n");
    break;
  ::(stateA1 == 3) ->
    printf("State 3 : Игрок проиграл\n");
    if
      ::stateA1 = 2;
      ::stateA1 = 6;
    fi;
  ::(stateA1 == 4) ->
    printf("State 4 : Ход компьютера\n");
    if
      ::stateA1 = 3;
        lastEvent = 13;
```

```
          ::stateA1 = 6;
            lastEvent = 12;
          ::stateA1 = 5;
            lastEvent = 14;
        fi;
      ::(stateA1 == 5) ->
        printf("State 5 : Игрок выиграл\n");
        if
          ::stateA1 = 6;
          ::stateA1 = 2;
        fi;
      ::(stateA1 == 6) ->
        printf("State 6 : Ход игрока\n");
        if
          ::stateA1 = 6;
            lastEvent = 16;
          ::stateA1 = 4;
            lastEvent = 7;
          ::stateA1 = 6;
            lastEvent = 7;
          ::stateA1 = 6;
            lastEvent = 17;
          ::stateA1 = 2;
            lastEvent = 100;
          ::stateA1 = 5;
            lastEvent = 14;
        fi;
      od;
    }
```

Процесс, запускающий модель автоматной программы.

```
proctype Model() {
  A1();
}
```

Главный процесс `init`, запускающий процесс с моделью автоматной программы.

```
init {
run Model();
}
```

Конструкции *never claim*.

```
never {    /* p1 U p2 */
T0_init:
  if
  :: ((p2)) -> goto accept_all
  :: ((p1)) -> goto T0_init
  fi;
accept_all:
  skip
}
```

По команде `spin -a incorr.ltl` верификатор *SPIN* по модели на языке *Promela* построит программу `pan.c` на языке *C*. Скомпилируем программу и запустим ее командой `pan -n`. Результатом работы этой программы является отчет.

В строке, приведенной ниже, сказано, что было нарушено условие *never claim*. Иными словами, модель вступила в противоречие с проверяемыми требованиями:

```
pan: claim violated! (at depth 19)
```

Далее идет информация о том, где лежит контрпример, номер версии *SPIN* и другая информация:

```
pan: wrote models/incorr.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 24 byte, depth reached 27, errors: 1
      20 states, stored
       4 states, matched
      24 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)

2.622  memory usage (Mbyte)
```

В этом отчете требуется обратить внимание только на первую строку (`pan: claim violated! (at depth 19)`) и сообщение о числе ошибок, выделенное жирным шрифтом. Таким образом, верификатор сообщил, что в нашей программе имеется ошибка, и она заключается в противоречии с проверяемым требованием к программе.

Кроме того, если в модели имеются ошибки, то программа `pan` строит *trail*-файл, в котором содержится информация о контрпримере. В данном примере файл называется `incorr.ltl.trail`, который представлен в такой «нечеловеческой» форме:

```
-2:2:-2
-4:-4:-4
1:0:55
2:1:51
3:0:55
4:2:0
5:0:55
6:2:1
7:0:55
8:2:2
9:0:55
10:2:3
11:0:55
12:2:31
13:0:55
14:2:32
15:0:55
16:2:43
17:0:53
18:2:25
19:0:59
20:1:52
```

1.2.5.3. Анализ контрпримера

По команде `spin -t -p incorr.ltl` верификатор *SPIN* обработает файл `incorr.ltl.trail` и построит отчет, содержащий контрпример. Для удобства в отчете жирным шрифтом выделены моменты входа в новое состояние.

Проанализируем отчет, выданный верификатором *SPIN*. В начале каждой строки трассы содержится номер процесса и его имя в круглых скобках (Например, `proc 1 (Model)`). После этого записан номер строки в модели на языке *Promela* и имя файла с моделью (`line 15 "models/incorr.ltl"`). Далее в круглых скобках написан номер состояния в *модели Крипке*, построенной верификатором *SPIN* (`(state 1)`). В квадратных скобках указан код на языке *Promela*, соответствующий состоянию в *модели Крипке* (`[stateA1 = 1]`). В качестве основного текста используются комментарии к отчету.

Начало контрпримера.

```
Starting :init: with pid 0
Starting :never: with pid 1
```

Отображение перехода *автомата Бюхи*.

```
Never claim moves to line 75  [((stateA1!=4))]
Starting Model with pid 2
```

Переход в *модели Крипке* (в круглых скобках записано состояние *модели Крипке*) и соответствующий код (в квадратных скобках) в модели на языке *Promela*.

```
  2:  proc  0 (:init:) line  69 "models/incorr.ltl" (state 1)
          [(run Model())]
```

Автомат `A1` переходит в состояние 1 (`начальное состояние`).

```
  4:  proc  1 (Model) line  15 "models/incorr.ltl" (state 1)
          [stateA1 = 1]
  6:  proc  1 (Model) line  17 "models/incorr.ltl" (state 2)
          [((stateA1==1))]
              State 1 : s2
  8:  proc  1 (Model) line  18 "models/incorr.ltl" (state 3)
          [printf('State 1 : s2\\n')]
```

Автомат `A1` переходит в состояние 6 (`Ход игрока`).

```
 10:  proc  1 (Model) line  20 "models/incorr.ltl" (state 4)
          [stateA1 = 6]
 12:  proc  1 (Model) line  47 "models/incorr.ltl" (state 2)
          [((stateA1==6))]
              State 6 : Ход игрока
 14:  proc  1 (Model) line  48 "models/incorr.ltl" (state 33)
          [printf('State 6 : Ход игрока\\n')]
```

Автомат `A1` переходит в состояние 5 (`Игрок выиграл`).

```
 16:  proc  1 (Model) line  60 "models/incorr.ltl" (state 44)
          [stateA1 = 5]
```

Отображение перехода *автомата Бюхи*.

```
Never claim moves to line 74  [((stateA1==5))]
```

```
 18:  proc  1 (Model) line  41 "models/incorr.ltl" (state 26)
          [((stateA1==5))]
```

Автомат Бюхи перешел в недопускающее состояние.

```
Never claim moves to line 78  [(1)]
SPIN: trail ends after 20 steps
```

Отображение общей информации.

```
#processes: 2
    lastEvent = 0
    stateA1 = 5
 20:  proc  1 (Model) line  42 "models/incorr.ltl" (state 27)
 20:  proc  0 (:init:) line  70 "models/incorr.ltl" (state 2)
        <valid end state>
 20:  proc  - (:never:) line  79 "models/incorr.ltl" (state 8)
        <valid end state>
2 processes created
```

Таким образом, из отчета следует, что контрпримером является путь (рис. 15): начальное состояние — состояние 6 (`Ход игрока`) — состояние 5 (`Игрок выиграл`).

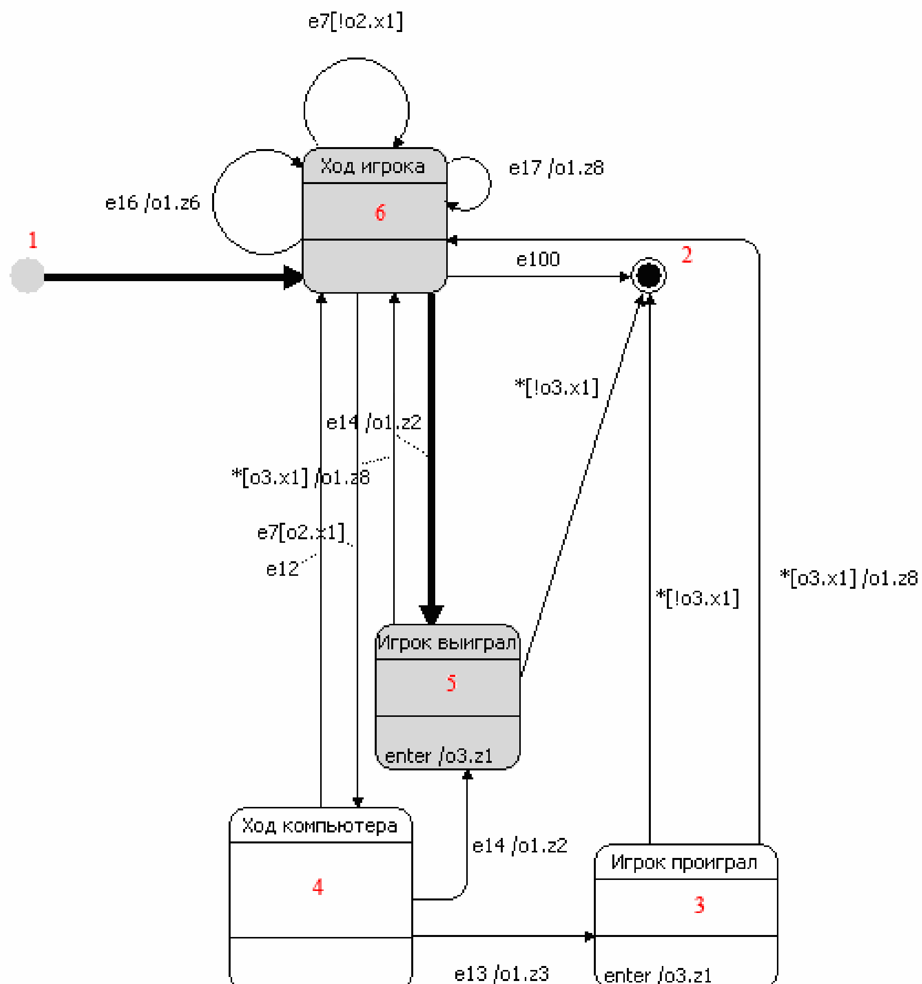

Рис. 15. Контрпример

Исправим автомат, удалив лишний переход. Построим для него модель на языке *Promela*. Файл с моделью правильного автомата назовем `corr.ltl`.

Подадим теперь на вход верификатору файл `corr.ltl` и такое же требование. Программа `pan`, построенная верификатором, выдаст следующий отчет:

```
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 24 byte, depth reached 27, errors: 0
      18 states, stored
       4 states, matched
      22 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)

2.622   memory usage (Mbyte)
```

Строка из отчета «`errors: 0`» говорит о том, что ошибка исправлена.

1.3. Метод верификации автоматных моделей на основе их эмуляции

1.3.1. Постановка задачи

Метод эмуляции позволяет решает задачу верификации путем «управляемой» эмуляции работы автоматной программы с наблюдением за верифицируемыми свойствами программы. Такой подход позволяет напрямую работать с верифицируемой автоматной программой без дополнительных преобразований или построений.

Опишем задачу верификации автоматных программ.

Рассмотрим реактивную систему, состоящую из трех компонент: источники событий, система управления и объекты управления. Реактивная система работает следующим образом: в источниках событий возникают события, которые передаются системе управления. Система управления обрабатывает события и вызывает действия в объектах управления. Во время обработки события система управления может запрашивать у объектов управления их свойства. Схема работы реактивной системы изображена на рис. 16.

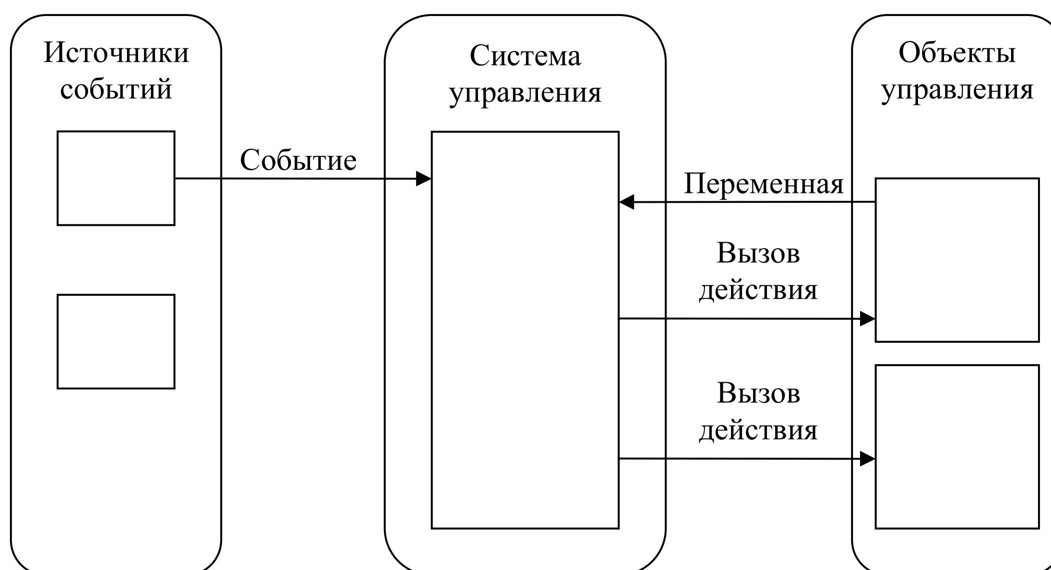

Рис. 16. Схема работы реактивной системы

Представим теперь, что система управления реализована в виде системы иерархически связанных конечных автоматов. Каждый автомат состоит из множества состояний и переходов между ними. Каждый переход помечен, во-первых, событием, при котором он может осуществиться, во-вторых, условием, которое должно выполняться для перехода, а, в-третьих, набором действий, которые будут вызваны в объектах управления при этом переходе. Таким образом, каждый автомат работает следующим образом. При получении события, автомат выбирает те переходы из своего текущего состояния, которые помечены этим событием. Затем он вычисляет условия на выбранных переходах, запрашивая значения переменных у объектов управления. Если найден переход, для которого условия выполняются, автомат вызывает перечисленные действия объектов управления и переходит в конечное для найденного перехода состояние. Рассмотрим переход автомата между состояниями на примере простого автомата, изображенного на рис. 17.

Рис. 17. Пример простого автомата

Начальным состоянием изображенного автомата является состояние `s1`, что обозначено черным кругом, указывающим на это состояние. Надпись на переходе из состояния `s1` в состояние `s2` означает следующее: этот переход выполняется, когда произошло событие `e1`, и переменная `x1`

объекта управления o1 равна True. На этом переходе будет вызвано действие z1 объекта управления o1. В дальнейшем для краткости могут опускаться префиксы вида «o1.», что будет означать, что в реактивной системе лишь один объект управления, и все переменные и действия относятся к нему. Пусть автомат находится в начальном состоянии s1. Тогда при возникновении события e2 он не изменит состояние. При возникновении события e1, если o1.x1 равно true, то автомат вызовет действие o1.z1 и перейдет в состояние s2. Если же значение o1.x1 равно false, то автомат вызовет действие o1.z2 и перейдет в состояние s3.

Иерархически связанная система автоматов — это множество автоматов, организованных в иерархическую структуру (дерево). Пример такой структуры изображен на рис. 18.

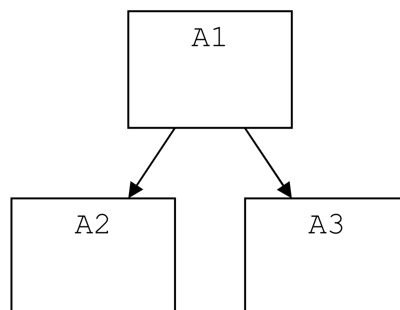

Рис. 18. Пример системы автоматов

Корень дерева – это главный автомат системы. Автоматы связываются путем вложения одних автоматов в состояния других. Дочерние автоматы могут вкладываться в состояния своего родительского автомата. Главный автомат не может быть вложен ни в одно состояние. Система автоматов работает следующим образом. Все возникающие события попадают на обработку главному автомату. Главный автомат обрабатывает событие. После этого, если в его новом состоянии есть вложенные автоматы, то в них вызывается обработка того же события. В результате обработки вложенным автоматом событие, в свою очередь, может быть передано следующему по вложенности автомату. Система автоматов завершает обработку события тогда, когда все вызванные автоматы завершили обработку события.

Важно, что система автоматов работает **в одном потоке**, что означает, что система не принимает на обработку новое событие, пока не завершена обработка предыдущего. Также в случае вложения нескольких автоматов в одно состояние, обработка события в них вызывается поочередно, а не параллельно.

Кроме вложения автоматов в состояния, возможны также следующие типы связи между автоматами. Во-первых, автомат может вызывать при переходе обработку любого события дочерним автоматом. Например, метка на переходе «e1/o1.z1,A2.e2» означает, что на этом переходе будет вызвано действие z1 объекта управления o1, а также будет вызвана обработка события e2 в автомате A2. Кроме того, состояния дочерних автоматов могут использоваться в условиях на переходах. Например, метка на переходе «e1[A2.s1]» означает, что этот переход возможен при условии, что автомат A2 находится в состоянии s1.

У автоматов существуют конечные состояния, автомат завершает работу, когда попадает в такое состояние. Если главный автомат системы попал в конечное состояние, то автоматная программа завершила работу. Наличие конечного состояния в автомате необязательно.

Верификация автоматных программ заключается в проверке того, что управляющая система автоматов реактивной системы работает корректно. Понятие корректности задается темпоральным

утверждением вида «если произошло событие `e1`, то когда-нибудь будет вызвано действие `z1`», или «если всегда неверно `x1` (`x1` всегда `false`), то автомат никогда не попадет в состояние `s2`». Утверждение, которое требуется проверить, называют *требованиями*. В том случае, если система автоматов удовлетворяет требованиям, что означает, что сформулированное утверждение выполняется для любой возможной истории работы системы, то верификация завершается успешно. Если же утверждение не выполняется, и существует последовательность действий, которая приводит систему автоматов к нарушению сформулированного требования, то выводится этот набор действий (его называют «*контрпример*»).

Для решения задачи верификации автоматных программ было выполнено несколько работ, в том числе работа [1], где был предложен метод верификации автоматных программ с помощью верификатора *SPIN*. В настоящей работе предлагается оригинальный метод верификации автоматных программ с помощью *эмуляции* (или имитации) работы автоматной программы. Этот метод позволяет значительно снизить сложность преобразований над исходной автоматной программой, необходимых для верификации, по сравнению с методом, описанным в работе [1].

1.3.2. Верификация с использованием алгоритма двойного поиска в глубину

Метод эмуляции использует алгоритм двойного поиска в глубину [2] для верификации автоматных программ. Этот алгоритм используется во многих верификаторах, в том числе в верификаторах *SPIN* и *Bogor*. В настоящем разделе будет описана схема применения этого алгоритма для верификации программ, и сам алгоритм.

Теоретически верификация с применением алгоритма двойного поиска в глубину выполняется следующим образом. Сначала для верифицируемой программы строится модель Крипке [2] – граф элементарных (атомарных) состояний, в которых может находиться программа, и переходов между ними.

При использовании верификатора, такого, как *SPIN*, построение модели Крипке выполняется верификатором. Собственно верификация выполняется автоматически.

Построенная модель Крипке преобразуется в автомат. Это необходимо для применения алгоритма двойного поиска в глубину: представление модели Крипке в виде автомата позволяет пересечь его с автоматом Бюхи для поиска контрпримера (подробнее об этом будет рассказано ниже). Пример модели Крипке и автомата, полученного из нее, изображен на рис. 19.

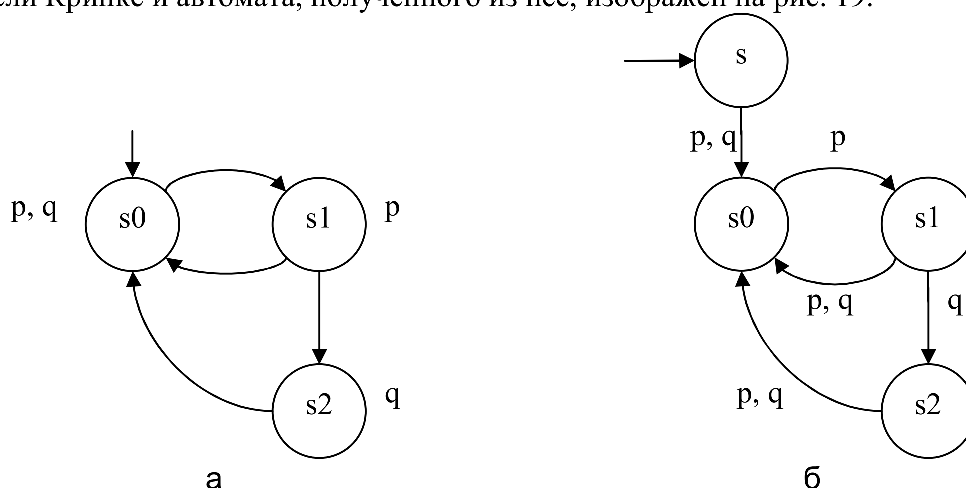

Рис. 19. Пример модели Крипке (а) и ее преобразования в автомат (б)

Модель Крипке является подробной схемой работы программы. На рисунке программа начинает работу в состоянии, соответствующем состоянию *s0* модели Крипке. Затем совершаются какие-то действия, и состояние программы изменяется, что отражается переходом модели Крипке в состояние *s1*. Далее, в зависимости от ситуации, программа может совершить действия соответствующие переходу в состояние *s2* или обратно в состояние *s0*.

При построении автомата из модели Крипке вместо состояний предикатами помечаются переходы. Это делается достаточно просто: добавляется одно начальное состояние, а каждый переход помечается теми предикатами, которые выполнялись в конечном состоянии перехода в исходной модели Крипке. Стоит отметить, что при этом полученный автомат является автоматом Мура [2]: метки на переходах зависят только от конечного состояния перехода.

Требования к программе формулируются в виде формул темпоральной логики. Такие формулы позволяют специфицировать работу программы *во времени*. Для применения алгоритма двойного поиска в глубину используется литейная темпоральная логика (*LTL, Linear Temporal Logic*) [2]. Темпоральные формулы состоят из *предикатов* – элементарных утверждений о программе, логических операторов («не», «и», «или») и темпоральных операторов – операторов, описывающих выполнение утверждений во времени.

В ходе работы программы изменяются свойства и параметры программы. В каждый момент работы программы (после каждого элементарного шага выполнения программы, или, что то же, в каждом состоянии модели Крипке) можно создать список значений интересующих нас параметров (предикатов). Тогда вся история работы программы записывается в виде последовательности значений набора предикатов. Поскольку работа программы может быть недетерминирована, возникает «ветвление» историй работы программы: в некоторые моменты времени в зависимости от внешних условий возможно разное будущее работы программы. Таким образом, набор всех историй можно представить в виде дерева, в узлах которого находятся значения набора предикатов. Любой путь в этом дереве будет отражать один из вариантов работы программы во времени. Далее термин «путь» используется для обозначения истории работы программы.

В логике *LTL* используются следующие темпоральные операторы:

- **X** (neXt) – «*X p*» выполняется для тех путей, во втором состоянии которых выполняется предикат *p*;
- **G** (Globally) – «*G p*» выполняется для тех путей, в каждом узле которых выполняется *p*;
- **F** (Future) – «*F p*» выполняется для тех путей, в которых существует хотя бы одно состояние, в котором выполняется *p*;
- **U** (Until) – «*p U q*» выполняется для тех путей, в которых существует состояние, где выполняется *q*, и в каждом предыдущем состоянии (до первого *q*) выполняется *p*;
- **R** (Release) – «*q R p*» выполняется для тех путей, в которых *p* выполняется до тех пор, пока не станет выполняться *q* (включительно), или всегда, если *q* не выполняется никогда.

После построения автомата по модели Крипке, формула *LTL*, описывающая требования к программе, преобразуется в автомат Бюхи [2] – конечный автомат над бесконечными словами. Переходы автомата Бюхи помечены предикатами из исходной формулы *LTL*. Автомат работает следующим образом. На каждом шаге он берет очередной набор значений предикатов из последовательности, отражающей историю работы программы. Используя эти значения, он вычисляет метки на переходах из текущего состояния. Активными называются те переходы, метки которых были вычислены как `True`. Активных переходов может быть больше одного, автомат недетерминированно выбирает один из них. Таким образом, последовательность значений предикатов определяет возможные наборы переходов в автомате Бюхи.

Пример автомата Бюхи, полученного из формулы *LTL*, изображен на рис. 20. Формула «*G F p*» читается как «всегда когда-нибудь *p*» или «*p* будет выполняться бесконечно часто».

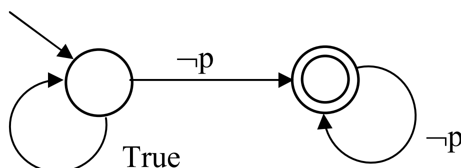

Рис. 20. Автомат Бюхи для формулы «*G F p*»

В автомате Бюхи существует набор «допускающих» состояний. На рис. 20 такое состояние помечено двойным кругом. Если для некоторой последовательности значений предикатов возможна такая история переходов в автомате Бюхи, что допускающие состояния посещаются *бесконечно часто*, то говорят, что автомат *допускает* эту последовательность значений предикатов.

Формула *LTL* и автомат Бюхи, построенный из нее, эквивалентны в том смысле, что если некоторый путь (история работы программы) удовлетворяет формуле, то он допускается автоматом Бюхи. И наоборот, любая последовательность значений предикатов, допускаемая автоматом Бюхи, удовлетворяет исходной формуле *LTL*.

Поскольку задача верификатора – найти контрпример, автомат Бюхи строится для *отрицания* исходной формулы *LTL* (смысл этого объясняется ниже). Таким образом, автомат Бюхи допускает любые последовательности значений предикатов, которые не удовлетворяют требованиям.

Для пояснения рассмотрим автомат, изображенный на рис. 20. Если существует момент времени, после которого *p* ни разу не выполнится, то автомат в этот момент перейдет в допускающее состояние и останется там навсегда. Таким образом, будет получен допускающий путь. Автомат Бюхи на рис. 20 недетерминирован: возможны два перехода из недопускающего состояния при не выполнении *p*. Кроме того, часто автомат строится неполным: например, если автомат на рис. 20 находится в допускающем состоянии и выполнено *p*, то не существует активного перехода. В таком случае считается, что история не допускается. Действительно, при выполнении *p* нарушится предположение о том, что начиная с выбранного момента времени *p* не выполнится ни разу.

После построения автомата по модели Крипке и автомата Бюхи, создается их автомат-пересечение. Это такой автомат, который допускает только те последовательности предикатов, которые допускаются и автоматом Бюхи, и моделью Крипке (в автомате модели Крипке все состояния считаются допускающими, а значит любая история его работы – допускающая). Построенное пересечение автоматов также является автоматом Бюхи. Над ним запускается алгоритм *двойного поиска в глубину* [2], который находит допускающую последовательность предикатов, если она существует. Если такая последовательность существует, значит, есть путь в модели Крипке, или, что то же самое, история работы исходной программы, в ходе которой возникают такие значения проверяемых свойств, которые допускаются автоматом Бюхи, а значит, нарушают исходную формулу *LTL* (вспомним, что автомат Бюхи строился для отрицания исходной формулы). Другими словами, найден контрпример.

Опишем алгоритм двойного поиска в глубину.

Для начала покажем, что структура пути, допускаемого автоматом Бюхи, имеет вид $\alpha\beta^*$, где α – некоторый префикс, а β – суффикс, порождаемый проходом автомата Бюхи по циклу, содержащему допускающее состояние. Структура пути изображена на рис. 21.

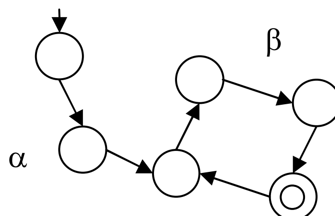

Рис. 21. Структура допускаемого пути в автомате Бюхи

Пусть *p* – путь, допускаемый автоматом Бюхи. Так как множество состояний автомата конечно, то найдется суффикс *p'* прохода *p*, такой что всякое состояние из него встречается бесконечное число раз. Это означает, что любое состояние этого суффикса достижимо из любого другого состояния суффикса. Следовательно, состояния из *p'* входят в состав некоторой сильно связной компоненты графа, описывающего автомат Бюхи. Причем, так как одно из допускающих состояний также входит в *p'*, то сильно связная компонента содержит допускающее состояние.

Обратно, если в графе автомата Бюхи существует достижимая сильно связная компонента, содержащая допускающее состояние, то легко построить допускающий путь. Он будет иметь структуру $\alpha\beta^*$, где α – префикс, который ведет к сильно связной компоненте, а β – цикл в этой компоненте, содержащий допускающее состояние.

Таким образом, проверка пустоты языка автомата Бюхи равносильна поиску сильно связной компоненты, достижимой из начального состояния и содержащей допускающее состояние. Для этого используется алгоритм двойного поиска в глубину (двойного *DFS – Depth First Search*).

В этом алгоритме чередуются два поиска в глубину. Первый из них может запускать второй, а второй, в свою очередь, может либо завершить работу всего алгоритма, либо передать управление обратно в первый *DFS*. В этом случае первый поиск в глубину продолжает свою работу. Каждый *DFS* используют свой флаг для пометки посещенных состояний.

Первый поиск в глубину запускает второй в тот момент, когда он готов к откату из допускающего состояния. Если второй поиск в глубину в процессе обхода попадает в состояние, находящееся в стеке первого, то допускающий путь получен. Если этого не происходит, то после завершения обхода второй поиск в глубину возвращает управление в первый.

Формально алгоритм проверки на пустоту языка автомата Бюхи с помощью двойного *DFS* выглядит следующим образом (на псевдокоде, взято из работы [2]):

```
procedure emptiness
for all q0 ∈ Q0 do
    dfs1(q0);
terminate(False);
end procedure

procedure dfs1(q)
local q';
hash(q);
for all последователей q' вершины q do
    if q' не содержится в хэш-таблице then dfs1(q');
if accept(q) then dfs2(q);
end procedure
```

```
procedure dfs2(q)
local q';
flag(q);
for all последователей q' вершины q do
    if q' в стеке dfs1 then terminate(True);
    else if q' не является помеченной then dfs2(q');
    end if;
end procedure
```

Алгоритм возвращает `True`, если был найден допускающий путь, и `False` – в противном случае. Если алгоритм вернул значение `True`, то можно восстановить допускающий путь: в стеке первого *DFS* хранится путь из начального состояния в некоторое допускающее состояние *q1*. Этот путь и будет искомым префиксом α. При этом в стеке второго *DFS* хранится путь из состояния *q1* в некоторое состояние *q2*, содержащееся в стеке первого. Тогда, достроив этот путь состояниями, находящимися в стеке первого *DFS* выше состояния *q2*, получим цикл $q1 \to q2 \to q1$, проходящий через допускающее состояние *q1*, а, следовательно, искомый суффикс β. Таким образом, будет получен допускающий путь $\alpha\beta^*$.

Доказательство корректности алгоритма подробно описано в работе [2].

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

Такие возможности позволяют верификатору избежать явного построения пересечения автомата модели Крипке с автоматом Бюхи, работая по следующей схеме.

- Верификатор работает с программой и с автоматом Бюхи, построенным по темпоральной формуле.
- Глобальное состояние верифицируемой системы складывается из глобального состояния программы и состояния, в котором находится автомат Бюхи.
- Каждый элементарный шаг совершается в два хода: сначала происходит элементарный переход в программе. После этого вычисляются необходимые предикаты, и совершается переход в автомате Бюхи. Если ни один переход в автомате Бюхи не возможен, то происходит откат: в данной ветви истории не удалось найти контрпример.
- Если переход в автомате Бюхи или в программе недетерминирован (возможны более одного перехода), то верификатор запоминает текущее состояние верифицируемой системы и набор возможных переходов. При откате в это состояние в ходе работы алгоритма двойного поиска в глубину, исполнение программы будет запущено снова с другими выбранными переходами. Таким образом, в каждой ситуации недетерминированности будут перебираться все возможные варианты работы верифицируемой системы.
- Верификатор считает, что верифицируемая система попала в допускающее состояние, когда автомат Бюхи находится в допускающем состоянии.

Такая схема действий позволяет работать по алгоритму двойного поиска в глубину без явного построения пересечения автомата модели Крипке с автоматом Бюхи.

Таким образом, для применения описанной схемы верификации, требуется решить пять задач.

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

Перечисленные пять необходимых свойств верифицируемой программы используются в дальнейшем описании предлагаемого метода.

1.3.3. Общее описание метода

Подробно опишем для автоматной программы решение задач, указанных в предыдущем разделе.

При верификации выделяется верифицируемая программа и внешняя среда. Внешняя среда неуправляема программой. Верификатор управляет внешней средой, и его цель – найти такие условия для программы, при которых требования не будут выполнены. Задача программы – удовлетворять требованиям при любых условиях.

Выделим верифицируемую систему и внешнюю среду для реактивной системы. Поскольку стоит задача верификации автоматной системы управления, то можно абстрагироваться от состояний объектов управления и считать, что состояние системы в них не хранится. Таким образом, объекты управления выносятся во внешнюю среду для верифицируемой системы. Источники событий также находятся во внешней среде. Автоматная система получает из внешней среды события и значения входных переменных. В свою очередь, во внешнюю среду она выдает команды объектам управления. Данная идея представлена на рис. 22.

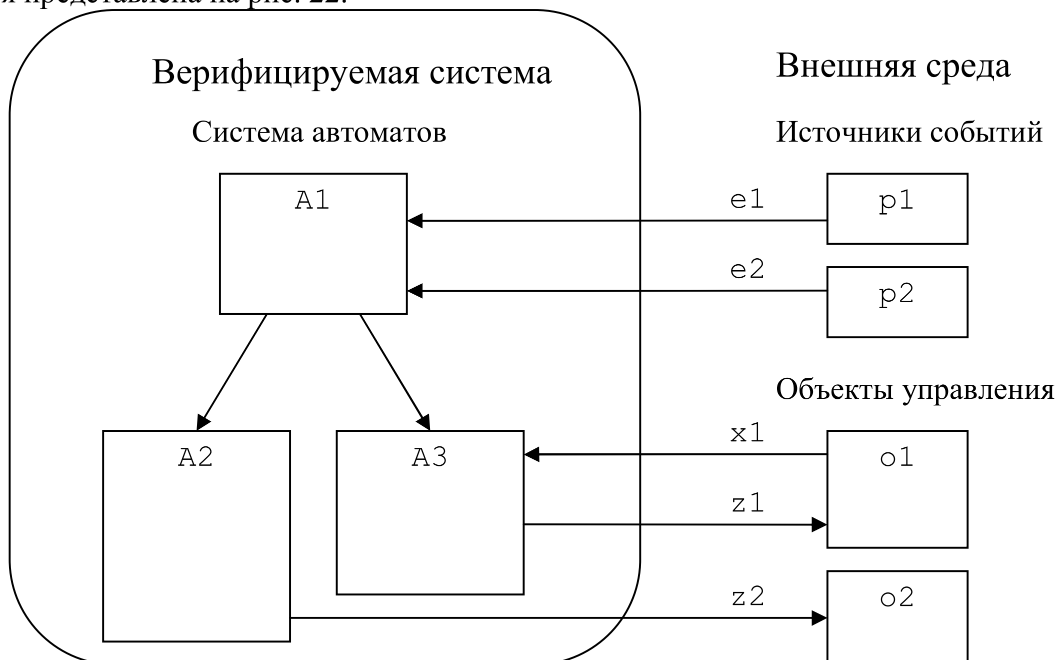

Рис. 22. Общая схема верифицируемой системы

При таком подходе состояние верифицируемой системы определяется набором состояний автоматов. Действительно, при одинаковом наборе состояний автоматов автоматная программа будет реагировать одинаково, если считать, что внешняя среда непредсказуема, и любое событие или значение переменной равновозможно в любой момент времени. Таким образом, глобальное состояние верифицируемой автоматной программы – это набор состояний автоматов, входящих в систему автоматов.

Из такого определения глобального состояния программы непосредственно вытекает определение элементарного шага работы автоматной системы – это обработка события, в результате которой может смениться 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 сложность верификации.

Мелкое дробление элементарных шагов могло бы понадобиться для того, чтобы более точно формулировать требования к автоматной программе. Например, рассмотрим переход вида «`e1/o1.z1,o1.z3`». Пусть требование такое, что после действия `z1` всегда должно выполняться действие `z2`. Оно выражается автоматом Бюхи, изображенным на рис. 23.

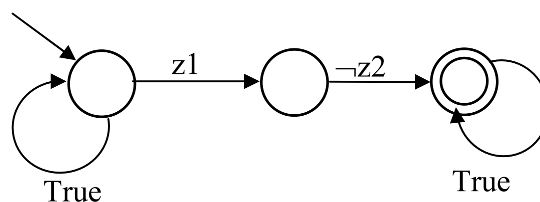

Рис. 23. Автомат Бюхи для формулы «$G\ (z1 \rightarrow X\ z2)$»

По схеме работы верификатора, при выполнении описанного перехода будет вызвано действие `z1`, а после этого будет вызвано действие `z3` (а не `z2`) – требование будет нарушено. Однако после перехода в верифицируемом автомате, автомат Бюхи сделает лишь один шаг, поскольку весь переход в автомате был выполнен атомарно. Автомат Бюхи не попадет в допускающее состояние, хотя должен был попасть. Это результат того, что требования описывают более мелкие действия, чем принятый элементарный шаг работы автоматной программы. В результате такого «большого» атомарного шага может измениться (и не раз) сразу несколько предикатов, участвующих в требованиях.

Однако описанная проблема возникает не так часто, поскольку обычно автоматная система проектируется таким образом, что переходы в автоматах достаточно мелкие, и интересующие разработчика утверждения описывают более глобальные свойства программы, чем те, которые изменяются в пределах одного перехода.

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.

При этом отметим, что для автоматной программы строго определена схема работы системы автоматов, определена последовательность передачи управления между автоматами, а система работает в одном потоке. Поэтому недетерминированность в работе автоматной программы возникает только в результате разных последовательностей входных событий и разных 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 ветвление в работе автоматной программы обрабатывается следующим образом. Перед совершением очередного элементарного шага у автоматной системы запрашивается набор событий, которые она 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 задает недетерминированность по событию: при возникновении разных событий из этого списка система поведет себя по-разному. Для того чтобы перебрать все возможные варианты, при каждом откате в текущее глобальное состояние верификатор будет подавать на обработку автоматной системе очередное еще не обработанное событие из списка.

В ходе обработки события может возникнуть необходимость вычислить значение условия на одном из переходов, и в этом условии 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 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.

При совершении шага автоматной программы, вся информация о происходящих действиях записывается и используется затем для вычисления значений предикатов, которые участвуют в требованиях. Предикаты применяются для формулировки 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. 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.

Эта информация, кроме того, позволяет однозначно определить набор выполненных переходов в автоматах.

Сохраняемая информация позволяет вычислять значения предикатов. В методе эмуляции используется 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`. Эквивалетно `(isInState(sm, s) && !wasInState(sm, s))`;

- `cameToFinalState()` – возвращает `True`, если после совершения шага корневой автомат модели вошел в свое конечное состояние. Это означает, что автоматная программа завершила работу.
- `wasAction(z)` – возвращает `True`, если в ходе выполнения шага было вызвано выходное воздействие `z`;
- `wasFirstAction(z)` – возвращает `True`, если в ходе выполнения шага первым вызванным действием было `z`;
- `wasLastAction(z)` – возвращает `True`, если в ходе выполнения шага последним вызванным действием было `z`;
- `getActionIndex(z)` – возвращает номер действия в списке действий, вызванных в ходе выполнения последнего шага. Этот предикат предназначен для того, чтобы формулировать утверждения, задающие порядок вызова действий объектов управления в автоматной программе;
- `wasTrue(g)` – возвращает `True`, если в ходе выполнения последнего шага один из переходов был помечен условием `g`, и его значение было определено как `True`. Пометка `g` описывает целое условие, а не значение одной переменной. Например, пометка `g = «!o1.x1 && o1.x2»`;
- `wasFalse(g)` – возвращает `True`, если в ходе выполнения последнего шага на одном из переходов было встречено условие `g`, и его значение было определено как `False`.

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`» будет сформулировано в виде:

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

Повторим еще раз решение каждой из пяти задач, необходимых для верификации автоматной программы по схеме, описанной в разд. 1.3.2:

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 за счет выбора всех возможных значений условий на переходах.
5. Предикаты описывают события, действия и значения переменных, полученные в последнем шаге.

1.3.4. Пример использования

Приведем пример применения метода эмуляции. Рассмотрим модель автоматной программы, управляющей работой дверей лифта. Для простоты автоматная модель состоит из единственного автомата, который изображен на рис. 24.

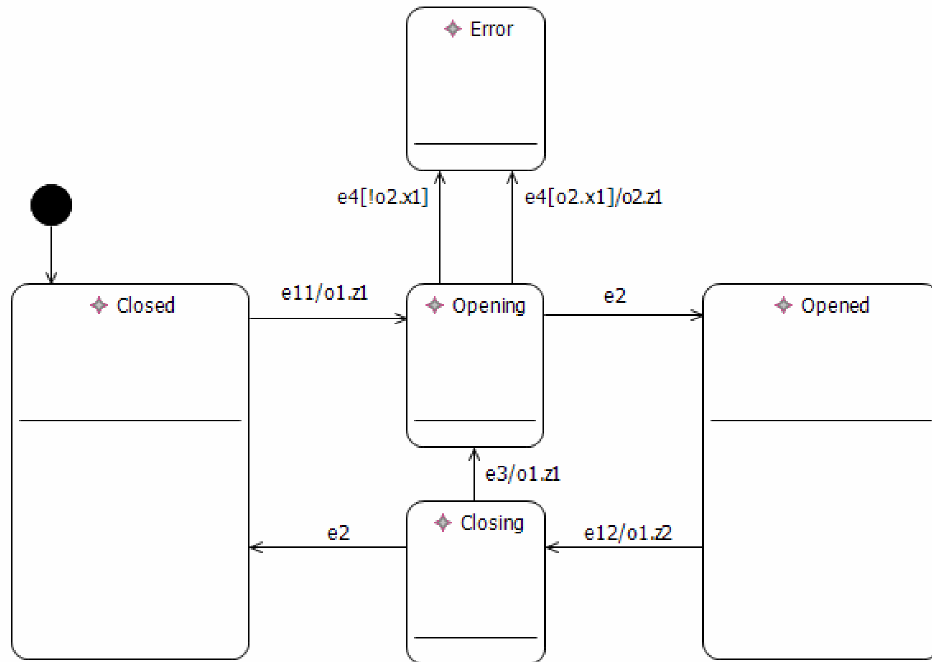

Рис. 24. Автомат, управляющий дверьми лифта

Работа начинается при закрытых дверях в состоянии `Closed` («Закрыты»). При нажатии кнопки «Открыть» (событие `e11`), запускается механизм открытия дверей (`o1.z1`) и автомат переходит в состояние `Opening` («Открываются»). При получении сообщения об успешном завершении открытия или закрытия (`e2`), автомат переходит в состояние `Opened` («Открыты»). Процесс закрытия происходит аналогичным образом. Здесь `e12` – нажатие кнопки «Закрыть», `o1.z2` – запуск закрытия дверей. Если какое-либо препятствие мешает дверям закрыться, то происходит событие `e3`, двери снова открываются, а автомат переходит в состояние `Opening`. Также при открытии дверей возможно возникновение ошибки (событие `e4`), что приводит автомат в состояние `Error` (Ошибка). При этом если включен механизм оповещения (`o2.x1`), то происходит звонок в аварийную службу (`o2.z1`). Состояние `Error` создано для демонстрации возможностей метода эмуляции. Поэтому, для простоты, в это состояние можно попасть только из состояния `Opening`.

Сформулируем следующее требование к автоматной программе: «Автомат никогда не попадет в состояние `Error`». Это требование описывается следующей темпоральной формулой: «$G\neg$ `isInState(A, Error)`», где `A` – название единственного автомата системы (для краткости название автомата будет опускаться, поскольку это единственный автомат в системе). Такая формула преобразуется в автомат Бюхи, изображенный на рис. 25.

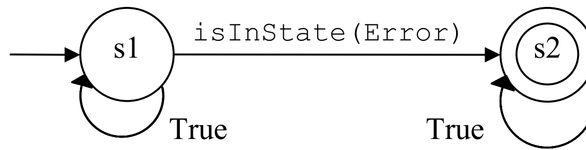

Рис. 25. Автомат Бюхи для формулы «*G*¬ isInState(Error)»

Верифицируем это утверждение, используя метод эмуляции. Первым состоянием верифицируемой системы является пара (Closed, *s1*). Это означает, что автомат дверей лифта находится в состоянии Closed, а автомат Бюхи – в состоянии *s1*. Эта пара размещается в стеке первого обхода в глубину. Из состояния Closed существует лишь один переход по событию e11. Так как этот переход единственный, то здесь возможен только один вариант работы автоматной программы. Событие e11 передается на обработку автомату. В результате обработки и перехода автомата формируется следующая информация:

Событие	e11
Действия	o1.z1
Условия	
Состояние до шага	Closed
Текущее состояние	Opening

После перехода в верифицируемом автомате, совершается переход в автомате Бюхи. Для этого необходимо вычислить значение предиката isInState(Error). Автомат находится в состоянии Opening. Поэтому значение предиката – False. В автомате Бюхи активен лишь один переход – петля в состояние *s1*. Таким образом, в результате выполнения элементарного шага получаем состояние верифицируемой системы (Opening, *s1*). Это состояние помещается в стек. Запишем состояния стека, выбранное событие и переход в автомате Бюхи:

(Closed, *s1*)	Event=(**e11**), Buechi=(***s1*→*s1***)
(Opening, *s1*)	

(метка Event используется для обозначения выбранного события, а метка Buechi – для обозначения перехода, выполненного в автомате Бюхи).

В новом состоянии верифицируемого автомата обрабатываются два события: e4 и e2. Выбираем первое из них, и записываем информацию о наборе возможных событий и выбранном событии напротив текущего состояния в стеке. При откате в это состояние будет запущен снова шаг вперед по истории с еще не использованным событием. По событию e4 есть два перехода. Первый переход ограничен условием !o2.x1. Выбираем значение False и записываем информацию о сделанном выборе. Новый стек выглядит следующим образом:

(Closed, *s1*)	Event=(**e11**), Buechi=(***s1*→*s1***)
(Opening, *s1*)	Event=(**e4**, e2); !o2.x1=(**False**, True)

(жирным помечены выбранные варианты).

Условие на переходе не выполняется. Поэтому выбирается другой переход по событию e4. Он помечен условием o2.x1. Пусть его значение – False. Тогда получается противоречие: во время выбора активного перехода автомата не могут одновременно принимать значения False и !o2.x1, и o2.x1. Поэтому значение o2.x1 автоматически принимается за True, и ветвления истории не происходит. Автомат дверей лифта переходит в новое состояние. В результате создается следующая информация о пройденном шаге:

Событие	e4
Действия	o2.z1
Условия	!o2.x1=False; o2.x1=True
Состояния до шага	Opening
Текущие состояния	Error

Далее совершается переход в автомате Бюхи. Значение предиката isInState(Error) вычисляется как True, и возможны два перехода в автомате Бюхи. Пусть снова выбирается переход-петля. При этом информация о сделанном выборе записывается. Автомат Бюхи совершает переход, и новое состояние верифицируемой системы – (Error, *s1*). После этого стек с записанными результатами выбора выглядит следующим образом:

(Closed, *s1*)	Event=(**e11**), Buechi=(***s1→s1***)
(Opening, *s1*)	Event=(**e4**, e2); !o2.x1=(**False**, True); Buechi=(***s1→s1***, *s1→s2*)
(Error, *s1*)	

В новом состоянии автомат дверей лифта не обрабатывает ни одного события. Поэтому он остается в том же состоянии, и выполняется шаг автомата Бюхи. По-прежнему активны два перехода. Пусть снова выбирается переход-петля. Информация об этом записывается к текущему состоянию системы, и выполняется переход. Система оказывается в уже посещенном состоянии (Error, *s1*):

(Closed, *s1*)	Event=(**e11**), Buechi=(***s1→s1***)
(Opening, *s1*)	Event=(**e4**, e2); !o2.x1=(**False**, True); Buechi=(***s1→s1***, *s1→s2*)
(Error, *s1*)	Buechi=(***s1→s1***, *s1→s2*)
(Error, *s1*)	

По алгоритму поиска в глубину происходит откат в предыдущее состояние. В нем были произведены не все возможные выборы. Поэтому снова делается шаг вперед с другим переходом автомата Бюхи. При этом верифицируемая система оказывается в состоянии (Error, *s2*):

(Closed, *s1*)	Event=(**e11**), Buechi=(***s1→s1***)
(Opening, *s1*)	Event=(**e4**, e2); !o2.x1=(**False**, True); Buechi=(***s1→s1***, *s1→s2*)
(Error, *s1*)	Buechi=(*s1→s1*, ***s1→s2***)
(Error, *s2*)	

Это состояние системы является допускающим, поскольку состояние *s2* автомата Бюхи – допускающее. По алгоритму двойного поиска в глубину, из текущего состояния запускается вложенный поиск в глубину. В текущем состоянии системы в верифицируемом автомате нет

активных переходов, а в автомате Бюхи он только один. После выполнения этого перехода стек вложенного поиска в глубину выглядит следующим образом:

(`Error`, *s2*)	Buechi=(***s2***→***s2***)
(`Error`, *s2*)	

Вложенный поиск в глубину привел в одно из состояний, содержащихся в стеке первого поиска в глубину. Это означает, что найден допускающий путь. В стеках поисков в глубину хранится контрпример. В стеке первого поиска в глубину хранится путь до допускающего состояния (`Error`, *s2*), а в стеке вложенного поиска в глубину – цикл, проходящий через это состояние. В рассматриваемом примере он состоит из единственного состояния. Графическое представление контрпримера изображено на рис. 26.

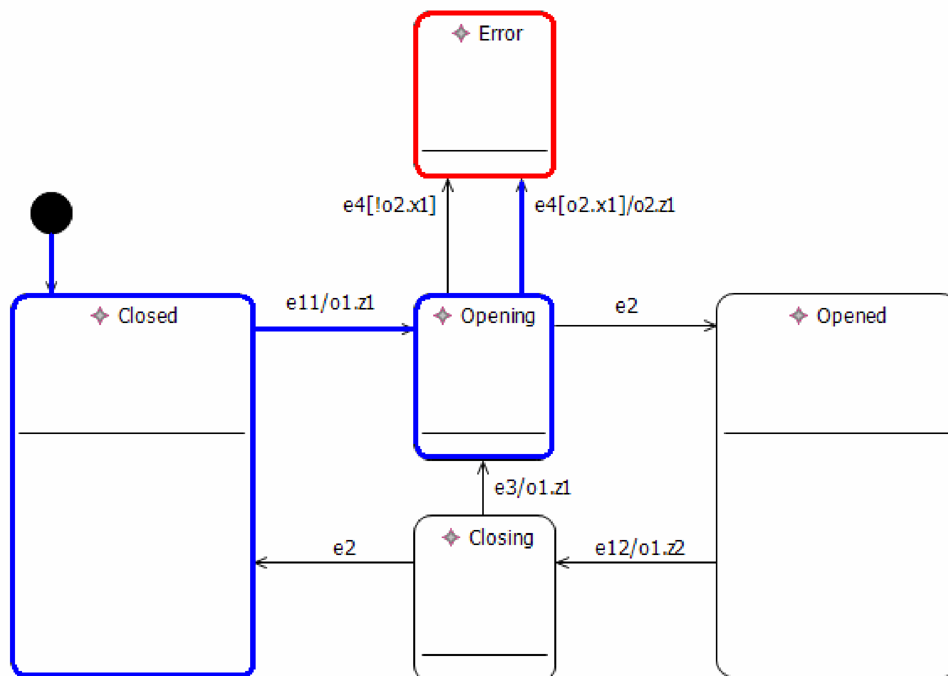

Рис. 26. Контрпример для утверждения «*G*¬ `isInState(Error)`»

Таким образом, верифицируемое утверждение о том, что автомат никогда не попадет в состояние `Error`, нарушается следующим образом. Автомат в начальном состоянии получает событие `e11`, затем происходит событие `e4` при условии, что `o2.x1` равно `True`, и автомат оказывается в состоянии `Error`.

Полученный контрпример является не единственным. Все возможные контрпримеры можно получить, если продолжить верификацию, прогоняя работу системы со всеми возможными выборами в недетерминированных ситуациях.

Отметим, что запись в стек состояний системы, выбранных событий и значений переменных позволяет получить контрпример в терминах исходного автомата, без каких либо преобразований или анализа.

1.3.5. Модель Крипке

Определив правила, по которым верификатор работает с автоматной программой, неявно определена модель Крипке для автоматной модели. Опишем ее структуру.

Состояние системы автоматов кодируется набором состояний всех автоматов, а обработка событий происходит атомарно. Это означает, что, например, для системы из одного автомата, граф автомата будет совпадать с графом модели Крипке. Действительно, каждое состояние автомата является состоянием модели Крипке, а других состояний в модели Крипке нет, поскольку переходы между состояниями автомата происходят атомарно. Однако при таком подходе возникает следующая проблема.

Как известно, в каждом состоянии модели Крипке существует список предикатов, которые в нем выполняются. В автоматной программе используются не только предикаты, характеризующие состояние автомата, но и такие, как `wasEvent` – предикаты, характеризующие переход, по которому автомат попал в текущее состояние. Однако в одно и то же состояние автомата может вести несколько переходов. Следовательно, в разные моменты времени в этом состоянии могут, как выполняться, так и не выполняться некоторые предикаты. Таким образом, и модель Крипке с тем же набором состояний, что и у автомата, строго говоря, не является моделью Крипке, поскольку в одном состоянии у нее в разные моменты времени предикаты могут иметь различные значения. Эта идея изображена на примере автомата на рис. 27.

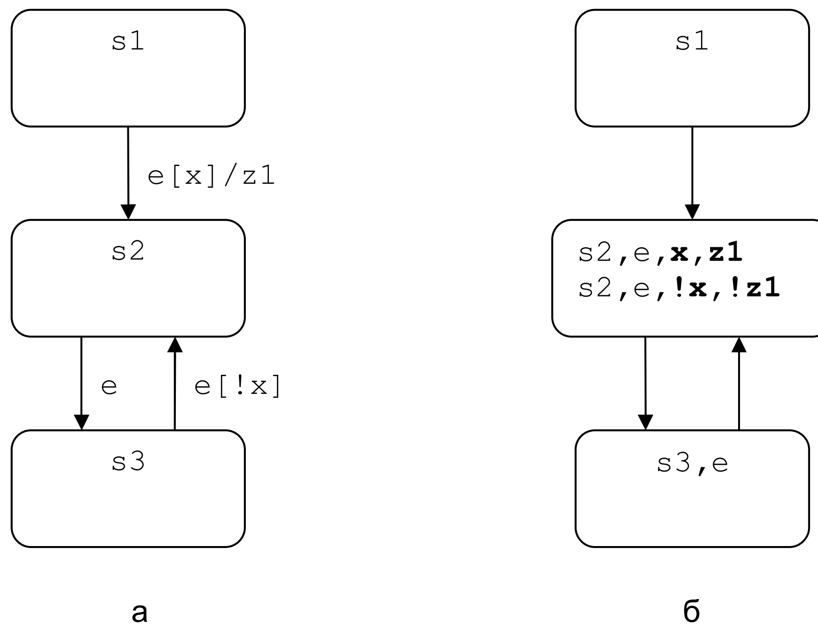

Рис. 27. Исходный автомат (а) и полученная модель Крипке (б). Жирным отмечены конфликтующие предикаты.

На рисунке слева изображен исходный автомат, а справа – построенная по нему модель Крипке. В состояниях модели Крипке сокращенно записаны предикаты, которые в нем выполняются. Смысл предикатов следующий:

- `s` – `isInState(s)`;
- `x` – `wasTrue(x)`;
- `z` – `wasAction(z)`.

Для состояния, соответствующего состоянию `s2` автомата, в двух строках записаны предикаты, выполняющиеся в разные моменты времени: когда автомат попал в состояние `s2` из состояния `s1` или из состояния `s3`. Жирным шрифтом помечены предикаты, которые в этих случаях принимают различные значения.

Таким образом, выходит, что для метода эмуляции не получается построить правильную модель Крипке. Возникает вопрос, почему тогда описанный метод верификации автоматных программ работает.

Для ответа на этот вопрос вспомним алгоритм верификации, который используется для верификации темпоральных формул *LTL*. Модель Крипке преобразуется в *автомат*, который потом пересекается с автоматом Бюхи, построенным по темпоральной формуле. При преобразовании модели Крипке в автомат получается автомат Мура [2]. Это означает, что выполняющиеся предикаты на переходах зависят только от конечного состояния перехода. Стандартная схема верификации – это построение по программе модели Крипке, преобразование ее в автомат (назовем этот автомат автоматом Крипке) и пересечение полученного автомата с автоматом Бюхи.

В рассматриваемом методе отсутствует этап построения модели Крипке по программе, и сразу неявно строится автомат Крипке, который затем пересекается с автоматом Бюхи. Заметим, что при этом автомат Крипке оказывается автоматом Мили. Однако при этом алгоритм двойного поиска в глубину, использующийся для поиска контрпримера, не перестает работать.

В случае если верифицируемая система состоит из одного автомата, то этот автомат непосредственно является необходимым автоматом Крипке (при замене действий и запросов переменных на соответствующие предикаты), который можно использовать в алгоритме двойного поиска в глубину. Систему автоматов также можно непосредственно использовать в качестве автомата Крипке. Для этого необходимо помечать глобальные состояния системы (характеризующиеся набором состояний автоматов) предикатами так же, как это делалось для одного автомата. Это можно делать прямо в ходе интерпретации работы автоматной программы (эмуляции).

На рис. 28 схематично изображен обход глобальных состояний автоматной программы при использовании метода эмуляции.

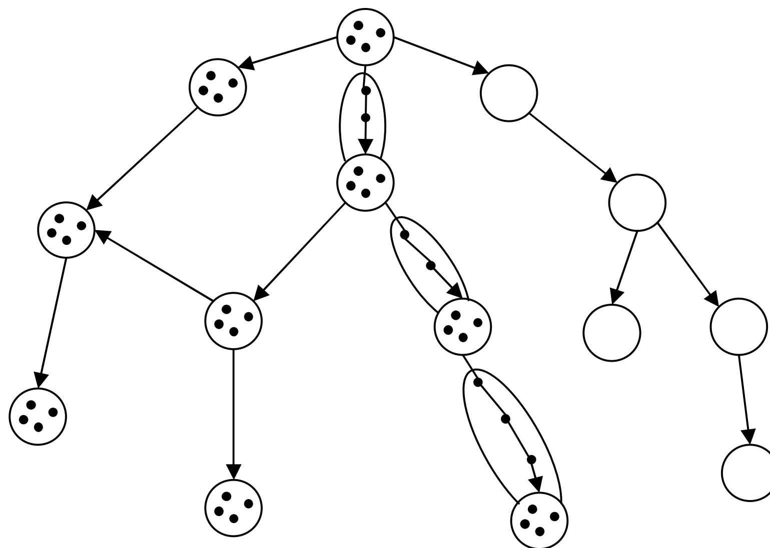

Рис. 28. Схематичное изображение обхода автоматной программы

На рисунке вершины с точками изображают глобальное состояние системы. Стрелки между ними соответствуют глобальному переходу системы при обработке одного события. При этом

показано, что каждый такой переход может состоять из нескольких переходов в нескольких автоматах, однако весь этот набор действий рассматривается как единое целое.

После каждого шага (обработки одного события) фиксируется глобальное состояние системы, и набор выполненных в ходе перехода предикатов. В результате динамически формируется состояние автомата модели Крипке, и набор предикатов, выполняющихся при входе в это состояние, что и используется потом в алгоритме верификации. При этом оказывается все равно, происходит работа с одним автоматом или с системой автоматов, поскольку процесс вычисления предикатов происходит динамически и не требуется предварительного анализа системы.

1.4. Выводы

Метод построения модели Крипке по автоматной модели управляющей программы строит модель Крипке в явном виде, но за счет учета специфики автоматных программ простроенная модель имеет меньший размер по сравнению с моделью Крипке, построенной традиционным образом.

Метод верификации визуальных моделей автоматных программ позволяет верифицировать автоматные программы без привязки к конкретному языку программирования. В этом методе модель Крипке строится верификатором на основе промежуточной модели автоматной программы, подаваемой ему на вход.

Метод верификации на основе эмуляции позволяет верифицировать автоматные программы с помощью построения неявной модели Крипке и преобразования ее в автомат Крипке. Такая замена становится возможной за счет специфики автоматной модели и метода двойного обхода в глубину.

Для разработанных методов требуется провести анализ функциональных особенностей и характеристик, а также условий и ограничений функционирования. Это позволит определить их область применимости. Для исследования эффективности предложенных методов на практике необходимо создать прототипы программных средств.

2. ФУНКЦИОНАЛЬНЫЕ ОСОБЕННОСТИ И ХАРАКТЕРИСТИКИ МЕТОДОВ ВЕРИФИКАЦИИ АВТОМАТНЫХ МОДЕЛЕЙ

В данной главе рассматриваются вопросы применимости разработанных методов верификации автоматных моделей управляющих программ, в частности, определяются их функциональные особенности (разд. 2.1), функциональные особенности методов восстановления контрпримеров (разд. 2.2), характеристики методов (разд. 2.3), а также условия и ограничения их функционирования (разд. 2.4). Анализ этих вопросов позволит сформулировать области применения разработанных алгоритмов.

2.1. Функциональные особенности методов верификации автоматных моделей

В данном разделе рассматриваются функциональные особенности методов верификации автоматных программ, разработанных в главе 1.

2.1.1. Метод построения модели Крипке по автоматной модели

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 узнавать о состояниях других автоматов.
- Один автомат может передавать управление другому автомату, предавая ему событие.

На переходе автомата может находиться:

- 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 in sij` (оно истинно, если автомат `Ai` находится в состоянии `sij)`.
- Последовательность действий. Она может содержать выходные воздействия `o.zi()` и передачу управления другим автоматам `Ai.ej()`.

2.1.2. Метод верификации визуальных автоматных моделей

Метод верификации визуальных автоматных моделей абстрагируется от переменных в автоматах, оставляя на переходах события. Поддерживается одно событие на переходе. Кроме того, метод абстрагируется от выходных воздействий. На этом уровне абстракции автоматы *Мура* и *Мили* [18] неотличимы друг от друга, поэтому метод принимает оба типа автоматов.

Метод верификации визуальных автоматных моделей также поддерживает верификацию систем, состоящих из вложенных автоматов. Модель каждого автомата на языке *Promela* записывается в отдельной функции. Тогда вызов вложенного автомата будет эквивалентен в модели на языке *Promela* вызову функции, в которой записана модель автомата.

2.1.3. Метод верификации автоматных моделей на основе эмуляции

Метода эмуляции позволяет верифицировать темпоральные утверждения *LTL*-логики для систем иерархически связанных автоматов. Структура системы автоматов была описана в разд. 1.3.1. Напомним, что автоматы связываются между собой путем вложения дочерних автоматов в состояния родительских автоматов. Событие обрабатывается сначала родительским автоматом, а затем передается на обработку автоматам, вложенным в текущее состояние родительского автомата. Кроме того, в родительском автомате на переходе может явно вызываться обработка заданного события в

дочернем автомате. Также состояние дочернего автомата может использоваться в условиях на переходах родительского автомата.

Система автоматов работает в одном потоке и обрабатывает события по одному. Система не начинает обработку нового события, пока не завершена обработка предыдущего.

Каждый автомат в системе может быть смешанным автоматом. Переходы в автоматах помечаются событием и условием, при которых переход активируется, и списком действий, выполняемых при переходе. Среди действий могут быть вызовы действий в объектах управления, а также команды на обработку заданных событий в дочерних автоматах. В условиях на переходах могут использоваться:

1. Переменные объектов управления, возвращающие булево значение (`True` или `False`).
2. Переменные объектов управления, возвращающие целое значение. Такие переменные используются в виде сравнения с константой или другой переменной (например, «`(o1.x1 > 0) & (o1.x1 < o1.x2)`»).
3. Проверка, что дочерний автомат находится в указанном состоянии (например, условие «`A1.s2`» выполнено, когда дочерний автомат `A1` находится в состоянии `s2`).

Приведем пример перехода, помеченного следующим образом:

```
e1 [o1.x1 & (o1.x2 > 2) & A2.s1] / o1.z1, A2.e3
```

Эта пометка означает, что переход активируется при возникновении события `e1`, и при условии, что переменная `x1` объекта управления `o1` равна `True`, переменная `x2` больше двух, а автомат `A2` находится в состоянии `s1`. При переходе выполняется действие `z1` объекта `o1`, а также запускается обработка события `e3` в автомате `A2`.

Утверждения для верификации формулируются в виде формулы *LTL*-логики. При этом используются стандартные логические операторы и темпоральные *LTL*-операторы, как описано в разд. 1.3.2. При формулировке темпорального утверждения следует помнить, что в методе эмуляции один элементарный шаг – это обработка события системой автоматов. Таким образом, например, если два предиката выполнились в пределах одного перехода, то считается, что они выполнились одновременно. Также из этого следует, что темпоральный оператор **X** («в следующий момент времени») означает «при обработке следующего события».

Для формулирования требований используются предикаты, описывающие:

1. Возникающие события.
2. Состояния автоматов.
3. Вызываемые действия объектов управления.
4. Вычисленные значения условий на переходах.

Список предикатов приведен в разд. 1.3.3.

Например, формула

```
G((wasEvent(e1) & wasTrue(!o1.x1)) → isInState(A1,s2))
```

означает, что всегда при возникновении события `e1`, если условие `!o1.x1` выполнено, то автомат `A1` окажется в состоянии `s2` (после обработки события).

Стоит заметить, что 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.

К особенностям метода эмуляции можно отнести также следующее. В работе [1] верификация системы автоматов производится по следующей схеме:

1. Исходная система автоматов преобразуется в модель Крипке (записанную на входном языке существующего верификатора).

2. Полученная модель верифицируется существующим верификатором, который выполняет над ней стандартный алгоритм верификации *LTL*-формул (двойной обход в глубину).

В результате этого возникают следующие проблемы. Во-первых, при преобразовании верифицируемой программы возникает необходимость преобразовать и требования к ней. При этом необходимо, чтобы выполнение исходных требований в исходной программе было эквивалентно выполнению преобразованных требований в преобразованной программе. Во-вторых, в случае нарушения требований, сценарий ошибки будет записан в терминах преобразованной программы. Возникает необходимость преобразовать сценарий в термины исходной системы автоматов.

Таким образом, для решения исходной задачи (заданы система автоматов и требования к ней, и необходимо получить сценарий поступления событий, при котором нарушаются требования) путем описанных преобразований, требуется выполнить следующее:

1. Преобразовать исходную систему автоматов в модель Крипке, записанную на входном языке существующего верификатора.
2. Преобразовать исходные требования в термины полученной модели.
3. Подать полученную модель и требования на вход верификатора.
4. В случае неудачной верификации, полученный от верификатора сценарий ошибки в преобразованной модели необходимо преобразовать обратно в сценарий работы исходной системы автоматов.

Описанная схема верификации изображена на рис. 29.

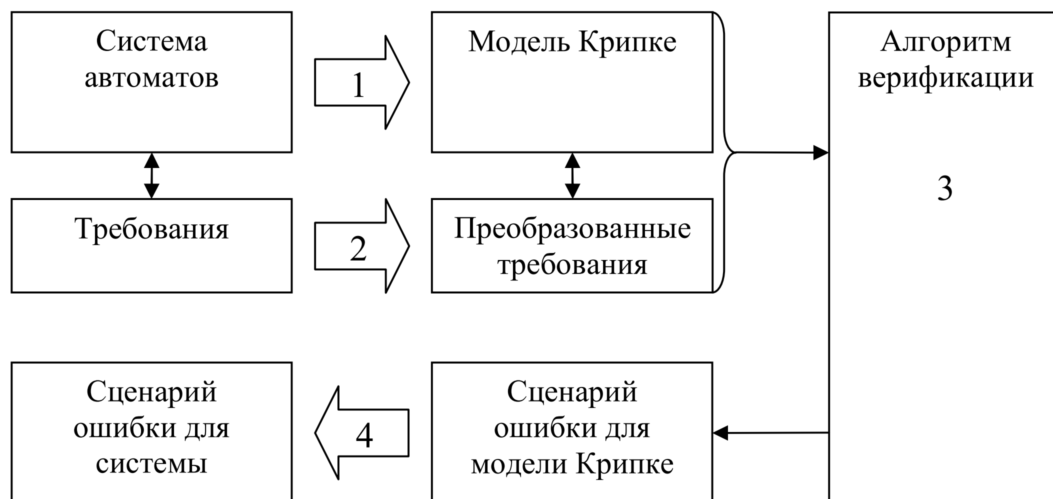

Рис. 29. Схема верификации с явным построением модели Крипке

2.2. Функциональные особенности методов преобразования контрпримеров

В данном разделе рассматриваются функциональные особенности методов преобразования контрпримеров из терминов модели Крипке в термины автоматных программ.

2.2.1. Метод верификации автоматных программ с использованеим верификатора NuSMV

При использовании метода верификации автоматных программ с использованием верификатора *NuSMV* восстановление контрпримера не представляет существенных трудностей, так

как для каждого состояния известен его прообраз (прообразы, в случае системы автоматов) и на каждом шаге происходит изменение состояния ровно одного автомата. Таким образом, контрпример в терминах автоматной модели восстанавливается конструктивно и единственным образом.

2.2.2. Метод верификации визуальных автоматных моделей

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

2.2.3. Метод верификации на основе эмуляции

В методе эмуляции не строится явно модель Крипке, и алгоритм верификации работает напрямую с системой автоматов. В результате не требуется выполнение лишних преобразований, и требуется только выполнить работу алгоритма верификации. Таким образом, схема верификации при использовании предлагаемого метода выглядит, как изображено на рис. 30.

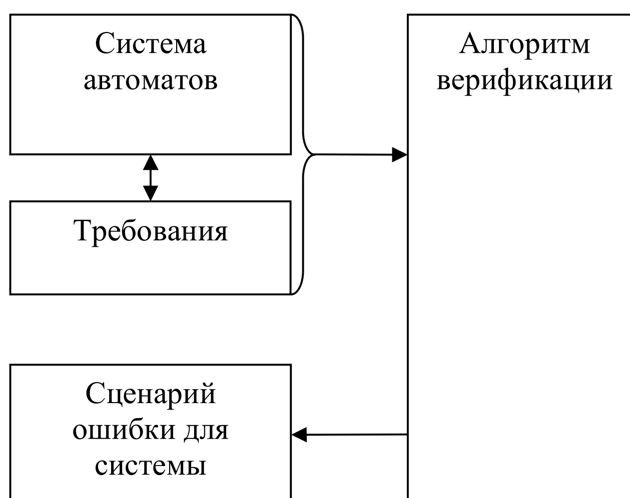

Рис. 30. Схема верификации предлагаемым методом

2.3. Характеристики методов верификации автоматных моделей

В данном разделе рассматриваются характеристики методов верификации автоматных моделей, разработанных в главе 1.

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

Достоинством метода верификации автоматных программ с использованием верификатора *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 важной характеристикой является число состояний модели Крипке в зависимости от размеров системы. Так как в рассматриваемом методе структура Крипке непосредственно не строится, а лишь приходится преобразовывать автоматную модель в модель на языке *SMV*, число вершин в структуре Крипке может зависеть от конкретной реализации верификатора для языка *SMV*. Будем считать, что число состояний в структуре Крипке - это произведение числа значений всех переменных в системе. В модели используются следующие переменные: `yk`, `Operation_k`, `Active_k`, `(0 ≤ k ≤ n)`, где `n` - число автоматов, и `xk(0 ≤ k < m)`, где `m` – число входных переменных. `Event` (для всей системы в целом). Рассмотрим рост числа значений каждой из переменных в зависимости от параметров системы автоматов:

- число значений принимаемых переменной `yk` равно числу состояний автомата `Ak`;
- число значений `Operation_k` растет пропорционально числу переходов автомата `Ak`;
- число значений `Active_k` равно числу автоматов в системе;
- `xk` может принимать два значения – {0, 1};
- `Event` может принимать число значений не больше числа всех событий в системе.

Из перечисленных переменных можно выделить наиболее быстро растущую часть (`yk * Operation_k`) – это выражение растет пропорционально числу переходов автомата. Следовательно, можно считать, что число состояний в структуре Крипке будет расти пропорционально произведению числа переходов всех автоматов.

2.3.2. Метод верификации визуальных автоматных моделей

Проанализируем скорость работы алгоритма создания модели визуальной автоматной программы на языке *Promela*.

Введем обозначения. Обозначим переменной a число автоматов, переменной i – номер текущего автомата, переменной j – номер текущего состояния в автомате.

Для каждого автомата с номером i обозначим переменной s_i число состояний в этом автомате, а переменной s – общее число состояний в автоматной программе. Для каждого состояния j в автомате i обозначим переменной p_{ij} число переходов, которые исходят из этого состояния, переменной p_i – число переходов в автомате i, а переменной p – общее число переходов между состояниями в программе. Для каждого перехода с номером k обозначим переменной e_{ijk} число событий на данном переходе (e_{ijk} равно либо нуля, либо единице). Во всех состояниях j переменной e_{ij} обозначим число событий на всех переходах, которые исходят из этого состояния. Переменная e_i обозначает число событий на всех переходах в автомате i, а e – общее число событий на переходах в программе. Таким образом, каждое событие будет посчитано столько раз, на скольких переходах оно используется.

Для каждого автомата i согласно предлагаемому алгоритму генерируется отдельная функция за константное время. В каждое состояние j автомата алгоритм построения модели заходит ровно один раз. Алгоритм генерирует общий код для состояния за время $\Theta(1)$ и обрабатывает все переходы, которые ведут из этого состояния. Генерация кода для каждого перехода k из состояния занимает время $\Theta(1)$ плюс время, требуемое на обработку всех событий. Для обработки одного события требуется время $\Theta(1)$. Таким образом, для обработки одного перехода требуется время $\Theta(e_{ijk})$. Тогда на обработку одного состояния требуется время $\Theta(\sum_k (1+e_{ijk})) = \Theta(p_{ij}+e_{ij})$. Исходя из этого, время, требуемое на обработку одного автомата, пропорционально $\sum_j (1+\Theta(p_{ij}+e_{ij})) = \Theta(s_i+p_i+e_i)$, а

время, требуемое на обработку всей программы, пропорционально $\sum_i (1 + \Theta(s_i + p_i + e_i)) = \Theta(a + s + p + e)$.

Таким образом, время работы алгоритма создания модели автоматной визуальной программы на языке *Promela* линейно:

- по числу автоматов;
- по общему числу состояний в автоматной программе;
- по общему числу переходов между состояниями в программе;
- по числу событий на переходах.

Определим время работы алгоритма преобразования *LTL*-формул. Алгоритм проходит по всей формуле один раз и для каждой последовательности символов в фигурных скобках *{expression}* с номером *k* делает следующее:

- в модель пишет определение `#define pk (expression);`
- заменяет выражение `{expression}` на идентификатор `pk` в формуле.

Каждое выражение `{expression}` обрабатывается за время, пропорциональное его длине. Следовательно, время, которое требуется для работы алгоритма преобразования *LTL*-формул, пропорционально длине строки с формулой.

Таким образом, метод обеспечивает линейный рост модели на языке *Promela*.

2.3.3. Метод верификации на основе эмуляции

При верификации системы автоматов с помощью предлагаемого метода, алгоритм двойного обхода в глубину работает с системой автоматов. Каждый шаг работы – это обработка системой события.

В худшем случае алгоритм двойного обхода в глубину работает за $O((E + N)^2)$, где N – число вершин в графе обхода, а E – число ребер. В нашем случае, N – это число глобальных состояний верифицируемой системы, а E – число переходов в ней. В методе эмуляции глобальное состояние верифицируемой системы складывается из глобального состояния системы автоматов и состояния автомата Бюхи. Глобальное состояние системы автоматов, в свою очередь, задается набором состояний автоматов системы. Тогда оценка сверху на число состояний системы автоматов – это произведение числа состояний автоматов системы. В свою очередь, оценка сверху на число состояний верифицируемой системы – это произведение числа состояний автоматов и числа состояний автомата Бюхи.

Число состояний автомата Бюхи зависит от сложности исходной *LTL*-формулы и может быть крайне большим для сложных формул.

В каждом состоянии верифицируемой системы число возможных глобальных переходов равно произведению числа возможных переходов в системе автоматов на число возможных переходов автомата Бюхи.

Таким образом, сложность верификации зависит от сложности *LTL*-формулы и размеров системы автоматов. Кроме того, следует заметить, что при каждом шаге верификации осуществляется обработка события. Поэтому скорость верификации напрямую зависит от скорости работы интерпретатора автоматной программы.

В работе [1] переходы в автоматах дробятся на элементарные шаги, поэтому число состояний системы автоматов сильно увеличивается. Действительно, получается, что для каждого перехода в автомате выделяется целый набор дополнительных промежуточных состояний. Это может на порядки увеличить число глобальных состояний системы автоматов, и, следовательно, значительно увеличить и сложность верификации.

В методе эмуляции такого дробления не производится, что позволяет рассчитывать на более быструю верификацию.

2.4. Условия и ограничения функционирования методов верификации автоматных моделей

2.4.1. Метод построения модели Крипке по автоматной модели

Для задания спецификации системы используется формулы логики *CTL*. В спецификации данный метод позволяет задавать следующие свойства состояний системы:

- В каком состоянии будет находиться автомат или несколько автоматов.
- На какой дуге будет находиться вызываемый автомат.
- Какими будут значения входных переменных при выполнении перехода.

В силу того, что требования представляются в виде формул логики *CTL*, алгоритм не позволяет проверять условия на конкретный путь исполнения, как использовании логики *LTL*.

2.4.2. Метод верификации визуальных автоматных моделей

Метод может быть применен только там, где может быть применен верификатор *SPIN*.

Учитывая рассмотренные выше характеристики работы алгоритмов (линейное время работы), ограничения метода связаны с ограничением верификатора *SPIN* на размер модели: если подать на вход верификатору *SPIN* слишком большую модель, то верификатор не сможет ее обработать и выдаст сообщение, что размер модели слишком велик. В результате, на основе эксперимента было установлено, что метод позволяет верифицировать визуальные автоматные программы примерно с 550 состояниями, если из каждого состояния исходит один переход, и примерно с 210 состояниями, если из каждого состояния исходят, в среднем, пять переходов.

2.4.3. Метод верификации на основе эмуляции

Поскольку в соответствии с предлагаемым методом обработка событий в системе происходит атомарно, то относительно темпоральной формулы все, что происходит в пределах одной обработки события – происходит одновременно. Это накладывает определенное ограничение на выразительность формулы, а, следовательно, и на выразительность требований, которые можно сформулировать для системы автоматов. Ограничение заключается в том, что если для выполнения требований важен порядок возникновения некоторых условий (выполнения предикатов), то эти условия не должны выполняться в пределах обработки одного события.

Например, пусть требование состоит в том, чтобы всегда после действия `z1` выполнялось действие `z2`. Тогда нельзя это требование представить формулой «**G**`(z1 → `**X**` z2)`», поскольку оператор **X** означает «при обработке следующего события», а не «следующим действием». Для возможности формулировки таких требований для действий объектов управления был введен предикат `getActionIndex`. С его помощью описанное требование будет сформулировано следующим образом:

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

Другое ограничение, возникающее при использовании предлагаемого метода, связано с проблемой логики в объектах управления и источниках событий. Опишем это ограничение.

В данной работе 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.

Приведем простой пример. Пусть несколько однотипных автоматов (рис. 31) используют один ресурс, к которому необходим исключительный доступ: в каждый момент времени лишь один автомат должен использовать ресурс. И пусть ресурс представлен в виде объекта управления.

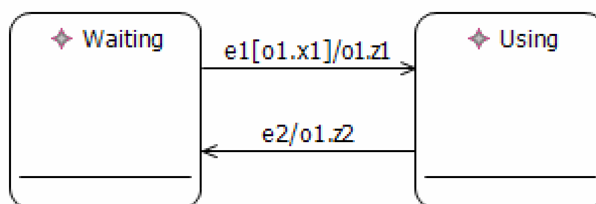

Рис. 31. Простой автомат, использующий «ресурс»

Объект управления `o1` имеет внутреннюю булеву переменную, которая символизирует, является ли ресурс в данный момент свободным. Значение ее возвращается переменной `o1.x1`. Также определены действия: «занять ресурс» (`o1.z1`) и «освободить ресурс» (`o1.z2`), которые соответственно изменяют внутреннюю переменную. Ясно, что если один из автоматов занял ресурс, вызвав `o1.z1`, то ни один другой автомат не перейдет в состояние `Using`, поскольку не будет выполнено условие `o1.x1`.

Однако при верификации такая логика теряется: нет информации о том, что если было выполнено действие `o1.z1`, то `o1.x1` возвращает `False`. В результате при верификации возникнет ситуация, когда несколько автоматов одновременно попадают в состояние `Using`, хотя это невозможно в исходной модели. Если при этом будет нарушено верифицируемое требование, то будет получена наведенная ошибка верификации – ошибка, которая невозможна в реальной реактивной системе, но была найдена при верификации.

Кроме такого простого примера с внутренней булевой переменной в объекте управление, сложнее дело обстоит, если у объекта управления есть переменные типа `int`. В этом случае число внутренних состояний объекта управления становится практически необозримым.

То же касается и источников событий. Возможно, например, что в реальной системе события могут возникать только в определенной последовательности.

Проблема объектов управления с внутренней логикой, влияющей на переходы автомата, заключается не только в появлении фиктивных историй. Важнее то, что в этом случае при верификации больше не работает предпосылка о том, что одинаковые состояния автоматов порождают одинаковое поведение реактивной системы, так как поведение системы будет определяться также и состояниями объектов управления и источников событий.

Для решения описанной проблемы можно применять следующие идеи.

- Если промоделировать объект «Ресурс» в виде автомата, проблема решается сама собой. Логика объекта управления добавляется в верифицируемую модель в виде нового автомата из двух состояний: свободен и не свободен. При этом вместо условий на переменную будут использоваться условия на состояния нового автомата.
- Ограничения на истории развития можно добавлять в верифицируемую формулу, в виде:

  `Ограничение1 & Ограничение2 & … → Требование`

Таким образом, логика объектов управления переносится в автомат Бюхи, и проблема решается (вспомним, что состояние системы при верификации складывается из состояний автоматов и состояния автомата Бюхи). Однако сформулировать такое ограничение в терминах темпоральной логики может оказаться сложно. Даже для такой простой логики, как в объекте «Ресурс», вывод темпоральной формулы нетривиален. Формула, описывающая поведение ресурса («если ресурс заняли, то он не свободен, пока его не освободят»), выглядит следующим образом:

```
o1.x1 W o1.z1 &
G (o1.z2 → (o1.x1 W o1.z1) &
   o1.z1 → (¬o1.x1 W o1.z2))
```

Из этого примера следует, что даже достаточно простая логика выражается нетривиальной формулой.

2.5. Выводы

Метод верификации автоматных программ с использованием верификатора *NuSMV* позволяет проводить верификации требований, записанных в терминах логики *CTL*.

Метод верификации визуальных автоматных моделей позволяет верифицировать автоматные модели управляющих программ без ограничения на сложность объектов управления. При этом поведение объекта правления должно быть описано в терминах *LTL*.

Метод верификации на основе эмуляции должен позволить верифицировать автоматные модели управляющих программ, которые не содержат логику в объектах управления. При этом автоматная модель может состоять из нескольких взаимодействующих автоматов.

3. ТЕХНОЛОГИЯ ВЕРИФИКАЦИИ АВТОМАТНЫХ МОДЕЛЕЙ УПРАВЛЯЮЩИХ ПРОГРАММ

В данной главе рассматриваются прототипы дополнительных модулей к верификаторам, реализующих разработанные методы. Прототипы позволят определить применимость этих методов на практике.

Программное средство *SMVVerifier*, реализующее метод верификации автоматных программ и являющееся дополнительным модулем к верификатору *NuSMV*, описано в разд. 3.1.

В разд. 3.2 рассматривается программное средство *Unimod.verifier*, созданное на основе метода верификации визуальных моделей автоматных программ. Это средство является дополнительным модулем к верификатору *SPIN*.

Дополнительный модуль *Converter* к верификатору *Bogor*, реализующий на метод верификации на основе эмуляции, описан в разд. 3.3.

3.1. SMVVERIFIER

SMVVerifier — это программное средство, которое позволяет производить верификацию системы автоматов. На вход программного средства дается *XML* описание системы автоматов и требования к этой системе. Средство возвращает информацию, что она удовлетворяет требованиям, а в противном случае — контрпример.

3.1.1. Модель программного средства

Схема работы программы показана на рис. 2. В качестве программы верификатора используется программа *NuSMV*.

Средство написано на языке *Java* с использованием классов изображенных на рис. 32.

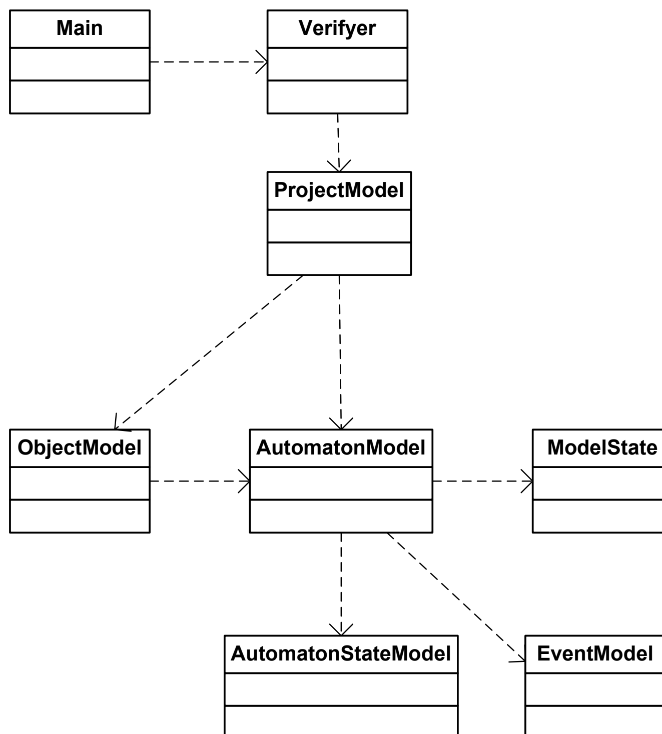

Рис. 32. Диаграмма классов программного средства

Данные классы выполняют следующие функции:

- `Main` – главный класс программы, обрабатывает параметры командной строки, вызывает класс `Verifyer`;
- `Verifyer` – класс, выполняющий функции верификации: читает программу из файла, строит её модель, запускает верификатор и обрабатывает полученный контрпример;
- `ProjectModel` – класс, реализующий модель системы автоматов, по описанию системы автоматов строит модель и формирует текст на языке *SMV*;
- `ObjectModel` – класс, реализующий модель конкретного экземпляра автомата, содержит описание связей с другими экземплярами и модель автомата;
- `AutomatonModel` – класс, реализующий модель автомата, хранит в себе набор *состояний модели*. Формирует текст модуля соответствующего автомата на языке *SMV*;
- `ModelState` – класс, реализующий *состояние модели,* хранит в себе набор переходов из данного состояния;
- `AutomatonStateModel` – класс, реализующий модель состояния автомата. Он преобразует состояние автомата в состояние модели, а также условия на переходах;
- `EventModel` – класс, реализующий модель события. Он хранит имя события, и информацию о том может ли оно поступать от источника событий.

3.1.2. Прототип программного средства

В качестве примера для тестирования прототипа рассмотрим задачу об обедающих философах. За столом расставлены стулья, на каждом из которых сидит философ. В центре стола – большое блюдо спагетти, а на столе лежат вилки – каждая между двумя соседними тарелками. Все философы хотят есть. Для того, чтобы начать есть, философу необходимы две вилки: одна – в правой руке, другая – в левой. Закончив еду, философ кладет вилки слева и справа от своей тарелки и засыпает.

Реализуем данную задачу на автоматах. На рис. 33 показан автомат `Fork`, реализующий вилку.

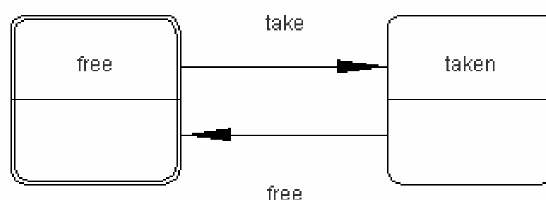

Рис. 33. Диаграмма состояний автомата `Fork`.

Автомат имеет следующие состояния:

- `free` – вилка свободна;
- `taken` – вилка взята.

Автомат принимает два события:

- событие `take` – философ берет вилку;
- событие `free` – философ освобождает вилку.

На рис. 34 изображен автомат `Philosopher`. Он имеет следующие состояния:

- `WaitLeft` – философ ждет левую вилку;
- `WaitRight` – философ ждет правую вилку;
- `Eat` – философ ест;

- `Sleep` – философ спит.

Автомат принимает единственное событие `step`, когда он переходит в следующее состояние. Автоматы содержит ссылки на автоматы `right` и `left` – правая и левая вилки соответственно.

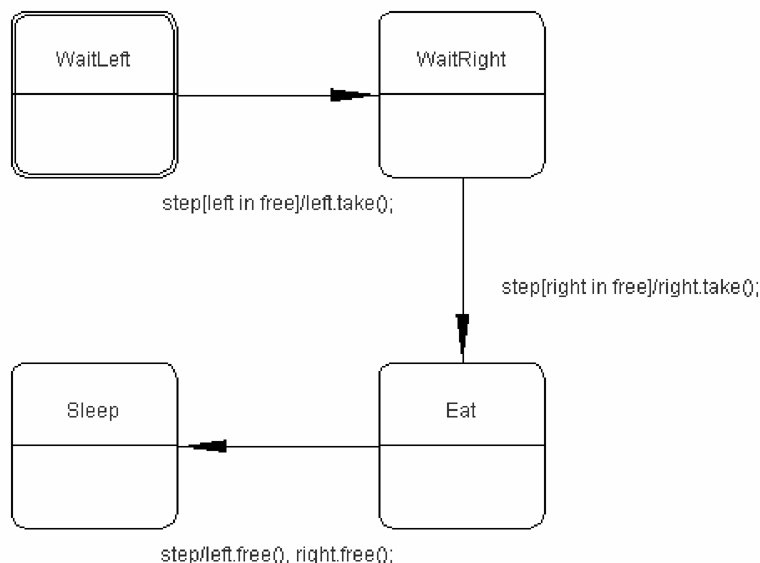

Рис. 34. Диаграмма состояний автомата `Philosopher`

В данном случае алгоритм составлен неправильно, и может возникнуть ситуация, при которой все философы возьмут левую вилку, и ни один из них не сможет взять правую.

Проводилась проверка свойства, что все философы всегда смогут перейти в состояние `Sleep`. На рис. 35 приводятся результаты проверки для разного числа автоматов.

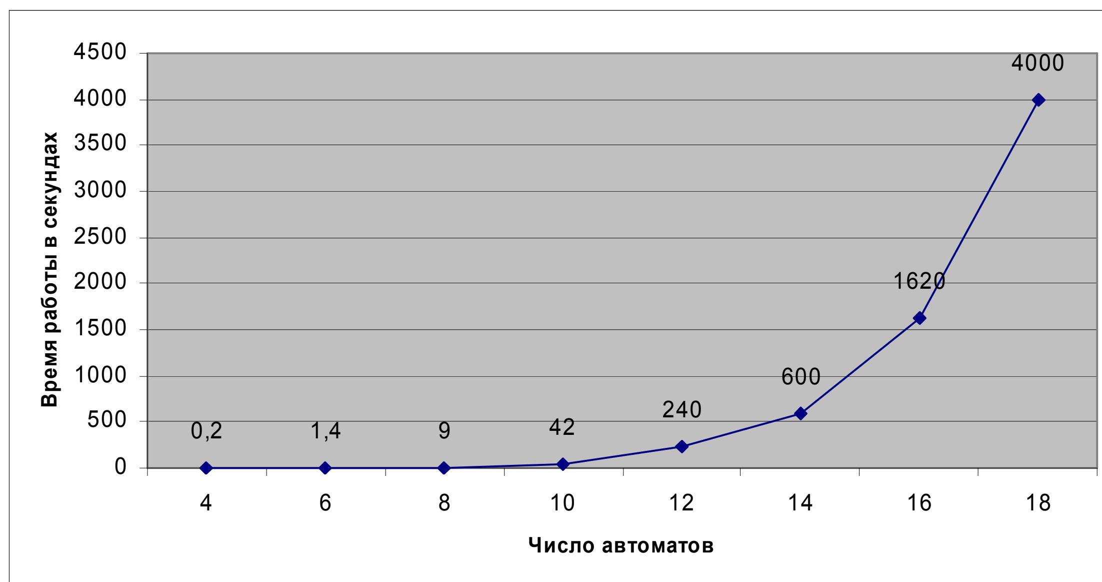

Рис. 35. Зависимость времени работы верификатора от числа автоматов

Из рассмотрения рис. 35 следует, что средство позволяет верифицировать системы из 10 – 15 автоматов с небольшим числом состояний каждый. При проверке систем с большим числом

автоматов время работы быстро возрастает. Из изложенного следует, что данный метод может применяться для достаточно большого круга задач, построенных на основе автоматного подхода.

В прототипе программного средства отсутствует возможность отслеживать выходные воздействия автоматов, входные воздействия могут иметь только тип `Boolean`, если в системе присутствуют рекурсивные вызовы, то модель может получиться неадекватной. Однако, это не влияет на применимость созданного прототипа верификатора на практике по следующим причинам:

- выходные воздействия автоматов не оказывают непосредственного влияния на верифицируемое поведение автомата;
- в автоматном программировании обычно используются булевы переменные;
- в автоматном программировании используется либо вложенность автоматов, либо взаимодействие посредством обмена событиями.

3.2. UNIMOD.VERIFIER

В настоящем разделе описывается программное средство *UniMod.verifier*, созданное на основе метода верификации визуальных моделей автоматных программ

3.2.1. Модель программного средства

Поскольку в рассматриваемом методе используется стандартный алгоритм верификации формул темпоральной *LTL*-логики, то возможно применение метода эмуляции с использованием существующего верификатора. Также для реализации метода эмуляции требуется интерпретировать автоматные программы. Для этой цели также можно использовать существующий интерпретатор.

В настоящей работе было создано программное средство *UniMod.verifier*. В нем реализуется метод верификации автоматных программ на основе их эмуляции. Для интерпретации автоматных программ в нем используется интерпретатор *UniMod*, а алгоритм верификации выполняет верификатор *Bogor*. В данном разделе сначала описывается верификатор *Bogor* и интерпретатор *UniMod*, а после этого – программное средство *UniMod.verifier*.

Bogor [3, 4] – это верификатор с расширяемым входным языком и модульной структурой. Во входной язык *BIR* (*Bogor Input Representation*) верификатора *Bogor* можно добавлять новые типы и абстракции, а сам верификатор разделен на модули, реализующие различные аспекты верификации (такие как алгоритм обхода, кодирование состояния и т.д.). Эти модули достаточно просто можно заменять, и за счет этого изменять логику верификатора, не переписывая его заново. Таким образом, *Bogor* представляет собой гибкий верификатор, который достаточно легко можно настроить для оптимальной верификации конкретного типа задач.

Bogor поддерживает верификацию формул темпоральной *LTL*-логики. Для этого он использует алгоритм двойного поиска в глубину, описанный в разд. 1.3.2.

Расширение входного языка *Bogor* – это создание в нем новых классов (типов). Новые классы используются для того, чтобы абстрагироваться от несущественных деталей реализации верифицируемой программы и сконцентрироваться лишь на той части логики программы, которую требуется верифицировать. Например, пусть верифицируемая система оперирует сложными типами данных, такими как «множество объектов». Тогда для моделирования этой системы было бы удобно, если во входном языке верификатора был бы определен такой тип данных как «множество». Однако обычно входные языки верификаторов бывают достаточно ограничены и оперируют лишь простыми типами данных. В верификаторе *Bogor* можно решить эту проблему, создавая новые типы данных при необходимости.

Для создания нового типа данных (класса) в языке *BIR*, необходимо специфицировать следующие его свойства, необходимые для работы алгоритма двойного поиска в глубину.

1. Требуется уметь кодировать состояние объекта в виде массива целых чисел. Алгоритм обхода использует этот массив для построения глобального состояния верифицируемой системы: глобальное состояние – это совокупность состояний всех объектов в системе.
2. Требуется описать методы (действия) для нового класса. В языке *BIR* есть два типа методов:
   а. `expdef` – методы, которые возвращают значение и не изменяют внутреннее состояние объекта.
   б. `actiondef` – методы, которые производят действия над объектом и изменяют его состояние, и не возвращают значений.
3. Если при выполнении метода возникает необходимость недетерминированного выбора, информация о возможных вариантах предоставляется верификатору, и он делает выбор.
4. Для методов `actiondef` кроме самой реализации действия должен определяться способ обратного действия, возвращающего объект в предыдущее состояние.

Для выполнения перечисленных пунктов верификатор *Bogor* предоставляет набор интерфейсов, написанных на языке программирования *Java*. Новый тип создается путем программирования *Java*-класса, реализующего необходимые интерфейсы.

UniMod [6, 7] – это инструментальное средство, предназначенное для создания и интерпретации реактивных систем, управляемых иерархической системой автоматов. *UniMod* позволяет создавать автоматы в виде *UML*-диаграмм, в то время как источники событий и объекты управления реализуются в виде *Java*-классов. *UniMod* сохраняет созданные автоматные программы в виде *XML*-файла.

В программном средстве *UniMod.verifier* было создано расширение языка *BIR* новым типом `AutomataModel`. Этот тип является адаптером между интерпретатором *UniMod* и верификатором *Bogor*, и позволяет верификатору работать с автоматной программой как с отдельным объектом нового типа. При этом верификация происходит на основе описанного выше метода. Созданный адаптер позволяет подавать на вход верификатору автоматную программу, сохраненную средством *UniMod* в *XML*-файле, и требования в виде *LTL*-формулы (во входном файле на языке *BIR*). На выход верификатор выдает текстовый список действий, которые привели к нарушению требований, или сообщение об удачном завершении верификации.

Объявление типа `AutomataModel` во входном файле верификатора (на языке *BIR*) производится следующим образом:

```
extension AutomataModel for
    com.unimod.verifier.bogorextension.AutomataModelModule
{
    typedef type;

    expdef AutomataModel.type create();

    expdef boolean wasEvent(AutomataModel.type model, string
        event);
    expdef boolean wasInState(AutomataModel.type model, string
        stateMachine, string state);
    expdef boolean isInState(AutomataModel.type model, string
        stateMachine, string state);
    expdef boolean cameToState(AutomataModel.type model, string
        stateMachine, string state);
```

```
        expdef boolean cameToFinalState(AutomataModel.type model);
        expdef boolean wasAction(AutomataModel.type model, string
            action);
        expdef boolean wasFirstAction(AutomataModel.type model,
            string action);
        expdef boolean wasLastAction(AutomataModel.type model,
            string action);
        expdef int getActionIndex(AutomataModel.type model, string
            action);
        expdef boolean wasTrue(AutomataModel.type model, string
            guard);
        expdef boolean wasFalse(AutomataModel.type model, string
            guard);

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

В объявлении класса перечисляются его методы. Заметим, что методы вызываются не у объекта, а у класса (аналог `static`-методов в *Java*). Объект, над которым производится действие, передается как параметр.

Опишем элементы объявления класса `AutomataModel`.

extension AutomataModel **for** com.unimod.verifier.bogorextension.AutomataModelModule	Объявление нового типа и указание *Java*-класса, который его реализует.
typedef type;	Поле, обозначающее новый тип. Используется в *BIR*-файле при объявлении переменной в виде `AutomataModel.type model;`
expdef AutomataModel.type *create*();	Создает новый объект данного типа, представляющий автоматную программу. Используется в виде `model := AutomataModel.create();` Файл, описывающий автоматную программу, указывается отдельно.
actiondef *step* (AutomataModel.type model);	Производит над передаваемой автоматной системой `model` один элементарный шаг выбора и обработки события на основе рассмотренного метода.
wasEvent, wasInState, isInState, cameToState, wasAction, wasFirstAction, wasLastAction, getActionIndex, wasTrue, wasFalse	Эти методы используются для вычисления предикатов в соответствии с описанием рассмотренного метода (разд. 1.3.3).

Автомат передается как параметр в виде *пути* автомата. Поясним это. Так как автоматная программа представляет собой иерархическую систему автоматов, одни автоматы могут быть вложены в состояния других.

При этом одинаковые автоматы могут быть вложены в различные состояния, и одного имени становится недостаточно, чтобы их идентифицировать. Поэтому 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>*».

Например, если автомат `A1` вложен в состояние `s2` корневого автомата `A`, то путь автомата `A1` – это «`/A:s2/A1`».

За счет создания нового класса исполнимая часть *BIR*-спецификации сводится к минимуму: инициализации глобальной переменной типа `AutomataModel` и бесконечному циклу, вызывающему метод `step` у этой переменной. На языке *BIR* это записывается следующим образом:

```
AutomataModel.type model;

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

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

Здесь описана модель программы для верификатора *Bogor*. Программа состоит из двух состояний: состояния `init`, в котором инициализируется автоматная система и происходит переход в состояние `loop`, и состояния `loop`, в котором происходит шаг автоматной системы и переход в себя. Модификатор **`invisible`** при вызове инициализации означает, что это действие произойдет «незаметно» для верифицируемого свойства. Это необходимо для того, чтобы при верификации требование не проверялось на еще не созданном объекте. Бесконечный цикл шагов в автоматной программе останавливается верификатором по алгоритму верификации каждый раз, когда автоматная программа оказывается снова в уже посещенном состоянии.

Опишем теперь, как формулируются требования к автоматной системе. Они записываются в *BIR*-файл с помощью расширения языка *BIR*, позволяющего формулировать утверждения в темпоральной логике *LTL*. К примеру, свойство «автомат `A` никогда не попадет в состояние `Error`» записывается в *BIR*-файле следующим образом:

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

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

```
        )
    );
```

Сначала создается пропозиционная формула «`is_Error`», которой соответствует вызов соответствующего метода `AutomataModel`. Затем эта формула используется в *LTL*-формуле. Приведенная выше запись соответствует формуле *G*¬ `isInState(Error)` (всегда не «Ошибка»). Приведем полный список темпоральных операторов, используемых в языке *BIR*.

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

С помощью перечисленных операторов можно задать любую *LTL*-формулу, описывающую требование к автоматной модели.

В алгоритме двойного обхода в глубину *LTL*-формула преобразуется в автомат Бюхи, который потом используется при верификации. В языке *BIR* имеется возможность сразу записывать требования в виде автомата Бюхи. Например, приведенная выше формула ***G*¬**(`is_Error`) будет преобразована в следующий автомат:

```
function generated$FSA()
{
    loc T0_init:
        when true do
        {
        }
```

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

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

Схема взаимодействия верификатора *Bogor*, интерпретатора *UniMod* и созданного расширения AutomataModel изображена на рис. 36.

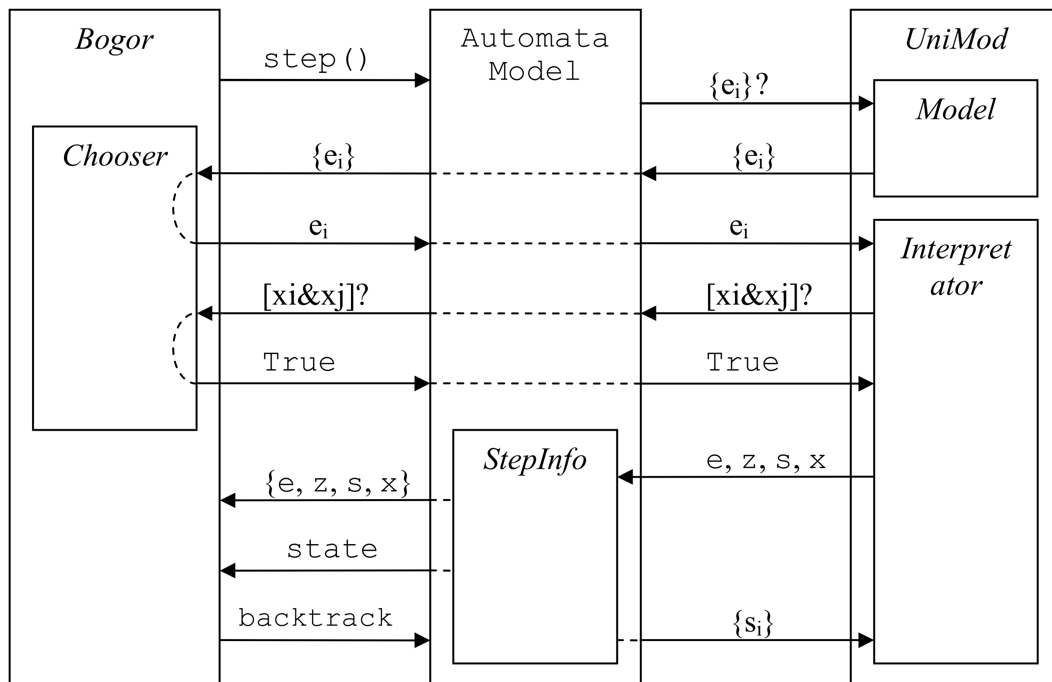

Рис. 36. Схема взаимодействия *Bogor*, *UniMod* и AutomataModel

Верифицируемая программа с точки зрения *Bogor* – это единственный цикл, в котором у AutomataModel вызывается метод step. Опишем действия, которые происходят при таком вызове. Сначала AutomataModel запрашивает у средства *UniMod* набор событий, которые обрабатываются в текущем состоянии системы автоматов. Этот запрос поступает к компоненте *Model*, в которой содержится информация о структуре системы автоматов и текущая ее конфигурация. Полученный набор событий (на рисунке обозначено как «$\{e_i\}$») AutomataModel передает в верификатор *Bogor* для выбора события. Выбор осуществляет специальная компонента *Chooser*. Выбранное событие e_i передается в интерпретатор *UniMod* для обработки. В процессе обработки события и поиска активных

переходов, интерпретатору может понадобиться вычислить значение условия на переходе. В таком случае это условие передается в расширение `AutomataModel`, а оттуда в компоненту *Chooser* для выбора одного из значений `True` или `False`. Выбранное значение передается обратно в интерпретатор.

Интерпретатор обрабатывает событие, осуществляя переходы в автоматах. При этом информация об обрабатываемом событии, вызываемых действиях, состояниях автоматов и значениях условий на переходах («`e, z, s, x`») записывается в контейнер *StepInfo*. Когда интерпретатор заканчивает обработку события, из контейнера *StepInfo* в верификатор *Bogor* передается информация о совершенных действиях – для вычисления значений предикатов, а также текущее состояние системы («`state`») – необходимое для работы алгоритма двойного обхода в глубину. Если состояние уже было посещено, то верификатор *Bogor* передает в `AutomataModel` команду `backtrack`, и из контейнера *StepInfo* в интерпретатор передается набор предыдущих состояний, с тем чтобы вернуть систему автоматов в предыдущее состояние.

3.2.2. Прототип программного средства

Разработанное программное средство *UniMod.verifier* было апробировано на нескольких примерах и успешно доказывало верные утверждения о модели и опровергало неверные утверждения. Рассмотрим работу программного средства на примере, приведенном в описании метода эмуляции (разд. 1.3.4). Напомним, что рассматривалась простая модель работы дверей лифта (рис. 24).

Разработанному программному средству на вход подавались *UniMod*-файл, содержащий описанную модель, *BIR*-файл, содержащий служебную информацию и набор требований для верификации, а также название требования, сформулированного в *BIR*-файле, которое требуется верифицировать. Далее перечисляются утверждения и результаты их верификации.

«Автомат никогда не попадает в состояние `Error`». В *BIR*-файле это требование формулируется следующим образом:

```
fun NeverError_Failing() returns boolean =
  LTL.temporalProperty(
    Property.createObservableDictionary(
      Property.createObservableKey("is_Error",
                AutomataModel.isInState(model, "/A", "Error"))
    ),

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

Это запись соответствует формуле темпоральной логики «$G\neg$ `isInState(Error)`» («`A`» – название корневого и единственного автомата в данном примере). Приведем результат работы верификатора:

```
D:\>verifier NeverError_Failing

Transitions: 1, States: 1, Matched States: 0, Max Depth: 1, Errors found:
0, Used Memory: 2MB
Transitions: 9, States: 7, Matched States: 3, Max Depth: 6, Errors found:
1, Used Memory: 1MB
```

```
Transitions: 10, States: 7, Matched States: 4, Max Depth: 6, Errors
found: 2, Used Memory: 1MB
Total memory before search: 702a880 bytes (0,67 Mb)
Total memory after search: 1a109a016 bytes (1,06 Mb)
Total search time: 1156 ms (0:0:1)
States count: 7
Matched states count: 4
Max depth: 6

Generating error trace 0...

Generating error trace 1...

Done!


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

Replaying trace by key: 1
Stack of transitions leading to the error:
Model [ step [0] event [null] guards [null] transitions [null] actions
       [null] states [null] ] fsaState [T0_init]
Model [ step [0] event [] guards [] transitions [] actions [] states
[(/A)
       - (Top)] ] fsaState [T0_init]
Model [ step [1] event [*] guards [] transitions [s1#Closed#*#true]
actions
       [] states [(/A) - (Closed)] ] fsaState [T0_init]
Model [ step [2] event [e11] guards [true->true] transitions
       [Closed#Opening#e11#true] actions [o1.z1] states [(/A) - Opening)]
]
       fsaState [T0_init]
Model [ step [3] event [e4] guards [o2.x1->true] transitions
       [Opening#Error#e4#o2.x1] actions [o2.z1] states [(/A) - (Error)] ]
       fsaState [bad$accept_all]

Done!
```

Поясним протокол работы верификатора. В начале указывается служебная информация, и выдаются сообщения о прогрессе процесса верификации. Затем верификатор нашел два сценария («2 traces found») нарушения приведенного требования и вывел ошибку для наиболее короткого из них. Поясним формат сценария.

В сценарии каждый шаг имеет следующий формат:

```
step [<порядковый номер шага>]
event [<полученное событие>]
guards [<вычисленные условия на переходах с их значениями>]
transitions [<осуществленные переходы в автоматах>]
```

```
actions [<список вызванных воздействий в объектах управления>]
states [<состояние каждого автомата после обработки события>]
fsaState [<состояние автомата Бюхи после обработки события>]
```

При генерации сценария выводится избыточная информация, и для понимания произведенных переходов достаточно только полученного события и вычисленных условий на переходах. Первые два шага – инициализация модели. Они не несут смысловой нагрузки. С третьей строки сценария имеет место получение события «*». Это событие происходит в инструментальном средстве *UniMod* тогда, когда не определено ни одного другого события для исходящих из текущего состояния переходов. Автомат перешел из начального состояния в состояние `Closed`. Далее было получено событие `e11`, и автомат перешел в состояние `Opening`. Затем произошло событие `e4`, условие `o2.x1` было вычислено как `True`, и автомат оказался в состоянии `Error`.

Допускающий путь имеет структуру $\alpha\beta^*$, но в сценарий выводится только часть $\alpha\beta$. Выделить цикл β достаточно просто: требуется лишь найти шаг сценария с теми же состояниями автоматов модели и автомата Бюхи, как в последнем шаге. Заметим, что в алгоритме двойного обхода в глубину суффикс β всегда начинается с допускающего состояния. Названия допускающих состояний в сгенерированном автомате Бюхи в верификаторе *Bogor* начинаются с «`bad$`». В приведенном сценарии цикл β состоит из единственного шага в состоянии `Error`.

Итак, верифицируемое требование было нарушено и представлен сценарий событий и условий, при которых автомат попадает в состояние `Error`.

Изменим свойство, добавив условие, что никогда не происходит событие `e4`:

```
G(¬ wasEvent("e4") ) → G (¬ isInState("/A", "Error") )
```

Результат работы верификатора:

```
D:\>verifier NeverError_Successive
Transitions: 1, States: 1, Matched States: 0, Max Depth: 1, Errors found: 0, Used Memory: 2MB
Transitions: 9, States: 6, Matched States: 3, Max Depth: 6, Errors found: 0, Used Memory: 1MB
Total memory before search: 706a320 bytes (0,67 Mb)
Total memory after search: 1a116a888 bytes (1,07 Mb)
Total search time: 625 ms (0:0:0)
States count: 6
Matched states count: 3
Max depth: 6
Done!
Verification successful!
```

Как правильно определил верификатор, данное свойство выполняется на модели.

Проверим теперь следующее свойство: если в состоянии `Opening` произошло событие `e4`, то следующее состояние будет `Error`. Это свойство записывается следующим образом:

```
G ( wasInState("/A", "Opening") & wasEvent("e4") →
        isInState("/A", "Error") )
```

Результат – свойство выполняется. Данный пример предназначен для демонстрации работы анализатора условий на переходах. До того, как был добавлен анализатор, для данного свойства

генерировался контрпример, в котором присутствовал следующий невозможный набор значений условий:

```
guards [o2.x1->false, !o2.x1->false]
```

Теперь проверим следующее свойство: если не происходит ошибки, то для любой истории автомат попадает в состояние Opened бесконечно часто. Это свойство записывается следующим образом:

```
G¬ wasEvent("e4") → G F isInState("/A", "Opened")
```

Результат верификации – условие выполняется. Стоит заметить, что при верификации рассматриваются только *справедливые* [2] истории. Это означает, что любое ожидаемое событие когда-либо произойдет, и автомат не остановится навсегда в состоянии, из которого есть переход по некоторому событию. Применительно к данному примеру, это означает, например, что если дверь стала открываться, то событие e2 обязательно произойдет. Рассматривать несправедливые истории, пожалуй, не имеет смысла, поскольку тогда для любого свойства найдется контрпример, когда автомат просто не получает никаких событий.

Последнее свойство, которое проверяется на данном примере – такое же, как предыдущее, только для состояния Closed: при условии отсутствия ошибки, автомат бесконечно часто проходит через состояние Closed. Это свойство запишем следующим образом:

```
G¬ wasEvent("e4") → G F isInState("/A", "Closed")
```

Результат верификации – свойство нарушается. Выводимый сценарий ошибки:

```
[1] event [*] ... states [(/A) - (Closed)] ] fsaState [T0_init]
[2] event [e11] ... states [(/A) - (Opening)] ] fsaState [bad$accept_S2]
[3] event [e2] ... states [(/A) - (Opened)] ] fsaState [bad$accept_S2]
[4] event [e12] ... states [(/A) - (Closing)] ] fsaState [bad$accept_S2]
[5] event [e3] ... states [(/A) - (Opening)] ] fsaState [bad$accept_S2]
```

Как следует из протокола, получен цикл, состоящий из шагов 2–5, в котором автомат не попадает в состояние Closed. Полученный контрпример соответствует ситуации, когда при каждой попытке закрыться дверь встречает препятствие (e3) и снова открывается. Графически контрпример представлен на рис. 37.

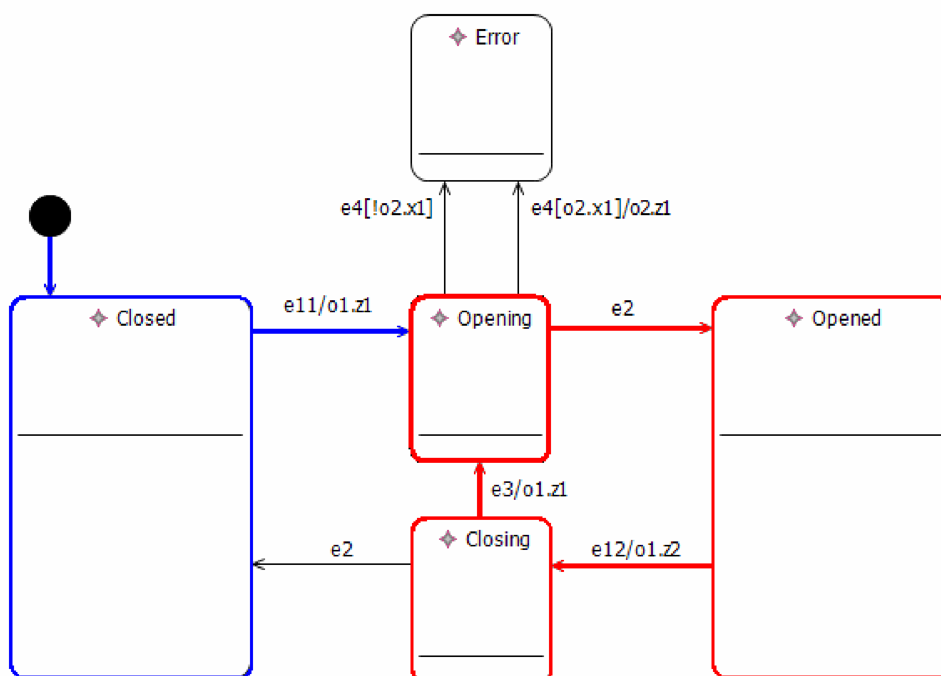

Рис. 37. Графическое представление контрпримера для утверждения «G¬ `wasEvent("e4")` → G F `isInState("/A", "Closed")`»

3.3. CONVERTER

Инструментальное средство *Converter* является дополнительным модулем к верификатору *SPIN*, реализующим метод верификации визуальным моделей автоматных программ.

3.3.1. Модель программного средства

3.3.1.1. Общее описание инструментального средства *Converter*

Инструментальное средство *Converter* позволяет автоматизировать верификацию визуальных автоматных программ, разработанных при помощи инструментального средства *UniMod* [3.2].

По автоматной программе инструментальное средство *Converter* создает модель, в которой отброшены несущественные детали. *LTL*-формула преобразовывается в пригодный для верификатора *SPIN* вид.

Инструментальное средство *Converter* должно получать на вход три параметра:

1. Путь к файлу с визуальной автоматной программой, разработанной при помощи инструментального средства *UniMod* и сохраненной в *XML*- формате.
2. Имя файла, в который будет записана созданная модель.
3. Одну *LTL*-формулу с требованиями к модели.

На выходе инструментального средства *Converter* формируется файл, в котором записан полный отчет о проведенной верификации, включая автоматически построенный верификатором *SPIN* контрпример, представленный в текстовом виде, если он найден.

3.3.1.2. Описание работы инструментального средства

Автоматическая верификация автоматных программ с помощью инструментального средства *Converter* состоит из нескольких этапов (рис. 38).

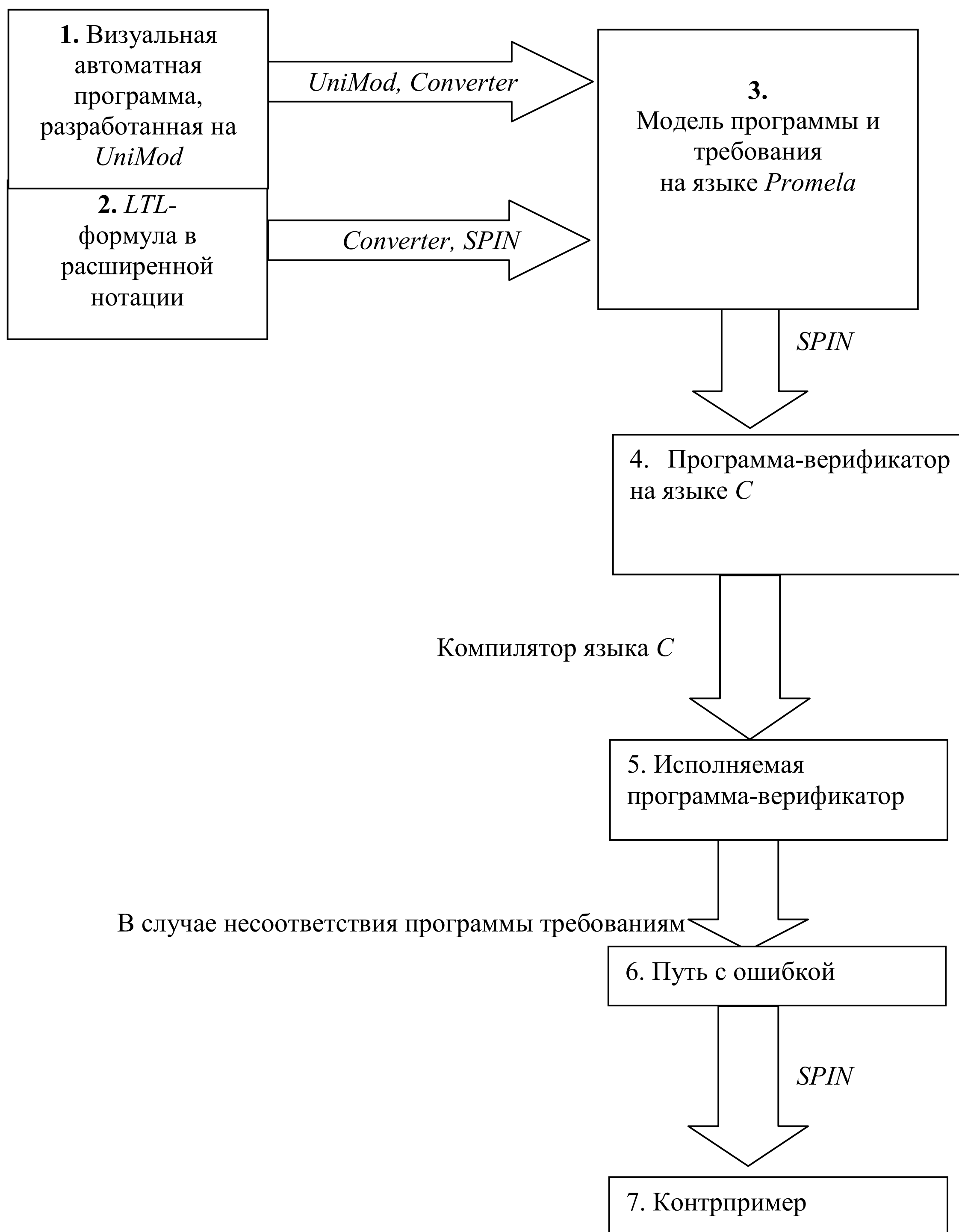

Рис. 38. Общее описание работы программного средства

1. *Converter* принимает на вход визуальную автоматную программу, разработанную при помощи инструментального средства *UniMod*, сохраненную в формате *XML*, и требования к ней, записанные на языке *LTL* в расширенной нотации (рис. 38, переход 1 – 2). **Важно: верификатору на вход подаются не требования, а их отрицание.**
2. При помощи инструментального средства *UniMod* из *XML*-файла получается автоматная модель на языке *Java* (рис. 39) – переход 1-3 на рис. 38.

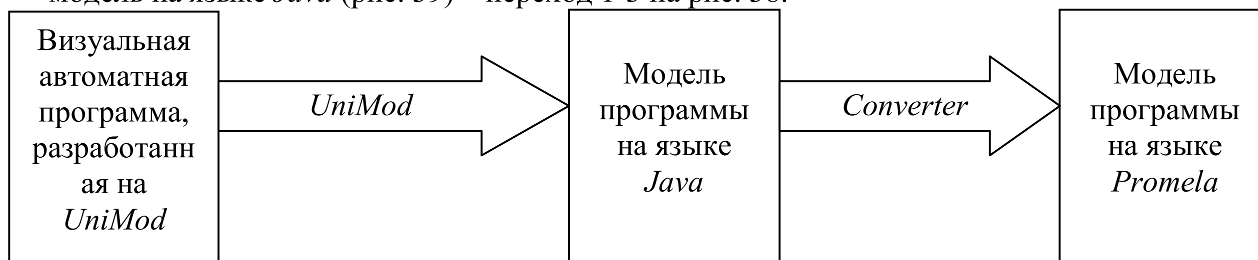

Рис. 39. Взаимодействие с *UniMod*

Converter транслирует автоматную модель с языка *Java* на язык *Promela* (рис. 39) – переход 1-3 на рис. 38.

Converter преобразует *LTL*-формулу в нотацию верификатора *SPIN* (рис. 40).

Рис. 40. Преобразование *LTL*-формул

Это преобразование выполняется следующим образом:

- Все элементарные высказывания должны быть записаны в фигурных скобках.
- Все элементарные высказывания должны удовлетворять синтаксису языка *Promela*.
- Каждому элементарному высказыванию присваивается идентификатор `pk`, где *k* — порядковый номер элементарного высказывания в формуле.
- Для каждого элементарного высказывания *Converter* генерирует макрос на языке *Promela*, закрепляющий идентификатор за элементарным высказыванием. Если *Converter* распознает формулу как неправильную, то он выводит в консоль сообщение «Wrong formula».
- Идентификаторы событий `exx`, где *xx* – номер события, преобразовываются в число *xx*.
- Пример. Формула (элементарное высказывание)
  ```
  {lastEvent == e13}
  ```
  преобразовывается в строку на языке *Promela*:
  ```
  #define pk (lastEvent == 13),
  ```
  где *k* — номер элементарного высказывания.

3. *Converter* запускает верификатор *SPIN* в режиме генерации конструкции *never claim* (с помощью команды `spin -f <формула>`). Верификатор по *LTL*-формуле генерирует

конструкцию n*ever claim*, представляющую собой *автомат Бюхи*, записанный на языке *Promela* (рис. 40). Это также переход 2–3 на рис. 38.

4. После того, как на языке *Promela* описаны модель и требование к ней, *Converter* запускает *SPIN* на верификацию (командой `spin -a <Модель>`). Верификатор *SPIN* генерирует файл `pan.c`, представляющий собой программу-верификатор на языке *C* (переход 3 – 4 на рис. 38).
5. После компиляции файла `pan.c` получается программа-верификатор для данной конкретной модели с заданными требованиеми. Программа *pan* выполняет верификацию построенной модели. При обнаружении ошибок программа *pan* выдает *trail*-файл, в котором описана трасса ошибки в формате, понятном верификатору *SPIN* (переходы 4–6 на рис. 38). Кроме того, программа-верификатор *pan* выводит отчет, в котором содержится краткая информация об ошибках (переход 5–6 на рис. 38), использованной памяти, версии верификатора *SPIN* и т.д.
6. По команде `spin -t -p <Модель>` верификатор выводит отчет, содержащий контрпример. *Converter* собирает воедино отчет, созданный *SPIN*, и отчет, созданный программой *pan* (переход 6–7 на рис. 38).

3.3.1.3. Диаграмма классов инструмента

На рис. 41 приведена диаграмма классов инструмента *Converter*.

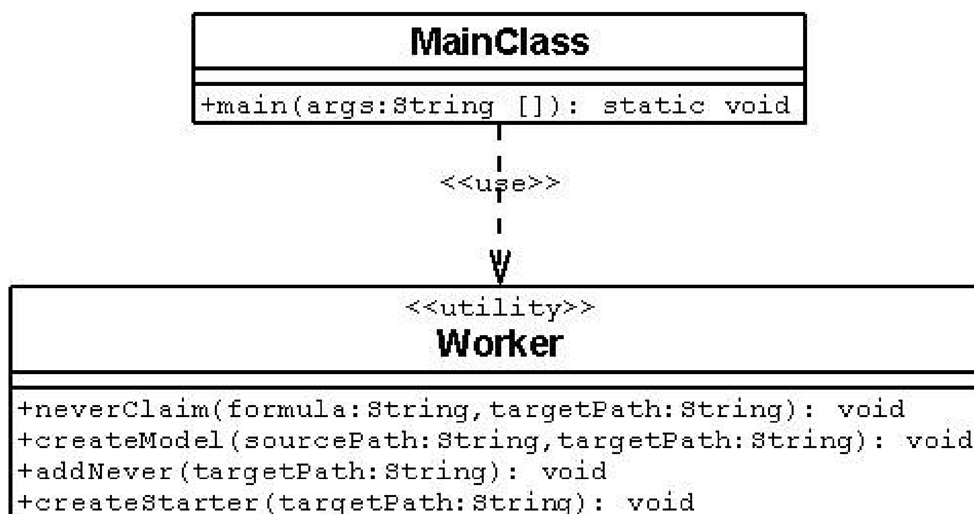

Рис. 41. Диаграмма классов прототипа программного средства

3.3.2. Прототип программного средства

Инструментальное средство на всех тестовых задачах давало правильный результат.

Для тестирования был создан генератор автоматов с заданными числами состояний и переходов. В результате тестовых прогонов было установлено, что инструментальное средство позволяет верифицировать:

1. Визуальную автоматную программу, состоящую из одного автомата с 568 состояниями и с одним переходом в каждом состоянии. При большем числе состояний *SPIN* не может верифицировать модель, сгенерированную инструментальным средством.
2. Визуальную автоматную программу, состоящую из одного автомата с 214 состояниями с пятью переходами в каждом состоянии. При большем числе состояний *SPIN* не может верифицировать модель, сгенерированную инструментальным средством.

Все тестовые примеры верифицировались менее чем за десять секунд на компьютере *Intel Pentium 4* с тактовой частотой 1.8 Ггц с 1.4 Гб ОЗУ в операционной системе *Windows XP*. Версия языка *Java* 1.4.2_01.

Продемонстрируем работу прототипа на примере. Рассмотрим автоматную реализацию игры *Ним* из работы [15]. Эта версия содержит три автомата. В программе автомат *A1* (рис. 42) – главный.

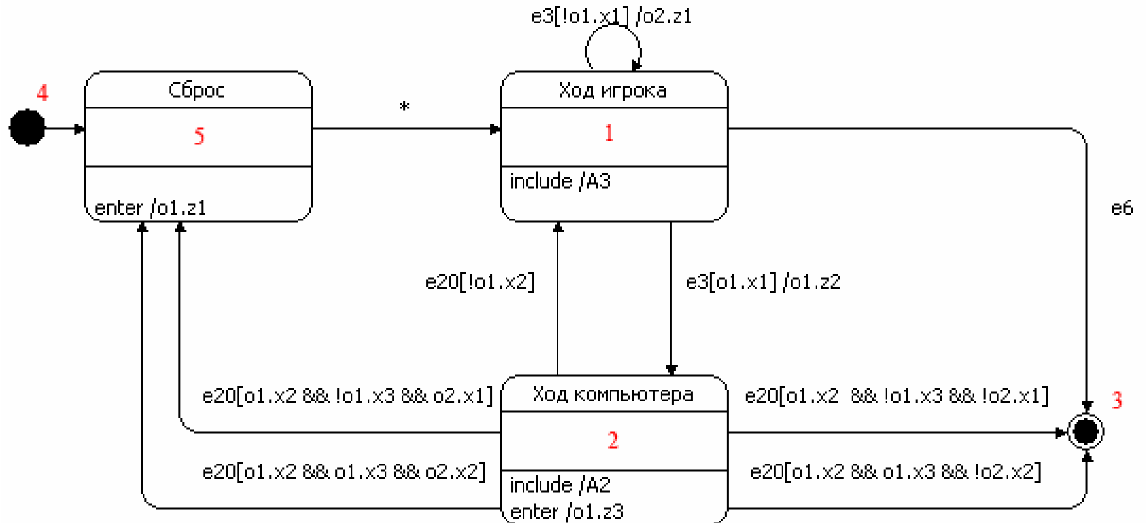

Рис. 42. Автомат *A1*

В автомате *A2* (рис. 43) реализована логика компьютерного игрока.

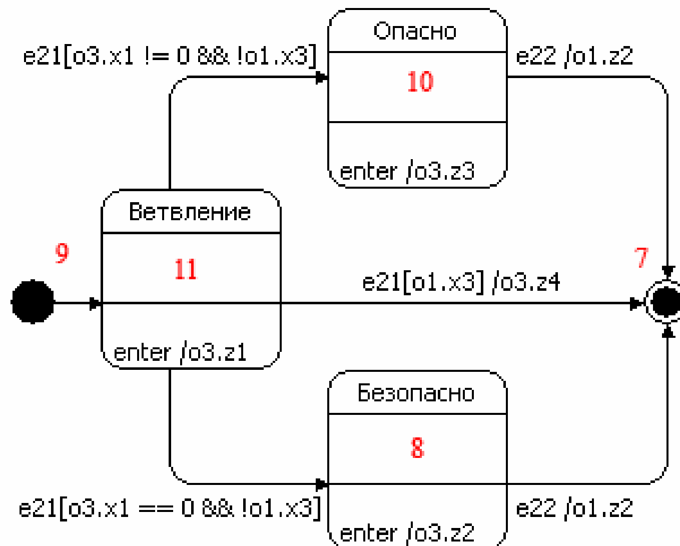

Рис. 43. Автомат *A2*

Автомат *A3* (рис. 44) – интерфейсный.

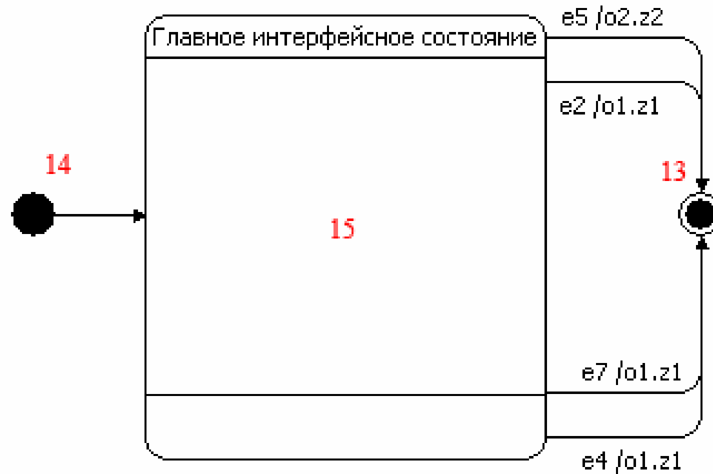

Рис. 44. Автомат *A3*

Внесем ошибку в автомат *A1* (рис. 45): в состоянии `Ход игрока` вместо автомата *A3* вызовем автомат *A2*.

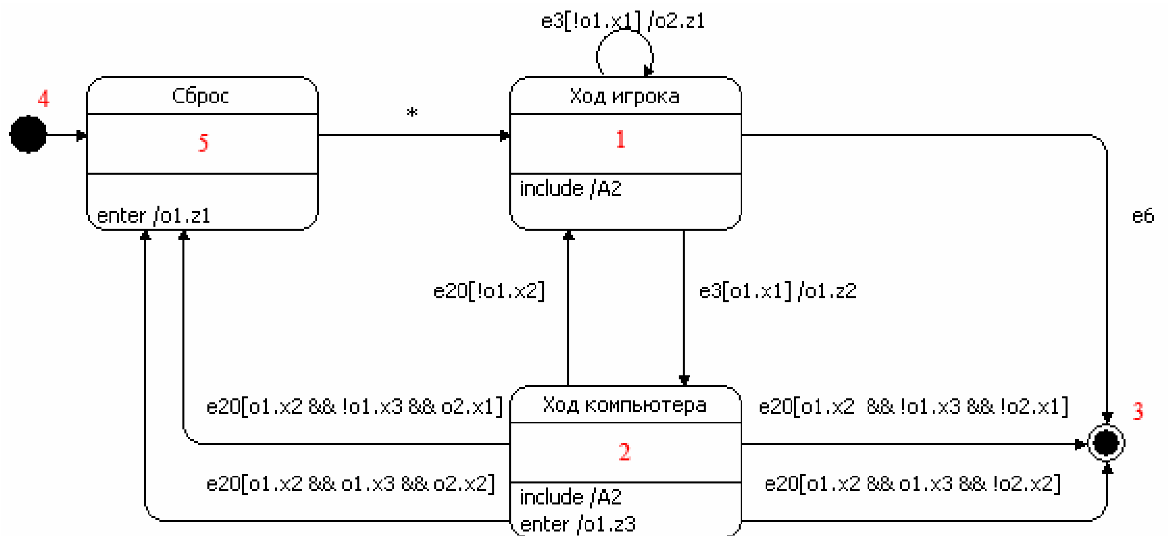

Рис. 45. Автомат *A1* с ошибкой

Одним их эффектов этой ошибки является факт, что автомат *A3* больше не вызывается. Подадим программу с ошибкой, сохраненную в *XML*-формате на вход инструментального средства *Converter*. Это средство присвоит всем состояниям автоматов номера, в соответствии с рис. 43-45. Преобразуем требование о том, что автомат *A3* должен быть вызван хотя бы один раз, с языка *LTL* в расширенную нотацию *SPIN*:

```
(<>{stateA3 == 0}) U {stateA1 == 3}
```

Подадим на вход инструментального средства *Converter* автоматную программу и формулу с требованием. *Converter* сгенерирует следующую модель на языке *Promela*:

```
#define p1   (stateA3 == 0)
#define p2   (stateA1 == 3)

int lastEvent;
#define STATE_0  0 /*Top*/
```

```
#define STATE_1  1 /*Ход игрока*/
#define STATE_2  2 /*Ход компьютера*/
#define STATE_3  3 /*s2*/
#define STATE_4  4 /*s1*/
#define STATE_5  5 /*Сброс*/

int stateA1;
inline A1() {
  stateA1 = STATE_4;
  do
  ::(stateA1 == STATE_1) ->
    printf("State 1 : Ход игрока\n");
    A2();
    if
      ::stateA1 = STATE_2;
        lastEvent = 3;
      ::stateA1 = STATE_1;
        lastEvent = 3;
      ::stateA1 = STATE_3;
        lastEvent = 6;
    fi;
  ::(stateA1 == STATE_2) ->
    printf("State 2 : Ход компьютера\n");
    A2();
    if
      ::stateA1 = STATE_1;
        lastEvent = 20;
      ::stateA1 = STATE_5;
        lastEvent = 20;
      ::stateA1 = STATE_3;
        lastEvent = 20;
      ::stateA1 = STATE_5;
        lastEvent = 20;
      ::stateA1 = STATE_3;
        lastEvent = 20;
    fi;
  ::(stateA1 == STATE_3) ->
    printf("State 3 : s2\n");
    break;
  ::(stateA1 == STATE_4) ->
    printf("State 4 : s1\n");
    if
      ::stateA1 = STATE_5;
    fi;
  ::(stateA1 == STATE_5) ->
    printf("State 5 : Сброс\n");
    if
      ::stateA1 = STATE_1;
```

```
    fi;
  od;
}
#define STATE_6  6 /*Top*/
#define STATE_7  7 /*s2*/
#define STATE_8  8 /*Безопасно*/
#define STATE_9  9 /*s1*/
#define STATE_10   10 /*Опасно*/
#define STATE_11   11 /*Ветвление*/

int stateA2;
inline A2() {
  stateA2 = STATE_9;
  do
  ::(stateA2 == STATE_7) ->
    printf("State 7 : s2\n");
    break;
  ::(stateA2 == STATE_8) ->
    printf("State 8 : Безопасно\n");
    if
      ::stateA2 = STATE_7;
        lastEvent = 22;
    fi;
  ::(stateA2 == STATE_9) ->
    printf("State 9 : s1\n");
    if
      ::stateA2 = STATE_11;
    fi;
  ::(stateA2 == STATE_10) ->
    printf("State 10 : Опасно\n");
    if
      ::stateA2 = STATE_7;
        lastEvent = 22;
    fi;
  ::(stateA2 == STATE_11) ->
    printf("State 11 : Ветвление\n");
    if
      ::stateA2 = STATE_10;
        lastEvent = 21;
      ::stateA2 = STATE_8;
        lastEvent = 21;
      ::stateA2 = STATE_7;
        lastEvent = 21;
    fi;
  od;
}
#define STATE_12   12 /*Top*/
#define STATE_13   13 /*s2*/
```

```
#define STATE_14   14 /*s1*/
#define STATE_15   15 /*Главное интерфейсное состояние*/

int stateA3;
inline A3() {
  stateA3 = STATE_13;
  do
  ::(stateA3 == STATE_13) ->
    printf("State 13 : s2\n");
    if
      ::stateA3 = STATE_15;
    fi;
  ::(stateA3 == STATE_14) ->
    printf("State 14 : s1\n");
    break;
  ::(stateA3 == STATE_15) ->
    printf("State 15 : Главное интерфейсное состояние\n");
    if
      ::stateA3 = STATE_14;
        lastEvent = 4;
      ::stateA3 = STATE_14;
        lastEvent = 7;
      ::stateA3 = STATE_14;
        lastEvent = 5;
      ::stateA3 = STATE_14;
        lastEvent = 2;
    fi;
  od;
}
proctype Model() {
  A1();
}

init {
run Model();
}
never {    /* (<>p1) U p2 */
T0_init:
  if
  :: ((p2)) -> goto accept_all
  :: ((p1)) -> goto T0_init
  :: (1) -> goto T0_S9
  fi;
T0_S9:
  if
  :: ((p1)) -> goto T0_init
  :: (1) -> goto T0_S9
  :: ((p1) && (p2)) -> goto accept_all
```

```
  :: ((p2)) -> goto T0_S16
  fi;
T0_S16:
  if
  :: ((p1)) -> goto accept_all
  :: (1) -> goto T0_S16
  fi;
accept_all:
  skip
}
```

Файл, содержащий модель, назовем `3_a_incorrect.ltl`. Отчет о проведенной верификации:

```
>spin -a models/3_a_incorrect.ltl
>gcc pan.c -o pan.exe
>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 125)
pan: wrote models/3_a_incorrect.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 32 byte, depth reached 129, errors: 1
      67 states, stored
      20 states, matched
      87 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)

2.622  memory usage (Mbyte)


J:\IFMO\unimod\Converter>spin -t -p models/3_a_incorrect.ltl
Starting :init: with pid 0
Starting :never: with pid 1
Never claim moves to line 142 [((stateA3==0))]
Starting Model with pid 2
  2: proc  0 (:init:) line 136 "models/3_a_incorrect.ltl" (state 1)
          [(run Model())]
  4: proc  1 (Model) line  14 "models/3_a_incorrect.ltl" (state 1) [stateA1
         = 4]
  6: proc  1 (Model) line  45 "models/3_a_incorrect.ltl" (state 99)
          [((stateA1==4))]
            State 4 : s1
  8: proc  1 (Model) line  46 "models/3_a_incorrect.ltl" (state 100)
          [printf('State 4 : s1\\n')]
```

```
10: proc  1 (Model) line  48 "models/3_a_incorrect.ltl" (state 101)
          [stateA1 = 5]
12: proc  1 (Model) line  50 "models/3_a_incorrect.ltl" (state 104)
          [((stateA1==5))]
            State 5 : Сброс
14: proc  1 (Model) line  51 "models/3_a_incorrect.ltl" (state 105)
          [printf('State 5 : Сброс\\n')]
16: proc  1 (Model) line  53 "models/3_a_incorrect.ltl" (state 106)
          [stateA1 = 1]
18: proc  1 (Model) line  16 "models/3_a_incorrect.ltl" (state 2)
          [((stateA1==1))]
            State 1 : Ход игрока
20: proc  1 (Model) line  17 "models/3_a_incorrect.ltl" (state 3)
          [printf('State 1 : Ход игрока\\n')]
22: proc  1 (Model) line  66 "models/3_a_incorrect.ltl" (state 4) [stateA2
        = 9]
24: proc  1 (Model) line  77 "models/3_a_incorrect.ltl" (state 14)
          [((stateA2==9))]
            State 9 : s1
26: proc  1 (Model) line  78 "models/3_a_incorrect.ltl" (state 15)
          [printf('State 9 : s1\\n')]
28: proc  1 (Model) line  80 "models/3_a_incorrect.ltl" (state 16)
          [stateA2 = 11]
30: proc  1 (Model) line  88 "models/3_a_incorrect.ltl" (state 25)
          [((stateA2==11))]
            State 11 : Ветвление
32: proc  1 (Model) line  89 "models/3_a_incorrect.ltl" (state 26)
          [printf('State 11 : Ветвление\\n')]
34: proc  1 (Model) line  91 "models/3_a_incorrect.ltl" (state 27)
          [stateA2 = 10]
36: proc  1 (Model) line  92 "models/3_a_incorrect.ltl" (state 28)
          [lastEvent = 21]
38: proc  1 (Model) line  82 "models/3_a_incorrect.ltl" (state 19)
          [((stateA2==10))]
            State 10 : Опасно
40: proc  1 (Model) line  83 "models/3_a_incorrect.ltl" (state 20)
          [printf('State 10 : Опасно\\n')]
42: proc  1 (Model) line  85 "models/3_a_incorrect.ltl" (state 21)
          [stateA2 = 7]
44: proc  1 (Model) line  86 "models/3_a_incorrect.ltl" (state 22)
          [lastEvent = 22]
46: proc  1 (Model) line  68 "models/3_a_incorrect.ltl" (state 5)
          [((stateA2==7))]
            State 7 : s2
48: proc  1 (Model) line  69 "models/3_a_incorrect.ltl" (state 6)
          [printf('State 7 : s2\\n')]
50: proc  1 (Model) line  20 "models/3_a_incorrect.ltl" (state 39)
          [stateA1 = 2]
52: proc  1 (Model) line  21 "models/3_a_incorrect.ltl" (state 40)
          [lastEvent = 3]
54: proc  1 (Model) line  27 "models/3_a_incorrect.ltl" (state 47)
          [((stateA1==2))]
            State 2 : Ход компьютера
56: proc  1 (Model) line  28 "models/3_a_incorrect.ltl" (state 48)
          [printf('State 2 : Ход компьютера\\n')]
58: proc  1 (Model) line  66 "models/3_a_incorrect.ltl" (state 49)
          [stateA2 = 9]
```

```
 60: proc  1 (Model) line  77 "models/3_a_incorrect.ltl" (state 59)
           [((stateA2==9))]
             State 9 : s1
 62: proc  1 (Model) line  78 "models/3_a_incorrect.ltl" (state 60)
           [printf('State 9 : s1\\n')]
 64: proc  1 (Model) line  80 "models/3_a_incorrect.ltl" (state 61)
           [stateA2 = 11]
 66: proc  1 (Model) line  88 "models/3_a_incorrect.ltl" (state 70)
           [((stateA2==11))]
             State 11 : Ветвление
 68: proc  1 (Model) line  89 "models/3_a_incorrect.ltl" (state 71)
           [printf('State 11 : Ветвление\\n')]
 70: proc  1 (Model) line  91 "models/3_a_incorrect.ltl" (state 72)
           [stateA2 = 10]
 72: proc  1 (Model) line  92 "models/3_a_incorrect.ltl" (state 73)
           [lastEvent = 21]
 74: proc  1 (Model) line  82 "models/3_a_incorrect.ltl" (state 64)
           [((stateA2==10))]
             State 10 : Опасно
 76: proc  1 (Model) line  83 "models/3_a_incorrect.ltl" (state 65)
           [printf('State 10 : Опасно\\n')]
 78: proc  1 (Model) line  85 "models/3_a_incorrect.ltl" (state 66)
           [stateA2 = 7]
 80: proc  1 (Model) line  86 "models/3_a_incorrect.ltl" (state 67)
           [lastEvent = 22]
 82: proc  1 (Model) line  68 "models/3_a_incorrect.ltl" (state 50)
           [((stateA2==7))]
             State 7 : s2
 84: proc  1 (Model) line  69 "models/3_a_incorrect.ltl" (state 51)
           [printf('State 7 : s2\\n')]
 86: proc  1 (Model) line  31 "models/3_a_incorrect.ltl" (state 84)
           [stateA1 = 1]
 88: proc  1 (Model) line  32 "models/3_a_incorrect.ltl" (state 85)
           [lastEvent = 20]
 90: proc  1 (Model) line  16 "models/3_a_incorrect.ltl" (state 2)
           [((stateA1==1))]
             State 1 : Ход игрока
 92: proc  1 (Model) line  17 "models/3_a_incorrect.ltl" (state 3)
           [printf('State 1 : Ход игрока\\n')]
Never claim moves to line 143 [(1)]
 94: pr  1 (Model) line  66 "models/3_a_incorrect.ltl" (state 4)   [stateA2
        = 9]
Never claim moves to line 148 [(1)]
 96: proc  1 (Model) line  77 "models/3_a_incorrect.ltl" (state 14)
           [((stateA2==9))]
             State 9 : s1
 98: proc  1 (Model) line  78 "models/3_a_incorrect.ltl" (state 15)
           [printf('State 9 : s1\\n')]
100: proc  1 (Model) line  80 "models/3_a_incorrect.ltl" (state 16)
           [stateA2 = 11]
102: proc  1 (Model) line  88 "models/3_a_incorrect.ltl" (state 25)
           [((stateA2==11))]
             State 11 : Ветвление
104: proc  1 (Model) line  89 "models/3_a_incorrect.ltl" (state 26)
           [printf('State 11 : Ветвление\\n')]
Never claim moves to line 147 [((stateA3==0))]
106: proc  1 (Model) line  93 "models/3_a_incorrect.ltl" (state 29)
           [stateA2 = 8]
```

```
Never claim moves to line 142 [((stateA3==0))]
108: proc  1 (Model) line  94 "models/3_a_incorrect.ltl" (state 30)
            [lastEvent = 21]
110: proc  1 (Model) line  71 "models/3_a_incorrect.ltl" (state 8)
            [((stateA2==8))]
              State 8 : Безопасно
112: proc  1 (Model) line  72 "models/3_a_incorrect.ltl" (state 9)
            [printf('State 8 : Безопасно\\n')]
114: proc  1 (Model) line  74 "models/3_a_incorrect.ltl" (state 10)
            [stateA2 = 7]
Never claim moves to line 143 [(1)]
116: proc  1 (Model) line  75 "models/3_a_incorrect.ltl" (state 11)
            [lastEvent = 22]
Never claim moves to line 148 [(1)]
118: proc  1 (Model) line  68 "models/3_a_incorrect.ltl" (state 5)
            [((stateA2==7))]
              State 7 : s2
120: proc  1 (Model) line  69 "models/3_a_incorrect.ltl" (state 6)
            [printf('State 7 : s2\\n')]
Never claim moves to line 147 [((stateA3==0))]
122: proc  1 (Model) line  24 "models/3_a_incorrect.ltl" (state 43)
            [stateA1 = 3]
Never claim moves to line 141 [((stateA1==3))]
124: proc  1 (Model) line  25 "models/3_a_incorrect.ltl" (state 44)
            [lastEvent = 6]
Never claim moves to line 158 [(1)]
spin: trail ends after 126 steps
#processes: 2
     lastEvent = 6
     stateA1 = 3
     stateA2 = 7
     stateA3 = 0
126: proc  1 (Model) line  15 "models/3_a_incorrect.ltl" (state 109)
126: proc  0 (:init:) line 137 "models/3_a_incorrect.ltl" (state 2) <valid
          end state>
126: proc  - (:never:) line 159 "models/3_a_incorrect.ltl" (state 26) <valid
          end state>
2 processes created
```

Из отчета следует, что модель нарушает условие *never claim* – не соответствует сформулированному требованию. Выпишем явно из отчета контрпример (Замечание: в рассматриваемой программе во всех автоматах *начальное состояние* называется *s1*, а *конечное состояние – s2*):

Автомат *A1* перешел в состояние *s1*.
Автомат *A1* перешел в состояние *Сброс*.
Автомат *A1* перешел в состояние *Ход игрока*.
В состоянии *Ход игрока* автомата *A1* был запущен автомат *A2*.
Автомат *A2* перешел в состояние *s1*.
Автомат *A2* перешел в состояние *Ветвление*.
Автомат *A2* перешел в состояние *Опасно*.
Автомат *A2* перешел в состояние *s2*.
Автомат *A1* перешел в состояние *Ход компьютера*.
В состоянии *Ход компьютера* автомата *A1* был запущен автомат *A2*.
Автомат *A2* перешел в состояние *s1*.
Автомат *A2* перешел в состояние *Ветвление*.

Автомат *A2* перешел в состояние *Опасно*.
Автомат *A2* перешел в состояние *s2*.
Автомат *A1* перешел в состояние *Ход игрока*.
В состоянии *Ход игрока* автомата *A1* был запущен автомат *A2*.
Автомат *A2* перешел в состояние *s1*.
Автомат *A2* перешел в состояние *Ветвление*.
Автомат *A2* перешел в состояние Безо*пасно*.
Автомат *A2* перешел в состояние *s2*.
Автомат *A1* перешел в состояние *s2*.

Исправим ошибку. Подадим инструментальному средству *Converter* на вход правильную программу и поверяемое требование. Файл с моделью назовем `3_a_correct.ltl`. Получим следующий отчет:

```
J:\IFMO\unimod\Converter>spin -a models/3_a_correct.ltl

J:\IFMO\unimod\Converter>gcc pan.c -o pan.exe

J:\IFMO\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 118, errors: 0
     114 states, stored
      54 states, matched
     168 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)

2.622   memory usage (Mbyte)


J:\IFMO\unimod\Converter>spin -t -p models/3_a_correct.ltl
Starting :init: with pid 0
spin: cannot find trail file
```

Из данного отчета следует, что ошибка исправлена и модель соответствует сформулированному свойству.

3.4. Методические рекомендации и указания по верификации автоматных моделей управляющих программ

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 *CTL*. Так как модель При использовании этого метода контрпример в виде пути в модели может быть непосредственно отображен в путь в системе конечных автоматов. Однако *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.

Сравнительные характеристики разработанных методов приведены в табл.

Таблица. Сравнительные характеристики разработанных методов верификации

Метод	Язык спецификации	Используемый верификатор	Построение модели Крипке	Поддержка состояния объекта управления	Поддержка зависимостей между событиями
Метод верификации автоматных программ с использованием верификатора *NuSMV*	CTL	NuSMV	Неявная	Нет	Нет
Метод верификации визуальных	LTL	SPIN	Неявная	При специальном описании	При специальном описании

автоматных моделей					
Метод верификации на основе эмуляции	LTL	Bogor	Автомат Крипке	Да	Да

Из 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 отсутствия состояний у объектов управления или возможности описания этих состояний в терминах *LTL*, рекомендуется использовать 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 получается подтверждение соответствия автоматной модели требованиям (в этом случае, процесс верификации оканчивается успешно), либо контрпример в терминах автоматной модели. Так как контрпример формулируется в терминах автоматной модели, а не в терминах модели Крипке, ошибки при восстановлении контрпримера из модели также отсутствуют.

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

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

ЗАКЛЮЧЕНИЕ

В результате исследований, выполненных на втором этапе работ по контракту, были предложены методы верификации автоматных моделей управляющих программ. Для этих методов были определены функциональные особенности и характеристики. Также были разработаны прототипы программных средств, реализующих разработанные методы, и сформулирована технология верификации автоматных моделей управляющих программ.

В первой главе были разработаны методы верификации автоматных моделей управляющих программ. Эти методы позволяют верифицировать автоматные модели более эффективно, чем существующие методы верификации программ общего вида. Таким образом, указанные методы обеспечивают возможность более эффективного решения таких задач.

Исследования, выполненные во второй главе, позволили оценить эффективность разработанных методов и область применения каждого из них. Разработаны рекомендации по применению указанных методов.

Практическая реализация разработанных методов позволила подтвердить их эффективность при применении к различным задачам.

Таким образом, были получены решения всех задач, поставленных в техническом задании на проведение второго этапа работы.

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

ИСТОЧНИКИ

1. *Васильева К. А., Кузьмин Е. В.* Верификация автоматных программ с использованием LTL // Моделирование и анализ информационных систем. Ярославль: ЯрГУ. 2007. Т. 14, № 1, с. 3-14.
2. *Кларк Э., Грамберг О., Пелед Д.* Верификация моделей программ: Model Checking. М.: МЦНМО, 2002.
3. *Robby, Dwyer M., Hatcliff J.* Bogor: A Flexible Framework for Creating Software Model Checkers. TAIC PART 2006, pp 3–22.
4. *Robby, Dwyer M., Hatcliff J.* Bogor: An Extensible and Highly-Modular Model Checking Framework, March 2003. In the Proceedings of the Fourth Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003). Technical Report, SAnToS-TR2003-3.
5. *Deng W., Dwyer M., Hatcliff J., Jung G., Robby, Singh G.* Model-checking Middleware-based Event-driven Real-time Embedded Software, March 2003. / Proceedings of the First International Symposium on Formal Methods for Components and Objects (FMCO 2002). Technical Report, SAnToS-TR2003-2.
6. *Гуров В. С., Мазин М. А., Шалыто А. А.* UniMod — Инструментальное средство для автоматного программирования // Научно-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
7. *Гуров В., Мазин М., Нарвский А., Шалыто А.* UML. SWITCH-технология. Eclipse // Информационно-управляющие системы. 2004. № 6, с. 12-17. http://is.ifmo.ru/works/uml-switch-eclipse/
8. *Новиков Ф. А.* Визуальное конструирование программ. // Информационно-управляющие системы. 2005. № 6, с. 9–22. http://is.ifmo.ru/works/visualcons/
9. *Шалыто А.А.* Технология автоматного программирования / Труды первой Всероссийской научной конференции «Методы и средства обработки информации». М.: МГУ. 2003, с. 528-535. http://is.ifmo.ru/works/tech_aut_prog/
10. *SPIN home page*. http://SPINroot.com
11. *Лифшиц Ю.* Верификация программ и темпоральные логики. Лекция №3 курса «Современные задачи теоретической информатики». http://download.yandex.ru/class/lifshits/lecture-note03.pdf
12. *Linear temporal logic.* http://en.wikipedia.org/wiki/Linear_temporal_logic
13. *Büchi automaton.* http://en.wikipedia.org/wiki/Büchi_automaton
14. *Васильева К. А., Кузьмин Е. В., Соколов В. А.* Верификация автоматных программ с использованием логики LTL. http://is.ifmo.ru/verification/_ltl_aut_ver_1.pdf
15. *Яковлев А. В., Лукин М. А., Шалыто А. А.* Реализация классической игры "Ним" на основе автоматного подхода. СПбГУ ИТМО, 2005. http://is.ifmo.ru/UniMod-projects/knim/
16. *Promela reference – ltl*. http://www.spinroot.com/spin/Man/ltl.html
17. *The Promela Language / http://www.dai-arc.polito.it/dai-arc/manual/tools/jcat/main/node168.html.*
18. *Классификация абстрактных автоматов*. http://ru.wikipedia.org/wiki/Классификация_абстрактных_автоматов
19. *UniMod home page*. http://UniMod.sourceforge.net/
20. *Шалыто А. А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука. 1998. http://is.ifmo.ru/books/switch/1

21. *Вельдер С. Э., Шалыто А. А.* Введение в верификацию автоматных программ на основе метода Model checking. СПбГУ ИТМО. 2006. http://is.ifmo.ru/verification/modelchecking/
22. Mealy machine. http://en.wikipedia.org/wiki/Mealy_machine
23. Symbolic Model Verifier. http://www.cs.cmu.edu/~modelcheck/smv.html
24. New Symbolic Model Verifier. http://nusmv.irst.itc.it/