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

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 I ЭТАПУ «ВЫБОР НАПРАВЛЕНИЯ ИССЛЕДОВАНИЙ И БАЗОВЫХ МЕТОДОВ»

ЛИСТОВ 107

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 докт. техн. наук, профессор	А. А. Шалыто	Отчет в целом
Ответственный исполнитель канд. техн. наук	Г. А. Корнеев	Отчет в целом
Заведующий кафедрой, канд. физ.-мат. наук, профессор	В. А. Соколов	Глава 1
Ведущий научный сотрудник, докт. техн. наук, профессор	В. В. Антипов	Глава 2
Ведущий научный сотрудник, канд. техн. наук	В. В. Киселев	Глава 2
Ведущий научный сотрудник, канд. техн. наук	Р. Н. Котляр	Глава 2
Ведущий научный сотрудник, канд. техн. наук	Ю. П. Московцев	Глава 1
Ведущий научный сотрудник, канд. техн. наук, доцент	В. А. Третьяков	Глава 2
Ведущий научный сотрудник, канд. техн. наук	Г. М. Файкин	Глава 1
Доцент, канд. физ.-мат. наук	Е. В. Кузьмин	Глава 1
Ассистент	В. С. Гуров	Глава 2
Руководитель ВТК, ассистент кафедры КТ	А. П. Мельничук	Раздел 2.2.
Член ВТК, профессор кафедры ИС	Е. Ю. Михайлова	Раздел 2.2.1.
Член ВТК, доцент кафедры КТ	В. Д. Наумчик	Раздел 2.2.2.
Член ВТК, доцент кафедры КТ	М. Ю. Осипов	Раздел 2.2.2.
Член ВТК, доцент кафедры КТ	А. Н. Воробьев	Раздел 2.2.1.
Член ВТК, доцент кафедры КТ	Ю. А. Щупак	Раздел 2.2.1.
Член ВТК, доцент кафедры КТ	С. В. Чириков	Раздел 2.2.2.
Член ВТК, доцент кафедры КТ	А. С. Сегаль	Раздел 2.2.1.
Член ВТК, доцент кафедры КТ	Д. Г. Шопырин	Раздел 2.2.2.
Член ВТК, доцент кафедры ИС	Д. А. Зубок	Раздел 2.2.1.
Член ВТК, ассистент кафедры ИС	В. В. Повышев	Раздел 2.2.2.
Член ВТК, ассистент кафедры ИС	В. В. Ильин	Раздел 2.2.1.
Член ВТК, ассистент кафедры ИС	М. Г. Холин	Раздел 2.2.2.
Магистрант	Б. Р. Яминов	Раздел 1.1.2.
Магистрант	С. Э. Вельдер	Разделы 2.1, 2.2.1 и 2.3.1.
Магистрант	М. А. Лукин	Разделы 1.1.1, 1.2 и 2.4.
Студент	М. Э. Дворкин	Раздел 2.2.
Студент	Е. А. Курбацкий	Раздел 2.3.
Студент	Ю. А. Беляева	Раздел 2.4.

РЕФЕРАТ

Излагаются результаты анализа методов верификации автоматных моделей и базовые методы верификации таких моделей с целью выбора направления исследований при проведении 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 требований, которым должны удовлетворять методы верификации автоматных моделей управляющих программ. Описывается структура модели Крипке для автоматных программ и методы ее построения для различных классов моделей. Изложены методы описания требований к автоматным моделям управляющих программ в терминах темпоральной логики.

ОГЛАВЛЕНИЕ

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 выбор направления исследований и разработка базовых методов верификации автоматных моделей управляющих программ. Задачами этапа являются:

1. Проведение анализа требований к методам верификации.
2. Формирование требований к методам верификации.
3. Разработка базовых методов верификации на моделях.

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*).

Как показывает опыт, дедуктивная верификация трудоемка и требует высококвалифицированных специалистов в области доказательства теорем и логического вывода.

Верификация на модели состоит из трех основных этапов.

1. Построение модели программы.
2. Задание требований в терминах выбранной темпоральной логики.
3. Верификация модели с целью проверки выполнения формализованных требований.

Выполнение первого этапа в общем случае достаточно трудно 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 количество состояний в ней должно быть не слишком большим, однако, существующие методы построения моделей для программ, написанных традиционным способом, дают очень большое количество состояний, так как в них обычно не разделяются управляющие и вычислительные состояния.

Затруднение вызывает также и выполнение второго этапа, так как для верификации требования должны быть сформулированы в терминах модели. При этом также встает вопрос об адекватности построенных требований исходной спецификации программы.

По сравнению с первыми двумя этапами, третий этап достаточно хорошо автоматизируется. Известны инструментальные средства для верификации моделей (верификаторы), в том числе свободные, например, *SPIN, CPN Tools* и *Bogor* (http://bogor.projects.cis.ksu.edu/). На этом этапе для программ общего вида при нахождении ошибки в модели часто возникает проблема переноса 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 и допускает простой переход к ней. При этом количество состояний в получаемой модели пропорционально количеству 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. При этом потребуется только доказательство корректности этого преобразования, а не доказательство корректности построения модели для каждой автоматной программы в отдельности. Это может существенно упростить процесс верификации рассматриваемого класса программ.

Патентные исследования, проведенные в рамках первого этапа работы (отчет о патентных исследованиях № 2007.08.31-01 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 превышать мировой уровень разработок в рассматриваемой области.

1. АНАЛИЗ И ФОРМИРОВАНИЕ ТРЕБОВАНИЙ К МЕТОДАМ ВЕРИФИКАЦИИ АВТОМАТНЫХ МОДЕЛЕЙ УПРАВЛЯЮЩИХ ПРОГРАММ

1.1. Анализ предметной области

1.1.1. Автоматные модели управляющих программ

В данном подходе к программированию выделяются источники входных воздействий и автоматизированные объекты, каждый из которых содержит систему управления СУ (систему взаимодействующих конечных автоматов) и объект управления ОУ. Объект управления формирует выходные воздействия, а также входные воздействия второго типа, реализующие обратную связь объекта управления с системой управления. Входные воздействия разделяются на события, действующие кратковременно, и входные переменные, вводимые путем опроса. Входные воздействия целесообразно реализовывать в виде входных переменных, а применять события — для сокращения времени реакции системы. При этом одно и то же входное воздействие может быть одновременно представлено и событием, и входной переменной. Группы входных и выходных воздействий в общем случае связываются с состояниями, выделяемыми в каждом автомате.

На рис. 1 изображена схема автоматизированного объекта. Парадигма автоматного программирования состоит в представлении программ как систем автоматизированных объектов.

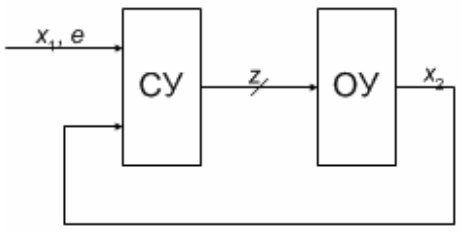
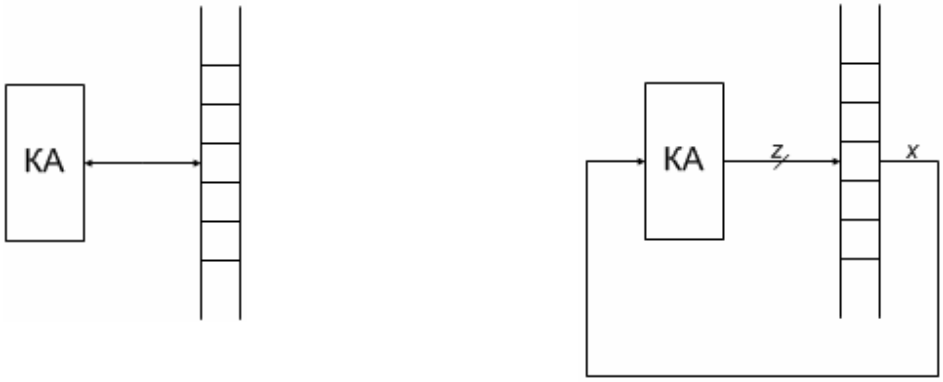


Рис. 1. Схема автоматизированного объекта

Данный подход является расширением машины Тьюринга, изображенной на рис. 2, на которой можно реализовать любой алгоритм.



А б

Рис. 2. Машина Тьюринга:
классическое (а) и модифицированное (б) изображения

Базовым понятием автоматного программирования является «*состояние*». Все состояния в автоматах должны быть явно выделены. Входные воздействия рассматриваются как средства для изменения состояний. Выходные воздействия автоматы могут формировать как в состояниях, так и на переходах между состояниями. Автоматы делятся на два типа: абстрактные и структурные. В абстрактных автоматах входные и выходные воздействия формируются последовательно. В структурных – «параллельно». В автоматном программировании применяются структурные автоматы.

Состояния также бывают двух типов: управляющие (автоматные) и вычислительные (неавтоматные). Управляющие состояния отражают качественные особенности поведения вычислительные – количественные.

В машине Тьюринга управляющее устройство с небольшим числом состояний, представляющее собой конечный автомат, может управлять практически бесконечным количеством состояний на ленте. В технологии автоматного программирования основное внимание уделяется управляющим состояниям. Управляющие состояния выделяются и перечисляются явно, и в дальнейшем при использовании термина "состояние" понимается управляющее состояние.

В качестве основного документа, определяющего структуру программы, как и при автоматизации технологических (и не только) процессов, в автоматном программировании используется схема связей. Схема связей определяет интерфейс автоматов и позволяет применять в

графах переходов и в реализующих их программах символьные обозначения. Для объектно-ориентированных программ эта схема может реализовываться диаграммой классов.

Поведение автоматов формализуется с помощью графа переходов. Графы переходов в наглядной для человека форме отражают переходы между состояниями, а также "привязку" выходных воздействий и других автоматов к состояниям и/или переходам. Для компактности входные и выходные воздействия обозначаются символами, а слова используются только для названий пронумерованных состояний.

На дугах и петлях графов переходов помечаются произвольными логическими формулами, в которых могут содержаться входные переходы и предикаты, проверяющие номера состояний других автоматов и номера событий. Вершины графов переходов содержат петли. Петли, на которых не выполняются никакие выходные действия, могут умалчиваться.

Имя автомата начинается с символа *A*, имя события — с символа *e* (от английского слова event — событие), имя входной переменной — с символа *x*, имя переменной состояния автомата — с символа *y*, а имя выходного воздействия — с символа *z*. После каждого из указанных символов следует номер соответствующего автомата или воздействия.

Приведем пример автоматного описания управляющих программ на основе одного автомата (рис. 3).

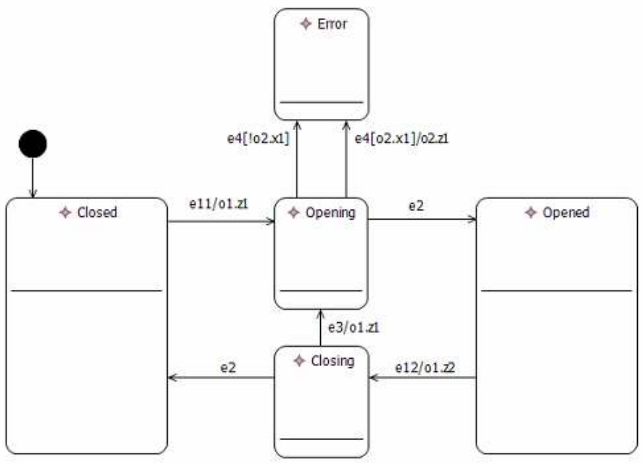


Рис. 3. Автомат, управляющий дверьми лифта

На этом рисунке приведена модель работы дверей лифта. Модель состоит из единственного автомата. 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, где *e12* – нажатие кнопки «закрыть», *o1.z2* – запуск закрытия дверей. Если какое-либо препятствие мешает дверям закрыться, то происходит событие *e3*, и двери снова открываются, 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 верификатора, поэтому, для простоты, в него можно попасть только из состояния *Opening*.

В общем случае автоматы рассматриваются не изолированно, а как составные части взаимосвязанной системы — системы взаимосвязанных автоматов.

Автоматы между собой могут взаимодействовать тремя способами. При этом может иметь место:

1. Вложенность. Один автомат вложен в одно или несколько состояний другого.
2. Вызываемость. Один автомат вызывается другим автоматом.
3. Взаимодействие по номерам состояний. Один автомат проверяет, в каком состоянии находится другой автомат.

Вложенность может рассматриваться как вызываемость с любым событием. Число автоматов, вложенных в состояние, не ограничено. Глубина вложенности также не ограничена.

Вложенные автоматы последовательно запускаются с передачей «текущего» события в соответствии с путем в схеме взаимодействия автоматов, определяемым их состояниями в момент запуска головного автомата. При этом последовательность запуска и завершения работы автоматов напоминает алгоритм поиска в глубину.

Вызываемые автоматы запускаются из выходных воздействий с передачей соответствующих «внутренних» событий. При этом автоматы могут запускаться однократно с передачей какого-либо события или многократно (в цикле) с передачей одного и того же события.

Систему автоматов можно представить как единый автомат с помощью *прямого произведения* [5]. Состояниями *автомата-произведения C* двух автоматов *A* и *B* являются пары состояний автоматов *A* и *B*. Таким образом, число состояний автомата *C* равно произведению числа состояний автоматов *A* и *B*.

Для построения переходов в *автомате-произведении* требуется проследить «параллельную» работу автоматов *A* и *B*. Каждый из этих автоматов в зависимости от входных действий совершает переходы. При этом, если один из автоматов при получении некоторого входного воздействия не может перейти ни в какое состояние, автомат *C*, то также не сможет перейти ни в одно состояние.

Рассмотрим пример автоматной программы, состоящей из нескольких автоматов. Для этого добавим к автомату, управляющему дверьми лифта, изображенному на рис. 3, автомат, управляющий движением лифта, изображенный на рис. 4.

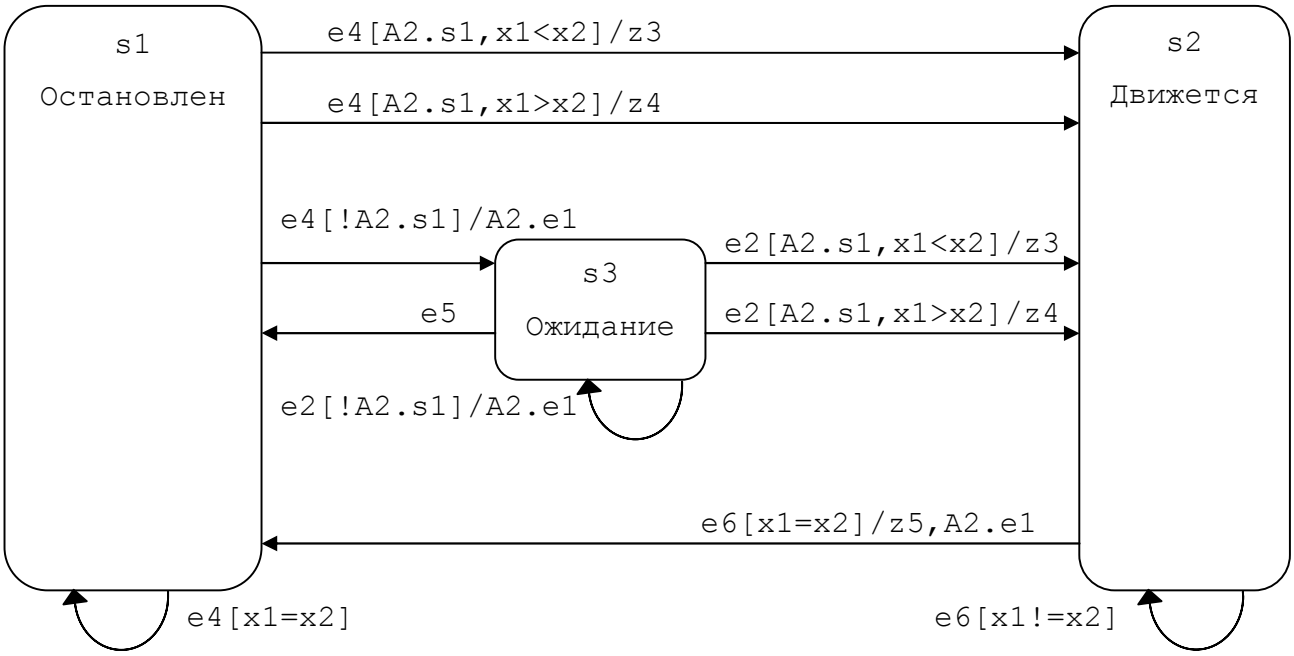


Рис. 4. Автомат, управляющий движением лифта

Автомат, который управляет движением лифта, назовем `A1`, а автомат, управляющий дверьми лифта — `A2`.

В автомате `A1` использованы следующие обозначения:

- `A2.e1` – вызов автомата `A2` с событием `e1`;
- `A2.s1` – условие, выполняющееся если автомат `A2` находится в состоянии `s1`;
- `e4` – событие, происходящее, когда нажата кнопка этажа назначения лифта;
- `e5` – событие, происходящее, когда нажата кнопка «Отмена»;
- `e6` – 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.

На рис. 5 изображена упрощенная схема связей системы, управляющей лифтом, включая его двери.

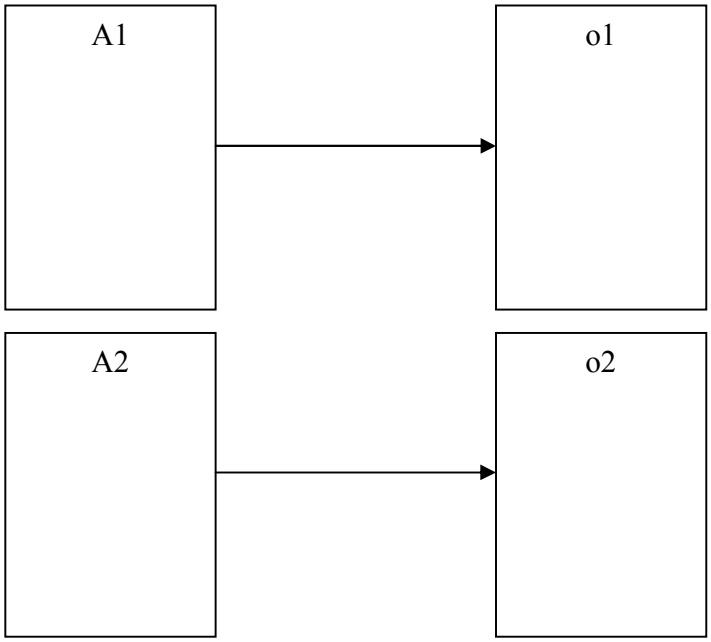


Рис. 5. Упрощенная схема связей

На схеме присутствуют два автомата и два объекта управления. В данных обозначениях `o1` является объектом управления «лифт». `o2` – объект управления «двери лифта». Данная схема обозначает, что автомат управления лифтом `A1` связан только с объектом управления «лифт» `o1`, а автомат управления дверьми лифта `A2` связан только с объектом управления «двери лифта». Связь между автоматами на схеме не показана.

1.1.2. Методы проверки управляющих программ

В настоящее время программные системы используются повсюду: в медицине, транспорте, бытовой технике, космических полетах. Таким образом, все больше ответственности возлагается на программы, управляющие теми или иными устройствами.

Ошибки в управляющих программах могут приводить к неприятностям. Например, был случай, когда в результате ошибки в бортовом компьютере новейшего бронированного автомобиля высокопоставленное лицо в центре Парижа не смогло выбраться из машины, пока не приехали спасатели. Ошибки могут приводить и к катастрофическим последствиям, как это случилось с ракетой «Ариан-5», взорвавшейся через сорок секунд после запуска.

В результате постоянно увеличивающегося объема автоматизации в жизни человека, все более острой становится проблема обнаружения и устранения ошибок в управляющих программах. И чем раньше ошибки будут найдены, тем меньше вреда они принесут.

Для проверки работы управляющих программ используются следующие методы:

- *тестирование*;
- *имитационное моделирование*;
- *дедуктивный анализ*;
- *верификация на модели*.

Тестирование является самым простым и распространенным методом проверки работы систем. Общий принцип тестирования заключается в том, чтобы работающей системе подавать на вход определенные входные значения, и проверять, что на выходе получаются требуемые выходные значения. Положительными качествами тестирования являются, во-первых, простота а, во-вторых, надежность, так как тестируется обычно сама работающая система. Поэтому гарантируется, что если тесты выполнялись при тестировании, то они будут выполняться и при реальной работе. Недостатком тестирования является неполнота. Тесты проверяют функциональность системы лишь на некоторых примерах, что, естественно, не гарантирует, что система будет работать на *всех* примерах. С увеличением числа тестов можно быть достаточно уверенным в корректности работы программы. Однако гарантии корректности при этом не получить. Обычно стремятся, чтобы тесты, по крайней мере, покрывали все переходы в спроектированной системе. Например, желательно, чтобы каждая строка кода выполнялась хотя бы в одном тесте. Для эффективного тестирования создана технология программирования, называемая *Test Driven Development*, и существуют утилиты, указывающие на

части кода, которые не выполняются ни при одном тесте. При этом улучшается качество тестирования, однако не решают проблему проверки правильности работы управляющей системы принципиально.

Имитационное моделирование сходно с тестированием, однако проверяется не работающая система, а модель, имитирующая ее работу. Отсюда и название метода. Имитационное моделирование используется в тех случаях, когда тестирование реальной системы не представляется возможным или требуется проверить корректность проекта до создания прототипа. Таким образом, при неточной модели возможно возникновение ситуации, когда некоторый тест прошел этап имитационного тестирования, но на реальной системе не выполнится. Поэтому в этом методе очень важно точно перенести логику работы системы в ее модель.

Дедуктивный анализ – это формальное доказательство свойств системы. Для управляющей системы строится набор аксиом, из которых затем с помощью формальной логики пытаются доказать выполнимость этих свойств. Преимущество дедуктивного анализа в том, что в случае успешного доказательства можно с точностью утверждать, что свойство выполняется *всегда*. При этом система может даже иметь бесконечное число состояний. Недостаток дедуктивного анализа состоит в том, что он требует большой ручной работы и высокой квалифицированности специалистов, его применяющих. При этом сложно заранее определить, сколько времени потребуется для доказательства утверждения или для его опровержения. Поэтому дедуктивный анализ применяется крайне редко и только в тех областях, в которых действительно критически важна корректность управляющей системы. В них, поочередно применяя те или иные правила вывода, пытаются из аксиом вывести заданное предположение. Однако обычно количество утверждений, выводимых из заданных аксиом, слишком велико, для того, чтобы найти среди этих утверждений требуемое. Поэтому человеку приходится «помогать» средству автоматического доказательства теорем (при его наличии), подсказывая, какое правило применить на очередном шаге. Таким образом, человек должен представлять, как доказывается заданное утверждение. Поэтому такой процесс доказательства нельзя назвать автоматическим. Необходимость в больших человеческих и временных ресурсах делает дедуктивный анализ неприменимым для большинства прикладных задач.

Верификации на модели является практически полностью автоматическим методом для проверки свойств систем с *конечным* числом состояний. Как следует из названия метода, он работает не с реальной системой, а с ее *моделью*. Для проверяемой системы сначала строится формальная

модель, описывающая ее поведение. Затем для нее формулируется спецификация – утверждения, истинность которых требуется проверить. После этого выполняется автоматическая верификация, в результате которой либо доказывается, что модель удовлетворяет спецификации, либо это опровергается. Опровержение представляет собой набор действий над моделью, которые приводят нарушению спецификации.

Сравним верификацию на модели с другими методами. Верификация на модели лучше тестирования, во-первых, тем, что проверяет не просто некоторый набор пар вход-выход, а все возможные варианты входных данных. Достоинство метода верификации на модели по сравнению с тестированием состоит в том, что в случае невыполнения спецификации выводится сценарий ее нарушения, анализируя который проще найти причину ошибки, чем в случае простого ответа «нарушается». Стоит заметить, что нарушение сценария может возникать не только в случае ошибочной модели системы, но также в случае неверной спецификации. Оба типа ошибок легко находятся с помощью анализа сценария. Стоит помнить, что процесс отладки систем итеративен: после обнаружения ошибки требуется ее исправить, после чего заново все проверить. Автоматизация проверки на порядки ускоряет этот процесс, и позволяет следовать девизу «компьютер должен считать, а человек – думать».

Если сравнивать метод верификации на модели и дедуктивный анализ, то верификация на модели уступает тем, что применима только к моделям с конечным количеством состояний. Тем не менее, существует большое число задач, для которых это ограничение выполняется. Кроме того, многие задачи с бесконечным числом состояний можно свести к конечным, поставив определенные ограничения, которые не сильно изменяют смысл задачи. Преимущество метода верификации на модели в том, что проверка проходит полностью автоматически и для ее выполнения не требуется особых знаний, опыта и времени. За счет этого верификация на модели применима в гораздо большем наборе областей, где используются управляющие системы, нежели дедуктивный анализ.

1.1.3. Верификация на модели

Инструментальные средства, реализующие метод верификации на модели, называются *верификаторами*. Существует большое число верификаторов, у каждого из которых свои особенности. Однако, в общем случае процесс верификации можно разбить на три этапа: моделирование, спецификация и верификация.

Моделирование – это построение модели системы, или ее формализация для того, чтобы верификатор смог с ней работать. Для формализации каждый верификатор имеет свой *входной язык*. Входные языки верификаторов предназначены для того, чтобы проектирующий систему человек сам выделил важные детали поведения и абстрагировался от неважных деталей реализации. Языки верификаторов различаются по своей выразительной способности, удобству и уровням абстракции. Существуют верификаторы, которые принимают модель в виде программы на языке, похожем на обычный язык программирования. Примером такого верификатора является верификатор *SPIN*. Синтаксис его входного языка *Promela* намеренно создан похожим на синтаксис языка программирования *C*, для того, чтобы было проще переносить логику работы программы, написанной на *C*, в модель верификатора.

Существует верификатор *SMV*, который принимает модель в виде раскрашенной *сети Петри*. Такой входной язык удобен для моделирования, например, бизнес-процессов.

Также известен верификатор *Bogor* с гибким входным языком. В нем можно создавать собственные абстракции, а также писать код как на языке высокого уровня, похожем на язык программирования *Java*, так и на языке низкого уровня, описывая каждое состояние системы.

Такое разнообразие входных языков верификаторов связано с тем, что определенные входные языки оказываются более удобными для соответствующих типов систем. Этап моделирования системы очень важен, так как неверная или неполная модель может привести к ошибкам. В результате неточности моделирования верификатор может найти в модели ошибки, которые не существуют в исходной системе, или, что еще хуже, провести успешную верификацию, в то время, как исходная система не удовлетворяет предъявленным требованиям.

В общем случае полученную модель верификатор разбивает на элементарные состояния и строит граф переходов между ними. Этот граф называется *моделью Крипке*. В первых верификаторах он строился явно как набор вершин (элементарных состояний) и переходов между ними, но в современных верификаторах для экономии памяти такой граф не строится явно. От размеров построенного графа зависит время верификации, и в случае чересчур большого числа элементарных состояний модели, верификация может даже оказаться невозможной из-за нехватки памяти или огромного времени, требующегося для верификации. Поэтому верификаторы не принимают модель в виде программы на языке программирования, таком как, например, *Java*. Такие программы оказываются слишком подробными, и если их разбивать их элементарные состояния, то полученная

модель Крипке в большинстве случаев окажется необозримой. Тем не менее, в данном направлении ведутся активные исследования.

Спецификация – это процесс формирования требований к модели, выполнимость которых верификатор должен проверить. Во-первых, требования можно формировать, в виде элементарных проверок прямо в коде модели (*assertion*). С помощью них можно проверить, например, что в неком состоянии некоторая переменная всегда принимает заданное значение. Такие требования задают ограничения на элементарные состояния системы. Большинство верификаторов позволяют формулировать утверждения в виде формул *темпоральной логики*. Такие формулы задают ограничения на поведение системы во времени, что делает ограничения гораздо более выразительными, чем элементарные проверки. Например, можно сформулировать утверждение вида «если было выполнено условие 1, то *когда-нибудь в будущем* выполнится условие 2». Такие утверждения, вообще говоря, описывают бесконечные истории работы модели, и возможность их проверки основывается на предположении о конечности количества элементарных состояний модели.

Верификация производится с помощью специальных алгоритмов, которые проверяют, что построенная модель Крипке удовлетворяет заданной темпоральной формуле. В результате работы такого алгоритма либо доказывается, что для любой истории формула выполняется, либо находится опровержение – история (набор переходов в модели Крипке), которая приводит к нарушению темпоральной формулы. Отметим, что перед верификатором стоит также задача трансляции полученного набора переходов в модели Крипке в термины исходной модели, написанной на входном языке верификатора. Только после этого пользователь верификатора сможет увидеть сценарий в построенной им модели, на котором нарушается свойство, и исправить модель или спецификацию.

1.1.4. Обзор методов верификации на модели

Ниже рассматриваются основные идеи, используемые при верификации на модели. Более широкий круг подходов описан в отчете о патентных исследования, выполненном в рамках настоящей работы.

Существует несколько типов темпоральных логик. При этом в верификации наиболее распространены два из них: *CTL* – логика ветвящегося времени и *LTL* – логика линейного времени.

Алгоритм автоматической проверки темпоральных утверждений на модели был впервые предложен в 80-х годах Э. Кларком и Э. Эмерсоном. Они построили алгоритм верификации формул логики *CTL*, который имеет полиномиальную сложность относительно размеров модели Крипке и

длины темпоральной формулы. Позднее этот алгоритм был улучшен до линейной сложности относительно произведения длины формулы на размер модели Крипке [20] и был реализован в верификаторе *EMC*. Этот верификатор может проверять модели Крипке, содержащие большое число состояний.

В 1985 году был создан первый алгоритм проверки формул *LTL*, основанный на табличном методе. Со временем улучшались как алгоритмы проверки темпоральных формул на модели, так и компьютерная техника. В результате размеры моделей, с которыми могли работать верификаторы, постоянно увеличивались.

Кроме темпоральных формул было предложено задавать ограничения в виде автоматов. С помощью несложных преобразований можно получить из модели Крипке автомат, и тогда получается, что и спецификация и модель задаются одной математической моделью, что дает возможность применять результаты из теории автоматов. В 1983 году М. Варди и П. Вольпером было показано, что любую формулу логики *LTL* можно преобразовать в эквивалентный автомат Бюхи – один из простейших конечных недетерминированных автоматов над бесконечными словами [56]. Эквивалентность означает, что любая история, на которой выполняется исходная темпоральная формула, допускается также и построенным автоматом Бюхи, и наоборот. Тогда для доказательства выполнения исходной формулы на исходной модели Крипке требуется доказать включение языка автомата, построенного из модели Крипке, в язык автомата Бюхи, построенного по темпоральной формуле.

Подход М. Варди и П. Вольпера к проверке на модели поставил две задачи: преобразование формулы в автомат и проверка пустоты автомата.

Первую задачу решает алгоритм преобразования *LTL*-формулы в автомат, предложенный в работе [27]. Алгоритм предназначен для осуществления проверки «на лету», следовательно, процесс генерации автомата-спецификации может идти во время генерации модели и быть в процессе скорректированным необходимым образом. Алгоритм даёт на выходе обобщённый автомат Бюхи, который может быть преобразован в классический автомат Бюхи.

Для решения второй задачи группой учёных [23] был предложен алгоритм поиска цикла с допускающим состоянием, достижимым из начального, который для графа размера n требует $O(n)$ памяти с произвольным доступом. Таким образом, это позволяет делать проверку пустоты пересечения двух автоматов Бюхи с использованием линейной памяти. Предложенный алгоритм

использует два поиска в глубину. Так как нет необходимости держать в памяти весь исследуемый автомат, то метод можно использовать для верификации «на лету». Позднее М. Варди и П. Вольпер доказали [56], что проверка пустоты автомата Бюхи *NLOGSPACE*-полна [1].

Для проверки формул логик ветвящегося времени используются автоматы на бесконечных деревьях (*automata on infinite trees*). Например, М. Варди и П. Вольпер в статье [55] использовали их для верификации формул логики *ADPDL* (вариант *DPDL*, *Deterministic Propositional Dynamic Logic*).

Главной проблемой верификации является так называемая проблема «комбинаторного взрыва». Она связана с тем, что при наличии в системе параллельных процессов, число элементарных состояний системы растет экспоненциально от числа элементарных состояний каждого процесса. Это происходит из-за того, что в каждый момент времени управление может передаться в другой процесс, который осуществит свой очередной переход. В результате, верификатору приходится перебирать все возможные цепочки переходов. Поэтому даже несложные на первый взгляд модели могут оказаться слишком сложными для верификации.

В 1986 году Р. Бриан в работе [11] предложил новое компактное представление булевых функций в виде ациклических ориентированных графов и алгоритмы для работы с ними. Это представление является канонической формой — у каждой функции есть свой уникальный граф, причем размер этого графа минимален. В работе Р. Бриана были изложены алгоритмы для работы с новым представлением, такие как композиция, подстановка значения вместо одного из аргументов, сложение и умножение.

На основе представления Р. Бриана, названного упорядоченные двоичные разрешающие диаграммы (*Ordered Binary Decision Diagram*, *OBDD*), К. МакМиллан и соавторы в работе [13] разработали методику неявного представления пространства состояний вместо списков смежности. Каждое состояние системы может быть закодировано как вектор булевых переменных. При этом отношение между двумя состояниями можно представить в виде булевой функции. *OBDD* не всегда решают проблему комбинаторного взрыва, но многие реальные системы могут быть верифицированы с их помощью. В работе [13] говорилось о верификации систем с большим числом состояний. Был описан алгоритм символьной проверки на модели для формул μ-исчисления, а также методы трансляции в μ-исчисление формулы логик *CTL* и *PTL*. Исследователи продемонстрировали применения своего алгоритма для проверки выполнимости формул логики линейного времени,

сильной и слабой эквивалентности конечных систем переходов, и включения языков для конечных ω-автоматов. Они также провели верификацию простой синхронной конвейерной схемы.

К. МакМиллан разработал систему верификации *SMV*, которую описал в своей диссертации [44]. Входной язык *SMV* предназначен для описания модульных иерархических параллельных систем с конечным числом состояний, как синхронных, так и асинхронных. Язык допускает булев и перечислимые типы данных. Программа на этом языке может быть снабжена спецификациями в виде формул языка *CTL*. Верификатор строит по программе модель и, используя поиск в пространстве состояний, проверяет ее на соответствие спецификации. Если модель не соответствует спецификации, то система выдает контрпример в виде трассы выполнения. Протокол для согласования кеша распределенной памяти мультипроцессора *Encore Gigamax* был смоделирован на языке *SMV* и проверен с использованием символьной верификации в комбинации с индукцией. Система была представлена как асинхронная композиция синхронных конечных автоматов. Для упрощения верификации была использована абстракция. При этом, не был точно воспроизведен механизм замены кеша, что добавило еще больше недетерминизма. За несколько минут было найдено несколько ошибок, которые до этого не были обнаружены другими методами из-за большого размера пространства состояний модели.

В диссертации обсуждались методики символьной проверки на модели, часть из которых была реализована в системе *SMV*. Техника поиска в пространстве состояний при помощи *OBDD* может быть использована и при проверки на модели формул логики *CTL*. В работе также описан класс схем, для которых графы переходов могут быть эффективно представлены в виде *OBDD*. Диссертация также содержит альтернативный символьному подход к проблеме «комбинаторного взрыва», основанный на развертывании сетей Петри в специальную структуру «сеть расширения» (*occurrence net*).

В работах [15, 14] описана улучшенная методика символьной проверки, которая позволяет верифицировать системы с большим числом состояний, чем позволяет метод, описанный в работе [19]. Она основана на так называемых *группированных отношениях перехода* (*partitioned transition relations*). Основная идея состоит в том, чтобы представлять отношение перехода не в виде одной монолитной функции, а в виде множества функций, по одной для каждой переменной состояния. Часто имеет смысл объединять некоторые из этих функций в конъюнкции или дизъюнкции.

Методика подходит как для синхронных, так и для асинхронных схем. Время, затрачиваемое на верификацию, зависит полиномиально от размера схемы.

В 1992 году в университете Карнеги-Меллона [21] построили точную модель протокола согласования кэша, описанного в черновом варианте стандарта *IEEE Futurebus+* (стандарт *IEEE 896.1-1991*) и проверили ее на соответствие спецификации. Для формализации и верификации была использована система *SMV*. Модель протокола на языке *SMV* состояла из 2300 строк кода, не считая комментариев. Некоторые детали протокола были скрыты из-за того, что не были еще четко прописаны в стандарте (были помечены словом «возможно»). Исследователи отметили, что одной из самых полезных частей проекта является модель мостов шины, которые соединяют их в иерархических конфигурациях системы. Эти компоненты не были описаны в стандарте детально, но без их моделирования невозможно анализировать иерархические конфигурации, так как именно в них возникает наиболее сложное поведение. С использованием *SMV* и модели мостов стало возможным найти некоторые потенциальные ошибки в иерархическом протоколе. Также была найдена ошибка и в протоколе для одной шины.

Опишем, как с помощью эти диаграмм можно представлять модели Крипке. *OBDD* используются для кодирования функций, зависящих от набора булевых переменных. Для того, чтобы перейти к функциям от переменных, принимающих произвольный конечный набор значений, используется следующая простая идея: для такой функции можно каждое значение переменной представить в виде набора бит (двоичного числа). Затем можно соединить булевы представления всех переменных и рассматривать каждый бит как новую булеву переменную. Таким образом, будет получена функция от некоторого множества булевых переменных, которую можно будет представить при помощи указанной диаграммы. Далее, для кодирования множества элементов при помощи *OBDD* строится функция, которая в зависимости от элемента возвращает значение `true`, если элемент принадлежит множеству, и значение `false` в противном случае. Таким образом, модель Крипке можно закодировать следующим образом: построить *OBDD*, кодирующую множество состояний модели Крипке, а также построить *OBDD*, возвращающую по паре состояний значение `true`, если существует переход в модели Крипке из первого состояния во второе, и значение `false` в противном случае.

Проблема комбинаторного взрыва особенно остро стоит в области верификации распределенных асинхронных систем. Если рассматривать все возможные последовательности

событий в такой системе, то число состояний в модели растет экспоненциально. Однако исследования в этой области показали, что рассмотрение всех последовательностей выполнения не всегда является необходимым для верификации. Действительно, в асинхронных системах для двух событий часто не имеет значения, в каком порядке они произошли. На этом базируются методики *редукции частичных порядков* (*partial order reduction*). Впервые они независимо появились в работах А. Валмари [51, 52, 53], П. Годфруа [29] и П. Вольпера [31].

При наличии параллельных процессов в верифицируемой модели, в элементарные состояния модели входят все возможные варианты последовательного выполнения этих процессов. В результате число состояний модели Крипке возрастает на порядки. Каждый процесс задает частичный порядок на события. При этом число вариантов выполнения программы – это число способов дополнить частичный порядок. Однако можно заметить, что обычно большинство различных последовательностей выполнения не различаются с точки зрения верифицируемой формулы, и перебирать все нет необходимости. Например, если работают два параллельных процесса, порядок выполнения которых не влияет на результат программы, а в спецификации используется лишь этот результат, то нет смысла перебирать все возможные варианты последовательного выполнения, так как для них результат будет одинаков — система все равно придет в одно и то же глобальное состояние. Однако это неверно, например, если верифицируемая спецификация затрагивает порядок выполнения процессов. В этом случае может понадобиться перебрать все возможные варианты последовательного выполнения процессов. Таким образом, можно построить отношение эквивалентности между порядками выполнения, эквивалентными с точки зрения спецификации, и для каждого класса эквивалентности использовать лишь одного его представителя. В результате сильно уменьшится число состояний в модели Крипке. Поэтому память и время, необходимые для верификации, тоже уменьшатся.

Редукция частичных порядков основывается на понятиях устойчивых и спящих множеств.

Устойчивые множества (*persistent sets*) [28, 30] в начале использовались для определения взаимных блокировок (deadlock). Множество переходов T, которые возможных из состояния s, называется устойчивым, если для всех переходов не из множества T из s или из состояний, которые достижимы из состояния s по переходам не из множества T, верно, что они независимы с переходами из множества T. Можно показать, что в ходе поиска по графу переходов для каждого состояния достаточно обследовать только устойчивое множество переходов.

Спящие множества (*sleep sets*) [28, 29] определяются для каждого состояния, как множество переходов, которые не будут обследованы. Например, если в каком-либо состоянии *s* возможны два независимых перехода *a* и *b* и переход *b* ведет в состояние *t*, то зная, что переход *a* из *s* обследован, можно не исследовать его из состояния *t* (он относится к спящему множеству для состояния *t*), так как последовательности переходов *ab* и *ba* обе ведут в одно и то же состояние. Таким образом, спящие множества предназначены для того, чтобы обходить все достижимые состояния системы без перебора всех переходов.

Устойчивые множества и спящие множества, хотя и различаются между собой, основаны на сходных идеях. Отличный от них метод разработан Ж. Хольцманом и Д. Пеледом [34] и используется в системе верификации *SPIN* [33].

Кроме применения *OBDD* и редукции частичных порядков, выделяют также следующие методы, позволяющие применять верификацию на модели даже для систем, не поддающихся верификации напрямую:

- декомпозиция;
- абстракция;
- симметрия;
- индукция.

Декомпозиция применяется для верификации сложных систем, разделенных на модули. Вместо того чтобы верифицировать всю систему сразу, можно верифицировать каждый модуль в отдельности. При верификации каждого модуля делается предположения относительно других модулей, с которыми он взаимодействует. Затем проверяется, что сделанные предположения действительно выполняются.

Абстракция – это, пожалуй, самый эффективный способ упрощения задачи для верификатора. Суть этого метода состоит в абстрагировании от несущественных (как кажется пользователю) деталей реализации. За счет этого число состояний модели Крипке уменьшается. Здесь важно не потерять важные детали, которые могут повлиять на результат верификации. Некоторые верификаторы, такие, как, например, верификатор *Bogor*, предоставляют пользователю возможность создавать свои типы во входном языке. За счет этого может быть создана новая абстракция. Например, если в системе используется сущность «множество», в которую можно добавлять и удалять элементы, а также проверять их наличие, то запрограммировать такое, казалось бы, простое поведение будет достаточно

сложно, если пользоваться только возможностями такого языка, как, например *Promela* (входной язык верификатора *SPIN*). В верификаторе *Bogor* можно один раз создать новый тип, который будет описывать новую сущность, после этого свободно пользоваться им при формализации модели для верификатора.

Симметрия как метод упрощения верификации заключается в следующем наблюдении. Во многих системах содержатся одинаковые элементы, например, несколько одинаковых процессов. Такая симметрия может порождать одинаковые подграфы в модели Крипке. Этот факт можно использовать для редукции графа и за счет этого добиться сокращения числа состояний. Иногда можно преобразовать исходную симметричную модель так, что суть ее работы останется та же, но модель Крипке окажется проще.

Индукция позволяет верифицировать целые семейства систем с конечным числом состояний. Например, протокол взаимного исключения должен работать для произвольного числа процессов, что порождает бесконечное семейство конечных систем. Для верификации строится некая инвариантная система, представляющая работу произвольного элемента семейства систем. Верификация такой системы будет означать верификацию всего семейства исходных систем.

Методы проверки на модели, основанные на проверке выполнимости булевых формул (*SAT*), являются альтернативой подходу, использующему *OBDD*. Прогресс в методах решения задачи *SAT* [32, 41, 40] позволяет разработать эффективные алгоритмы верификации.

Ограниченная проверка на модели (*bounded model checking*, *BMC*) была предложена в университете Карнеги-Меллона [9]. Основная идея метода состоит в том, чтобы пытаться найти контрпример заданной длины *k*, сгенерировав формулу, которая выполняется только если такой контрпример существует. Формула является конъюнкцией трёх групп условий: условия на начальные состояния, условия отношений переходов и условия на конечные состояния. Генерация формулы из спецификации на языке *LTL* может быть выполнена за полиномиальное время. Контрпример находится достаточно быстро и при этом имеет минимальную длину. Метод требует меньше памяти, чем алгоритмы с использованием *OBDD*. Проблема ограниченной проверки состоит в том, что она может установить только отсутствие контрпримера заданной длины, но не отсутствие контрпримера вообще.

В работе [26] были скомбинированы *BDD* и *SAT*, что позволило обрабатывать большие по размеру графы.

Метод неограниченной проверки на модели (*unbounded model checking*) разработанный К. МакМилланом [43] обобщает ограниченную проверку и интерполяцию. Предложенный К. МакМилланом алгоритм в цикле запускает процедуру ограниченной проверки для какой-то длины контрпримера. Если формула невыполнима (контрпример не найден), то формула разбивается на две части: первая соответствует начальным условиям и первому условию перехода, а вторая — остальной формуле. Для двух частей считается интерполянт, который характеризует состояния, достижимые из начальных за один шаг, но из которых нельзя дойти до конечных состояний. Это новое множество используется для следующего запуска процедуры ограниченной проверки.

Способность решателя *SAT* выдавать доказательство того, почему формула невыполнима, в работе [8] используется в комбинации с *детализацией абстракции* (*abstraction refinement*). Предложенный метод верификации формул логики *LTL* чередует ограниченную проверку с неограниченной. На каждой итерации алгоритма запускается *BMC* для некоторого k. Если он не находит контрпример, то, основываясь на выданном решателем *SAT* доказательстве невыполнимости, строится абстрактная модель, на которой запускается процедура обычной (неограниченной) проверки.

Ещё один метод, использующий детализацию абстракции, был предложен Э. Кларком и соавторами в работе [17]. Алгоритм вначале запускает неограниченную проверку на абстрактной модели. Если свойство выполнено, то алгоритм завершается, а если не выполнено, то на основе контрпримера генерируется формула для решателя *SAT*. В случае существования переменных, для которых формула выполняется, контрпример на модели *SAT*. Если формула невыполнима, то на основе доказательства, выданного решателем *SAT*, абстракция уточняется. Это происходит за счёт добавления в модель некоторых переменных, которые были спрятаны.

Ещё одна интересная техника проверки на модели использует конъюнктивно-нормальные формы. В работе [42] было показано, что, модифицировав некоторые *SAT*-алгоритмы, можно получить способ удаления квантора всеобщности из формулы, получив формулу в *КНФ*. Становится возможным преобразование *CTL* формулы **AX** p, где p – булева формула, в эквивалентную формулу в *КНФ*. Таким образом, любая *CTL* формула вычисляется с использованием неподвижных точек.

Более подробные обзоры методов верификации с использованием *SAT* можно найти в работах [7, 10].

Наличие такого количества алгоритмов и приемов верификации на модели позволяет сделать вывод о перспективности данного подхода. Отметим, что если задача проверки семантических свойств программы алгоритмически неразрешима [5], то проверка свойств модели может быть произведена за конечное время. При этом благодаря появлению новых алгоритмов в области верификации на модели постоянно уменьшаются требования к объему памяти и времени, требующихся для верификации. Кроме того, возможность широкого выбора средств верификации и языков задания спецификации заметно расширяет возможности метода верификации на модели. Так, например, для описания простых свойств модели можно использовать одну из простейших темпоральных логик. Это позволит уменьшить время проверки данного свойства для конкретной модели. Так как автоматные модели являются весьма простыми и имеют конечное число состояний, то из изложенного следует, что верификацию таких моделей можно эффективно проводить на основе рассмотренных методов.

1.1.5. Верификатор *SPIN*

В мире существует большое число верификаторов. Одним из самых известных, является верификатор *SPIN* [33].

Верификатор *SPIN* исходно был создан для верификации протоколов. Он позволяет создавать модели, содержащие несколько процессов, взаимодействующих с помощью глобальных переменных или с помощью каналов с ограниченным размером буфера.

Существует консольная версия верификатора *SPIN*. Однако обычно с ним работают через удобную для пользователя оболочку – *XSPIN*.

При использовании этого верификатора сначала разрабатывается модель программы на языке *Promela*. В моделях на этом языке создаются процессы (они характеризуются модификатором `proctype`), которые взаимодействуют между собой. В языке *Promela* существует несколько операторов недетерминированного выбора, что позволяет создавать недетерминированные модели. Затем формируется спецификация для модели при помощи формул темпоральной логики *LTL*. После этого запускается так называемый *имитационный режим* (*interactive simulation*). В нем написанная модель запускается со случайными значениями в недетерминированных условиях и проверяется выполнение спецификации. В таком режиме проверяется лишь один путь развития модели, и это предназначено не для верификации, а для проверки пользователем, что разработанная им модель делает именно то, что он хотел. Тем не менее, возможно, что принципиальные ошибки в модели или

спецификации будут найдены уже на этом этапе. Для полной верификации по модели и спецификации генерируется специальная программа на языке *C*, которая производит полный перебор путей в пространстве состояний модели для поиска нарушений спецификации. В результате работы программы либо выводится сообщение об удачном завершении верификации, либо генерируется сценарий ошибки, который используется в имитационном режиме для представления истории ошибки. Схема работы верификатора изображена на рис. 6.

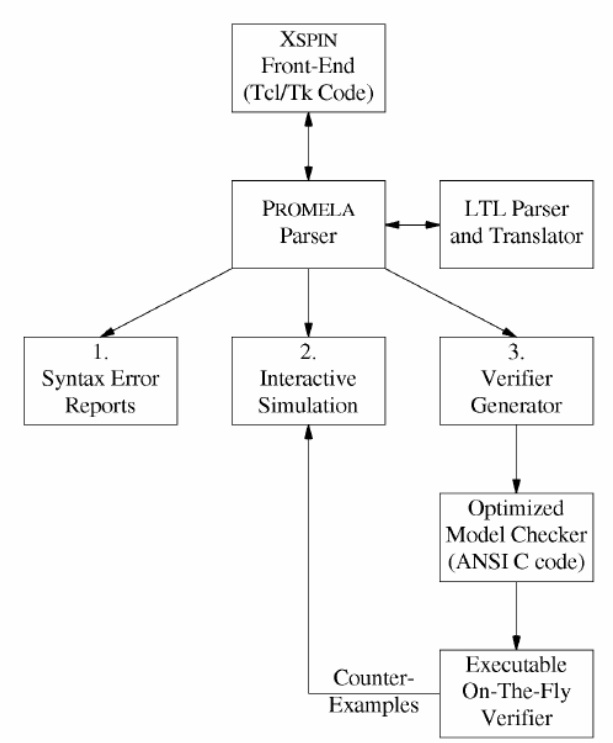


Рис. 6. Схема работы верификатора *SPIN*

Верификатор *SPIN* может работать со спецификациями заданными в виде автомата Бюхи. В случае если спецификация задана в виде *LTL*-формулы, запускается специальная утилита, которая преобразует ее в эквивалентный автомат Бюхи.

Автоматы Бюхи [4] – это одни из простейших конечных автоматов над бесконечными словами — ω-автоматов. Путь в автомате Бюхи называется допускающим, если он бесконечное число раз проходит через хотя бы одно из его допускающих состояний. Автомат Бюхи допускает слова, порождаемые допускающим путем. Оказывается, любую формулу в логике *LTL* можно преобразовать

в недетерминированный автомат Бюхи [4]. Это означает, что множество путей, удовлетворяющих формуле и допускаемых автоматом Бюхи, будут совпадать. К примеру, *LTL*-формула *FGp* («Существует состояние, после которого всегда будет выполняться *p*») преобразуется в автомат Бюхи, изображенный на рис. 7.

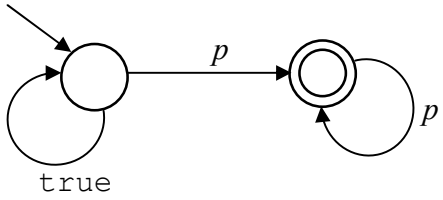


Рис. 7. Автомат Бюхи для формулы FG¬ p

Этот автомат является недетерминированным, так как он недетерминированно выбирает момент, с которого условие *p* всегда будет выполняться, и переходит в допускающее состояние. Этот автомат не является неполным: при условии $\neg p$ из допускающего состояния нет перехода. Это следует трактовать как переход в сток – недопускающее состояние с циклом в себя. Таким образом, если в автомате не определен переход по очередному условию из текущего состояния, то слово считается недопускающим.

Модель Крипке, (граф элементарных переходов в исходной модели) достаточно просто можно преобразовать в ω-автомат. Языком полученного автомата будут пути в модели Крипке. Также известно, что можно построить ω-автомат, допускающий пересечение языков двух ω-автоматов [4]. Тогда возникает следующая идея: построить два автомата Бюхи: из *отрицания* исходной *LTL*-формулы и из модели Крипке и их перемножить. Тогда любое слово, допускаемое полученным автоматом, будет одновременно путем в модели Крипке, и не будет удовлетворять исходной *LTL*-формуле. Следовательно, это будет искомый контрпример – путь в модели Крипке, нарушающий свойство. Обратно, если контрпример существует, то он будет допускаться обоими автоматами Бюхи, а следовательно, и их пересечением.

Приведенная идея используется во многих верификаторах *LTL*-формул, в том числе в верификаторе *SPIN*. Для ее реализации требуется:

- преобразовать *отрицание LTL*-формулы в автомат Бюхи;

- построить пересечение полученного автомата и модели Крипке (точнее, ее автоматного представления);
- найти допускающий путь в полученном пересечении или убедиться, что его не существует.

Для трансляции *LTL*-формулы в автомат Бюхи был предложен алгоритм [4], и в открытом доступе существуют программы, его реализующие (например, *LTL2ba*, который для верификатора *SPIN* строит автомат Бюхи на языке *Promela*).

Построение пересечения автомата Бюхи и модели Крипке можно делать неявно по правилам, изложенными ниже.

- Состояние пересечения – это комбинация состояния модели Крипке и состояния автомата Бюхи.
- Каждый шаг в пересечении совершается в два действия: сначала происходит переход в модели Крипке, а затем в автомате Бюхи. Таким образом, они работают «параллельно».
- Если автомат Бюхи неполный (что бывает при использовании программы *LTL2ba*), то при отсутствии активных переходов текущее состояние направляется в «сток»: недопускающее состояние с единственным переходом-петлей.
- Если активных переходов несколько, то возможные переходы «перемножаются» — создается ветвление как по активным переходам в автомате Бюхи, так и в модели Крипке. Например, если в текущем состоянии системы активны два перехода в автомате Бюхи и три перехода в модели Крипке, то для «перемноженной» системы будет ветвление из текущего состояния в шесть переходов.
- Если автомат Бюхи оказался в допускающем состоянии, следовательно, и пересечение автоматов оказалось в допускающем состоянии.

Перейдем к описанию алгоритма проверки на пустоту полученного пересечения (напомним, что пересечение – тоже автомат Бюхи).

Пусть p – допускающий путь автомата Бюхи. Так как множество состояний автомата конечно, то найдется суффикс p' пути 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 суффикса p' входят в состав некоторой сильно связной

компоненты графа, описывающего автомат Бюхи. Причем, так как одно из допускающих состояний также входит в этот суффикс то сильно связная компонента содержит допускающее состояние.

Обратно, если в графе автомата Бюхи существует достижимая сильно связная компонента, содержащая допускающее состояние, то можно построить допускающий путь. Он будет иметь структуру $\alpha\beta^*$, где α – префикс, ведущий к сильно связной компоненте, а β – цикл в этой компоненте, содержащий допускающее состояние. Структура допускающего пути изображена на рис. 8.

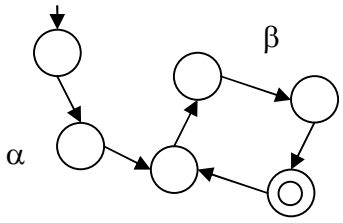


Рис. 8. Структура допускающего пути

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* [4] (*double Depth-First Search*, *DFS*).

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

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

Более формально алгоритм проверки на пустоту языка автомата Бюхи с помощью двойного поиска в глубину может быть записан следующим (на псевдокоде из работы [4]) образом:

```
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 procedure
```

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

Доказательство корректности алгоритма подробно описано в работе [4].

Для борьбы с комбинаторным взрывом в верификаторе *SPIN* используется эффективный алгоритм редукции частичных порядков, особенностью которого по сравнению с другими аналогичными алгоритмами является то, что он не требует дополнительной памяти для работы. Эксперименты показали [33], что на примере задачи выбора лидера среди нескольких процессов, число состояний системы без редукции растет экспоненциально от числа процессов, в то время как применение алгоритма редукции частичных порядков позволяет снизить этот рост до линейного.

Также в верификаторе *SPIN* используются эвристики для компактного хранения пройденных состояний. Это позволяет верифицировать большие системы. Для уменьшения памяти, занимаемой множеством посещенных состояний, можно было бы использовать сжатие с помощью алгоритма Лемпеля – Зива – Велча или алгоритма Хаффмана. Это приводит к сильному увеличению времени работы верификатора (до 400%), в то время как используемая память уменьшается не так сильно – 10−20% [33]. Используемый в верификаторе *SPIN* алгоритм приводит к экономии памяти на 60–80% при небольших затратах времени.

В случае если задача слишком велика и не поддается верификации по причине нехватки памяти, можно проводить неполную верификацию. При этом желательно добиться как можно большего покрытия пространства состояний верификатором, для того, чтобы иметь наиболее близкую к полной верификацию. В верификаторе *SPIN* для этого используется специальный алгоритм, кодирующий каждое посещенное состояние двумя битами, адреса которых генерируются двумя независимыми хэш-функциями. Этот алгоритм очень эффективен и позволяет добиться покрытия близкого к 100% даже для таких задач, максимальное покрытие пространства состояний которых без использования этого алгоритма равнялось 10%.

Перечисленные алгоритмы и улучшения делают верификатор *SPIN* эффективным и позволяют применять его для верификации реальных систем [33].

Так как верификатор *SPIN* создан для верификации протоколов, имеющих конечное число состояний, то можно предположить, что его применение для верификации автоматных моделей будет весьма эффективно. При этом верификатор *SPIN* поддерживает параллелизм, верифицируемых протоколов, и поэтому позволит верифицировать системы взаимодействующих автоматов.

1.2. Требования к методам верификации автоматных моделей управляющих программ

В данном разделе описаны, требования к методам верификации рассматриваемого класса моделей. Сначала рассмотрим методы простейшей верификации автоматов. Далее будут рассмотрены требования для методов верификации отдельных автоматов, и, наконец, будут рассмотрены особенности методов верификации систем из нескольких автоматов.

1.2.1. Требования к методам простейшей проверки автоматов.

Методы верификации должны позволять проверять простейшие свойства автоматных моделей, которые должны выполняться всегда. Объект, у которого эти свойства не выполняются, не принадлежит к рассматриваемому классу моделей.

1. Полнота условий на переходах из состояния.
2. Непротиворечивость условий на переходах из состояния.
3. Наличие одного и только одного начального состояния в автомате.
4. Наличие одного и только одного перехода из начального состояния.
5. Отсутствие переходов в начальное состояние.
6. Отсутствие переходов из конечного состояния.
7. Достижимость всех состояний автомата из начального состояния.
8. Сформулируем эти простейшие свойства.

Проверка полноты условий на переходах из состояния. Под полнотой условий понимается то, что для каждого события существует хотя бы один переход при любом значении переменных. Пример неполного автомата изображен на рис. 3. В этом автомате отсутствует переход для состояния *Opening* по событию *e4* при условии *o2.x1* = `false`.

Проверка непротиворечивости условий на переходах из состояния. Под непротиворечивостью условий понимается то, что для каждого набора значений переменных для любого события имеется только один допустимый переход. Пример противоречивого автомата приведен на рис. 9.

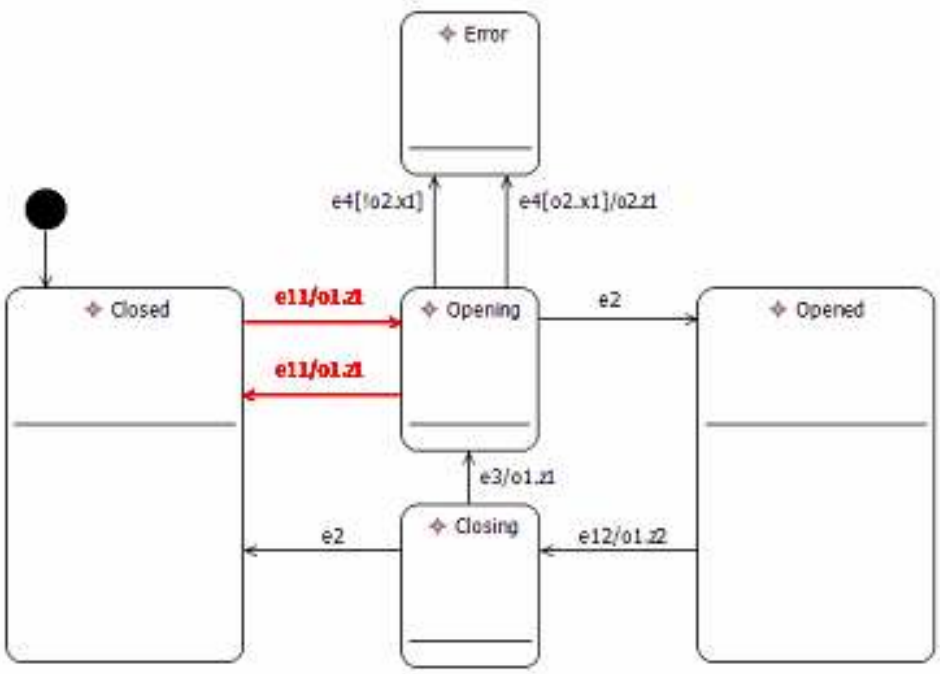


Рис. 9. Пример противоречивого автомата

Наличие одного и только одного начального состояния в автомате. На рис. 10 изображен вариант автомата, нарушающего это правило.

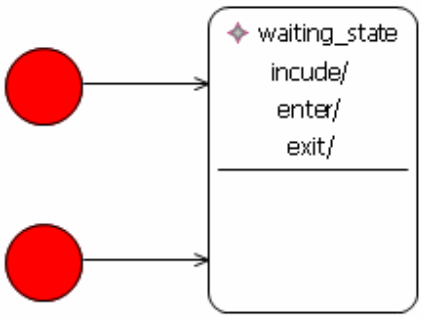


Рис. 10. Пример автомата, в котором нарушается ограничение наличия одного начального состояния в автомате

Наличие одного и только одного перехода из начального состояния. Напомним, что при запуске автомат находится в состоянии, в которое ведет переход из начального состояния. Таким образом, при наличии более чем одного перехода возникает неоднозначность — в каком из состояний начинать работу. Пример нарушения этого ограничения приведен на рис. 11.

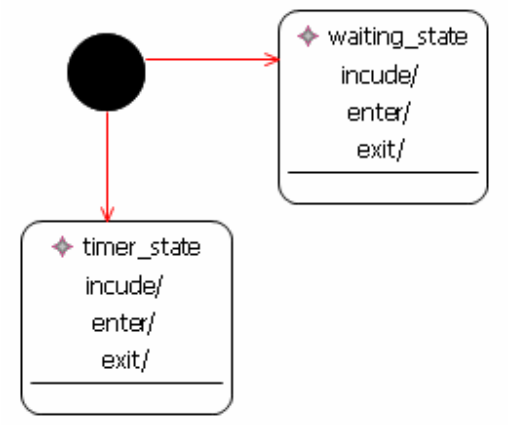


Рис. 11. Пример автомата, в котором нарушается свойство наличия одного перехода из начального состояния

Отсутствие переходов в начальное состояние. Автомат никогда не находится в начальном состоянии (даже в начале работы автомат находится в состоянии, в которое ведет переход из начального состояния). Поэтому перейти в это состояние в ходе работы автомата нельзя. Пример автомата нарушающего это правило приведен на рис. 12.

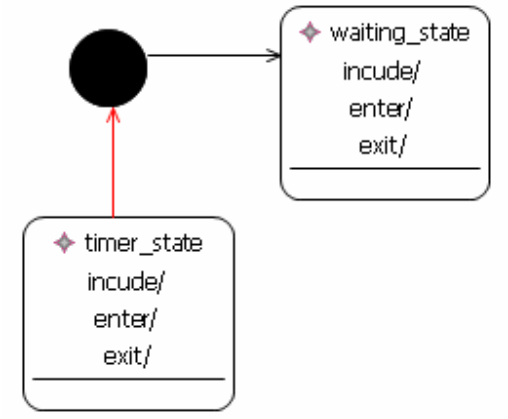


Рис. 12. Пример автомата, в котором нарушается свойство отсутствия переходов в начальное состояние

Отсутствие переходов из конечного состояния. После перехода в конечное состояние автомат завершает свою работу. Поэтому переходы из конечного состояния никогда не будут задействованы.

Достижимость всех состояний автомата из начального состояния. Состояние *S* считается достижимым, если существует путь из начального состояния в состояние *S*. На рис. 13 изображен автомат с недостижимым состоянием *Opened*.

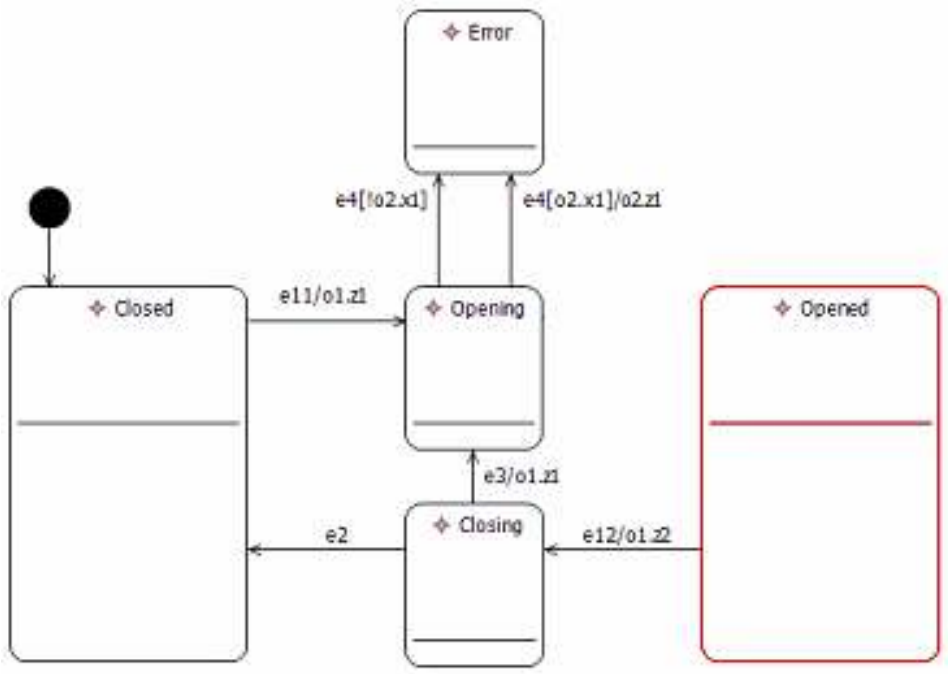


Рис. 13. Пример автомата с недостижимым состоянием

1.2.2. Требования к методам верификации отдельных автоматов

Методы верификации автоматных программ должны позволять проверять темпоральные свойства автоматных моделей.

Так как по автоматной модели модель Крипке можно построить автоматически (что будет показано в разд. 2.1), то главное требование к методам верификации автоматных моделей — автоматизация. Поэтому, требуется разработать алгоритм автоматического построения модели Крипке по автомату. Для проверки построенной модели существует три подхода.

1. Выполнить трансляцию автоматной модели и требований к ней во входной язык одного из известных верификаторов. Если верификатор находит ошибку, то транслировать полученный сценарий ошибки обратно в автоматную модель.
2. Написать свой верификатор, реализовав самостоятельно все необходимые аспекты верификации.

3. Комбинированный подход: использовать готовый верификатор, однако четко определив правила создания модели Крипке по автоматной модели.

На данный момент существует большое число готовых верификаторов (разд. 1.1.2). Подход, использующий готовый верификатор имеет свои ограничения: модель надо описывать на входном языке выбранного верификатора, ограничение на выбор формального языка для описания спецификации к модели и т. д. Однако, среди готовых верификаторов есть хорошо оптимизированные и проверенные годами.

Подход, предполагающий создание собственного верификатора лишен этих ограничений, но является очень трудоемким.

В третьем подходе используется готовый верификатор, но для него определяются правила, по которым он будет строить модель *Крипке* для автоматной модели. Для реализации такого подхода требуется верификатор с достаточной степенью гибкости входного языка. Таким является верификатором *Bogor*.

Для методов верификации автоматных моделей желательно наличие формального доказательства правильности алгоритма построения модели. Оно позволяет исключить ошибки в методе верификации. Однако оно не позволяет исключить ошибки в программной реализации методов.

Методы верификации должны позволять верифицировать достаточно большие системы за разумное время. Они должны позволять верифицировать, автоматы с несколькими сотнями состояний.

1.2.3. Требования к методам верификации систем автоматов

Несмотря на то, что систему автоматов $A_1, \ldots, A_n$ можно представить как один автомат A в виде декартова произведения $A = A_1 \times \ldots \times A_n$, верификация автомата A может быть слишком трудоемкой. Рассмотрим 10 автоматов A_1, A_2, …, A_{10}, в каждом из которых по 10 состояний. Тогда их декартово произведение содержит 10^{10} состояний, что не позволяет произвести верификацию автомата A. Таким образом, методы верификации должны позволять верифицировать системы автоматов.

В хорошей системе не должно быть взаимных блокировок (deadlock). Поэтому методы верификации должны уметь проверять автоматные модели на отсутствие блокировок. В качестве примера рассмотрим задачу об обедающих философах. За столом расставлены стулья, на каждом из которых сидит определенный философ. В центре стола – большое блюдо спагетти, а на столе лежат

вилки – каждая между двумя соседними тарелками. Каждый философ может находиться только в двух состояниях: либо он спит, либо ест спагетти. Начать спать философу ничто не мешает. Однако, 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 опять начинает спать до тех пор, пока снова не проголодается.

Создадим диаграмму классов для нашей модели. Она будет состоять из двух классов: `Philosopher` и `Fork` (рис. 14).

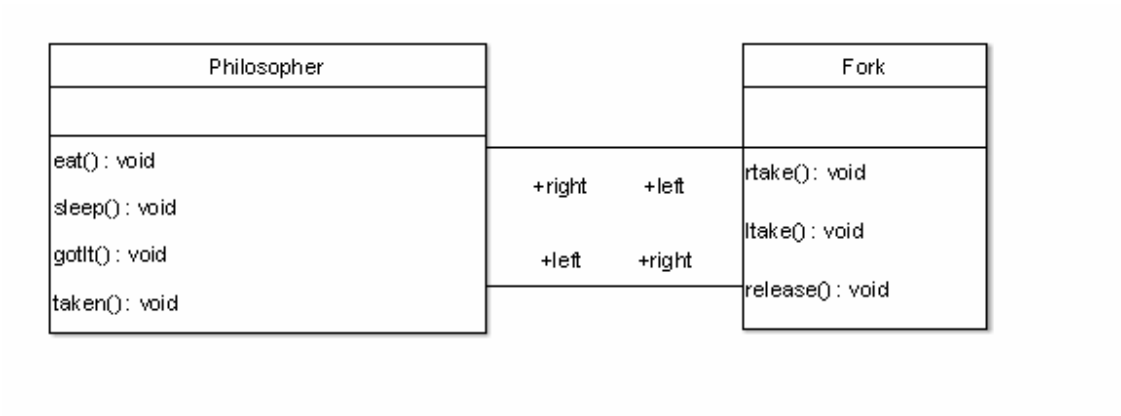


Рис. 14. Диаграмма классов для задачи об обедающих философах

Опишем поведение философа и вилки в виде диаграммы состояний. На рис. 15 представлена диаграмма состояний для вилки. Будем считать, что в модели допущена ошибка, если философ пытается освободить вилку, которая не занята. Можно представить это свойство, добавив переход в “плохое” состояние `Released`.

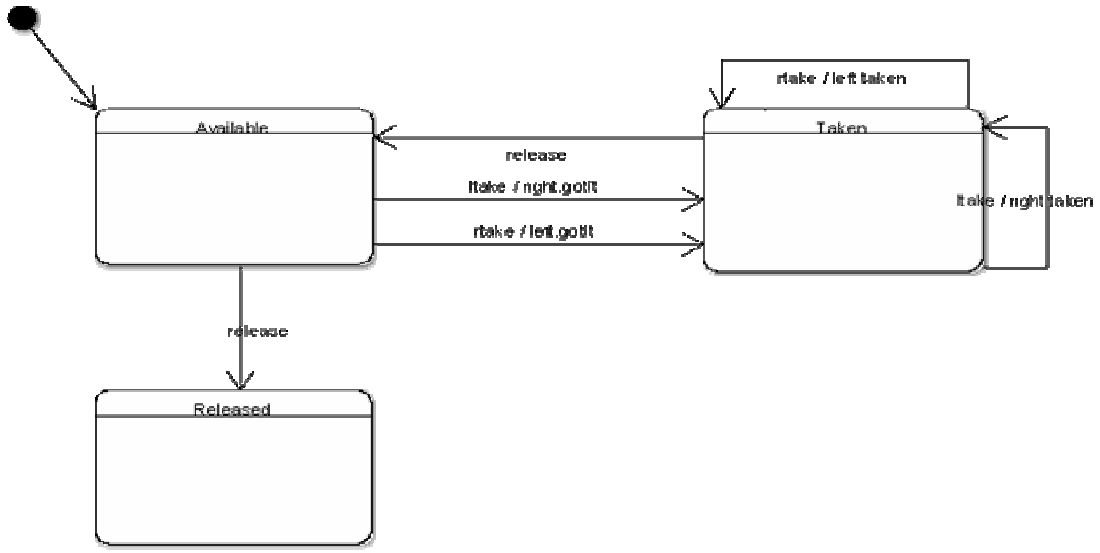


Рис. 15. Диаграмма состояний для вилки

На рис. 16 представлена диаграмма состояний для философа. В данном примере философ пытается сначала взять левую вилку, затем правую и начать есть.

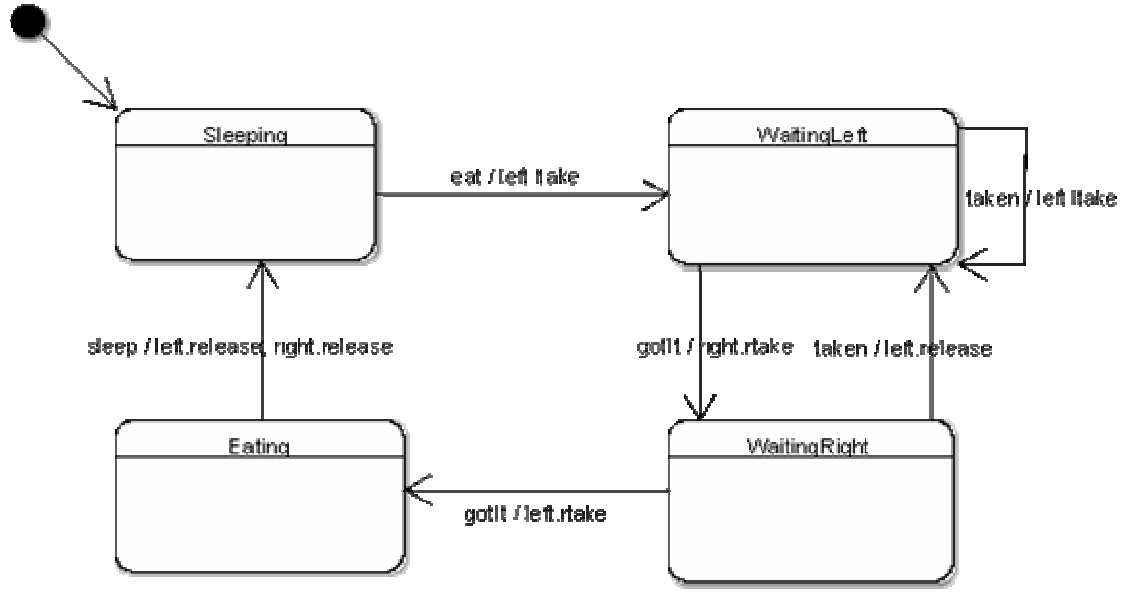


Рис. 16. Диаграмма состояний для философа

Используем для задания начального состояния системы. Для этого зададим объекты введенных классов и установим связи между ними. Объекты на диаграмме взаимодействуют друг с другом путем передачи сообщений от одного объекта другому.

Выберем начальную конфигурацию модели, на которой представлено четыре философа, которые пытаются разделить между собой четыре вилки (рис. 17).

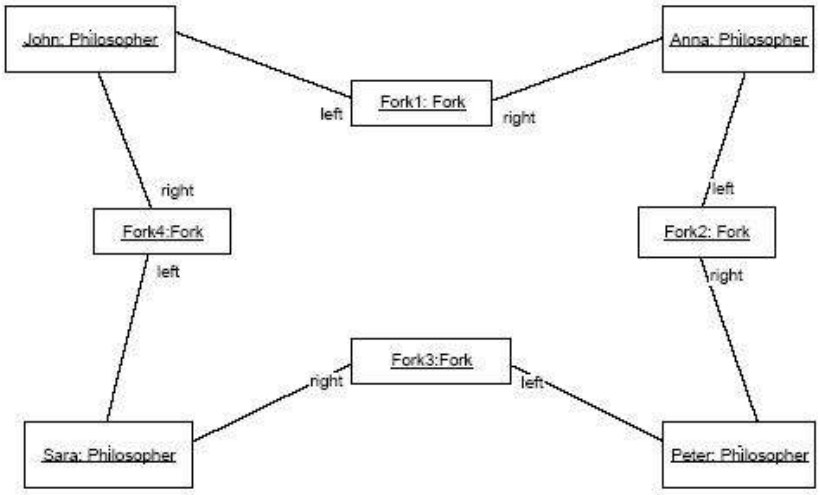


Рис. 17. Обед четырех философов

Диаграмма объектов представляет одну из возможных начальных конфигураций системы. Соответственно, существует большое число возможных поведений данной системы после приведения ее в действие. В нашем примере, если класс философа плохо сконструирован, то верификация может определить ошибку взаимной блокировки, когда ни один объект не может продолжать свое выполнение.

При верификации данного примера будет обнаружена взаимная блокировка. Пример такой ситуации изображен на рис. 18.

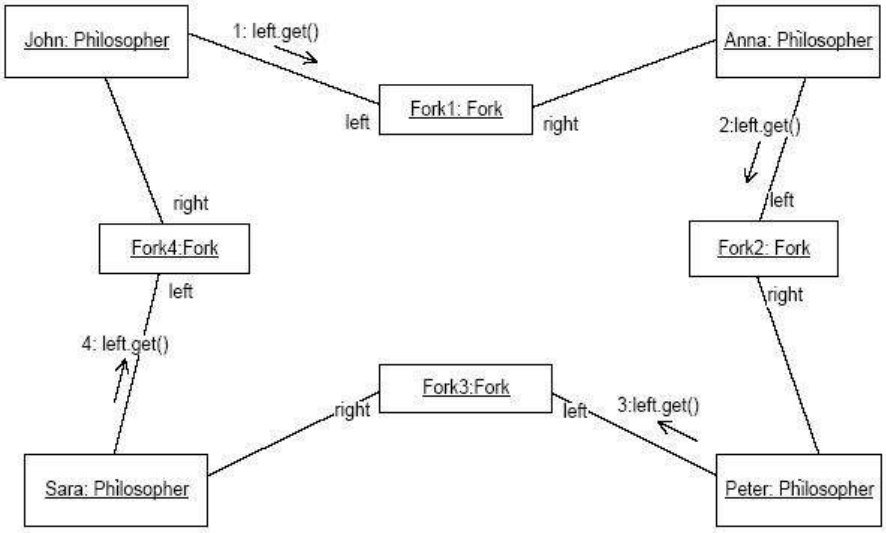


Рис. 18. Взаимная блокировка

Каждый философ взял левую от себя вилку и ждет, когда будет доступна правая. Однако, правую вилку никто из философов получить не может, так как ее уже взял сосед справа.

Сформулируем количественные требования. Методы верификации автоматных моделей должны позволять верифицировать такие системы как, например система управления дизель-генератором [6]. Система управления дизель-генератором состоит из 31 автомата. Обычно автоматы имеют не более шести состояний.

Исходя из этого можно сформулировать количественные требования с запасом для более крупных систем. Таким образом, методы верификации автоматных программ должны позволять верифицировать системы из 10 автоматов, каждый из которых содержит, по 10 состояний.

2. БАЗОВЫЕ МЕТОДЫ ВЕРИФИКАЦИИ НА МОДЕЛЯХ, ПРЕДНАЗНАЧЕННЫЕ ДЛЯ ВЕРИФИКАЦИИ АВТОМАТНЫХ МОДЕЛЕЙ УПРАВЛЯЮЩИХ ПРОГРАММ

2.1. Методы построения модели Крипке по автоматным моделям управляющих программ

Структурой Крипке над множеством элементарных высказываний P называется система переходов $(S, s_0, \to, L)$, где

1. S – конечное множество состояний;
2. $s_0 \in S$ – начальное состояние;
3. $\to \subseteq S \times S$ – тотальное отношение переходов (тотальность означает, что для каждого состояния $s \in S$ должно существовать состояние $s' \in S$, для которого имеет место $(s, s') \in \to$, т. е. $s \to s'$);
4. $L : S \to 2^P$ – функция, помечающая каждое состояние множеством элементарных высказываний, истинных в этом состоянии.

Путь в структуре Крипке из состояния s_0 – это бесконечная последовательность состояний $\pi = s_0 s_1 s_2 \ldots$ такая, что для всех $i \geq 0$ выполняется $s_i \to s_{i+1}$. Для некоторого пути $\pi = s_0 s_1 s_2 s_3 \ldots$ обозначим π^i суффикс π, который получается удалением из π первых i состояний – например, $\pi^1 = s_1 s_2 s_3 \ldots$, а $\pi(i)$ будет обозначать *i*-е состояние пути, $\pi(0) = s_0$, $\pi(1) = s_1$ и т. д.

В работах [22, 48, 39] рассмотрены различные способы преобразования автоматов в структуру Крипке и проверки их свойств, выраженных в формулах различных темпоральных логик. В работе [22] рассмотрены способы верификации систем состоящих из большого числа параллельных процессов. Синхронные системы в статье представлены в виде взаимодействующих автоматов Мура. Автоматы имеют множество входных воздействий, которыми моделируется внешняя среда. В статье приведены алгоритм для преобразования набора взаимодействующих автоматов Мура в один и способ последующего преобразования этого автомата в структуру Крипке. Состоянием в модели Крипке автор предложил считать пару «состояние автомата, входное воздействие». В качестве атомарных предположений для состояний предложено использовать входные и выходные

воздействия исходного автомата. В работе использовалась наиболее выразительная темпоральная логика *CTL**.

В работе [48] также рассмотрена верификация автоматов Мура, но уже с использованием более пригодной для практического применения логики *CTL*. В ней приведены различные способы преобразования автоматов в структуру Крипке: когда входные воздействия отбрасываются, когда входные воздействия относятся к состоянию, в котором они произошли, и когда они относятся к состоянию, в которое переходит автомат. В статье указана возможная неоднозначность формул из *CTL* при верификации автоматов Мура в зависимости от того, какой из способов преобразования используется. В связи с этим для задания спецификации необходимо точно знать, каким образом автомат преобразуется в структуру Крипке. В этой работе предложена новая логика *iCTL,* которая является расширением логики *CTL.* В *iCTL* добавлены два новых квантора: $\forall_I$ – для любого входного воздействия и $\exists_I$ – существует входное воздействие. Это позволяет избежать неоднозначностей при проверке.

В работе [39] предложен алгоритм верификации параллельных автоматов Мили с использованием логики *CTL.* В статье приведён алгоритм проверки формулы, при котором учитывается поведение не всех автоматов в системе. Алгоритм строит два множества состояний для формулы φ: $U[\varphi]$ и $L[\varphi]$. Причем если обозначить за $[\varphi]$ все состояния, в которых выполняется φ, то будет следующее вложение $L[\varphi] \subset [\varphi] \subset U[\varphi]$. Алгоритм по ограниченной модели производит приблизительную проверку выполнимости формулы. Постепенно улучшая эту оценку, можно точно узнать, выполняется ли данная формула в требуемом состоянии. При этом использование модели, в которой присутствуют не все автоматы, иногда позволяет сократить число состояний модели.

2.1.1. Анализ методов построения модели Крипке

В данном разделе будут описаны различные способы построения модели Крипке для автоматных моделей, и перечислены их достоинства и недостатки. Модель Крипке по своей структуре очень похожа на автомат. Более того, в некоторых верификаторах граф модели Крипке специально преобразуется в автомат. Таким образом, в случае использования таких алгоритмов, можно не строить модель Крипке явно, а вместо нее напрямую использовать автоматную модель. В других алгоритмах такой подход не работает.

Цель построения модели Крипке – разбиение верифицируемой программы на атомарные состояния, которые порождают атомарные действия – переходы между атомарными состояниями. Вообще говоря, не существует абсолютной меры атомарности разбиения, и в зависимости от решаемых задач и атомарность требуется разная. Важно, чтобы любой предикат, присутствующий в верифицируемой спецификации, всегда принимал единственное значение в любом атомарном состоянии. Это утверждение следует из определения модели Крипке. Если состояния недостаточно атомарны, то многие алгоритмы верификации перестают работать.

В рассматриваемых моделях автоматных программ система автоматов работает последовательно, а не параллельно, что значительно упрощает построение модели Крипке, и ее размеры получаются относительно малыми. В целом процесс работы автоматной программы линеен, и ветвление историй происходит только в результате разных последовательностей событий, происходящих в системе, а также в результате различных значений входных воздействий в объектах управления.

Рассмотрим для начала, как можно было бы строить структуру Крипке (модель Крипке) для автоматных моделей, состоящих из одного автомата. Приведем три схемы, по которым это можно сделать.

Переходы между состояниями автомата содержат большое количество внутренних действий: это получение произошедшего события, вычисление входных значений, выполнение выходных воздействий на объектах управления. Возникает необходимость разбить каждый переход на несколько атомарных действий. Это можно делать описанным ниже способом [3].

Для каждого перехода между состояниями `S` и `T` исходного автомата создадим не менее одного состояния в модели Крипке (назовем его состоянием-событием), атомарным предложением которого будет событие `E`, инициировавшее переход. При наличии выходных воздействий на переходе также создадим по одному состоянию на каждое воздействие `Z`, атомарным предложением которого (состояния) будет `Z` (такие состояния будем называть состояниями-выходными воздействиями). Добавим в модель переходы: между состоянием `S` и состоянием-событием `E`, а также между состоянием-событием `E` и первым состоянием-выходным воздействием. Далее последовательно (в порядке выполнения) между соседними состояниями для выходных воздействий, и, наконец, между последним таким состоянием-выходным воздействием и состоянием `T`.

Если выходное воздействие `Z` размещалось в состоянии `T` и выполнялось при входе в него, то при преобразовании добавляется еще одно состояние, соответствующее воздействию `Z`. В это состояние должен вести каждый переход, который первоначально вел в состояние `T`. Кроме этого, добавляется переход из состояния, соответствующего `Z`, в состояние `T`. Само же это воздействие после генерации состояний уничтожается.

Для всех полученных состояний модели Крипке естественным образом устанавливаются атомарные предложения. Добавим также три «управляющих» атомарных предложения: `InState`, `InEvent`, `InAction` – для состояний модели, построенных, соответственно, из состояний, событий и выходных воздействий исходного автомата. Это сделано для того, чтобы при записи формулы в темпоральной логике можно было различать тип исследуемого состояния.

Таким образом, множество атомарных предложений во всех трех схемах содержит объединение множеств состояний, событий, выходных воздействий и трех указанных атомарных предложений.

Первая схема называется «Состояния на событиях и выходных воздействиях» (*CCBB*). Как и другие две, она наследует общую идеологию разбития переходов на атомарные состояния, описанную выше. От других схем ее отличает то, что кроме указанного общего принципа в ней больше ничего не содержится. Таким образом, применяя схему *CCBB* для автоматной программы, можно полностью абстрагироваться от входных переменных, оставляя только состояния (без них не обойдется ни одна базовая модель), события и выходные воздействия. Это самый простой подход.

Рассмотрим пример. Пусть исходное автоматное приложение эмулирует (в довольно упрощенной форме) универсальный инфракрасный пульт для бытовой техники [3]. Эмулятор представлен с помощью одного автомата *ARemote*, схема связей и граф переходов которого изображены на рис. 19 и рис. 20 соответственно.

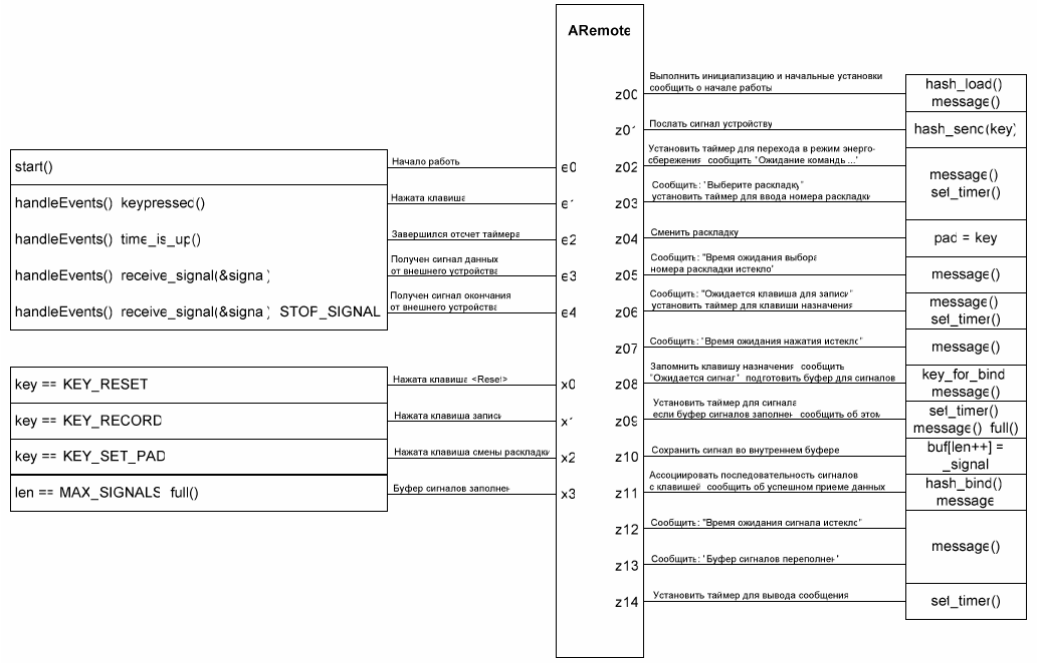


Рис. 19. Схема связей автомата *ARemote*

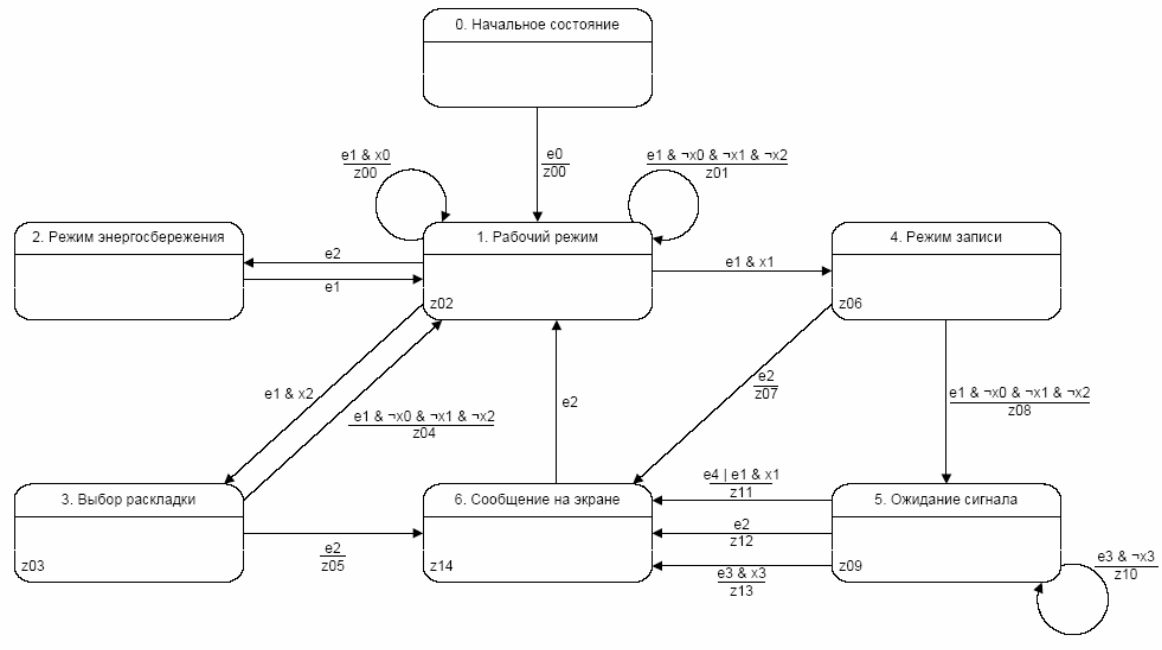


Рис. 20. Граф переходов автомата *ARemote*

В рассматриваемом примере модель Крипке для автомата, построенная по схеме *CCBB*, приведена на рис. 21.

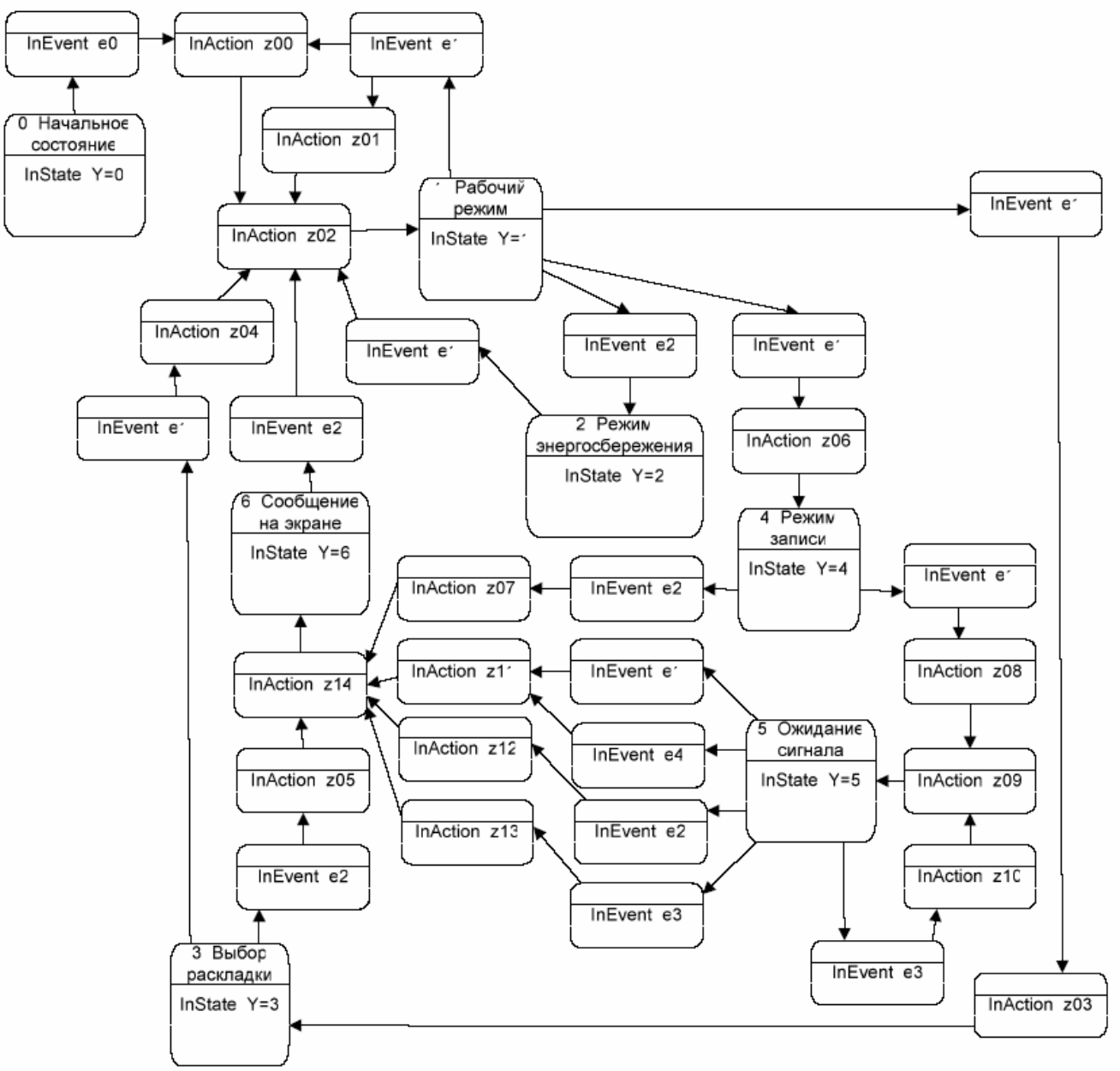


Рис. 21. Модель Крипке, построенная по схеме *CCBB*

В модели Крипке (рис. 21), состояния-события и состояния-выходные воздействия указаны явно. При интерактивном моделировании совместно с исполнением и визуализацией их целесообразно обозначать, как и в исходном автомате, метками на дугах.

Вторая схема называется «Полный автомат». В ней не абстрагируются от входных переменных, а автомат представляется моделью Крипке «во всей полноте» относительно входных воздействий. В исходном автомате переходы могут быть заданы не полностью – могут существовать не указанные петли. Это означает, что для некоторого состояния (некоторых состояний) дизъюнкция формул, составленных из входных переменных, которые помечают переходы из него по одному и тому же событию `E`, не является тавтологией.

В этом случае снабдим указанные состояния петлевыми переходами по событию `E`, соответствующими дополнению к рассматриваемой дизъюнкции. Это, конечно же, не изменит семантику автомата, а лишь полностью опишет его поведение. В конечном счете, в автомате из каждого состояния по каждому событию должно исходить 2^n переходов, где n – общее число входных переменных автомата. При этом каждому переходу соответствует набор значений всех переменных (или множество истинных переменных). После получения полного автомата преобразуем его в модель Крипке по общей схеме с одной модификацией: для каждого состояния-события добавим во множество его атомарных предложений набор входных переменных, истинных на том переходе, на котором находится рассматриваемое состояние-событие. Таким образом, во множество атомарных предложений по отношению к обобщенной схеме добавились еще и входные переменные. Достоинство такой схемы (несмотря на ее громоздкость, сокращение которой будет описано ниже) в том, что она и только она позволяет модели Крипке полностью отражать поведение исходного автомата.

Третья схема называется «Редуцированная схема». Основным недостатком предыдущей схемы было большое число генерируемых состояний для модели Крипке, а достоинством – полнота.

В данном методе семантика моделей будет изменена таким образом, чтобы число состояний в них можно было уменьшить, не потеряв при этом выразительную силу моделей. Это можно сделать так, что размер модели изменяется асимптотически линейно по отношению к размеру графа переходов исходного автомата и к числу переменных (билинейно), в отличие от предыдущей схемы, где размер модели увеличивался экспоненциально от числа входных переменных.

Множество атомарных предложений по отношению к предыдущей схеме также изменим.

Рассмотрим исходный автомат без дизъюнкций на переходах. Если такие переходы существуют, создадим эквивалентные переходы для каждого дизъюнкта, а сами переходы с

дизъюнкциями удалим. В качестве примера можно разбить переход «`(e4|e1&x1)/z11`» графа *ARemote*, изображенного на рис. 20, на два перехода: «`e4/z11`» и «`e1&x1/z11`».

Добавим в автомат состояния, соответствующие событиям, входным и выходным переменным так, как это было сделано в первой схеме, но с одним отличием: во множества атомарных предложений на состояниях-событиях добавим входные переменные в том виде, в котором они присутствуют на переходах (вместе с отрицаниями, если они существуют). Таким образом, в состав множества всех атомарных формул модели входят следующие элементы, и только они: состояния; события; выходные воздействия; все литералы, составленные из входных переменных (переменные и их отрицания). Кроме того, в атомарные предложения каждого полученного состояния-события добавим все литералы, составленные из несущественных входных переменных для данного перехода (несущественными будем называть те переменные исходного автомата, которые не обозначены на рассматриваемом переходе). Таким образом, будем добавлять на одно и то же состояние-событие и несущественные переменные, и их отрицания. С точки зрения синтаксиса и семантики темпоральной логики это допустимо: процесс обработки модели Крипке не предполагает совместность множества атомарных предложений состояния, так как интерпретирует эти предложения просто как строки. Причина такого обращения с несущественными переменными связана с тем, что требуется обеспечить, чтобы любая ссылка на несущественную в данном состоянии-событии переменную, упомянутая в *CTL*-формуле, давала истинный результат.

Не обязательно хранить все литералы, составленные из несущественных переменных в состоянии в явном виде. Важно лишь то, что во время обработки модели существенные и несущественные входные переменные интерпретируются раздельно: первые – в том виде, в каком они записаны на переходах исходного автомата, а вторые – «в двух экземплярах» (в прямом и инверсном виде).

Результат преобразования графа переходов автомата *ARemote*, выполненного с применением данной схемы, изображен на рис. 22.

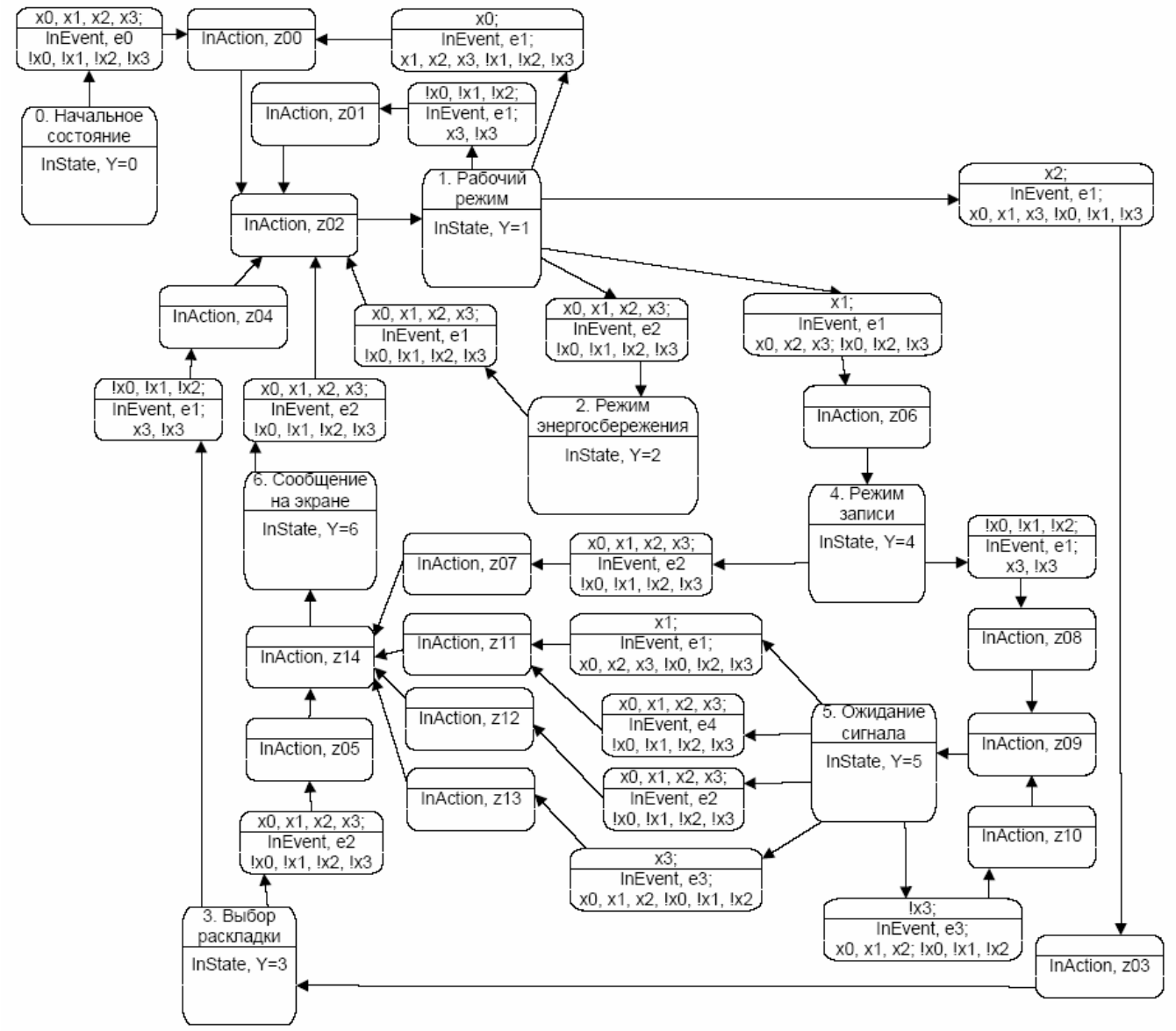


Рис. 22. Редуцированная модель Крипке для автомата *ARemote*

Размер модели на рис. 22 совпадает с размером модели, созданной по схеме «Состояния на событиях и выходных воздействиях». Вообще говоря, первая схема может рассматриваться по аналогии со второй схемой, в которой полностью исключены входные воздействия. Аналогично, третью схему можно рассматривать как видоизменение второй схемы, в котором отождествляются наборы значений несущественных переменных.

2.1.2. Общие рекомендации по построению модели Крипке для автоматных программ

В этом разделе будут рассмотрены рекомендации и дополнительные приемы, применимые на практике при построении модели Крипке по автоматной программе. Они позволят сократить размеры получаемой модели Крипке, а также приблизить модель Крипке к исходной автоматной программе, тем самым, упрощая понимание ее структуры.

Основным недостатком всех схем моделирования автоматов, описанных в разд. 2.1.1, является то, что при составлении требований к модели разработчику не всегда удобно различать, где состояния, которые перенесены из исходного графа, где состояния-события, а где состояния-выходные воздействия. Для различения состояний используются атомарные предложения `InState`, `InEvent` и `InAction`, но их применение может быть связано с дополнительными проверками. Для этого, а также для уменьшения числа состояний модели в принципе, можно при построении модели абстрагироваться от некоторых характеристик.

Приведем несколько приемов, которые могут оказаться полезными при моделировании автоматов.

1. Можно абстрагироваться не только от входных переменных, но и от событий, а также от выходных воздействий. Можно вообще преобразовать автомат в модель Крипке в один этап: например, с помощью исключения событий и выходных переменных на переходах. В этом случае для автомата *ARemote* будет построена модель на рис. 23.

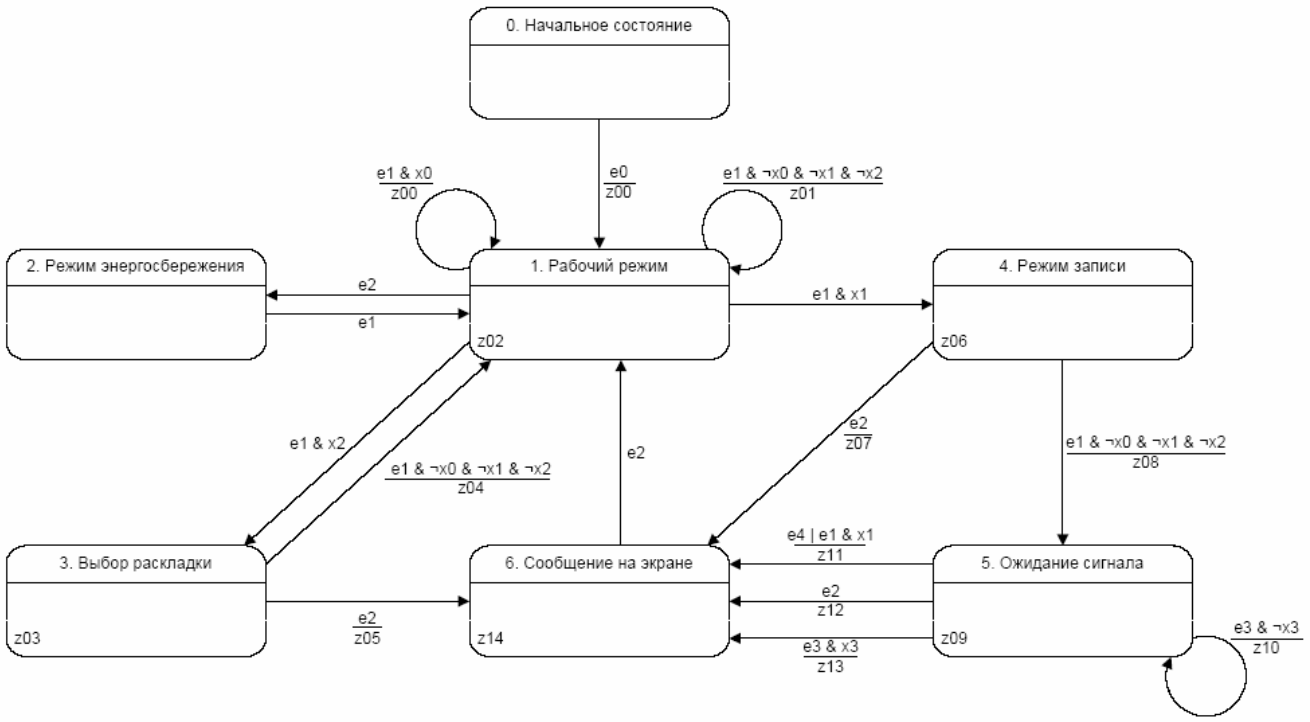


Рис. 23. Сокращенная модель *ARemote* (без событий и выходных переменных)

2. Для выделения существенных свойств автомата можно удалять части его графа переходов, оставляя только подграф, следя при этом за тем, чтобы не нарушалась качественная структура модели, обеспечивающая выполнение ее проверяемых свойств.
3. Заметим, что можно не только удалять подграфы, но и производить гомеоморфную замену одних участков графа другими, более простыми, опять же, с сохранением качественной структуры графа.

Выбор одного из этих приемов можно выполнять, руководствуясь представлениями о производительности и результативности. Основное внимание необходимо уделять атомарности переходов. Если они слишком большие (по числу действий), то разработчик может пропустить ошибку, если же слишком маленькие – то размер модели может немотивированно увеличиться за счет появления несущественных свойств.

Приведенные схемы построения структуры Крипке предназначены для автоматных моделей, состоящих из одного автомата. Тем не менее, их можно применять и для систем автоматов, воспользовавшись одним из перечисленных ниже методов.

1. Для внешних и внутренних автоматов можно выполнять моделирование, спецификацию и верификацию независимо (конечно, этот способ влечет утрату определенных характеристик автомата при моделировании).
2. Также можно «раскрыть» состояние `S` автомата `A`, внутри которого (состояния) находится другой автомат `B`, добавив для каждого перехода из состояния `S` в состояние `T` по одному эквивалентному переходу из каждого состояния автомата `B` в состояние `T`. Все переходы, которые ведут в состояние `S`, следует перенаправить в стартовое состояние автомата `B`. В результате, внутренний автомат `B` превращается в часть автомата `A`, и для него можно выполнять верификацию вместе с автоматом `A`.
3. Систему взаимодействующих автоматов можно привести к одному с помощью композиции (произведения). Также можно выполнять сначала моделирование каждого автомата, а после него – композицию моделей Крипке (рис. 24).

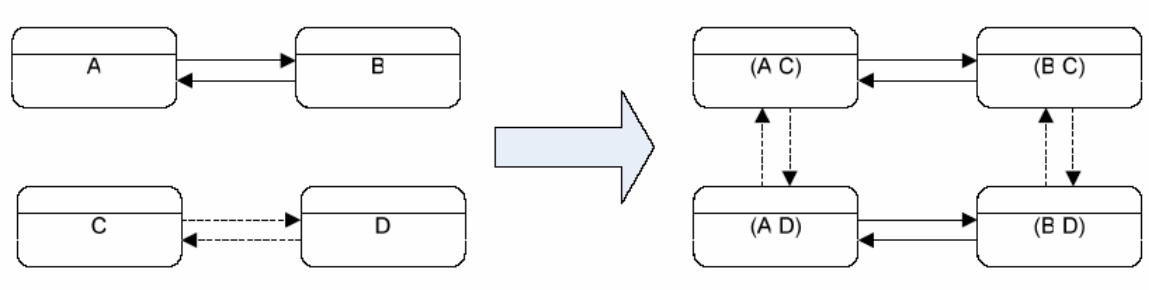


Рис. 24. Композиция структур Крипке

К достоинствам описанных выше методов можно отнести следующее.

- Методы позволяют по автоматной модели построить готовую к верификации модель Крипке, причем делать это автоматически. Можно написать программу, которая будет реализовывать описанные действия по разбиению каждого перехода на множество атомарных переходов.
- Методы позволяют достаточно просто по пути ошибки в модели Крипке построить сценарий ошибки в исходном автомате, поскольку состояния автомата изоморфно преобразуются в состояния `InState` модели Крипке. Проследив путь в модели Крипке, легко восстановить состояния и переходы в исходном автомате.

Недостатком этих методов является то, что они предназначены для автоматных моделей, состоящих из одного автомата. Для систем автоматов эти методы применимы только после дополнительных преобразований.

Рассмотрим теперь другой метод создания структуры Крипке, который лишен недостатка предыдущих методов и работает для системы иерархически связанных автоматов [2].

Рассмотрим для произвольной иерархической программы, построенной на принципах автоматного программирования, ее автоматную модель A, представляющую собой систему взаимодействующих автоматов $(A_0, A_1, \ldots, A_n)$. На систему накладывается следующее ограничение: автомат, находящийся выше по иерархии управляет своими вложенными автоматами путем генерации событий и передачи им управления с этими событиями.

Для этой автоматной модели A построим структуру Крипке, которая, с одной стороны, описывает все возможные состояния системы A, а с другой – задает семантику элементарных высказываний, истинных в этих состояниях.

Рассмотрим произвольный автомат A_k из системы автоматов A. Выделим в автомате A_k, помимо основных состояний, множество его промежуточных состояний, в которых автомат пребывает во время перехода из одного основного состояния в другое, по аналогии с тем, как это было сделано в предыдущих методах. Промежуточное состояние перехода автомата будем фиксировать каждый раз, когда автомат совершит одно из элементарных действий – автомат отреагирует на некоторое событие e_k, обратится к объекту управления с запросом значений входных переменных x_k ($!x_k$) или произведет некоторое выходное воздействие z_k на объект управления или вложенный автомат.

Продемонстрируем идею выделения промежуточных состояний для перехода некоторого автомата A_k (рис. 25).

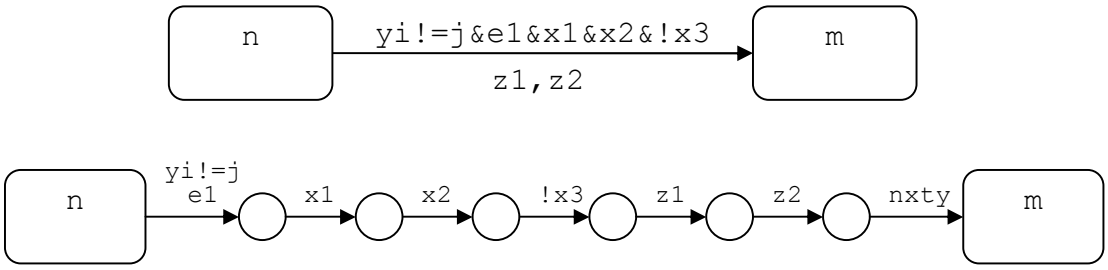


Рис. 25. Выделение промежуточных состояний для автоматного перехода

На этом рисунке промежуточный переход с пометкой `nxty` введен для того, чтобы иметь возможность непосредственно отслеживать момент перехода вложенного автомата в следующее основное состояние с одновременной передачей управления главному автомату (в случае промежуточного перехода `nxty` основного автомата, активным остается по-прежнему основной автомат). Внутренний переход с пометкой e_1 может произойти при наступлении события e_1, если выполняется условие $y_i != j$. Все остальные переходы не имеют условий и являются активными при попадании автоматом в соответствующие промежуточные состояния. В указанной на рисунке последовательности переходов по внутренним состояниям обязательно сначала расположен переход, соответствующий входному событию, далее переходы с пометками входных запросов, а затем переходы, помеченные выходными воздействиями. Таким образом, можно разделить последовательность внутренних состояний на три части, которые расположены друг за другом строго в соответствии с указанным порядком. В каждой части метки внутренних переходов помечены в порядке следования соответствующих элементов в выражении на дуге основного (исходного) перехода.

Далее для произвольного автомата под переходом в следующее состояние будем понимать внутренний переход в основное или промежуточное состояние.

Каждому автомату A_k из набора `A` сопоставим переменные y_k, xm_k, zd_k, ev_k и st_k ($0 \le k \le n$), а для всей системы автоматов `A` в целом введем переменные `ev`, `xm`, `zd`, `act`, `auto` и `evnt`.

Введенные переменные имеют следующий смысл.

1. Переменная y_k хранит последнее основное состояние автомата A_k.

2. Переменная xm_k содержит последний выполненный запрос параметров объекта управления. Значениями xm_k являются имена входных воздействий в прямой или инверсной форме (xm_k может содержать либо x, либо $!x$, где $x \in X_{Ak}$).
3. Переменная zd_k используется для хранения имени последнего выполненного выходного воздействия $z \in Z_{Ak}$.
4. Переменная ev_k содержит имя последнего обработанного автоматом A_k события $e \in E_{Ak}$, (имя такого события, на которое автомат A_k отреагировал).
5. Переменная st_k хранит текущее состояние автомата A_k. Значениями st_k являются как основные, так и промежуточные состояния автомата.
6. Переменная auto содержит номер активного в данный момент времени автомата из системы A – номер последнего автомата, получившего управление.
7. Переменная evnt хранит имя поступившего на обработку события для вложенного автомата, которому было передано управление. Значениями переменной являются имена событий всех вложенных автоматов из системы A и пустое значение 0. Переменная evnt принимает значение 0 тогда, когда событие было обработано или же проигнорировано автоматом (если в текущем состоянии автомат не может отреагировать на данное входное событие). В переменную evnt записывается имя некоторого события, если произошла передача управления с этим событием от главного автомата к вложенному.
8. Переменная act используется для хранения имени последнего элементарного действия, произошедшего в системе автоматов A. Значениями переменной act могут быть имена входных событий $e \in E_A$, входных запросов x и $!x$, где $x \in X_A$, выходных воздействий $z \in Z_A$, а также имена специальных действий 0, nxty и end. После перехода активного автомата в новое промежуточное или основное состояние переменная act содержит метку этого промежуточного перехода. Переменная act принимает значение 0, когда вложенный автомат, находясь в основном состоянии, не может обработать входное событие и игнорирует его. При этом автомат совершает пустое действие 0 и переходит в то же самое основное состояние, в котором он находился (автомат остается в прежнем состоянии). Если act = nxty, то это

означает, что некоторый вложенный автомат перешел в свое следующее основное состояние с одновременной передачей управления главному автомату. Значение `end` используется для идентификации тупикового состояния всей системы автоматов. Для того чтобы выполнить условие тотальности отношения переходов для структуры Крипке устанавливается, что из тупикового состояния (структуры Крипке) есть всего лишь один переход сам в себя и при этом совершается действие `act = end`.

9. Переменные `ev`, `xm` и `zd` используются для хранения имен последних выполненных входных событий, входных запросов и выходных воздействий соответственно в рамках всей системы `A` в целом.

Тогда состоянием S структуры Крипке S_A автоматной модели `A` является вектор значений переменных

$$(act, auto, evnt, ev, xm, zd, ev_0, xm_0, zd_0, y_0, st_0, \ldots, ev_n, xm_n, zd_n, y_n, st_n)$$

В этом случае состояние S можно рассматривать как отображение, которое задается над множеством переменных и используется для определения значения этих переменных в состоянии S. Например, запись S (`act`) представляет собой значение переменной `act` в состоянии S. В начальном состоянии S_0 структуры Крипке S_A все переменные y_i и st_i содержат начальные состояния соответствующих автоматов A_i, переменная `auto = 0` (активным является основной автомат A_0), переменная `evnt` не содержит событий и установлена в `0`, а всем остальным переменным присваивается начальное значение `intl`, которое вводится специально для инициализации и далее не используется.

Отношение переходов структуры Крипке S_A автоматной модели `A` определяется в соответствии с поведением системы автоматов `A`.

1. Переход системы `A` соответствует только одному из внутренних переходов некоторого автомата из `A`.
2. При выполнении условий перехода некоторый автомат A_k может совершить этот переход, если в данном состоянии он является активным (ему передано управление), что отражает значение переменной `auto = k`.
3. Для активного вложенного автомата A_k переход с пометкой `e`, (`e` – некоторое входное событие), может произойти, если переменная `evnt = e` и выполняется условие для этого перехода – истинен предикат над значениями переменных состояний y_i,

сопоставленный переходу. После срабатывания этого перехода изменяются значения переменных act, evnt, ev, ev_k и st_k. Переменная evnt принимает нулевое значение 0, означающее, что событие было обработано, а переменным act, ev и ev_k присваивается имя события e. В переменную st_k помещается внутреннее состояние, в которое произошел переход. Если активным автоматом является основной автомат A_0, то переход с пометкой e происходит при выполнении условия перехода, а evnt = 0.

4. Если вложенный автомат A_k получил от главного автомата управление с событием e, которое находится в переменной evnt, но не может на него отреагировать – не существует из данного основного состояния перехода с пометкой e или для него не выполнены условия перехода, то происходит пустое действие act:= 0, переменная evnt обнуляется и управление передается главному автомату – в переменную auto заносится номер главного автомата, а все остальные переменные не изменяют своих значений.
5. Если произошел переход с пометкой x, где x – некоторый входной запрос в прямой или инверсной форме (x может иметь вид !x′), то происходят изменения следующих переменных: act:= x, xm:= x, xm_k:= x, а переменная st_k получает значение внутреннего состояния, в которое был сделан переход.
6. При срабатывании перехода автомата A_k с пометкой z, где z – выходное воздействие, происходят присваивания act:= z, zd:= z, zd_k:= z, и переменная st_k получает значение внутреннего состояния, в которое был сделан переход. Более того, если z принадлежит к выходным воздействиям второго типа, (z = $A_{k'}$(e), где $A_{k'}$ – вложенный автомат, а e – передаваемое вместе с управлением входное событие для автомата $A_{k'}$), то переменная evnt получает значение e, и активным становится вложенный автомат $A_{k'}$ за счет присваивания auto:= k′.
7. Если произошел переход с пометкой nxty в основное состояние j во вложенном автомате $A_{k'}$, то происходят присваивания act:= nxty, $y_{k'}$:= j, $st_{k'}$:= j и передача управления главному автомату A_k через auto:= k. Если переход nxty произошел в основном автомате A_0, то переменная act = 0 остается без изменения.

8. Если из некоторого состояния автомата A_k существует переход, который помечен переменными z, x, !x или nxty, то для его срабатывания не требуются дополнительных условия, кроме активности этого автомата auto = k.
9. Если система взаимодействующих автоматов A попала в тупиковое состояние, что равносильно невозможности совершить переход из основного состояния основного автомата A_0 из-за нарушения условий переходов, то происходит специально введенный переход act:= end, который направлен в то же самое состояние, а значения всех остальных переменных остаются без изменений.

Благодаря построенной структуре Крипке при спецификации и верификации, имеется возможность в качестве элементарных высказываний использовать предикаты над значениями введенных переменных. Это позволяет выражать следующие свойства состояний автоматных моделей.

1. В каждом состоянии автоматной модели существует возможность отслеживать, какое последнее действие произошло перед попаданием в текущее состояние с помощью переменной act.
2. Имеется возможность определения типа последнего действия (произошло ли входное событие, входной запрос или выходное воздействие) с помощью выражений act = ev, act = xm и act = zd. Например, если в текущем состоянии автоматной модели выполняется act = zd, то это означает, что последним действием, которое произошло при переходе в данное состояние, является некоторое выходное воздействие. Уточнить, к какому автомату системы A принадлежит выходное воздействие, можно с помощью выражений вида act = zd_i. Наконец, истинность выражения act = z означает, что последним действием при переходе в текущее состояние было выходное воздействие z.
3. Переменная auto в каждом состоянии автоматной модели определяется активный в данный момент времени автомат. Например, если в состоянии системы A выполняется auto = 0, то это означает, что активным является основной автомат A_0.
4. Для каждого автомата A_i в текущем состоянии системы A с помощью переменной y_i последнее основное состояние, в которое перешел автомат A_i. Более того, выражение

$y_i = st_i$ означает, что автомат A_i в данном состоянии системы A находится в своем основном состоянии.

5. С помощью выражения act = end можно отслеживать тупиковые состояния системы автоматов A, а с помощью выражения act = nxty —переход одного из автоматов системы в свое новое основное состояние с одновременной передачей управления главному автомату.

Рассмотренный метод построения структуры Крипке из автоматной модели является, одним из наиболее полных методов и обладает следующими достоинствами.

- Позволяет работать с системами иерархически связанных автоматов.
- Структуры Крипке на основе этого метода можно строить автоматически, по описанным выше правилам. При этом все действия происходят с простыми структурами данных — набором идентификаторов (перечислений). Поэтому такую структуру Крипке можно запрограммировать на входном языке какого-либо верификатора, а после этого автоматически верифицировать спецификацию.
- Кроме того, можно с помощью пути нарушения спецификации в модели Крипке построить сценарий ошибки в исходной системе автоматов, так как состояния автомата изоморфно отображаются в главные состояния построенной модели Крипке, а переходы – в цепочки ее промежуточных состояний.

К недостаткам описанного метода можно отнести то, что при его использовании требуется преобразование спецификации в термины модели Крипке, и автоматизация этого процесса стоит под вопросом.

Опишем еще один метод построения модели Крипке для автоматных моделей. Он основывается на следующих наблюдениях. В предыдущих методах модель Крипке строилась из автомата путем разбития каждого перехода в автомате на атомарные действия. За счет этого создавались цепочки переходов из одних состояний автомата в другие через цепь вспомогательных состояний.

Однако для каждого перехода автомата можно создать новую цепь, и, следовательно, у каждого промежуточного состояния будет лишь один входящий и один исходящий переход. В ходе обхода модели Крипке нельзя попасть в промежуточное состояние, не пройдя через глобальное

состояние, из которого начинается цепь. Тогда промежуточное состояние посещено тогда и только тогда, когда посещено глобальное состояние, в которое ведет цепь промежуточных состояний.

Промежуточные состояния предназначены для того, чтобы точно знать момент, когда некий предикат стал истинным. Однако в большинстве случаев не требуется точно знать, в какой момент во время перехода было выполнено то или иное выходное воздействие или запрошено входное воздействие. В таком случае можно не выделять промежуточные состояния, а все предикаты поместить в конечное состояние перехода. Эта идея проиллюстрирована на рис. 26.

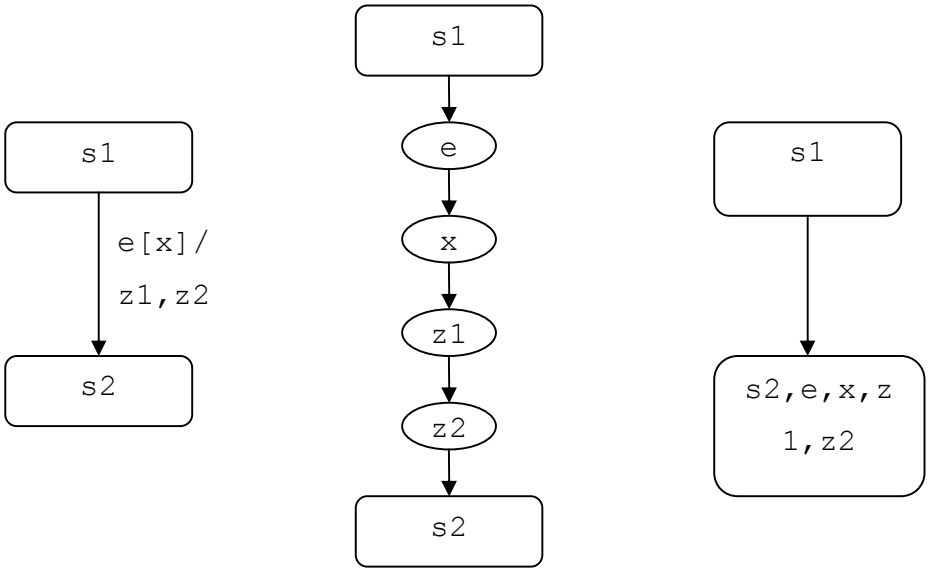


Рис. 26. Подходы к разбиению перехода для структуры Крипке:
исходных переход (а), с выделением промежуточных состояний (б) и
без выделения промежуточных состояний (в)

На рис. 26 в состояниях записаны предикаты, которые в них выполняются. На рис. 26 «б» выделяются отдельные состояния для каждого предиката, например, в первом промежуточном состоянии выполняется предикат `e`, который расшифровывается как «произошло событие `e`». На рис. 26 «в» в конечном состоянии выполняется сразу несколько предикатов. Они расшифровываются следующим образом:

- `s2` – автомат попал в состояние `s2`;
- `e` – произошло событие `e`;

- `x` – входное воздействие `x` было вычислено как `true`;
- `z1` – было выполнено выходное воздействие `z1`;
- `z2` – было выполнено выходное воздействие `z2`.

Однако приведенный способ пометки конечного состояния каждого перехода предикатами, которые в нем выполняются, не позволяет напрямую построить модель Крипке, поскольку в одном и том же состоянии могут, как выполняться, так и не выполняться некоторые предикаты, в зависимости от того, какой переход привел в это состояние.

Тем не менее, данный метод применим, когда верифицируется *LTL*-формула с преобразованием модели Крипке в автомат. Вместо того чтобы строить модель Крипке, а затем преобразовывать ее в автомат, можно было бы сразу использовать исходный автомат. При преобразовании модели Крипке в автомат получается автомат Мура. Это означает, что выполняющиеся предикаты при переходах полученного автомата зависят только конечного состояния перехода. Если же применить описанную схему пометки состояний в исходном автомате, то будет получен автомат Мили, для которого предикаты на переходе будут зависеть не только от конечного состояния, но и от перехода. При этом алгоритм двойного поиска в глубину, использующийся для поиска контрпримера спецификации, не перестанет работать.

На рис. 26 изображен метод, как помечать состояния для одного автомата. Однако на практике можно это делать динамически, во время обхода автомата алгоритмом верификатора. При этом можно помечать не состояния автомата, а *глобальные состояния* всей модели. Таким образом, все равно, работать с одним автоматом или с системой автоматов. На каждом шаге верификатор будет обрабатывать очередное событие и регистрировать, какие предикаты выполнились в ходе обработки. С точки зрения верификатора, а следовательно, и с точки зрения верифицируемой спецификации, обработка любого события будет происходить в автоматной модели атомарно.

После каждого шага обработки одного входящего в систему автоматов события сохраняется следующая информация: текущие состояния автоматов и информация о пройденных промежуточных состояниях. Таким образом, сразу можно построить автомат, отображающий модель Крипке, причем этот автомат будет глобальным — отражать работу всей системы исходных автоматов.

Если сравнивать этот метод с предыдущими методами, в которых явно выделяются промежуточные состояния, то можно отметить следующие достоинства и недостатки. К достоинствам относится следующее.

- Отсутствие необходимости строить модель Крипке из автоматной модели. Напрямую используется исходная автоматная модель.
- За счет этого не возникает проблемы построения сценария ошибки – сценарий сразу строится в исходной модели.
- Отсутствие проблемы переноса требований из терминов исходной системы автоматов в термины модели Крипке – напрямую используются требования в терминах автоматной модели.
- Количество состояний в модели Крипке оказывается меньше, так как не выделяются промежуточные состояния.
- Возможность применять темпоральный оператор X (*neXt*) в терминах глобальных состояний системы автоматов, тогда как при реализации с выделением промежуточных состояний это невозможно.
- Метод не зависит от реализации поведения автоматной модели программы. Он работает для любого вида иерархических автоматов. Можно написать программу, которая будет напрямую использовать интерпретатор автоматных моделей программ, с условием, что интерпретатор будет выдавать информацию о действиях, произведенных в ходе обработки очередного события, а также будет позволять изменять текущее состояние системы автоматов.

К недостаткам метода можно отнести следующее.

- Обработка события системой автоматов происходит атомарно. Поэтому сложно верифицировать утверждения, которые ограничивают порядок выполнения таких действий, которые могут быть выполнены в пределах одного шага. Однако информацию о порядке выполненных действий в ходе обработки одного события можно записывать в виде отдельных предикатов и использовать их в спецификации.
- Метод работает только для конкретного алгоритма верификации формул темпоральной логики *LTL*.

Подводя итог рассмотренных рекомендаций к построению структуры Крипке из автоматной модели можно заключить, что существует множество идей, как это можно было бы делать, и решение, какой идеей воспользоваться, зависит от конкретной задачи.

Исходя из изложенного, можно утверждать, что, так как модель Крипке имеет конечное число состояний и может формально строиться непосредственно по автоматной модели (графу переходов), то в отличие от программ общего вида, верификация автоматных программ на основе метода *model checking* резко упрощается. При этом показано, что выбор конкретного метода построения модели Крипке зависит от числа состояний верифицируемого автомата и сложности его переходов. Отметим, что в простых случаях модель Крипке в явном виде можно не строить, а использовать вместо нее непосредственно автоматную модель программы.

2.2. Методы описания требований к автоматным моделям управляющих программ в терминах темпоральной логики

Многие исследователи, такие как Р. Берсталл [16], П. Крогер [35], А. Пнуели [45], предлагали использовать темпоральные логики для проверки различных свойств программ. Однако алгоритмы, автоматизирующие такую проверку, были изобретены только в начале 80-х годов прошлого века. Первые алгоритмы проверки на модели для темпоральных логик были предложены независимо Э. Кларком и Э. Эмерсоном [19, 24] и Дж. Кейем и Дж. Сифакисом [46, 47].

В работе [19] Э. Кларк и Э. Эмерсон использовали темпоральные логики для проверки правильности работы параллельных программ. Они рассматривали модель синхронизации (*synchronization skeleton*) программы, опуская все остальные детали. Для такой модели, являющейся по сути конечным автоматом, спецификацию можно записать в виде формулы на языке темпоральной логики. Э. Кларк и Э. Эмерсон использовали логику ветвящегося времени *CTL*. Они разработали алгоритм проверки истинности спецификации, записанной в виде формулы логики *CTL*. Этот алгоритм имеет сложность пропорциональную произведению длины формулы и квадрата количества состояний модели. В этой работе также рассматривалось автоматическое построение модели синхронизации по спецификации.

В 1982 году Дж. Кей и Дж. Сифакис представили интерактивную систему для дизайна распределенных приложений *CESAR* [47]. Эта система позволяет выполнить проверку модели программы на соответствие множеству спецификаций. Модель представляется в виде интерпретируемой сети Петри, а спецификации к ней в виде формул подмножества логики *CTL*. Оценок времени работы алгоритма проверки приведено не было. В работе [47] в качестве примера была рассмотрена проверка корректности протокола чередования битов.

Сравнение линейного (*linear*) и ветвящегося (*branching*) времени в применении к верификации программ привели Э. Эмерсон и Дж. Гальперн в работе [25]. Исследователи рассмотрели заключение Л. Лампорта [38] о том, что ветвящееся время более подходит для проверки недетерминированных программ, тогда как для параллельных систем более предпочтительно линейное время. Они дали формализм для сравнения выразительности темпоральных логик и определили язык логики *CTL**, являющийся расширением *CTL*. Этот язык включает в себя операторы как линейного, так и ветвящегося времени. В статье были приведены сравнения выразительности некоторых подклассов *CTL**. Э. Эмерсон и Дж. Гальперн заключили, что если для проверки некоторых параллельных программ линейного времени достаточно, то для других оно не применимо, и привели пример задачи взаимного исключения, важные свойства которой невозможно выразить в логике линейного времени.

В работе [18] был приведен критерий формулы из *CTL**, выразимой в *LTL*, а также необходимое условие для выразимости формулы из *CTL** в *CTL.* Был показан метод доказательства того, что некоторые свойства (например, честность (fairness)) невозможно записать на языке логики *CTL*.

В 1983 году Э. Кларк, Э. Эмерсон и А. Систла в работе [20] представили улучшенный вариант своего алгоритма, который имеет линейную сложность относительно произведения длины формулы и количества состояний графа переходов в модели, рассматриваемой как структура Крипке. Они также показали, как расширить этот алгоритм для того, чтобы верифицировать один из фундаментальных типов свойства честности. Этот тип состоит в том, что некоторый предикат должен выполняться бесконечно часто вдоль всего честного пути. В статье была предложена модификация логики *CTL*, отличающаяся от нее только по семантике, которая позволяет выразить описанное выше свойство. Также были приведены определения логик *CTL**, *BT** и CTL^{+}, которые являются более выразительными, чем логика *CTL*. В частности, свойство честности легко записывается в виде формулы на *CTL**.

Сложность проверки на модели для логики линейного времени была рассмотрена Э. Кларком и А. Систлой в работе [49]. Они показали, что проверка истинности формулы логики с **F** *NP*-полна, а для формул логик с **F**, **X**, с **U**, с **U**, **S**, **X** *PSPACE*-полна [1].

Метод для работы с честностью, отличный от описанного в работе [20], был разработан Дж. Кейем и Дж. Сифакисом и представлен в [46]. Они использовали понятие честной достижимости (*fair reachability*) некоторого подмножества состояний *P*, то есть такой достижимости, что

рассматриваются только те выполнения, где если состояния из *P* достижимы бесконечно часто, то они посещаются бесконечно часто. В статье была определена логика *FCL* с двумя новыми темпоральными операторами. Дж. Кей и Дж. Сифакис показали, что для любой формулы *f* в логике *FCL* существует формула g в логике *CTL* такая, что из истинности g следует истинность *f*. Таким образом, изучение логики *FCL* может быть выполнено при помощи логики *CTL*.

2.2.1. Описание требований на основе темпоральных логик

Для описания требований и спецификаций относительно поведения управляющей программы необходим естественный способ работы с утверждениями, содержащими временные связи и отношения. Например, утверждения «на небе появляются тучи» и «идет дождь» имеет смысл связать в утверждение «на небе появляются тучи, после чего идет дождь». Раздел логики, занимающийся подобными высказываниями, называется *темпоральной логикой*.

Различают несколько видов темпоральной логики, отличающихся синтаксисом, семантикой и выразительными возможностями — множествами утверждений, которые можно выразить в рамках данной логики. Исторически, первой была предложена *линейная темпоральная логика* (*LTL*), которая эквивалентна по выразительной мощности регулярным выражениям в которых не используются замыкание Клини. Позже, для целей верификации Э. Кларком и Э. Эмерсоном была предложена *вычислительная темпоральная логика* (*CTL*), использующая ветвящуюся модель времени, что отражает возможность недетерминированного развития событий.

Базовые возможности темпоральной логики отражены в предложенной А. Пнуели пропозициональной линейной темпоральной логике (*LTL*).

Пусть символ *AP* обозначает множество, которое будем называть множеством атомарных предложений. Для автоматной технологии атомарными предложениями являются, например, события, входные и выходные переменные и т. п.

Тогда множество всех формул, определяемых в нотации Бэкуса-Наура, как

$\phi ::= p \in AP \mid \neg\phi \mid \phi \vee \phi \mid \mathbf{X}\phi \mid \phi \mathbf{U} \phi,$

составляет синтаксис языка *LTL*.

Обратим внимание, что множество формул, построенных по первым трем правилам, дает все формулы пропозициональной логики – истинный подкласс языка *LTL*. При этом **X** (*neXt*) и **U** (*Until*) являются темпоральными операциями.

Для упрощения введем следующие две темпоральные операции:

$\mathbf{F}\phi \equiv \mathbf{1U}\phi$;

$\mathbf{G}\phi \equiv \neg\mathbf{F}\neg\phi$.

В дальнейшем будем использовать упрощенные формулы, не обращаясь каждый раз к определению языка.

Таким образом, в языке *LTL* операции **F** (*Future*) и **G** (*Globally*) выражены через базовую операцию **U**. Как будет показано далее, в языке *CTL* эти операции (точнее, их «разветвленные» варианты) сами могут быть базовыми, причем выбор базовых операций даст большую гибкость в разработке алгоритмов верификации.

Определив таким образом синтаксис языка *LTL*, формализуем его семантику.

Пусть символ *S* обозначает множество, которое будем называть множеством состояний.

LTL-моделью называется тройка $\boldsymbol{M} = (S;\ R: S \rightarrow S;\ Label \subseteq S \times AP)$, где *R* – тотальная (всюду определенная) функция на *S*, называемая отношением переходов.

Смысл формул логики определяется в терминах отношения семантической истинности (выполнения) $\vDash$ между моделью $\boldsymbol{M}$, одним из ее состояний *s* и формулой ϕ. Выражение $(\boldsymbol{M}, s, \phi) \in \vDash$ будем записывать в инфиксной нотации: $\boldsymbol{M}, s \vDash \phi$. Когда модель $\boldsymbol{M}$ ясна из контекста, будем исключать ее из формулы и писать $s \vDash \phi$.

Отношение выполнения $\vDash$ определяется в табл. 1.

Таблица 1. Семантика языка *LTL*

Формула	Значение
$s \vDash p$	$(s, p) \in Label$
$s \vDash \neg\phi$	$\neg(s \vDash \phi)$
$s \vDash \phi \vee \psi$	$(s \vDash \phi) \vee (s \vDash \psi)$
$s \vDash \mathbf{X}\phi$	$R(s) \vDash \phi$
$s \vDash \phi \mathbf{U} \psi$	$\exists j \geq 0 \mid \left(R^j(s) \vDash \psi \wedge \left(\forall 0 \leq k < j : R^k(s) \vDash \varphi\right)\right)$

Стоит отметить, что для семантики языка *LTL* существует корректная и полная формальная система аксиом.

При всех достоинствах линейной темпоральной логики, она поддерживает лишь линейную — неветвящуюся модель времени, что соответствует детерминированному ходу истории. Во многих естественных случаях требуется возможность формулировать высказывания, отражающие существенную недетерминированность хода событий, например, «если тучи появляются, то рано или поздно (при любом ходе событий) пойдет дождь».

Введем для этих целей синтаксис вычислительной темпоральной логики *CTL*. Для множества атомарных формул *AP* набор *CTL*-формул определяется следующим образом:

$\phi ::= p \in AP \mid \neg\phi \mid \phi \vee \phi \mid \mathbf{EX}\phi \mid \mathbf{E}[\phi \mathbf{U} \phi] \mid \mathbf{A}[\phi \mathbf{U} \phi]$.

Введем также дополнительные темпоральные операции:

$$\mathbf{EF}\phi \equiv \mathbf{E}[\mathbf{1U}\phi], \quad \mathbf{AF}\phi \equiv \mathbf{A}[\mathbf{1U}\phi];$$

$$\mathbf{EG}\phi \equiv \neg\mathbf{AF}\neg\phi, \quad \mathbf{AG}\phi \equiv \neg\mathbf{EF}\neg\phi;$$

$$\mathbf{AX}\phi \equiv \neg\mathbf{EX}\neg\phi.$$

Таким образом, отличие языка *CTL* от языка *LTL* состоит в том, что каждой введенной темпоральной операции соответствует префикс **E** или **A**, называемый *квантификатором пути*. Традиционные техники, используемые для эффективной проверки моделей линейной темпоральной логики, принципиально отличаются от тех, которые применяются для ветвящейся темпоральной

логики. Синтаксис языка *CTL* требует, чтобы квантификаторы шли непосредственно перед темпоральными операциями **X**, **F**, **G** или **U**.

Если опустить это ограничение, то получится более выразительная ветвящаяся темпоральная логика *CTL**. Логика *CTL** позволяет квантификатору пути располагаться перед произвольной *LTL*-формулой. Формально язык *CTL** имеет следующую нотацию:

$\phi ::= p \in AP \mid \neg\phi \mid \phi \vee \phi \mid \mathbf{E}\psi$ – формулы состояния;

$\psi ::= \phi \mid \neg\psi \mid \psi \vee \psi \mid \mathbf{X}\psi \mid \psi \mathbf{U} \psi$ – формулы пути.

Опишем теперь семантику языка *CTL*.

CTL-моделью для множества состояний *S* называется тройка $\boldsymbol{M} = (S;\ R \subseteq S \times S;\ Label \subseteq S \times AP)$, где *R* – тотальное отношение на множестве *S*.

Путем называется бесконечная последовательность состояний $s_0 s_1 s_2 \ldots$ таких, что $(s_i, s_{i+1}) \in R$ для всех целых неотрицательных *i*.

Множество $P_{\boldsymbol{M}}(s)$ путей, которые начинаются в состоянии *s,* определяется следующим образом: $P_{\boldsymbol{M}}(s) = \{\sigma \in S^{\omega} \mid \sigma_0 = s\}$.

Отношение выполнения $\models$ строится как и в языке *LTL* и определяется согласно табл. 2.

Таблица 2. Семантика языка *CTL*

Формула	Значение
$s \models p$	$(s, p) \in Label$
$s \models \neg\phi$	$\neg(s \models \phi)$
$s \models \phi \vee \psi$	$(s \models \phi) \vee (s \models \psi)$
$s \models \mathbf{EX}\phi$	$\exists \sigma \in P_{\boldsymbol{M}}(s) \mid \sigma_1 \models \phi$
$s \models \mathbf{E}[\phi \mathbf{U} \psi]$	$\exists \sigma \in P_{\boldsymbol{M}}(s) \mid \left(\exists j \geq 0 \mid \left(\sigma_j \models \psi \wedge \left(\forall 0 \leq k < j : \sigma_k \models \varphi\right)\right)\right)$
$s \models \mathbf{A}[\phi \mathbf{U} \psi]$	$\forall \sigma \in P_{\boldsymbol{M}}(s) \mid \left(\exists j \geq 0 \mid \left(\sigma_j \models \psi \wedge \left(\forall 0 \leq k < j : \sigma_k \models \varphi\right)\right)\right)$

С практической точки зрения не имеет смысла подробно рассматривать применение языка *CTL** для *Model checking*, так как формальная семантика его интуитивно ясна из интерпретации *CTL*, а задача проверки моделей для этой логики *PSPACE*-полна по отношению к размеру спецификации. В то же время для языка *CTL* существуют эффективные алгоритмы проверки моделей.

Важным теоретическим шагом является сравнение выразительности описанных выше темпоральных логик. Для этого представим синтаксис языков *LTL* и *CTL* в терминах языка *CTL**. Например, будем говорить, что формула ψ логики *LTL* записана в терминах языка *CTL**, если $\psi = \mathbf{A}\phi$, где $\phi \in LTL(AP)$ (простейший подход).

Для языка *CTL* перепишем синтаксис в следующем виде:

$\phi ::= p \in AP \mid \neg\phi \mid \phi \vee \phi \mid \mathbf{E}\psi$ – формулы состояния;

$\psi ::= \neg\psi \mid \mathbf{X}\phi \mid \phi \mathbf{U} \phi$ – формулы пути.

Данное определение синтаксиса языка *CTL* эквивалентно исходному его определению.

Из рассмотрения синтаксиса следует, что *LTL*, $CTL \subseteq CTL^*$. Обратное неверно (как минимум синтаксически).

2.2.1.1. Описание темпоральной логики *CTL**

Jпишем наиболее выразительную логику — *CTL**. Формулы *CTL** строятся при помощи добавления темпоральных операторов: **X**, **F**, **G**, **U**, **R**. Приведем их неформальное описание:

- **X**φ — (neXt) оператор сдвига означает, что формула φ должна выполняться для следующего состояния пути. Оператор **X** можно использовать несколько раз подряд. Например, формула **XXX**φ означает, что φ выполниться, начиная с четвертого состояния пути.
- **F**φ — (Future) оператор будущего. В отличие от оператора **X**, означает, что формула φ выполнится не в каком-то конкретном состоянии на пути, а то, что на пути существует состояние, когда формула φ выполнится.
- **G**φ — (Global) оператор логически двойственен оператору **F** и означает что данная формула φ будет выполняться в каждом состоянии пути.
- φ**U**ψ — (Until) оператор истинен, только если когда-нибудь выполнится формула ψ, и до этого момента всегда будет выполняться φ.

- $\varphi\mathbf{R}\psi$ — (Release) оператор логически двойственен оператору **U**. Он значит, что формула ψ выполняется до тех пор, пока формула φ хоть раз не станет истиной.

Помимо этих операторов в темпоральных логиках используются операторы логики утверждений: $\wedge$, $\vee$, $\neg$, и кванторы: $\mathbf{E}\varphi$ – существует путь, который удовлетворяет формуле φ, $\mathbf{A}\varphi$ – любой путь удовлетворяет формуле φ.

Строго определим синтаксис *CTL** формулы. Будем различать два вида формул: формулы состояний — формулы, описывающие состояние модели и формулы пути — формулы, описывающие некоторый путь в модели Крипке. Формулой состояния, будем называть формулу, построенную следующим образом:

1. Атомарное предположение ($S \in AP$) является формулой состояния.
2. Если φ и ψ – формула состояния то $\varphi \wedge \psi$, $\varphi \vee \psi$, $\neg\varphi$ – формулы состояния.
3. Если φ – формула пути то $\mathbf{E}\varphi$, $\mathbf{A}\varphi$ – формулы состояния.
4. Формулу пути определим следующим образом.
5. Если φ – формула состояния, то φ – формула пути.
6. Если φ, ψ – формулы пути, то $\mathbf{X}\varphi$, $\mathbf{F}\varphi$, $\mathbf{G}\varphi$, $\varphi\mathbf{U}\psi$, $\varphi\mathbf{R}\psi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\neg\varphi$ – формулы пути.

Определим семантику формулы логики *CTL**. Смысл формул логики определяется в терминах отношения семантической истинности (выполнения) $\models$ между моделью M, одним из ее состояний s и формулой состояния φ. Выражение $(M, s, \varphi) \models$ будем записывать в инфиксной нотации: $M, s \models \varphi$. Когда модель M ясна из контекста, будем исключать ее из формулы и писать $s \models \varphi$. Аналогично будем записывать выражение истинности между моделью M, одним из ее путей π и формулой пути ψ. Будем использовать выражения $M, \pi \models \psi$ и $\pi \models \psi$, когда модель ясна из контекста. Соотношение $\models$ определяется рекурсивно следующим образом:

1. $M, s \models p \Leftrightarrow p \in \mathrm{L}(s)$;
2. $M, s \models \neg\varphi \Leftrightarrow M, s \not\models \varphi$;
3. $M, s \models \varphi_1 \wedge \varphi_2 \Leftrightarrow M, s \models \varphi_1$ и $M, s \models \varphi_2$;
4. $M, s \models \varphi_1 \vee \varphi_2 \Leftrightarrow M, s \models \varphi_1$ или $M, s \models \varphi_2$;
5. $M, s \models \mathbf{E}\psi \Leftrightarrow$ существует путь π, начинающийся с вершины s такой что $M, \pi \models \psi$;

6. $M, s \models \mathbf{A}\psi \Leftrightarrow$ для любого пути π, начинающийся с вершины s выполнено $M, \pi \models \psi$;
7. $M, \pi \models \varphi \Leftrightarrow$ для первого состояния s в пути π выполнено $M, s \models \varphi$;
8. $M, \pi \models \neg\psi \Leftrightarrow M, \pi \not\models \psi$;
9. $M, \pi \models \psi_1 \wedge \psi_2 \Leftrightarrow M, \pi \models \psi_1$ и $M, \pi \models \psi_2$;
10. $M, \pi \models \psi_1 \vee \psi_2 \Leftrightarrow M, \pi \models \psi_1$ или $M, \pi \models \psi_2$;
11. $M, \pi \models \mathbf{X}\psi \Leftrightarrow M, \pi^1 \models \psi$;
12. $M, \pi \models \mathbf{F}\psi \Leftrightarrow$ существует $i \geq 0$, такое что $M, \pi^i \models \psi$;
13. $M, \pi \models \mathbf{G}\psi \Leftrightarrow$ для любого $i \geq 0$, выполняется $M, \pi^i \models \psi$;
14. $M, \pi \models \psi_1 \mathbf{U} \psi_2 \Leftrightarrow$ существует $i \geq 0$, такое что $M, \pi^i \models \psi_2$, и для каждого $0 \leq j < i$ выполнено $M, \pi^j \models \psi_1$;
15. $M, \pi \models \psi_1 \mathbf{R} \psi_2 \Leftrightarrow$ для каждого $j \geq 0$, если для любого $i < j$ верно, что $M, \pi^i \not\models \psi_1$, то $M, \pi^j \models \psi_2$.

Формулу *CTL** можно записать, пользуясь меньшим набором операторов. Например, покажем что операторов $\vee$, $\neg$, **X**, **U**, **E** достаточно для выражения любой формулы *CTL**:

1. $\varphi \wedge \psi \equiv \neg(\neg\varphi \vee \neg\psi)$;
2. $\varphi \mathbf{R} \psi \equiv \neg(\neg\varphi \mathbf{U} \neg\psi)$;
3. $\mathbf{F}\varphi \equiv \text{True}\mathbf{U}\neg\varphi$;
4. $\mathbf{G}\varphi \equiv \neg\mathbf{F}\neg\varphi$;
5. $\mathbf{A}\varphi \equiv \neg\mathbf{E}\neg\varphi$;

2.2.1.2. Описание темпоральной логики *LTL*

LTL (*linear temporal logic*) – линейная темпоральная логика. Формулы из *LTL* это подмножество формул из *CTL**, которые соответствуют виду $\mathbf{A}\varphi$, где φ — формула, содержащая в себе только формулы пути и атомарные предположения.

Введем синтаксис *LTL*-формулы. Тогда *LTL*-формула определяется как **A**φ, где φ – *LTL*-формула пути. Она определяется по правилам:

1. Атомарное предположение ($S \in$ AP) является *LTL*-формулой пути.
2. Если φ, ψ – *LTL*-формулы пути, то **X**φ, **F**φ, **G**φ, φ**U**ψ, φ**R**ψ, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\neg\varphi$ – *LTL*-формулы пути.

Семантика *LTL*-формулы аналогична семантике соответствующей CTL*-формулы:

$(M, s \vDash LTL\ \varphi) \equiv (M, s \vDash CTL^*\ \varphi)$.

2.2.1.3. Описание темпоральной логики *CTL*

Формулы логики *CTL* содержат только формулы состояния. Все темпоральные операторы в *CTL* употребляются только вместе с кванторами пути (**AX**, **AF**, **AG**, **AU**, **AR**, **EX**, **EF**, **EG**, **EU**, **ER**).

Строго определим синтаксис *CTL*-формулы.

Атомарное предположение ($S \in$ AP) является *CTL*-формулой.

Если φ, ψ — *CTL*-формулы, то **AX**φ, **AF**φ, **AG**φ, **A**φ**U**ψ, **A**φ**R**ψ, **EX**φ, **EF**φ, **EG**φ, **E**φ**U**ψ, **E**φ**R**ψ, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\neg\varphi$ — *CTL*-формулы.

Семантика *CTL*-формулы аналогична семантике соответствующей *CTL**-формулы:

$(M, s \vDash_{CTL} \varphi) \equiv (M, s \vDash CTL^*\ \varphi)$.

Формулу *CTL* можно записать, пользуясь меньшим набором операторов. Например, каждый из операторов: **AX**, **AF**, **AG**, **AU,** **AR**, **EF**, **ER** может быть выражен с использованием всего трех операторов **EX**, **EG**, **EU**:

- **AX**$\varphi \equiv \neg$**EX**$\neg\varphi$;
- **EF**$\varphi \equiv$**E**(True**U**$\neg\varphi$);
- **AG**$\varphi \equiv \neg$**AF**$\neg\varphi$;
- **AF**$\varphi \equiv \neg$**EG**$\neg\varphi$;
- **A**(φ**U**ψ) $\equiv \neg$**E**($\neg\varphi$**U**($\neg\varphi \wedge \neg\psi$)) $\wedge$ **AF**φ;
- **A**(φ**R**ψ) $\equiv \neg$**E**($\neg\varphi$**U**$\neg\psi$);
- **E**(φ**R**ψ) $\equiv \neg$**A**($\neg\varphi$**U**$\neg\psi$).

2.2.1.4. Сравнение выразительной мощности различных темпоральных логик

Важным теоретическим шагом является сравнение выразительности описанных выше темпоральных логик. Из рассмотрения синтаксиса следует, что *LTL*, *CTL* $\subseteq$ *CTL**. Обратное неверно (как минимум синтаксически).

Будем называть формулы ϕ и ψ языка *CTL** эквивалентными, если

$\boldsymbol{M},s \models \phi \Leftrightarrow \boldsymbol{M},s \models \psi$

для любой модели $\boldsymbol{M}$ и ее состояния *s*.

Пусть *L* и *L′* – темпоральные логики. Будем говорить, что *L* не более выразительна, чем *L′*, если для любой формулы из *L* найдется эквивалентная ей формула из *L′*. Здесь предполагается, что *L, L′* $\subseteq$ *CTL**.

Рассмотрим формулу из *CTL**: **A**((**G** *life*) $\Rightarrow$(**GEX** *death*)).

Она обозначает, что для любого пути, у которого в каждом состоянии выполнено *life*, для каждого состояния существует следующее состояние, в котором выполнено *death*. В работе [38] Л. Лампорт показал, что для данной формулы не существует эквивалента из *LTL*.

Теперь все готово для сравнения выразительных возможностей. Рис. 27 отражает соотношения между тремя рассматриваемыми логиками в терминах выразительности.

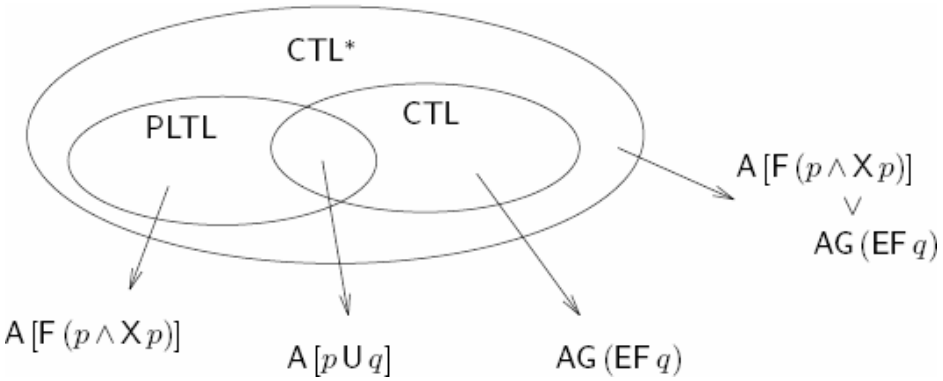


Рис. 27. Соотношения между *LTL*, *CTL* и *CTL**

2.2.1.5. Задача проверки модели

Возникает задача: для всех формул φ из некоторого языка L, всех конечных структур Крипке M и состояний s проверить, выполняется ли данная формула на данной структуре Крипке.

Дадим формальное определение задачи проверки модели. *Задача проверки модели* для логики L состоит в том, чтобы установить принадлежит ли слово языку:

$$MC(L) \overset{def}{=} \{\langle M, s_0, \varphi \rangle \mid \varphi \in L \wedge M, s_0 \vDash \varphi\}$$

Определим задачу проверки выполнимости формулы φ. Формула φ называется *выполнимой*, если существует такая структура Крипке M и такое состояние s, что выполняется $M, s_0 \vDash \varphi$. Задача *проверки выполнимости формулы* для логики L состоит в том, что бы проверить принадлежит ли слово языку:

$$SAT(L) \overset{def}{=} \{\varphi \mid \varphi \in L \wedge \varphi - выполнима\}$$

Приведем идею доказательства того, что задача проверки модели является *PSPACE*-полной, относительно длины формулы для формул из логик *LTL* и *CTL**. Сначала приведем доказательство для *LTL*. Для этого нам нужно показать, что она принадлежит *PSPACE*, и является *PSPACE*-трудной. Затем, пользуясь этим фактом, покажем, что *CTL** является *PSPACE*-полной.

Задачу проверки $M, s_0 \vDash \mathbf{A}\varphi$ перепишем в виде $M, s_0 \vDash \neg\mathbf{E}\neg\varphi$. Покажем, что формула вида $\mathbf{E}\varphi$, где φ – *LTL*-формула пути, принадлежит *NPSPACE* [1].

Сведем задачу проверки формулы к задаче выполнимости формулы. Для этого построим формулу φ_M, которая будет заменять модель M. Введем дополнительные атомарные предположения p_s для каждого $s \in S$. Для начального состояния s_0 заведем атомарное предположение p_{s0} соответственно. Для каждого состояния s построим формулы трех видов.

Формула f_s, утверждает, что мы находимся в состоянии s , и не находится не в каком другом состоянии:

$$f_s = p_s \wedge \neg(\bigwedge_{s' \in S/\{s\}} p_{s'})$$

Формула g_s утверждает, что в состоянии s выполняются атомарные предположения соответствующие ему и не выполняются все остальные:

$$g_s = \bigwedge_{p \in L(s)} p \wedge \neg(\bigwedge_{p' \notin L(s)} p').$$

Формула h_s утверждает, что следующим состоянием должно быть одно из состояний, в которые есть переход из s:

$$h_s = \mathrm{X}(\bigvee_{(s,s')\in R} p_{s'}).$$

Получим окончательную формулу $r(\mathbf{E}\varphi) = \mathbf{E}(\varphi \wedge \mathbf{G}(\bigvee_{s\in S}(f_s \wedge g_s \wedge h_s) \wedge p_{s0})$. Опустим доказательство того, что $M, s_0 \vDash \mathbf{E}\varphi$ тогда и только тогда, когда формула $r(\mathbf{E}\varphi)$ – выполнима.

Осталось доказать, что задача проверки выполнимости формулы принадлежит *PSPACE*. Для этого воспользуемся теоремой о периодической, в конечном счете, модели. Ее доказательство приводиться в работе [50].

Дадим ее формулировку. Формула $\mathbf{E}\varphi$ выполнима, только когда она выполнима на периодическом, в конечном счете, пути: $\pi = (s_0, s_1, ...)$. При этом путь имеет непериодическую часть $l \le 2^{1+|\varphi|}$ и период $k \le 4^{1+|\varphi|}$. Причем для каждой подформулы вида $\varphi\mathbf{U}\psi$, для которой $\pi^i \vDash \varphi\mathbf{U}\psi$, $i \ge l$ выполняется $\exists j, i \le j \le i + k$ такой, что $\pi^j \vDash \psi$.

Приведем алгоритм проверки выполнимости *LTL* формулы для недетерминированной машины Тьюринга.

Вычислим l, k, $Sub_{present}$:

```
If (φ ∉ Sub_present) Then Return false
For i := 1 To l Do
  Угадаем Sub_next
  If not consistent(Sub_present, Sub_next) Then Return false
  Sub_present := Sub_next
End For
Sub_period := Sub_present
∀g = φUψ ∈ Sub_period : A_g := false
For i := l+1 To l+k Do
  Угадаем Sub_next
  If not consistent(Sub_present, Sub_next) Then Return false
  ∀g = φUψ ∈ Sub_period, ψ ∈ Sub_present : A_g := true
  Sub_present := Sub_next
End For
If ∃g = φUψ ∈ Sub_period, A_g := false Then Return false
Sub_next := Sub_period
If not consistent(Sub_present, Sub_next) Then Return false
Return true
```

Функция *consistent*($Sub_{present}$, Sub_{next}), должна проверить, что все подформулы в текущем состоянии и в следующем состоянии, не противоречат друг другу. Будем считать, что в формуле используются только операторы **X**, **U**, $\neg$, $\wedge$. Тогда непротиворечивость можно выразить следующими условиями:

$$g = Xf_1 \in Sub_{present} \Leftrightarrow f_1 \in Sub_{next};$$
$$g = f_1 U f_2 \in Sub_{present} \Leftrightarrow f_2 \in Sub_{present} \; or \; (f_1 \in Sub_{present} \; and \; f_1 U f_2 \in Sub_{next});$$
$$g = f_1 \wedge f_2 \in Sub_{present} \Leftrightarrow f_1, f_2 \in Sub_{present};$$
$$g = \neg f_1 \in S_{present} \Leftrightarrow f_1 \notin Sub_{present}$$

Пользуясь данным алгоритмом, можно проверить *LTL*-формулу на недетерминированной машине Тьюринга. Для хранения множества подформул потребуется полиномиальная память, следовательно, задача проверки принадлежит *NPSPACE = PSPACE*.

Теперь докажем, что задача проверки *LTL*-формулы является *PSPACE*-трудной. Для этого сведем к ней задачу замощения. Пусть C — некоторое множество цветов, $D \subset C^4$ – множество плиток, каждое ребро плитки окрашено в один из цветов $c \in C$. Замощением некоторого региона $R \subset Z^2$ будем называть отображение $f : R \to D$ такое, что у плиток смежные ребра окрашены в одинаковый цвет. На рис. 28 приведен пример замощения квадрата 4×4 для трех цветов и четырех видов плиток.

Задача о замощении заключается в том, что у нас есть набор плиток D ($|D| = n$) и два выделенных цвета: $c_0, c_1 \in C$. Требуется определить, можно ли замостить этими плитками, какой-либо квадрат $n \times m$, так что бы верхняя сторона этого квадрата была окрашена в цвет c_0, а нижняя в цвет c_1. Данная задача является *PSPACE*-полной. Сведем ее к проверке формулы *LTL*.

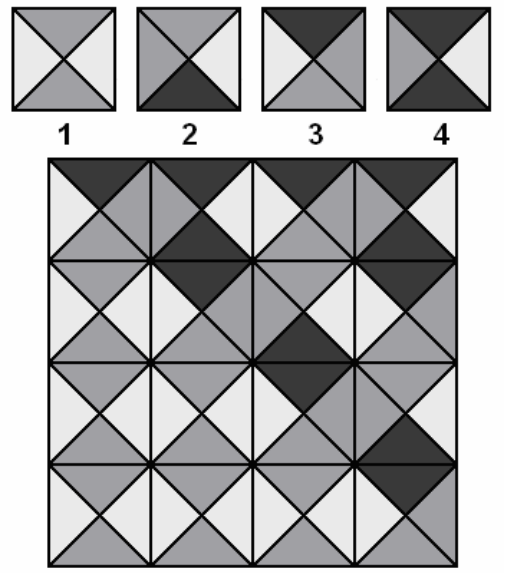


Рис. 28. Экземпляр задачи о замощении

Для данного набора плиток $D = \{d_1, d_2, ...d_n\}$ построим структуру Крипке, как показано на рис. 29.

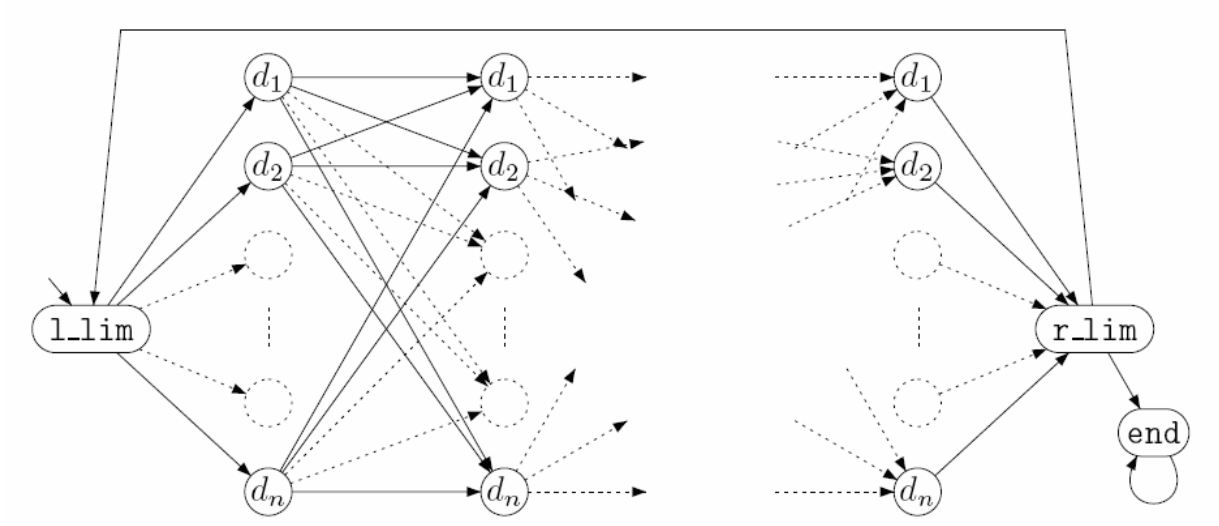


Рис. 29. Структура Крипке для задачи о замощении

Структура состоит из *n* рядов состояний. В каждом ряду состояния соответствуют плиткам, каждое состояние имеет четыре пометки: *up* = c_0, *down* = c_1, *left* = c_2 *right* = c_3, где c_0, c_1, c_2, c_3 — цвета

верхней, нижней, левой и правой граней плитки соответственно. Из каждого состояния из *i*-го ряда имеется переход в каждое состояние из *i*+1-го.

Путь, проходящий через эту структуру, сначала будет описывать плитки первого ряда, затем вернется в начальное состояние, и будет описывать плитки последующих рядов. В конце он переходит в состояние *end*. После этого постоянно переходит в указанное состояние. Такой путь соответствует некоторому расположению плиток на квадрате $n \times m$. Теперь напишем *LTL*-формулы, которым путь должен удовлетворять, для того чтобы описывать замощение.

Нижняя и верхняя границы существуют и окрашены в цвета c_0 и c_1:

$$\bigwedge_{k=1}^{n} \mathbf{X}^{k}(up = c_0) \wedge \mathbf{F}(l\_\lim \wedge \bigwedge_{k=1}^{n} \mathbf{X}^{k}(down = c_1) \wedge \mathbf{X}^{n+2} end) \quad (\varphi_1)$$

Соседние плитки имеют одинаковые цвета на смежных гранях:

$$\bigwedge_{c \in C} (right = c \Rightarrow \mathbf{X}(r\_\lim \vee left = c) \wedge up = c \Rightarrow \mathbf{X}(end \vee down = c)) \quad (\varphi_2)$$

Набором плиток *D* замостить никакой квадрат $n \times m$ невозможно, если $M_D, s \vDash \mathbf{A} \neg(\varphi_1 \wedge \varphi_2)$.

Данное сведение доказывает *PSPACE*-полноту задачи проверки формулы *LTL*.

Множество формул *LTL* является подмножеством *CTL**, следовательно, проверка формулы из *CTL** является *PSPACE*-трудной задачей.

Осталось показать, что проверка формулы из *CTL** принадлежит *PSPACE*. *M* – структура Крипке, *s* – начальное состояние, ψ – формула из *CTL**.

Задача состоит в том, чтобы проверить утверждение $M, s_0 \vDash \psi$. Будем использовать уже известный алгоритм проверки формулы из *LTL*.

Можно считать, что в формуле используются только операторы $\vee$, $\neg$, **X**, **U**, **A**.

Исключим из формулы все кванторы **A**. Для этого будем использовать индукцию по подформулам. Применим следующий алгоритм упрощения *CTL**-формулы:

```
simple(φ:Fornula): Formula;
Begin
  If φ ∈ AP Then Return φ
  If φ = ¬φ1 Then Return ¬simple(φ1)
  If φ = φ1∧φ2 Then Return simple(φ1)∧simple(φ2)
  If φ = Xφ1 Then Return X simple(φ1)
  If φ = φ1Uφ2 Then Return simple(φ1) U simple(φ2)
  If φ = Eφ1 Then Begin
    φ1 := simple(φ1);
```

```
    Создадим новое атомарное предположение p_φ, добавим его в
        AP.
    For s ∈ S do begin
      Так как φ_1 не содержит кванторов E, мы можем использовать
        для нее алгоритм проверки LTL.
      If M, s ⊨ φ Then пометим состояние s атомарным
        предположением p_φ
    End
    Return p_φ
  End
End.
```

После применения данного алгоритма задача проверки формулы станет задачей проверки обычной логической формулы, которая решается за полиномиальное время.

Для упрощения формулы нам потребуется $O(|\varphi| \cdot |S|)$ запусков алгоритма проверки *LTL*-формулы, что потребует полиномиальной памяти. Следовательно, задача проверки формулы *CTL** является *PSPACE*-полной.

2.2.2. Описание требований в терминах автоматных моделей управляющих программ

В данном разделе будет подробно рассмотрен такой этап верификации автоматных моделей программ, как разработка спецификации.

Целью верификации является доказательство того, что построенная модель работает правильно. Обычно условия корректности работы программы начинаются с некоторых общих представлений разработчика о том, как программа должна работать. Таким образом, спецификация начинается со словесного описания набора правил. Примерами таких правил являются, например, «лифт не должен двигаться с открытыми дверьми» и «бойлер не должен нагреваться, если в нем нет воды». Существуют также общие требования и свойства, которые должны выполняться в любой автоматной модели [2]. Опишем примеры таких свойств.

Тупиковое состояние. Свойство, описывающее возможность попадания автоматной программы в тупиковое состояние, из которого нельзя выйти, может быть проверено для любой автоматной программы. Такая ситуация может возникнуть из-за неправильных условий переходов или вообще при полном отсутствии переходов из состояния автомата. Также стоит отметить, что одно состояние может в разные моменты времени оказаться как не тупиковым, так и тупиковым. Это свойство состояния может зависеть от состояний других автоматов системы, которые влияют на

значения условий на переходах. Возникновение ситуации тупикового состояния свидетельствует о неправильном проектировании системы: в таком состоянии система «повисает» и перестает отвечать на любые воздействия. Единственным исключением является конечное состояние главного автомата – оно по определению должно быть тупиковым и останавливать работу всей системы.

Тупиковое состояние вложенного автомата. Это свойство, описывающее возможность возникновения ситуации, когда один из вложенных автоматов перестает реагировать на какие-либо события, оставаясь в тупиковом состоянии. Как и в предыдущем примере, такая ситуация может возникнуть в результате неправильных условий на переходах или вообще при отсутствии переходов. Это скорее глобальное свойство системы, а не свойство конкретного состояния конкретного вложенного автомата, так как состояния других автоматов также могут влиять на возникновение ситуации тупикового состояния вложенного автомата. Такая ситуация не фатальна для реактивной системы, в том смысле, что главный автомат продолжает обрабатывать события. Однако она может привести к неработоспособности некоторого модуля системы, за который отвечает «повисший» автомат. Это свидетельствует о неправильном проектировании автоматной модели.

Забытый автомат. Это ситуация, когда с некоторого момента времени некоторому вложенному автомату никогда не будет передано управление. Эта ситуация может возникнуть в результате того, что родительский автомат никогда не попадет в состояние, в которое вложен рассматриваемый вложенный автомат.

Живость переходов. Для любого перехода автоматной программы (любого перехода любого автомата) из любого состояния должен существовать путь в то состояние системы автоматов, в котором возможно выполнение этого перехода. Это свойство необходимо в предположении, что автоматная программа работает по бесконечному циклу. Однако, это не означает, что любой переход обязательно сработает на протяжении любого пути (любой истории). Может существовать множество путей из начального состояния, на всем протяжении которых этот переход не сработает. Такие пути являются просто бесконечными циклами, также проходящими через заданный переход. Однако если существует «мертвый» переход, который не будет активирован ни в одной истории работы системы, то это свидетельствует о том, что при разработке автоматной программы допущена ошибка.

Для верификации конкретных примеров автоматных моделей, однако, более интересны утверждения частного вида, а не только перечисленные свойства, применимые для всех автоматных моделей. Поскольку верификатор может работать только с формальной спецификацией, перед

разработчиком возникает задача переноса своих представлений о том, как должна работать программа на формальный язык. Решение этой задачи можно разбить на два этапа: перенос словесного описания спецификации на язык темпоральной логики в терминах автоматной модели, а затем перенос полученной спецификации из терминов автоматной модели в термины построенной модели Крипке.

Обычная словесная спецификация поведения системы состоит из элементарных утверждений, связанных логическими, условными и временными связками. Приведем пример такого утверждения: «кнопка «открыть» открывает двери лифта». Это утверждение можно перефразировать: «если была нажата кнопка «открыть», то двери откроются». В данном утверждении естественным образом содержатся два события: нажатие кнопки «открыть» и открытие дверей. Эти два события связаны условной и временной зависимостью: «*если* кнопка нажата, то двери *будут* открыты». Таким образом, исходное словесное утверждение естественным образом формулируется в терминах темпоральной логики (например, *LTL*): `(p1 → F p2)`, где `p1` означает «нажата кнопка «открыть», а `p2` — «двери открыты». Не удивительно, что это получилась весьма простая формула, ведь темпоральных логики специально создавались для формализации словесных утверждений, описывающих временную зависимость событий.

Теперь обратимся к задаче переноса выделенных элементарных утверждений в термины автоматной модели. Обычные элементарные утверждения, фигурирующие в словесной спецификации, описывают некоторые внешние свойства реактивной системы — происходящие события и состояния объектов. При хорошем проектировании автоматной модели управляющей программы такие утверждения должны естественным образом переноситься из словесной формы в формальные утверждения, касающиеся автоматной модели — в утверждения о происходящих событиях, состояниях автоматов и вызываемых воздействиях на объектах управления. Стоит заметить, что преимущество автоматных моделей состоит в том, что они могут наглядно изображать поведение системы. Если пользоваться этим преимуществом, то при проектировании системы обычно создаются те же естественные абстракции, что используются в спецификации. Поэтому процесс переноса словесных утверждений в утверждения автоматной модели обычно не составляют труда. Однако здесь есть некоторые тонкости. Опишем их на примере.

Рассмотрим задачу разработки автоматной модели, управляющей работой дверей лифта. В упрощенной системе сразу выделяются следующие очевидные объекты: двери и кнопки

открыть/закрыть для их открытия/закрытия. Так как кнопки используются для управления дверьми, то естественно для каждой кнопки выделить событие, происходящее при ее нажатии. Обозначим событие «нажатие кнопки «открыть» – как `e1`, а событие «нажатие кнопки «закрыть» – как `e2`. Для объекта управления «двери» в автомате выделяются выходные воздействия «открыть двери» (обозначим как `z1`) и «закрыть двери» (обозначим как `z2`). Построим для начала простейший управляющий автомат, состоящий лишь из одного состояния (рис. 30).

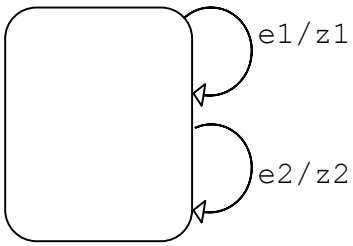


Рис. 30. Простейший автомат, управляющий дверьми лифта

Для данного автомата элементарное утверждение «нажата кнопка «открыть» будет соответствовать событию `e1`, а утверждение «открыть двери» будет соответствовать вызову выходного воздействия `z1`. Тогда итоговая формула спецификации примет вид: `(e1 → F z1)`. Отметим, что если в автомате `e1` – это обозначение события, то в приведенной формуле запись `e1` трактуется как предикат, принимающий значение `true` в состоянии модели Крипке, соответствующем моменту времени, когда произошло событие `e1` (для разных методов построения модели Крипке такое состояние может строиться по-разному).

Приведенная автоматная модель является очень простой. Она работает лишь тогда, когда управляющая система всегда одинаково реагирует на события. Теперь предположим, что имеется только одна кнопка как для открытия, так и для закрытия дверей. Обозначим событие, возникающее при нажатии этой кнопки как `e1`. В этом случае в управляющем автомате придется выделить, по крайней мере, два состояния, для того, чтобы различать реакцию дверей в зависимости от их текущего состояния. Новый автомат изображен на рис. 31.

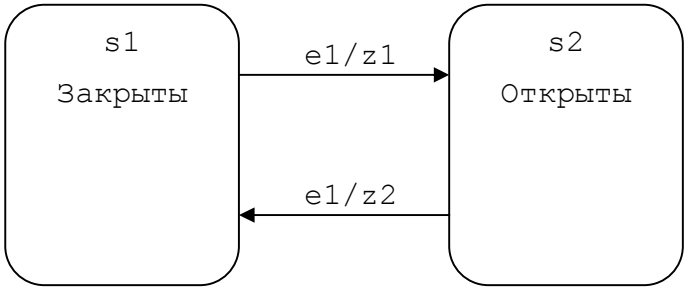


Рис. 31. Автомат, управляющий дверьми лифта с одной кнопкой

Рассмотрим теперь, как закодировать элементарное утверждение «двери открыты» в новом автомате. Теперь это можно сделать двумя способами, используя `z1` или `s2` (напомним, что в данном случае это предикаты). Первый предикат означает, что было вызвано выходное воздействие `z1` (открыть двери), а второй – что управляющий автомат находится в состоянии `s2` (Открыты).

Возникает вопрос, какое утверждение лучше, удобнее: утверждение, касающееся воздействия, или утверждение, касающееся состояния автомата. Оказывается, что исходные элементарные утверждения, определяющие *действие*, следует стремиться кодировать в формальной спецификации в виде вызовов автоматной моделью выходных воздействий, а утверждения, характеризующие *состояние* – в виде состояний автоматов. В свою очередь, автоматную модель следует проектировать так, чтобы спецификацию можно было таким образом формализовать. Это является хорошим тоном автоматного программирования и позволяет избежать ошибок.

Действительно, в нашем примере требуется сформулировать в терминах автоматной модели утверждение о том, что двери открыты, но выходное воздействие `z1` означает действие: «открыть двери». Вызов этого действия, вообще говоря, не гарантирует, что двери действительно станут открываться. Хотя в автоматной модели на рис. 31 это гарантируется, небольшое добавление функциональности уже может нарушить такое свойство.

Покажем это на примере. Для удобства, вместо формулировки утверждения «двери открыты» перейдем к формулировке утверждения «двери закрыты». При этом опять, как и для модели на рис. 31, возможны два способа определить это утверждение в терминах автоматной модели, используя `z2` или `s1`. Усложним модель. Пусть теперь, если при закрытии двери встречают препятствие, они снова открываются (как и происходит в обычных лифтах). Для этого определим

новое событие: e3, которое происходит тогда, когда на двигателях дверей возникает чрезмерная нагрузка из-за препятствия, и e2 – событие, происходящее, когда двери зафиксировались в одном из положений «открыты» или «закрыты». Управляющий автомат для новой модели приведен на рис. 32.

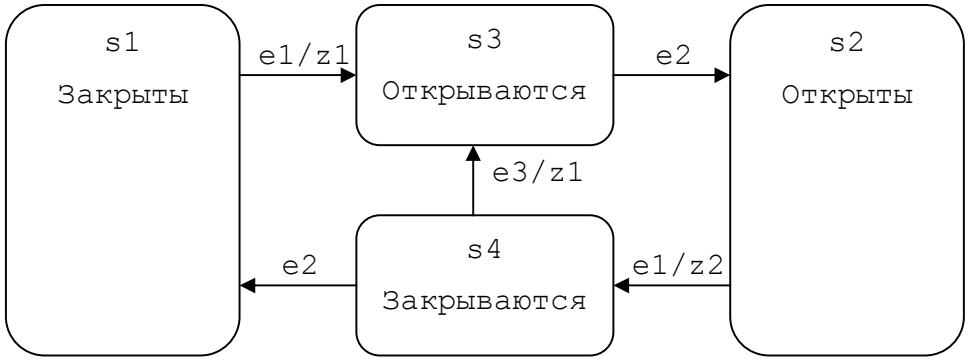


Рис. 32. Автомат, управляющий дверьми лифта с защитой от защемления

В этом автомате добавлены новые состояния, соответствующие состояниям дверей, когда они находятся в процессе открытия или закрытия. Теперь для формулировки утверждения «двери закрыты» возможно использование только предиката s1. Действительно, вызов выходного воздействия z2 уже не гарантирует, что двери действительно закроются. Этим примером объясняется, почему для словесных утверждений, описывающих состояние, следует использовать в формуле именно предикаты, определяющие состояния автоматов. Для словесных утверждений, описывающих действие, в свою очередь, можно использовать как предикаты выходных воздействий, так и формулировать их в виде переходов автоматов из одних состояний в другие.

Таким образом, если некоторое элементарное утверждение фигурирует в словесной спецификации модели, то часто имеет смысл выделить отдельное состояние, в котором это утверждение выполняется. Если отсутствуют отдельные состояния, в котором выполняется некоторое элементарное утверждение, то бывает сложно выразить такое утверждение. Например, если бы требовалось выразить утверждение «двери закрыты» в автомате на рис. 32, не используя состояния автомата, то пришлось бы сформулировать темпоральное утверждение вида «было вызвано воздействие z2, после этого не происходило события e3, но произошло событие e2». Такое выражение оказывается чрезвычайно неудобным: оно состоит из трех элементарных утверждений о

событиях и вызываемых воздействиях, и, что еще хуже, уже в себе самом содержит темпоральную зависимость. Выражение «автомат находится в состоянии `s1`» гораздо более просто, и поэтому проще верифицируются.

Итак, для автомата, изображенного на рис. 32, утверждение о том, что если при открытых дверях была нажата кнопка открытия/закрытия дверей, то двери закроются, можно преобразовать в формулу в темпоральной логике *LTL* следующим образом:

$$(s2 \ \& \ e1) \rightarrow F \ s1.$$

Однако, переводя словесную спецификацию в термины состояний автомата, невольно делается предположение о том, что состояния автомата корректно отображают внешние состояния системы. В рассмотренном примере это означает, что если управляющий автомат находится в состоянии `s1`, то двери действительно должны быть закрыты, а кроме того это должно быть *единственное* состояние, в котором двери оказываются закрыты. Вообще говоря, это может быть не так. Выполнимость такого предположения зависит от проектирования управляющей автоматной модели. Удачным является проектирование, когда некоторое ключевое утверждение о модели действительно выполняется в одном состоянии и только в нем. В таком случае достаточно просто перенести словесную спецификацию в формулу темпоральной логики. Однако при этом не стоит забывать, что корректность переноса основывается на сделанном предположении относительно зависимости между состояниями управляющего автомата и элементарными утверждениями, фигурирующими в словесной спецификации. Для того, чтобы доказать выполнение этого предположения можно провести отдельную верификацию на модели, доказывающую его.

Для приведенного примера управления дверьми лифта это означает следующее. Можно верифицировать утверждение о том, что двери лифта действительно закрыты тогда и только тогда, когда управляющий автомат находится в состоянии `s1`. В модели дверей лифта они закрываются после того, как было вызвано действие `z2`, после которого последовало событие `e2` (двери зафиксировались в закрытом состоянии). Двери перестают быть закрытыми после вызова выходного воздействия `z1`. Сформулируем это в виде формулы темпоральной логики *LTL*. Представим формулу в виде конъюнкции двух формул: первая будет описывать вхождение в состояние `s1`, а вторая – выход из нее. Тогда первая часть будет выглядеть следующим образом:

```
( z2 & (!z1 U e2) ) -> ( (e2 -> s1) U (e2 & s1) )
```

Левая часть импликации означает, что было вызвано воздействие `z2`, а затем произошло событие `e2`, причем между ними не было вызовов к воздействию `z1`. Правая часть импликации означает, что когда произойдет первое событие `e2`, автомат окажется в состоянии `s1`. Здесь важно поставить ограничение на проверку только первого возникновения события `e2` (первого при условии левой части — первого после вызова `z2`), поскольку `e2` может возникать и при последующем открытии двери.

Вторую часть конъюнкции запишем в виде:

```
z1 -> !s1 W ( z2 & (!z1 U e2) ).
```

Это означает, что когда было вызвано воздействие `z1` (двери стали открываться), автомат не попадет в состояние `s1` («Закрыты») до тех пор, пока не будет выполнено условие закрытия дверей, такое же, какое использовалось в первой части конъюнкции. Заметим, что здесь используется дополнительный оператор темпоральной логики *LTL*: `W` – «Weak Until». В отличие от оператора `U`, оператор `W` не требует, чтобы его правая часть когда-либо выполнилась. Действительно, условия закрытия дверей могут никогда и не выполниться, если каждый раз при попытке закрыться, двери будут встречать препятствие.

Конъюнкция двух приведенных выше формул создаст формулу, верификация которой позволит убедиться, что двери лифта закрыты тогда и только тогда, когда управляющий автомат, изображенный на рис. 32, находится в состоянии `s1` («Закрыты»). После этого можно уверенно интерпретировать словесное утверждение спецификации о том, что «двери закрыты» как утверждение об управляющем автомате «автомат находится в состоянии `s1`».

Приведенное доказательство кажется достаточно громоздким. Поэтому обычно обходятся без него и определяют на интуитивном уровне, как трактовать то или иное утверждение в терминах управляющего автомата. Однако стоит помнить, что при этом возможно возникновение ошибки.

2.3. Преобразование требований к автоматным моделям в требования к модели Крипке

Перейдем теперь к решению второго этапа задачи формулировки спецификации: перенос формальной спецификации автоматной модели в термины построенной по ней модели Крипке. Реализация этого этапа сильно зависит от метода, с помощью которого строится модель Крипке. Поэтому рассмотрим решение этой подзадачи на примере.

В этом примере будет использоваться уже приведенный ранее автомат, управляющий дверьми лифта, а для построения модели Крипке — метод расщепления автомата по элементарным состояниям. В данном методе каждый переход в автомате разбивается на цепочку переходов так, что в ходе каждого промежуточного перехода происходит лишь одно элементарное действие автомата. В полученном автомате весьма просто переносить элементарные утверждения об исходном автомате в термины модели Крипке. Действительно, каждое элементарное утверждение характеризует элементарное состояние автомата, а для каждого элементарного состояния автомата существует соответствующее состояние в модели Крипке. Таким образом, элементарное утверждение естественным образом переносится в состояние построенной модели Крипке.

Продемонстрируем изложенное на модели. Для этого разобьем автомат на рис. 32 по описанной схеме. Полученный автомат, являющийся в то же время моделью Крипке, изображен на рис. 33.

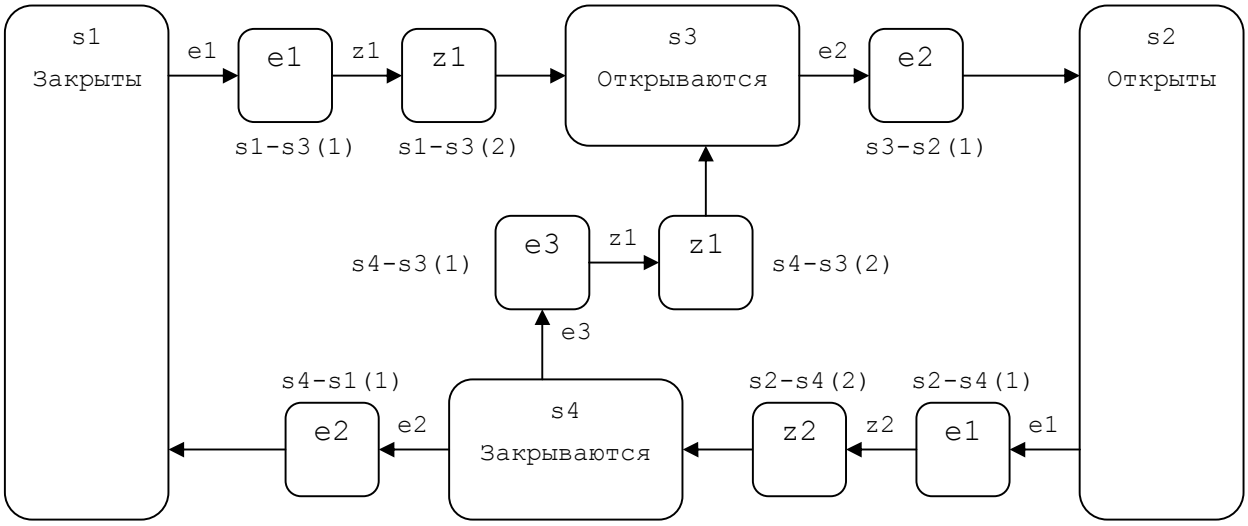


Рис. 33. Модель Крипке для автомата, управляющего дверьми лифта

В этом автомате промежуточные состояния для перехода кодируются следующим образом:

<начальное состояние>-<конечное состояние>(<порядковый номер>)

Теперь приведем утверждение, которое требуется верифицировать. Оно звучит так: «если при открытых дверях лифта была нажата кнопка, то двери закроются». При выполнении первого этапа (перевода словесного утверждения на язык темпоральной логики) оно записывается следующим образом:

$$(s2 \ \& \ e1) \rightarrow F \ s1.$$

Предикаты, характеризующие состояния автомата, остаются неизменными в модели Крипке: предикат `s2` означает, что автомат находится в состоянии `s2`, или, что то же самое, построенная модель Крипке находится в состоянии `s2`.

Предикаты, характеризующие происходящие события или производимые входные или выходные воздействия, в построенной модели Крипке характеризуются соответствующим промежуточным состоянием. Например, учитывается, что указанное утверждение принимает следующий вид:

$$s2\text{-}s4(1) \rightarrow F \ s1.$$

Заметим, что конъюнкция `(s2 & e1)` была заменена на одно состояние, поскольку это единственное состояние, в которое можно попасть по событию `e1` из состояния `s2`. Также заметим, что при данном способе построения модели Крипке, все элементарные утверждения превращаются в утверждения о состояниях в построенной модели Крипке.

Верификация полученной формулы показывает, что формула не выполняется: существует цикл в модели Крипке, проходя по которому бесконечно долго состояние `s1` никогда не будет достигнуто (это возникает в случае, когда препятствие не позволяет дверям закрыться).

Выше была рассмотрена процедура преобразования требований к реактивной системе в термины автоматной модели, а затем в термины построенной модели Крипке на примере автоматной модели, состоящей из одного автомата. Однако возникает вопрос, как осуществлять такие преобразования, в случае, если автоматная модель состоит из системы взаимодействующих автоматов. Рассмотрим подробно этот вопрос.

Для системы автоматов возникают те же проблемы, что и для одного автомата. Словесные требования используют состояния и действия объектов управления, в то время как необходимо сформулировать требования в терминах глобальных состояний и переходов системы автоматов. Под глобальным состоянием системы понимается набор состояний каждого автомата, входящего в систему. Глобальным переходом системы будем называть процесс обработки одного события,

полученного автоматной моделью (стоит заметить, что в процессе обработки этого события автоматы могут генерировать события и передавать их на обработку вызываемым автоматам).

Возможность и простота переноса словесных требований к автоматной модели в термины системы автоматов зависит от проектирования системы. Хорошо, когда некоторое элементарное утверждение, используемое в спецификации, выполняется лишь в одном состоянии или в фиксированном наборе состояний одного из автоматов системы. В таком случае это элементарное утверждение выполняется лишь в тех глобальных состояниях системы автоматов, в которых требуемый автомат находится в одном из состояний заданного набора. Плохо, когда некоторое элементарное утверждение для любого автомата в каждом состоянии может, как выполняться, так и не выполняться. При этом очень сложно сформулировать такое утверждение в терминах системы автоматов.

В силу специфики автоматного проектирования, обычно в автоматной модели каждое состояние отражает некоторое состояние реактивной системы и, таким образом, формулировка требований спецификации в терминах системы автоматов не должна представлять особого труда. Однако стоит помнить, что в результате ошибок проектирования возможна неверная формулировка требований. Это происходит в том случае, например, когда разработчик уверен, что некоторое утверждение выполняется только в одном состоянии одного из автоматов, в то время, как на самом деле это не так. В таком случае словесная спецификация и верифицируемая спецификация оказываются разными. При этом возможно возникновение ситуации, когда верифицируемая спецификация выполняется на автоматной модели, однако на самом деле реактивная система работает неправильно, поскольку верифицируемая спецификация оперирует состояниями и переходами автоматов, тогда как словесная спецификация описывает события и ответные действия объектов управления.

Для иллюстрации изложенного, приведем пример автоматной модели, состоящей из двух автоматов. Выше уже был построен автомат для управления дверьми лифта, построим теперь автомат, который будет управлять движением лифта. Два автомата будут связаны, как изображено на рис. 34.

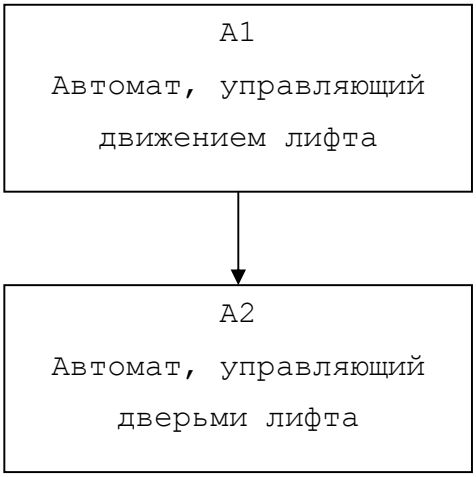


Рис. 34. Связь автоматов, управляющих лифтом

Автомат A1 управляет движением лифта. При нажатии кнопки с номером этажа, этот автомат передает в автомат A2 команду на закрытие дверей, и после их закрытия начинает движение. Когда достигнут требуемый этаж, лифт останавливается и в автомат A2 передается команда открытия дверей. Граф переходов автомата A1 изображена на рис. 4.

В автомате A1 использованы следующие обозначения:

- A2 – автомат, управляющий дверьми лифта (изображен на рис. 32);
- A2.e1 – вызов в автомате A2 обработки события e1;
- A2.s1 – условие, выполняющееся тогда, когда автомат A2 находится в состоянии s1;
- e4 – событие, происходящее, когда была нажата кнопка этажа назначения лифта;
- e5 – событие, происходящее, когда была нажата кнопка «Отмена»;
- e6 – 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.

Автомат A1 работает следующим образом. Он начинает работу в состоянии s1 (Остановлен). Когда пользователь нажимает кнопку этажа (событие e4), если двери закрыты (A2.s1), то в зависимости от того, указанный этаж ниже или выше текущего, автомат запускает двигатель лифта для движения вниз (z3) или вверх (z4) соответственно, переходя в состояние s2 (Движется). Если лифт уже находится на указанном этаже, автомат остается в состоянии s1. Если же при нажатии кнопки этажа двери не были закрыты, то автомату A2 передается на обработку событие e1. При этом возможны два варианта. Автомат A2 находился в состоянии s2 (Открыты), тогда обработка события e1 означает команду закрыться (состояние s3). Если автомат находился в процессе перехода между открытым и закрытым состояниями, то рано или поздно будет порождено событие e2. При этом если двери все еще не закрыты, автомат A1 повторит команду закрыться автомату A2. Однако двери могут никогда не закрыться, если есть препятствие. Для выхода из этого цикла используется событие e5 (кнопка «Отмена»). В случае же удачного закрытия дверей лифт начинает движение и автомат A1 переходит в состояние s2. Во время движения при достижении требуемого этажа происходит событие e6, и если достигнут требуемый этаж, лифт останавливается (z5) и подает команду автомату A2 на открытие дверей. При этом осуществляется переход в состояние s1 (Остановлен). Также при движении лифта событие e1 (кнопка открытия/закрытия дверей) перехватывается автоматом A1 и не передается в A2. За счет этого блокируется открытие дверей пользователем при движении. В остальных состояниях автомата A1 событие e1 не перехватывается, а следовательно автоматически передается вызываемому автомату A2.

На построенной автоматной модели лифта проверим следующее утверждение: лифт движется только при закрытых дверях.

Судя по названиям состояний автомата A1 при проектировании подразумевалось, что лифт движется только в состоянии s2. Проверим это. Лифт начинает движение только при выполнении выходного воздействия z3 или z4 и останавливается только при выполнении z5. Из рассмотрения схемы автомата (рис. 4), действительно, любой переход, ведущий в состояние s2, содержит действие z3 или z4 и не содержит действия z5. Из этого следует, что всегда, когда автомат попадает в состояние s2, он движется. При этом отсутствуют переходы в другие состояния, содержащие действия z3 или z4. В то же время, единственный переход, ведущий из состояния s2 в другое состояние, содержит действие z5. Это означает, что при выходе автомата из состояния s2 лифт

останавливается. Таким образом, лифт движется тогда и только тогда, когда автомат A1 находится в состоянии s2. При этом темпоральная формула, проверяющая описанное свойство автомата A1, имеет следующий вид:

```
[ (z3 | z4) → !(s1 | s3) U s2 ] & [ z5 → !s2 U (s1 | s3) ].
```

Первая часть конъюнкции определяет переход в состояние s2 при запуске двигателя лифта, а вторая — выход из этого состояния при остановке.

Что касается закрытых дверей лифта, то выше уже приводились формулы, доказывающие, что двери лифта закрыты тогда и только тогда, когда автомат A2 (рис. 32) находится в состоянии s1 (Закрыты). Вообще говоря, это свойство доказывалось только для одного автомата, и оно может нарушиться в системе автоматов. Однако системы автоматов специально используются для того, чтобы разделять ответственность за разные части реактивной системы между автоматами. При этом следует минимизировать зависимость между автоматами. Таким образом, хорошо, когда объект управляется лишь одним автоматом. Именно так происходит в построенной модели лифта: выходные воздействия z1 (начать открытие дверей) и z2 (начать закрытие дверей) — действия, выполняемые двигателем дверей лифта, вызываются только в автомате A2, а воздействия z3, z4 и z5, выполняемые двигателем лифта, выполняются только в автомате A1. Поэтому автомат A1 не может нарушить то свойство автомата A2, что двери закрыты лишь в состоянии s1, или, наоборот, автомат A2 не может вызвать или остановить движение лифта.

Итак, движение лифта равносильно тому, что автомат A1 находится в состоянии s2, а закрытость дверей равносильна тому, что автомат A2 находится в состоянии s1. Тогда утверждение «лифт движется только при закрытых дверях» преобразуется в следующую темпоральную формулу в терминах автоматной модели:

```
(A1.s2 → A2.s1).
```

Заметим, что эта формула выполняется на модели, поскольку любой переход в состояние s2 автомата A1 содержит условие A2.s1, и следовательно, автомат A1 переходит в состояние s2 только тогда, когда автомат A2 находится в состоянии s1. При этом автомат A2 может выйти из состояния s1 только при получении события e1, но он не получает этого события, пока лифт находится в движении (автомат A1 находится в s1).

Выполнение второй подзадачи (преобразование полученной формулы в термины модели Крипке) зависит от того, какой метод используется для построения модели Крипке из автоматной

модели, и решение этой подзадачи обычно не составляет большого труда. Рассмотрим, например, способ построения модели Крипке путем построения моделей Крипке для каждого автомата системы автоматов, а затем их перемножения. Элементарные утверждения в полученной формуле описывают состояния отдельных автоматов, события и воздействия. Все эти утверждения переносятся в модели Крипке соответствующих автоматов так же, как это делалось для одного автомата, в результате получается набор состояний, характеризующий состояние перемножения, а следовательно, состояние итоговой модели Крипке. Таким образом, любое утверждение из спецификации для системы автоматов легко переносится в утверждение о состоянии модели Крипке. В результате этого получается темпоральная формула в терминах модели Крипке, которая может использоваться для верификации.

Таким образом, можно утверждать, что преобразование спецификации реактивной системы в термины автоматной модели, а затем в термины модели Крипке является достаточно простым процессом, который, однако, требует внимательности, как для моделей с одним автоматом, так и для систем автоматов. При этом первый этап является творческим (выполняется вручную), а второй — можно выполнить автоматически. Реализация второго этапа зависит от стратегии построения модели Крипке из автоматной модели.

2.4. Методы преобразования контрпримеров из модели Крипке в автоматные модели управляющих программ

После того, как отработала программа-верификатор, необходимо определить выполнимость формул спецификации на определенных участках модели Крипке. Среди этих участков могут быть состояния, события, выходные воздействия. Будем считать, что недостаточно лишь знать ответ, верна или неверна формула на некотором участке. Требуется, чтобы сценарий, предъявленный программой, был представлен в исходном автомате. Сценарий для любой подформулы спецификации представляет собой бесконечный путь в модели Крипке, иллюстрирующий справедливость или ошибочность данной подформулы. Изображать этот путь следует как конечный.

Что касается «переноса» пути из модели Крипке в автомат, то данная операция (скажем, для редуцированной схемы) выполняется однозначно. Действительно, состояния модели, содержащие атомарное предложение *InState*, однозначно преобразуются в соответствующие им состояния автомата. Путь между любыми двумя соседними состояниями проходит ровно через одно состояние-событие, из атомарных предложений которого можно узнать, какое событие ведет по данному пути из

исходного состояния, а также значения существенных и список несущественных входных переменных в момент, когда произошло это событие. Эта информация однозначно определяет направление, вдоль которого строится путь в исходном автомате Мили. Если же путь (или его участок) начинается не в состоянии *InState*, то обратная трассировка пути позволяет узнать состояние *InState*, предшествующее текущему, и всю необходимую информацию относительно того, как попасть в текущее состояние.

Рассмотрим пример для автомата *ARemote*. Пусть для состояния 3 выполняется верификация формулы $\neg E[\neg(Y=6)U(Y=1)]$, которая трактуется следующим образом: в состояние 1 нельзя попасть, минуя состояние 6. Эта формула в состоянии 3 не выполняется. Верификатор сгенерировал (кратчайший и единственный в данном случае) контрпример, который на рис. 35 выделен серым цветом. Это конечный путь, любое продолжение которого удовлетворяет отрицанию указанной формулы.

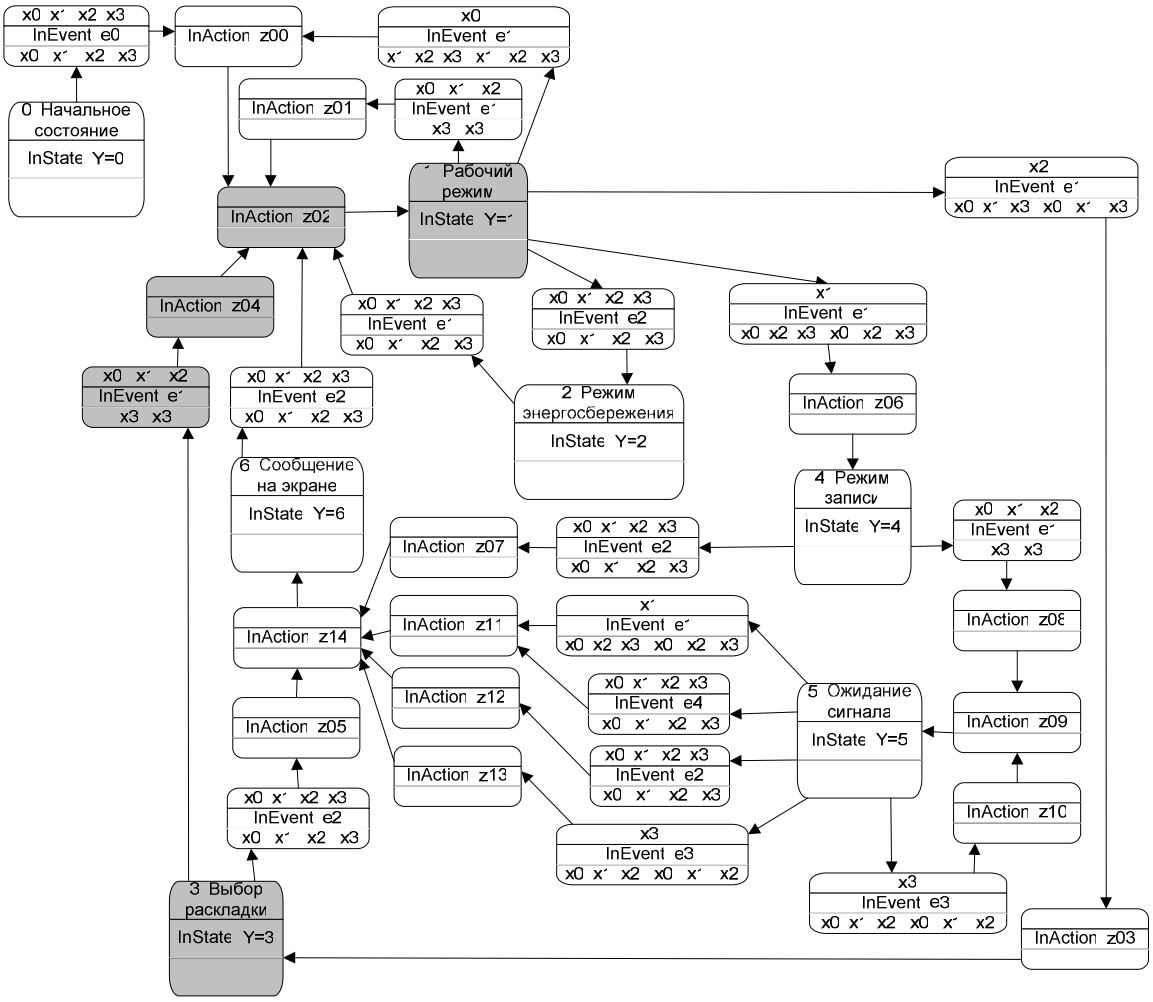


Рис. 35. Контрпример. Путь в модели Крипке автоматизированного объекта

Этот же путь, но представленный в исходном автомате, приведен на рис. 36.

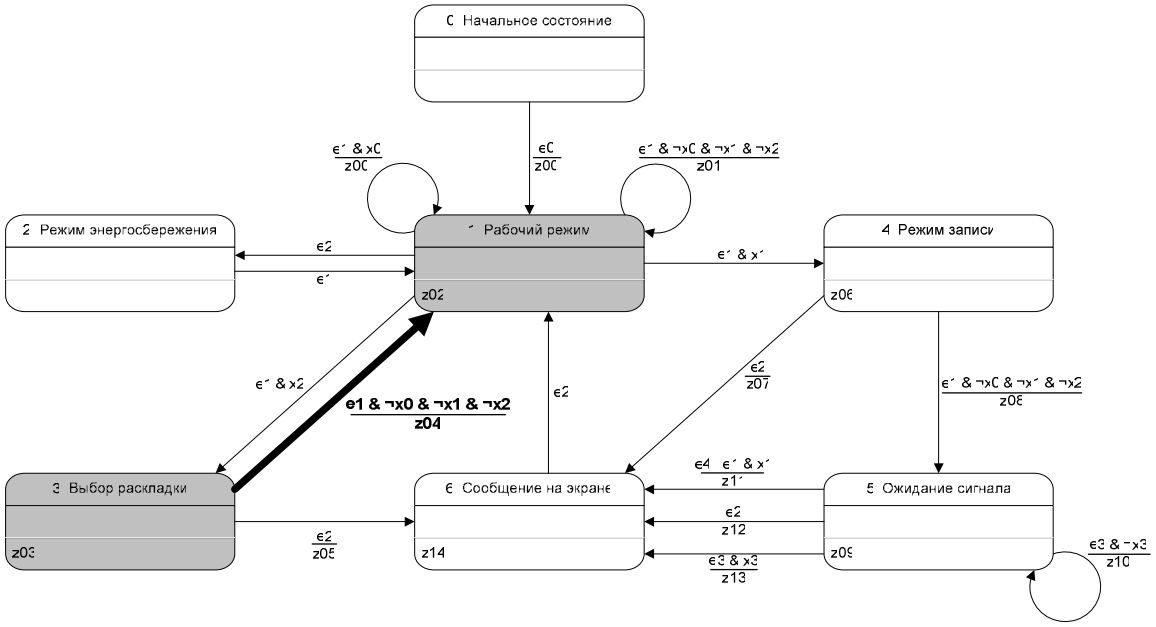


Рис. 36. Контрпример. Путь в автомате Мили

В случае если при моделировании выполнялась композиция автоматов/моделей Крипке, то независимая нумерация их состояний позволит для каждого перехода в пути, представленном в окончательной модели, однозначно решить вопрос о том, в какой именно индивидуальной компоненте системы взаимодействующих автоматов произошел переход. Это, опять же, дает возможность отобразить путь на модели в путь на исходном автомате.

Таким образом, при правильно организованной работе с верификатором, в случае невыполнения некоторого свойства для заданной автоматной модели, в качестве выходных данных пользователь получает путь в автомате, проход по которому приводит к невыполнению данного свойства. Эта информация позволяет локализовать ошибку в автомате и исправить ее. После этого следует заново выполнить верификацию исследуемого свойства для измененного автомата.

ЗАКЛЮЧЕНИЕ

В результате исследований, выполненных на первом этапе работ по контракту, был проведен анализ предметной области и существующих методов верификации автоматных моделей управляющих программ. При этом были сформированы требования, которым должны удовлетворять такие методы.

Были рассмотрены базовые методы верификации на моделях, позволяющие верифицировать автоматные программы. При этом была рассмотрена структура модели Крипке, специфичная для рассматриваемой предметной области и методы построения модели Крипке по автоматной модели.

Также были рассмотрены методы описания требований к автоматным программам в терминах темпоральной логики, и методы их преобразования в требования в терминах модели Крипке. Кроме того, были рассмотрены методы преобразования контрпримеров из модели Крипке в автоматную модель. Это позволило показать, что автоматные программы могут быть эффективно верифицированы.

В первой главе сформулированы требования к методам верификации автоматных программ. Отметим, что требования можно разделить на требования к верификации отдельных автоматов и требования к верификации систем автоматов. При этом разрабатываемые методы должны позволить верифицировать системы, содержащие порядка 10 автоматов, каждый из которых содержит до 10 состояний.

На основе исследований, выполненных во второй главе, можно утверждать, что спецификацию автоматной программы целесообразно задавать в терминах темпоральной логики для автоматной модели. При этом такая спецификация может быть легко преобразована в спецификацию для соответствующей модели Крипке. В случае нахождения контрпримера в терминах модели Крипке, для автоматных моделей он может быть сформулирован в терминах состояний и переходов. Таким образом, открывается возможность проведения верификации автоматных программ в автоматных терминах.

При выполнении работы по первому этапу авторами были проанализированы все доступные источники в области верификации автоматных моделей, и на их основе были получены решения всех задач, поставленных в техническом задании на проведение этого этапа работы.

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

ИСТОЧНИКИ

1. *Ахо А., Ульман Д., Хопкрофт Д.* Структуры данных и алгоритмы. М.: Вильямс. 2000. 384 с.
2. *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.
3. *Вельдер С. Э., Шалыто А. А.* О верификации автоматных программ на основе метода Model Checking // Информационно-управляющие системы. 2007. № 3, с. 27−38.
4. *Кларк Э., Грамберг О., Пелед Д.* Верификация моделей программ: Model Checking. М.: МЦНМО. 2002. 416 с.
5. *Хопкрофт Д., Мотвани Р., Ульман Д.* Введение в теорию автоматов, языков и вычислений. М.: Вильямс, 2002.
6. *Шалыто А. А. Туккель Н. И.* Проектирование программного обеспечения системы управления дизель-генераторами на основе автоматного подхода //Системы управления и обработки информации. 2002. Вып. 5, с. 66−82.
7. *Amla N., Du X., Kuehlmann A., Kurshan R. P., McMillan K. L.* An analysis of *SAT*-based model checking techniques in an industrial environment. CHARME. 2005, pp. 254−268.
8. *Amla N., McMillan K.* Automatic abstraction without counterexamples // TACAS, 2003.
9. *Biere A., Cimatti A., Clarke E., Zhu Y.* Symbolic model checking without BDDs // Lecture Notes in Computer Science. 1999. V. 1579, pp. 193−207.
10. *Biere A., Gupta A., Prasad M.* A survey of recent advances in *SAT*-based formal verification / STTT, 2005.
11. *Brayant R.* The complexity of propositional linear temporal logics //Journal of the ACM. 32(3), pp. 733–749, July 1985.
12. *Bryant R. E.* Graph-based algorithms for boolean function manipulation // IEEE Transactions on Computers. 1986. V. 35, I. 8, pp. 667−691.
13. *Burch J. R., Clarke E. M., Dill D. L., Hwang L. J., McMillan K. L.* Symbolic model checking: 1020 states and beyond // Information and Computation. 1990. V. 98, I. 2, pp. 142−170.
14. *Burch J. R., Clarke E. M., Dill D. L., Long D. E., McMillan K. L.* Symbolic model checking for sequential circuit verification // IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 1994. V. 13, I. 14, pp. 401−424.

15. *Burch J. R., Clarke E. M., Long D. E.* Symbolic model checking with partitioned transition relations / International Conference on Very Large Scale Integration. Edinburgh: North-Holland. 1991, pp. 49−58.
16. *Burstall R. M.* Program proving as hand simulation with a little induction /IFIP Congress. 1974. North Holland, pp. 308−312.
17. *Chauhan P., Clarke E., Kukula J., Sapra S., Veith H., Wang D.* Automated abstraction refinement for model checking large state spaces using *SAT* based conflict analysis / FMCAD '02: Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design. 2002. London: Springer-Verlag, pp. 33−51.
18. *Clarke E. M., Draghicescu I. A.* Expressibility results for linear time and branching time logics / In Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency. London: Springer-Verlag. 1989, p. 428−437.
19. *Clarke E. M., Emerson E. A.* Design and synthesis of synchronization skeletons using branching time temporal logic // Logic of Programs. 1981. V. 131, pp. 52−71.
20. *Clarke E. M., Emerson E. A., Sistla A. P.* Automatic verification of finite-state concurrent systems using temporal logic specifications: A Practical Approach / In Proceedings of the 10th Annual ACM Symposium on Principles of Programming Language. Austin. 1983, pp. 117−126.
21. *Clarke E. M., Grumberg O., Hiraishi H., Jha S., Long D., McMillan K. L., Ness, L.* Verification of the Futurebus + cache coherence protocol // Formal Methods In System Design, 1995. V. 6, I. 2, pp. 217−232.
22. *Clarke E. M., Long, D.E., McMillan K.L.* Compositional model checking / Proceedings of the Fourth Annual Symposium on Logic in Computer Science. NJ: IEEE Press. 1989, pp. 353−362.
23. *Courcoubetis C., Vardi M. Y., Wolper P., Yannakakis M.* Memory efficient algorithms for the verification of temporal properties // Formal Methods in System Design. 1992. V. 1, p. 275−288.
24. *Emerson E. A.* Branching time temporal logic and the design of correct concurrent programs. PhD thesis, Harvard University, 1981.
25. *Emerson E. A., Halpern J. Y.* “Sometimes” and “not never” revisited: On branching time versus linear time // Journal of the ACM. 1984. V. 33, pp. 151−178.
26. *Ganai M., Krohm F., Kuehlmann A., Paruthi V.* Robust boolean reasoning for equivalence checking and functional property verification // IEEE TCAD, Vol. 21(12), 2002, pp. 1377−1394.

27. *Gerth R., Peled D., Vardi M. Y., Wolper P.* Simple on-the-fly automatic verification of linear temporal logic / In Proceedings of 15th Workshop Protocol Specification, Testing, and Verification. Warsaw: Chapman & Hall. 1995, pp. 3−18.
28. *Godefroid P.* Partial-order methods for the verification of concurrent systems – an approach to the state-explosion problem. Lecture Notes in Computer Science. V.1032. Springer-Verlag, January 1996.
29. *Godefroid P.* Using partial orders to improve automatic verification methods / Proc. 2nd Workshop on Computer Aided Verification. Lecture Notes in Computer Science. V. 531. pp. 176−185, Rutgers, June 1990, Springer-Verlag. Extended version in ACM/AMS DIMACS Series, V. 3, pp. 321−340, 1991.
30. *Godefroid P., Pirottin D.* Refining dependencies improves partial-order verification methods /Proc. 5nd Conference on Computer Aided Verification. Lecture Notes in Computer Science. V.697. Springer-Verlag. 1993, pp. 438−449.
31. *Godefroid P., Wolper P.* Using partial orders for efficient verification of deadlock freedom and safety properties / Proc. 3rd Workshop on Computer Aided Verification. Lecture Notes in Computer Science. V.575. 1991, pp. 332−342.
32. *Goldberg E., Novikov Y.* Berkmin: A fast and robust *SAT*-solver /In DATA, 2002.
33. *Holzmann G. J.* The Model Checker *SPIN* // IEEE Transactions on software engineering. 1997. V. 23, I. 5.
34. *Holzmann G. J., Peled D.* An improvement in formal verification /Proc. FORTE'94. Bern. 1994, pp. 177−191.
35. *Kroger P.* LAR: A logic of algorithmic reasoning //Acta Informatica. 1977. V. 8, I. 3, pp. 243−266.
36. *Kuehlmann A.* Dynamic transition relation simplification for bounded property checking. // ICCAD, 2004.
37. *Lamport L.* “Sometimes” is sometimes “not never” /Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. NY. ACM Press. 1980, pp. 174−185.
38. *Lamport L.* “Sometimes” is sometimes “Not Never” // Proc. 7th ACM Symp. Principles of Programming Languages (POPL'80). 1980, pp. 174−185.
39. *Lind-Nielsen J., Andersen H. R. Stepwise CTL* Model Checking of State/Event Systems / Proceedings of the 11th International Conference on Computer Aided Verification. London: Springer-Verlag. 1999, pp. 316−327.

40. *Madigan C. F., Malik S., Moskewicz M. W., Zhang L., Zhao Y.* Chaff: engineering an efficient *SAT* solver /In DAC, 2001.
41. *Marques-Silva J., Sakallah K.* GRASP: A search algorithm for propositional *SAT*isfiability //IEEE Transactions on Computers. 1999. V. 48. № 5.
42. *McMillan K. L.* Applying *SAT* methods in unbounded symbolic model checking / CAV, 2003.
43. *McMillan K. L.* Interpolation and *SAT*-based model checking / CAV, 2003.
44. *McMillan K. L.* Symbolic model checking: an approach to the state explosion problem. PhD thesis. SCS. Carnegie Mellon University, 1992.
45. *Pnueli A.* The temporal logic of programs /In 18th IEEE Symposium on Foundation of Computer Science. IEEE Computer Society Press. 1977, pp. 46−57.
46. *Quielle J. P. and Sifakis J.* A temporal logic to deal with fairness in transition systems. In FOCS 1982, pp. 217−225
47. *Quielle J. P. and Sifakis J.* Specification and verification of concurrent systems in CESAR / In Proceedings of the 5th International Symposium of Programming. London: Springer-Verlag. 1982, pp. 337−351.
48. *Roux C., Encrenaz E. CTL* may be ambiguous when model checking Moore machines / CHARME, Springer. 2003, pp. 164−169.
49. *Sistla A. P., Clarke E. M.* Complexity of propositional temporal logics // Journal of the ACM. 1986. V. 32, I. 3, pp. 733−749.
50. *Sistla A., Clarke E.* Graph-based Algorithms for Boolean Function Manipulation // IEEE Transactions on Computers. 1986. № 8.
51. *Valmari A.* Error detection by reduced reachability graph generation / Proc. 9th International Conference On Application And Theory Of Petri Nets. Venice. 1988, pp. 95−112.
52. *Valmari A.* Heuristics for lazy state generation speeds up analysis of concurent systems /In Proc. Of the Finnish Artificial Intelligence Symposium STeP-88, V. 2, Helsinki. 1988, pp. 640−650.
53. *Valmari A.* Stubborn attack on state explosion. In Proc. 2nd Workshop on Computer Aided Verification. Lecture Notes in Computer Science. V.531. Springer-Verlag. 1990, pp. 156−165.
54. *Vardi M. Y., Wolper P.* An automata-theoretic approach to automatic program verification / Proceedings of the First Symposium on Logic in Computer Science, Cambridge. 1986. pp. 322–331.

55. *Vardi M. Y., Wolper P.* Automata-theoretic techniques for modal logics of programs // Journal of Computer and System Science. 1986. V. 32(2), pp. 182–221.
56. *Vardi M. Y., Wolper P.* Reasoning about infinite computations // Information and Computation. 1994. V. 115, pp. 1−37.