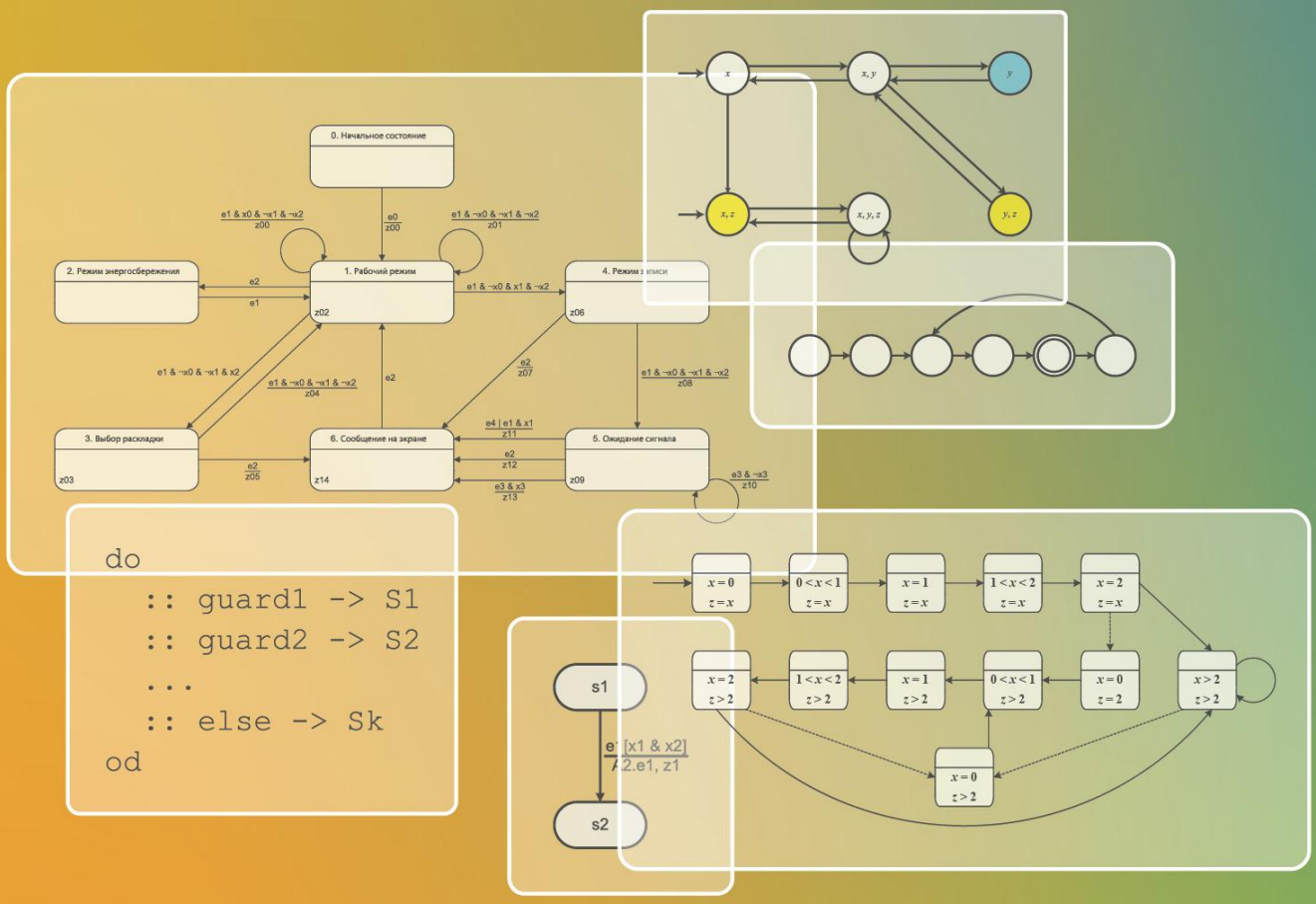
С. Э. Вельдер, М. А. Лукин,
А. А. Шалыто, Б. Р. Яминов

ВЕРИФИКАЦИЯ АВТОМАТНЫХ ПРОГРАММ

Учебное пособие



Санкт-Петербург
2011

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

**С. Э. Вельдер, М. А. Лукин,
А. А. Шалыто, Б. Р. Яминов**

ВЕРИФИКАЦИЯ АВТОМАТНЫХ ПРОГРАММ

Учебное пособие

Санкт-Петербург

2011

Рецензенты:
Мелехин В. Ф., докт. техн. наук, профессор, заведующий кафедрой компьютерных систем и программных технологий Санкт-Петербургского государственного политехнического университета
Сергеев М. Б., докт. техн. наук, профессор, заведующий кафедрой вычислительных систем и сетей Санкт-Петербургского государственного университета аэрокосмического приборостроения

УДК 004.42

Вельдер С. Э., Лукин М. А., Шалыто А. А., Яминов Б. Р. Верификация автоматных программ. СПбГУ ИТМО, 2011. – 242 с.

В учебном пособии рассматриваются вопросы верификации программного обеспечения на основе проверки моделей с использованием различных языков спецификации. Особое внимание уделяется верификации автоматных программ, которые моделируются в виде системы автоматизированных объектов управления и могут быть весьма эффективно верифицированы указанным методом. Математический аппарат и прикладные инструменты данной области позволяют создавать качественное программное обеспечение для ответственных систем и получать надежные подтверждения их правильности. Учебное пособие посвящено концепциям, алгоритмам и инструментам для проверки моделей программ. В нем излагаются теоретические вопросы проверки моделей, вводятся различные спецификационные формализмы и описываются алгоритмы проверки моделей для спецификаций, выраженных в этих формализмах. Алгоритмы проверки моделей демонстрируются на примерах конкретных инструментальных средств.

Материал учебного пособия предназначен для специалистов в области программирования, информатики, вычислительной техники и систем управления, а также студентов и аспирантов, обучающихся по специальностям «Прикладная математика и информатика», «Управление и информатика в технических системах» и «Вычислительные машины, системы, комплексы и сети». Предполагается знакомство читателя с основными понятиями математической логики, дискретной математики, теории графов и теории алгоритмов.

Рекомендовано к печати Ученым Советом университета

В 2009 году Университет стал победителем многоэтапного конкурса, в результате которого определены 12 ведущих университетов России, которым присвоена категория «Национальный исследовательский университет». Министерством образования и науки Российской Федерации была утверждена Программа развития государственного образовательного учреждения высшего профессионального образования «Санкт-Петербургский государственный университет информационных технологий, механики и оптики» на 2009–2018 годы.

Оглавление

Введение

Понятие валидации

В повседневной жизни явно (при использовании компьютеров и интернета) или неявно (при использовании телевизоров, микроволновых печей, мобильных телефонов, автомобилей, общественного транспорта и т. д.) все чаще используются информационные технологии. В 1995 г. было подсчитано, что человек в день взаимодействует с 25 устройствами, обрабатывающими информацию. Известно также, что 20 % стоимости разработки автомобилей, поездов и самолетов приходятся на компьютерные компоненты. Ввиду высокой интеграции информационных технологий во всех приложениях приходится все больше полагаться на надежность программных и аппаратных средств. Естественно полагать, что не должно быть ситуаций, когда телефон неисправен или когда видеомагнитофон непредсказуемо и неверно реагирует на команды, посылаемые с пульта управления. Тем не менее, эти ошибки в определенном смысле малозначимы. Однако ошибки в системах, критичных в отношении безопасности, таких как, например, атомные электростанции или системы управления полетами, неприемлемы. Основная проблема для таких систем состоит в том, что быстро возрастает их сложность и, как следствие, число возможных ошибок.

Даже если отвлечься от аспектов безопасности, ошибки все равно могут быть очень дорогими, особенно если они проявляются в процессе работы системы – после того, как продукт выпущен на рынок. Известно несколько впечатляющих примеров таких негативных последствий. Например, ошибка в команде деления чисел с плавающей запятой в процессоре Intel Pentium причинила ущерб около 500 миллионов долларов. Крупный ущерб был причинен также крушением ракеты Ariane-5, которое, вероятно, произошло вследствие ошибки в программе управления полетом.

Поэтому *валидация систем* (процесс проверки корректности спецификаций, дизайна и продукта) – это деятельность все возрастающей важности. Валидация является способом поддерживать контроль качества систем.

Особую важность имеет обеспечение качества программного обеспечения, так как в современных системах с его помощью реализуется основная функциональность. Современная практика программирования показывает, что системы проверяются в значительной степени людьми (экспертный анализ) и динамическим тестированием. Поддержка этих процессов даже относительно простыми инструментами, не говоря уже о методах и инструментах с серьезной математической базой, в настоящее время недостаточна.

Ввиду возрастания размеров и сложности систем, становится важным обеспечение процесса валидации систем с использованием методов и инструментов, которые облегчают *автоматический* анализ корректности, так как, например, *ручная верификация может быть настолько же ошибочна, как и сама программа*.

Методы валидации систем

Важнейшими методами валидации являются экспертный анализ, тестирование, симуляция, формальная верификация и проверка моделей. Остановимся на двух из них.

Тестирование – эффективный путь проверки того, что заданная *реализация* системы согласуется с абстрактной спецификацией. По своей природе тестирование может быть использовано только после реализации прототипа системы.

Формальная верификация, как противоположность тестированию, основана на математическом доказательстве корректности программ.

Оба этих метода могут поддерживаться и частично поддерживаются инструментами. Например, в области тестирования возрастает интерес к разработке алгоритмов и программ для автоматической генерации и выбора тестов по формальной спецификации системы. Основой формальной верификации являются программы для автоматического доказательства теорем и проверки доказательств, однако даже их использование обычно требует высокой квалификации пользователей.

Проверка моделей

В книге рассматривается другой метод валидации – *проверка моделей* (model checking). Это автоматизированный метод, который для заданной модели поведения системы с конечным числом состояний и логического свойства, записанного в подходящем логическом формализме (обычно в темпоральной логике), проверяет справедливость этого свойства в данной модели. Проверка моделей может применяться для верификации как аппаратуры, так и программ. Успешность ряда проектов с использованием этого метода верификации увеличивает интерес к нему. Например, корпорация Intel открыла несколько исследовательских лабораторий верификации новых микросхем. Интересный аспект проверки моделей состоит в том, что она поддерживает *частичную* верификацию: для системы может быть проверена частичная спецификация при рассмотрении только некоторого подмножества всех требований.

Предмет рассмотрения книги

Книга посвящена концепциям, алгоритмам и инструментам для проверки моделей программ.

Идеи проверки моделей имеют такой математический фундамент как логика, теория автоматов, структуры данных, алгоритмы на графах.

Книга имеет следующую структуру. Первая глава посвящена общим вопросам валидации программных систем. Рассматривается постановка задачи валидации и определяется роль валидации в производстве программного обеспечения. Также выделяются различные виды валидации, приводятся основные понятия и определения для каждого из них.

Во второй главе излагаются теоретические вопросы проверки моделей, вводятся формализмы различных *темпоральных логик* и описываются алгоритмы проверки моделей для спецификаций, выраженных в этих темпоральных логиках. Выделяются два типа моделей – *модели Крипке* и *сети Петри*. Также представлены некоторые математические результаты, связанные со свойствами рассматриваемых моделей.

Описываются три типа проверки моделей: для линейной и ветвящейся темпоральных логик, а также для темпоральной логики реального времени. Первые два из них позволяют отразить функциональные или качественные аспекты системы, а третий – проводить также и некоторый количественный анализ.

В третьей главе демонстрируются алгоритмы проверки моделей на примерах конкретных инструментальных средств. Приводятся краткое описание синтаксиса этих инструментальных средств и примеры реализации коммуникационных алгоритмов. Там же приведены результаты проверки моделей этих алгоритмов.

Изложение общих вопросов валидации, математического аппарата и примеров верификации моделей в первых трех главах следует курсу лекций П. Катоена [1].

Четвертая глава книги посвящена проверке моделей программ, построенных на основе автоматного подхода [2]. Основной особенностью таких программ является то, что модель поведения программы, представленная в виде *системы управляющих конечных автоматов*, по которой генерируется код, строится разработчиком уже на этапе проектирования, а не по готовой программе, как это предлагается делать при традиционном применении метода проверки моделей. При этом программы становятся в один ряд с другими инженерными разработками (например, самолетами и автомобилями), для которых сначала создаются модели, а в дальнейшем на их основе изготавливаются изделия, а не наоборот.

Автоматный подход позволяет отделить управляющие (качественные) состояния программы от ее вычислительных (количественных) состояний. Это позволяет резко уменьшить размер используемых моделей и повысить эффективность их построения и проверки. Основным достоинством верификации автоматных программ является возможность эффективного преобразования спецификации программы в формат, подходящий для алгоритмической проверки, и возможность автоматического обратного преобразования результата при обнаружении ошибки. В книге приводятся несколько методов прямого преобразования, предназначенных для работы с различными логическими формализмами, описываются инструментальные средства проверки моделей для автоматных программ и приводятся примеры использования этих средств для конкретных автоматных моделей. Автоматные программы, в отличие от традиционных, могут быть названы программами, ориентированными на верификацию.

Содержание четвертой главы основано на результатах работ по государственному контракту «Разработка технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода», выполнявшемуся в рамках Федеральной целевой программы «Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007–2012 годы» по мероприятию «Проведение проблемно-ориентированных поисковых исследований и создание научно-технического задела по перспективным технологиям в области информационно-телекоммуникационных систем». Работы выполнялись в Санкт-Петербургском государственном университете информационных технологий, механики и оптики (СПбГУ ИТМО).

Содержание книги подтверждает известное положение программной инженерии: «то, что не специфицировано формально, не может быть проверено, а то, что не может быть проверено, не может быть безошибочно».

Авторы благодарны В. Н. Васильеву и В. Г. Парфенову за многолетнюю поддержку и помощь.

Глава 1. Валидация систем

1.1. Задачи валидации систем

Введение и мотивация

Системы, которые, так или иначе, связаны с обработкой информации, все чаще предоставляют пользователям критические сервисы. Они

используются в различных обстоятельствах, когда неисправность в программе может привести к пагубным последствиям. Обработка информации играет значительную роль в системах управления такими объектами как, например, атомные электростанции или химические реакторы, где ошибки крайне опасны. Другим распространенным примером систем, критичных к безопасности, являются излучающие устройства в медицине.

Это примеры программных систем, для которых надежность (корректность работы) крайне важна. Ввиду расширения функциональности критических приложений, справедливо сказать, что *надежность обработки информации является ключевым фактором в процессе разработки таких систем*.

Как обеспечивается надежность программных систем? Обычно разработка начинается с анализа требований заказчика, а после прохождения нескольких фаз разработки в конце процесса получается прототип. Процесс выяснения того, что разработанный прототип удовлетворяет требованиям, называется *валидацией систем*. Схематически стратегия валидации изображена на рис. 1.1 [1].

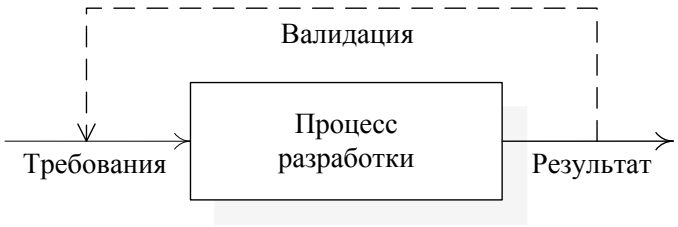


Рис. 1.1. Схематическое изображение апостериорной валидации систем

Какова общая практика решения вопроса, когда проводить валидацию? Ясно, что проверка на ошибки только в конце разработки неприемлема: если ошибка найдена, требуется много усилий для того, чтобы ее исправить, так как весь процесс разработки должна быть пройден вновь, для того чтобы увидеть, где и как ошибка могла возникнуть. Подобные операции обычно являются затратными. Следовательно, целесообразно проводить валидацию на лету – в процессе разработки системы, так как это значительно уменьшает стоимость разработки. Схематически такой процесс изображен на рис. 1.2.

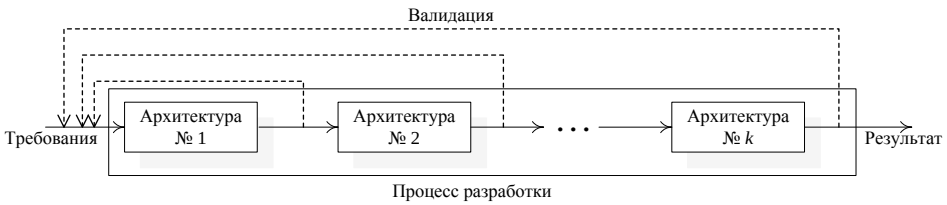


Рис. 1.2. Схематическое изображение валидации систем «на лету»

На практике для того, чтобы убедиться, что конечный результат выполняет то, что предписано, применяются два метода: *экспертный анализ* и *тестирование*. Исследования в области разработки программ показывают, что 80% всех проектов используют экспертный анализ – полностью ручную деятельность, в которой прототип или его часть исследуется командой специалистов, которые не были вовлечены в процесс разработки этой компоненты системы. Тестирование является важным методом валидации для проверки корректности полученных реализаций. Однако обычно тесты генерируются вручную, и инструментальной поддержке уделяется мало внимания.

Валидация систем – это важный шаг в их разработке: в большинстве проектов больше времени и усилий уходит на валидацию, чем на разработку. Ошибки могут быть очень дорогими. Например, ошибка в денверской системе управления багажом задержала открытие нового аэропорта на девять месяцев (ущерб – 1,1 миллиона долларов за день).

Валидация систем как часть процесса разработки

Интервью со специалистами ряда компаний, занимающихся разработкой программного обеспечения, в очередной раз показало, что валидация систем должна интегрироваться в процесс разработки на ранних этапах [1, 3, 4]. Это справедливо, в том числе, и с экономической точки зрения, однако в настоящее время большую часть ошибок находят в процессе тестирования, когда программные модули уже скомпонованы и исследуется вся система.

Формальные методы

Для обеспечения качества сложных программных систем требуется использовать современные методы и инструменты, которые поддерживают процесс валидации. Применяемые в настоящее время методы валидации являются узкоспециализированными и базируются на спецификациях, сформулированных на естественном языке. Поэтому в данной книге рассматриваются подходы к валидации, базирующиеся на *формальных методах*. При использовании таких методов проектирование систем должно быть определено в терминах точных и недвусмысленных спецификаций, которые предоставляют базу для систематического анализа. Перечислим основные подходы к валидации:

- *симуляция*;
- *тестирование*;
- *формальная верификация*;
- *проверка моделей* (*верификация на моделях*).

В дальнейшем будут кратко рассмотрены три первых подхода, а основной темой, как отмечалось выше, является проверка моделей.

Реактивные системы

В данной книге основное внимание уделяется валидации *реактивных систем*. Реактивные системы [5] характеризуются непрерывным взаимодействием с окружением. Они принимают значения входов из своего окружения и, обычно с маленькой задержкой, реагируют на них. Типичными примерами систем этого класса являются операционные системы, системы управления самолетами, автомобилями и технологическими процессами, коммуникационные протоколы и т. д. Например, система управления химическим процессом регулярно принимает управляющие сигналы, такие как температура и давление, в различных точках процесса. На основании этой информации программа может принять решение включить нагревательный элемент, выключить насос и т. д. Если возникла опасная ситуация, например, давление в резервуаре выходит за определенные пределы, управляющая программа должна выполнить определенные действия. Обычно подобные реактивные системы достаточно сложны.

Как отмечалось выше, корректность реактивных систем имеет решающее значение. Для построения корректно работающей реактивной системы требуется четкая методология, в которой выделяются следующие фазы.

- На основе исчерпывающего анализа требований должна быть сформирована их спецификация.
- Концептуальное проектирование позволяет получить абстрактную спецификацию проекта. Эта спецификация может быть проверена на непротиворечивость и ее соответствие требованиям. Данный процесс валидации может быть поддержан формальной верификацией, симуляцией и проверкой моделей – такими процессами, в которых модель абстрактной спецификации проекта (например, система конечных автоматов) может быть исчерпывающим образом проверена.
- Когда заслуживающая доверия спецификация получена, переходят к построению системы, которая реализует абстрактную спецификацию. При этом тестирование является полезным методом проверки реализации на соответствие оригинальным требованиям.

1.2. Симуляция

Симуляция базируется на модели, которая описывает возможное поведение системы. Эта модель в определенном смысле исполнима, при этом программный инструмент (называемый *симулятором*) может определить поведение системы по отношению к некоторым сценариям. Таким способом пользователь получает определенное понимание того, как система реагирует на стимулы. Сценарии могут быть предоставлены пользователем или инструментом, таким как, например, генератор, который строит случайные сценарии. Симуляция, в основном полезна для быстрой, первоначальной оценки качества проекта. Она в меньшей степени подходит для поиска тонких ошибок, так как симулировать все возможные сценарии непрактично (и часто невозможно).

1.3. Тестирование

Как отмечалось выше, широко используемым традиционным путем валидации корректности проекта является *тестирование* [6]. В тестировании система используется в том виде, в каком она реализована (программа, устройство или их комбинация). На ее вход подаются определенные значения входных данных, называемых *тестами*, и исследуется реакция системы. После этого проверяется, действительно ли реакция системы соответствует требуемому выходу. Принципы тестирования почти такие же, как и у симуляции. Их важное различие состоит в том, что тестирование проводится на действующей реализации системы, а симуляция – на модели системы.

Тестирование – это метод валидации, широко используемый на практике, но почти всегда выполняемый на базе неформальных и эвристических методов. Так как тестирование базируется на рассмотрении только небольшого подмножества возможных примеров поведения системы, оно никогда не может быть полным. Э. Дейкстра отметил, что тестирование может показать только наличие ошибок, но не их отсутствие.

Тестирование, однако, может дополнять формальную верификацию и проверку моделей, которые выполняются на математической модели системы, а не на реальной системе. Так как тестирование применяется к реализации, оно полезно, главным образом, в следующих случаях:

- когда корректную модель системы сложно построить;
- когда части системы не могут быть формально промоделированы (например, физические устройства);
- когда модель проприетарна.

Обычно, тестирование – доминирующий метод валидации систем. Он состоит в применении к реализации ряда тестов, которые обычно получают эвристически. В последнее время, однако, возрастает интерес к применению формальных методов в тестировании. Например, в области коммуникационных протоколов этот тип исследований привел к проекту международного стандарта «Формальные методы в тестировании пригодности» [7]. В нем процесс тестирования разделен на несколько фаз:

1. *Генерация тестов*. Абстрактные описания тестов систематически генерируются на основе точного и недвусмысленного задания требуемых свойств в спецификации.
2. *Выбор тестов*. Выбирается множество образцов абстрактных описаний тестов.
3. *Реализация тестов*. Абстрактные описания тестов трансформируются в исполнимые тесты.
4. *Исполнение тестов*. Исполнимые тесты применяются к тестируемой реализации путем выполнения их на тестирующей системе. Рассматриваются и протоколируются результаты исполнения.
5. *Тестовый анализ*. Запротоколированные результаты анализируются с целью определения, удовлетворяют ли они ожидаемым результатам.

Различные фазы тестирования могут пересекаться и на практике часто пересекаются, особенно последние.

Тестирование – это метод, который может быть применен как к прототипам, в форме систематической симуляции, так и к конечным продуктам. Известны два базовых подхода: *тестирование белого ящика*, когда внутренняя структура реализации может наблюдаться и иногда частично управляться (стимулироваться), и *тестирование черного ящика* [8]. Во втором случае может быть проверена только коммуникация между тестируемой системой и окружением, а внутренняя структура «ящика» полностью скрыта от тестера. В практических обстоятельствах тестирование находится где-то между этими двумя крайними случаями и иногда называется *тестированием серого ящика*.

Перечислим основные виды тестов.

1. Модульные тесты (*Unit tests*). Тест пишется на класс, отдельный метод этого класса и т. д. Обычно проверяется один сценарий использования класса, метода или иного модуля.

2. Функциональные тесты. Тестируется не элемент кода, а функциональность. Такие тесты позволяют выявлять структурные ошибки.
3. Приемочные тесты. Проверяется, что программа делает именно то, что хотел заказчик.
4. Стресс-тест. Тестируется работоспособность программы под сильной нагрузкой.
5. Тест на дурака (*monkey test*). Программе подаются случайные входные данные, и проверяются ее работоспособность на таких данных и отказоустойчивость. Также проверяется наличие уязвимостей. Такие тесты особенно важны для серверных приложений.
6. Параллельные тесты. Проверяется, что новая версия работает так же, как старая.
7. Регрессионные тесты. Пишутся после сообщения об ошибке. Тест повторяет сценарий, при котором произошла ошибка.

Разработка через тестирование (Test-driven development)

Одной из технологий разработки программного обеспечения является *разработка через тестирование* (*test-driven development – TDD*) [9]. Отметим, что *TDD* не является технологией *тестирования*. Этот процесс состоит из коротких итераций разработки программы, каждая из которых состоит из следующих этапов:

1. Написание теста.
2. Компиляция теста. Тест не должен компилироваться. Если тест компилируется, то это означает, что создана сущность, которая уже присутствует.
3. Устранение ошибки компиляции.
4. Запуск теста. Тест не должен проходить.
5. Написание кода. Пишется максимально простой код, какой только возможен.
6. Прогон теста. На этот раз тест должен быть пройден.
7. Рефакторинг (если требуется).

Второй и третий этапы выполняются только при создании новых модулей или классов.

Когда программа использует оборудование, доступ к которому затруднен, применяются *мок-объекты* или *моки* [9] (*mocks*). Моки – это автоматически сгенерированные заглушки. Также моки могут

использоваться для программирования сверху вниз [10]. Для создания мок-объектов существуют специальные фреймворки (*RhinoMock*, *NMock*, *JMock* и др.).

Свойства разработки через тестирование:

- подход заставляет разрабатывать программу с точки зрения пользователя;
- он заставляет писать классы, независимые друг от друга;
- тесты являются также и документацией к коду;
- требуется много времени для того, чтобы освоить этот подход на практике.

1.4. Формальная верификация

Дополняющим методом к симуляции и тестированию является строгое *доказательство* того, что система работает корректно. Такая математическая демонстрация корректности системы называется *формальной верификацией* [11]. Базовая идея состоит в том, чтобы построить формальную (математическую) модель исследуемой системы, которая отражает (специфицирует) возможное поведение системы. При этом требования корректности записываются в виде формальной *спецификации требований*, которая отражает желаемое поведение системы. На базе этих двух спецификаций можно формальным доказательством проверить, действительно ли возможное поведение согласуется с желаемым. Поскольку имеет место верификация в математической форме, представление о согласованности может быть точным, и верификация состоит в доказательстве корректности по отношению к этому формальному представлению.

Для формальной верификации требуются:

- Модель системы, обычно содержащая множество состояний, которые хранят информацию о значениях переменных, программных счетчиках и т. п., и отношение переходов, которое описывает, как система переходит из одного состояния в другое.
- Метод спецификации для выражения требований в формальном виде.
- Множество правил доказательства, позволяющих определить, удовлетворяет ли модель сформулированным требованиям.

Для того чтобы более конкретно понять, что имеется в виду, рассмотрим способ, с помощью которого могут быть формально верифицированы последовательные программы.

Верификация последовательных алгоритмов

Этот подход может быть использован для доказательства корректности последовательных алгоритмов [1, 12], таких как быстрая сортировка или вычисление наибольшего общего делителя двух целых чисел. Кратко опишем, как он работает. Он начинается с формализации желаемого поведения с использованием пред- и постусловий на основе формул логики предикатов. Синтаксис этих формул можно определить, например, следующим образом:

$$\varphi ::= p \mid \neg\varphi \mid (\varphi \vee \varphi).$$

Здесь p – базовое предложение (например, «x равно 2»), символ «$\neg$» – отрицание, а символ «$\vee$» – дизъюнкция. Другие логические связки могут быть определены следующим образом: $\varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi)$, $\text{true} = \varphi \vee \neg\varphi$, $\text{false} = \neg\text{true}$ и $\varphi \rightarrow \psi = \neg\varphi \vee \psi$. Для простоты опустим кванторы существования и всеобщности.

Предусловие описывает множество исследуемых стартовых состояний (допустимых значений входов), а *постусловие* – множество желаемых финальных состояний (требуемых значений выходов). Когда пред- и постусловия формализованы, алгоритм кодируется в некотором абстрактном псевдокоде, и по шагам доказывается, что он удовлетворяет спецификации.

Для построения доказательств используется *формальная система* – множество *правил вывода*. Эти правила обычно соответствуют построению программы (алгоритма). Они записываются следующим образом:

$$\{\varphi\}\ S\ \{\psi\}.$$

Здесь φ – предусловие, S – программный оператор, а ψ – постусловие. Тройка $\{\varphi\}\ S\ \{\psi\}$ известна как *тройка Хоара* [13] и названа в честь одного из пионеров в области формальной верификации компьютерных программ. Существуют две интерпретации троек Хоара в зависимости от того, частичная или тотальная корректность рассматривается.

- Формула $\{\varphi\}\ S\ \{\psi\}$ называется *частично корректной*, если каждое *останавливающееся* вычисление S, стартующее в состоянии, в котором выполняется φ, финиширует в состоянии, в котором выполняется ψ.
- Формула $\{\varphi\}\ S\ \{\psi\}$ называется *тотально корректной*, если каждое вычисление S, стартующее в состоянии, в котором выполняется φ, *останавливается* и финиширует в состоянии, в котором выполняется ψ.

Таким образом, в случае частичной корректности не делается никаких предположений относительно вычислений S, которые не останавливаются, а зависают. В дальнейших объяснениях рассматривается частичная корректность, если не сказано обратное.

Основная идея подхода Хоара – доказывать корректность программ на синтаксическом уровне, используя лишь тройки определенной выше формы. Детерминированные последовательные программы конструируются по следующей грамматике:

$$S ::= \textbf{skip} \mid x := E \mid S; S \mid \textbf{if } B \textbf{ then } S \textbf{ else } S \textbf{ fi} \mid \textbf{while } B \textbf{ do } S \textbf{ od}.$$

Здесь **skip** – отсутствие операции, $x := E$ – присваивание выражения E переменной x (предполагается, что x и E имеют одинаковый тип), S; S – композиция операторов. Последние два – альтернатива и итерация, соответственно (B – булево выражение).

Правила вывода должны читаться следующим образом: если все условия, расположенные выше черты, верны, тогда следствие под чертой тоже верно. Для правил с условием true записывается только следствие. Эти правила вывода называются *аксиомами*. Формальная система для последовательных детерминированных программ изображена в табл. 1.1.

Таблица 1.1. Формальная система для частичной корректности последовательных программ

Аксиома для **skip**	$\{\varphi\}\,\textbf{skip}\,\{\varphi\}$
Аксиома для присваивания	$\{\varphi[x := k]\}\,x := k\,\{\varphi\}$
Последовательная композиция	$\dfrac{\{\varphi\}\,S_1\,\{\chi\},\ \{\chi\}\,S_2\,\{\psi\}}{\{\varphi\}\,S_1; S_2\,\{\psi\}}$
Альтернатива	$\dfrac{\{\varphi \wedge B\}\,S_1\,\{\psi\},\ \{\varphi \wedge \neg B\}\,S_2\,\{\psi\}}{\{\varphi\}\,\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}\,\{\psi\}}$
Итерация	$\dfrac{\{\varphi \wedge B\}\,S\,\{\varphi\}}{\{\varphi\}\,\textbf{while } B \textbf{ do } S \textbf{ od}\,\{\varphi \wedge \neg B\}}$
Следствие	$\dfrac{\varphi \Rightarrow \varphi',\ \{\varphi'\}\,S\,\{\psi'\},\ \psi' \Rightarrow \psi}{\{\varphi\}\,S\,\{\psi\}}$

Правило вывода для оператора **skip**, который ничего не делает, вполне ожидаемо: при любых условиях, если φ верно перед оператором, то оно верно и после него.

В соответствии с аксиомой для присваивания, начинают с постусловия φ и определяют предусловие подстановкой $\varphi[x := k]$. Эта

подстановка означает формулу φ, в которой во всех вхождениях x заменено на k. Например,

$\{k^2$ четно и $k = y\}\ x := k\ \{x^2$ четно и $x = y\}$.

Если процесс доказательства начинается с анализа постусловия, то его обычно применяют последовательно к частям программы таким образом, что в конце можно доказать предусловие для всей программы.

Правило последовательной композиции использует промежуточный предикат χ, который характеризует финальное состояние S_1 и стартовое состояние S_2.

Правило альтернативы использует булево выражение B, значение которого определяет, что именно исполняется: S_1 или S_2.

Правило для итерации требует пояснения. Оно определяет, что предикат φ выполняется после окончания **while** B **do** S **od**, если справедливость φ может быть поддержана в течение каждого исполнения тела цикла S. Это объясняет, почему φ называется *инвариантом*. Одна из основных трудностей в доказательстве правильности программ с помощью рассматриваемого подхода состоит в нахождении подходящих инвариантов. В частности, это усложняет автоматизацию таких доказательств.

Все рассмотренные правила ориентированы на такой синтаксис, что каждой синтаксической конструкции соответствует правило вывода. Это отличается от правила следствия, которое устанавливает связь между верификацией программ и логикой.

Правило следствия позволяет усиливать предусловия и ослаблять постусловия. При этом оно облегчает применение остальных правил. В частности, это правило позволяет заменять пред- и постусловия эквивалентными. Тем не менее, следует отметить, что доказательство импликаций вида $\varphi \Rightarrow \varphi'$ в общем случае неразрешимо.

Теперь обсудим тотальную корректность. Системы доказательств в табл. 1.1 недостаточно для доказательства того, что последовательная программа останавливается. Единственная синтаксическая конструкция, которая может вести к зависающим (не останавливающимся) вычислениям, – это итерация. Для доказательства наличия останова можно усилить правило вывода для итерации следующим образом:

$$\frac{\{\varphi \wedge B\}\, S\, \{\varphi\},\ \{\varphi \wedge B \wedge n = N\}\, S\, \{n < N\},\ \varphi \Rightarrow n \geq 0}{\{\varphi\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}\ \{\varphi \wedge \neg B\}}.$$

Здесь вспомогательная переменная N не входит в φ, B, n или S. Смысл этого правила заключается в том, что N является стартовым значением n, и на каждой итерации значение n уменьшается, но остается неотрицательным. Эта конструкция в точности ликвидирует бесконечные вычисления, так как n не может уменьшаться бесконечно часто без нарушения условия $n \geq 0$. Переменная n называется *вариантой*.

Формальная верификация параллельных систем

Пусть для операторов S_1 и S_2 конструкция $S_1 \| S_2$ обозначает параллельную композицию этих операторов. Основным для применения формальной верификации к параллельным программам является следующее правило вывода:

$$\frac{\{\varphi\}\, S_1 \,\{\psi\},\ \{\varphi'\}\, S_2 \,\{\psi'\}}{\{\varphi \wedge \varphi'\}\, S_1 \| S_2 \,\{\psi \wedge \psi'\}}.$$

Это правило позволяет верифицировать параллельные системы тем же путем, что и последовательные, рассматривая части программ по отдельности. Ввиду взаимодействия между S_1 и S_2 хотя бы за счет доступа к разделяемым переменным или обмена сообщениями, это правило, к сожалению, в общем случае неверно. Много усилий было приложено к получению правил вывода описанной формы [14]. Существует несколько причин, почему достичь этого непросто.

Введение параллелизма по существу приводит к введению *недетерминизма*. Это означает, что для параллельных программ, которые взаимодействуют путем использования разделяемых переменных, поведение на входе-выходе существенно зависит от порядка, в котором к этим общим переменным получен доступ.

Например, если S_1 – это $x := x + 2$, S_2 – это $x := x + 1$; $x := x + 1$ и S_3 – это $x := 0$, значение x после выполнения $S_1 \| S_3$ может быть 0 или 2, а значение x после выполнения $S_2 \| S_3$ может быть 0, 1 или 2. Различные значения для x зависят от порядка выполнения операторов в S_1 и S_3 или S_2 и S_3. Более того, несмотря на то, что входное-выходное поведение S_1 и S_2 идентично (увеличение x на 2), нет никакой гарантии, что это будет верно в параллельном контексте.

Параллельные процессы потенциально могут взаимодействовать в любой момент их исполнения, а не только в начале вычисления. Если требуется сделать вывод о том, как параллельные программы взаимодействуют, недостаточно знать свойства их стартовых и финальных состояний. Необходимо также уметь формировать суждения о том, что происходит *во время* вычисления. Таким образом, свойства должны ссылаться не только на стартовые и финальные состояния, но и на сами вычисления.

Основная проблема классического подхода к верификации параллельных и реактивных систем состоит в том, как объяснено выше, что верификация полностью фокусируется на идее, как программа вычисляет функцию от входов к выходам. При этом заданы некоторые допустимые входы и вырабатываются некоторые желаемые выходы. Для параллельных систем вычисление обычно не завершается, и корректность зависит от поведения системы во времени, а не только от конечного результата вычисления (если только оно вообще завершается). Глобальные свойства параллельных программ часто не могут быть сформулированы в терминах отношений между входами и выходами.

Были сделаны различные попытки обобщить классическую формульную верификацию на параллельные программы. Ввиду взаимодействия между компонентами правила вывода обычно довольно сложны, и полная разработка формальной системы для параллельных систем, которые могут взаимодействовать путем использования разделяемых переменных или путем (синхронного или асинхронного) обмена сообщениями, затрудняется. Поэтому для реальных систем доказательства в таком стиле обычно становятся очень большими и сложными. При этом требуется учесть взаимодействие с пользователем и управление от него (в частности, в форме поиска подходящих инвариантов). Как результат, такие доказательства очень громоздки, неустойчивы к ошибкам, и организовать их в понятной форме затруднительно.

Темпоральная логика

Как было отмечено выше, корректность реактивных систем рассматривается применительно к поведению систем в течение времени, а не только к взаимоотношениям между входами и выходами (пред- и постусловиями) вычисления, так как обычно вычисления реактивных систем не завершаются. Рассмотрим, например, коммуникационный протокол между двумя агентами (отправителем и получателем), которые соединены двусторонним каналом связи. В этом случае свойство

«если процесс *P* посылает сообщение, он не пошлет следующее сообщение до тех пор, пока не примет подтверждения»

не может быть сформулировано в терминах пред- и постусловий.

Для того чтобы облегчить формальную спецификацию подобных свойств, пропозициональная логика должна быть расширена операторами, которые ссылаются на поведение системы во времени.

U (*until*) и **G** (*globally*) – это примеры операторов, которые относятся к последовательностям состояний (таким как, например, вычисления).

При этом $\varphi \mathbf{U} \psi$ означает, что свойство φ выполняется во всех состояниях до тех пор, пока не будет достигнуто состояние, в котором выполняется ψ, а $\mathbf{G}\,\varphi$ означает, что всегда, во всех будущих состояниях выполняется φ. Используя эти операторы, можно формализовать описанное выше свойство протокола, например, следующим образом:

$$\mathbf{G}\,[snd_P(m) \rightarrow \neg snd_P(nxt(m))\ \mathbf{U}\ rcv_P(ack)].$$

Иначе говоря, если сообщение m послано процессом P, то этот процесс не передаст следующее сообщение $nxt(m)$, пока не получит подтверждение.

Логики, расширенные операторами, которые позволяют выражать свойства о вычислениях (в частности, которые могут выражать свойства о взаимном порядке между событиями), называются *темпоральными логиками*. Эти логики в информатику ввел А. Пнуэли [15, 16]. Темпоральные логики – это широко используемый метод спецификации для выражения свойств вычислений реактивных систем на довольно высоком уровне абстракции.

Таким же способом, как и в верификации последовательных программ, можно построить правила вывода для темпоральной логики (для реактивных систем) и доказывать корректность этих систем тем же подходом, как было показано для последовательных программ с использованием предикатов. Недостатки метода верификации доказательств, который требует большого человеческого труда, такие же, как для проверки параллельных систем: доказательства громоздкие, трудоемкие и требуют высокого уровня пользовательского управления.

В данной книге рассматривается другой тип метода формальной верификации, который базируется на темпоральной логике, но, вообще говоря, требует меньшего вовлечения пользователя в процесс верификации. Он называется *проверкой моделей*.

1.5. Проверка моделей

Основная идея метода, который называется *проверкой моделей* (*model checking*) [1, 17, 18], состоит в запуске алгоритмов, исполняемых компьютером, для проверки корректности систем.

При использовании этого подхода пользователь вводит описание модели системы (возможное поведение) и описание спецификации требований (желаемое поведение), а верификацию проводит машина. Если ошибка найдена, инструментальное средство (верификатор)

предоставляет контрпример, показывающий при каких обстоятельствах ошибка может быть обнаружена. Контрпример представляет собой сценарий, в котором модель ведет себя нежелательным образом. Он свидетельствует о том, что модель неверна и должна быть исправлена[1]. Это позволяет пользователю обнаружить ошибку и исправить спецификацию модели перед тем, как продолжить верификацию. Если ни одной ошибки не найдено, то верификация может быть закончена или пользователь может уточнить модель (приняв во внимание больше проектных решений, так что модель становится более конкретной и реалистичной) и повторить процесс верификации. Схема проверки моделей приведена на рис. 1.3.

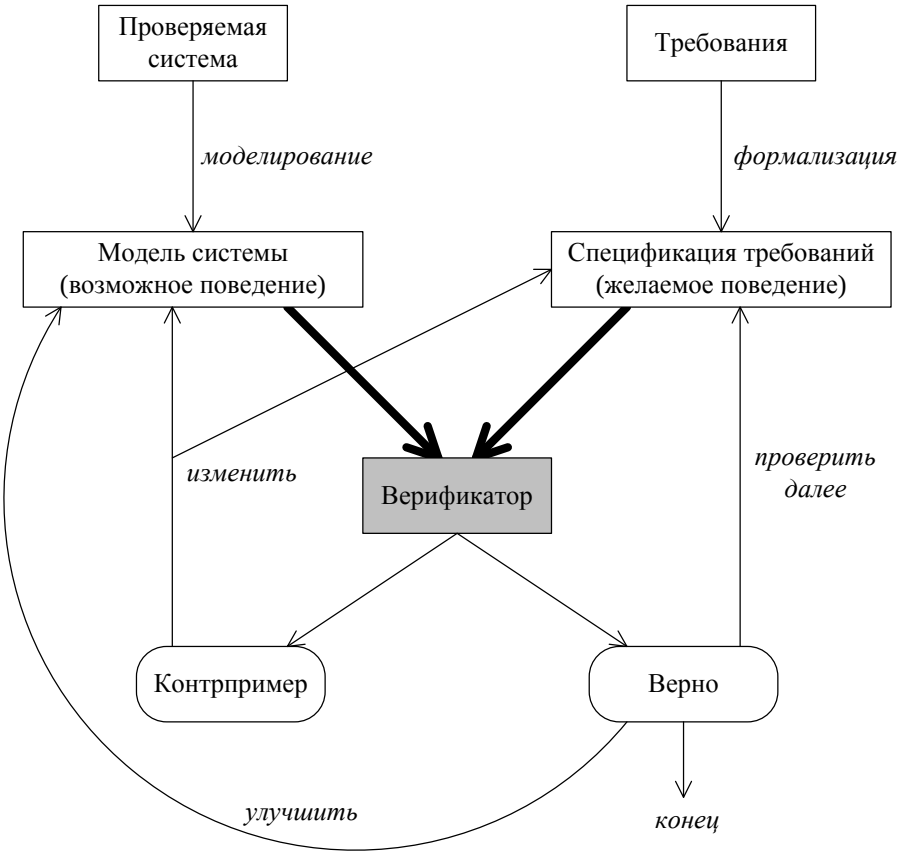


Рис. 1.3. Проверка моделей

Алгоритмы проверки моделей обычно базируются на *исчерпывающем обзоре множества всех состояний* модели системы: для каждого состояния системы проверяется, «ведет ли оно себя корректно» – удовлетворяет ли требуемому свойству. В самой простой форме этот метод известен как *анализ достижимости*. Например, пусть

[1] В некоторых случаях формальная спецификация требований может быть неверна, в том смысле, что верификатор проверяет нечто, что пользователь не хотел проверять. В этом случае пользователь, вероятно, допустил ошибку в формализации требований.

требуется выяснить, может ли система попасть в состояние, в котором вычисление не может продолжаться (так называемая *блокировка*). В этом случае достаточно определить все достижимые состояния и выяснить, существует ли достижимое состояние, в котором вычисление блокируется. Анализ достижимости применим только к доказательству отсутствия блокировок и доказательству инвариантных свойств, которые выполняются в течение всего вычисления. Этого недостаточно, например, для коммуникационных протоколов, для которых одним из важных свойств является следующее: если сообщение послано, то оно обязательно когда-нибудь будет получено. Такие типы свойств не покрываются обычной проверкой достижимости.

Протоколы моделируются множествами конечных автоматов, которые взаимодействуют путем асинхронного обмена сообщениями [19]. Начиная со стартового состояния системы, которое выражается в терминах состояний взаимодействующих автоматов и буферов сообщений, определяются все состояния системы, которые могут быть достигнуты путем обмена сообщениями. Проверка моделей может рассматриваться как преемник этих ранних методов обзора всех состояний для протоколов. Она позволяет проверять более широкий класс свойств и управляет множествами состояний гораздо эффективнее, чем ранние методы.

Методы проверки моделей

Известны два подхода в проверке моделей, которые различаются в отношении того, как описывается желаемое поведение – спецификация требований:

- *Логический или разнородный подход*. В этом случае желаемое поведение системы описывается путем формулировки множества свойств в подходящей логике (темпоральной или модальной). Система обычно моделируется как конечный автомат, в котором состояния отражают значения переменных и управляющие позиции, а переходы указывают, как система может изменять одно состояние на другое. Система считается корректной по отношению к требованиям, если заданное множество начальных состояний выполняет эти требования.
- *Поведенческий или однородный подход*. В этом случае и желаемое, и возможное поведение задаются в одной и той же нотации (автоматом), и в качестве критерия корректности используются отношения эквивалентности (или предпорядки). Отношения эквивалентности обычно фиксируют представления вида «ведет себя так же как», тогда как отношения предпорядка обозначают

представления вида «ведет себя как минимум так же как». Поскольку существуют разные взгляды относительно того, что означает для двух процессов «вести себя одинаково» (или «вести как минимум так же как»), то определены различные представления об эквивалентностях и предпорядках. Одним из наиболее известных представлений эквивалентности является двойное моделирование, в котором два автомата моделируют друг друга, если один автомат может симулировать каждый шаг другого автомата и наоборот. Часто встречающимся представлением предпорядка является включение языков. Автомат *A* включается в автомат *B*, если все слова, допускаемые *A*, допускаются также и *B*. Система считается корректной, если желаемое и возможное поведение эквивалентны (или упорядочены) по отношению к исследуемой эквивалентности (или предпорядку).

Несмотря на то, что оба этих подхода концептуально различны, связь между ними может быть установлена следующим образом. Логика индуцирует отношение эквивалентности на системах так: две системы эквивалентны, если и только если они удовлетворяют одинаковым формулам. Используя эту концепцию, можно установить взаимоотношения между логикой и отношениями эквивалентности. Например, известно, что два автомата моделируют друг друга, если они удовлетворяют одинаковым формулам логики CTL, широко используемой в процессе проверки моделей. Связь между двумя подходами теперь ясна: если две модели удовлетворяют одним и тем же свойствам (это проверяется с использованием логического подхода), то они поведенчески эквивалентны (это может быть проверено с использованием поведенческого подхода). Обратный путь более интересен, так как в общем случае непрактично проверять все свойства в определенной логике, но проверка эквивалентностей, таких как двойное моделирование, может быть проделана эффективно.

В данной книге используется логический подход [20]. Так как по существу здесь проверяется, что описание системы является моделью формул темпоральной логики, этот логический подход первоначально и назывался проверкой моделей.

Исчерпывающий обзор множества состояний гарантированно завершается ввиду конечности модели.

Достоинства проверки моделей

- Это *общий подход* с приложениями к верификации аппаратуры, программ, коммуникационных протоколов, мультиагентных систем, встраиваемых систем и т. д.

- Подход поддерживает *частичную* верификацию: проект может быть верифицирован по частичной спецификации при рассмотрении только подмножества всех требований. Это обеспечивает высокую эффективность подхода, так как можно ограничить валидацию проверкой наиболее важных свойств, игнорируя проверку менее важных, зато вычислительно более дорогих требований.
- Встраивание проверки моделей в процесс проектирования не требует больше времени, чем симуляция и тестирование. В некоторых случаях использование проверки моделей приводит к уменьшению времени разработки. Кроме того, благодаря использованию соответствующих методов, программы для проверки моделей могут работать с достаточно большими пространствами состояний.
- Программы для проверки моделей потенциально могут *регулярно использоваться* специалистами по разработке систем с той же легкостью, с какой применяются, например, компиляторы, так как проверка моделей не требует высокой степени взаимодействия с пользователем.
- *Надежный математический фундамент*: моделирование, семантика, логика и теория автоматов, структуры данных, алгоритмы на графах.

Ограничения проверки моделей

Главные ограничения проверки моделей:

- Она применима в основном к *управляющим приложениям*, в которых компоненты взаимодействуют между собой. Она меньше подходит к приложениям обработки данных, так как в таких приложениях обычно вводятся бесконечные пространства состояний.
- При использовании проверки моделей верифицируется только *модель* системы, а не сама система. Тот факт, что модель обладает определенными свойствами, не гарантирует, что финальная реализация будет обладать теми же свойствами (для проверки финальных реализаций требуются дополнительные методы, такие как систематическое тестирование). Проще говоря, любая валидация с использованием проверки моделей настолько же хороша, как и модель системы.
- Нахождение подходящей абстракции (такой как модель системы и подходящие свойства в темпоральной логике) требует

соответствующей квалификации (однако меньшей, чем доказательная верификация).

- Как и всякий инструмент, программы для проверки моделей могут быть ненадежными. Однако, поскольку основу проверки моделей составляют стандартные и хорошо известные алгоритмы, в обеспечении надежности таких программ обычно не возникает больших проблем. В некоторых случаях корректность наиболее сложных частей программ проверки моделей уже доказана с использованием программ автоматического доказательства теорем.
- Проверка моделей не позволяет проверять обобщения [21]. Если, например, протокол верифицирован для одного, двух и трех процессов с использованием проверки моделей, это не дает никакого результата для другого числа процессов – проверка моделей практична только для частных случаев. Проверка моделей может, однако, помочь сформулировать теоремы с произвольными параметрами, которые впоследствии могут быть доказаны с использованием формальной верификации.

Достичь абсолютно гарантированной корректности систем реального размера невозможно. Несмотря на указанные ограничения, можно сказать, что проверка моделей существенно повышает уровень доверия к системам.

Поскольку в проверке моделей основная идея состоит в описании поведения системы конечными автоматами, при определенных условиях число состояний может выйти за пределы размеров доступной памяти. Это, в частности, может быть для параллельных и распределенных систем, у которых много системных состояний – размер множества состояний таких систем в худшем случае пропорционален произведению размеров множеств состояний индивидуальных компонент. Проблема чрезмерного увеличения числа состояний называется *проблемой комбинаторного взрыва* [17]. Как будет показано ниже, использование автоматного программирования [2] позволяет взглянуть на указанную проблему с другой точки зрения.

1.6. Автоматическое доказательство теорем

Автоматическое доказательство теорем может быть эффективно использовано в областях, где доступны математические абстракции задач. Для случая валидации систем и спецификация, и реализация системы рассматриваются как формулы, например, φ и ψ, записанные в определенной логике. При этом проверка того, что реализация

удовлетворяет спецификации, сводится к проверке формулы $\psi \to \varphi$. Это означает, что любое поведение реализации, удовлетворяющее ψ, является возможным поведением спецификации системы, и поэтому удовлетворяет φ. Заметим, что спецификация системы может позволять и другое поведение, которое не реализуется. Для доказательства $\psi \to \varphi$ используются программы автоматического доказательства теорем.

Проверка доказательств – это область, тесно связанная с доказательством теорем. Пользователь может передать доказательство теоремы программе для проверки доказательств. Программа отвечает, верно ли это доказательство. Программы проверки доказательств выполняют более простую задачу, чем программы автоматического доказательства теорем. Поэтому они могут работать с более сложными доказательствами.

Для того чтобы уменьшить объем поиска при доказательстве теорем, имеет смысл осуществлять взаимодействие с пользователем, который может быть хорошо осведомлен о наилучшей стратегии построения доказательства. Обычно такие интерактивные системы помогают в поиске доказательства за счет сохранения списка действий, которые должны быть проделаны, и предоставления подсказок, как еще не доказанные теоремы могут быть доказаны. Более того, каждый шаг доказательства верифицируется системой. Как правило, для доказательства должно быть сделано много мелких шагов, и требуется высокий уровень взаимодействия с пользователем. Обычно люди пропускают небольшие части доказательств («тривиально», «аналогично»), тогда как программа требует явного присутствия этих частей. Процесс верификации с использованием программ автоматического доказательства теорем является медленным и трудоемким. Кроме того, используемые инструменты обычно требуют довольно высокой квалификации пользователей.

Логика, используемая программами доказательства теорем и программами проверки доказательств, обычно является вариантом логики предикатов первого порядка. В этой логике имеется неограниченное множество переменных, множества функциональных и предикатных символов заданной *арности*. Арность означает число аргументов функционального или предикатного символа. *Терм* – это переменная или строка вида $f(t_1, \ldots, t_n)$, где f – функциональный символ арности n и t_i – термы. *Константы* могут быть рассмотрены как функции арности 0. *Предикат* имеет форму $P(t_1, \ldots, t_n)$, где P – предикатный символ арности n, а t_i – термы. *Предложения* в логике первого порядка – это предикаты, логические комбинации предложений или предложения, снабженные квантификацией

существования или всеобщности. В *типизированной логике* существует также множество типов, и каждая переменная имеет тип (как программная переменная `x` имеет тип `int`). Каждый функциональный символ имеет множество типов аргументов и тип результата, а каждый предикатный символ имеет множество типов аргументов, однако не имеет типа результата. По этой причине в такой логике квантификации также типизированы.

Алгоритмические компоненты программ доказательства теорем – это методы применения правил вывода и получения следствий. Важными подходами, используемыми такими программами для этого, являются естественная дедукция (например, из истинности φ_1 и φ_2 можно сделать заключение об истинности $\varphi_1 \wedge \varphi_2$), резолюция и унификация (процедура, которая используется для сопоставления двух термов друг с другом путем предоставления всех подстановок переменных, при которых термы совпадают). В отличие от традиционной проверки моделей, доказательство теорем может работать непосредственно с бесконечными множествами состояний и проверять справедливость свойств с произвольными значениями параметров.

Эти методы недостаточны для поиска доказательства заданной теоремы, если доказательство существует. Инструмент должен иметь стратегию, которая говорит, как искать доказательство. Данная стратегия может предлагать использовать правила вывода с конца, начиная с предложения, которое требуется доказать. Стратегии, которые люди применяют для поиска доказательств, неформализованы. Стратегии, которые используются программами доказательства теорем, базируются на алгоритмах обхода в ширину и в глубину.

Наконец, программы автоматического доказательства теорем не очень полезны на практике: проблема доказательства теорем экспоненциально сложна – предложение длины n может иметь доказательство экспоненциального размера от n. Поиск такого доказательства требует времени экспоненциального от его длины. Поэтому в общем случае доказательство теорем дважды экспоненциально по отношению к длине доказываемого предложения. Для интерактивных программ доказательства теорем эта сложность значительно уменьшается.

Перечислим различия между доказательством теорем и проверкой моделей:

- Проверка моделей полностью автоматическая и быстрая.
- Проверка моделей может быть применена к частичным реализациям. Поэтому она может предоставлять полезную

информацию относительно корректности систем даже в случае, если система не полностью определена.

- Программы проверки моделей имеют дружественный интерфейс и легко используются, в то время как использование программ проверки доказательств требует достаточно высокой квалификации пользователей для того, чтобы направлять и сопровождать процесс верификации. В частности, затруднительно познакомить кого-либо с логическим языком программы доказательства теорем, которым обычно является выразительная логика высокого порядка.
- Проверку моделей целесообразно применять к управляющим приложениям, например, реактивным системам. Доказательство теорем применимо для работы с бесконечными множествами состояний, и поэтому оно может использоваться для приложений обработки данных.
- В случае успешности доказательство теорем дает (почти) максимальный уровень точности и надежности доказательства.
- Проверка моделей позволяет генерировать контрпримеры, которые могут использоваться для помощи в отладке.
- При использовании проверки моделей, проект проверяется для фиксированного (и конечного) множества параметров. Применение программ доказательства теорем возможно для произвольных значений параметров.

Проверка моделей не рассматривается как «лучший» подход по сравнению с доказательством теорем. Эти методы дополняют друг друга, так как каждый из них обладает определенными достоинствами. Делаются попытки интеграции этих методов для того, чтобы получить эффект от объединения достоинств обоих подходов.

Глава 2. Математический аппарат верификации моделей

2.1. Моделирование системы

Как следует из названия, метод верификации моделей работает не напрямую с программой, а с ее моделью. В этом разделе описано, как строить такую модель.

Сначала для системы создается спецификация – набор свойств, которым она должна удовлетворять. Поскольку в методе model checking спецификация проверяется на модели, необходимо, чтобы в

модели проявлялись все свойства, которые присутствуют в спецификации. В то же время, не имеет смысла добавлять в модель те детали системы, которые не влияют на выполнение требуемых свойств. Например, если в спецификации системы управления лифтом указано, что двери лифта должны оставаться закрытыми во время движения, то во время верификации этого свойства не важно, с какой скоростью движется лифт, главное – что он движется.

В рамках данной книги наибольший интерес представляет задача верификации реактивных систем [17]. Такие системы нельзя моделировать в терминах входа-выхода, так как их работа может продолжаться бесконечно долго, и поэтому для них верифицируется поведение во времени. Для этого вводится понятие *состояния* системы, под которым в общем случае понимается ее мгновенное описание с фиксированными значениями всех переменных системы. Работу реактивной системы при этом можно представить в виде переходов между состояниями.

Формально работу системы можно представить в виде *модели Крипке* [17] – графа переходов, в котором для каждого состояния известен набор предикатов, выполняющихся в этом состоянии. Путь в модели Крипке соответствует сценарию работы реактивной системы. Опишем модель Крипке формально.

Сначала введем понятие множество *атомарных предложений* (*элементарных высказываний*) – предложений, внутренняя структура которых не может изменяться. Атомарные предложения – это базовые предложения, которые могут быть сделаны. Множество атомарных предложений обозначается *AP*. Примерами атомарных предложений являются предложения «x больше 0» или «x равно 1» для некоторой переменной x. Другими примерами таких предложений являются «идет дождь» или «в магазине нет покупателей». В принципе, атомарные предложения определяются над множеством переменных $x, y, \ldots$, констант $0, 1, 2, \ldots$, функций $max, gcd, \ldots$ и предикатов $x = 2$, $x \; mod \; 2 = 0, \ldots$; допускаются предложения вида $max(x, y) \leq 3$ или $x = y$. Не будем точно определять множество *AP*, а просто постулируем его существование.

Заметим, что выбор множества атомарных предложений *AP* важен, так как он фиксирует базовые свойства, которые могут быть сформулированы для исследуемой системы. Поэтому фиксация множества атомарных предложений может быть названа *первой ступенью абстракции*. Если кто-то решит, например, не позволить множеству *AP* ссылаться на некоторые переменные системы, то ни одно свойство, ссылающееся на эти переменные, не может быть

сформулировано, и, как следствие, ни одно такое свойство не может быть проверено.

Моделью Крипке (*структурой Крипке*) над множеством атомарных предложений *AP* называется тройка (*S*, *R*, *Label*), где

- S – непустое множество *состояний*;
- $R \subseteq S \times S$ – *тотальное отношение переходов* на S, которое сопоставляет элементу $s \in S$ его возможных потомков;
- $Label: S \to 2^{AP}$ сопоставляет каждому состоянию $s \in S$ атомарные предложения $Label(s)$, которые верны в s.

Отношение $R \subseteq S \times S$ называется *тотальным*, если оно ставит в соответствие каждому состоянию $s \in S$ как минимум одного потомка $(\forall s \in S: \exists s' \in S: (s, s') \in R)$.

Иногда еще требуют, чтобы для модели Крипке был задан набор *начальных состояний* $S_0 \subseteq S$.

Путь в модели 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 $R(s_i, s_{i+1})$.

Представления первого порядка

Опишем, как можно представить модель Крипке при помощи логики предикатов первого порядка.

Пусть $V = \{v_1, v_2, \ldots, v_n\}$ – это множество всех переменных рассматриваемой системы. Поскольку model checking не работает с системами с бесконечным числом состояний, предположим, что переменные системы принимают значения из некоего конечного множества D.

Оценкой для V назовем функцию, которая каждой переменной v из V сопоставляет значение из множества D. Состояние системы описывается мгновенными значениями всех ее переменных, и поэтому состояние является просто оценкой $s: V \to D$. Состояние можно описать в виде формулы, которая принимает истинное значение, когда значения переменных соответствуют этому состоянию.

Например, пусть $V = \{v_1, v_2, v_3\}$, $D = \{1, 2, 3\}$ и состояние s соответствует оценке $(v_1 = 1, v_2 = 2, v_3 = 3)$. Тогда это состояние можно описать в виде следующей формулы: $(v_1 = 1) \wedge (v_2 = 2) \wedge (v_3 = 3)$. При помощи оператора дизъюнкции можно соединить несколько формул для состояний и получить формулу, описывающую набор состояний: такая формула будет принимать истинное значение только в том случае, если значения переменных соответствуют одному из

выбранных состояний. Таким образом, набор начальных состояний может быть описан в виде формулы первого порядка $S_0(v_1, v_2, \ldots, v_n)$.

Опишем теперь переходы в виде формул первого порядка. Для этого понадобится дополнительное множество переменных $V' = \{v_1', v_2', \ldots, v_n'\}$. Будем называть переменные v – переменными текущего состояния, а переменные v' – переменными следующего состояния. Переходом назовем пару оценок V и V'. Тогда можно написать формулу первого порядка $R(V, V')$, которая принимает истинные значения тогда и только тогда, когда оценки V и V' соответствуют существующему переходу в системе.

Атомарные высказывания о системе можно задавать как высказывания относительно значений переменных системы. Высказывание $v = d$ истинно в состоянии s системы, если $s(v) = d$.

Теперь можно описать модель Крипке в виде формул логики предикатов первого порядка.

1. Множество состояний модели Крипке – множество всех оценок над V.
2. Множество начальных состояний модели Крипке – множество таких оценок s: $V \to D$, для которых формула первого порядка $S_0(s)$ принимает истинное значение.
3. Отношение переходов задается следующим образом. Переход между оценками s_1 и s_2 существует тогда и только тогда, когда формула первого порядка $R(s_1, s_2)$ принимает истинное значение. Для того чтобы обеспечить тотальность отношения, для всех состояний s_k, из которых нет ни одного перехода в другое состояние, введем переход в само это состояние: $R(s_k, s_k) = \text{true}$.
4. $Label$: $S \to 2^{AP}$ определяется так, чтобы $Label(s)$ принадлежали все атомарные предложения, выполняющиеся в состоянии s. Например, если $s(v_k) = d_k$, то атомарное высказывание $(v_k = d_k) \in Label(s)$.

Рассмотрим пример построения модели Крипке на основе формул логики предикатов первого порядка, который приведен в книге [17]. Пусть в системе имеются две переменные x и y, принимающие значения из множества $D = \{0, 1\}$, и пусть переход в системе задается формулой $x := x + y \pmod 2$. В начальном состоянии значения этих переменных равны единице.

Тогда формула для начального состояния принимает вид:

$$S_0(x, y) = (x = 1) \wedge (y = 1),$$

а формула для переходов:

$$R(x, y, x', y') = (x' = x + y \pmod 2) \wedge (y' = y).$$

При этом модель Крипке описывается следующими формулами:

1. $S = \{0, 1\} \times \{0, 1\}$;
2. $S_0 = \{(1, 1)\}$;
3. $R = \{((0, 0), (0, 0)); ((0, 1), (1, 1)); ((1, 0), (1, 0)); ((1, 1), (0, 1))\}$;
4. $L((0, 0)) = \{x = 0, y = 0\}$,
   $L((0, 1)) = \{x = 0, y = 1\}$,
   $L((1, 0)) = \{x = 1, y = 0\}$,
   $L((1, 1)) = \{x = 1, y = 1\}$.

Степень детализации переходов

При построении модели Крипке большое внимание следует уделять степени детализации переходов. Переходы в модели Крипке должны моделировать *атомарные* переходы исходной системы. Нельзя допускать ситуаций, когда в исходной системе наблюдались промежуточные состояния внутри одного перехода из модели Крипке. В таком случае может оказаться, что при верификации будет пропущена ошибка, которая проявляется как раз в одном из таких пропущенных состояний.

В то же время нельзя допускать ситуации, когда в модели Крипке есть состояния, которые реально не наблюдаются в верифицируемой системе. В этом случае может оказаться, что при верификации будут найдены реально не существующие ошибки.

Для пояснения приведем пример из книги [17]. Рассматривается система с двумя переменными x, y и двумя переходами α и β, которые могут выполняться параллельно:

α: $x := x + y$,

β: $y := y + x$.

В начальном состоянии $x = 1 \wedge y = 2$. Рассмотрим более детальную реализацию переходов α и β, в которой используются команды ассемблера:

α_0: load R_1, x	β_0: load R_2, y
α_1: add R_1, y	β_1: add R_2, x
α_2: store R_1, x	β_2: store R_2, y

Если в программе выполняется сначала переход α, а затем переход β, то программа попадает в состояние $x = 3 \wedge y = 5$. Если переходы выполняются в обратном порядке, то программа попадает в состояние $x = 4 \wedge y = 3$. Если же переходы совмещаются в следующем порядке:

$\alpha_0, \beta_0, \alpha_1, \beta_1, \alpha_2, \beta_2, \ldots$, то в результате программа попадет в состояние $x = 3 \wedge y = 3$.

Пусть состояние $x = 3 \wedge y = 3$ является ошибочным для системы, и требуется проверить, может ли система оказаться в этом состоянии. Допустим также, что система реализована так, что переходы α и β выполняются атомарно. В этом случае система не может оказаться в ошибочном состоянии. Однако если построить модель Крипке с более детальными переходами, то в процессе верификации будет найдена ошибка, которой не существует в исходной программе.

Пусть теперь, программа реализована детально, и возможно выполнение последовательности $\alpha_0, \beta_0, \alpha_1, \beta_1, \alpha_2, \beta_2$. Тогда программа неверна, поскольку может попасть в ошибочное состояние. Однако если построить модель Крипке с использованием лишь переходов α и β, то верификация покажет, что программа верна, но это будет неправильным результатом.

2.2. Проверка моделей для линейной темпоральной логики

Реактивные системы характеризуются *непрерывным взаимодействием* с окружением. Они реагируют на стимулы, посылаемые извне. Непрерывный характер взаимодействия реактивных систем отличает их от традиционного вида последовательных систем, которые рассматриваются обычно как функции, которые преобразуют входы в выходы. После получения входа последовательная система генерирует выход, впоследствии завершаясь после конечного числа вычислительных шагов. Благодаря трансформации входов в выходы этот тип систем иногда также называют *трансформирующими системами*. Свойства таких систем относятся к выходам, сгенерированным по входам, и формулируются в терминах пред- и постусловий.

Во многих случаях реактивные системы не завершаются. Поэтому свойства систем этого класса обычно относятся к взаимному порядку событий в системе – поведению системы в течение времени. Требования, которые разработчики налагают на реактивные системы, разбиваются на два типа:

- Свойства *безопасности* (утверждающие, что нечто плохое никогда не случится). Система удовлетворяет такому свойству, если она не демонстрирует запрещенное поведение.
- Свойства *живучести* (утверждающие, что нечто хорошее когда-либо случится). Для того чтобы удовлетворять такому свойству, система должна демонстрировать желаемое поведение.

Несмотря на неформальность, эта классификация довольно полезна: два типа свойств почти дизъюнктны, и большинство свойств может быть описано комбинацией свойств безопасности и живучести. Впоследствии некоторые авторы расширили эту классификацию другими свойствами.

Для того чтобы точно описывать эти типы свойств, были определены логические формализмы. Наиболее широко известными являются *темпоральные логики*, которые описывают спецификации систем [15]. Темпоральные логики поддерживают формулировку свойств поведения систем во времени. Существуют различные интерпретации темпоральной логики в зависимости от того, как рассматриваются временные изменения систем.

1. *Линейная* темпоральная логика позволяет формулировать свойства исполнимых вычислительных последовательностей системы.
2. *Ветвящаяся* темпоральная логика позволяет пользователю писать формулы, которые включают некоторую зависимость от выборов, совершаемых в системе в процессе выполнения. Она позволяет формулировать свойства о возможных последовательностях выполнения, которые стартуют в некотором состоянии.

Изложение теоретических вопросов и примеров проверки моделей для темпоральных логик основано на курсе лекций П. Катоена [1].

2.2.1. Синтаксис LTL

Следующее определение устанавливает множество базовых формул, которые могут быть выражены в пропозициональной линейной темпоральной логике (LTL – *linear temporal logic*).

Пусть AP – множество атомарных предложений. Тогда:

1. p является формулой для всех $p \in AP$.
2. Если φ – формула, то $\neg\varphi$ – формула.
3. Если φ и ψ – формулы, то $\varphi \vee \psi$ – формула.
4. Если φ – формула, то $\mathbf{X}\,\varphi$ – формула.
5. Если φ и ψ – формулы, то $\varphi\ \mathbf{U}\ \psi$ – формула.

Множество формул, построенных в соответствии с этими правилами, называется *формулами LTL*.

Заметим, что множество формул, полученных на основе первых трех пунктов, определяет множество всех формул пропозициональной логики. Пропозициональная логика является, таким образом,

собственным подмножеством LTL. Темпоральными операторами являются только **X** (*neXt*) и **U** (*Until*).

Синтаксис LTL может быть задан и в нотации Бэкуса-Наура. Для $p \in AP$ множество LTL-формул определяется следующим образом:

$$\varphi ::= p \mid \neg\varphi \mid (\varphi \vee \varphi) \mid \mathbf{X}\,\varphi \mid (\varphi\ \mathbf{U}\ \varphi).$$

Для логических операторов $\wedge$ (конъюнкция), $\rightarrow$ (импликация) и $\leftrightarrow$ (эквиваленция) справедливы соотношения:

$$\varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi),$$

$$\varphi \rightarrow \psi = \neg\varphi \vee \psi,$$

$$\varphi \leftrightarrow \psi = (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi).$$

Формула true равна $\varphi \vee \neg\varphi$, а false – это $\neg$true. *Темпоральные операторы* **G** (Globally, «всегда») и **F** (Future, «когда-нибудь») определяются следующим образом:

$$\mathbf{F}\,\varphi = \text{true}\ \mathbf{U}\ \varphi,$$

$$\mathbf{G}\,\varphi = \neg\mathbf{F}\,\neg\varphi.$$

Так как true верно во всех состояниях, формула $\mathbf{F}\,\varphi$ в действительности означает, что φ выполняется в какой-то момент в будущем. Предположим, что нет момента в будущем, когда выполняется $\neg\varphi$. Тогда φ выполняется все время. Это объясняет определение $\mathbf{G}\,\varphi$. В литературе **F** иногда обозначается как $\Diamond$, а **G** как $\Box$. В данной книге используется традиционная нотация **F** и **G**.

Итак, можно утверждать, что формула без темпорального оператора (**X**, **F**, **G**, **U**) на «верхнем уровне» относится к текущему состоянию, формула $\mathbf{X}\,\varphi$ – к следующему состоянию, $\mathbf{G}\,\varphi$ – ко всем будущим состояниям, $\mathbf{F}\,\varphi$ – к некоторому будущему состоянию, а **U** – ко всем будущим состояниям до тех пор, пока определенное условие не станет верным.

Для уменьшения числа скобок приоритет на темпоральных операторах вводится следующим образом. Как обычно, унарные операции имеют более высокий приоритет, чем бинарные. Например, пишется $\neg\varphi\ \mathbf{U}\ \mathbf{F}\ \psi$ вместо $(\neg\varphi)\ \mathbf{U}\ (\mathbf{F}\ \psi)$. Темпоральный оператор **U** имеет более высокий приоритет по сравнению с $\wedge$, $\vee$ и $\rightarrow$, а импликация $\rightarrow$ имеет более низкий приоритет по сравнению с $\wedge$ и $\vee$. Например,

$$((\neg\varphi) \rightarrow \psi)\ \mathbf{U}\ ((\mathbf{X}\,\varphi) \wedge (\mathbf{F}\,\psi))$$

может быть записано следующим образом:

$$(\neg\varphi \rightarrow \psi)\ \mathbf{U}\ (\mathbf{X}\,\varphi \wedge \mathbf{F}\,\psi).$$

Пример. Пусть $AP = \{x = 1,\ x < 2,\ x \geq 3\}$ – множество атомарных предложений. Примерами LTL-формул являются $\mathbf{X}(x = 1)$, $\neg(x < 2)$, $x < 2 \vee x = 1$, $(x < 2)\ \mathbf{U}\ (x \geq 3)$, $\mathbf{F}(x < 2)$ и $\mathbf{G}(x = 1)$. Вторая и третья формула также являются пропозициональными формулами. Пример LTL-формулы с вложенными темпоральными операторами: $\mathbf{G}[(x < 2)\ \mathbf{U}\ (x \geq 3)]$.

2.2.2. Семантика LTL

Приведенное выше определение дает способ построения LTL-формул, но не дает интерпретаций этим операторам. Формально LTL интерпретируется на последовательностях состояний. Интуитивно $\mathbf{X}\ \varphi$ означает, что φ верно в следующем состоянии, $\mathbf{F}\ \varphi$ – что φ становится верным впоследствии (сейчас или в какой-то момент в будущем). Однако, что при этом понимается под словами «состояние», «следующее состояние», «какой-то момент в будущем»? Для того чтобы недвусмысленно определить эти понятия, задается формальная интерпретация, обычно называемая семантикой. Формальный смысл свойств в темпоральной логике определяется в терминах *модели*.

LTL-модель – это тройка $M = (S, R, Label)$, в которой:

- S – непустое конечное множество *состояний*;
- R: $S \to S$ сопоставляет элементу $s \in S$ единственный следующий за ним элемент $R(S)$;
- $Label$: $S \to 2^{AP}$ сопоставляет каждому состоянию $s \in S$ атомарные предложения $Label(s)$, которые верны в s.

Для состояния $s \in S$ состояние $R(S)$ – единственное состояние, следующее за s. Важной характеристикой функции R является то, что она работает как генератор бесконечных последовательностей s, $R(S)$, $R(R(S))$, $R(R(R(S)))$, … Последовательности состояний для семантики LTL являются краеугольным камнем. Можно с тем же успехом определить LTL-модель как структуру $(S, \sigma, Label)$, где σ – бесконечная последовательность состояний, а S и $Label$ определены, как это сделано выше.

Функция $Label$ указывает, какие атомарные предложения верны для заданного состояния M. Если для состояния s имеем $Label(s) = \varnothing$, то это означает, что ни одно атомарное предложение не верно в состоянии s. Состояние s, в котором предложение p верно ($p \in Label(s)$), иногда называется p-состоянием.

Пример. Пусть $AP = \{x = 0,\ x = 1,\ x \neq 0\}$ – множество атомарных предложений, $S = \{s_0, \ldots, s_3\}$ – множество состояний, $R(s_i) = s_{i+1}$ для

$0 \le i < 3$ и $R(s_3) = s_3$ – функция следования и $Label(s_0) = \{x \neq 0\}$, $Label(s_1) = Label(s_2) = \{x = 0\}$, $Label(s_3) = \{x = 1,\ x \neq 0\}$ – функция, сопоставляющая атомарные предложения состояниям. В модели $M = (S, R, Label)$ атомарное предложение «$x = 0$» верно в состояниях s_1 и s_2, «$x \neq 0$» верно в s_0 и s_3, а «$x = 1$» верно только в состоянии s_3.

Заметим, что атомарное предложение может либо выполняться, либо не выполняться в любом состоянии модели M. Дальнейшая интерпретация не дается. Например, если «$x = 1$» верно в состоянии s, это не означает, что «$x \neq 0$» также верно в этом состоянии. Технически это следует из того факта, что на пометки состояний атомарными предложениями *Label* не накладывается никаких ограничений.

Смысл формул в логике определяется в терминах *отношения выполняемости* (обозначаемого $\models$) между моделью M, одним из ее состояний s и формулой φ.

$(M, s, \varphi) \in \models$ обозначается в инфиксной нотации: $M, s \models \varphi$. Идея заключается в том, что $M, s \models \varphi$ тогда и только тогда, когда φ верно в состоянии s модели M. Когда модель M ясна из контекста, будем опускать модель и писать $s \models \varphi$ вместо $M, s \models \varphi$.

Семантика LTL определяется следующим образом. Пусть $p \in AP$ – атомарное предложение, $M = (S, R, Label)$ – LTL-модель, $s \in S$ и φ, ψ – LTL-формулы. Отношение выполняемости $\models$ задается таким способом:

$$
\begin{aligned}
s &\models p && \Leftrightarrow p \in Label(s);\\
s &\models \neg\varphi && \Leftrightarrow \neg(s \models \varphi);\\
s &\models (\varphi \vee \psi) && \Leftrightarrow (s \models \varphi) \vee (s \models \psi);\\
s &\models \mathbf{X}\,\varphi && \Leftrightarrow R(s) \models \varphi;\\
s &\models (\varphi\ \mathbf{U}\ \psi) && \Leftrightarrow \exists\, j \ge 0\colon R^j(s) \models \psi \wedge (\forall 0 \le k < j\colon R^k(s) \models \varphi).
\end{aligned}
$$

Здесь $R^0(s) = s$ и $R^{n+1}(s) = R(R^n(s))$ для любого $n \ge 0$.

Если $R(s) = s'$, то состояние s' называется *прямым потомком* s. Если $R^n(s) = s'$ для $n \ge 1$, то состояние s' называется *потомком* s. Если $M, s \models \varphi$, то говорят, что модель M удовлетворяет формуле φ в состоянии s. Иначе говоря, формула φ выполняется в состоянии s модели M.

Формальная интерпретация остальных связок true, false, $\wedge$, $\rightarrow$, **G** и **F** может быть получена теперь из определений непосредственной подстановкой. Связка true верна во всех состояниях: так как true = $p \vee \neg p$, выполнимость true в состоянии s сводится к формуле $p \in Label(s) \vee p \notin Label(s)$, которая является верным предложением

для всех состояний s. Таким образом, $M, s \models \text{true}$ для всех состояний s. В качестве примера рассмотрим семантику $\mathbf{F}\ \varphi$:

$$s \models \mathbf{F}\ \varphi$$

$\Leftrightarrow$ { по определению **F** }

$$s \models \text{true}\ \mathbf{U}\ \varphi$$

$\Leftrightarrow$ { семантика **U** }

$$\exists\, j \geq 0\colon R^j(s) \models \varphi \wedge (\forall\ 0 \leq k < j\colon R^k(s) \models \text{true})$$

$\Leftrightarrow$ { вычисления }

$$\exists\, j \geq 0\colon R^j(s) \models \varphi.$$

Следовательно, $\mathbf{F}\ \varphi$ верно в состоянии s, если и только если есть некоторый (не обязательно прямой) потомок состояния s или само состояние s, в котором φ верно.

Используя этот результат для **F**, получим семантику для $\mathbf{G}\ \varphi$:

$$s \models \mathbf{G}\ \varphi$$

$\Leftrightarrow$ { по определению **G** }

$$s \models \neg\mathbf{F}\ \neg\varphi$$

$\Leftrightarrow$ { используем предыдущий вывод }

$$\neg(\exists\, j \geq 0\colon R^j(s) \models \neg\varphi)$$

$\Leftrightarrow$ { семантика $\neg$ }

$$\neg(\exists\, j \geq 0\colon \neg(R^j(s) \models \varphi))$$

$\Leftrightarrow$ { исчисление предикатов }

$$\forall\, j \geq 0\colon R^j(s) \models \varphi.$$

Таким образом, формула $\mathbf{G}\ \varphi$ верна в s, если и только если во всех потомках s, включая само s, верна формула φ.

Пример. Если формула $\varphi\ \mathbf{U}\ \psi$ верна в состоянии s, то $\mathbf{F}\ \psi$ тоже верно в этом состоянии. Следовательно, когда утверждается $\varphi\ \mathbf{U}\ \psi$, неявно подразумевается, что в будущем существует какое-то состояние, в котором выполняется ψ. Более слабый вариант оператора *Until*, оператор *Weak until* (*unless*) **W** утверждает, что φ выполняется непрерывно либо до момента, когда впервые будет выполняться ψ, либо в течение всей последовательности. Оператор **W** определяется следующим образом:

$$\varphi\ \mathbf{W}\ \psi = \mathbf{G}\ \varphi \vee (\varphi\ \mathbf{U}\ \psi).$$

Оператор **U** можно выразить через **W** двойственным способом:

φ **U** ψ = **F** $\psi \wedge (\varphi$ **W** $\psi)$.

Пример. Пусть M задается последовательностью состояний, изображенной в первом ряду рис. 2.1. Состояния обозначаются вершинами, а функция R – стрелками (стрелка идет из s в s', если и только если $R(s) = s'$). Так как R – тотальная (всюду определенная) функция, у каждого состояния есть ровно одна исходящая стрелка. Пометки *Label* указаны под состояниями.

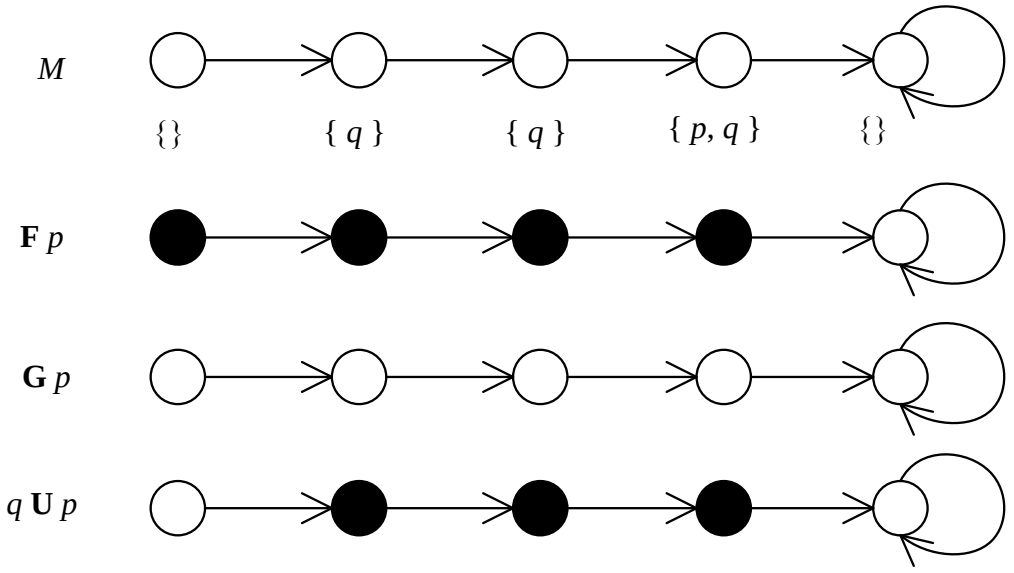


Рис. 2.1. Примеры интерпретации LTL-формул

В нижних рядах показана выполнимость трех формул: **F** p, **G** p и q **U** p. Состояние отмечено черным цветом, если формула верна в этом состоянии, и белым – в противном случае. Формула **F** p верна во всех состояниях, кроме последнего. Ни одно p-состояние не может быть достигнуто из последнего. Формула **G** p неверна во всех состояниях, так как нет ни одного состояния, потомками которого были бы только p-состояния. Из всех q-состояний (единственное) p-состояние может быть достигнуто за нуль или более шагов. Следовательно, во всех этих состояниях выполняется формула q **U** p.

Пример. Пусть M задается последовательностью состояний, изображенной на рис. 2.2, где p, q, r, s, t – атомарные предложения. Рассмотрим формулу $\mathbf{X}[r \rightarrow (q\ \mathbf{U}\ s)]$, которую запишем в виде $\mathbf{X}[\neg r \vee (q\ \mathbf{U}\ s)]$. Для всех состояний, прямой потомок которых не удовлетворяет r, формула верна ввиду первого дизъюнкта. Это применимо ко второму состоянию. В третьем состоянии формула неверна, так как r выполняется в его потомке, а q и s – нет. Наконец, в первом и четвертом состояниях формула верна, так как в их прямом потомке верно q, а в следующем состоянии верно s, и поэтому q **U** s

верно в их прямом потомке. Справедливость формул **G** *p*, **F** *t*, **GF** *r* и формулы **X**[$r \rightarrow (q$ **U** $s)$] указана в рядах ниже модели *M*.

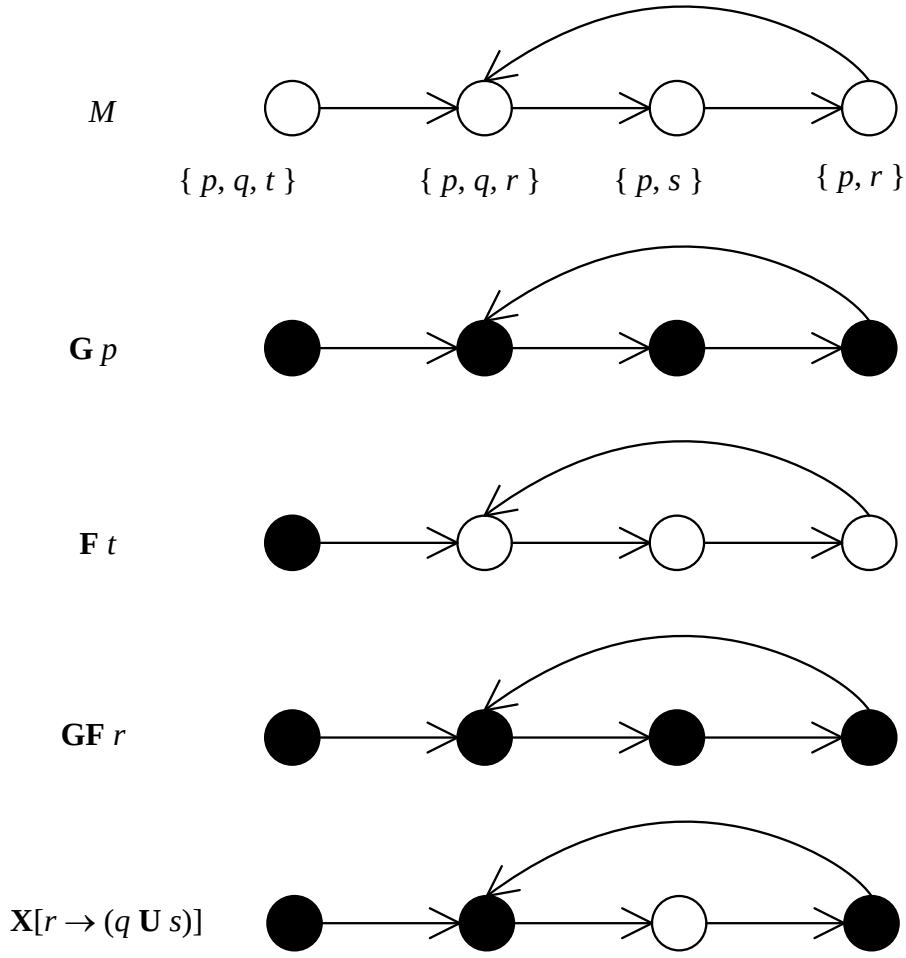


Рис. 2.2. Другие примеры интерпретации LTL-формул

Опишем теперь некоторые часто используемые предложения φ логики LTL и приведем их словесную интерпретацию такую, что $M, s \models \varphi$. Семантика последнего выражения: φ выполняется в состоянии *s* модели *M*.

1. $\varphi \rightarrow$ **F** ψ: если вначале (то есть в *s*) φ, то когда-нибудь ψ.
2. **G**[$\varphi \rightarrow$ **F** ψ]: если φ, то когда-нибудь ψ (для всех потомков *s*, включая само *s*).
3. **GF** φ: φ выполняется бесконечно часто.
4. **FG** φ: начиная с какого-то момента, φ выполняется всегда.

Проверка моделей, выполнимость и тавтологичность

В первой главе было дано неформальное определение задачи проверки моделей. Так как формальный механизм уже разработан, теперь можно дать более точную характеризацию этой задачи. Она может быть формально определена следующим образом:

дана (конечная) модель M, состояние s
и формула φ, верно ли $M, s \models \varphi$?

Задачу проверки моделей не следует путать с более традиционной задачей выполнимости в логике.

Задача выполнимости может быть сформулирована так:

дано свойство φ, существует ли модель M
и состояние s, в котором $M, s \models \varphi$?

При этом отметим, что если для проверки моделей заданы модель M и состояние s, то в задаче выполнимости это не так. Для LTL задача выполнимости разрешима. Поэтому и более простая задача проверки моделей для LTL также разрешима. Выполнимость относится к валидации систем с использованием формальной верификации. В качестве примера рассмотрим спецификацию системы и ее реализацию, которые сформулированы в виде LTL-формул φ и ψ. Проверка того, что реализация удовлетворяет спецификации, теперь сводится к проверке формулы $\psi \rightarrow \varphi$. Путем определения выполнимости этой формулы можно проверить, существует ли модель, для которой выполняется указанное соотношение. Если формула невыполнима, то такой модели не существует. Это означает, что соотношение между спецификацией и реализацией не может быть выполнено ни в какой модели.

Сходная задача, часто встречающаяся в литературе – *задача тавтологичности*:

дано свойство φ, верно ли $M, s \models \varphi$
для всех моделей M и состояний s?

Отличие этой задачи от задачи выполнимости состоит в том, что для решения задачи тавтологичности необходимо проверить, верно ли, что $M, s \models \varphi$ для *всех* существующих моделей M и состояний s, а не определять существование одной (или более) такой пары M и s. Следовательно, задача тавтологичности для формулы φ совпадает с отрицанием задачи выполнимости для формулы $\neg\varphi$.

2.2.3. Аксиоматизация

Тавтологичность LTL-формулы может быть выведена с использованием семантики, как описано выше. Это обычно громоздкая задача, так как приходится рассуждать о формальной семантике, которая определена в терминах модели M. В качестве примера попробуем вывести, что $\varphi \mathbf{U} \psi \equiv \psi \vee [\varphi \wedge \mathbf{X}(\varphi \mathbf{U} \psi)]$.

Интуитивно эта формула верна: если в текущем состоянии выполняется ψ, то выполняется и φ **U** ψ (для произвольной φ), так как ψ может быть достигнуто через путь длины 0. В противном случае, если φ выполняется в текущем состоянии, а в следующем состоянии выполняется φ **U** ψ, то φ **U** ψ выполняется в текущем состоянии. В результате получаем следующий вывод:

$$s \models \psi \vee [\varphi \wedge \mathbf{X}(\varphi \mathbf{U} \psi)]$$

$\Leftrightarrow$ { семантика $\vee$ и $\wedge$ }

$$(s \models \psi) \vee (s \models \varphi) \wedge s \models \mathbf{X}(\varphi \mathbf{U} \psi)$$

$\Leftrightarrow$ { семантика **X** }

$$(s \models \psi) \vee (s \models \varphi) \wedge R(s) \models \varphi \mathbf{U} \psi$$

$\Leftrightarrow$ { семантика **U** }

$$(s \models \psi) \vee (s \models \varphi) \wedge [\exists\, j \geq 0\colon R^j(R(s)) \models \psi \wedge$$
$$\wedge\ \forall\ 0 \leq k < j\colon R^k(R(s)) \models \varphi]$$

$\Leftrightarrow$ { вычисления с использованием $R^{n+1}(s) = R^n(R(s))$ }

$$(s \models \psi) \vee [\exists\, j \geq 0\colon R^{j+1}(s) \models \psi \wedge$$
$$\wedge\ \forall\ 0 \leq k < j\colon (R^{k+1}(s) \models \varphi \wedge s \models \varphi)]$$

$\Leftrightarrow$ { вычисления с использованием $R^0(s) = s$ }

$$(s \models \psi) \vee [\exists\, j \geq 0\colon R^{j+1}(s) \models \psi \wedge \forall\ 0 \leq k < j + 1\colon R^k(s) \models \varphi]$$

$\Leftrightarrow$ { вычисления с использованием $R^0(s) = s$ }

$$[\exists\, j = 0\colon R^0(s) \models \psi \wedge \forall\ 0 \leq k < j\colon R^k(s) \models \varphi] \vee$$
$$\vee\ [\exists\, j \geq 0\colon R^{j+1}(s) \models \psi \wedge \forall\ 0 \leq k < j + 1\colon R^k(s) \models \varphi]$$

$\Leftrightarrow$ { вычисления }

$$[\exists\, j \geq 0\colon R^j(s) \models \psi \wedge \forall\ 0 \leq k < j\colon R^k(s) \models \varphi]$$

$\Leftrightarrow$ { семантика **U** }

$$s \models \varphi \mathbf{U} \psi.$$

Как можно видеть из этих вычислений, манипуляция формулами трудоемка и ненадежна. Более эффективный путь проверять правильность формул – это использовать их синтаксис вместо их семантики. Идея состоит в определении множества правил вывода, которые позволяют переписывать LTL-формулы в семантически эквивалентные LTL-формулы на синтаксическом уровне.

Если, например, $\varphi \equiv \psi$ – это правило (аксиома), то $s \models \varphi$ тогда и только тогда, когда $s \models \psi$ для всех моделей M и состояний s.

Семантическая эквивалентность означает, что для всех состояний всех моделей эти правила выполняются. Множество правил вывода, полученных таким образом, называют *аксиоматизацией*.

В этом разделе приводятся несколько удобных базовых правил. Правила, перечисленные ниже, сгруппированы, и каждой группе дано имя. С использованием правил идемпотентности и поглощения, любая непустая последовательность **F** и **G** может быть сокращена до **F**, **G**, **FG** или **GF**. Справедливость этих правил вывода может быть доказана и путем использования семантической интерпретации, как было видно ранее для первого закона расширения. Разница в том, что эти громоздкие доказательства требуется провести только однажды, а после этого правила могут применяться повсеместно. Отметим, что справедливость правил расширения для **F** и **G** следует непосредственно из справедливости правила расширения для **U** по определению операторов **F** и **G**.

Аксиоматическое правило называется *корректным*, если оно выполняется. Формально, аксиома $\varphi \equiv \psi$ называется корректной, если и только если для всех моделей M и состояний s в M:

$M, s \models \varphi$ тогда и только тогда, когда $M, s \models \psi$.

Правила двойственности:

$$\neg \mathbf{G}\ \varphi \equiv \mathbf{F}\ \neg\varphi;$$

$$\neg \mathbf{F}\ \varphi \equiv \mathbf{G}\ \neg\varphi;$$

$$\neg \mathbf{X}\ \varphi \equiv \mathbf{X}\ \neg\varphi;$$

$$\neg(f\ \mathbf{U}\ g) \equiv \neg g\ \mathbf{W}\ \neg(f \vee g);$$

$$\neg(f\ \mathbf{W}\ g) \equiv \neg g\ \mathbf{U}\ \neg(f \vee g).$$

Правила идемпотентности:

$$\mathbf{GG}\ \varphi \equiv \mathbf{G}\ \varphi;$$

$$\mathbf{FF}\ \varphi \equiv \mathbf{F}\ \varphi;$$

$$\varphi\ \mathbf{U}\ (\varphi\ \mathbf{U}\ \psi) \equiv \varphi\ \mathbf{U}\ \psi;$$

$$(\varphi\ \mathbf{U}\ \psi)\ \mathbf{U}\ \psi \equiv \varphi\ \mathbf{U}\ \psi.$$

Правила поглощения:

$$\mathbf{FGF}\ \varphi \equiv \mathbf{GF}\ \varphi;$$

$$\mathbf{GFG}\ \varphi \equiv \mathbf{FG}\ \varphi.$$

Правило коммутирования:

$$\mathbf{X}(\varphi\ \mathbf{U}\ \psi) \equiv (\mathbf{X}\ \varphi)\ \mathbf{U}\ (\mathbf{X}\ \psi).$$

Правила расширения:

$$\varphi \mathbf{U} \psi \equiv \psi \vee [\varphi \wedge \mathbf{X}(\varphi \mathbf{U} \psi)];$$

$$\varphi \mathbf{W} \psi \equiv \psi \vee [\varphi \wedge \mathbf{X}(\varphi \mathbf{W} \psi)];$$

$$\mathbf{F} \varphi \equiv \varphi \vee \mathbf{XF} \varphi;$$

$$\mathbf{G} \varphi \equiv \varphi \wedge \mathbf{XG} \varphi.$$

Применение корректных аксиом к заданной формуле означает, что справедливость формулы не изменяется: аксиомы не изменяют семантику формулы. Если для любых семантически эквивалентных φ и ψ эту эквивалентность возможно вывести, используя аксиомы, то аксиоматизацию называют *полной*. Приведенная аксиоматизация для LTL является корректной и полной [22].

2.2.4. Расширения LTL

Строгая и нестрогая интерпретации. $\mathbf{G}\varphi$ означает, что φ выполняется во всех состояниях, включая текущее. Это называется *нестрогой* интерпретацией, так как формула также ссылается на текущее состояние. В отличие от нее, *строгая* интерпретация не ссылается на текущее состояние. Строгая интерпретация **G**, обозначаемая $\tilde{\mathbf{G}}$, определяется как $\tilde{\mathbf{G}}\varphi \equiv \mathbf{XG}\varphi$. Следовательно, $\tilde{\mathbf{G}}\varphi$ означает, что φ выполняется для всех состояний-потомков, не говоря ничего о текущем состоянии. Аналогично, строгие варианты **F** и **U** определяются как $\tilde{\mathbf{F}}\varphi \equiv \mathbf{XF}\varphi$ и $\varphi \tilde{\mathbf{U}} \psi \equiv \mathbf{X}(\varphi \mathbf{U} \psi)$. (Отметим, что для **X** нет смысла различать строгую и нестрогую интерпретации). Эти определения показывают, что строгая интерпретация может быть определена в терминах нестрогой интерпретации. В обратную сторону имеем:

$$\mathbf{G} \varphi \equiv \varphi \wedge \tilde{\mathbf{G}} \varphi;$$

$$\mathbf{F} \varphi \equiv \varphi \vee \tilde{\mathbf{F}} \varphi;$$

$$\varphi \mathbf{U} \psi \equiv \psi \vee [\varphi \wedge (\varphi \tilde{\mathbf{U}} \psi)].$$

Первые два равенства следуют из данных выше определений $\tilde{\mathbf{G}}$ и $\tilde{\mathbf{F}}$. Третье равенство подтверждается правилом расширения из предыдущего раздела: если подставить $\varphi \tilde{\mathbf{U}} \psi \equiv \mathbf{X}(\varphi \mathbf{U} \psi)$ в третье равенство, то получим правило расширения для **U**. Несмотря на то, что иногда используется строгая интерпретация, более распространена нестрогая, которая и рассматривается в этой книге.

Операторы предшествования. Все операторы ссылаются на будущее, включая текущее состояние. Как следствие, эти операторы известны

как *операторы будущего*. Иногда LTL расширяется *операторами предшествования*. Это может быть полезно в некоторых приложениях, так как иногда свойства легче выражаются в терминах прошлого, чем в терминах будущего. Например, $\underline{\mathbf{G}}\,\varphi$ («всегда в прошлом») означает (в нестрогой интерпретации), что φ выполняется сейчас и в любом состоянии ранее. $\underline{\mathbf{F}}\,\varphi$ («когда-нибудь в прошлом») – φ выполняется в текущем состоянии или в некотором состоянии ранее, а $\underline{\mathbf{X}}\,\varphi$ означает, что φ выполняется в предыдущем состоянии, если такое состояние существует. Как и для операторов будущего, для операторов предшествования могут быть даны строгая и нестрогая интерпретации. Основная причина введения операторов предшествования состоит в упрощении записи спецификации. При этом отметим, что выразительная сила LTL не увеличивается при добавлении операторов предшествования[2] [23].

2.2.5. Спецификация свойств в LTL

Для того чтобы сформулировать идею о том, как выражать неформальные свойства в линейной темпоральной логике, рассмотрим два примера. В первом из них попытаемся сформулировать свойства простой коммуникационной системы, а во втором – обратимся к формальной спецификации требований к протоколу выбора лидера в распределенной системе, в которой процессы могут начинаться в произвольные моменты времени.

Коммуникационный канал

Рассмотрим однонаправленный канал между двумя сообщающимися процессами: отправителем (*S*) и получателем (*R*). Отправитель снабжен выходным буфером (`S.out`), а получатель – входным. Оба буфера имеют неограниченную емкость. Если *S* посылает сообщение *m* к *R*, то он помещает сообщение в свой выходной буфер `S.out`. Выходной буфер `S.out` и входной буфер `R.in` соединены однонаправленным каналом. Получатель принимает сообщения путем удаления их из своего входного буфера `R.in`. Считаем, что все сообщения одинаково идентифицированы, а множество атомарных предложений $AP = \{m \in S.out,\ m \in R.in\}$, где *m* обозначает сообщение. При этом неявно предполагается, что все свойства формулируются для любого сообщения *m* (предполагается универсальная квантификация по *m*). Это делается для удобства и не повлияет на

[2] Если в качестве базовой семантической идеи взята дискретная модель, как в данном разделе.

разрешимость задачи выполнимости, если предположить, что число сообщений конечно. Кроме того, будем считать, что буферы `S.out` и `R.in` ведут себя нормально – не «портят» и не теряют сообщения, и сообщение не может находиться в буфере бесконечно долго. Схема взаимодействия отправителя и получателя приведена на рис. 2.3.

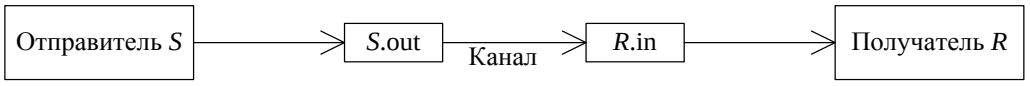


Рис. 2.3. Взаимодействие отправителя и получателя

Формализуем следующие неформальные требования к каналу с помощью LTL:

- «Сообщение не может быть в обоих буферах одновременно»:
  $\mathbf{G}\,\neg(m \in S.out \wedge m \in R.in)$.
- «Канал не теряет сообщения». Если канал не теряет сообщения, то это означает, что сообщения, помещенные в `S.out`, в конечном счете будут доставлены в `R.in`:
  $\mathbf{G}(m \in S.out \rightarrow \mathbf{F}(m \in R.in))$.
  Отметим, что если уникальность сообщений не подразумевается, то этого свойства недостаточно, так как если, например, переданы две копии m и только последняя из них получена, то такая ситуация удовлетворяла бы этому свойству. Уникальность сообщений – это предпосылка спецификации требований в темпоральной логике для систем, передающих сообщения. Если предполагать справедливость предыдущего свойства, то эквивалентно можно записать:
  $\mathbf{G}(m \in S.out \rightarrow \mathbf{XF}(m \in R.in))$,
  так как m не может быть в `S.out` и `R.in` одновременно.
- «Канал сохраняет порядок». Это означает, что если m, а затем m' передано отправителем в его выходной буфер `S.out`, то m будет принято получателем перед m':
  $\mathbf{G}[m \in S.out \wedge \neg m' \in S.out \wedge \mathbf{F}(m' \in S.out) \rightarrow$
  $\rightarrow \mathbf{F}(m \in R.in \wedge \neg m' \in R.in \wedge \mathbf{F}(m' \in R.in))]$.
  Отметим, что в предпосылке конъюнкт $\neg m' \in S.out$ требуется для того, чтобы специфицировать, что m' отправлено в `S.out` после m. $\mathbf{F}(m' \in S.out)$ само по себе не исключает, что m' в буфере отправителя, когда m в `S.out`.
- Канал не генерирует сообщения спонтанно. Это означает, что любое m в `R.in` должно быть заранее послано отправителем `S`. С использованием оператора предшествования **F** это может быть удобно формализовано следующим образом:

$\mathbf{G}[(m \in R.in) \rightarrow \underline{\mathbf{F}}(m \in S.out)]$.

При отсутствии операторов предшествования можно использовать оператор **U**:

$\mathbf{G}[\neg(m \in R.in)\mathbf{U}(m \in S.out)]$.

Динамический выбор лидера

В современных распределенных системах некоторые функции (или сервисы) предоставляются выделенными процессами в системе. Речь может идти о присвоении адреса и регистрации, координации запросов в распределенной базе данных, распределении времени, обновлении маркера после его потери в кольцевой сети с маркерным доступом, балансировке загрузки и т. д. Обычно многие процессы в системе потенциально могут это делать. Однако для совместимости в любой момент времени только одному процессу в действительности разрешается предоставлять функцию. Таким образом, один процесс, называемый «лидером», должен быть выбран для поддержки данной функции. Иногда достаточно выбрать произвольный процесс, но для других функций важно выбрать процесс с наилучшими возможностями выполнения данной функции. Данный пример будет абстрагироваться от индивидуальных возможностей и использовать ранжирование на базе идентификаторов процессов. Таким образом, идея состоит в том, что процесс с более высоким идентификатором имеет лучшие возможности.

Предположим, что имеется конечное число процессов, соединенных некоторыми коммуникационными средствами. Коммуникация между процессами асинхронна, как в предыдущем примере. Схематично это изображено на рис. 2.4.

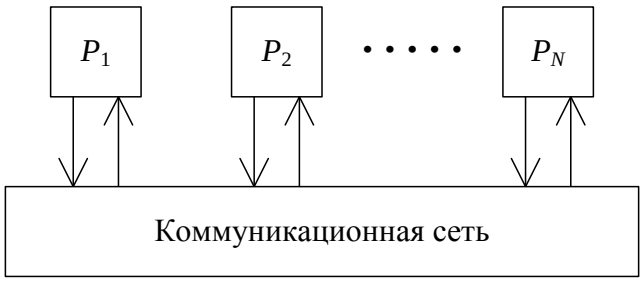


Рис. 2.4. Коммуникация между процессами

Каждый процесс имеет уникальный идентификатор, и предполагается существование линейного порядка на этих идентификаторах. Процессы ведут себя динамически в том смысле, что они в начале неактивны (не участвуют в выборе) и могут стать активными (принять участие в выборе) в произвольные моменты времени. Для того чтобы система удовлетворяла некоторому *свойству прогресса*, считается,

что процесс не может быть неактивным неограниченно долго – каждый процесс становится активным в какой-то момент. Как только процесс принимает участие в выборах, он продолжает это делать и более не становится неактивным. Для заданного множества активных процессов должен быть выбран лидер. Если неактивный процесс становится активным, то выполняется новый выбор, если этот процесс имеет более высокий идентификатор, чем текущий лидер.

Рассмотрим множество атомарных предложений:

$$AP = \{leader_i, active_i, i < j \mid 1 \le i, j \le N\},$$

где $leader_i$ означает, что процесс i является лидером, $active_i$ – что процесс i активен, а $i < j$ – что идентификатор i меньше, чем идентификатор j (в линейном порядке идентификаторов). Используем i и j в качестве идентификаторов процессов, а через N обозначим число процессов. Будем считать, что неактивный процесс не является лидером.

В дальнейших формулировках используем универсальную и экзистенциальную квантификацию по множеству идентификаторов. Строго говоря, кванторы не является частью LTL. Так как в данном случае рассматривается конечное число процессов, универсальная квантификация $\forall i: P(i)$ (где P – некоторое предложение о процессах) может быть расширена до $P(1) \wedge \ldots \wedge P(N)$. Так же может быть расширено $\exists i: P(i)$. Квантификация в данном случае может быть рассмотрена как сокращение:

- «Всегда имеется ровно один лидер»:
  $\mathbf{G}[\exists i: leader_i \wedge (\forall j \neq i: \neg leader_j)]$
  Так как в данном случае работаем с динамической системой, в которой все процессы могут быть первоначально неактивны (следовательно, нет и лидера), то это свойство, вообще говоря, неверно. Также в распределенной системе с асинхронной коммуникацией переключение от одного лидера к другому с трудом может быть сделано атомарным, и удобно разрешить временное отсутствие лидера. В качестве первой попытки можно изменить приведенную выше формулу на $\mathbf{GF}[\exists i: leader_i \wedge (\forall j \neq i: \neg leader_j)]$, которая разрешает временное отсутствие лидера, но разрешает также на время иметь нескольких лидеров одновременно. С точки зрения логичности это нежелательно. Поэтому рассмотрим следующие два свойства.
- «Всегда имеется максимум один лидер»:
  $\mathbf{G}[leader_i \rightarrow \forall j \neq i: \neg leader_j]$

- «В свое время будет достаточно лидеров». (Это требование избегает конструкции протокола выбора лидера, который никогда не выбирает лидера. Такой протокол удовлетворяет предыдущему требованию, но он не желателен):
  $\mathbf{GF}[\exists i: leader_i]$
  Это свойство не означает, что будет бесконечно много лидеров. Оно утверждает лишь то, что будет бесконечно много состояний, в которых лидер существует.
- «В присутствии активного процесса с более высоким идентификатором лидер в какой-то момент уступит»:
  $\mathbf{G}[\forall i, j: ((leader_i \wedge i < j \wedge \neg leader_j \wedge active_j) \rightarrow \mathbf{F}\ \neg leader_i)]$.
  Из соображений эффективности нежелательно, чтобы лидер в какой-то момент уступал в присутствии неактивного процесса, который может участвовать в выборах в какой-то неизвестный момент в будущем. Поэтому требуется, чтобы j был активным процессом.
- «Новый лидер будет лучше предыдущего». Это свойство требует, чтобы последовательность лидеров имела возрастающие идентификаторы:
  $\mathbf{G}[\forall i, j: (leader_i \wedge \neg \mathbf{X}\ leader_i \wedge \mathbf{XF}\ leader_j) \rightarrow i < j]$.
  Отметим, что это требование предполагает, что процесс, который уступил один раз, больше не станет лидером.

Другие примеры спецификации темпоральных свойств приведены в работе [24].

2.2.6. Проверка моделей для LTL

Задача проверки моделей для линейной темпоральной логики формулируется в терминах моделей Крипке.

Каждый путь в модели Крипке, начинающийся в состоянии s_0, представляет собой LTL-модель. Задача проверки LTL-формулы для модели Крипке состоит в том, чтобы определить, *на всех ли путях выполняется данная формула* (или, что то же самое, *существуют ли пути, на которых формула не выполняется*).

Начнем с неформального описания алгоритма верификации моделей [22]. По заданной формуле f и модели Крипке M построим *размеченную LTL-таблицу T* для формулы f. Эта таблица представляет собой модель Крипке с фиксированным множеством *начальных состояний*, состоящую из *всех* путей, которые удовлетворяют формуле f. Построение таблицы происходит следующим образом.

Приведение формулы к разделенной нормальной форме

Иногда формулу для удобства вначале приводят к форме с тесными отрицаниями (которые стоят только перед атомарными предложениями). Для этого применяют следующие правила:

$\neg\neg f \equiv f$;

$\neg(f \vee g) \equiv \neg f \wedge \neg g$;

$\neg(f \wedge g) \equiv \neg f \vee \neg g$;

$f \to g \equiv \neg f \vee g$;

$\neg \mathbf{X}\, f \equiv \mathbf{X}\, \neg f$;

$\neg \mathbf{F}\, f \equiv \mathbf{G}\, \neg f$;

$\neg \mathbf{G}\, f \equiv \mathbf{F}\, \neg f$;

$\neg(f\ \mathbf{U}\ g) \equiv \neg g\ \mathbf{W}\ \neg(f \vee g)$;

$\neg(f\ \mathbf{W}\ g) \equiv \neg g\ \mathbf{U}\ \neg(f \vee g)$.

Пример.

Было: $\neg(f \wedge (\mathbf{F}\, g \vee \mathbf{X}(h\ \mathbf{W}\ r)))$.

Стало: $\neg f \vee \mathbf{G}\, \neg g \wedge \mathbf{X}(\neg r\ \mathbf{U}\ (\neg h \wedge \neg r))$.

Отметим, что этот процесс не является обязательным – излагаемый ниже алгоритм работает и для формул общего вида.

После этого декомпозируем формулу, вводя новую атомарную переменную p_i для каждой ее внутренней подформулы f_i, у которой последняя операция (с самым низким приоритетом) – темпоральная, и добавляя в формулу новый конъюнкт $\mathbf{G}(p_i \to f_i)$. Этот конъюнкт свидетельствует о том, что всегда, когда верно p_i, формула f_i тоже верна. Это означает, что переменная p_i «кодирует» f_i. В самой же формуле подформулы f_i заменим соответствующими им атомарными переменными p_i.

Пример.

Было: $\neg f \vee \mathbf{G}\, \neg g \wedge \mathbf{X}(\neg r\ \mathbf{U}\ (\neg h \wedge \neg r))$.

Стало: $\mathbf{G}(p_1 \to \mathbf{G}\, \neg g) \wedge \mathbf{G}(p_2 \to (\neg r\ \mathbf{U}\ (\neg h \wedge \neg r))) \wedge \mathbf{G}(p_3 \to \mathbf{X}\, p_2) \wedge (\neg f \vee p_1 \wedge p_3)$.

Здесь добавлены три атомарных переменных: одна для подформулы $\mathbf{G}\, \neg g$, одна – для подформулы $(\neg r\ \mathbf{U}\ (\neg h \wedge \neg r))$ и одна – для подформулы $\mathbf{X}(\neg r\ \mathbf{U}\ (\neg h \wedge \neg r))$. После этого все такие подформулы были заменены на соответствующие атомарные переменные. Смысл формулы при этом не изменился.

На данном этапе рассматриваемая формула представляет собой конъюнкцию одной формулы, не содержащей темпоральных операторов, с несколькими формулами вида $\mathbf{G}(p_i \to \mathbf{T})$, где $\mathbf{T}$ – применение некоторого темпорального оператора к одному или двум операндам, и эти операнды не содержат других темпоральных операторов. Теперь требуется избавиться от конъюнкций определенного вида.

1. Избавляемся от $\mathbf{G}(p_i \to \mathbf{G}\, f_i)$. Для каждого такого конъюнкта введем новую атомарную переменную r_i и заменим этот конъюнкт такой формулой:
   $\mathbf{G}(p_i \to r_i) \wedge \mathbf{G}(r_i \to f_i) \wedge \mathbf{G}(r_i \to \mathbf{X}\, r_i)$.
   Смысл переменной r_i состоит в том, что она дает «обещание», что f_i будет верно всегда.

2. Избавляемся от $\mathbf{G}(p_i \to (f_i\, \mathbf{W}\, g_i))$. Для каждого такого конъюнкта введем новую атомарную переменную r_i и заменим этот конъюнкт такой формулой:
   $\mathbf{G}(p_i \to g_i \vee f_i \wedge r_i) \wedge \mathbf{G}(r_i \to \mathbf{X}(g_i \vee f_i \wedge r_i))$.
   Переменная r_i дает такое обещание: если g_i «сейчас» еще не выполняется, то, во-первых, сейчас выполняется f_i, а во-вторых, то же самое гарантируется на следующем шаге.

3. Избавляемся от $\mathbf{G}(p_i \to (f_i\, \mathbf{U}\, g_i))$. Этот этап отличается от предыдущего только тем, что свойство g_i когда-нибудь гарантированно наступит. Для каждого такого конъюнкта введем новую атомарную переменную r_i и заменим этот конъюнкт формулой:
   $\mathbf{G}(p_i \to g_i \vee f_i \wedge r_i) \wedge \mathbf{G}(r_i \to \mathbf{X}(g_i \vee f_i \wedge r_i)) \wedge \mathbf{G}(p_i \to \mathbf{F}\, g_i)$.
   Формула отличается от предыдущей только добавлением конъюнкта $\mathbf{G}(p_i \to \mathbf{F}\, g_i)$. На следующем шаге этот конъюнкт будет ликвидирован.

4. Избавляемся от $\mathbf{G}(p_i \to \mathbf{F}\, f_i)$. Для каждого такого конъюнкта введем новую атомарную переменную r_i и заменим этот конъюнкт такой формулой:
   $\mathbf{G}(p_i \to (\neg f_i \to r_i)) \wedge \mathbf{G}(r_i \to \mathbf{X}(\neg f_i \to r_i)) \wedge \mathbf{GF}\, \neg r_i$.
   Смысл переменной r_i состоит в следующем: она верна, если f_i еще не наступило (после активации p_i), причем конъюнкт $\mathbf{GF}\, \neg r_i$ «обещает», что после любой активации p_i она (r_i) когда-нибудь станет неверна и, соответственно, f_i станет верна.

Пример.

Было: $\mathbf{G}(p_1 \to \mathbf{G}\, \neg g)$.

Стало: $\mathbf{G}(p_1 \to r_1) \wedge \mathbf{G}(r_1 \to \neg g) \wedge \mathbf{G}(r_1 \to \mathbf{X}\, r_1)$.

Было: $\mathbf{G}(p_2 \to (\neg r\ \mathbf{U}\ (\neg h \wedge \neg r)))$.

Стало: $\mathbf{G}(p_2 \to (\neg h \wedge \neg r \vee \neg r \wedge r_2)) \wedge \mathbf{G}(r_2 \to \mathbf{X}(\neg h \wedge \neg r \vee \neg r \wedge r_2)) \wedge \mathbf{G}(p_2 \to \mathbf{F}(\neg h \wedge \neg r))$.

Было: $\mathbf{G}(p_2 \to \mathbf{F}(\neg h \wedge \neg r))$.

Стало:
$\mathbf{G}(p_2 \to (\neg(\neg h \wedge \neg r) \to r_3)) \wedge \mathbf{G}(r_3 \to \mathbf{X}(\neg(\neg h \wedge \neg r) \to r_3)) \wedge \mathbf{GF}\ \neg r_3$.

После выполнения всех этих операций вынесем за скобку все операторы **G** и пропозициональные подформулы (их можно также упростить). Исходная формула при этом приобретет вид:

$$p \wedge \mathbf{G}(q \wedge (r_1 \to \mathbf{X}\ s_1) \wedge \ldots \wedge (r_n \to \mathbf{X}\ s_n) \wedge \mathbf{F}\ t_1 \wedge \ldots \wedge \mathbf{F}\ t_m),$$

где p, q, $r_1, \ldots, r_n$, $s_1, \ldots, s_n$, $t_1, \ldots, t_m$ — пропозициональные подформулы, не содержащие темпоральных операторов.

Смысл полученной формулы такой: формула p «действует только в стартовый момент времени», формула q «действует всегда», формулы r_i и s_i (при i от 1 до n) определяют условие на следующий шаг в любой позиции модели Крипке, а формулы t_j (при j от 1 до m), называемые *условиями справедливости*, должны быть достижимы из любого состояния модели Крипке.

Пример.

Выше была рассмотрена формула $\neg(f \wedge (\mathbf{F}\ g \vee \mathbf{X}(h\ \mathbf{W}\ r)))$.

После выполнения всех описанных операций она превращается в следующую:

$$(\neg f \vee p_1 \wedge p_3) \wedge \mathbf{G}($$
$$(p_1 \to r_1) \wedge (r_1 \to \neg g) \wedge (p_2 \to \neg r \wedge (h \to r_2 \wedge r_3)) \wedge$$
$$\wedge (r_1 \to \mathbf{X}\ r_1) \wedge (r_2 \to \mathbf{X}(\neg r \wedge (h \to r_2))) \wedge$$
$$\wedge (r_3 \to \mathbf{X}(h \vee r \to r_3)) \wedge (p_3 \to \mathbf{X}\ p_2) \wedge \mathbf{F}\ \neg r_3).$$

Построение таблицы

Далее для формул p, q, $r_1, \ldots, r_n$, $s_1, \ldots, s_n$ и $t_1, \ldots, t_m$ таблица строится следующим образом. Пусть AP – множество всех атомарных предложений полученной формулы. Тогда для них имеется всего $2^{|AP|}$ вариантов значений (двоичных наборов). Построим по одной вершине на каждый возможный двоичный набор и пометим эту вершину теми атомарными предложениями, которые в этом наборе истинны. Например, для $AP = \{a, b, c\}$ в вершине, соответствующей набору

$\{a, b, c\} = \{1, 0, 1\}$, активны атомарные предложения a и c, а предложение b не активно.

Среди этих вершин оставим только те, у которых соответствующий двоичный набор удовлетворяет формуле q (делает ее значение истинным). Остальные вершины удалим, так как q должна выполняться всегда. Далее, для любых двоичных наборов x и y добавляем ребро из «вершины» x в «вершину» y, если и только если для любого i от 1 до n выполняется импликация $r_i(x) \to s_i(y)$. Это ребро соответствует условию $(r_1 \to \mathbf{X}\, s_1) \wedge \ldots \wedge (r_n \to \mathbf{X}\, s_n)$ и означает, что для любого i «если сейчас выполнено r_i, то на следующем шаге будет выполнено s_i».

Далее, отметим в качестве стартовых состояний (их может быть несколько) те, у которых соответствующий двоичный набор удовлетворяет формуле p. Это соответствует условию, что формула p верна «в момент времени 0».

Последний этап называется «чисткой таблицы» и является необязательным. Он предназначен для более эффективной работы с моделью. На этом этапе обеспечим требование, состоящее в том, что все пути в графе должны быть бесконечными и что отовсюду должны быть пути в вершины, в которых выполняются формулы t_j (j от 1 до m). Делается это «рекурсивным» удалением из модели всех вершин, у которых нет ни одного потомка, и всех таких вершин, из которых для какого-то j нет пути ни в одну вершину, в которой выполняется формула t_j. Проще говоря, в таблице следует оставить только те состояния, из которых существует хотя бы один бесконечный путь, удовлетворяющий условиям справедливости (это значит, что все t_j выполняются на таком пути бесконечно часто). Алгоритм, позволяющий проделать это эффективно (работающий за линейное время от размера таблицы), будет обсуждаться в контексте поиска справедливых путей в разд. 2.4.

После всех этих действий следует оставить в таблице только те вершины, которые достижимы из начальных, а остальные – удалить.

Полученная таблица естественным образом отражает смысл формулы, а ее бесконечные пути удовлетворяют двум свойствам:

1. Пусть σ – некоторый *абстрактный* путь (последовательность двоичных наборов, соответствующих атомарным предложениям). Если рассматриваемая формула верна для пути σ, то в таблице присутствует экземпляр этого пути, стартующий в *начальном состоянии*.
2. Пусть σ – путь *в построенной таблице* (стартующий в *начальном состоянии*) такой, что каждая пропозициональная формула t_j (при j

от 1 до m) становится верной на пути σ бесконечно часто. Тогда рассматриваемая формула верна на пути σ.

В первом свойстве гарантируется присутствие пути σ в модели. Действительно, на этом пути формула выполняется. Поэтому t_j для j от 1 до m становятся верными бесконечно часто, и каждая вершина этого пути удовлетворяет двум условиям:

- из нее стартует некоторый бесконечный путь;
- для любого j из нее можно добраться до любой вершины, в которой верна формула t_j.

Однако эту вершину можно было рекурсивно удалить только в том случае, если какое-то из этих условий нарушается. Следовательно, эту вершину не удалили, и она присутствует в модели на реализации пути σ. Из содержания формулы автоматически следует, что на *реализации* этого пути в модели каждая пропозициональная формула t_j (при j от 1 до m) становится верной бесконечно часто.

При построении таблицы процесс ее «чистки» может привести к тому, что из нее будут исключены все стартовые состояния (а значит, и все остальные, так как в модели должны быть оставлены только те состояния, которые достижимы из начальных), и модель станет пустой. Это означает, что рассматриваемая формула невыполнима – нет такого пути, на котором она становится истинной.

Во втором свойстве (которое утверждает почти то же самое, но в обратную сторону) условие на t_j существенно. Пусть построенная модель не пуста. Тогда действительно, из каждой вершины модели (по построению) можно достичь всех вершин, которые соответствуют формулам t_j. Следовательно, из каждой вершины *существует путь, на котором все формулы t_j выполняются бесконечно часто*. Этот путь строится следующим образом. Вначале добираемся из рассматриваемой вершины в вершину, для которой верно t_1, потом из t_1 в t_2, и т. д. до t_m. Из t_m снова в t_1, и так до бесконечности.

Все изложенное можно проделать, так как любое t_j достижимо из любой вершины. Но, *кроме пути, на котором все t_j становятся верными бесконечно часто, там могут быть и другие пути, для которых это условие неверно*. Поэтому условие на t_j указано явно. Тем не менее, каждый путь в модели, безусловно, удовлетворяет формуле:

$$p \wedge \mathbf{G}(q \wedge (r_1 \rightarrow \mathbf{X}\, s_1) \wedge \ldots \wedge (r_n \rightarrow \mathbf{X}\, s_n)),$$

которая отличается от исходной тем, что в ней исключили требования на t_j.

Обратим внимание, что указанные два свойства выполняются для путей и в том случае, если «чистка таблицы» не производилась.

Пример. Рассматриваем формулу:

$$x \wedge \neg y \wedge \mathbf{G}($$
$$(x \vee y) \wedge (\neg x \wedge y \rightarrow \mathbf{X}(x \wedge y \wedge \neg z)) \wedge$$
$$\wedge (x \wedge y \wedge \neg z \rightarrow \mathbf{X}(\neg x \wedge y \vee \neg(y \vee z))) \wedge$$
$$\wedge (\neg(y \vee z) \rightarrow \mathbf{X}(x \wedge y \wedge \neg z \vee \neg y \wedge z)) \wedge$$
$$\wedge (x \wedge z \rightarrow \mathbf{X}(x \wedge z)) \wedge (\neg y \wedge z \rightarrow \mathbf{X}(y \wedge z)) \wedge$$
$$\wedge \mathbf{F}\, \neg(x \vee z) \wedge \mathbf{F}(\neg(x \wedge y) \wedge z)\mathbf{)}.$$

Имеется 3 атомарных предложения: x, y, z. Это дает 8 состояний (рис. 2.5).

Поскольку всегда должна быть верна формула $x \vee y$, исключаем состояния, в которых она неверна. На рис. 2.6 они обозначены пунктиром.

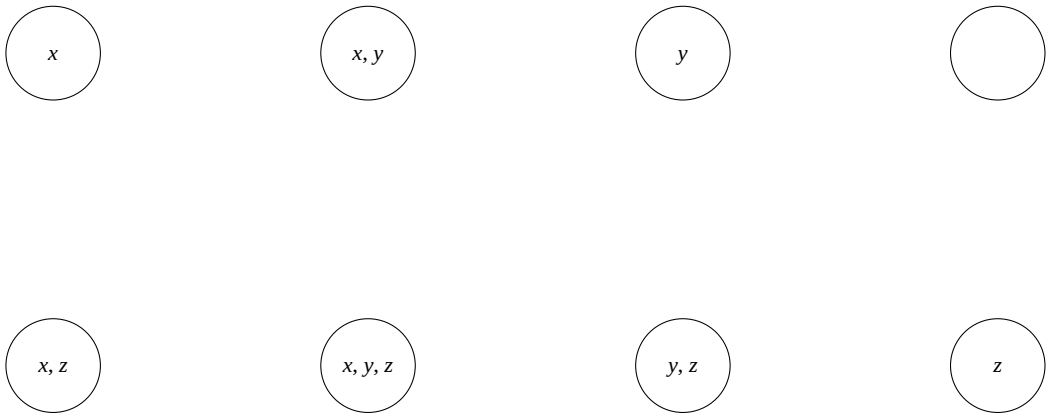


Рис. 2.5. Состояния таблицы

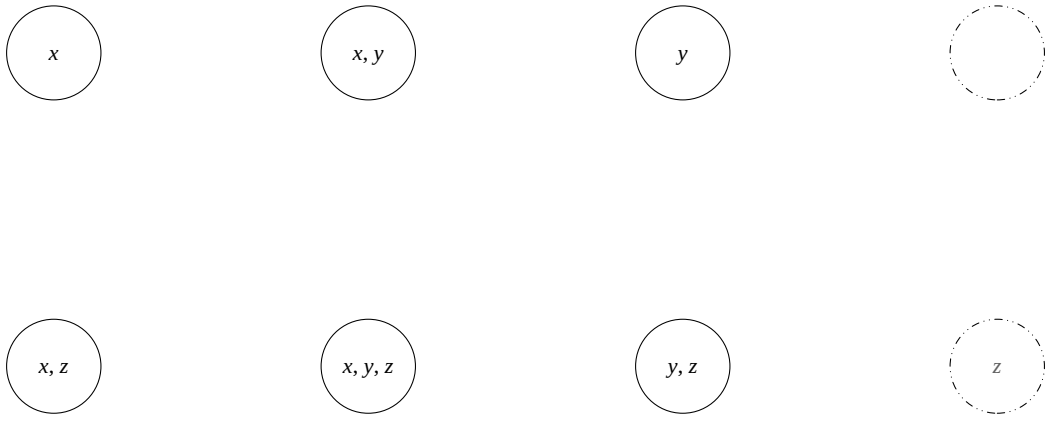


Рис. 2.6. Активные состояния таблицы

Теперь свойство $\mathbf{G}((r_1 \to \mathbf{X}\, s_1) \wedge \ldots \wedge (r_n \to \mathbf{X}\, s_n))$ формирует переходы (пунктирными стрелками на рис. 2.7 обозначены переходы, у которых хотя бы один из концов находится в удаленном состоянии).

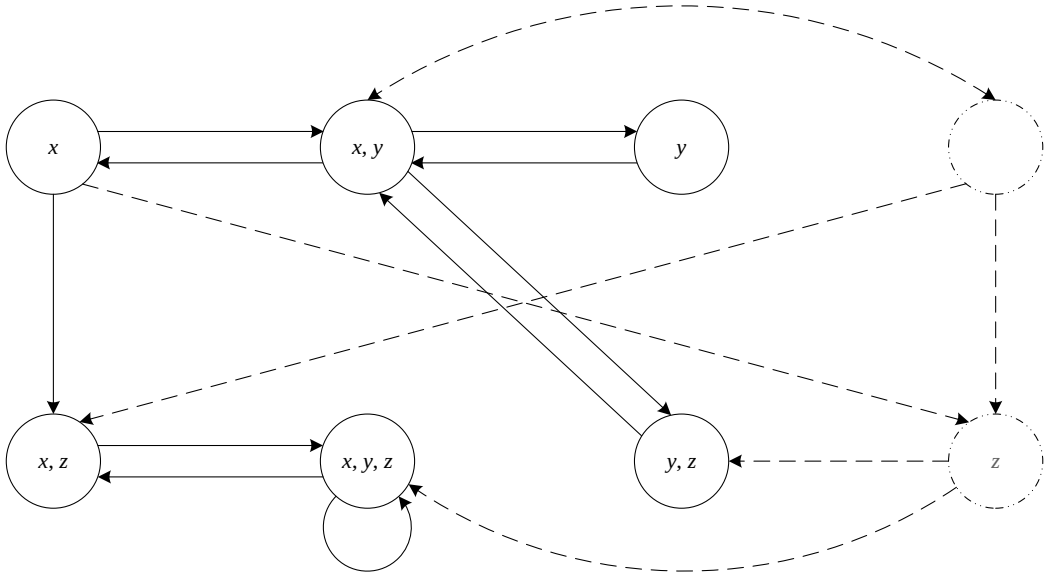


Рис. 2.7. Переходы таблицы

Стартовые состояния – это те, в которых верна формула $x \wedge \neg y$. Обозначим их жирными стрелками (рис. 2.8).

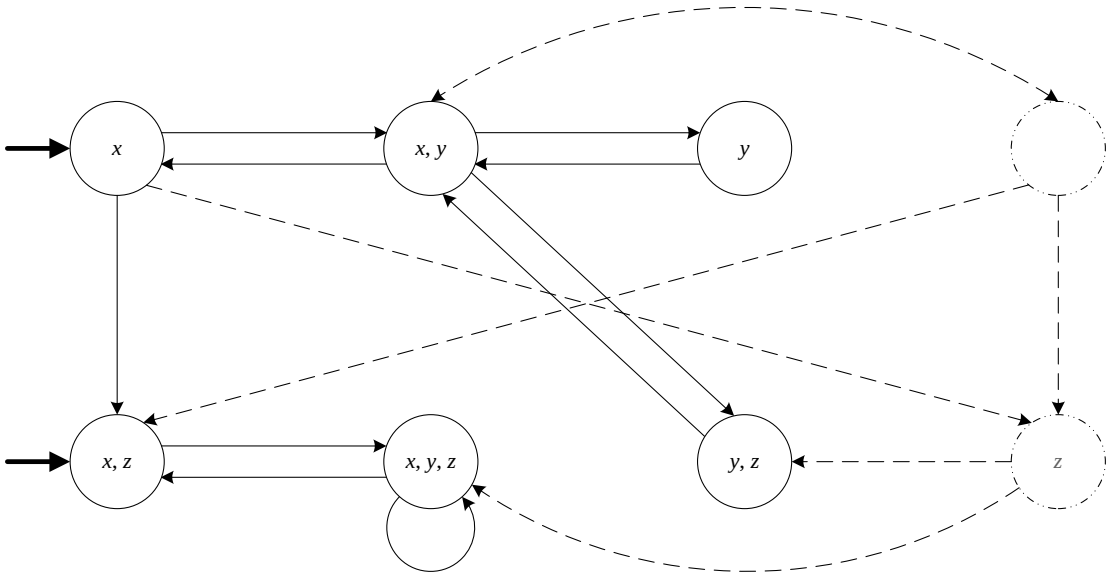


Рис. 2.8. Стартовые состояния таблицы

Каждая из подформул $\mathbf{F}\, \neg(x \vee z)$ и $\mathbf{F}(\neg(x \wedge y) \wedge z)$ задает класс состояний, удовлетворяющих такому условию: из каждого состояния результирующей модели есть путь в некоторое состояние этого класса. Обозначим классы номерами (рис. 2.9).

Покажем, что получилось на текущем этапе (уберем из иллюстрации все «пунктирные» детали). В классе $\mathbf{F}\,\neg(x \vee z)$ теперь будет только одно состояние, а в классе $\mathbf{F}(\neg(x \wedge y) \wedge z)$ – два (рис. 2.10).

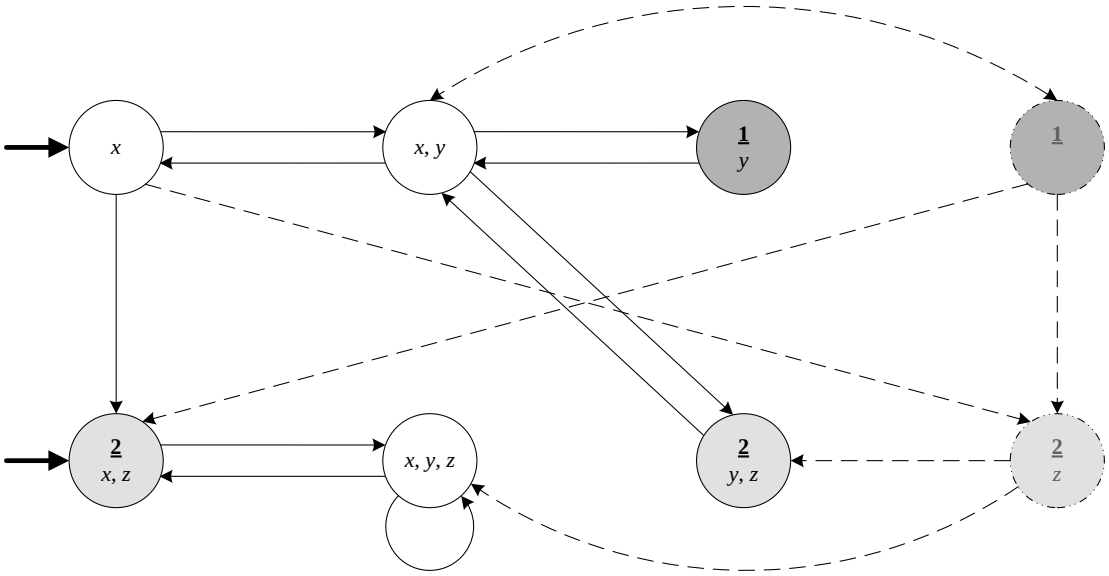


Рис. 2.9. Терминальные классы состояний

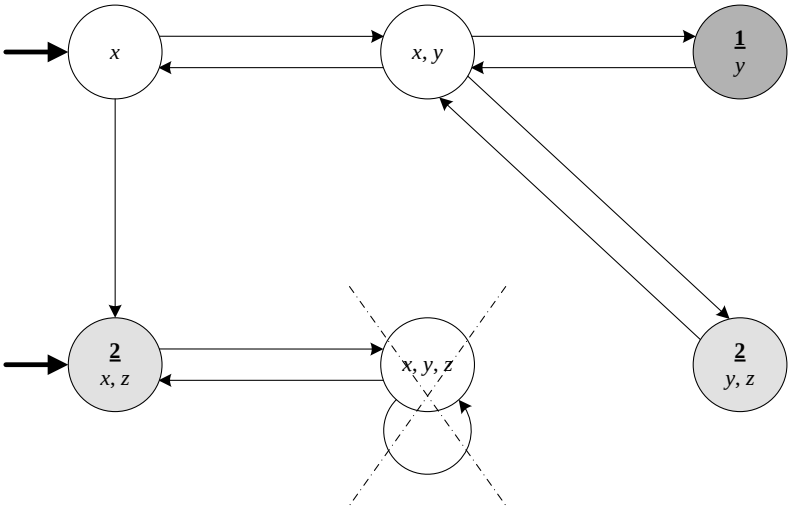


Рис. 2.10. Таблица после построения

Теперь выполним «чистку таблицы». Заметим, что из состояния (x, y, z) нельзя добраться до класса 1. Поэтому на рис. 2.10 оно зачеркнуто. Удалим его из модели (рис. 2.11).

Теперь заметим, что из состояния (x, z) нет переходов. Поэтому на рис. 2.11 оно зачеркнуто (есть еще одна причина, по которой его следует удалить: из него нельзя добраться в класс 1). Другие состояния остаются, так как из них всегда можно начать бесконечный путь и добраться как до класса 1, так и до класса 2. Модель, которая получается в результате, выглядит, как показано на рис. 2.12.

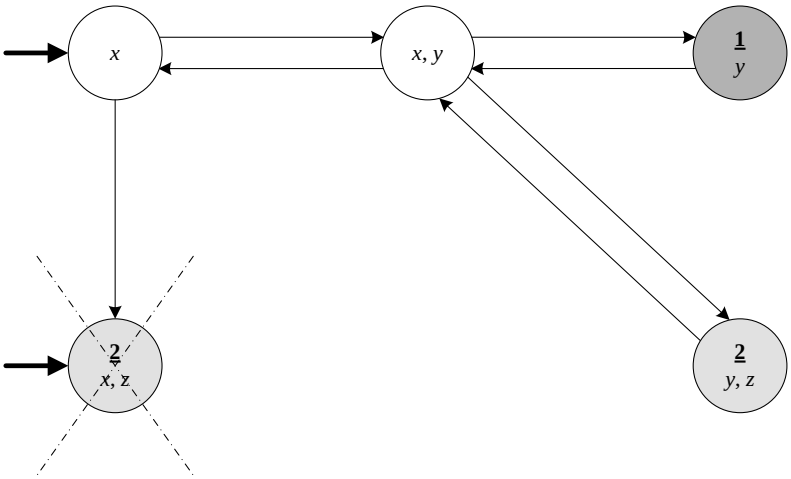


Рис. 2.11. Таблица в процессе чистки

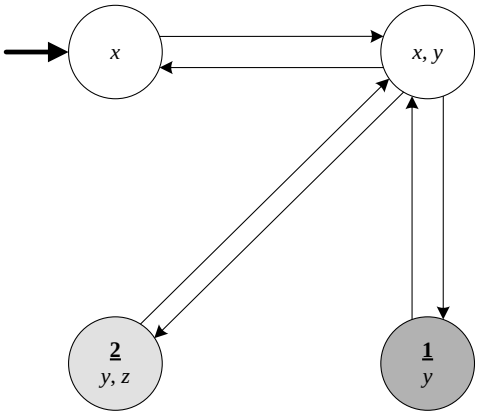


Рис. 2.12. Таблица после чистки

Пример. Рассматриваем формулу $\mathbf{G}(z_1 \to (\neg z_2\ \mathbf{W}\ z_3))$.

Переводим ее в нормальную форму:

$$\mathbf{G}((z_1 \to z_3 \vee \neg z_2 \wedge r) \wedge (r \to \mathbf{X}(z_3 \vee \neg z_2 \wedge r))).$$

Получается таблица, изображенная на рис. 2.13.

Все состояния таблицы являются стартовыми, и нет состояний, которые требуется посещать бесконечно часто. Множество состояний разделено на три блока из 2, 4 и 6 состояний, соответственно, и имеется еще одно состояние. Каждый блок представляет собой полный граф (*внутри* каждого блока из любого состояния есть переход в любое, включая петлю в себя). Стрелки на рисунке между блоками означают, что из каждого состояния блока, из которого исходит стрелка, есть переход в каждое состояние блока, в который она входит. Если имеется стрелка между блоком и верхним состоянием, то это означает, что все состояния этого блока соответствующим образом связаны с верхним состоянием.

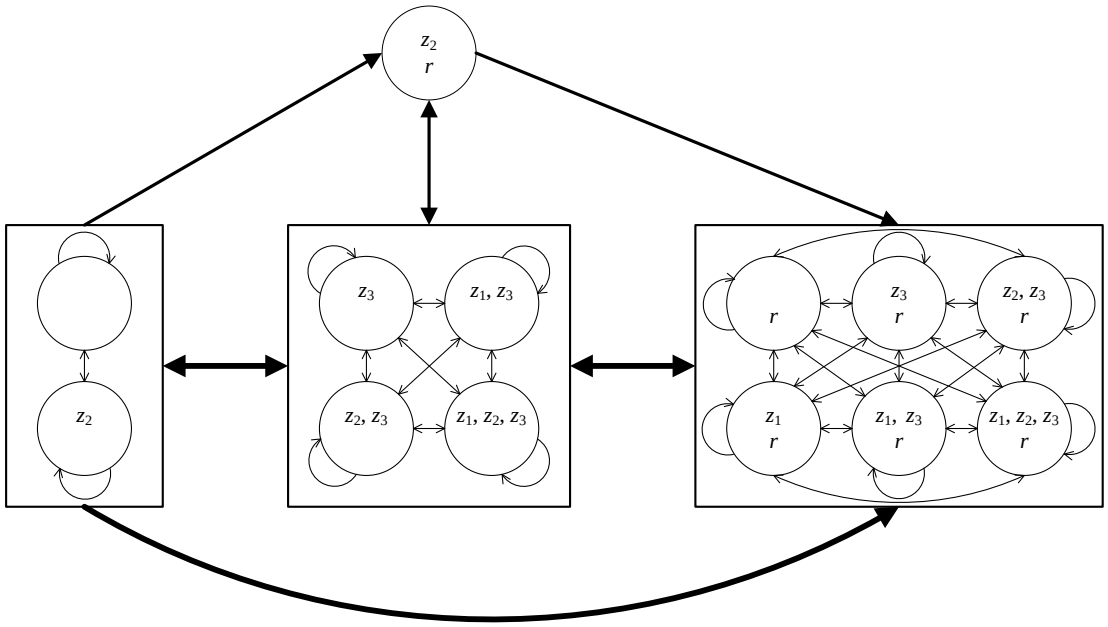


Рис. 2.13. Таблица для формулы $\mathbf{G}(z_1 \rightarrow (\neg z_2\ \mathbf{W}\ z_3))$

Таким образом, в наличии все переходы:

- *между состояниями* левого и среднего блоков;
- *между состояниями* среднего и правого блоков;
- *из состояний* левого блока *в состояния* правого;
- *между* верхним состоянием и средним блоком;
- *из состояний* левого блока в верхнее состояние;
- из верхнего состояния *в состояния* правого блока.

Никаких других переходов нет (в частности, нет перехода из верхнего состояния в себя).

Проверка моделей

Через $L_\omega(M)$ обозначим множество (бесконечных) путей модели M. Оно называется *языком модели*. Таблица, соответствующая формуле φ, обозначается T_φ. Имея такой ключевой результат, как построение таблиц, можно описать базовую схему проверки моделей для LTL-формул.

Наивный метод проверки, выполняется ли LTL-формула φ для заданной модели M, состоит в следующем: построим таблицу T_φ для формулы φ, а после этого проверим, верно ли $L_\omega(M) \subseteq L_\omega(T_\varphi)$. Допускающие вычисления системы M соответствуют возможному поведению модели, в то время как допускающие вычисления таблицы T_φ соответствуют желаемому поведению модели. Если любое возможное поведение является желаемым **(**если $L_\omega(M) \subseteq L_\omega(T_\varphi)$**)**, то

можно заключить, что модель удовлетворяет формуле φ. Данный подход кажется приемлемым. Однако задача проверки включения языков для моделей Крипке *PSPACE*-полна относительно своих входных данных. Это затрудняет проверку моделей таким методом.

Заметим, что $L_\omega(M) \subseteq L_\omega(T_\varphi) \Leftrightarrow (L_\omega(M) \cap L_\omega(\neg T_\varphi) = \varnothing)$, где $\neg T_\varphi$ (это дополнение T_φ) – такая модель, множеством путей которой является дополнение множества путей T_φ. Однако, наилучший известный алгоритм построения $\neg T_\varphi$ по таблице T_φ квадратично экспоненциален: если T_φ содержит n состояний, то $\neg T_\varphi$ содержит c^{n^2} состояний для некоторой константы $c > 1$. Это означает, что результирующая модель очень велика.

Используя свойство, состоящее в том, что дополняющая модель для T_φ эквивалентна модели для отрицания φ **(**$L_\omega(\neg T_\varphi) = L_\omega(T_{\neg\varphi})$**)**, получим следующий более эффективный метод проверки моделей LTL. Его идея состоит в том, чтобы построить таблицу для отрицания желаемого свойства φ. Таким образом, таблица $T_{\neg\varphi}$ моделирует нежелательные вычисления системы, которую требуется проверить. Если M содержит определенный допускающий путь, который также является допускающим путем $T_{\neg\varphi}$, то это пример вычисления, которое нарушает свойство φ. В этом случае можно сделать вывод о том, что φ не является свойством модели M. Если же нет такого общего вычисления, то φ выполняется. Это объясняет последний шаг следующего метода: построить таблицу T_φ для формулы φ и после этого проверить, верно ли $L_\omega(M) \cap L_\omega(T_{\neg\varphi}) = \varnothing$.

Для проверки последнего равенства строится декартово произведение моделей M и $T_{\neg\varphi}$. Пусть модель M имеет вид $(S, R, S_0, Label)$, а модель $T_{\neg\varphi}$ представляет собой четверку $(S_D, R_D, S_{0D}, Label_D)$, где S_{0D} – множество стартовых состояний, которых может быть несколько. Тогда модель-произведение $M' = (S', R', S_0', Label')$ строится следующим образом:

$$S' = \{(s, s_D) \mid s \in S, s_D \in S_D \text{ и } Label_D(s_D) \cap AP = Label(q)\};$$

$$R' = \{((s, s_D), (s', s_D')) \mid (s, s') \in R, (s_D, s_D') \in R_D\} \cap (S' \times S');$$

$$S_0' = \{(s_0, s_{0D}) \mid s_0 \in S_0, s_{0D} \in S_{0D}\} \cap S';$$

$$Label'(s, s_D) = Label_D(s_D).$$

Определим условия справедливости t_j для данной системы и проверим, существует ли в модели M' бесконечный путь, стартующий из начального состояния и удовлетворяющий условиям справедливости (эффективный алгоритм, решающий эту задачу, обсуждается в разд. 2.4). Если такой путь существует, то в M

существует путь, удовлетворяющий $\neg\varphi$. Этот путь является контрпримером для формулы φ.

Временная сложность данного алгоритма $O(|S| \times 2^{|\varphi|})$. Отметим, что задача проверки моделей для LTL является *PSPACE*-полной – для ее решения требуется размер памяти как минимум полиномиальный относительно размера входных данных [25].

2.2.7. Верификация LTL при помощи автоматов Бюхи

При проверке LTL-свойств могут использоваться различные способы представления LTL-формулы в виде графа переходов. Один из таких способов был рассмотрен в предыдущем разделе. Еще один, похожий способ, использует так называемые *автоматы Бюхи*.

Пусть AP – множество атомарных предложений. *Автоматом Бюхи* над алфавитом 2^{AP} называется четверка $A = (Q, q_0, \delta, F)$, где

- Q – конечное множество *состояний*;
- q_0 – *начальное состояние*;
- $\delta \subseteq Q \times 2^{AP} \times Q$ – *тотальное отношение переходов*;
- $F \subseteq Q$ – множество *допускающих* состояний.

Опишем алгоритм Герта, Пеледа, Варди и Волпера [16, 17] для построения автомата Бюхи по LTL-формуле. Введем новый темпоральный оператор **R** (Release), который определяется следующим образом:

$$\varphi \mathbf{R} \psi = \neg(\neg\varphi \mathbf{U} \neg\psi).$$

Для него верно, например, аналогичное *тождество расширения*:

$$\varphi \mathbf{R} \psi \equiv \psi \wedge (\varphi \vee \mathbf{X}(\varphi \mathbf{R} \psi)).$$

Для работы алгоритма требуется, чтобы LTL-формула была приведена в *негативную нормальную форму* – отрицание должно применяться только к атомарным предложениям.

Опишем метод приведения LTL-формулы к негативной нормальной форме.

1. Заменим все подформулы вида $\mathbf{F}\ \varphi$ на $\text{true}\ \mathbf{U}\ \varphi$.
2. Заменим все подформулы вида $\mathbf{G}\ \varphi$ на $\text{false}\ \mathbf{R}\ \varphi$.
3. Используя булевы тождества, оставим в формуле только три логические операции: $\neg$, $\vee$, $\wedge$.
4. Используя тождества LTL

- $\neg(\varphi \mathbf{U} \psi) \equiv \neg\varphi \mathbf{R} \neg\psi$,
- $\neg(\varphi \mathbf{R} \psi) \equiv \neg\varphi \mathbf{U} \neg\psi$ и
- $\neg\mathbf{X} \varphi \equiv \mathbf{X} \neg\varphi$,

погружаем отрицания внутрь темпоральных операторов.

Для алгоритма потребуются следующие структуры данных:

- *UID* – уникальный идентификатор;
- *Formula* – LTL-формула;
- *Node* – вершина графа переходов автомата Бюхи.

Для алгоритма неважно, в каком виде будут представлены уникальные идентификаторы и формулы, поэтому опишем только структуру *Node* (листинг 2.1):

Листинг 2.1. Структура *Node*

```
struct Node
{
  UID id;
  list<NodeID> incoming;
  list<Formula> old;
  list<Formula> new;
  list<Formula> next;
};
```

Здесь *incoming* – список вершин-предшественников (вершин, из которых идет дуга в текущую вершину). В полях *old*, *new* и *next* содержатся списки подформул исходной формулы.

Функция *CreateAutomaton* (листинг 2.2) строит граф переходов автомата Бюхи по формуле *f*.

Листинг 2.2. Функция *CreateAutomaton*

```
list<Node> CreateAutomaton (Formula f)
{
  Node n;
  n.incoming = {init};
  n.old = ∅;
  n.new = {f};
  n.next = ∅;
  return expand(n, ∅);
}
```

Добавление новой вершины выполняется функцией *Expand* (листинг 2.3):

Листинг 2.3. Функция *Expand*

```
list<Node> Expand (Node currentNode, list<Node> nodes)
{
```

```
if (currentNode.new == ∅)
{
  if (∃ Node r ∈ nodes: r.old == currentNode.old
                    && r.next == currentNode.next)
  {
    r.incoming = r.incoming ∪ currentNode.incoming;
    return nodes;
  }
  else
  {
    Node newNode;
    newNode.incoming = {currentNode};
    newNode.old = newNode.next = ∅;
    newNode.new = currentNode.next;
    Expand(newNode, nodes ∪ {currentNode});
  }
}
else   // currentNode.new не пуст.
{
  Выбрать Formula n из currentNode.new;
  currentNode.new = currentNode.new \ {n};
  if (n ∈ currentNode.old) Expand(currentNode, nodes);
  else
  {
    if (n == false or !n ∈ currentNode.old) return
            nodes;
    if (n ∈ AP or !n ∈ AP or n == true)
    // Замещение вершины.
    {
      node newNode;
      newNode.incoming = currentNode.incoming;
      newNode.old = currentNode.old ∪ {n};
      newNode.new = currentNode.new;
      newNode.next = currentNode.next;
      Expand(newNode, nodes);
    }
    if (n имеет вид f ∨ g)
    // Замещение вершины currentNode вершиной newNode.
    {
      node newNode1, newNode2;
      newNode1.incoming = currentNode.incoming;
      newNode1.old = currentNode.old ∪ {n};
      newNode1.new = currentNode.new ∪ {f};
      newNode1.next = currentNode.next;
      newNode2.incoming = currentNode.incoming;
      newNode2.old = currentNode.old ∪ {n};
      newNode2.new = currentNode.new ∪ {g};
      newNode2.next = currentNode.next;
      Expand(newNode2, Expand(newNode1, nodes));
    }
```

```
  if (n имеет вид f U g)
  // f U g ⇔ g ∨ (f ∧ X (f U g)).
  // Расщепление на две вершины.
  {
    node newNode1, newNode2;
    newNode1.incoming = currentNode.incoming;
    newNode1.old = currentNode.old ∪ {n};
    newNode1.new = currentNode.new ∪ {f};
    newNode1.next = currentNode.next ∪ {f U g};
    newNode2.incoming = currentNode.incoming;
    newNode2.old = currentNode.old ∪ {n};
    newNode2.new = currentNode.new ∪ {g};
    newNode2.next = currentNode.next;
    Expand(newNode2, expand(newNode1, nodes));
  }
  if (n имеет вид f R g)
  // f R g ⇔ g ∧ (f ∨ X (f R g)).
  // Расщепление на две вершины.
  {
    node newNode1, newNode2;
    newNode1.incoming = currentNode.incoming;
    newNode1.old = currentNode.old ∪ {n};
    newNode1.new = currentNode.new ∪ {f};
    newNode1.next = currentNode.next;
    newNode2.incoming = currentNode.incoming;
    newNode2.old = currentNode.old ∪ {n};
    newNode2.new = currentNode.new ∪ {f, g};
    newNode2.next = currentNode.next ∪ {f R g};
    Expand(newNode2, expand(newNode1, nodes));
  }
  if (n имеет вид f ∧ g)
  // Замещение вершины currentNode вершиной newNode.
  {
    node newNode;
    newNode.incoming = currentNode.incoming;
    newNode.old = currentNode.old ∪ {n};
    newNode.new = currentNode.new ∪ {f, g};
    newNode.next = currentNode.next;
    Expand(newNode, nodes);
  }
  if (n имеет вид X f)
  // Замещение вершины currentNode вершиной newNode.
  {
    node newNode;
    newNode.incoming = currentNode.incoming;
    newNode.old = currentNode.old ∪ {n};
    newNode.new = currentNode.new;
    newNode.next = currentNode.next ∪ {f};
    Expand(newNode, nodes);
  }
}
```

```
    }
  }
```

Пример. Рассмотрим первые несколько шагов алгоритма для формулы $\mathbf{F}((p\ \mathbf{R}\ q) \to r)$. Приведем формулу к негативной нормальной форме:

$$\mathbf{F}((p\ \mathbf{R}\ q) \to r) \equiv \text{true}\ \mathbf{U}\ ((p\ \mathbf{R}\ q) \to r) \equiv$$

$$\equiv \text{true}\ \mathbf{U}\ (\neg(p\ \mathbf{R}\ q) \vee r) \equiv \text{true}\ \mathbf{U}\ ((\neg p\ \mathbf{U}\ \neg q) \vee r).$$

Таким образом, на вход алгоритму подается формула $f = \text{true}\ \mathbf{U}\ ((\neg p\ \mathbf{U}\ \neg q) \vee r)$. Создаем вершину *init* и вершину *n*. После этого создадим для вершины *n* пустые списки *old* и *next* и список *new*, содержащий формулу *f*. Далее создаем пустой список *nodes*, в котором будут храниться вершины построенного графа переходов автомата Бюхи.

Запустим функцию *Expand* и извлечем формулу *f* из списка *n.new*. Формула *f* имеет вид $a\ \mathbf{U}\ b$. Поэтому расщепим вершину *n* на две вершины: n_1 и n_2 (табл. 2.1).

Таблица 2.1. Расщепление вершины *n* на вершины n_1 и n_2

	n	$\boldsymbol{n_1}$	$\boldsymbol{n_2}$
incoming	*init*	*init*	*init*
old	$\varnothing$	f	f
next	$\varnothing$	f	$\varnothing$
new	f	true	$\text{true}\ \mathbf{U}\ ((\neg p\ \mathbf{U}\ \neg q) \vee r)$

Далее применяем функцию *Expand* сначала к вершине n_1, а после этого к построенному этой же функцией списку и вершине n_2.

При извлечении из списка n_1.*new* формулы true вершина n_1 замещается вершиной n_3 (табл. 2.2).

Таблица 2.2. Замещение вершины n_1 вершиной n_3

	$\boldsymbol{n_1}$	$\boldsymbol{n_3}$
incoming	*init*	*init*
old	f	$\{f, \text{true}\}$
next	f	f
new	true	$\varnothing$

Список new вершины n_3 пуст. Так как список *nodes* пуст и не содержит вершину *r*, у которой с текущей вершиной совпадают

списки *old* и *next*, добавляем в список *nodes* вершину n_3. Создаем новую вершину n_4 (табл. 2.3):

Таблица 2.3. Вершина n_4

	n
incoming	*init*
old	∅
next	∅
new	*f*

Дальнейшие итерации выполняются тем же способом.

Проверка моделей с использованием автоматов Бюхи

Пусть даны модель Крипке и LTL-формула, выполнение которой на модели требуется проверить. Общая идея алгоритма следующая:

- Из отрицания LTL-формулы строится эквивалентный ей автомат Бюхи.
- Модель Крипке также преобразуется в автомат Бюхи.
- Строится третий автомат Бюхи как пересечение первых двух. Такой автомат будет допускать пути исходной модели, которые не удовлетворяют LTL-формуле спецификации.
- Если язык, допускаемый построенным автоматом-пересечением, пуст, то верификация успешна. Если нет, то путь, допускаемый автоматом-пересечением, является контрпримером.

Сначала строится автомат Бюхи, соответствующий верифицируемой LTL-формуле. Напомним, что он собой представляет. Автомат Бюхи – это конечный автомат над бесконечными словами. Переходы автомата Бюхи помечены предикатами из исходной формулы LTL. Автомат работает следующим образом. На каждом шаге он берет очередной набор значений предикатов из последовательности, отражающей историю работы программы. Используя эти значения, автомат вычисляет метки на переходах из текущего состояния. Активными называются те переходы, метки которых были вычислены как true. Если активных переходов больше одного, автомат недетерминированно выбирает один из них. Таким образом, последовательность значений предикатов определяет возможные наборы переходов в автомате Бюхи.

Пример автомата Бюхи, полученного из формулы LTL, изображен на рис. 2.14. Формула «**GF** *p*» читается как «всегда когда-нибудь *p*» или «*p* будет выполняться бесконечно часто».

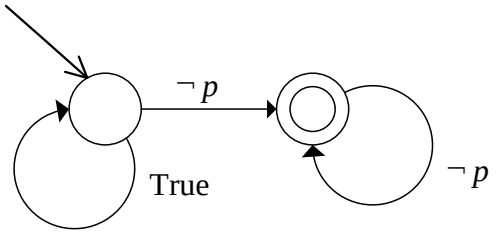


Рис. 2.14. Автомат Бюхи для формулы «**GF** *p*»

В автомате Бюхи имеется набор допускающих состояний. На рис. 2.14 такое состояние отмечено двойным кругом. Если для некоторой последовательности значений предикатов возможна такая последовательность переходов в автомате Бюхи, что допускающие состояния посещаются бесконечно часто, то говорят, что автомат допускает эту последовательность значений предикатов.

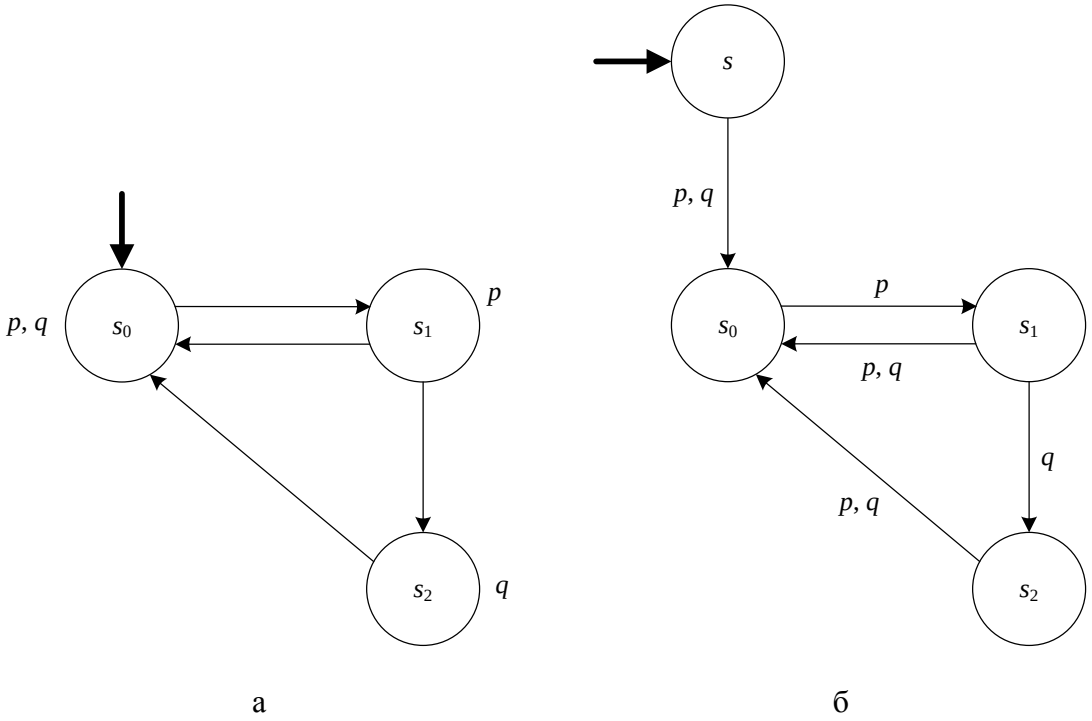
Формула LTL и автомат Бюхи, построенный из нее, эквивалентны в том смысле, что если некоторый путь (история работы программы) удовлетворяет формуле, то он допускается автоматом Бюхи. И наоборот, любая последовательность значений предикатов, допускаемая автоматом Бюхи, удовлетворяет исходной формуле LTL.

Поскольку задача верификатора – найти контрпример, автомат Бюхи строится для отрицания исходной формулы LTL. Таким образом, автомат Бюхи допускает любые последовательности значений предикатов, которые не удовлетворяют требованиям.

Для пояснения рассмотрим автомат, изображенный на рис 2.14. Если существует момент времени, после которого *p* ни разу не выполнится, то автомат в этот момент перейдет в допускающее состояние и останется там навсегда. Таким образом, будет получен допускающий путь. Автомат Бюхи на рис. 2.14 недетерминирован: возможны два перехода из недопускающего состояния при невыполнении *p*. Кроме того, часто автомат строится неполным: например, если автомат на рис. 2.14 находится в допускающем состоянии и выполнено *p*, то не существует активного перехода. В этом случае считается, что вычисление не допускается. Действительно, при выполнении *p* нарушится предположение о том, что, начиная с выбранного момента времени, *p* не выполнится ни разу.

Следующий этап для верификации LTL-формулы – это преобразование модели Крипке в автомат Бюхи. Пример модели Крипке и автомата, полученного из нее, изображен на рис. 2.15.

При построении автомата из модели Крипке вместо состояний предикатами помечаются переходы. Это делается достаточно просто: добавляется одно начальное состояние, из которого создаются



а б

Рис. 2.15. Пример модели Крипке (а) и ее преобразования в автомат (б)

переходы в начальные состояния исходной модели Крипке. Затем каждый переход помечается теми предикатами, которые выполняются в конечном состоянии перехода в исходной модели Крипке. Таким образом, предикаты переносятся из состояний на переходы. Автомат Бюхи, полученный в результате, является разновидностью автомата Мура [17]: метки на переходах зависят только от конечного состояния перехода.

Любой путь в построенном автомате является одним из возможных сценариев работы модели Крипке. Поэтому все состояния автомата помечаются как допускающие. В результате построенный автомат допускает любую последовательность предикатов, соответствующую возможному сценарию работы верифицируемой модели. В то же время любая допускаемая автоматом последовательность является одним из сценариев работы модели.

После построения автомата по модели Крипке и автомата Бюхи, создается автомат-пересечение, который, по определению, допускает только те последовательности предикатов, которые допускаются и автоматом Бюхи, и моделью Крипке. Построенное пересечение автоматов также является автоматом Бюхи. При этом если для него существует допускающий путь, то такой путь:

- является одним из возможных сценариев работы модели;
- нарушает верифицируемую LTL-формулу спецификации.

Таким образом, если доказать, что не существует допускающего пути в построенном пересечении автоматов, то будет доказано, что модель удовлетворяет спецификации. С другой стороны, если будет найден допускающий путь, то он будет экземпляром сценария работы модели, нарушающего спецификацию. Другими словами, будет предъявлен контрпример.

Теперь покажем, что структура пути, допускаемого автоматом Бюхи, имеет вид $\alpha\beta^{\omega}$, где α – некоторый префикс, а β – суффикс, порождаемый проходом автомата Бюхи по циклу, содержащему допускающее состояние. Структура пути изображена на рис. 2.16.

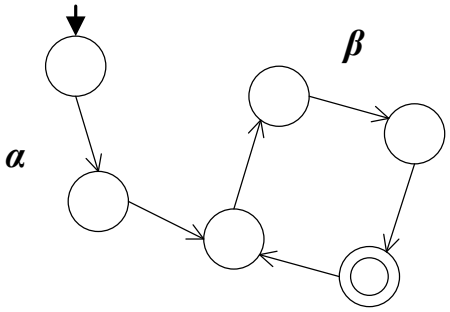


Рис. 2.16. Структура допускаемого пути в автомате Бюхи

Пусть p – путь, допускаемый автоматом Бюхи. Поскольку множество состояний автомата является конечным, найдется суффикс p' пути p такой, что всякое состояние в нем встречается бесконечное число раз. Это означает, что любое состояние этого суффикса достижимо из любого другого состояния суффикса. Следовательно, состояния из p' входят в состав некоторой сильно связной компоненты графа, описывающего автомат Бюхи. Причем, так как одно из допускающих состояний также входит в суффикс p', 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 $\alpha\beta^{\omega}$, где α – префикс, который ведет к сильно связной компоненте, а β – цикл в этой компоненте, содержащий допускающее состояние.

Таким образом, проверка пустоты языка автомата Бюхи равносильна поиску бесконечного пути, стартующего в начальном состоянии и удовлетворяющего специальному *условию справедливости*: в нем бесконечно часто должно встречаться какое-либо из допускающих состояний. Соответствующие алгоритмы основаны на поиске сильно связной компоненты, достижимой из начального состояния и содержащей допускающее состояние, и будут описаны в разд. 2.4.

Применение на практике

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

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

- Верификатор работает с программой и с автоматом Бюхи, построенным по отрицанию темпоральной формулы.
- Глобальное состояние верифицируемой системы складывается из глобального состояния программы и состояния, в котором находится автомат Бюхи.
- Каждый элементарный шаг совершается в два хода: сначала происходит элементарный переход в программе. После этого вычисляются необходимые предикаты и совершается переход в автомате Бюхи. Если в автомате Бюхи невозможно выполнить ни один переход, то происходит откат: в данной ветви истории не удалось найти контрпример.
- Если в некотором состоянии возможен более чем один переход в программе или в автомате Бюхи, то верификатор запоминает текущее состояние верифицируемой системы и набор возможных переходов. При откате в такое состояние исполнение программы будет запущено снова вперед, но уже по другим возможным переходам. Если же все возможные переходы ранее были пройдены – верификатор откатывает глобальное состояние дальше назад. Таким образом, в каждой ситуации недетерминированности будут перебираться все возможные варианты работы верифицируемой системы.
- Считается, что верифицируемая система попала в допускающее состояние, когда автомат Бюхи находится в допускающем состоянии.

Такая схема действий позволяет работать по алгоритму поиска справедливых путей без явного построения пересечения автомата модели Крипке с автоматом Бюхи.

2.3. Проверка моделей для ветвящейся темпоральной логики

А. Пнуэли ввел темпоральную логику в информатику для спецификации и верификации реактивных систем [15]. Выше был рассмотрен один важный представитель темпоральной логики – LTL. Эта логика называется *линейной*, так как при ее использовании качественное представление о времени считается линейным: в каждый момент времени существует только одно состояние-потомок, а поэтому только одно будущее. Формально говоря, это следует из того факта, что интерпретация формул темпоральной логики, использующая отношение выполняемости $\models$, определена в терминах модели, в которой состояние s имеет ровно одного потомка $R(s)$. Таким образом, для каждого состояния s модель порождает уникальную бесконечную последовательность состояний s, $R(s)$, $R(R(s))$, … Последовательность состояний представляет вычисление. Так как семантика линейной темпоральной логики базируется на таких «порождающих последовательность» моделях, темпоральные операторы **X**, **U**, **F** и **G** фактически описывают порядок событий вдоль *одного* временного пути (одного вычисления системы).

В начале 80-х годов для целей спецификации и верификации был предложен другой тип темпоральной логики, который базируется не на линейном, а на ветвящемся представлении о времени. Эта логика формально базируется на моделях, в которых в каждый момент может быть несколько различных возможных будущих. Ввиду такого ветвящегося представления о времени, данный класс темпоральных логик называется *ветвящимися темпоральными логиками*. Строго говоря, ветвление сводится к тому факту, что у состояния может быть несколько различных допустимых потомков. Следовательно, $R(s)$ – это (непустое) множество состояний, а не одно состояние, как в LTL. Представление о семантике ветвящейся темпоральной логики, таким образом, базируется на *дереве* состояний вместо последовательности. Каждый путь в таком дереве должен представлять одно возможное вычисление. Дерево само по себе представляет все возможные вычисления. Более точно, дерево, подвешенное в состоянии s, представляет все возможные бесконечные вычисления, которые стартуют в состоянии s.

Темпоральные операторы в ветвящейся темпоральной логике позволяют выражать свойства (всех или некоторых) вычислений в системе. Например, свойство **EF** φ обозначает, что существует вычисление, вдоль которого выполняется **F** φ. Суть этого свойства состоит в том, что существует как минимум одно возможное вычисление, в котором, в конечном счете, достигается состояние, выполняющее φ. Это, однако, не исключает того факта, что могут существовать вычисления, для которых это свойство не выполняется – вычисления, в которых никогда не выполняется φ. Свойство **AF** φ, например, отличается от этого экзистенциального свойства вычислений тем, что оно требует, чтобы все вычисления удовлетворяли свойству **F** φ.

Существование двух типов темпоральной логики – линейной и ветвящейся – привело к развитию двух «школ» проверки моделей. Несмотря на достоинства и недостатки каждой из них, имеются две причины, которые оправдывают рассмотрение в данной книге проверки моделей как для линейной, так для и ветвящейся темпоральной логики:

- *Выразительная сила* многих линейных и ветвящихся темпоральных логик несравнима. Это означает, что некоторые свойства, выразимые в линейной темпоральной логике, не могут быть выражены в определенной ветвящейся темпоральной логике и наоборот.
- Традиционные *методы*, используемые для эффективной проверки моделей для линейной темпоральной логики, сильно отличаются от методов, применяемых для ветвящейся темпоральной логики. Это приводит, в том числе, и к совершенно различным оценкам сложности.

В данном разделе рассматривается проверка моделей для *ветвящейся темпоральной логики* CTL (*Computational Tree Logic*). Важно, что она может быть рассмотрена как ветвящийся аналог LTL, для которого возможна эффективная проверка моделей.

Раздел излагается на основе курса [1] и содержит примеры из [4, 26].

2.3.1. Синтаксис CTL

Синтаксис ветвящейся темпоральной логики (логики деревьев вычислений) определяется следующим образом. Элементарными выражениями в CTL являются *атомарные предложения*, как и в определении LTL. Будем определять синтаксис CTL в нотации

Бэкуса-Наура (*p* принадлежит множеству атомарных предложений *AP*):

$$\varphi ::= p \mid \neg\varphi \mid (\varphi \vee \varphi) \mid \mathbf{EX}\,\varphi \mid \mathbf{E}[\varphi\ \mathbf{U}\ \varphi] \mid \mathbf{A}[\varphi\ \mathbf{U}\ \varphi].$$

Используются темпоральные операторы следующего вида:

- **EX** (в следующий момент для некоторого пути);
- **E** (для некоторого пути);
- **A** (для всех путей);
- **U** (до тех пор).

Здесь **X** и **U** – линейные темпоральные операторы, которые выражают свойства на фиксированном пути, в то время как экзистенциальный квантификатор пути **E** выражает свойство на некотором пути, а универсальный квантификатор пути **A** – свойство на всех путях. Квантификаторы пути **E** и **A** могут быть использованы только в комбинации с **X** или **U**. Отметим, что оператор **AX** неэлементарен и определяется ниже.

Булевы операторы true, false, $\wedge$, $\rightarrow$ и $\leftrightarrow$ определяются как обычно. Из равенства $\mathbf{F}\ \varphi$ = true $\mathbf{U}\ \varphi$ следуют два сокращения:

$\mathbf{EF}\ \varphi = \mathbf{E}[\text{true}\ \mathbf{U}\ \varphi]$;

$\mathbf{AF}\ \varphi = \mathbf{A}[\text{true}\ \mathbf{U}\ \varphi]$.

EF φ произносится как «φ выполняется потенциально», а **AF** φ – как «φ неизбежно». Так как $\mathbf{G}\ \varphi \equiv \neg\mathbf{F}\ \neg\varphi$ и $\mathbf{A}\ \varphi \equiv \neg\mathbf{E}\ \neg\varphi$, то дополнительно[3]:

$\mathbf{EG}\ \varphi = \neg\mathbf{AF}\ \neg\varphi$;

$\mathbf{AG}\ \varphi = \neg\mathbf{EF}\ \neg\varphi$;

$\mathbf{AX}\ \varphi = \neg\mathbf{EX}\ \neg\varphi$.

Например, справедливо:

$\neg\mathbf{A}(\mathbf{F}\ \neg\varphi)$

$\Leftrightarrow$ { $\mathbf{A}\ \varphi \equiv \neg\mathbf{E}\ \neg\varphi$ }

$\neg\neg\mathbf{E}\ \neg(\mathbf{F}\ \neg\varphi)$

$\Leftrightarrow$ { $\mathbf{G}\ \varphi \equiv \neg(\mathbf{F}\ \neg\varphi)$; вычисления }

$\mathbf{EG}\ \varphi$.

---

[3] Равенство $\mathbf{A}\ \varphi \equiv \neg\mathbf{E}\ \neg\varphi$ выполняется только для таких CTL-формул φ, в которых снят внешний экзистенциальный или универсальный квантификатор пути. Это позволяет, например, переписать строку $\mathbf{E}[\neg\mathbf{F}\ \neg\varphi]$ в формулу $\neg\mathbf{AF}\ \neg\varphi$.

EG φ произносится «потенциально всегда φ», **AG** φ – «безусловно φ», а **AX** φ – «для всех путей в следующий момент φ».

Пример. Пусть $AP = \{x = 1,\ x < 2,\ x \geq 3\}$ – множество атомарных предложений.

- Примерами CTL-формул являются: **EX**$(x = 1)$, **AX**$(x = 1)$, $x < 2 \vee x = 1$, **E**$[(x < 2)$**U**$(x \geq 3)]$ и **AF**$(x < 2)$.
- Выражение **E**$[x = 1 \wedge$ **AX**$(x \geq 3)]$ не является CTL-формулой, так как $x = 1 \wedge$ **AX**$(x \geq 3)$ не является **X**- или **U**-формулой. Выражение **EF**[**G**$(x = 1)]$ также не является CTL-формулой. **EG**$[x = 1 \wedge$ **AX**$(x \geq 3)]$ – это, однако, правильно построенная CTL-формула. Также **EF**[**EG**$(x = 1)]$ и **EF**[**AG**$(x = 1)]$ – это правильно построенные CTL-формулы.

Синтаксис CTL требует, чтобы линейным темпоральным операторам **X**, **F**, **G** и **U** немедленно предшествовали квантификаторы пути **E** или **A**. Если это ограничение опустить, то получится более выразительная ветвящаяся темпоральная логика CTL*.

Логика CTL* позволяет квантификаторам **E** и **A** предшествовать любой LTL-формуле. Она содержит, например, **E**$[p \wedge$ **X** $q]$ и **F** $p \wedge$ **G** $q]$ – формулы, которые не являются синтаксическими термами CTL. Логика CTL* может быть рассмотрена как ветвящийся аналог логики LTL, так как каждая подформула LTL может быть использована в CTL-формуле. Точные взаимоотношения между LTL, CTL и CTL* будут рассмотрены ниже. Несмотря на то, что CTL не обладает выразительной силой CTL*, рассмотрение большого числа примеров показало, что она часто достаточно выразительна для формулировки большинства требуемых свойств.

2.3.2. Семантика CTL

Как было отмечено в соответствующем разделе, интерпретация линейной темпоральной логики LTL определяется в терминах модели $M = (S, R, Label)$, где S – множество состояний, $Label$ – назначение атомарных предложений состояниям, а R – тотальная функция, которая каждому заданному состоянию ставит в соответствие единственное состояние-потомок. Так как потомок $R(s)$ состояния s только один, модель M порождает для каждого состояния s последовательность состояний s, $R(s)$, $R(R(s))$, … Эти последовательности представляют вычислительные пути, которые начинаются в s, а так как LTL-формула обращается к одному пути, интерпретация LTL определена в терминах таких последовательностей.

Ветвящаяся темпоральная логика, однако, обращается не к одному вычислительному пути, а к некоторым (или всем) вычислительным путям. Одной последовательности, таким образом, недостаточно для того, чтобы это промоделировать. С целью адекватно представить моменты, в которых возможно ветвление, понятие последовательности было заменено понятием *дерева*. Соответственно, *CTL-модель* – модель, порождающая дерево. Формально, CTL-модель является *моделью Крипке*, так как Саул Крипке использовал сходные структуры с целью дать семантику модальной логике, типу логики, который тесно связан с темпоральной логикой [27].

Отметим, что единственное различие с LTL-моделью в том, что R теперь – *тотальное отношение* вместо тотальной функции.

Пример. Пусть $AP = \{x = 0,\ x = 1,\ x \neq 0\}$ – множество атомарных предложений, $S = \{s_0, \ldots, s_3\}$ – множество состояний с пометками

$$Label(s_0) = \{x \neq 0\},$$

$$Label(s_1) = Label(s_2) = \{x = 0\},$$

$$Label(s_3) = \{x = 1, x \neq 0\},$$

а отношение переходов R следующее:

$$R = \{(s_0, s_1), (s_1, s_2), (s_1, s_3), (s_3, s_3), (s_2, s_3), (s_3, s_2)\}.$$

Рассмотрим CTL-модель $M = (S, R, Label)$. Она изображена на рис. 2.17 (а). Здесь состояния изображены вершинами, а отношение R обозначено стрелками: стрелка из s в s' присутствует, если и только если $(s, s') \in R$. Пометки $Label(s)$ указаны рядом с состоянием s.

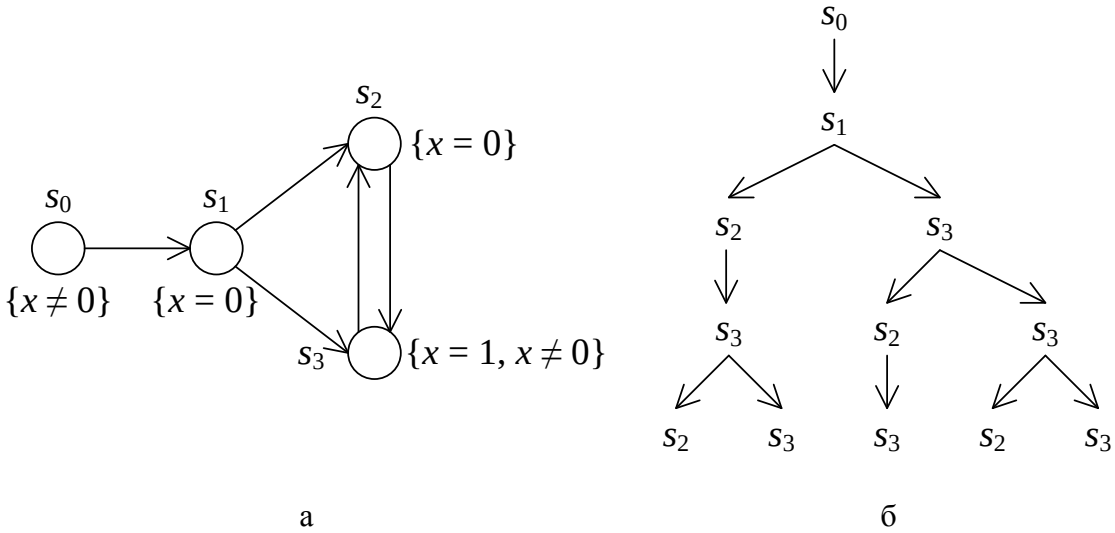
Перед описанием семантики введем несколько вспомогательных понятий. Пусть $M = (S, R, Label)$ – CTL-модель.

Напомним, что *путь* – это бесконечная последовательность состояний $s_0 s_1 s_2 \ldots$, такая что $(s_i, s_{i+1}) \in R$ для всех $i \geq 0$.

Пусть σ обозначает путь (из состояний). Для $i \geq 0$ через $\sigma[i]$ обозначим $(i + 1)$-й элемент σ, а через σ^i – *суффикс пути* σ, начинающийся с этого элемента. Например, если $\sigma = t_0 t_1 t_2 \ldots$, то $\sigma[i] = t_i$ (где t_i – состояние), а $\sigma^i = t_i t_{i+1} t_{i+2} \ldots$

Множество путей, начинающихся в состоянии s модели M, определяется следующим образом: $P_M(s) = \{\sigma \in S^\omega \mid \sigma[0] = s\}$.

Для любой CTL-модели $M = (S, R, Label)$ и состояния $s \in S$ имеется бесконечное вычислительное дерево с корнем, помеченным s, таким что (s', s'') – дуга в дереве, если и только если $(s', s'') \in R$. Состояние s, для которого $p \in Label(s)$, иногда называется p-состоянием. Путь σ называется p-путем, если он состоит только из p-состояний.



а б

Рис. 2.17. Пример CTL-модели (а)
и префикс одного из ее вычислительных деревьев (б)

Пример. Рассмотрим CTL-модель на рис. 2.17 (а). Конечный префикс бесконечного вычислительного дерева, подвешенного в состоянии s_0, изображен на рис. 2.17 (б). Примерами путей являются $s_0 s_1 s_2 s_3^{\omega}$, $s_0 s_1 (s_2 s_3)^{\omega}$ и $s_0 s_1 (s_3 s_2)^{*} s_3^{\omega}$. Множество $P_M(s_3)$, например, можно описать таким способом: $\{(s_3 s_2)^{*} s_3^{\omega}, (s_3^{+} s_2)^{\omega}\}$.

Семантика CTL определяется в терминах *отношения выполняемости* (обозначаемого $\models$) между моделью M, одним из ее состояний s и формулой φ. Как и ранее, запишем $M, s \models \varphi$ вместо $(M, s, \varphi) \in \models$. При этом имеем $M, s \models \varphi$ тогда и только тогда, когда φ верно в состоянии s модели M. Как и ранее, будем опускать M, когда модель ясна из контекста.

Пусть $p \in AP$ – атомарное предложение, а тройка $M = (S, R, Label)$ – CTL-модель, $s \in S$ и φ, ψ – CTL-формулы. Отношение выполняемости $\models$ определяется следующим образом:

$$
\begin{aligned}
s &\models p && \Leftrightarrow p \in Label(s);\\
s &\models \neg\varphi && \Leftrightarrow \neg(s \models \varphi);\\
s &\models (\varphi \vee \psi) && \Leftrightarrow (s \models \varphi) \vee (s \models \psi);\\
s &\models \mathbf{EX}\,\varphi && \Leftrightarrow \exists \sigma \in P_M(s)\colon \sigma[1] \models \varphi;\\
s &\models \mathbf{E}[\varphi\ \mathbf{U}\ \psi] && \Leftrightarrow \exists \sigma \in P_M(s)\colon (\exists j \geq 0\colon \sigma[j] \models \psi \wedge\\
& && \quad \wedge (\forall 0 \leq k < j\colon \sigma[k] \models \varphi));\\
s &\models \mathbf{A}[\varphi\ \mathbf{U}\ \psi] && \Leftrightarrow \forall \sigma \in P_M(s)\colon (\exists j \geq 0\colon \sigma[j] \models \psi \wedge\\
& && \quad \wedge (\forall 0 \leq k < j\colon \sigma[k] \models \varphi)).
\end{aligned}
$$

Интерпретации атомарных предложений, отрицания и конъюнкции стандартны.

EX φ верно в состоянии s, если и только если существует путь σ, начинающийся в состоянии s, такой, что в следующем состоянии этого пути $\sigma[1]$ выполняется свойство φ.

A[φ **U** ψ] верно в состоянии s, если и только если каждый путь, начинающийся в s, имеет в начале конечный префикс (возможно, состоящий только из s) такой, что ψ выполняется в последнем состоянии этого префикса и φ выполняется во всех состояниях префикса перед s.

E[φ **U** ψ] верно в состоянии s, если и только если существует путь, начинающийся в s, который удовлетворяет свойству φ **U** ψ.

Интерпретация темпоральных операторов **AX** φ, **EF** φ, **EG** φ, **AF** φ и **AG** φ может быть выведена с использованием указанного выше определения. Чтобы это проиллюстрировать, выведем формальную семантику **EG** φ.

$$s \models \mathbf{EG}\ \varphi$$

$\Leftrightarrow$ { по определению **EG** }

$$s \models \neg\mathbf{AF}\ \neg\varphi$$

$\Leftrightarrow$ { по определению **AF** }

$$s \models \neg\mathbf{A}[\text{true}\ \mathbf{U}\ \neg\varphi]$$

$\Leftrightarrow$ { семантика $\neg$ }

$$\neg(s \models \mathbf{A}[\text{true}\ \mathbf{U}\ \neg\varphi])$$

$\Leftrightarrow$ { семантика **A**[φ **U** ψ] }

$$\neg[\forall\sigma \in P_M(s)\colon (\exists j \geq 0\colon \sigma[j] \models \neg\varphi \wedge (\forall 0 \leq k < j\colon \sigma[k] \models \text{true}))]$$

$\Leftrightarrow$ { $s \models$ true для всех состояний s }

$$\neg[\forall\sigma \in P_M(s)\colon (\exists j \geq 0\colon \sigma[j] \models \neg\varphi)]$$

$\Leftrightarrow$ { семантика $\neg$; вычисления }

$$\exists\sigma \in P_M(s)\colon \neg(\exists j \geq 0\colon \neg(\sigma[j] \models \varphi))$$

$\Leftrightarrow$ { вычисления }

$$\exists\sigma \in P_M(s)\colon (\forall j \geq 0\colon \sigma[j] \models \varphi).$$

Следовательно, **EG** φ верно в состоянии s, если и только если существует некоторый путь, начинающийся в s, такой, что для каждого состояния на этом пути выполняется свойство φ.

Также можно вывести, что **AG** φ верно в состоянии s, если и только если для всех состояний на любом пути, начинающемся в s, выполняется свойство φ.

При этом **EF** φ верно в состоянии s, если и только если φ в конечном счете выполняется вдоль некоторого пути, стартующего в s, а **AF** φ верно, если и только если это свойство выполняется для всех путей, стартующих в s.

Пример. Рассмотрим CTL-модель M, изображенную на рис. 2.18. На этом рисунке показана справедливость нескольких формул для всех состояний M. Состояние отмечено черным цветом, если формула верна для него, и белым в противном случае. При этом:

- Формула **EX** p верна для всех состояний, так как у всех состояний есть прямой потомок, удовлетворяющий p.
- **AX** p неверна в состоянии s_0, так как возможный путь, начинающийся в s_0, идет прямо в состояние s_2, в котором p не выполняется. Так как остальные состояния имеют только прямых потомков, в которых p выполняется, **AX** p верно в остальных состояниях.
- Для всех состояний, кроме s_2, существует вычисление (например, $s_0 s_1 s_3^{\omega}$), для которого p глобально верно. Следовательно, **EG** p верно в этих состояниях. Ввиду того, что $p \notin Label(s_2)$, путь, начинающийся в s_2, для которого p глобально верно, отсутствует.
- **AG** p верно только в состоянии s_3, так как его единственный путь s_3^{ω} всегда проходит через состояния, в которых p выполняется. Для всех остальных состояний существует путь, содержащий состояние s_2, которое не удовлетворяет p. Следовательно, для этих состояний **AG** p неверно.
- **EF**(**EG** p) верно для всех состояний, так как из каждого из них в конечном счете может быть достигнуто состояние s_0, s_1 или s_3, из которого начинается некоторое вычисление, вдоль которого p глобально верно.
- **A**[p **U** q] неверно в состоянии s_3, так как его единственное вычисление s_3^{ω} никогда не достигает состояния, в котором выполняется q. Для всех остальных состояний это, однако, верно, и вдобавок свойство p верно перед тем, как становится верным q.
- Наконец, формула **E**[p **U** ($\neg p \wedge$ **A**[$\neg p$ **U** q])] неверна в s_3, поскольку из s_3 не может быть достигнуто q-состояние. Для состояний s_0 и s_1 свойство верно, так как состояние s_2 может быть достигнуто из

этих состояний через p-путь, $\neg p$ верно в s_2, и все возможные стартующие в s_2 пути удовлетворяют формуле $\neg p$ **U** q, потому что s_2 – q-состояние. Например, в состоянии s_0 путь $(s_0 s_2 s_1)^\omega$ удовлетворяет свойству p **U** $(\neg p \wedge \mathbf{A}[\neg p \mathbf{U} q])$, так как $p \in Label(s_0)$, $p \notin Label(s_2)$ и $q \in Label(s_1)$. Для состояния s_2 свойство верно, поскольку p неверно в s_2, и для всех путей, начинающихся в s_2, ближайшее состояние – это q-состояние. Таким образом, свойство $\neg p \wedge \mathbf{A}(\neg p \mathbf{U} q)$ достигается через путь длины 0.

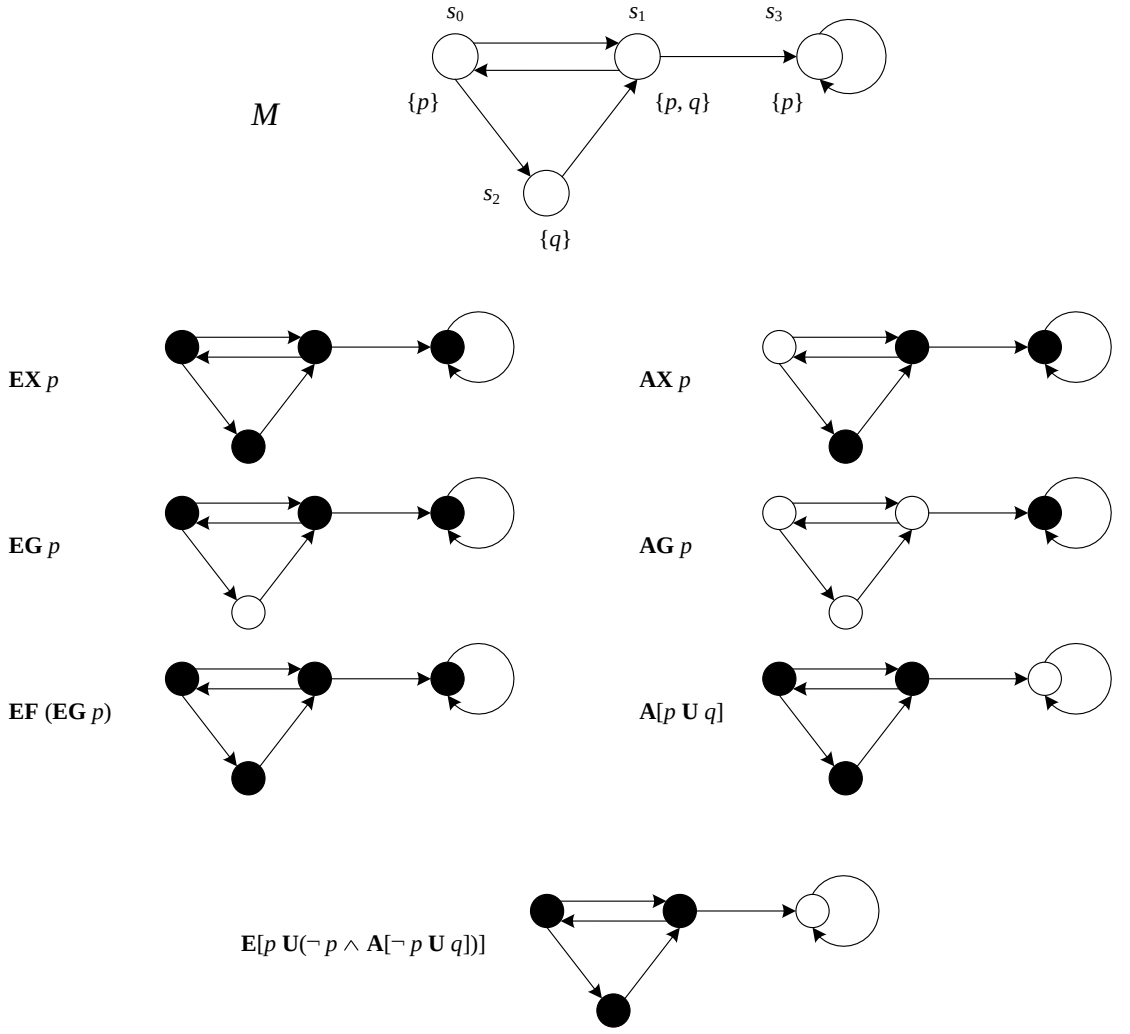


Рис. 2.18. Пример интерпретации нескольких CTL-формул на модели

2.3.3. Некоторые аксиомы CTL

В разделе о линейной темпоральной логике было показано, что аксиомы могут быть полезными для того, чтобы доказывать эквивалентность между формулами: часто вместо доказательства эквивалентности с использованием семантических определений достаточно применять аксиомы, которые определены в терминах

синтаксиса формул. Это облегчает доказательство эквивалентности формул. Важной аксиомой для проверки LTL-моделей является правило расширения оператора **U**:

$$\varphi \mathbf{U} \psi \equiv \psi \vee (\varphi \wedge \mathbf{X}[\varphi \mathbf{U} \psi]).$$

При подстановке этого правила в определения **F** и **G** получим:

$$\mathbf{G}\,\varphi \equiv \varphi \wedge \mathbf{XG}\,\varphi;$$

$$\mathbf{F}\,\varphi \equiv \varphi \vee \mathbf{XF}\,\varphi.$$

Для логики CTL существуют аналогичные аксиомы. Учитывая, что каждый линейный темпоральный оператор **U**, **F** или **G** должен быть предварен экзистенциальным или универсальным квантификатором, получаем аксиомы, перечисленные в табл. 2.4.

Таблица 2.4. Аксиомы расширения для CTL

$\mathbf{EG}\,\varphi$	$\equiv$	$\varphi \wedge \mathbf{EX\,EG}\,\varphi$
$\mathbf{AG}\,\varphi$	$\equiv$	$\varphi \wedge \mathbf{AX\,AG}\,\varphi$
$\mathbf{EF}\,\varphi$	$\equiv$	$\varphi \vee \mathbf{EX\,EF}\,\varphi$
$\mathbf{AF}\,\varphi$	$\equiv$	$\varphi \vee \mathbf{AX\,AF}\,\varphi$
$\mathbf{E}[\varphi\,\mathbf{U}\,\psi]$	$\equiv$	$\psi \vee (\varphi \wedge \mathbf{EX}(\mathbf{E}[\varphi\,\mathbf{U}\,\psi]))$
$\mathbf{A}[\varphi\,\mathbf{U}\,\psi]$	$\equiv$	$\psi \vee (\varphi \wedge \mathbf{AX}(\mathbf{A}[\varphi\,\mathbf{U}\,\psi]))$

У всех этих аксиом базовая идея состоит в выражении справедливости формулы предложением о текущем состоянии (где нет необходимости в использовании темпоральных операторов) и предложением о прямых потомках этого состояния (с использованием **EX** или **AX** в зависимости от того, с экзистенциальным или универсальным квантификатором рассматривается формула). Например, формула **EG** φ верна в состоянии *s*, если φ верна в *s* (предложение о текущем состоянии) и φ выполняется для всех состояний вдоль пути, начинающегося с прямого потомка *s* (предложение о потомках состояний). Первые четыре аксиомы могут быть выведены из последних двух.

Например, для **AF** φ получим:

AF φ

$\Leftrightarrow$ { по определению **AF** }

A[true **U** φ]

$\Leftrightarrow$ { аксиома для **A**[φ **U** ψ] }

$\varphi \vee (\text{true} \wedge \mathbf{AX}[\mathbf{A}(\text{true}\ \mathbf{U}\ \varphi)])$

$\Leftrightarrow$ { исчисление предикатов; определение **AF** }

$\varphi \vee \mathbf{AX}[\mathbf{AF}\ \varphi]$.

Используя этот результат, выводим правило расширения для **EG** φ:

EG φ

$\Leftrightarrow$ { по определению **EG** }

$\neg\mathbf{AF}\ \neg\varphi$

$\Leftrightarrow$ { результат предыдущего вывода }

$\neg(\neg\varphi \vee \mathbf{AX}[\mathbf{AF}\ \neg\varphi])$

$\Leftrightarrow$ { исчисление предикатов }

$\varphi \wedge \neg\mathbf{AX}[\mathbf{AF}\ \neg\varphi]$

$\Leftrightarrow$ { по определению **AX** }

$\varphi \wedge \mathbf{EX}(\neg[\mathbf{AF}\ \neg\varphi])$

$\Leftrightarrow$ { по определению **EG** }

$\varphi \wedge \mathbf{EX}[\mathbf{EG}\ \varphi]$.

Аналогичные преобразования могут быть проделаны для того, чтобы получить аксиомы для **EF** и **AG**. Как отмечено выше, элементарными аксиомами являются две последние аксиомы расширения. Они могут быть доказаны с использованием семантики CTL, и эти доказательства очень похожи на доказательство закона расширения для оператора **U** логики LTL, изложенное в соответствующем разделе.

2.3.4. Сравнение выразительной силы CTL, CTL* и LTL

Для того чтобы лучше представить взаимоотношения между LTL, CTL и CTL*, опишем синтаксис CTL* и выполним альтернативную характеризацию синтаксиса CTL и LTL в терминах CTL*. Сделаем это, явно различая *формулы состояний* – свойства, выполняющиеся в состояниях, и *формулы пути* – свойства, которые выполняются для путей.

Так как CTL и CTL* интерпретируются на ветвящихся моделях, тогда как LTL интерпретируется на последовательностях, сравнение между этими тремя логиками не прямолинейное. Сравнение упрощается, если немного поменять определение LTL, так что ее семантика будет определена в терминах ветвящейся модели. Несмотря на то, что существуют различные способы интерпретации LTL на ветвящихся

структурах (например, должно φ выполняться в формуле $\mathbf{X}\,\varphi$ для всех или для некоторого потомка текущего состояния?), наиболее простой, но, в то же время, общий подход состоит в том, чтобы считать, что LTL-формула φ выполняется на *всех* путях. Это приводит к следующему рассмотрению LTL в терминах CTL*.

Формулы состояний LTL определены как $\varphi ::= \mathbf{A}\,\beta$, где β – формула пути. Формулы пути определены согласно следующему правилу (здесь $p \in AP$):

$$\beta ::= p \mid \neg\beta \mid (\beta \vee \beta) \mid \mathbf{X}\,\beta \mid (\beta\ \mathbf{U}\ \beta).$$

В результате CTL, CTL* и LTL теперь интерпретируются в терминах одной модели, и эта общая семантическая база дает возможность для сравнения. Табл. 2.5 суммирует синтаксис трех рассмотренных типов логики.[4]

Таблица 2.5. Синтаксис LTL, CTL и CTL*

LTL	формулы состояния	$\varphi ::= \mathbf{A}\,\beta$
	формулы пути	$\beta ::= p \mid \neg\beta \mid (\beta \vee \beta) \mid \mathbf{X}\,\beta \mid (\beta\ \mathbf{U}\ \beta)$
CTL	формулы состояния	$\varphi ::= p \mid \neg\varphi \mid (\varphi \vee \varphi) \mid \mathbf{E}\,\beta$
	формулы пути	$\beta ::= \neg\beta \mid \mathbf{X}\,\varphi \mid (\varphi\ \mathbf{U}\ \varphi)$
CTL*	формулы состояния	$\varphi ::= p \mid \neg\varphi \mid (\varphi \vee \varphi) \mid \mathbf{E}\,\beta$
	формулы пути	$\beta ::= \varphi \mid \neg\beta \mid (\beta \vee \beta) \mid \mathbf{X}\,\beta \mid (\beta\ \mathbf{U}\ \beta)$

Для CTL* используются те же сокращения, что и для LTL, и еще одно дополнительное: $\mathbf{A}\,\beta = \neg\mathbf{E}\,\neg\beta$.

Интерпретация CTL* интуитивно ясна из интерпретации CTL. Определим формальную семантику CTL*. Пусть $p \in AP$ – атомарное предложение, $M = (S, R, Label)$ – *CTL-модель* (*модель Крипке*), $s \in S$ – состояние модели, $\sigma \in P_M(s)$ – путь, φ и ψ – формулы состояния, α и β – формулы пути. Введем два *отношения выполняемости*, справедливость которых будем обозначать так: $M, s \models_{State} \varphi$ и $M, \sigma \models_{Path} \alpha$.

Как и ранее, будем опускать модель M в случае, когда она подразумевается контекстом.

Отношение $\models_{State}$ задается следующим образом:

---

[4] Здесь выбран альтернативный путь для определения синтаксиса CTL, который упрощает сравнение. Следует иметь в виду, что $\neg\mathbf{E}\,\neg\beta$ равняется $\mathbf{A}\,\beta$ для формулы пути β.

$$s \models_{State} p \quad \Leftrightarrow p \in Label(s);$$
$$s \models_{State} \neg\varphi \quad \Leftrightarrow \neg(s \models_{State} \varphi);$$
$$s \models_{State} (\varphi \vee \psi) \quad \Leftrightarrow (s \models_{State} \varphi) \vee (s \models_{State} \psi);$$
$$s \models_{State} \mathbf{E}\,\beta \quad \Leftrightarrow \exists\sigma \in P_M(s): (\sigma \models_{Path} \beta).$$

Аналогично зададим отношение $\models_{Path}$:

$$\sigma \models_{Path} \varphi \quad \Leftrightarrow \sigma[0] \models_{State} \varphi;$$
$$\sigma \models_{Path} \neg\beta \quad \Leftrightarrow \neg(\sigma \models_{Path} \beta);$$
$$\sigma \models_{Path} (\alpha \vee \beta) \quad \Leftrightarrow (\sigma \models_{Path} \alpha) \vee (\sigma \models_{Path} \beta);$$
$$\sigma \models_{Path} \mathbf{X}\,\beta \quad \Leftrightarrow \sigma^1 \models_{Path} \beta;$$
$$\sigma \models_{Path} (\alpha\ \mathbf{U}\ \beta) \quad \Leftrightarrow \exists j \geq 0: \sigma^j \models_{Path} \beta \wedge (\forall 0 \leq k < j: \sigma^k \models_{Path} \alpha).$$

Здесь σ^1, σ^j и σ^k – соответствующие *суффиксы пути* σ.

Из рассмотрения синтаксиса следует, что CTL – это подмножество CTL*: все CTL-формулы принадлежат CTL*, но обратное не выполняется (синтаксически). Вначале проясним, как сравнивается выразительная сила двух темпоральных логик. Одна логика является более выразительной, чем другая, если она позволяет выражать термы, которые не могут быть выражены в другой. Этот синтаксический критерий применяется к CTL в сравнении с CTL*: первая синтаксически является подмножеством второй. Такой критерий в общем случае слишком прост. Более точно, он не исключает того факта, что для некоторой формулы φ в одной темпоральной логике L существует эквивалентная формула ψ (которая синтаксически отличается от φ) в темпоральной логике L'. Под эквивалентностью здесь подразумевается следующее.

Формулы φ и ψ называются *эквивалентными*, если и только если для всех моделей M и состояний s выполняется

$M, s \models \varphi$ тогда и только тогда, когда $M, s \models \psi$.

Теперь можно формально определить, что значит для двух темпоральных логик быть выразительно эквивалентными.

Темпоральные логики L и L' *выразительно эквивалентны*, если для каждой формулы $\varphi \in L$ существует эквивалентная ей формула $\psi \in L'$ (такая, что для всех моделей M и состояний s: $M, s \models \varphi \Leftrightarrow M, s \models \psi$) и для каждой формулы $\psi \in L'$ существует эквивалентная ей формула $\varphi \in L$ (такая, что для всех моделей M и состояний s: $M, s \models \varphi \Leftrightarrow M, s \models \psi$).

Если первое из этих условий верно, а второе нет, то логика L (строго) *менее выразительна*, чем L'.

Рис. 2.19 отражает взаимоотношения между тремя логиками, рассмотренными в данном разделе. Из этого рисунка следует, что CTL* более выразительна, чем обе логики LTL и CTL, тогда как LTL и CTL несравнимы. Для каждой из логик приведены примеры формул, выделяющих части областей, которым они принадлежат.

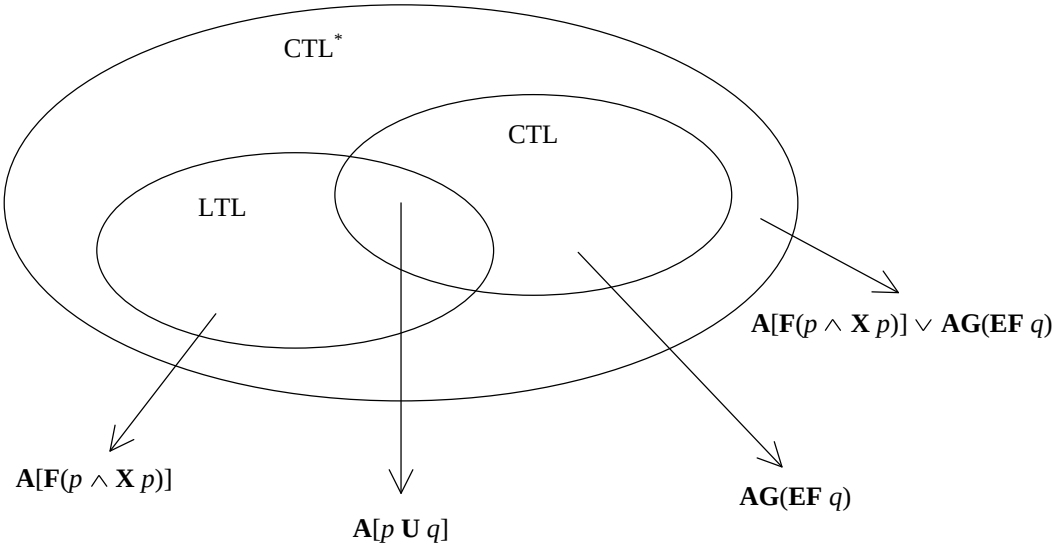


Рис. 2.19. Взаимоотношения между LTL, CTL и CTL*

Существует формула в LTL, но формула в CTL отсутствует. Например, **A**[**FG** p] и **A**[**F**($p \wedge$ **X** p)] – это LTL-формулы, для которых нет эквивалентных формул в CTL [4, 28]. Другой пример формулы из LTL, для которой нет эквивалентной формулы в CTL:

$$\mathbf{A}[\mathbf{GF}\ p \rightarrow \mathbf{F}\ q].$$

В соответствии с этой формулой, если p выполняется бесконечно часто, то q когда-нибудь станет верно. Это интересное свойство, которое появляется часто в доказательстве корректности систем. Например, типичное свойство коммуникационного протокола над ненадежным каналом связи формулируется следующим образом: «если сообщение посылается бесконечно часто, оно когда-нибудь достигнет получателя».

Докажем, что для LTL-формул **A**[**FG** a] и **A**[**F**($a \wedge$ **X** a)] не существует эквивалентных CTL-формул. Рассмотрим модели M_0, M_1, M_2, …, которые определяются индуктивно таким образом, как это показано на рис. 2.20.

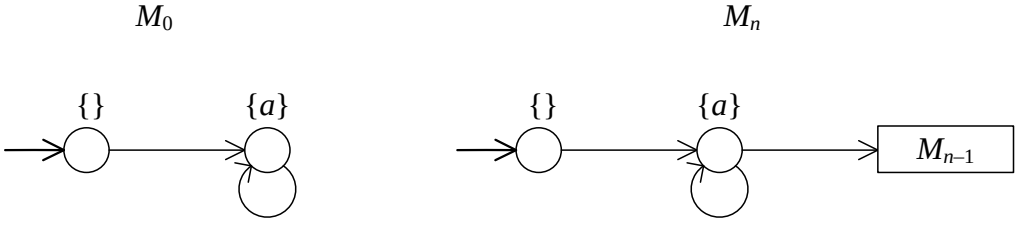


Рис. 2.20. Базовая модель M_0 и индуктивное построение модели M_n

На рис. 2.21 те же модели изображены другим способом.

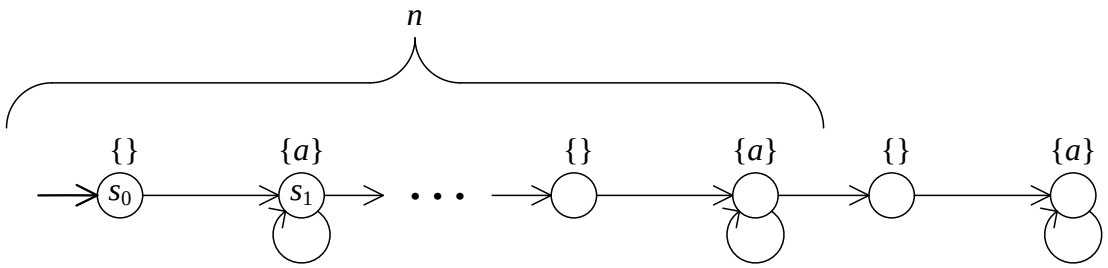


Рис. 2.21. Модели M_n для $n \geq 0$

Обозначим через M модель, изображенную на рис. 2.22.

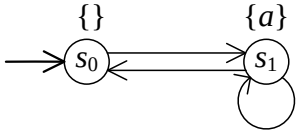


Рис. 2.22. Модель M

Через s_0 будем обозначать стартовые состояния моделей M и M_n, а через s_1 – соответственно их прямые потомки. Будем говорить, что формула φ не различает модели M' и M'', если

- условие M', $s_0 \models \varphi$ выполняется или не выполняется одновременно с условием M'', $s_0 \models \varphi$;
- условие M', $s_1 \models \varphi$ выполняется или не выполняется одновременно с условием M'', $s_1 \models \varphi$.

Из построения моделей M, M_0, M_1, … следует, что M_n, $s_0 \models \mathbf{A}[\mathbf{FG}\ a]$ и M_n, $s_0 \models \mathbf{A}[\mathbf{F}(a \wedge \mathbf{X}\ a)]$ для всех $n \geq 0$, в то время как M, $s_0 \not\models \mathbf{A}[\mathbf{FG}\ a]$ и M, $s_0 \not\models \mathbf{A}[\mathbf{F}(a \wedge \mathbf{X}\ a)]$. Это можно доказать следующим образом. Модель M содержит путь, которому соответствуют следующие пометки

$$\varnothing \to \{a\} \to \varnothing \to \{a\} \to \varnothing \to \ldots$$

и на котором не выполняются формулы $\mathbf{FG}\ a$ и $\mathbf{F}(a \wedge \mathbf{X}\ a)$. С другой стороны, так как в моделях M_n нет возможности бесконечно часто попадать в $\neg a$-состояние, каждый путь имеет следующий вид:

$$\varnothing \to \{a\}\ \ldots\ \varnothing \to (\{a\})^{\omega},$$

и на нем выполняются формулы $\mathbf{FG}\ a$ и $\mathbf{F}(a \wedge \mathbf{X}\ a)$. Таким образом, формулы $\mathbf{FG}\ a$ и $\mathbf{F}(a \wedge \mathbf{X}\ a)$ различают модели M и M_n.

Определим *темпоральную глубину* CTL-формулы φ (обозначается через $\|\varphi\|$) как максимальную вложенность темпоральных операторов в φ. Например, $\|a \wedge b\| = 0$ и $\|\mathbf{E}[(a \to b \wedge \neg c)\mathbf{U}((\mathbf{EX}\ a) \wedge \mathbf{E}[b\ \mathbf{U}\ c])]\| = 2$.

Используя индукцию по n, можно доказать, что модель M и модель M_n невозможно различить никакой CTL-формулой темпоральной глубины не больше n. Это означает, что $\forall \varphi \in \text{CTL}, \|\varphi\| \leq n$: $M_n, s_0 \models \varphi$ если и только если $M, s_0 \models \varphi$ (и аналогичное утверждение для s_1 вместо s_0). Это утверждение следует из трех фактов, проверяемых непосредственно:

1. Для любого $n \geq 0$ любая формула φ темпоральной глубины 0 не различает модели M и M_n.
2. Если ни формула φ, ни формула ψ не различают модели M и M_n, то такие формулы, как $\neg\varphi$ и $\varphi \wedge \psi$, тоже не различают эти модели.
3. Если ни формула φ, ни формула ψ не различают модели M, M_n и M_{n+1}, то такие формулы, как $\mathbf{EX}\,\varphi$, $\mathbf{AX}\,\varphi$, $\mathbf{E}[\varphi\ \mathbf{U}\ \psi]$ и $\mathbf{A}[\varphi\ \mathbf{U}\ \psi]$, не различают модели M и M_{n+1}.

Последний шаг доказательства состоит в следующем. Пусть существует CTL-формула φ, эквивалентная формуле $\mathbf{A}[\mathbf{FG}\,a]$ (или, соответственно, $\mathbf{A}[\mathbf{F}(a \wedge \mathbf{X}\,a)]$). С одной стороны, из того, что $M, s_0 \not\models \mathbf{A}[\mathbf{FG}\,a]$ и $M_n, s_0 \models \mathbf{A}[\mathbf{FG}\,a]$ (для всех n), следует, что $M, s_0 \not\models \varphi$ и $M_{\|\varphi\|}, s_0 \models \varphi$. С другой стороны, из того, что модели M и $M_{\|\varphi\|}$ невозможно различить CTL-формулой темпоральной глубины не более $\|\varphi\|$, следует, что φ имеет одно и то же истинностное значение на моделях M и $M_{\|\varphi\|}$. Это приводит к противоречию.

Существует формула в CTL, но формула в LTL отсутствует. Формула $\mathbf{AG\,EF}\,p$ – это CTL-формула, для которой нет эквивалентной формулы в LTL. Это свойство используется на практике, так как оно выражает тот факт, что можно добраться до состояния, в котором верно p, независимо от текущего состояния. Если p характеризует состояние, в котором исправляется определенная ошибка, то формула выражает, что всегда можно восстановиться после определенной ошибки. Доказательство того, что для $\mathbf{AG\,EF}\,p$ нет эквивалентной формулы в LTL, состоит в следующем.

Пусть φ – это LTL-формула, для которой $\mathbf{A}\,\varphi$ эквивалентно $\mathbf{AG\,EF}\,p$. Рассмотрим модель M на рис. 2.23 (а). Так как $M, s \models \mathbf{AG\,EF}\,p$, отсюда следует, что $M, s \models \mathbf{A}\,\varphi$. Пусть M' – подмодель M, показанная на рис. 2.23 (б). Пути, стартующие из s в M', входят в множество путей, начинающихся из s в M. Следовательно, $M', s \models \mathbf{A}\,\varphi$. Однако при этом свойство $M', s \models \mathbf{AG\,EF}\,p$ не выполняется, так как p никогда не становится верным вдоль единственного пути s^{ω}.

Аналогичный факт можно доказать для случая, когда LTL-формулы квантифицируются не по всем путям, а по некоторому пути. Пусть φ – это LTL-формула, для которой $\mathbf{E}\,\varphi$ эквивалентно $\mathbf{AG\,EF}\,p$.

Рассмотрим модель M. Из $M, s \models \mathbf{AG}\,\mathbf{EF}\,p$ следует $M, s \models \mathbf{E}\,\varphi$. Пусть M'' – надмодель M, показанная на рис. 2.23 (в). Пути, стартующие из s в M, входят в множество путей, начинающихся из s в M''. Следовательно, $M'', s \models \mathbf{E}\,\varphi$. Но при этом не выполняется свойство $M'', s \models \mathbf{AG}\,\mathbf{EF}\,p$, так как p никогда не становится верным вдоль единственного пути $(s'')^{\omega}$, стартующего в состоянии s'', достижимом из s.

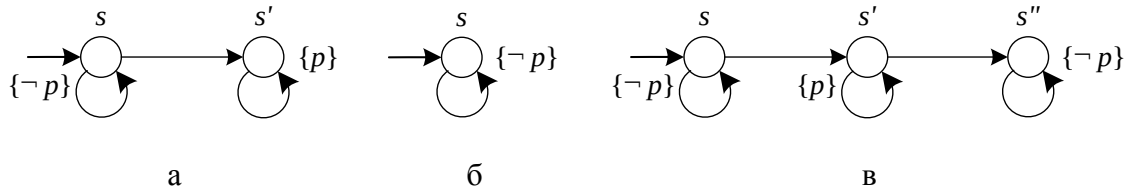


Рис. 2.23. Модель M, подмодель M' и надмодель M''

Взаимоотношения между CTL* и LTL. Между LTL и CTL* выполняется следующее взаимоотношение [26]: для произвольной CTL*-формулы φ эквивалентная LTL-формула (если только она существует) должна иметь форму $\mathbf{A}\,f(\varphi)$, где $f(\varphi)$ – это формула φ, в которой удалены все квантификаторы пути. Например, для $\varphi = \mathbf{EF}\,\mathbf{EG}\,p \rightarrow \mathbf{AF}\,q$ получаем $f(\varphi) = \mathbf{FG}\,p \rightarrow \mathbf{F}\,q$. Таким образом, φ – выделяющая формула для CTL*, если $\mathbf{A}\,f(\varphi)$ ей не эквивалентна. Если рассматривать такую интерпретацию LTL, в которой формулы квантифицируются не по всем путям, а по некоторому пути, то отсюда следует аналогичное утверждение: если для CTL*-формулы φ существует эквивалентная ей LTL-формула, то существует и эквивалентная LTL-формула вида $\mathbf{E}\,f(\varphi)$.

В качестве последнего примера различия в выразительности между LTL, CTL и CTL* рассмотрим LTL-формулу $\mathbf{GF}\,p$ – «бесконечно часто верно p». Нетрудно видеть, что предварение этой формулы экзистенциальным или универсальным квантификатором пути ведет к CTL*-формуле: $\mathbf{AG}\,\mathbf{F}\,p$ и $\mathbf{EG}\,\mathbf{F}\,p$ – это CTL*-формулы. Формула $\mathbf{AG}\,\mathbf{F}\,p$ эквивалентна формуле $\mathbf{AG}\,\mathbf{AF}\,p$ (для любой модели M справедливость этих формул равносильна), и поэтому для формулы $\mathbf{AG}\,\mathbf{F}\,p$ существует эквивалентная CTL-формула, так как формула $\mathbf{AG}\,\mathbf{AF}\,p$ – это CTL-формула. Для формулы $\mathbf{EG}\,\mathbf{F}\,p$, однако, не существует эквивалентной CTL-формулы.

2.3.5. Спецификация свойств в CTL

Для иллюстрации способа, с помощью которого в CTL записываются свойства, рассмотрим двухпроцессную программу взаимного исключения. Каждый процесс (P_1 или P_2) может быть в одном из

следующих трех состояний: критическая секция (C), секция запроса (T) и некритическая секция (N). Процесс начинается в некритической секции и указывает, что хочет войти в критическую секцию, путем входа в секцию запроса. Он остается в секции запроса до тех пор, пока не получит доступ к критической секции. Из критической секции процесс переходит в некритическую секцию. Состояние процесса P_i обозначается $P_i.s$ для $i = 1, 2$. Перечислим некоторые требуемые свойства и их формальную спецификацию в CTL.

1. «Оба процесса не могут находиться в критической секции одновременно»:
   $\mathbf{AG}[\neg(P_1.s = C \wedge P_2.s = C)]$.
2. «Процесс, который хочет попасть в критическую секцию, когда-нибудь сможет сделать это»:
   $\mathbf{AG}[P_1.s = T \rightarrow \mathbf{AF}(P_1.s = C)]$.
3. «Процессы должны строго чередоваться в получении доступа к критической секции»:
   $\mathbf{AG}[P_1.s = C \rightarrow \mathbf{A}(P_1.s = C\ \mathbf{U}\ (P_1.s \neq C \wedge \mathbf{A}(P_1.s \neq C\ \mathbf{U}\ P_2.s = C)))]$.

2.3.6. Условия справедливости в CTL

Для того чтобы оперировать *условиями справедливости* на моделях Крипке, задачу проверки CTL-моделей дополняют множеством ограничений справедливости $F = \{f_1, \ldots, f_k\}$. Ограничение справедливости – это пропозициональная формула, задающая некоторое подмножество состояний модели, либо *явно заданное* подмножество ее состояний. Например, для алгоритма взаимного исключения ограничение справедливости может быть таким: «процесс 1 не находится в своей критической секции». Смысл такого условия состоит в том, что в вычислении должно быть бесконечно много состояний, в которых процесс 1 не находится в критической секции. Основной принцип рассмотрения таких условий состоит в том, чтобы интерпретировать квантификаторы в CTL-формулах не на всех возможных путях, на которых они оперируют (этот вариант – вариант обычной семантики CTL – рассматривался в предыдущих разделах). Вместо этого следует рассматривать только *справедливые* вычисления – те, которые удовлетворяют указанным ограничениям $f_1, \ldots, f_k$.

Для заданного множества F будем говорить, что путь $\sigma = s_0 s_1 s_2 \ldots$ в модели M является F-справедливым, если для любого $f_i \in F$ в пути σ есть бесконечно много состояний, для которых выполняется f_i.

Если $lim(\sigma)$ обозначает множество состояний модели M, через которые путь σ проходит бесконечно часто, а f_i задаются подмножествами состояний модели M, то путь σ F-справедлив, если:

$lim(\sigma) \cap f_i \neq \varnothing$ для всех i.

Определим синтаксис Fair CTL-формул – формул с условиями справедливости:

$$\varphi ::= p \mid \neg\varphi \mid (\varphi \vee \varphi) \mid \mathbf{EX}\,\varphi \mid \mathbf{E}[\varphi\ \mathbf{U}\ \varphi] \mid \mathbf{A}[\varphi\ \mathbf{U}\ \varphi] \mid$$
$$\mid \mathbf{E}_F\,\mathbf{X}\,\varphi \mid \mathbf{E}_F\,[\varphi\ \mathbf{U}\ \varphi] \mid \mathbf{A}_F\,[\varphi\ \mathbf{U}\ \varphi].$$

Здесь F – некоторые множества ограничений справедливости. Справедливая семантика для CTL задается почти так же, как и классическая. Через $P_M^F(s)$ обозначим множество путей модели M, стартующих в состоянии s и являющихся F-справедливыми. При этом $P_M^F(s) \subseteq P_M(s)$. Интерпретация справедливых CTL-формул определяется в терминах *отношения выполняемости*, как и ранее: $M, s \models \varphi$ если и только если справедливая формула φ выполняется в состоянии s модели M:

$$s \models \mathbf{E}_F\,\mathbf{X}\,\varphi \quad \Leftrightarrow \exists\sigma \in P_M^F(s)\colon \sigma[1] \models \varphi;$$
$$s \models \mathbf{E}_F\,[\varphi\ \mathbf{U}\ \psi] \quad \Leftrightarrow \exists\sigma \in P_M^F(s)\colon (\exists j \geq 0\colon \sigma[j] \models \psi \wedge$$
$$\wedge\, (\forall 0 \leq k < j\colon \sigma[k] \models \varphi));$$
$$s \models \mathbf{A}_F\,[\varphi\ \mathbf{U}\ \psi] \quad \Leftrightarrow \forall\sigma \in P_M^F(s)\colon (\exists j \geq 0\colon \sigma[j] \models \psi \wedge$$
$$\wedge\, (\forall 0 \leq k < j\colon \sigma[k] \models \varphi)).$$

Выражения для пропозициональной логики и классической логики CTL идентичны семантике, введенной ранее. Для справедливых темпоральных операторов отличие состоит в том, что они квантифицируются по *справедливым*, а не по всем *путям*. Выразительная сила справедливой логики CTL строго больше, чем у классической CTL, и она содержится в CTL*. Справедливая логика CTL, как и классическая, несравнима с LTL.

Пример. Рассмотрим CTL-модель M на рис. 2.24. Пусть требуется проверить свойство $M, s_0 \models \mathbf{AG}[p \rightarrow \mathbf{AF}\ q]$. Оно неверно, так как существует путь $s_0 s_1 \left(s_2 s_4\right)^{\omega}$, никогда не попадающий в q-состояние. Причина, по которой этой свойство не выполняется, состоит в наличии недетерминированного выбора в состоянии s_2 между переходами в s_3 и в s_4. Поэтому если всегда игнорировать возможность перехода в s_3, то можно получить вычисление, для которого формула $\mathbf{G}[p \rightarrow \mathbf{AF}\ q]$ не выполняется:

$M, s_0 \not\models \mathbf{AG}[p \rightarrow \mathbf{AF}\ q]$.

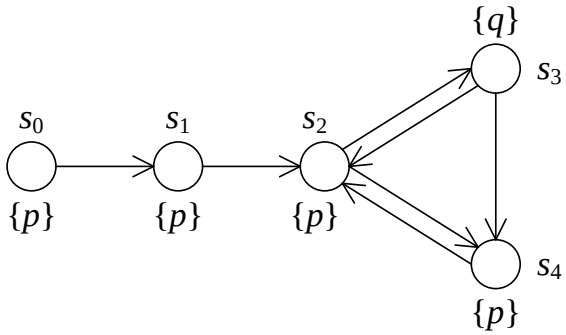


Рис. 2.24. Пример CTL-модели *M* для формул справедливости

Преобразуем исходную CTL-формулу в *справедливую* путем определения условий $F = \{f_1, f_2\}$, где $f_1 = \{s_3\}$, а $f_2 = \{s_4\}$. Проверим теперь справедливую формулу $\mathbf{AG}[p \to \mathbf{A}_F\,\mathbf{F}\,q]$ на модели. Более точно, рассмотрим условие $M, s_0 \models \mathbf{AG}[p \to \mathbf{A}_F\,\mathbf{F}\,q]$. Произвольный *F*-справедливый путь, стартующий в s_0, обязан проходить бесконечно часто через некоторое состояние из f_1 и некоторое состояние из f_2. Следовательно, состояния s_3 и s_4 должны достигаться бесконечно часто. Среди таких справедливых путей исключаются пути вида $s_0 s_1 \left(s_2 s_4\right)^{\omega}$, так как на таком пути никогда не встречается s_3. Отсюда следует, что формула $M, s_0 \models \mathbf{AG}[p \to \mathbf{A}_F\,\mathbf{F}\,q]$ выполняется.

2.3.7. Проверка моделей для CTL и CTL*

Известные эффективные алгоритмы верификации CTL базируются на использовании так называемой *теории неподвижной точки* совместно с существенным уменьшением размера модели. Такое уменьшение, как правило, происходит за счет применения технологии *упорядоченных двоичных разрешающих диаграмм*. С другой стороны, при использовании программирования с явным выделением состояний (автоматного программирования), существенное уменьшение модели обычно невозможно и нецелесообразно, так как рассматриваются только управляющие состояния, которых относительно немного, и при использовании эффективных алгоритмов выигрыш по времени обычно незаметен.

Под *локальной задачей верификации* обычно понимается вопрос: выяснить для данных *M*, *s* и φ, справедливо ли $M, s \models \varphi$.

При построении алгоритма его авторы Кларк и Эмерсон формулируют *глобальную задачу верификации*: для данных *M* и φ построить множество всех *s*, для которых $M, s \models \varphi$. Когда число состояний невелико, это множество можно строить в явном виде. Опишем идею их алгоритма на псевдокоде (листинг 2.4).

Листинг 2.4. Алгоритм «индукция по подформулам»

```
set<State> Sat(Formula φ)
{
   if (φ == true)          return S;
   if (φ == false)         return ∅;
   if (φ ∈ AP)             return {s | φ ∈ Label(s)};
   if (φ == ¬φ1)           return S \ Sat(φ1);
   if (φ == (φ1 ∨ φ2))     return Sat(φ1) ∪ Sat(φ2);
   if (φ == EX φ1)         return {s ∈ S | ∃(s,s') ∈ R:
                            s' ∈ Sat(φ1)};
   if (φ == E[φ1 U φ2])    return SatEU(φ1, φ2);
   if (φ == A[φ1 U φ2])    return SatAU(φ1, φ2);
   // Sat(φ) = {s | M,s ⊨ φ}
}
```

Как следует из рассмотрения текста программы, множество состояний, выполняющих формулу φ, строится индукцией по построению φ.

Для подформул вида $\mathbf{E}[\varphi_1\ \mathbf{U}\ \varphi_2]$ и $\mathbf{A}[\varphi_1\ \mathbf{U}\ \varphi_2]$ множество выполняющих состояний получается с использованием методов неподвижной точки. При этом модель Крипке кодируется в упорядоченную бинарную разрешающую диаграмму.

Упорядоченные *бинарные разрешающие диаграммы* (*OBDD – Ordered Binary Decision Diagram*) [29] – это мощная техника сжатия обрабатываемых данных, которая позволяет выполнять манипуляции над объектами прямо в сжатом виде, минуя распаковку, преобразование и обратную упаковку. Она состоит в том, что обрабатываемый объект неявно кодируется в непрерывный битовый поток из 2^n бит, который потом интерпретируется как булева функция n переменных. Именно для этой функции и строится корневой направленный ациклический граф, представляющий собой разрешающую диаграмму. Первоначально этот граф может представлять собой полное двоичное дерево высоты n, уровень i которого соответствует i-й переменной, а каждый путь от корня к листу в нем соответствует определенному двоичному набору и значению функции на этом наборе (*основное свойство*). Для многих битовых строк, используемых на практике, такое представление сильно избыточно, так как в полном дереве даже число вершин равно 2^n – числу, равному длине битового образа. Однако, преобразуя (редуцируя) этот граф и не утрачивая при этом его основного свойства, можно добиться очень эффективного представления (сжатых) данных. Над разрешающими диаграммами можно выполнять логические операции [30, 31]. Теоретические вопросы двоичных разрешающих диаграмм подробно изложены в монографии [32].

Таким образом, многие задачи могут решаться прямо на *ROBDD* (*Reduced OBDD*), без составления таблиц истинности и без представления графов в явном виде. В частности, к таким задачам относится задача проверки моделей [33, 34], для которой технология *ROBDD* успешно применяется, так как она позволяет работать с весьма большими пространствами состояний.

Возвращаясь к традиционным программам, отметим, что, несмотря на то, что данные исследования интересны и многообещающи, существует эффективный и твердо устоявшийся метод, основанный на другой парадигме, которая является концептуально более простой. Этот метод базируется на переформулировке синтаксиса языка CTL. Алгоритм, который будет описан далее, работает для всех Fair CTL-формул и дополнен таким образом, что позволяет строить подтверждающие сценарии для проверяемых формул.

В предыдущем разделе было дано определение синтаксиса языка Fair CTL и введены несколько дополнительных темпоральных операций в нем. Сейчас некоторые из этих операций преобразуем в базовые – выразим темпоральную часть синтаксиса языка Fair CTL через операции **EX**, **EU**, $\mathbf{E}_F\,\mathbf{G}$:

$$\varphi ::= p \mid \neg\varphi \mid (\varphi \vee \varphi) \mid \mathbf{EX}\,\varphi \mid \mathbf{E}[\varphi\ \mathbf{U}\ \varphi] \mid \mathbf{E}_F\,\mathbf{G}\,\varphi.$$

Здесь $p \in AP$ – атомарное предложение, а F – множество *ограничений справедливости*. Используются следующие сокращения:

- $\mathbf{AX}\,\varphi = \neg\mathbf{EX}\,\neg\varphi$;
- $\mathbf{A}_F\,\mathbf{X}\,\varphi = \neg\mathbf{E}_F\,\mathbf{X}\,\neg\varphi$;
- $\mathbf{E}_F\,\mathbf{X}\,\varphi = \mathbf{EX}(\varphi \wedge \mathbf{E}_F\,\mathbf{G}\ \text{true})$;
- $\mathbf{EG}\,\varphi = \mathbf{E}_\varnothing\mathbf{G}\,\varphi$;
- $\mathbf{A}[\varphi\ \mathbf{U}\ \psi] = \neg(\mathbf{E}[\neg\psi\ \mathbf{U}\ \neg(\varphi \vee \psi)] \vee \mathbf{EG}\,\neg\psi)$;
- $\mathbf{A}_F\,[\varphi\ \mathbf{U}\ \psi] = \neg(\mathbf{E}_F\,[\neg\psi\ \mathbf{U}\ \neg(\varphi \vee \psi)] \vee \mathbf{E}_F\,\mathbf{G}\,\neg\psi)$;
- $\mathbf{E}_F\,[\varphi\ \mathbf{U}\ \psi] = \mathbf{E}[\varphi\ \mathbf{U}\ (\psi \wedge \mathbf{E}_F\,\mathbf{G}\ \text{true})]$,

а также классические сокращения для **F** и **G**.

Алгоритм, который описывается далее, наследует идею Кларка и Эмерсона [35]: множество выполняющих состояний строится для каждой подформулы входной формулы (для каждого состояния создается список выполненных в нем подформул). Ввиду изменения набора базовых правил языка CTL, псевдокод, записанный в листинге 2.4, превращается в текст листинга 2.5.

Листинг 2.5. Индукция по построению формулы

```
set<State> Sat(Formula φ)
{
   if (φ == true)          return S;
   if (φ == false)         return ∅;
   if (φ ∈ AP)             return {s | φ ∈ Label(s)};
   if (φ == ¬φ1)           return S \ Sat(φ1);
   if (φ == (φ1 ∨ φ2))     return Sat(φ1) ∪ Sat(φ2);
   if (φ == EX φ1)         return {s ∈ S | ∃(s,s') ∈ R:
                            s' ∈ Sat(φ1)};
   if (φ == E[φ1 U φ2])    return SatEU(φ1, φ2);
   if (φ == EF G φ1)       return SatEG(φ1, F);
   // Sat(φ) = {s | M,s ⊨ φ}
}
```

Кратко опишем алгоритм. Будем считать для удобства, что для исходной модели Крипке построена симметричная ей модель – граф, в котором все переходы заменены на противоположные им.

1. Можно считать, что выполняющие множества для атомарных предложений содержатся в исходных данных (информация об этих множествах указана в самой модели Крипке).
2. Если известно выполняющее множество для формулы φ, то можно построить его и для формулы $\neg\varphi$.
3. Аналогично, если известны выполняющие множества для φ и ψ, то построим их и для $\varphi \vee \psi$.
4. Сделаем «один шаг назад» от выполняющего множества для φ – получим выполняющее множество для **EX** φ.
5. Для построения выполняющего множества формулы **E**[φ **U** ψ] вначале отмечаем в качестве выполняющих все вершины, в которых соблюдается формула ψ, а потом построим от них деревья «обратных путей» *вдоль тех вершин, в которых выполнена формула* φ. Для этого пригодится симметричный граф. Пример приведен на рис. 2.25.

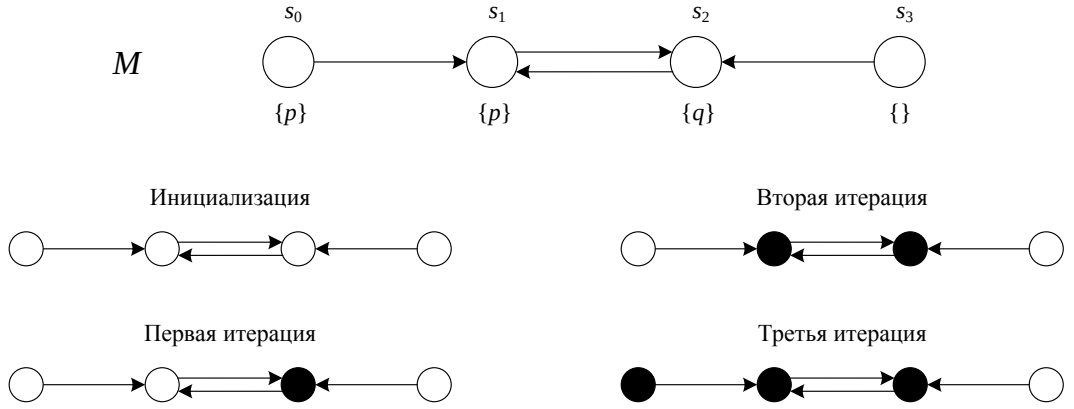


Рис. 2.25. Пример итеративного вычисления $Sat_{EU}(p, q)$

6. Осталось только научиться строить выполняющее множество для $\mathbf{E}_F\,\mathbf{G}\,\varphi$. Эта формула означает, что в модели существует *справедливый* путь, все состояния которого удовлетворяют формуле φ. Это построение выполняется следующим образом. Вначале исключаем из графа вершины, которые не выполняют формулу φ (так как в них формула $\mathbf{E}_F\,\mathbf{G}\,\varphi$ тем более не выполняется). В оставшемся графе на всех путях выполняется формула $\mathbf{G}\,\varphi$, и поэтому осталось только проверить, из каких состояний в нем существует хотя бы один бесконечный справедливый путь (или, что то же самое, построить выполняющее множество для формулы $\mathbf{E}_F\,\mathbf{G}$ true). Алгоритм, решающий эту задачу, основан на поиске компонент сильной связности модели Крипке и работает за билинейное время относительно размеров модели Крипке и множества ограничений справедливости F: $O(|F| \times (|S| + |R|))$. Если ограничений справедливости нет ($F = \varnothing$), то время работы линейно от размера модели – $O(|S| + |R|)$. Этот алгоритм обсуждается в разд. 2.4.

Трудоемкость итогового алгоритма составляет $O(|\varphi| \times (|S| + |R|))$ и растет билинейно относительно размеров формулы и модели.

Теперь осталось только дополнить этот алгоритм методами *предоставления подтверждений* истинности формул в моделях. Иными словами, требуется построить способ генерации сценариев. Итак, требуется *показать*, что в данном состоянии s модели M выполняется (или не выполняется) формула φ.

1. Для доказательства $\mathbf{EX}\,\varphi$ предъявим вершину в модели Крипке, в которую из вершины s существует переход и которая выполняет φ. Такая вершина обязательно существует, иначе на этапе верификации не обнаружилось бы, что формула $\mathbf{EX}\,\varphi$ верна.
2. Доказательство формул $\mathbf{E}[\varphi\,\mathbf{U}\,\psi]$ и $\mathbf{EG}\,\varphi$ выполняется рекуррентным способом с использованием предыдущего пункта. Выполним разложение этих формул согласно аксиомам расширения. Тогда для доказательства $\mathbf{E}[\varphi\,\mathbf{U}\,\psi]$ достаточно построить путь в графе, применяя шаг за шагом предыдущий пункт к рекуррентному разложению этой формулы до тех пор, пока не попадем в вершину, выполняющую ψ.
   Для доказательства $\mathbf{EG}\,\varphi$ сделаем то же самое, пока не попадем в вершину, в которой уже были. Путь в этом случае, начиная с некоторого состояния, становится периодическим (рис. 2.26).
3. Для доказательства $\mathbf{E}_F\,\mathbf{G}\,\varphi$ требуется предъявить путь, ведущий из текущей вершины в *справедливую* компоненту сильной связности (такую, которая содержит состояние, удовлетворяющее свойству f_i,

для всех $f_i \in F$), причем такой, что во всех его состояниях выполняется свойство φ. Его можно найти одним из алгоритмов, описанных в разд. 2.4. Путь при этом имеет такую же форму, как и в случае формулы **EG** φ.

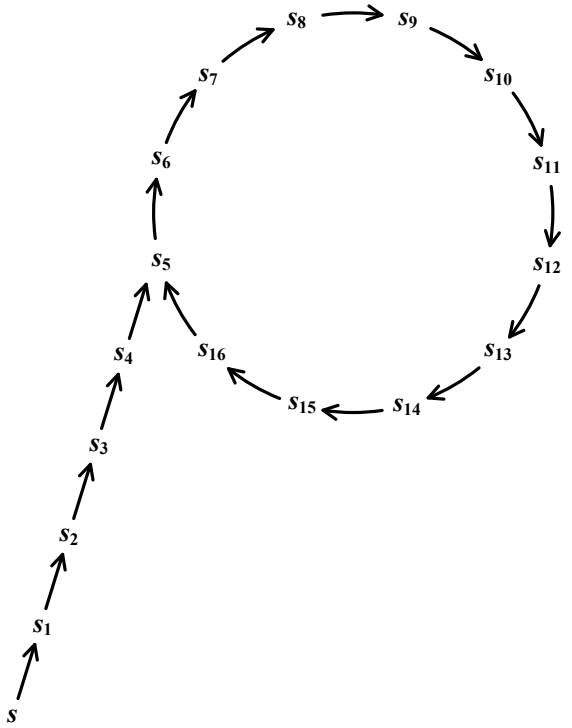


Рис. 2.26. Бесконечный «ρ-путь»

На этом изложение алгоритма завершено. Анализ построенного алгоритма формирования сценариев, а также семантики языка CTL позволяет сформулировать следующее утверждение:

- Если в модели Крипке существует бесконечный путь, выполняющий заданную CTL-формулу или являющийся контрпримером к ней, то существует и путь «в ρ-форме» (аналогично, выполняющий или опровергающий ее), представимый в виде объединения «предциклической» и «циклической» частей (рис. 2.26).

Доказательство этого факта является конструктивным и целиком опирается на применение описанного алгоритма. Достаточно только заметить, что алгоритм всегда завершается, выдавая сценарий, который необходимо обладает свойством периодичности.

Проверка моделей для логики CTL*

Задача проверки CTL*-формулы на модели Крипке имеет такую же асимптотическую сложность, как и аналогичная задача для логики LTL. Она выполняется поэтапно для подформул, имеющих структуру формулы LTL. Опишем эту конструкцию подробнее.

Пусть заданы модель Крипке M и CTL*-формула φ. Требуется определить множество состояний модели, для которых формула выполняется. Рассмотрим все подформулы формулы φ, имеющие вид $\mathbf{A}\beta$ или $\mathbf{E}\beta$, где β – LTL-формула, которая не содержит квантификаторов пути. Запустим алгоритм проверки LTL-формул для формул β или $\neg\beta$, соответственно, и получим в результате множество состояний модели, на которых выполняются рассматриваемые формула $\mathbf{A}\beta$ или $\mathbf{E}\beta$. Пометим все эти состояния в модели M данными подформулами. После этого будем интерпретировать эти подформулы как пропозициональные переменные. Можно даже заменить их новыми атомарными предложениями, поскольку теперь для каждого состояния модели известно, какие из новых атомарных предложений в нем соблюдаются. При этом в исходной формуле число квантификаторов пути уменьшается как минимум на единицу. Дальше этот процесс следует повторить с другими подформулами формулы φ, внешними по отношению к уже обработанным, и обработать их аналогичным образом до тех пор, пока, в конечном счете, не будет обработана исходная формула φ.

При этом для модели M будет получено множество состояний, в которых соблюдается формула φ.

Замечания о длине формулы

Вспомним, что проверка моделей для LTL выполняется за экспоненциальное время относительно размера формулы. Несмотря на то, что разница во временной сложности по отношению к длине формулы выглядит серьезной, здесь следует сделать несколько замечаний. Формулы в LTL никогда не бывают длиннее, и, как правило, бывают короче, чем эквивалентные формулы в CTL. Это следует из того факта, что для формул, которые могут быть переведены из CTL в LTL, эквивалентная LTL-формула получается путем удаления всех квантификаторов пути, и поэтому она обычно получается короче. Более того, для каждой модели M существует LTL-формула φ такая, что любая CTL-формула, эквивалентная $\mathbf{E}\varphi$ (или $\mathbf{A}\varphi$) – если такая формула существует в CTL – имеет более чем полиномиальную длину по отношению к длине LTL-формулы (либо ее нельзя построить за полиномиальное время при условии, что $P \neq NP$). Это наглядно иллюстрируется адаптированным примером [36], приводимым ниже.

Для требования, которое может быть специфицировано и в CTL, и в LTL, кратчайшая возможная формулировка в LTL никогда не бывает длиннее CTL-формулы, и, более того, может быть экспоненциально

короче. Таким образом, достоинство проверки моделей CTL, состоящее в том, что она линейна по отношению к длине формулы, уменьшается (или даже полностью устраняется) тем фактом, что заданное свойство требует (гораздо) более длинной формулировки в CTL по сравнению с LTL.

Пример. Рассмотрим задачу поиска гамильтонова пути в произвольном связном графе, используя термины LTL и CTL. Пусть задан граф $G = (V, E)$, где V – множество вершин, а $E \subseteq V \times V$ – множество ребер. Пусть $V = \{v_1, \ldots, v_n\}$. Гамильтонов путь – это путь в графе, который проходит через каждую вершину ровно один раз (эта задача аналогична задаче коммивояжера, в которой стоимость прохода по каждому ребру одинакова для всех ребер).

Вначале опишем задачу гамильтонова пути в LTL. Используем метод, который состоит в рассмотрении графа G в виде модели Крипке, которая строится из G следующим образом.

Пометим каждую вершину v_i в V уникальным атомарным предложением p_i, то есть $Label(v_i) = \{p_i\}$. Кроме того, введем новую (терминальную) вершину w такую, что $w \notin V$ с $Label(w) = \{q\}$, где q – атомарное предложение, отличное от всех p_i. Сделаем вершину w прямым потомком каждого узла v_i (добавим ребра (v_i, w) в граф), а также самой себя – (ребро (w, w) тоже добавим в граф). Рис. 2.27 отображает эту конструкцию для связного графа с 4 вершинами.

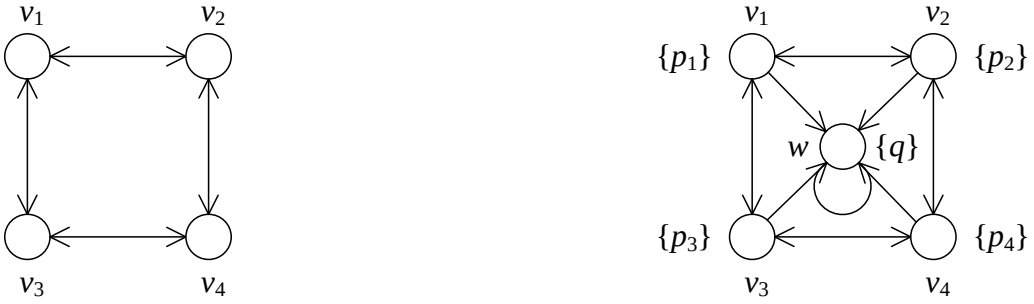


Рис. 2.27. Кодирование задачи гамильтонова пути в модели

Для данной структуры свойство существования гамильтонова пути в графе может быть сформулировано в LTL следующим образом[5]:

$$\mathbf{E}[(\forall i\colon \mathbf{F}\, p_i) \wedge \mathbf{X}^n\, q],$$

[5] Строго говоря, это не является правильно построенной LTL-формулой, так как квантификатор пути **E** не является частью LTL. Однако, для $\mathbf{E}\,\varphi$, где φ не содержит никаких квантификаторов пути (как в этом примере), можно взять эквивалентное свойство $\neg\mathbf{A}\,\neg\varphi$, которое может быть проверено путем проверки свойства $\mathbf{A}\,\neg\varphi$, записанного правильно построенной LTL-формулой.

где $\mathbf{X}^1 q = \mathbf{X} q$ и $\mathbf{X}^{n+1} q = \mathbf{X}(\mathbf{X}^n q)$. Эта формула верна в каждом состоянии, из которого начинается путь, который удовлетворяет каждому атомарному предложению один раз. Это соответствует однократному посещению каждого состояния v_i. Желаемый бесконечный допустимый путь должен иметь суффикс вида w^ω, так как в противном случае он посетит одно или более состояний больше одного раза. Отметим, что длина вышеописанной формулы линейна по отношению к числу вершин в графе.

Формулировка задачи гамильтонова пути в CTL существует и может быть получена следующим образом. Начнем с построения CTL-формулы $g(p_1, \ldots, p_n)$, которая выполняется, если и только если существует путь из текущего состояния, проходящий через состояния, для которых $p_1 \ldots p_n$ становятся верными в естественном порядке:

$$g(p_1, \ldots, p_n) = p_1 \wedge \mathbf{EX}(p_2 \wedge \mathbf{EX}(\ldots \wedge \mathbf{EX}\, p_n)\ldots).$$

Ввиду ветвящейся интерпретации CTL формулировка свойства гамильтоновости графа в CTL требует явного перечисления всех возможных гамильтоновых путей. Пусть P является множеством перестановок множества $\{1, \ldots, n\}$. Тогда получим:

$$\exists \theta \in P\colon g(p_{\theta(1)}, \ldots, p_{\theta(n)}).$$

После явного перечисления всех возможных перестановок получим формулу экспоненциальной длины по отношению к числу вершин в графе. Если бы существовала эквивалентная CTL-формула полиномиальной длины и ее можно было бы найти полиномиальным алгоритмом, то *NP*-полная задача гамильтонова пути была бы разрешима за полиномиальное время ввиду полиномиальности алгоритма проверки CTL-моделей, а это возможно лишь в случае $P = NP$. Это означает, что за полиномиальное время нельзя построить CTL-формулу, описывающую свойство гамильтоновости графа (при условии $P \neq NP$).

2.4. Поиск справедливых путей

В данном разделе рассматривается задача поиска справедливых путей в графе переходов общего вида. Граф переходов может представлять собой подграф *модели Крипке*, размеченной *LTL-таблицы* или *автомата Бюхи*. Можно сказать, что граф имеет структуру, идентичную одной из перечисленных, с одним отличием: *отношение переходов в нем не обязано быть тотальным* – могут существовать состояния без потомков.

Пусть дано состояние s в графе и множество условий справедливости F. Требуется определить, существует ли в графе хотя бы один бесконечный путь, стартующий в состоянии s и удовлетворяющий всем условиям справедливости (это значит, что все условия соблюдаются на этом пути бесконечно часто).

Данная задача имеет эквивалентную формулировку: определить, выполняется ли в заданном состоянии s графа Fair CTL-формула $\mathbf{E}_F\,\mathbf{G}$ true. Эта задача является принципиальной для проверки моделей, как для справедливых CTL-формул, так и для LTL-формул.

В случае $F = \varnothing$ это задача поиска произвольного бесконечного пути, а в случае $F \neq \varnothing$ любой справедливый путь бесконечен, и это задача поиска справедливого пути.

Иногда задачу ставят в широком смысле: требуется определить множество всех состояний графа, из которых стартует бесконечный справедливый путь.

Для узкой постановки задачи больше подходит алгоритм *двойного обхода в глубину*, а для широкой – алгоритм, основанный на *поиске всех компонент сильной связности графа*.

2.4.1. Двойной обход в глубину

Один из популярных методов проверки спецификаций, заданных в виде формул темпоральной логики LTL, – это применение алгоритма *двойного обхода в глубину* [17]. Данный алгоритм предназначен для поиска допускающих путей в *автомате Бюхи* и используется во многих верификаторах, в том числе в верификаторах *SPIN* и *Bogor*.

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

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

Формально алгоритм проверки на пустоту языка автомата Бюхи с помощью двойного DFS описывается псевдокодом из листинга 2.6 [17]. В алгоритме используется команда `terminate` для прекращения выполнения всей программы и возврата значения.

Листинг 2.6. Алгоритм двойного обхода в глубину

```
bool emptiness()
{
   для всех q0 ∈ Q0
      dfs1(q0);
   terminate false;
}

void dfs1(q)
{
   пометить q флагом flag1;
   для всех последователей q′ вершины q
      if (q′ не помечена флагом flag1)
         dfs1(q′);
   if (accept(q))
      dfs2(q);
}

void dfs2(q)
{
   пометить q флагом флагом flag2;
   для всех последователей q′ вершины q
   {
     if (q′ помечена флагом flag1)
         terminate true;
     else if (q′ не помечена флагом flag2)
         dfs2(q′);
   }
}
```

Алгоритм возвращает true, если был найден допускающий путь, и false – в противном случае. Если алгоритм вернул значение true, то можно восстановить допускающий путь: в стеке первого DFS хранится путь из начального состояния в некоторое допускающее состояние q_1. Этот путь будет искомым префиксом α. При этом в стеке второго DFS хранится путь из состояния q_1 в некоторое состояние q_2, содержащееся в стеке первого. Тогда, достроив этот путь состояниями, находящимися в стеке первого DFS выше состояния q_2, получим цикл $q_1 \to q_2 \to q_1$, проходящий через допускающее состояние q_1 – это искомый суффикс β. Таким образом, будет получен допускающий путь $\alpha\beta^{\omega}$.

Доказательство корректности алгоритма

Докажем корректность приведенного алгоритма [17]. Требуется доказать, что алгоритм двойного поиска в глубину находит

последовательность, допускаемую автоматом Бюхи, тогда и только тогда, когда она существует.

Если алгоритм вернул значение true, то в стеке первого DFS находится корректный путь в автомате Бюхи из начального состояния s в допускающее состояние q_2. В стеке второго DFS находится путь из состояния q_2 в состояние q_1, которое принадлежит пути из s в q_2. Другими словами, состояние q_2 достижимо из состояния q_1. Тогда искомый контрпример, допускаемый автоматом Бюхи, представляет собой бесконечный путь вида $\alpha\beta^{\omega}$, где α – это последовательность $s \to q_1$, а β – это последовательность $q_1 \to q_2 \to q_1$. Это изображено на рис. 2.28.

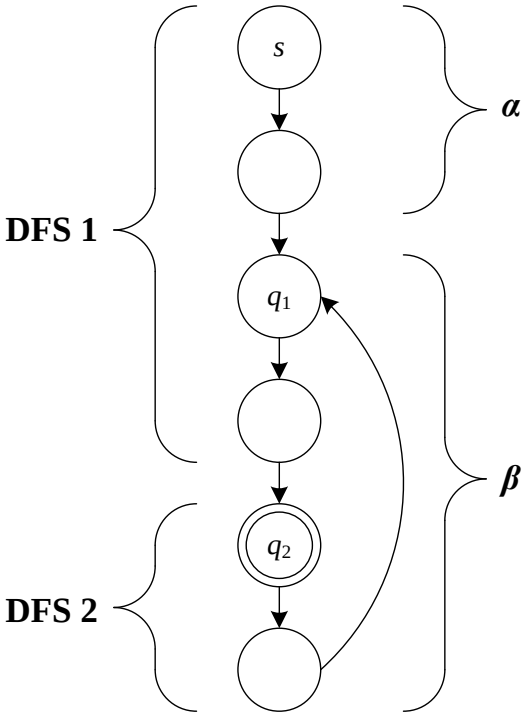


Рис. 2.28. Структура контрпримера, полученного алгоритмом двойного обхода в глубину

Докажем теперь обратное: если алгоритм вернул значение false, то допускающего пути действительно не существует. Предположим, что это не так. Пусть существует цикл, содержащий допускающие состояния и достижимый из начального состояния, а q – допускающее состояние этого цикла. Поскольку первый DFS обходит все состояния автомата, то существует шаг, на котором он откатывается из q. В этот момент запускается второй обход в глубину. Поскольку q лежит в цикле, существует путь или несколько путей из q в одно из состояний, находящихся в стеке первого DFS. Как минимум, существует путь в это само q. Тогда возможны два варианта:

1. Существует путь из q в некоторое состояние, содержащееся в стеке первого DFS, все состояния которого еще не были помечены

вторым DFS. Тогда по алгоритму второй DFS найдет этот путь, и, следовательно, найдет контрпример. Это противоречит сделанному предположению о том, что алгоритм не может найти контрпример.

2. На любом пути из q в состояния, содержащиеся в стеке первого DFS, лежит состояние r, помеченное ранее вторым DFS. Покажем, что такая ситуация не может возникнуть.

Пусть это не так. Тогда существуют допускающие состояния, которые входят в циклы, но второй DFS, будучи запущенным из них, никогда не сможет найти цикл. Пусть среди таких состояний второй DFS был первым запущен для состояния q. Далее, пусть r – это первое состояние, посещенное вторым DFS, среди тех, которые:

- входят в цикл, проходящий через q;
- были помечены ранее вторым DFS.

Наконец, пусть q' – допускающее состояние, из которого был запущен второй обход в глубину, который пометил состояние r. Состояния q, r и q' схематически изображены на рис. 2.29.

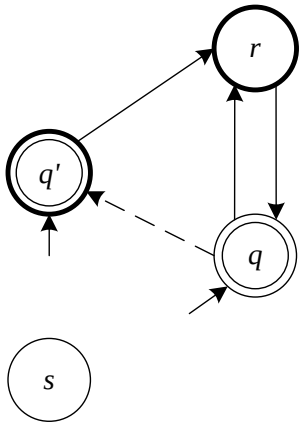


Рис. 2.29. Состояния q, q' и r. Допускающие состояния изображены двойными кругами. Состояния, помеченные вторым DFS – жирной окружностью

На данном этапе доказательства возможны два варианта.

1. Состояние q' достижимо из состояния q. Тогда существует цикл $q' \to r \to q \to q'$. Если бы этот цикл был обнаружен ранее при запуске второго DFS из q', то алгоритм был бы уже остановлен. Следовательно, он не был обнаружен. Однако это противоречит предположению о том, что состояние q – первое из состояний, из которых второму DFS не удается найти существующий цикл.
2. Состояние q' недостижимо из состояния q. Состояние q' не лежит в цикле, иначе будет получено то же противоречие, что и в пункте 1. Сделаем важное замечание: если некая вершина не входит в цикл, то откат из нее происходит лишь тогда, когда все вершины,

достижимые из нее, были обойдены и откаты из них были произведены. Как видно на рис. 2.29, q достижимо из q'. Используя это утверждение, получим, что первый DFS должен был сделать откат из q' лишь после того, как он сделал откат из q. Однако в таком случае, по описанию алгоритма двойного обхода в глубину, и второй DFS должен был быть запущен из q раньше, чем из q', а это противоречит предположению о порядке запусков второго обхода в глубину.

Таким образом, оказывается, что невозможна ситуация, в которой существующий цикл с допускающими состояниями не был обнаружен алгоритмом двойного обхода в глубину.

2.4.2. Поиск сильно связных компонент

Рассматривается задача поиска справедливых путей в широком смысле. Дан размеченный граф G и множество ограничений справедливости F. Требуется определить множество всех состояний графа, из которых стартует бесконечный справедливый путь.

Это построение выполняется в три этапа:

1. Находим у графа компоненты сильной связности. Учитываем при этом, что если компонента состоит лишь из одной вершины без петли, то она не считается компонентой сильной связности (исключением является случай, когда на данной вершине есть петля) – отношение сильной связности будем считать в общем случае иррефлексивным.
2. Те из компонент сильной связности, которые для каждого ограничения справедливости содержат состояние, в котором это ограничение выполняется, назовем *справедливыми компонентами*.
3. Строим в графе деревья обратных путей, начиная от всех вершин *справедливых компонент*, попутно маркируя все посещенные вершины как выполняющие искомое свойство (сами справедливые компоненты маркируем с самого начала).

Для поиска сильно связных компонент применяем алгоритм Тарьяна [37, 38]. Главная идея алгоритма состоит в запуске обхода в глубину на рассматриваемом графе. Для этого будем поддерживать стек поиска `S`. Сильно связные компоненты формируют непрерывные участки этого стека. Вершину сильно связной компоненты, расположенную наиболее глубоко в стеке, будем называть *корнем* этой компоненты.

Основная задача заключается в определении того, когда вершина графа является корнем своей компоненты сильной связности. Для ее

решения каждую вершину v снабдим переменной-индексом v.index, которая нумерует вершины последовательно в порядке их просмотра поиском в глубину.

Кроме того, сопоставим этой вершине значение v.lowest, которое на каждом рекурсивном вызове удовлетворяет условию, инвариантному на протяжении всего алгоритма:

$$v.lowest = \mathop{\textbf{Min}}_{\substack{\text{по всем } v',\\ \text{в которые был}\\ \text{найден путь из } v}} v'.index$$

Отсюда следует, что вершина v является корнем сильно связной компоненты графа тогда и только тогда, когда после ее обработки выполняется равенство v.lowest = v.index. Алгоритм работает таким образом, что всегда, когда требуется значение v.lowest, оно уже вычислено.

После обработки всех потомков вершины v в конце стека S находятся все вершины, принадлежащие ее сильно связной компоненте, в порядке их обхода поиском в глубину, начиная с самой вершины v. Поэтому следующим шагом является удаление всех этих вершин, включая v, из стека с выводом этой сильно связной компоненты.

Алгоритм представлен в листинге 2.7.

Листинг 2.7. Алгоритм Тарьяна поиска сильно связных компонент графа

```
На входе граф G = (V, E)

void computeSCC()
{
  // Счетчик номера вершины в DFS
  index = 0;
  // Пустой стек вершин
  S = ∅;

  foreach (v ∈ V) do
    // Запустим DFS на всех еще не обработанных вершинах
    if (v.index не определено)
      visit(v);
}

void visit(v)
{
  // Установим номер для v
  v.index = index;
  v.lowest = index;
  index = index + 1;
  S.push(v);
```

```
    // Рассмотрим потомков v
    foreach ((v, v') ∈ E) do
      // Если вершина v' не посещалась
      if (v'.index не определено)
      {
        visit(v');
        v.lowest = min(v.lowest, v'.lowest);
      }
      else   // Добавлять v' в дерево не следует
        v.lowest = min(v.lowest, v'.index);

    // Если v - корень сильно связной компоненты
    if (v.lowest == v.index)
    {
      print "Сильно связная компонента:";
      repeat
      {
        v' = S.pop;
        print v';
      }
      until (v' == v);
    }
  }
```

Алгоритм работает за линейное время от размера графа. Отсюда следует, что задача определения множества состояний модели, из которых существуют бесконечные справедливые пути, решается описанным методом за время $O(|F| \times |M|)$, билинейное относительно размера модели M и размера множества ограничений справедливости. Если ограничений справедливости нет, то алгоритм работает за линейное время от размера модели.

2.5. Проверка моделей для темпоральной логики реального времени

Системы, критичные ко времени

Такие темпоральные логики, как LTL и CTL, предназначены для спецификации свойств, которые фокусируются на временном порядке событий. Этот временной порядок является качественным понятием, так как время не рассматривается в количественном отношении. Например, пусть предложение p соответствует тому, что произошло событие A, а предложение q соответствует тому, что произошло событие B. Тогда LTL-формула $\mathbf{G}[p \rightarrow \mathbf{F}\, q]$ соответствует тому, что за событием A всегда рано или поздно будет следовать событие B, но она ничего не говорит о том, насколько длинный будет период между наступлением событий A и B. Это отсутствующее количественное

представление о времени существенно для спецификации *систем, критичных ко времени*. В соответствии с общепринятым определением, *системы, критичные ко времени* – это такие системы, в которых корректность зависит не только от логического результата вычисления, но также и от времени, за которое результаты получены. Системы, критичные ко времени, должны удовлетворять не только требованиям функциональной корректности, но и требованиям своевременности. Такие системы обычно характеризуются количественными временными свойствами, относящимися к наступлению событий.

Пример системы, критичной ко времени, – это железнодорожный переезд: если обнаружено приближение поезда, переезд должен быть закрыт в течение определенного времени с целью блокировки автомобильного и пешеходного трафика перед тем, как поезд достигнет переезда. Коммуникационные протоколы – другой пример системы, критичной ко времени: после передачи данных повторная передача должна быть запущена, если подтверждение не получено в течение определенного времени. Третий пример системы, критичной ко времени – это аппарат лучевой терапии, благодаря которому пациенты получают высокую дозу радиации в течение ограниченного периода времени. Даже небольшое превышение этого периода опасно и может привести к смерти пациента.

Количественная временная информация существенна не только для систем, критичных ко времени, но также является необходимой составной частью в задачах анализа и спецификации аспектов производительности систем. Типичное предложение, относящееся к производительности, такое, как «выполнение задания происходит в течение 30 секунд», может быть сформулировано, только если имеется возможность *измерять* прошедшее время.

Количественное задание времени в темпоральной логике

В этом разделе рассматривается расширение ветвящейся темпоральной логики количественными представлениями о времени. Выше было показано, что линейная и ветвящаяся темпоральная логика интерпретируются в терминах моделей, где главную роль играет понятие состояния. Наиболее важное изменение при переходе к количественным временным характеристикам является возможность измерять время. Можно спросить, каким образом понятие времени включается в модель, базирующуюся на состояниях, и каковы взаимоотношения между состояниями и временем. При включении временных характеристик в модель должны быть рассмотрены несколько аспектов:

- какова нотация временных ссылок (абсолютная или относительная)?
- какова семантика временной области (дискретная или непрерывная)?

Выбор временной области

Одним из наиболее важных аспектов является выбор семантики временной области. Так как в природе время имеет непрерывную структуру, наиболее естественным выбором является непрерывная временная область, такая, как вещественные числа. Например, для синхронных систем, в которых компоненты выполняются в пошаговом режиме, предпочтительна дискретная временная область. Известным примером дискретной темпоральной логики является RTTL (*Real-Time Temporal Logic*). Для синхронных систем альтернативно можно рассматривать использование обычной темпоральной логики, в которой для отражения дискретных временных характеристик используется оператор следования. Кратко опишем, как это работает.

Базовая идея состоит в том, что вычисление может быть представлено как последовательность состояний, в которой каждый переход из одного состояния в следующее рассматривается как единственный такт некоторых вычислительных часов. При этом $\mathbf{X}\,\varphi$ верно, если через один такт часов выполняется φ. Пусть $\mathbf{X}^k$ – это серия из k последовательных операторов следования, определенная по правилам $\mathbf{X}^0\,\varphi = \varphi$ и $\mathbf{X}^{k+1}\,\varphi = \mathbf{X}^k\,(\mathbf{X}\,\varphi)$ для $k \geq 0$. Тогда, например, требование того, чтобы максимальная задержка между событием A и событием B была меньше 32 временных единиц, может быть специфицировано как $\mathbf{G}[p \rightarrow \mathbf{X}^{<32}\,q]$, где $\mathbf{X}^{<k}\,\varphi$ – это сокращение для конечной дизъюнкции $\mathbf{X}^0\,\varphi \vee \ldots \vee \mathbf{X}^{k-1}\,\varphi$, а p и q означают наступление событий A и B, соответственно. При этом отметим, что введение $\mathbf{X}^k$-оператора делает задачу выполнимости для временной линейной темпоральной логики *EXPSPACE*-трудной. Если не ограничиваться синхронными системами, то следует рассмотреть вещественнозначную временную область. Это позволяет более аккуратно специфицировать несинхронные системы.

Проверка моделей для временной логики CTL

Введение времени в темпоральную логику должно выполняться так, чтобы сохранить разрешимость задачи проверки моделей. Будем концентрироваться на темпоральной логике Timed CTL (более коротко, TCTL) [39], для которой задача проверки моделей разрешима. Это *ветвящаяся темпоральная логика реального времени*, которая поддерживает относительные временные ссылки и

интерпретируется в терминах непрерывной временной области ($\mathbb{R}^+$ – неотрицательные вещественные числа).

Базовая идея Timed CTL состоит в использовании временных ограничений в качестве параметров обычных темпоральных операторов CTL. В Timed CTL можно, например, специфицировать предложение «максимальная задержка между событием *A* и событием *B* меньше 32 временных единиц» формулой $\mathbf{AG}[p \rightarrow \mathbf{AF}_{<32}\, q]$, где, как и ранее, предложение p (q) соответствует наступлению события *A* (*B*). Логика Timed CTL – это веществennозначное расширение CTL, для которого существуют алгоритмы проверки моделей и некоторые инструментальные средства.

Основная сложность проверки моделей, в которых временная область непрерывна, состоит в том, что проверяемая модель имеет *бесконечно много* состояний – для каждого значения времени система может быть в определенном состоянии, а множество этих значений бесконечно. Кажется, что это неразрешимая задача. Фактически, главный вопрос в проверке моделей для темпоральной логики реального времени состоит в том, *как справляться с бесконечным множеством состояний*. Так как решения данной ключевой проблемы достаточно похожи у линейной и ветвящейся темпоральных логик, рассмотрим один случай – Timed CTL.

Основная идея выполнения проверки моделей для непрерывной временной области состоит в том, чтобы реализовать дискретизацию этой области по запросу – в зависимости от свойства, которое должно быть доказано и модели системы.

В качестве спецификационного формализма для систем реального времени вводится понятие *временного автомата*, расширения конечного автомата, в котором для измерения времени используются часы. Временные автоматы использовались для спецификации различных типов систем, критичных ко времени и безопасности, в том числе для коммуникационных протоколов. Для временных автоматов разработано несколько инструментов проверки моделей. Основной задачей данного раздела является разработка алгоритма проверки моделей, который определяет истинность TCTL-формулы по отношению к временному автомату. Теория и примеры временных автоматов описаны в соответствии с курсом лекций [1].

2.5.1. Временные автоматы

Временные автоматы используются для моделирования конечных систем реального времени. Временной автомат – это фактически

конечный автомат, снабженный конечным множеством вещественнозначных *часовых переменных* или, более коротко, часов. Часы используются для измерения временных характеристик.

Часы – это переменная, принимающая значения в $\mathbb{R}^+$.

В данном разделе в качестве часов используются буквы *x*, *y* и *z*. Состояние временного автомата состоит из текущей *позиции* автомата и текущих значений всех часовых переменных[6]. Часы могут быть инициализированы (в нуль), когда система делает переход. После инициализации они начинают скрыто увеличивать свое значение. Все часы идут с одинаковой скоростью. Значение часов, таким образом, обозначает время, прошедшее с момента их инициализации. Условия на значения часов используются в качестве активирующих условий переходов: переход активен и может быть пройден, только если часовое ограничение выполнено, в противном случае переход блокирован. Для ограничения времени, которое может быть проведено в позиции, используются инварианты на часах. Активирующие условия и инварианты являются ограничениями для часов.

Для множества C часов (где $x, y \in C$) множество *часовых ограничений* над C обозначается $\Psi(C)$ и определяется следующим образом:

$$\alpha ::= x \sim c \mid x - y \sim c \mid \neg\alpha \mid (\alpha \wedge \alpha).$$

Здесь $c \in \mathbb{N}$ и $\sim \in \{<, \leq\}$.

В данном разделе используются такие сокращения, как $x \geq c$ для $\neg(x < c)$ и $x = c$ для $x \leq c \wedge x \geq c$ и т. д. Допускается также сложение констант в часовых ограничениях, например, $x \leq c + d$ для $d \in \mathbb{N}$. Сложение же часовых переменных, как в ограничении $x + y < 3$, сделало бы проверку моделей неразрешимой. Также если бы c могло быть действительным числом, то проверка моделей для временной темпоральной логики, которая интерпретируется, как в текущем случае, на плотной временной области, стала бы неразрешимой. Следовательно, требуется, чтобы c было натуральным числом. Разрешимость не затрагивается, если ограничение смягчить так, что c могло бы быть рациональным числом. В этом случае рациональные числа в каждой формуле могут быть преобразованы в натуральные при подходящем масштабировании (это будет показано ниже).

Временной автомат – это набор (L, l_0, E, *Label*, C, *clocks*, *guard*, *inv*), где

---

[6] Отметим сознательно введенное различие между *позицией* и *состоянием*.

- L – непустое конечное множество *позиций* с начальной позицией $l_0 \in L$;
- E – множество ребер, каждое из которых (будем далее обозначать его e) снабжено пометками *from*(e) и *into*(e), указывающими, соответственно, из какого состояния исходит данное ребро и в какое состояние оно входит (таким образом, одну и ту же пару состояний может соединять сколько угодно ребер);
- *Label*: $L \to 2^{AP}$ – функция, которая сопоставляет каждой позиции $l \in L$ множество *Label*(l) атомарных предложений;
- C – конечное множество часов;
- *clocks*: $E \to 2^C$ – функция, которая сопоставляет каждому ребру $e \in E$ множество часов *clocks*(e);
- *guard*: $E \to \Psi(C)$ – функция, которая помечает каждое ребро $e \in E$ часовым ограничением *guard*(e) над C;
- *inv*: $L \to \Psi(C)$ – функция, которая сопоставляет каждой позиции *инвариант*.

Функция *Label* играет ту же роль, что и в моделях для CTL и LTL, и сопоставляет позиции множество атомарных предложений, которые верны в этой позиции. Как будет показано ниже, эта функция важна только для определения выполняемости атомарных предложений в семантике TCTL. Для ребра e множество *clocks*(e) обозначает множество часов, которые должны быть сброшены при проходе e. Функция *guard*(e) – активирующее условие, которое определяет, когда e может быть пройдено. Для позиции l условие *inv*(l) ограничивает время, которое может быть проведено в этой позиции.

При отображении временных автоматов будем придерживаться следующих соглашений. Вершины обозначают позиции, а ребра обозначаются стрелками. Инварианты указаны рядом с позициями, если только они не равны true (это означает, что имеется условие на задержку). Стрелки снабжены пометками, которые состоят из (необязательного) часового ограничения и (необязательного) множества часов, которые должны быть сброшены. Пометки разделены прямой горизонтальной линией. Если часовое ограничение равно true и если нет часов, которые должны быть сброшены, то пометка на стрелке опускается. Далее часто будем опускать маркирующее отношение *Label*, так как оно играет роль только при обсуждении алгоритма проверки моделей и не важно для остальных вводимых концепций.

Пример. Рис. 2.30 (а) отображает простой временной автомат с одними часами x и одной позицией l с петлевым переходом. Петлевой

переход может быть пройден, если часы x имеют значение как минимум 2. При прохождении этого перехода часы x обнуляются. Первоначально часы x имеют значение 0. На рис. 2.30 (б) приведен пример запуска этого временного автомата, в котором отображены значения часов x. В каждый момент, когда часы сбрасываются в 0, автомат переходит из позиции l в нее же. Ввиду инварианта true в позиции l, время может идти неограниченно долго в процессе пребывания в l.

Небольшое изменение временного автомата на рис. 2.30 (а) путем введения инварианта $x \leq 3$ в позиции l приводит к тому, что часы x больше не могут идти вперед без ограничений. Только при $x \geq 2$ (активирующее ограничение) и $x \leq 3$ (инвариант) исходящее ребро может и должно быть пройдено. Это отображено на рис. 2.30 (в) и (г).

Заметим, что другой эффект наблюдается при усилении активирующего ограничения на рис. 2.30 (а) до условия $2 \leq x \leq 3$, если сохранять инвариант true в позиции l. В этом случае исходящее ребро может быть пройдено только когда $2 \leq x \leq 3$ (как в предыдущем сценарии), но оно не обязано быть пройдено – оно может быть просто проигнорировано, если разрешить ожидание без переходов, находясь в позиции l. Это проиллюстрировано на рис. 2.30 (д) и (е).

Важно то, что различные часы могут стартовать в различные моменты времени и, следовательно, нет никакой нижней границы на разницу их значений. Это невозможно, например, в модели дискретного времени, когда разница между двумя параллельными часами является кратной одной единице времени. Наличие многих часов позволяет моделировать многие параллельные задержки. Это иллюстрируется следующим примером.

Пример. Рис. 2.31 (а) отображает временной автомат с двумя часами x и y. Первоначально и те и другие часы начинают отсчет со значения 0 до тех пор, пока не пройдут две временных единицы. Начиная с этого момента, оба ребра активны и могут быть недетерминированно пройдены. В результате либо часы x, либо часы y обнуляются, в то время как оставшиеся часы продолжают идти. При этом разница между значениями часов x и y может быть произвольной. Пример эволюции этого временного автомата изображен на рис. 2.31 (б).

Пример. Рис. 2.32 показывает временной автомат с двумя позициями, называемыми *off* и *on*, и двумя часами x и y. Часы инициализируются в 0 при вхождении в стартовую позицию *off*.

Временной автомат на рис. 2.32 моделирует переключатель. Переключатель может быть включен в любой момент времени, происходящий как минимум через две временных единицы начиная с

момента последнего переключения, даже если он все еще включен. Он выключается автоматически ровно через 9 временных единиц после последнего момента, когда он был переключен из позиции *off* в *on*. Часы x используются для отслеживания задержки с последнего момента времени, когда он был переключен. Ребра, отмеченные часовым ограничением $x \geq 2$, моделируют переходы по включению. Часы y используются для того, чтобы отслеживать задержку с последнего момента, когда автомат был переключен из позиции *off* в *on*, и контролируют выключение.

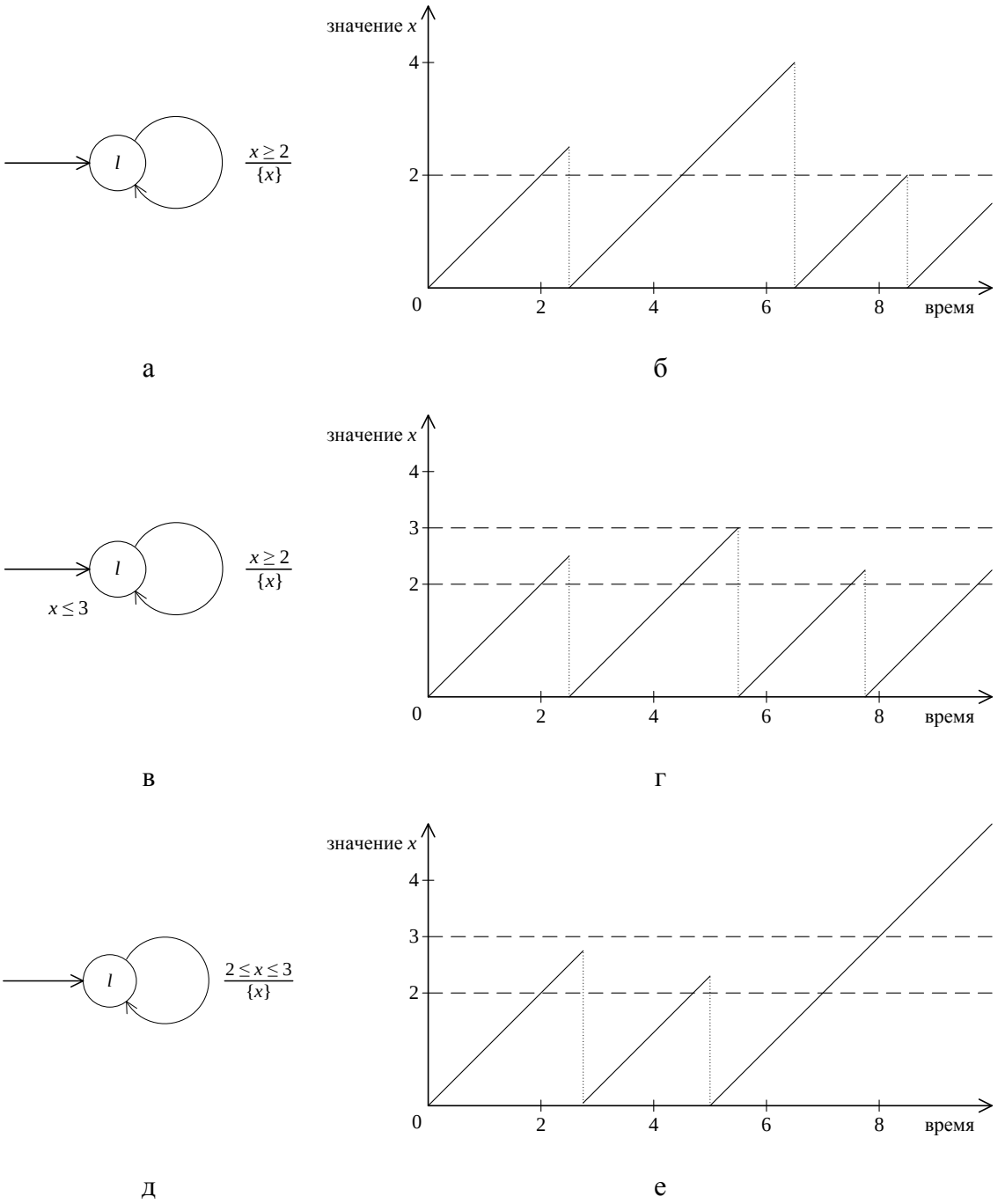
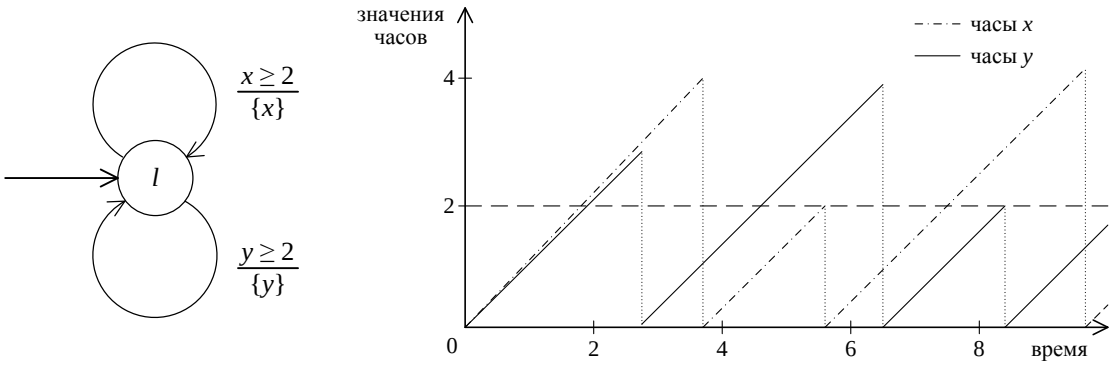


Рис. 2.30. Временной автомат с одними часами и пример его эволюции



а б

Рис. 2.31. Временной автомат с двумя часами и пример его эволюции

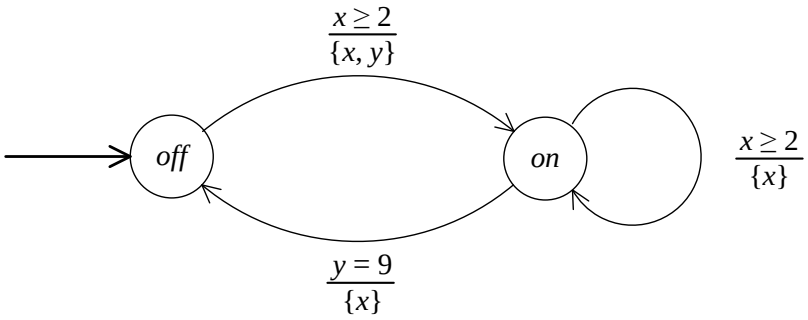


Рис. 2.32. Пример временного автомата: переключатель

В приведенных примерах было показано, как часы эволюционируют. Значения часов формально определяются *часовыми оценками*.

Часовая оценка v для множества часов C – это функция $v: C \to \mathbb{R}^+$, которая присваивает каждым часам $x \in C$ их текущее значение $v(x)$.

Пусть $V(C)$ обозначает множество всех часовых оценок над C. *Состоянием* временного автомата A называется пара (l, v), где l – позиция A, а v – оценка над множеством часов C автомата A.

Пример. Рассмотрим временной автомат на рис. 2.32. Некоторые состояния этого автомата – это пары (off, v) с $v(x) = v(y) = 0$, (off, v') с $v'(x) = 4$ и $v'(y) = 13$, и (on, v'') с $v''(x) = 4$ и $v''(y) = 3$. Отметим, что последнее состояние недостижимо.

Пусть v – часовая оценка над множеством часов C. Часовая оценка $v + t$ означает, что все часы увеличены на t по отношению к часовой оценке v. Она определяется как $(v + t)(x) = v(x) + t$ для всех часов $x \in C$. Часовая оценка $[x]v$, означающая оценку v со сброшенными часами x, определяется следующим образом:

$$([x]v)(y) = \begin{cases} v(y), & \text{если } y \neq x \\ 0, & \text{если } y = x. \end{cases}$$

Для вложенных вхождений операции сбрасывания используются стандартные сокращения. Например, $[x]([y]v)$ записывается просто как $[x, y]v$.

Пример. Рассмотрим часовые оценки v и v' из предыдущего примера. Оценка $v + 9$ определена как $(v + 9)(x) = (v + 9)(y) = 9$. В часовой оценке $[x](v + 9)$ часы x имеют значение 0, а часы y – 9. При этом часовая оценка v' равняется $[x](v + 9) + 4$.

Теперь можно формально определить, что означает утверждение, что часовое ограничение верно или нет. Это может быть сделано тем же путем, что и характеризация семантики темпоральной логики – определением *отношения выполняемости*. В этом случае отношение выполняемости $\models$ – это отношение между часовыми оценками (над множеством часов C) и часовыми ограничениями (над C).

Для $x, y \in C$; $v \in V(C)$; $\alpha, \beta \in \Psi(C)$ имеем:

$$v \models x \sim c \quad \Leftrightarrow v(x) \sim c;$$
$$v \models x - y \sim c \quad \Leftrightarrow v(x) - v(y) \sim c;$$
$$v \models \neg\alpha \quad \Leftrightarrow v \not\models \alpha;$$
$$v \models (\alpha \wedge \beta) \quad \Leftrightarrow (v \models \alpha) \wedge (v \models \beta).$$

Для отрицания и конъюнкции правила идентичны правилам пропозициональной логики. Для проверки, выполняется ли $x \sim c$ в v, требуется проверить, верно ли $v(x) \sim c$. Аналогично трактуются интервалы между часами.

Пример. Рассмотрим часовые оценки v, $v + 9$ и $[x](v + 9)$ из предыдущего примера и предположим, что требуется проверить справедливость формул $\alpha := x \leq 5$ и $\beta := (x - y = 0)$. Ввиду $v(x) = v(y) = 0$ имеем $v \models \alpha$ и $v \models \beta$. Далее, $v + 9 \not\models \alpha$, так как $(v + 9)(x) = 9 > 5$, и $v + 9 \models \beta$, так как $(v + 9)(x) = (v + 9)(y) = 9$. Так же можно проверить, что $[x](v + 9) \models \alpha$ и $[x](v + 9) \not\models \beta$.

2.5.2. Семантика временных автоматов

Интерпретация временных автоматов определяется в терминах бесконечной системы переходов $(S, \longrightarrow)$, где S – это множество состояний (пар, состоящих из позиции и часовой оценки), а $\longrightarrow$ – отношение переходов, которое определяет, как переходить из одного

состояния в другое. Существует два возможных пути, по которым временной автомат может действовать: переход по ребру в автомате или ожидание при нахождении в одном состоянии. Оба пути представляются одним отношением переходов $\longrightarrow$.

Система переходов, ассоциированная с автоматом A и называемая $M(A)$, определяется как $(S, s_0, \longrightarrow)$, где:

- $S = \{(l, v) \in L \times V(C) \mid v \models inv(l)\}$;
- $s_0 = (l_0, v_0)$, где $v_0(x) = 0$ для всех $x \in C$ (s_0 называется *начальным состоянием временного автомата*);
- отношение переходов $\longrightarrow \subseteq S \times (\mathbb{R}^+ \cup \{\circ\}) \times S$ определяется по правилам:

  1) $(l, v) \xrightarrow{\circ} (l', [clocks(e)]v)$, если следующие условия выполняются:
     (a) $e = (l, l') \in E$,
     (b) $v \models guard(e)$ и
     (c) $([clocks(e)]v) \models inv(l')$;

  2) $(l, v) \xrightarrow{d} (l, v + d)$ для положительного вещественного d, если следующее условие выполняется:
     $\forall d' \leq d: v + d' \models inv(l)$.

Множество состояний – это множество пар (l, v), где l – позиция A, а v – часовая оценка над C (множеством часов A), таких, что v не нарушает инвариант l. Отметим, что это множество может включать недостижимые состояния (строгое определение достижимости приведено ниже). Для перехода, который соответствует (a) прохождению ребра e во временном автомате, v должно (b) удовлетворять часовому ограничению e (в противном случае ребро заблокировано) и (c) новая часовая оценка, полученная путем сброса всех часов, сопоставленных ребру e в v, удовлетворяет инварианту целевой позиции l' (в противном случае в l' находиться запрещено). Ожидание в позиции (второе правило) в течение ненулевого промежутка времени разрешено, если инвариант сохраняется с течением времени. Отметим, что недостаточно потребовать только $v + d \models inv(l)$, так как это может нарушить инвариант для некоторого $d' < d$. Например, для $inv(l) = (x \leq 2) \vee (x > 4)$ и состояния (l, v) с $v(x) = 1.5$ не должно быть разрешено ожидание в течение 3 временных единиц: несмотря на то, что результирующая оценка $v + 3 \models inv(l)$, некоторые промежуточные оценки (например, $v + 2$) нарушают часовое ограничение.

Состояние s временного автомата A называется *достижимым*, если $s_0 \xrightarrow{*} s$, где s_0 начальное состояние автомата A, а отношение $\xrightarrow{*}$ – это рефлексивное и транзитивное замыкание отношения $\longrightarrow$.

Путем называется бесконечная последовательность $s_0\, a_0\, s_1\, a_1\, s_2\, a_2 \ldots$ состояний, чередующихся с маркерами переходов, таких, что $s_i \xrightarrow{a_i} s_{i+1}$ для всех $i \geq 0$.

Отметим, что маркеры могут быть либо $\circ$ – в случае перехода по ребру во временном автомате, либо (положительными) вещественными числами – в случае перехода по задержке. Множество путей, начинающихся в заданном состоянии, определяется, как и прежде.

Позиция пути – это пара (i, d) такая, что d равно 0, если $a_i = \circ$ – в случае перехода по ребру, и не превосходит a_i – в противном случае. Множество позиций пути вида (i, d) характеризует множество состояний, входящих в путь σ, который возникает при прохождении от состояния s_i к потомку s_{i+1}. Состояние s_i будем представлять парой (l_i, v_i). Пусть $Pos(\sigma)$ обозначает множество позиций пути σ. Для удобства пусть $\sigma(i, d)$ определяет состояние $(l_i, v_i + d)$. Линейный порядок на позициях пути определяется следующим образом:

$$(i, d) \ll (j, d'), \text{ если и только если } i < j \vee (i = j \wedge d < d').$$

Следовательно, позиция пути (i, d) предшествует позиции пути (j, d'), если l_i посещается перед l_j в σ или если эти позиции пути указывают на одну позицию в автомате и d меньше d'.

С целью «измерения» времени, которое проходит в течение пути, введем следующую величину.

Для пути $\sigma = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots$ и натурального i время, прошедшее с s_0 по s_i, обозначается $\Delta(\sigma, i)$ и определяется как

$$\Delta(\sigma,\ 0) = 0$$
$$\Delta(\sigma,\ i+1) = \Delta(\sigma,\ i) + \begin{cases} 0 & \text{если } a_i = \bigcirc; \\ a_i & \text{если } a_i \in \mathbb{R}^+. \end{cases}$$

Путь σ называется *расходящимся*, если $\lim_{i\to\infty} \Delta(\sigma, i) = \infty$. Множество расходящихся путей, начинающихся в состоянии s, обозначается $P_M^{\infty}(s)$. Примером *сходящегося* пути является путь, который посещает бесконечное число состояний в ограниченный промежуток времени. Например, путь

$$s_0 \xrightarrow{2^{-1}} s_1 \xrightarrow{2^{-2}} s_2 \xrightarrow{2^{-3}} s_3 \ldots$$

является сходящимся, так как бесконечное число состояний посещается в течение ограниченного интервала $[\frac{1}{2}, 1]$. Так как такие пути не являются реалистичными, обычно рассматривают *расходящиеся автоматы*.

Временной автомат называется *расходящимся*, если из каждого его *достижимого* состояния стартует некоторый расходящийся путь.

Путь $\sigma = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots$ называется *зеноновым*[7], если он является сходящимся и содержит бесконечно много переходов по ребру.

Временной автомат называется *незеноновым*, если любой путь, стартующий из любого *достижимого* состояния в этом автомате, не является зеноновым.

Можно дать эквивалентное определение: временной автомат называется *незеноновым*, если любой путь, стартующий из *начального состояния* этого автомата, не является зеноновым.

Алгоритмическая проверка условия незеноновости временного автомата достаточно трудна. Вместо этого обычно рассматривают достаточные условия, которые легко проверить путем статического анализа временного автомата. Следующее достаточное условие основывается на интуиции, что временной автомат является незеноновым, если на любом цикле, составленном из позиций этого автомата, время продвигается как минимум на единицу. Это позволяет сформулировать следующую теорему [4].

Теорема. Пусть A – временной автомат с множеством часов C такой, что для любого простого цикла в A

$$l_0 \xrightarrow{e_1} l_1 \xrightarrow{e_2} \ldots \xrightarrow{e_n} l_n, \text{ где } l_0 = l_n,$$

существуют часы $x \in C$ и индексы i, j от 1 до n такие, что:

- $x \in clocks(e_i)$ и
- для всех часовых оценок $\eta \in V(C)$ таких, что $\eta \models guard(e_j)$ и $[clocks(e_j)]\eta \models inv(l_j)$, выполняется $\eta \models x \geq 1$.

Тогда автомат A незенонов.

Доказательство. Пусть A – временной автомат над C, удовлетворяющий указанным требованиям, и π – путь, содержащий бесконечно много переходов по ребру. Так как автомат A содержит

---

[7] Название дано в честь *апории Зенона* «Ахиллес и черепаха», рассматривающей ситуацию, в которой описывается бесконечное число действий, выполненных за конечное время.

лишь конечное число позиций, через какую-то из этих позиций путь π пройдет бесконечное число раз. Обозначим эту позицию через l. Следовательно, начиная с некоторого момента, путь π разбивается на циклы, которые соединяются друг с другом в позиции l. Эти циклы, в свою очередь, разбиваются на простые циклы (циклы, не содержащие двух одинаковых вершин). При этом простые циклы не обязаны идти строго друг за другом; они, например, могут разбивать друг друга на части. Так или иначе, число простых циклов в автомате A конечно, и поэтому какой-то из них (обозначим его $l_0 \to l_1 \to \ldots \to l_n = l_0$) обязательно присутствует в пути π бесконечное число раз. Для этого простого цикла, согласно условию теоремы, существуют индексы i, j и часы x, удовлетворяющие двум описанным ограничениям. Предположим, что $i = n$ (это не уменьшает общности, так как позиции в цикле могут быть соответствующим образом перенумерованы ввиду того, что цикл встречается в пути π бесконечно часто). Рассмотрим фрагмент пути в системе переходов автомата A, который стартует и завершается в позиции l_0:

$$(l_0, \eta_0) \ldots (l_{j-1}, \eta_{j-1}) \xrightarrow{\circ} (l_j, \eta_j) \ldots (l_n, \eta_n).$$

Из условия следует, что на переходе $l_{n-1} \to l_n = l_0$ обнуляются часы x и что переход $l_{j-1} \to l_j$ возможен, только если $\eta_{j-1}(x) \geq 1$ (в противном случае при $\eta_{j-1}(x) < 1$ нарушается либо активирующее условие, либо инвариант позиции l_j). Отсюда следует, что проход цикла $l_0 \to \ldots \to l_n = l_0$ занимает как минимум одну временную единицу. Таким образом, путь π является расходящимся, и автомат A незенонов.

Пример. Вновь рассмотрим переключатель из предыдущего примера (рис. 2.32). Приведем пример префикса пути в переключателе:

$$\sigma = (off, v_0) \xrightarrow{3} (off, v_1) \xrightarrow{\circ} (on, v_2) \xrightarrow{4} (on, v_3) \xrightarrow{\circ}$$

$$(on, v_4) \xrightarrow{1} (on, v_5) \xrightarrow{2} (on, v_6) \xrightarrow{2} (on, v_7) \xrightarrow{\circ} (off, v_8) \ldots$$

с $v_0(x) = v_0(y) = 0$, $v_1 = v_0 + 3$, $v_2 = [x, y]v_1$, $v_3 = v_2 + 4$, $v_4 = [x]v_3$, $v_5 = v_4 + 1$, $v_6 = v_5 + 2$, $v_7 = v_6 + 2$ и $v_8 = [x]v_7$. Эти часовые оценки объединяются в табл. 2.6.

Таблица 2.6. Часовые оценки

	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
x	0	3	0	4	0	1	3	5	0
y	0	3	0	4	4	5	7	9	9

Переход $(off, v_1) \xrightarrow{\circ} (on, v_2)$, например, возможен, поскольку (a) существует ребро e из off в on, (b) $v_1 \models x \geq 2$, так как $v_1(x) = 3$,

и (c) $v_2 \models inv(on)$. Типичными позициями пути σ являются пары (0, 3), (1, 0), (2, 4), (3, 0), (4, 1), (5, 2), (6, 2), (7, 0), соответственно. При этом выполняется, например, $(1, 0) \ll (5, 2)$. Кроме того, $\Delta(\sigma, 3) = 7$ и $\Delta(\sigma, 7) = 12$.

Другая возможная эволюция переключателя состоит в том, чтобы оставаться бесконечно долго в позиции *off*, выполняя бесконечно много переходов по задержке. Несмотря на то, что в какой-то момент времени, например, при $v(x) \geq 2$, ребро в позицию *on* активно, оно может постоянно игнорироваться. Аналогично, переключатель может оставаться неограниченно долго в позиции *on*. Такое поведение связано тем, что $inv(off) = inv(on) = \text{true}$. Если изменить переключатель так, что $inv(off)$ станет $y \leq 9$, а $inv(on)$ останется равным true, упомянутый выше путь σ становится недопустимым (так как из состояния (off, v_8) нет переходов), хотя описанный в таблице префикс этого пути все еще допустим.

2.5.3. Синтаксис TCTL

Пусть A – временной автомат, AP – множество атомарных предложений и D – непустое множество часов, которые отличны от часов автомата A ($C \cap D = \varnothing$). $z \in D$ иногда называется *формульными часами*. Поскольку часы образуют часть синтаксиса логики, такая логика иногда называется темпоральной логикой с явными часами.

Для $p \in AP$, $z \in D$ и $\alpha \in \Psi(C \cup D)$ множество TCTL-формул определяется следующим образом:

$$\varphi ::= p \mid \alpha \mid \neg\varphi \mid (\varphi \vee \varphi) \mid z \textbf{ in } \varphi \mid \mathbf{E}[\varphi \textbf{ U } \varphi] \mid \mathbf{A}[\varphi \textbf{ U } \varphi].$$

Отметим, что α – это часовое ограничение над формульными часами и часами автомата. Оно допускает, например, сравнение формульных часов и часов автомата. Булевы операции true, false, $\wedge$, $\rightarrow$ и $\leftrightarrow$ определяются стандартным образом. Часы z в формуле z **in** φ называются *фиксирующим идентификатором* и ограничивают формульные часы z в φ. Интуитивная интерпретация такова: z **in** φ выполняется в состоянии s, если φ выполняется в том же состоянии с одним отличием: часы z в нем принимают значение 0. Например, z **in** $(z = 0)$ является верным предложением, тогда как z **in** $(z > 1)$ – нет. Использование фиксирующего идентификатора полезно в комбинации с until-конструкциями для того, чтобы специфицировать типичные требования своевременности. Обычно нет заинтересованности в формулах, в которых формульные часы из D являются свободными, а не связанными какими-либо фиксирующими идентификаторами. Для

простоты в дальнейшем изложении будем предполагать, что все TCTL-формулы являются замкнутыми. Это означает, что все формульные часы в них связаны фиксирующими идентификаторами. Например, такие формулы, как $x \leq 2$ и z **in** $(z - y = 4)$ недопустимы, если x, y – формульные часы ($x, y \in D$). Формулы x **in** $(x \leq 2)$ и z **in** (y **in** $(z - y = 4)$), однако, являются допустимыми.

Так же, как и для CTL, базовые операторы TCTL являются until-формулами с экзистенциальными и универсальными квантификаторами над путями. Формулы **EF**, **EG** и т. д. являются производными от этих until-формул, как и ранее. Отметим, что в TCTL нет временных вариантов **EX** и **AX**. Причина этого будет ясна при рассмотрении семантики логики. Привязка операторов идентична случаю CTL.

При использовании фиксирующего идентификатора такие темпоральные операторы CTL, как $\mathbf{E}[\varphi\ \mathbf{U}\ \psi]$, $\mathbf{EF}\ \varphi$ и т. д. могут быть снабжены часовым ограничением лаконичным способом. Например, формула

$$\mathbf{A}[\varphi\ \mathbf{U}_{\leq 7}\ \psi]$$

означает, что вдоль любого пути, начинающегося из текущего состояния, свойство φ непрерывно выполняется до тех пор, пока в течение 7 временных единиц не станет верным ψ. Это может быть определено как

$$z \textbf{ in } \mathbf{A}[\varphi\ \mathbf{U}\ (\psi\ \wedge\ z \leq 7)].$$

Аналогично, формула $\mathbf{EF}_{<5}\ \varphi$ означает, что вдоль некоторого пути, начинающегося из текущего состояния, свойство φ становится верным в течение 5 временных единиц, и она определяется как z **in** $\mathbf{EF}(z < 5\ \wedge\ \varphi)$, где **EF** определено, как ранее. Говоря иначе, $\mathbf{EF}_{<c}\ \varphi$ означает, что φ-состояние достижимо в течение c временных единиц. Двойственное выражение $\mathbf{AF}_{<c}\ \varphi$ выполняется, если все вычисления ведут к φ-состоянию в течение c временных единиц.

Пример. Пусть множество атомарных предложений AP равно $\{b = 1, b < 2, b \geq 3\}$. Примерами TCTL-формул являются $\mathbf{E}[(b < 2)\mathbf{U}_{\leq 21}(b \geq 3)]$, $\mathbf{AF}_{\geq 1}(b < 2)$ и $\mathbf{EF}_{<7}[\mathbf{EF}_{<3}(b = 1)]$. Формула $\mathbf{AX}_{<2}(b = 1)$ не является TCTL-формулой, так как в логике нет оператора следования. Формула $\mathbf{AF}_{\leq \pi}(b < 2)$ не является допустимой TCTL-формулой, так как $\pi \notin \mathbb{N}$. По этой же причине $\mathbf{AF}_{=\frac{2}{3}}\left[\mathbf{AG}_{<\frac{4}{5}}(b < 2)\right]$ также не допускается. Однако при использовании подходящего масштабирования эта формула может быть преобразована в допустимую TCTL-формулу $\mathbf{AF}_{=5}[\mathbf{AG}_{<6}(b < 2)]$.

Отметим, что невозможно записать временное свойство ограниченного отклика такое, как «существует некоторое неизвестное время t такое, что если φ выполняется, то в течение времени t свойство ψ выполняется». Например, $\exists t$: z **in** (**AG**[$(b = 1) \rightarrow$ **AF**($z < t \ \wedge \ b \geq 3$)]). Такие квантификации по времени делают проверку моделей неразрешимой.

2.5.4. Семантика TCTL

Вспомним, что интерпретация темпоральной логики определяется в терминах модели. Для LTL такая модель состоит из (непустого) множества S состояний, тотальной функции следования R на состояниях и присваивания *Label* атомарных предложений состояниям. Для CTL функция R превращается в отношение и принимается во внимание ветвящаяся природа. Для TCTL вдобавок должно быть введено представление о реальном времени. Базовая идея состоит в том, чтобы рассматривать *состояние* – позицию и часовую оценку. Позиция определяет, какие атомарные предложения верны (с помощью маркирующего отношения *Label*), а часовая оценка используется для определения справедливости часовых ограничений в рассматриваемой формуле. Так как часовые ограничения могут содержать, помимо часов из автомата, формульные часы, для определения справедливости предложений об этих часах используется дополнительная часовая оценка. Таким образом, справедливость определяется для состояния $s = (l, v)$ и формульной часовой оценки w. Будем обозначать через $v \cup w$ оценку, которая для часов из автомата x равна $v(x)$ и для формульных часов z равна $w(z)$.

Семантика TCTL определяется отношением выполняемости (обозначаемым $\models$), которое связано с системой переходов M, состоянием (позицией и часовой оценкой над часами автомата), часовой оценкой над формульными часами, встречающимися в φ, и самой формулой φ. Будем записывать $(M, (s, w), \varphi) \in \models$ таким образом: $M, (s, w) \models \varphi$. Выражение $M, (s, w) \models \varphi$ верно, если и только если формула φ выполняется в состоянии s модели M над формульной часовой оценкой w. Будем опускать модель M в случаях, когда она ясна из контекста.

Состояние $s = (l, v)$ удовлетворяет φ, если «расширенное» состояние (s, w) удовлетворяет φ, где w – часовая оценка такая, что $w(z) = 0$ для всех формульных часов z.

Пусть $p \in AP$ – атомарное предложение, $\alpha \in \Psi(C \cup D)$ – часовое ограничение над $C \cup D$, $M = (S, \longrightarrow)$ – бесконечная система

переходов, $s \in S$, $w \in V(D)$ и φ, ψ – TCTL-формулы. Отношение выполняемости $\models$ определяется следующим образом:

$$
\begin{aligned}
s, w &\models p && \Leftrightarrow p \in \mathit{Label}(s);\\
s, w &\models \alpha && \Leftrightarrow v \cup w \models \alpha;\\
s, w &\models \neg\varphi && \Leftrightarrow \neg(s, w \models \varphi);\\
s, w &\models (\varphi \vee \psi) && \Leftrightarrow (s, w \models \varphi) \vee (s, w \models \psi);\\
s, w &\models z\ \mathbf{in}\ \varphi && \Leftrightarrow s, [z]w \models \varphi;\\
s, w &\models \mathbf{E}[\varphi\ \mathbf{U}\ \psi] && \Leftrightarrow \exists\, \sigma \in P_M^{\infty}(s): \exists\, (i, d) \in \mathit{Pos}(\sigma):\\
& && \quad (\sigma(i, d), w + \Delta(\sigma, i) + d \models \psi \ \wedge\ (\forall (j, d') \ll (i, d):\\
& && \quad \sigma(j, d'), w + \Delta(\sigma, j) + d' \models \varphi \vee \psi));\\
s, w &\models \mathbf{A}[\varphi\ \mathbf{U}\ \psi] && \Leftrightarrow \forall\, \sigma \in P_M^{\infty}(s): \exists\, (i, d) \in \mathit{Pos}(\sigma):\\
& && \quad (\sigma(i, d), w + \Delta(\sigma, i) + d \models \psi \ \wedge\ (\forall (j, d') \ll (i, d):\\
& && \quad \sigma(j, d'), w + \Delta(\sigma, j) + d' \models \varphi \vee \psi)).
\end{aligned}
$$

Для атомарных предложений, отрицаний и дизъюнкций семантика определена аналогичным образом, как и ранее. Часовое ограничение α верно в (s, w), если все значения часов в v (оценке из s) и в w удовлетворяют α. Формула z **in** φ верна в (s, w), если φ верно в (s, w'), где w' получается из w путем обнуления z. Формула $\mathbf{E}[\varphi\ \mathbf{U}\ \psi]$ верна в (s, w), если существует расходящийся путь σ, начинающийся в состоянии s, который удовлетворяет ψ в некоторой позиции пути и удовлетворяет $\varphi \vee \psi$ во всех предыдущих позициях пути. Формула $\mathbf{A}[\varphi\ \mathbf{U}\ \psi]$ верна в (s, w), если *все* расходящиеся пути, стартующие в состоянии s, удовлетворяют описанному такому же условию.

Следует обратить внимание на то, что в этом последнем аспекте логика TCTL *принципиально отличается* от нетаймированных темпоральных логик. Причина такой трактовки состоит в том, что часовые ограничения в TCTL определяют не одно, а сразу целый класс состояний (ниже будет показано, что эти состояния неотличимы друг от друга). Например, рассмотрим временной автомат на рис. 2.30 (а). Запишем для него TCTL-свойство $s_0, w \models \mathbf{E}[(x \le 1)\mathbf{U}(x > 1)]$. При неограниченном ожидании в стартовой позиции свойство $x > 1$ обязательно наступит, а до него всегда будет выполняться $x \le 1$. Поэтому интуиция подсказывает, что данная формула должна быть верна в начальном состоянии. Тем не менее, если бы семантическая интерпретация этой формулы была такой же, как в нетаймированном случае, то возникла бы следующая ситуация: на пути присутствуют состояния s вида (l, x^*), для которых $x^* > 1$, но у любого такого состояния обязательно имеются предшественники, для которых формула $x \le 1$ не выполняется. Такими предшественниками

будут любые состояния вида (l, x), где $1 < x < x^*$. Истинности рассматриваемой формулы мешает тот факт, что вещественный интервал не содержит свою нижнюю грань. Это означает, что если в определении семантики формулы $\mathbf{E}[\varphi\ \mathbf{U}\ \psi]$ заменить условие $\varphi \vee \psi$ на φ, получив тем самым интерпретацию, аналогичную классической, то свойство $s_0, w \models \mathbf{E}[(x \le 1)\mathbf{U}(x > 1)]$ для указанного временного автомата выполняться не будет. Такая специфика вызвала бы неудобства при работе с темпоральными формулами, и поэтому для формулы вида $\mathbf{E}[\varphi\ \mathbf{U}\ \psi]$ выбрана другая семантика, которая учитывает особенности вещественных интервалов и в которой такая формула, как $\mathbf{E}[(x \le 1)\mathbf{U}(x > 1)]$, выполняется. Эта семантика для формул с часовыми ограничениями оказывается более удачной, в отличие от нетаймированных логик, в которых она эквивалентна классической семантике. Действительно, на *LTL-моделях* классические формулы пути $[\alpha\ \mathbf{U}\ \beta]$ и $[(\alpha \vee \beta)\mathbf{U}\ \beta]$ эквивалентны (при этом вторая из них в таймированном случае соответствует той интуиции, которая вложена в семантику TCTL-формул $\mathbf{E}[\varphi\ \mathbf{U}\ \psi]$ и $\mathbf{A}[\varphi\ \mathbf{U}\ \psi]$). Для принятой семантки таймированных формул на временных автоматах эквивалентными являются, соответственно, пары формул $\mathbf{E}[\varphi\ \mathbf{U}\ \psi]$ и $\mathbf{E}[(\varphi \vee \psi)\mathbf{U}\ \psi]$, $\mathbf{A}[\varphi\ \mathbf{U}\ \psi]$ и $\mathbf{A}[(\varphi \vee \psi)\mathbf{U}\ \psi]$.

Рассмотрим теперь в качестве примера формулу $\mathbf{E}[\varphi\ \mathbf{U}_{<12}\ \psi]$ и состояние s в M. Эта формула означает, что существует расходящийся путь (начинающийся в s), который имеет начальный префикс, продлевающийся менее 12 единиц времени, такой, что ψ выполняется в конце префикса и φ выполняется во всех промежуточных состояниях. Формула $\mathbf{A}[\varphi\ \mathbf{U}_{<12}\ \psi]$ требует, чтобы это свойство было верным для всех расходящихся путей, начинающихся в состоянии s.

Расширение выразительности CTL аспектами реального времени требует определенных затрат. Вспомним, что задача выполнимости для некоторой логики состоит в следующем: существует ли модель M (для этой логики) такая, что $M \models \varphi$ для заданной формулы φ? Для LTL и CTL выполнимость разрешима, но для TCTL оказывается, что задача выполнимости *сильно неразрешима*. В действительности, для большинства логик реального времени задача выполнимости неразрешима: только очень слабая арифметика над дискретной временной областью может привести к логике, для которой выполнимость разрешима [40].

Еще одно различие с нетаймированными вариантами состоит в расширении логики операторами *предшествования* – операторами, которые ссылаются на прошлое вместо будущего (как **F**, **U** и **G**). Примерами операторов предшествования являются $\underline{\mathbf{X}}$ (двойственный к **X**) и $\underline{\mathbf{U}}$ (двойственный к **U**). Известно, что расширение CTL и LTL

операторами предшествования не повышает их выразительности. Главная причина рассмотрения таких операторов для этих логик состоит в увеличении удобства спецификации – некоторые свойства легче формулировать, когда включены операторы предшествования (при этом формулы получаются значительно короче). Для TCTL, однако, это не так: расширение TCTL операторами предшествования увеличивает ее выразительность [41]. Проверка моделей для такой логики, однако, все еще возможна, и ее сложность в худшем случае не увеличивается.

2.5.5. Спецификация временных свойств в TCTL

В этом разделе будут рассмотрены некоторые распространенные временные требования для систем, критичных ко времени, и эти свойства будут формально специфицированы в TCTL.

1. Требование *максимальной задержки*: специфицирует максимальную задержку между наступлением события и реакцией. Например, за каждой передачей сообщения следует ответ в течение 5 единиц времени. Формально:
   $\mathbf{AG}[send(m) \rightarrow \mathbf{AF}_{<5}\, receive(r_m)]$,
   где предполагается, что m и r_m уникальны, и что $send(m)$ и $receive(r_m)$ выполняются, если m послано, и, соответственно, если r_m получено, и не выполняются в противном случае.
2. Требование *точности*: специфицирует точную задержку между событиями. Например, существует вычисление, в течение которого задержка между передачей m и получением ответа составляет ровно 11 единиц времени. Формально:
   $\mathbf{EG}[send(m) \rightarrow \mathbf{AF}_{=11}\, receive(r_m)]$.
3. Требование *периодичности*: специфицирует условие, что событие наступает в течение определенного периода. Например, рассмотрим машину, которая делает отверстия на движущемся ремне, который перемещается с постоянной скоростью. Для того чтобы обеспечить равное расстояние между последовательными отверстиями на ремне, от машины требуется делать отверстия с периодом, например, в 25 временных единиц. Это периодическое поведение можно попытаться специфицировать формулой:
   $\mathbf{AG}[\mathbf{AF}_{=25}\, putbox]$,
   где атомарное предложение *putbox* выполняется, если машина делает отверстие на ремне, и не выполняется в противном случае. Такая формулировка, однако, предписывает делать дополнительные отверстия на ремне между двумя

последовательными созданиями отверстий, интервал между которыми 25 единиц времени. Например, машина, которая удовлетворяет вышеуказанной формулировке, должна делать отверстия на ремне не только в моменты времени 25, 50, 75, …, но и, например, в моменты времени 35, 60, 85, … Поэтому более точная формулировка требования периодичности такова:

$\mathbf{AG}[putbox \rightarrow \neg putbox\ \mathbf{U}_{=25}\ putbox]$.

Это требование специфицирует задержку в 25 единиц времени между последовательными созданиями отверстий и вдобавок требует, чтобы такие события не случались в промежутках[8].

4. Требование *минимальной задержки*: специфицирует минимальную задержку между событиями. Например, для того чтобы убедиться в безопасности железнодорожной системы, задержка между двумя поездами на переезде должна быть сделана как минимум 180 единиц времени. Пусть *tac* – атомарное предложение, которое выполняется, когда поезд на переезде. Требование минимальной задержки может быть сформулировано следующим образом:

   $\mathbf{AG}[tac \rightarrow \neg tac\ \mathbf{U}_{\geq 180}\ tac]$.

   Причина использования until-конструкции аналогична предыдущему случаю для периодичности.

5. Требование *интервальной задержки*: специфицирует условие на событие, которое должно наступить в течение определенного интервала после другого события. Предположим, например, что с целью улучшения пропускной способности железнодорожной системы требуется, чтобы поезда находились на расстоянии максимум в 900 временных единиц. Безопасность железнодорожной системы должна быть сохранена. Формально, можно легко расширить предыдущее требование минимальной задержки:

   $\mathbf{AG}[tac \rightarrow (\neg tac\ \mathbf{U}_{\geq 180}\ tac \wedge \neg tac\ \mathbf{U}_{\leq 900}\ tac)]$.

   В качестве альтернативы можно записать:

   $\mathbf{AG}[tac \rightarrow \neg tac\ \mathbf{U}_{=180}(\mathbf{AF}_{\leq 720}\ tac)]$.

   Эта формула специфицирует условие, что после того как поезд оказывается на переезде, проходит 180 единиц времени (требование безопасности) перед тем, как прибудет следующий поезд, причем следующий поезд прибывает в течение

---

[8] Можно поинтересоваться, почему такая конструкция не используется для требования точности. Это следует из того факта, что r_m уникально. Следовательно, *receive*(r_m) может выполняться максимум однократно. Это неверно для предиката *putbox*, так как создание отверстий на ремне предполагается повторяющимся процессом.

720 + 180 = 900 временных единиц (требование пропускной способности).

2.5.6. Эквивалентность часовых оценок

Отношение выполняемости $\models$ для TCTL определено не в терминах временных автоматов, а в терминах бесконечных систем переходов. Задача проверки моделей для временных автоматов может легко быть определена в терминах $M(A)$ – системы переходов над A. Пусть $s_0 = (l_0, v_0)$ – *начальное состояние* $M(A)$. Здесь l_0 – начальная позиция A, и v_0 – часовая оценка, которая присваивает нулевое значение всем часам из A. Теперь можно формализовать утверждение о том, что временной автомат удовлетворяет заданной TCTL-формуле.

Для TCTL-формулы φ и временного автомата A будем писать $A \models \varphi$, если и только если $M(A), (s_0, w_0) \models \varphi$, где $w_0(y) = 0$ для всех формульных часов y.

Рассматриваемая задача проверки моделей состоит в определении, удовлетворяет ли заданный временной автомат A некоторой TCTL-формуле φ. Базой для проверки модели автомата A является система переходов $M(A)$. Главная проблема, однако, в том, что пространство состояний $M(A)$ – множество $L \times V(C)$ – является *бесконечным*! Это проиллюстрировано на рис. 2.33 (например, для временного автомата с одной позицией, инвариант в которой равен true).

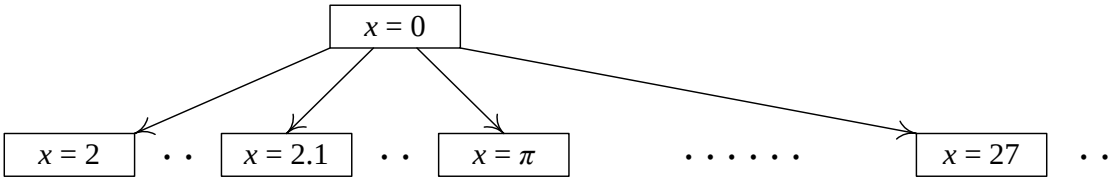


Рис. 2.33. Пример бесконечной системы переходов

Каким образом можно эффективно проверять, верна ли заданная формула для $M(A)$, если требуется рассматривать бесконечное число состояний?

Из рассмотрения описанных выше временных автоматов следует, что число состояний временного автомата бесконечно. Ключевая идея, которая упрощает проверку моделей временных автоматов (и подобных структур), состоит в том, чтобы ввести подходящее отношение эквивалентности ($\approx$) на часовых оценках такое, что обеспечиваются: *корректность* (эквивалентные часовые оценки удовлетворяют одинаковым TCTL-формулам) и *конечность* (число классов эквивалентности над $\approx$ конечно).

Свойство конечности позволяет заменить бесконечное множество часовых оценок конечным множеством (множествами). Вдобавок свойство корректности гарантирует, что для модели M эти часовые оценки удовлетворяют тем же TCTL-формулам. Таким образом, не существует TCTL-формулы, которая различает бесконечную и эквивалентную ей конечную часовую оценку. Формально, для любого временного автомата A это приводит к следующему соотношению:

$$M(A), ((l, v), w) \models \varphi \text{ если и только если } M(A), ((l, v'), w') \models \varphi$$
$$\text{для } v \cup w \approx v' \cup w'.$$

Ключевое наблюдение, которое приводит к определению этого отношения эквивалентности, состоит в том, что пути временного автомата A, начинающиеся в состояниях, которые согласуются в целых частях значений всех часов и в порядке дробных частей всех часов, очень похожи. В частности, поскольку временные ограничения в TCTL-формулах ссылаются только на натуральные числа, не существует TCTL-формулы, которая различает эти «почти одинаковые» пути. Грубо говоря, две часовых оценки эквивалентны, если они удовлетворяют двум вышеупомянутым ограничениям. Кроме того, если показатель часов превышает максимальную константу, с которой он сравнивается, то его точная величина не имеет значения. Это наблюдение позволяет сделать число классов эквивалентности, полученных таким образом, не только счетным, но и конечным.

Тот факт, что число классов эквивалентности конечно и эквивалентные TCTL-модели временных автоматов удовлетворяют одинаковым формулам, наводит на мысль о том, что проверка моделей должна выполняться на базе этих классов. Конечная модель, которая получается из временного автомата A, называется его *регионным автоматом*. Соответственно, классы эквивалентности над $\approx$ называются *регионами*. Проверка модели временного автомата, таким образом, сводится к проверке моделей связанного с ним конечного регионного автомата. Эта сводная задача может быть решена тем же путем, как было показано ранее для проверки моделей ветвящейся темпоральной логики CTL – путем итеративной маркировки состояний подформулами проверяемого свойства (будет рассмотрено позднее). Поэтому, грубо говоря, *проверка модели временного автомата относительно TCTL-формулы эквивалентна проверке модели его регионного автомата относительно CTL-формулы*.

Подведя итог, изложим метод проверки моделей для TCTL-формулы φ над временным автоматом A. Обозначим класс эквивалентности часовой оценки v над отношением $\approx$ через $[v]$. Как будет указано ниже, регионный автомат зависит также от часовых ограничений в

проверяемой формуле, но учет этой зависимости с целью упрощения здесь опускается.

В дальнейшем будем обозначать $(l, [v])$ через $[s]$ для $s = (l, v)$. Кроме того, будем писать $[s, w]$ как сокращение для $(l, [v \cup w])$, где w – часовая оценка над формульными часами. Также для $s = (l, v)$ и $s' = (l', v')$ будем использовать запись $s, w \approx s', w'$, подразумевая под этим одновременное выполнение равенства $s = s'$ и условия $v \cup w \approx v' \cup w'$.

Базовый способ проверки TCTL-моделей для временного автомата состоит в следующем:

1. Определить классы эквивалентности (регионы) над отношением $\approx$.
2. Построить регионный автомат $\boldsymbol{R}(A)$.
3. Применить процедуру проверки моделей CTL к $\boldsymbol{R}(A)$.
4. $A \models \varphi$ если и только если $[s_0, w_0] \in Sat^{\boldsymbol{R}}(\varphi)$.

В следующем разделе будет рассмотрено построение регионного автомата $\boldsymbol{R}(A)$. Начнем с обсуждения в деталях идей рассмотренного выше отношения $\approx$. Так как идея эквивалентности имеет ключевую важность для этого подхода, ее определение будет пояснено примерами.

Далее при записи часовых оценок используется следующая нотация. Для часовой оценки v: $C \to \mathbb{R}^+$ и часов $x \in C$ пусть $v(x) = [x] + \{x\}$, где $[x]$ – это целая часть значения часов и $\{x\}$ – дробная часть. Например, для $v(x) = 2.134$ имеем $[x] = 2$ и $\{x\} = 0.134$. Построение отношения $\approx$ на часовых оценках исходит из четырех наблюдений, которые последовательно ведут к улучшению представления об эквивалентности. Пусть v и v' – две часовых оценки, определенные над множеством часов C.

Первое наблюдение. Рассмотрим временной автомат, изображенный на рис. 2.34, и следующие состояния автомата: (l_0, v) и (l_0, v') с $v(x) = 3.2$ и $v'(x) = 3.7$. В обоих состояниях ребро из позиции l_0 в l_1 активно в любой момент времени, превосходящий или равный 2. Дробные части $v(x)$ и $v'(x)$ не влияют на его активность. Аналогично, если $v(x) = 1.2$ и $v'(x) = 1.7$, то ребро блокировано для обеих часовых оценок и дробные части снова несущественны. Поскольку, вообще говоря, часовые ограничения имеют вид $x \sim c$, где $c \in \mathbb{N}$, можно считать, что только целые части часов имеют значение. Это приводит к первому предложению об эквивалентности часов:

- $v \approx v'$, только если $[v(x)] = [v'(x)]$ для всех $x \in C$. (*)

Эта достаточно простая идея приводит к счетному (но все еще бесконечному) числу классов эквивалентности, но это слишком грубое деление. Причина состоит в том, что это деление отождествляет часовые оценки, которые различимы в терминах TCTL-формул.

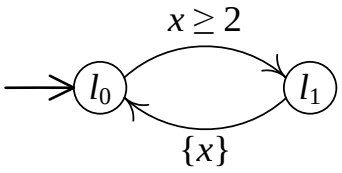


Рис. 2.34. Часовые оценки, согласующиеся в целых частях, эквивалентны

Второе наблюдение. Рассмотрим временной автомат на рис. 2.35 и следующие его состояния: $s = (l_0, v)$ и $s' = (l_0, v')$ с $v(x) = 0.4$, $v'(x) = 0.2$ и $v(y) = v'(y) = 0.3$. В соответствии с рассмотренным выше предложением, можно иметь $v \approx v'$, так как $[v(x)] = [v'(x)] = 0$ и то же самое для часов y. Однако из состояния s можно достичь позиции l_2, тогда как это невозможно для состояния s'. Это можно объяснить следующим образом. Рассмотрим состояние s через 0.6 единицы времени. Его часовая оценка $v + 0.6$ означает $(v + 0.6)(x) = 1$ и $(v + 0.6)(y) = 0.9$. В этой часовой оценке ребро, которое ведет из l_0 в l_1, активно. Если после прохождения этого ребра в позиции l_1 последовательно подождать 0.1 единицы времени, то ребро, которое ведет в позицию l_2, станет активным. Для s' не существует аналогичного сценария. Чтобы попасть из состояния s' в позицию l_1, показания часов x должны быть увеличены на 0.8 единицы времени. Однако теперь $v'(y) = 1.1$ и ребро в l_2 будет постоянно блокировано. Важное различие между v и v' состоит в том, что $\{v(x)\} > \{v(y)\}$, но $\{v'(x)\} < \{v'(y)\}$. Это приводит к следующему расширению предложения (*):

- $v(x)\} \leq \{v(y)\}$ если и только если $\{v'(x)\} \leq \{v'(y)\}$ для всех $x, y \in C$. (**)

Результирующая эквивалентность снова приводит к счетному числу классов эквивалентности, но все еще слишком грубая.

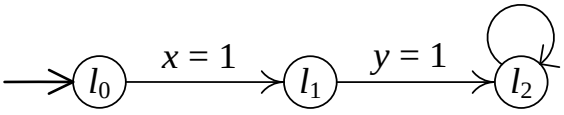


Рис. 2.35. Порядок дробных частей часовых оценок в различных часах важен

Третье наблюдение. Рассмотрим временной автомат на рис. 2.36 и следующие состояния: $s = (l_0, v)$ и $s' = (l_0, v')$ с $v(x) = 0$ и $v'(x) = 0.1$.

В соответствии с предложениями (*) и (**) имеем $v \approx v'$, однако, в состояниях s и s' выполняются разные формулы. Например, $s, w \models \mathbf{EF}_{=1}\, p$, но $s', w \not\models \mathbf{EF}_{=1}\, p$, где p – атомарное предложение, которое верно только в позиции l_1. Главное различие между v и v' состоит в том, что часы x в s в точности равны 0, тогда как в состоянии s' они уже прошли через 0. Это приводит к расширению предложений (*) и (**) следующим условием:

- $\{v(x)\} = 0$ если и только если $\{v'(x)\} = 0$ для всех $x \in C$. (***)

Результирующая эквивалентность больше не является слишком грубой, но все еще ведет к бесконечному, а не конечному числу классов эквивалентности.

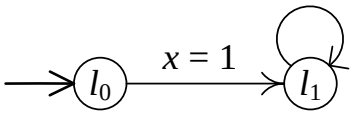


Рис. 2.36. Целочисленные значения часов должны быть в соответствии друг с другом

Четвертое наблюдение. Пусть c_x – максимальная константа, с которой часы x сравниваются в часовом ограничении вида $x \sim c$ или $x - y \sim c$ в рассматриваемом временном автомате. Это часовое ограничение появляется в активирующем условии, связанном с ребром, или в инварианте. Поскольку c_x является наибольшей константой, с которой x сравнивается, отсюда следует, что если $v(x) > c_x$, то фактическое значение x несущественно. Самого факта, что $v(x) > c_x$, достаточно для определения активности всех ребер во временном автомате.

В комбинации с тремя вышеуказанными условиями это наблюдение приводит к *конечному* числу классов эквивалентности. Для определения этих классов важны целые части часов вплоть до определенной границы **(**(*) и четвертое наблюдение**)** плюс порядок их дробных частей (**) и факт, равны дробные части нулю или нет (***). Последние два условия теперь также важны, только в том случае, когда часы не превышают их границу. Например, если $v(y) > c_y$, то равенство дробной части y нулю не играет роли, так как значение y в любом случае больше не будет проверено.

В конечном счете, приходим к следующему определению эквивалентности часов.

Пусть A – временной автомат с множеством часов C и $v, v' \in V(C)$. Тогда $v \approx v'$, если и только если одновременно выполняются четыре условия:

1. $[v(x)] = [v'(x)]$ или $v(x) > c_x$ и $v'(x) > c_x$ для всех $x \in C$.
2. $\{v(x)\} \le \{v(y)\}$, если и только если $\{v'(x)\} \le \{v'(y)\}$ для всех $x, y \in C$ таких, что $v(x) \le c_x$ и $v(y) \le c_y$.
3. $\{v(x)\} = 0$, если и только если $\{v'(x)\} = 0$ для всех $x \in C$ при $v(x) \le c_x$.
4. Для любой подформулы вида $x - y \sim c$ в часовых ограничениях и любого $\sim\sim \in \{<, \le\}$ неравенство $v(x) - v(y) \sim\sim c$ верно, если и только если $v'(x) - v'(y) \sim\sim c$.

Это определение может быть изменено непосредственным образом, чтобы отношение $\approx$ было определено на множестве часов C' таком, что $C \subseteq C'$. Это охватит случай, когда в C' также включены формульные часы (помимо часов автомата). Для формульных часов z в рассматриваемом свойстве φ значение c_z – наибольшая константа, с которой z сравнивается в часовом ограничении вида $z \sim c$ или $z - y \sim c$, содержащемся в формуле φ. Например, для формулы z **in AF**$[(p \wedge z \le 3) \vee (q \wedge z > 4) \wedge (z - x > 5)]$ значение c_z равно 5.

Пример. Рассмотрим временной автомат на рис. 2.30 (а). Он имеет множество часов $\{x\}$ с $c_x = 2$, так как единственное часовое ограничение $x \ge 2$. Будем постепенно строить регионы этого автомата путем рассмотрения каждого условия в определении отношения $\approx$ по отдельности. Часовые оценки v и v' эквивалентны, если $v(x)$ и $v'(x)$ принадлежат одному и тому же классу эквивалентности на вещественной полупрямой (в общем случае для n часов это приводит к рассмотрению n-мерного пространства над $\mathbb{R}^+$). Для удобства обозначим через $[x \sim c]$ сокращение $\{x \mid x \sim c\}$ для натурального c и операции сравнения $\sim$.

1. Требование того, что $[v(x)] = [v'(x)]$, приводит к следующему разбиению вещественной полупрямой:
   $[0 \le x < 1]$, $[1 \le x < 2]$, $[2 \le x < 3]$, $[3 \le x < 4]$, …
2. Так как $c_x = 2$, для данного автомата не следует делать различия между оценками $v(x) = 3$ и $v'(x) = 27$. Таким образом, классы эквивалентности, которые получаются при рассмотрении первого требования в определении отношения $\approx$, таковы:
   $[0 \le x < 1]$, $[1 \le x < 2]$, $[x = 2]$ и $[x > 2]$.
3. В соответствии со вторым и четвертым требованиями, порядок часов и их дробных частей должен быть сохранен. В данном случае это требование тривиально выполняется, так как имеются только одни часы. Следовательно, при рассмотрении данного требования новые классы эквивалентности не вводятся.

4. Условие, что $\{v(x)\} = 0$ если и только если $\{v'(x)\} = 0$ при $v(x) \le c_x$, приводит к разбиению, например, класса $[0 \le x < 1]$ на классы $[x = 0]$ и $[0 < x < 1]$. Аналогично разбивается класс $[1 \le x < 2]$. Класс $[x > 2]$ далее разбивать не требуется, так как для этого класса нарушается условие $v(x) \le c_x$. В качестве результата получаем для простого временного автомата 6 классов эквивалентности:
   $[x = 0]$, $[0 < x < 1]$, $[x = 1]$, $[1 < x < 2]$, $[x = 2]$ и $[x > 2]$.

Пример. Рассмотрим множество часов $C = \{x, y\}$ с $c_x = 2$ и $c_y = 1$, и пусть часовые ограничения не содержат подформул вида $x - y \sim c$ или $y - x \sim c$. На рис. 2.37 отображено последовательное построение регионов путем рассмотрения каждого условия в определении отношения $\approx$ по отдельности. Рисунок отображает разбиение двумерного квадранта $\mathbb{R}^+ \times \mathbb{R}^+$. Часовые оценки v и v' эквивалентны, если вещественнозначные пары $(v(x), v(y))$ и $(v'(x), v'(y))$ являются элементами одного и того же класса эквивалентности в двумерном пространстве.

1. Требование того, что $[v(x)] = [v'(x)]$ для всех часов из C, приводит, например, к классам эквивалентности $[(0 \le x < 1), (0 \le y < 1)]$ и $[(1 \le x < 2), (0 \le y < 1)]$ и т. д. Условие, что $v(x) > c_x$ и $v'(x) > c_x$ для всех часов из C, приводит к классу эквивалентности $[(x > 2), (y > 1)]$. Это означает, что для любой часовой оценки v, для которой $v(x) > 2$ и $v(y) > 1$, точные значения x и y несущественны. В результате получаются следующие классы эквивалентности:
   $[(0 \le x < 1), (0 \le y < 1)]$, $[(1 \le x < 2), (0 \le y < 1)]$,
   $[(0 \le x < 1), (y = 1)]$, $[(1 \le x < 2), (y = 1)]$,
   $[(0 \le x < 1), (y > 1)]$, $[(1 \le x < 2), (y > 1)]$,
   $[(x = 2), (0 \le y < 1)]$, $[(x > 2), (0 \le y < 1)]$,
   $[(x = 2), (y = 1)]$, $[(x > 2), (y = 1)]$,
   $[(x = 2), (y > 1)]$, $[(x > 2), (y > 1)]$.
   Эти 12 классов отображены на рис. 2.37 (а).

2. Рассмотрим класс эквивалентности $[(0 \le x < 1), (0 \le y < 1)]$, который был получен на предыдущем шаге. Так как порядок дробных частей часов сейчас становится важным, этот класс эквивалентности разбивается на классы $[(0 \le x < 1), (0 \le y < 1), (x < y)]$, $[(0 \le x < 1), (0 \le y < 1), (x = y)]$ и $[(0 \le x < 1), (0 \le y < 1), (x > y)]$. Аналогичное рассуждение применимо к классу $[(1 \le x < 2), (0 \le y < 1)]$. Другие классы более не разбиваются. Например, класс $[(0 \le x < 1), (y = 1)]$ не должен разбиваться, так как порядок на дробных частях часов x и y в данном классе фиксирован, $\{v(x)\} \ge \{v(y)\}$. Класс $[(1 \le x < 2), (y > 1)]$ не разбивается, так как для этого класса нарушается условие $v(x) \le c_x$ и $v(y) \le c_y$. Рис. 2.37 (б) показывает результирующие классы эквивалентности.

3. Наконец, применим третий критерий в определении отношения ≈. В качестве примера рассмотрим класс [$(0 \le x < 1)$, $(0 \le y < 1)$, $(x = y)$], который был получен на предыдущем шаге. Этот класс теперь разбивается на [$(x = 0)$, $(y = 0)$] и [$(0 < x < 1)$, $(0 < y < 1)$, $(x = y)$]. Рис. 2.37 (в) показывает 28 результирующих классов эквивалентности: 6 угловых точек, 14 открытых прямолинейных участков и 8 открытых областей (регионов).
4. Поскольку, согласно условию, подформул вида $x - y \sim c$ в часовых ограничениях нет, больше регионы не разбиваются.

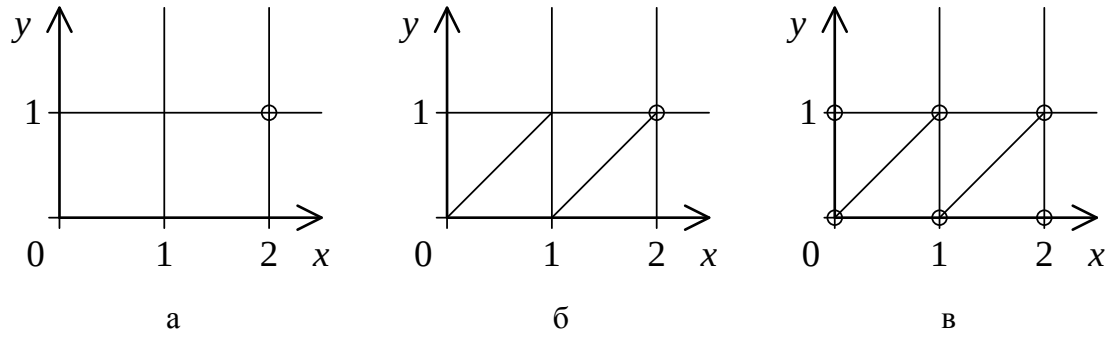


Рис. 2.37. Разбиение $\mathbb{R}^+ \times \mathbb{R}^+$ в соответствии с ≈ для $c_x = 2$ и $c_y = 1$

2.5.7. Регионные автоматы

Классы эквивалентности над отношением ≈ являются базой для проверки моделей временных автоматов. Комбинация такого класса и позиции называется *регионом*.

Регион r – это пара (l, [v]) с позицией $l \in L$ и оценкой $v \in V(C)$.

Так как существует только конечное число классов эквивалентности над отношением ≈ и любой временной автомат содержит лишь конечное число позиций, число регионов конечно. Следующий результат утверждает, что состояния, принадлежащие одному и тому же региону, удовлетворяют одинаковым TCTL-формулам. Это важный результат для проверки моделей реального времени.

Теорема [39]. Пусть $s, s' \in S$ и $w, w' \in V(D)$ таковы, что $s, w \approx s', w'$. Тогда для любой TCTL-формулы φ имеем: $M(A), (s, w) \models \varphi$, если и только если $M(A), (s', w') \models \varphi$.

В соответствии с этим результатом не требуется различать эквивалентные состояния s и s', так как их не может различить никакая TCTL-формула. Это дает критерий корректности при использовании классов эквивалентности над отношением ≈ (регионов), в качестве базы для проверки моделей. Используя регионы

в качестве состояний, можно построить конечный граф, называемый *регионным автоматом*. Этот граф имеет структуру модели Крипке и состоит из *достижимых* регионов-состояний и переходов между регионами. Данные переходы соответствуют либо прохождению времени, либо ребрам в первоначальном временном автомате.

Вначале рассмотрим построение регионного автомата на примере. Между регионами могут быть два типа переходов. Это либо протекание времени, либо переходы в рассматриваемом временном автомате. Протекание времени будем обозначать сплошными стрелками, а переходы во временном автомате – пунктирными стрелками.

Пример. Рассмотрим временной автомат с единственным ребром на рис. 2.30 (а). Максимальная константа, с которой сравнивается x, – это 2. Поэтому $c_x = 2$. Регионный автомат изображен на рис. 2.38.

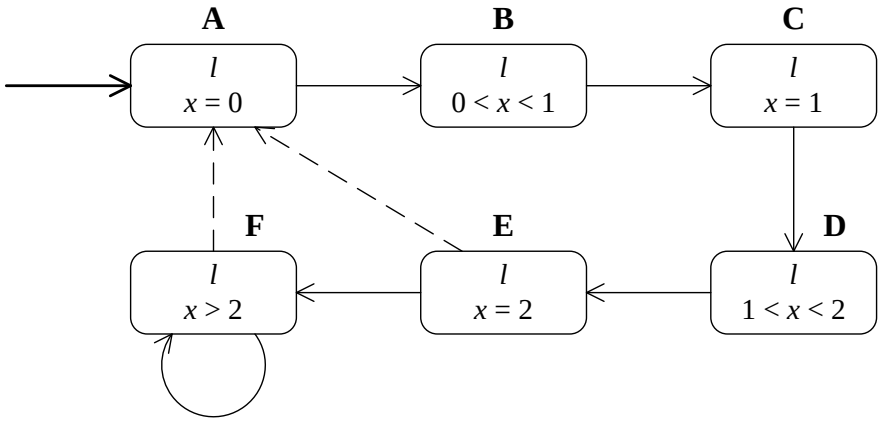


Рис. 2.38. Регионный автомат для простого временного автомата

Так как существует только одна позиция, каждый достижимый регион временного автомата находится в позиции l. Регионный автомат содержит два перехода, соответствующих ребру временного автомата. Они оба ведут в стартовый регион **A**. Пунктирные переходы из **E** и **F** в **A** обозначают ребра. Классы, в которых существует возможность ждать в течение любого количества времени без ограничений, оставаясь в текущем классе, называются *неограниченными классами*. В этом примере только один неограниченный класс – **F**, он снабжен сплошным петлевым переходом. В классе **F** часы x могут возрастать без ограничений (при этом можно оставаться в том же классе).

Предположим теперь, что требуется проверить модель над временным автоматом на рис. 2.30 (а) для TCTL-формулы, которая содержит единственные формульные часы z с $c_z = 2$ (часы z не сравниваются в формуле с константой, большей 2), причем в ней нет подформул вида $x - z \sim c$ и $z - x \sim c$. Регионный автомат над $\{x, z\}$ изображен на рис. 2.39.

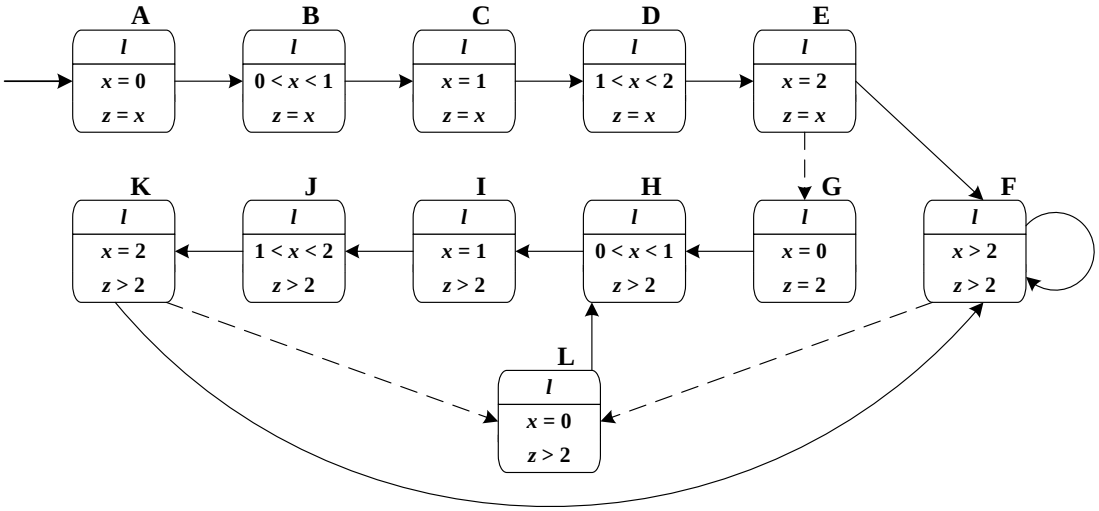


Рис. 2.39. Регионный автомат простого временного автомата для формульных часов z с $c_z = 2$

Отметим, что это фактически предыдущий регионный автомат, расширенный своей «копией»: регионами с **G** по **L**. Эта копия вводится для ограничения $z > 2$. Отметим также, что формульные часы z никогда не сбрасываются. Для формульных часов так и должно быть, так как в регионном автомате нет смысла обнулять формульные часы с тех пор, как они введены.

Пусть r, r' – два различных региона ($r \neq r'$). Регион r' называется *потомком задержки* для региона r (обозначается $r' = delsucc(r)$), если для всех $[s, w] = r$ существует такое $d \in \mathbb{R}^+$, что $s \xrightarrow{d} s'$ и $r' = [s', w + d] = [s + d, w + d]$, причем для всех $0 \leq d' < d$ состояние с часовой оценкой $(s + d', w + d')$ находятся либо в регионе r, либо в регионе r'.

Здесь для $s = (l, v)$ состояние $s + d$ обозначает $(l, v + d)$. Говоря иначе, $[s', w']$ – потомок задержки $[s, w]$, если любая оценка в $[s, w]$ через некоторое время переходит в оценку из $[s', w']$ без возможности покинуть регионы $[s, w]$ и $[s', w']$ в какой-то более ранний момент времени. Отметим, что для каждого региона существует максимум один потомок задержки. Это не должно удивлять читателя, так как потомки задержки соответствуют продвижению времени, а время с необходимостью течет детерминированно.

Пример. Рассмотрим классы эквивалентности временного автомата на рис. 2.30 (а). Регионы, содержащие часы x и дополнительные часы z (рис. 2.39), представляют части двумерного вещественного пространства. Временные переходы соответствуют восходящему движению вдоль диагональной прямой $x = z$. Например, $[x = z = 1]$ – это потомок задержки $[(0 < x < 1), (x = z)]$, а регион $[(1 < x < 2), (x = z)]$ имеет потомка задержки $[x = z = 2]$. Класс $[(x = 1), (z > 2)]$ не является

потомком задержки $[(0 < x < 1), (z = 2)]$ (недостижимого региона, который не изображен), так как существует некоторое вещественное значение d' такое, что в промежутке, спустя d' единиц времени, достигается регион $[(0 < x < 1), (z > 2)]$.

Регион r называется *неограниченным регионом*, если для всех часовых оценок v таких, что $r = [v]$, имеем $v(x) > c_x$ для всех $x \in C$.

В неограниченном регионе все часы превысили максимальную константу, с которой они сравниваются, и, следовательно, все часы могут возрастать без ограничений. На рис. 2.38 и 2.39 неограниченные регионы снабжены сплошными петлевыми переходами.

Регионный автомат теперь состоит из множества состояний (регионов), стартового состояния и двух отношений переходов: одно соответствует переходам по задержке и одно соответствует ребрам рассматриваемого временного автомата.

Для временного автомата A и множества часовых ограничений Ψ (над C и над формульными часами из D) *регионный автомат* $\boldsymbol{R}(A, \Psi)$ – это система переходов $(R, r_0, \rightarrow)$, где

- $R = S/\approx = \{[s, w] \mid s \in S, w \in V(D)\}$;
- $r_0 = [s_0, w_0]$;
- $r \rightarrow r'$, если и только если

  1) $\exists s, s', w$: $(r = [s, w] \wedge r' = [s', w] \wedge s \xrightarrow{\circ} s')$ или
  2) r – неограниченный регион и $r = r'$ или
  3) $r \neq r'$ и $r' = delsucc(r)$.

Отметим, что из этого определения следует, что неограниченные регионы являются единственными регионами, которые имеют петлю, представляющую собой переход по задержке.

Как было указано выше, в системе переходов учитываются только регионы-состояния, достижимые из начального. Таким образом, для регионного автомата вместо множества регионов R и, соответственно, тройки $(R, r_0, \rightarrow)$ рассматривается только его подмножество $R' = \{r \in R \mid r_0 \rightarrow^* r\}$, где $\rightarrow^*$ – это рефлексивное транзитивное замыкание отношения $\rightarrow$. Как следствие, рассматривается тройка $(R', r_0, \rightarrow)$.

Пример. Для того чтобы проиллюстрировать построение регионного автомата, немного изменим автомат для переключателя (рис. 2.40). Для демонстрации инвариантов позиция *on* снабжена инвариантом $y \leq 3$. Чтобы сделать регионный автомат не слишком большим, на

переходах, которые изменяют позицию, расположены другие активирующие условия. В частности, константы сделаны меньше, чем они были в исходном переключателе.

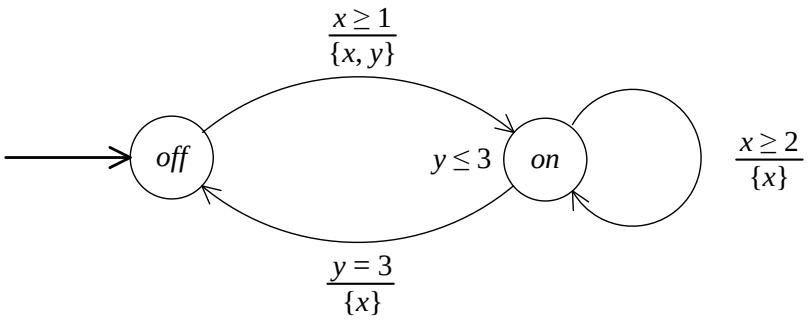


Рис. 2.40. Временной автомат для видоизмененного переключателя

Регионный автомат $\boldsymbol{R}(A, \Psi)$, где A – видоизмененный переключатель, изображен на рис. 2.41. Множество часовых ограничений следующее: $\Psi = \{x \geq 1,\ x \geq 2,\ y = 3,\ y \leq 3\}$ – это множество всех активирующих ограничений и инвариантов в A. Для простоты никакие формульные часы не рассматриваются. Имеем, например, $\mathbf{D} \rightarrow \mathbf{E}$, так как существует переход из состояния $s = (off, v)$, где $v(x) = v(y) > 1$, в состояние $s' = (on, v')$, где $v'(x) = v'(y) = 0$ и $\mathbf{D} = [s]$, $\mathbf{E} = [s']$. Существует переход по задержке $\mathbf{D} \rightarrow \mathbf{D}$, так как регион $[v]$, где $v(x) = v(y) > 1$, является неограниченным. Вывод основан на том факте, что в позиции *off* время может возрастать без ограничений: $inv(off) = \text{true}$.

2.5.8. Проверка моделей для регионных автоматов

При заданном регионном автомате $\boldsymbol{R}(A, \Psi)$ для временного автомата A и TCTL-формулы φ с часовыми ограничениями Ψ в A и φ алгоритм проверки моделей строится так же, как и в нетаймированном CTL. Опишем кратко идею маркировки. Базовая идея проверки моделей состоит в том, чтобы *пометить* каждое состояние (регион) в регионном автомате подформулами φ, которые верны в этом состоянии. Эта маркирующая процедура выполняется итеративно, начиная с пометки состояний подформулами φ *глубины* ноль – атомарными предложениями (включая true и false), которые встречаются в φ. В $(i + 1)$-й итерации маркирующего алгоритма рассматриваются подформулы глубины $i + 1$ и соответственно маркируются состояния. С этой целью используются пометки, уже присвоенные состояниям, для подформул φ глубины не более i $(i \geq 1)$. Маркирующий алгоритм завершается рассмотрением подформулы максимальной глубины – самой формулы φ. Алгоритм представлен в листинге 2.8.

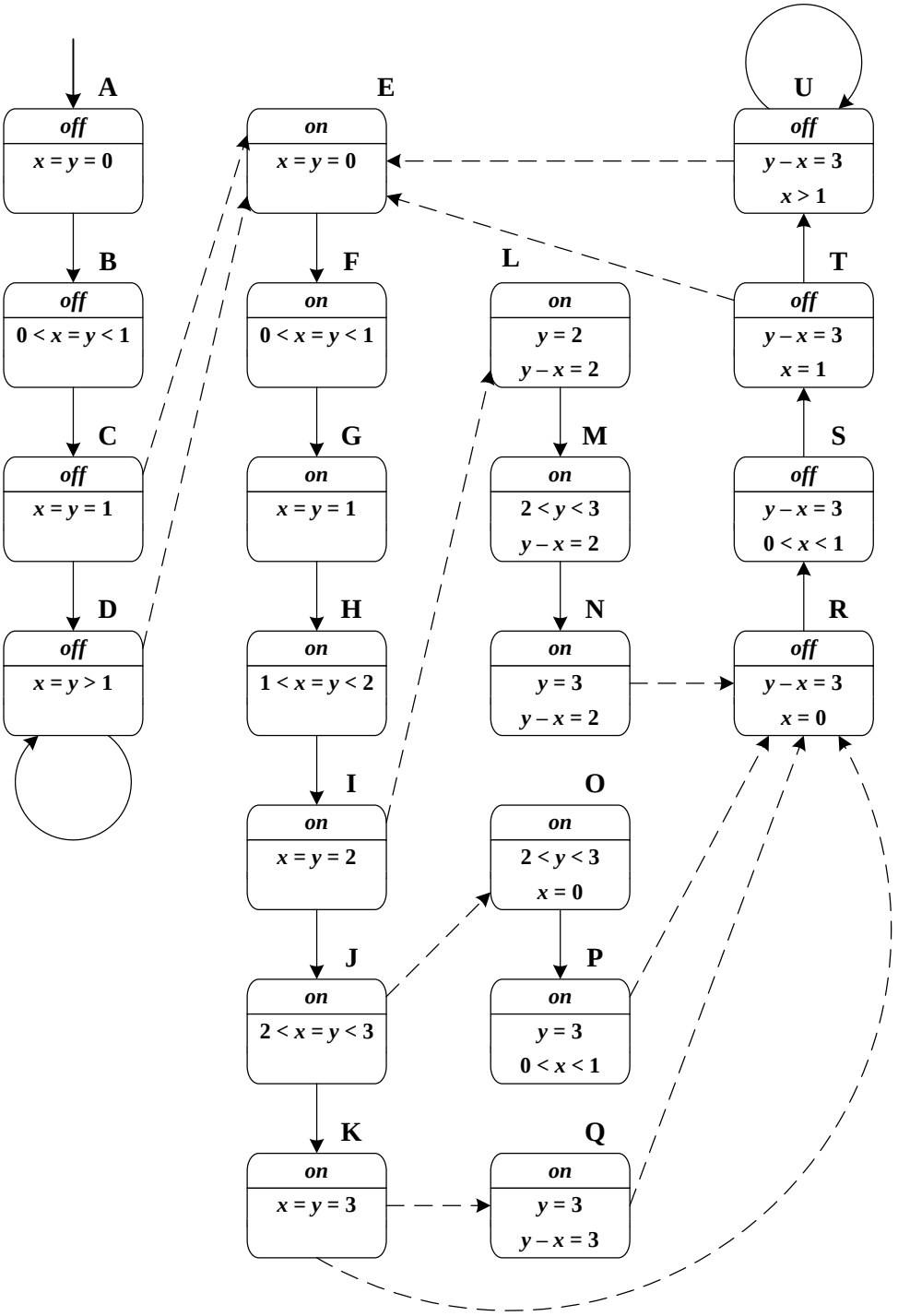


Рис. 2.41. Регионный автомат для переключателя на рис. 2.40

Листинг 2.8. Псевдокод основного алгоритма проверки моделей TCTL

```
set<Region> Sat^R(Formula φ)
{
   if (φ == true)          return S/≈;
   if (φ == false)         return ∅;
   if (φ ∈ AP)             return {[s,w]≈ | φ ∈ Label(s)};
   if (φ == α)             return {[s,w]≈ | (s=(l,v)) ∧
                            ((v ∪ w) ⊨ α)};
   if (φ == ¬φ1)           return (S/≈) \ Sat^R(φ1);
```

```
    if (φ == (φ1 ∨ φ2))       return Sat^R(φ1) ∪ Sat^R(φ2);
    if (φ == z in φ1)         return {[s,w]≈ |
                               (s, [z]w) ∈ Sat^R(φ1)};
    if (φ == E[φ1 U φ2])      return Sat^R_EU(φ1, φ2);
    if (φ == A[φ1 U φ2])      return Sat^R_AU(φ1, φ2);
    // Sat^R(φ) = {[s,w]≈ | M,(s,w) ⊨ φ}
}
```

Задача проверки моделей $A \models \varphi$ или, эквивалентно, $M(A), (s_0, w_0) \models \varphi$ теперь будет решена, если рассмотреть маркировку:

$$M(A), (s_0, w_0) \models \varphi, \text{ если и только если } [s_0, w_0] \in Sat^{R}(\varphi)$$

(корректность данного утверждения следует из приведенной ниже теоремы).

Вычисление $Sat^{R}(\varphi)$ выполняется путем рассмотрения синтаксической структуры φ. Случаи пропозициональных формул (true, false, отрицание, дизъюнкция и атомарные предложения) идентичны таковым в CTL. Для часового ограничения α результат $Sat^{R}(\alpha)$ является множеством регионов $[s, w]$ таких, что w и v (часовая оценка для часов автомата из s) удовлетворяют α. Регион $[s, w]$ принадлежит множеству $Sat^{R}(z \textbf{ in } \varphi)$, если он удовлетворяет φ для w', где $w'(z) = 0$ (формульные часы стартуют в значении 0) и $w'(x) = w(x)$ для $x \neq z$.

Для until-формул определяются вспомогательные функции. Они представлены в листингах 2.9 и 2.10.

Листинг 2.9. Проверка моделей для формулы **E**[φ **U** ψ] над регионами

```
set<Region> Sat^R_EU(Formula φ)
{
   set<Region> Q, Q';
   Q  = Sat^R(ψ);
   Q' = ∅;
   while (Q ≠ Q')
   {
      Q' = Q;
      Q  = Q ∪ ({s | ∃s'∈Q: s → s'} ∩ Sat^R(φ));
   }
   return Q;
   // Sat^R_EU(φ,ψ) = {[s,w]≈ | s,w ⊨ E[φ U ψ]}
}
```

Листинг 2.10. Проверка моделей для формулы **A**[φ **U** ψ] над регионами

```
set<Region> Sat^R_AU(Formula φ)
{
   set<Region> Q, Q';
   Q  = Sat^R(ψ);
   Q' = ∅;
```

```
    while (Q ≠ Q′)
    {
       Q′ = Q;
       Q  = Q ∪ ({s | (∃s′∈Q: s → s′) ∧
                (∀s′: если s → s′, то s′ ∈ Q)} ∩ Sat^R(φ));
    }
    return Q;
    // Sat^R_AU(φ,ψ) = {[s,w]≈ | s,w ⊨ A[φ U ψ]}
}
```

Код для функции $Sat^{\boldsymbol{R}}_{EU}$ идентичен нетаймированному случаю. Для $Sat^{\boldsymbol{R}}_{AU}$ существует небольшое, хотя и существенное, отличие от нетаймированного алгоритма для $\mathbf{A}[\varphi\ \mathbf{U}\ \psi]$. По аналогии с CTL итерация содержала бы следующий оператор:

$$Q := Q \cup (\{s \mid \forall s'\text{: если } s \to s', \text{ то } s' \in Q\} \cap Sat^{\boldsymbol{R}}(\varphi)).$$

Корректность этого оператора, однако, предполагает, что каждое состояние s имеет некоторого потомка над $\to$. Это в действительности верно для CTL, так как в CTL-модели каждое состояние имеет хотя бы одного потомка. В регионных автоматах могут существовать регионы, которые не имеют ни одного исходящего перехода – ни перехода по задержке, ни перехода, который соответствует ребру во временном автомате. Значит, следует придерживаться другого подхода и расширять Q теми регионами, для которых все прямые потомки содержатся в Q и которые имеют хотя бы один переход в Q. Это дает код в листинге 2.10. Проверять, что данный переход ведет именно в Q, необязательно – достаточно проверить, что из региона есть хотя бы один переход (любой). Пример регионного автомата, который содержит регион без каких-либо потомков, изображен на рис. 2.42. Потомков не имеет регион **C**: задержка в нем невозможна в силу наличия инварианта в позиции l, и нет ни одного ребра, которое было бы активно для часовой оценки $x = 1$.

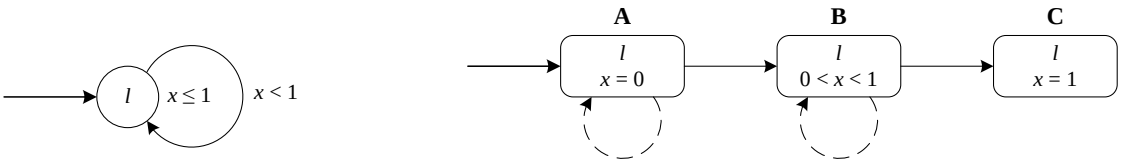


Рис. 2.42. Регионный автомат с регионом без потомков

Пример. Рассмотрим регионный автомат (рис. 2.41) измененного переключателя и предположим, что позиция *on* помечена атомарным предложением q, а позиция *off* – атомарным предложением p. Теперь можно убедиться в том, что невозможно попасть в позицию *on* из позиции *off* менее чем за одну временную единицу:

$M(A), (s_0, w_0) \not\models \mathbf{E}[p\ \mathbf{U}_{<1}\ q]$,

так как не существует пути в регионном автомате, который стартует в **A** (регион, соответствующий $[s_0, w_0]$), и ведет в некоторый регион с **E** по **Q**, где переключатель включен, причем менее чем за одну единицу времени. Формально это можно объяснить следующим образом. Формула $\mathbf{E}[p\ \mathbf{U}_{<1}\ q]$ сокращает $z\ \mathbf{in}\ \mathbf{E}[p\ \mathbf{U}\ (q \wedge z < 1)]$. Снабдим регионный автомат формульными часами z, которые сброшены в **A** и продолжают увеличиваться до бесконечности. Самый ранний момент, когда его можно включить (сделать q выполненным) может быть при прохождении перехода из **C** в **E**. Однако потом значение часов z становится равным 1, и поэтому свойство не выполняется.

Вспомним, что временной автомат называется *незеноновым*, если любой путь, стартующий из любого *достижимого* состояния в этом автомате, является расходящимся или содержит конечное число переходов по ребру. Свойство незеноновости является необходимым условием для рассмотренных выше процедур проверки моделей, как вытекает из следующего результата.

Теорема [42]. Для незенонова временного автомата $(s, w) \in Sat(\varphi)$, если и только если $[s, w] \in Sat^{\boldsymbol{R}}(\varphi)$.

Таким образом, корректность алгоритма проверки моделей гарантирована, только если рассматриваемый временной автомат является незеноновым.

Вспомним также, что временной автомат называется *расходящимся*, если из каждого его достижимого состояния исходит хотя бы один расходящийся путь (существует путь, в котором время растет до бесконечности).

Для того чтобы проверить, выполняется ли условие расходимости для автомата A, надо выяснить, из каждого ли достижимого состояния $M(A)$ время может истечь ровно на одну единицу.

Теорема. Автомат A является расходящимся, если и только если для любого его достижимого состояния $s \in S$ выполняется $M(A), s \models\!\models \mathbf{EF}_{=1}\ \text{true}$.

В контексте регионных автоматов можно дать следующую интуицию для свойств временного автомата:

- Незеноновость временного автомата означает отсутствие в регионном автомате цикла, состоящего из пунктирных ребер. Эти ребра соответствуют переходам временного автомата. В частности, это означает отсутствие пунктирных петель.

- Расходимость временного автомата означает, что из каждого состояния регионного автомата стартует некоторый путь, в котором сплошные ребра (переходы по задержке) встречаются бесконечно часто.
- Если известно, что временной автомат незенонов, то его расходимость означает, что из каждого состояния регионного автомата исходит хотя бы одно ребро (это значит, что отношение $\rightarrow$ является тотальным).

Например, автомат, изображенный на рис. 2.42, не является ни незеноновым, ни расходящимся.

При проверке моделей для незеноновых автоматов неудобств, связанных с расходимостью, можно избежать. Для этого нужно рекурсивно удалить из регионного автомата те достижимые регионы, из которых не исходит ни одного ребра. Свойство незеноновости при этом сохранится, и, кроме того, полученный автомат будет расходящимся (за исключением случая, при котором будут удалены все регионы, включая стартовый, и автомат станет пустым). Для расходящегося автомата код для функции $Sat_{AU}^{\boldsymbol{R}}$ можно упростить за счет отказа от проверки излишних условий.

Размер модели

Алгоритм проверки моделей размечает регионный автомат $\boldsymbol{R}(A, \Psi)$ для временного автомата A и множества часовых ограничений Ψ. Число состояний в $\boldsymbol{R}(A, \Psi)$ равно числу регионов автомата A по отношению к Ψ. Число регионов пропорционально произведению числа позиций в A и числа классов эквивалентности над отношением $\approx$. Оценим теперь число регионов, соответствующих одной позиции.

Пусть C – множество часов автомата A и формульных часов. Для простоты пока будем считать, что часовые ограничения не содержат подформул вида $x - y \sim c$. Тогда число регионов ограничено снизу и сверху следующими выражениями [4]:

$$|C|! \prod_{x \in C} c_x \leq |\text{Regions}| \leq 2^{|C|-1} |C|! \prod_{x \in C} (2c_x + 2).$$

Нижняя и верхняя оценки определяются путем рассмотрения такого представления множества регионов, что существует взаимно однозначное соответствие между представлениями регионов и самими регионами. Это представление дает возможность вывести оценки.

Пусть C – множество часов и $\eta \in V(C)$. Каждый регион r может быть представлен тройкой (P, π, D), где P – индексированное часами семейство интервалов, π – перестановка множества часов и $D \subseteq C$ –

это множество часов, причем тройка (P, π, D) удовлетворяет следующим условиям:

- $P = (P_x)_{x \in C}$ – это семейство подмножеств $\mathbb{R}^+$ вида
  $P_x \in \{ \{0\}, (0, 1), \{1\}, (1, 2), \ldots, (c_x - 1, c_x), \{c_x\}, (c_x, +\infty) \}$
  таких, что $\eta(x) \in P_x$ для всех часов $x \in C$ и всех часовых оценок $\eta \in r$.
- Пусть C_{open} – множество часов $x \in C$ таких, что P_x – открытый интервал, то есть
  $C_{open} = \{x \in C \mid P_x \in \{ (0, 1), (1, 2), \ldots, (c_x - 1, c_x), (c_x, +\infty) \} \}$.
  Тогда $\pi = (x_1, \ldots, x_k)$ – перестановка $C_{open} = \{x_1, \ldots, x_k\}$ такая, что для каждого $\eta \in r$ часы упорядочены в соответствии с их дробными частями (это означает, что из $i \le j$ следует $\{\eta(x_i)\} \le \{\eta(x_j)\}$).
- $D \subseteq C_{open}$ содержит все часы из C_{open} такие, что для всех часовых оценок $\eta \in r$ дробная часть часов $x_i \in D$ совпадает с дробной частью их предшественника x_{i-1} в перестановке π: из $x_i \in D$ следует $\{\eta(x_{i-1})\} = \{\eta(x_i)\}$.

Это является взаимно однозначным соответствием между регионами и тройками (P, π, D).

Указанная верхняя оценка для числа регионов получается за счет комбинаторного наблюдения, состоящего в том, что существуют

- ровно $\prod_{x \in C} (2c_x + 2)$ различных семейств P,
- максимум $|C_{open}|! \le |C|!$ различных перестановок множества C_{open} и
- максимум $2^{|Copen|-1} \le 2^{|C|-1}$ различных вариантов выбора множества $D \subseteq C \setminus \{x_1\}$.

Указанная нижняя оценка получается, когда все часы принимают значения в ограниченном открытом интервале и имеют попарно различные дробные части. В этом случае $D = \varnothing$ и $P_x \in \{ (0, 1), (1, 2), \ldots, (c_x - 1, c_x) \}$.

Из того, что существует ровно

$$\prod_{x \in C} c_x$$

возможностей для P и максимум $|C|!$ перестановок, следует нижняя оценка.

Пусть $c = \max\{c_x \mid x \in C\}$. Верхнюю оценку для удобства можно сделать более грубой:

$$|\text{Regions}| \le 2^{|C|-1} |C|! (2c + 2)^{|C|}.$$

Теперь рассмотрим более общий случай, когда часовые ограничения могут содержать подформулы вида $x - y \sim c$. Определим тройки (P, π, D) так же, как и ранее, но с одним отличием: теперь P_x может иметь вид любого из множеств

$$\{0\}, (0, 1), \{1\}, (1, 2), \ldots, (2c - 1, 2c), \{2c\}, (2c, +\infty).$$

Тогда множество таких троек можно сюръективно отобразить на множество всех регионов, откуда следует верхняя оценка для общего случая:

$$|\text{Regions}| \leq 2^{|C|-1} \, |C|! \, (4c + 2)^{|C|}.$$

Таким образом, проверка моделей для TCTL:

1) линейна относительно длины формулы φ;
2) экспоненциальна относительно числа часов в A и φ;
3) экспоненциальна относительно максимальных констант, с которыми часы сравниваются в A и φ.

Нижняя оценка сложности проверки TCTL-моделей для заданного временного автомата – это *PSPACE*-полнота [39]. Поэтому требуется память размера как минимум полиномиального относительно размера проверяемой системы.

2.6. Сети Петри

Кратко приведем основные понятия теории сетей Петри, которые требуются для формулировки результатов этого раздела. Идеи, на которых базируются сети Петри, описаны в работе [43]. В данном разделе используется терминология из работ [44, 45].

Сеть N – это тройка (S, T, W), где

- S – конечное множество *позиций*;
- T – конечное множество *переходов*;
- S и T дизъюнктны;
- W: $(S \times T) \cup (T \times S) \to \{0, 1, 2, 3, \ldots\}$ – мультимножество *дуг*. Оно определяет дуги и ставит в соответствие каждой дуге неотрицательное целое число – *кратность дуги*; ни одна дуга не может соединять две позиции или два перехода.

Разметкой сети $N = (S, T, W)$ называется отображение M: $S \to \{0, 1, 2, 3, \ldots\}$. Разметка M *активирует* переход t, если для любой позиции s значение $M(s)$ больше либо равно $W(s, t)$. Если переход t активен в разметке M, то он может быть *запущен* и его

запуск приводит к следующей разметке M', которая определена для каждой позиции s следующим образом:

$$M'(s) = M(s) - W(s, t) + W(t, s),$$

то есть из каждой входной позиции перехода t удаляется по маркеру для каждой входной дуги, и в каждую выходную позицию перехода t добавляется по маркеру для каждой выходной дуги. Это обозначается так: $M \xrightarrow{t} M'$. Например, на рис. 2.43 изображены разметки M и M', актуальные до и после выполнения перехода t. Позиции обозначены кругами, переход – вертикальной чертой, а разметка – маркерами.

Рис. 2.43. Пример разметок сетей

Сетью Петри называется пара (N, M_0), где N – сеть, а M_0 – разметка сети N, называемая *начальной*. Последовательность $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} M_n$ – *конечная последовательность запусков*, которая ведет из M_0 в M_n. Она записывается следующим образом: $M_0 \xrightarrow{t_1 \ldots t_n} M_n$. Последовательность $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots$ – *бесконечная последовательность запусков*. *Последовательность запусков максимальна*, если она бесконечна или ведет к разметке, которая не активирует ни один переход. Разметка M сети N *достижима*, если $M_0 \xrightarrow{\sigma} M$ для некоторой конечной последовательности σ. *Граф достижимости сети Петри* – это маркированный граф, вершинами которого являются достижимые разметки. Для двух достижимых разметок M и M' граф достижимости содержит ребро из M в M', помеченное переходом t, если и только если $M \xrightarrow{t} M'$.

Маркированная сеть – это пятерка (S, T, W, Σ, l), где (S, T, W) – сеть, а l – маркирующая функция, которая сопоставляет каждому переходу букву определенного алфавита Σ. Эта функция не обязана быть инъективной. Граф достижимости маркированной сети определяется так же, как и в обычных сетях. Единственное отличие состоит в том, что если $M \xrightarrow{t} M'$, то соответствующее ребро из M в M' помечено $l(t)$.

Иногда «обычные» сети Петри называют *немаркированными*. Немаркированные сети Петри могут рассматриваться как частный

случай маркированных сетей Петри, в которых маркирующая функция инъективна. В маркированных сетях Петри функцию l можно расширить на последовательности переходов так, что каждой такой последовательности эта функция будет сопоставлять строку в алфавите Σ по правилу $l(\sigma_1\sigma_2) = l(\sigma_1)l(\sigma_2)$.

Для сети Петри (N, M_0) и разметки M^f сети N (называемой *финальной разметкой*) определим *язык* этой сети по отношению к M^f как

$$L(N, M_0, M^f) = \{\sigma \mid M_0 \xrightarrow{\sigma} M^f\}$$

и *множество путей* сети как

$$T(N, M_0) = \{\sigma \mid M_0 \xrightarrow{\sigma} M \text{ для некоторой разметки } M\}$$

(иногда используются термины «язык» и «терминальный язык» вместо «множества путей» и «языка»).

Для маркированной сети Петри (N, M_0), где $N = (S, T, W, \Sigma, l)$ и разметки M^f сети N *язык* этой сети по отношению к M^f определяется как

$$L(N, M_0, M^f) = \{l(\sigma) \mid M_0 \xrightarrow{\sigma} M^f\}$$

и *множество путей* сети определяется как

$$T(N, M_0) = \{l(\sigma) \mid M_0 \xrightarrow{\sigma} M \text{ для некоторой разметки } M\}.$$

Временная сеть Петри – это сеть Петри, к переходам которой присоединены временные интервалы. Формально, временная сеть Петри – это набор (S, T, W, M_0, Is), где S – множество позиций, T – множество переходов, W – мультимножество дуг, M_0 – начальная разметка, а функция $Is: T \to \mathbb{Q}^+ \times (\mathbb{Q}^+ \cup \{\infty\})$ сопоставляет каждому переходу t интервал, называемый *интервалом запуска* t. Для любого перехода t левый элемент кортежа $Is(t)$ не превосходит правый.

Для разметки M обозначим через $En(M)$ множество всех переходов, активных в M. Характеризация состояний временной сети Петри сопоставляет каждому переходу t модели *часы* для измерения времени, прошедшего с последнего момента, когда t стал активен. *Состоянием* временной сети Петри называется пара (M, v), где M – разметка, а v – функция оценки часов $v: En(M) \to \mathbb{R}^+$. Когда переход t становится активным, его часы инициализируются в ноль. Значение этих часов синхронно увеличивается со временем до тех пор, пока t не будет запущен или не будет блокирован путем запуска другого перехода. Переход t может быть запущен, если значение его часов принадлежит статическому интервалу запуска $Is(t) = [t_{min}(t), t_{max}(t)]$. Он должен быть запущен немедленно, без дополнительной задержки, когда часы достигают $t_{max}(t)$.

Семантика временных сетей Петри может быть определена с использованием состояний. Пусть $d \in \mathbb{R}^+$ – неотрицательное действительное число, $t \in T$ – переход, s и s' – два состояния временной сети Петри. Будем писать $s \xrightarrow{d} s'$, если и только если состояние s' достижимо из состояния s спустя d единиц времени. Также будем писать $s \xrightarrow{t} s'$, если и только если состояние s' достижимо из состояния s немедленно путем запуска перехода t.

Система переходов временной сети Петри определена как структура $(\boldsymbol{S}, \longrightarrow, s_0)$, где s_0 – *начальное состояние* (пара (M_0, v), где $v \equiv 0$) и $\boldsymbol{S} = \{s \mid s_0 \xrightarrow{*} s\}$ – это множество *достижимых* состояний (здесь $\xrightarrow{*}$ – рефлексивное и транзитивное замыкание $\longrightarrow$). Множество достижимых состояний временной сети Петри, вообще говоря, бесконечно и не подходит для перечисления. В этом случае требуются абстракции состояний.

Основная задача для заданной сети Петри и формулы темпоральной логики формулируется следующим образом: выполняет ли сеть данную формулу? Это задача *проверки моделей*. Линейные темпоральные логики для сетей Петри обычно интерпретируются на множестве максимальных последовательностей запусков. Ветвящиеся темпоральные логики интерпретируются на графе достижимости.

Результаты относительно ветвящихся темпоральных логик в основном отрицательные [44]. Задача проверки моделей для сетей Петри для одной из самых слабых ветвящихся темпоральных логик неразрешима. В этой логике базовые предикаты (атомарные предложения) имеют форму $ge(s, c)$, где s – позиция сети, а c – неотрицательная целая константа. Предикат $ge(s, c)$ читается как «число маркеров в разметке в позиции s больше либо равно c». Он выполняется в разметке M, когда $M(s) \geq c$. Операторами этой логики являются стандартные булевы связки, **EX** («Existential neXt») и **EF** («Existential Future»). Достижимая разметка удовлетворяет свойству **EX** φ, если она активирует переход t такой, что разметка, достигнутая путем запуска t, удовлетворяет φ. Разметка удовлетворяет **EF** φ, если она активирует последовательность запусков σ такую, что разметка, достигнутая путем исполнения σ, удовлетворяет φ.

Такие ветвящиеся темпоральные логики, как CTL и CTL*, более выразительны, чем вышеописанная логика, поэтому результаты о неразрешимости переносятся и на них.

Линейные темпоральные логики для сетей Петри исследованы более глубоко. Для того чтобы составить объединяющую структуру и рассмотреть в ней результаты, добавим еще два базовых предиката к

предикатам $ge(s, c)$ и построим темпоральную логику на их основе. Предикаты теперь интерпретируются на разметках максимальной последовательности запусков. Будем говорить, что последовательность запусков удовлетворяет формуле темпоральной логики, если ее начальная разметка удовлетворяет этой формуле. И, наконец, сеть Петри удовлетворяет формуле, если хотя бы одна максимальная последовательность запусков удовлетворяет ей (или, что равносильно, если не каждая максимальная последовательность запусков удовлетворяет ее отрицанию). Новые предикаты следующие:

- *first*(t), где t – переход сети. Данный предикат выполняется в разметке M, если t – это переход, непосредственно следующий за M в последовательности запусков.
- *en*(t), где t – переход сети. Он выполняется в разметке M, если M активирует t[9].

Задача проверки моделей для логики с этими тремя базовыми предикатами и темпоральным оператором **F** (разметка последовательности запусков удовлетворяет **F** φ, если какая-то более поздняя разметка удовлетворяет φ) неразрешима.

Задача проверки моделей, однако, разрешима для некоторых фрагментов:

- Фрагмент, в котором отрицания применяются только к предикатам. Этот фрагмент содержит формулу **F** *first*(t), которая соответствует тому, что t когда-нибудь наступает, но не содержит формулы **GF** *first*(t), где G = ¬**F** ¬, которая определяет то, что t встречается бесконечно часто. Задача проверки моделей для этого фрагмента может быть полиномиально сведена к задаче достижимости разметки сети Петри.
- Фрагмент, в котором разрешен только составной оператор **GF**, и отрицания применяются только к предикатам. Этот фрагмент содержит формулу **GF** *first*(t), но не содержит, например, формулу **GF** *first*(t) → **GF** *first*(t') (после замены импликации по ее определению перед оператором появляется отрицание). Задача проверки моделей для этого фрагмента сводится к экспоненциальному числу экземпляров задачи достижимости. Для формул, имеющих форму **GF** φ, где φ – это булева комбинация базовых предикатов, существует лучший результат: задача проверки моделей может быть полиномиально сведена к задаче достижимости.

---

[9] Предикат *en*(t) может быть определен как конъюнкция предикатов $ge(s, W(s, t))$ для всех входных позиций. Он включен в базовые предикаты для удобства.

Глава 3. Обзор верификаторов

3.1. *SPIN*

SPIN [46] – это верификатор, который поддерживает проверку моделей и разработку систем асинхронных процессов. В верификаторе *SPIN* процессы могут взаимодействовать между собой следующими способами:

- с помощью примитивов *rendezvous* [46];
- с помощью асинхронных передач сообщений через буферизованные каналы;
- через общие переменные;
- комбинированным способом.

Кроме собственно верификации, инструментальное средство поддерживает еще и режимы интерактивной и случайной эмуляции. При этом запускается интерпретатор модели. Недетерминированные переходы в режиме интерактивной эмуляции выбираются пользователем, а в режиме случайной эмуляции выбираются случайно.

SPIN допускает симуляцию спецификации, написанной на языке *PROMELA* (*PROtocol Meta-Language*) и верификацию некоторых типов свойств, например, проверку моделей LTL-формул (без оператора следования **X**) и верификацию свойств состояний, недостижимого кода и т. д.

Общая структура инструмента *SPIN* приведено на рис. 3.1. Алгоритмы, которые использует *SPIN* при проверке LTL-формул на моделях, базируются на конвертации LTL-формулы в *автомат Бюхи*.

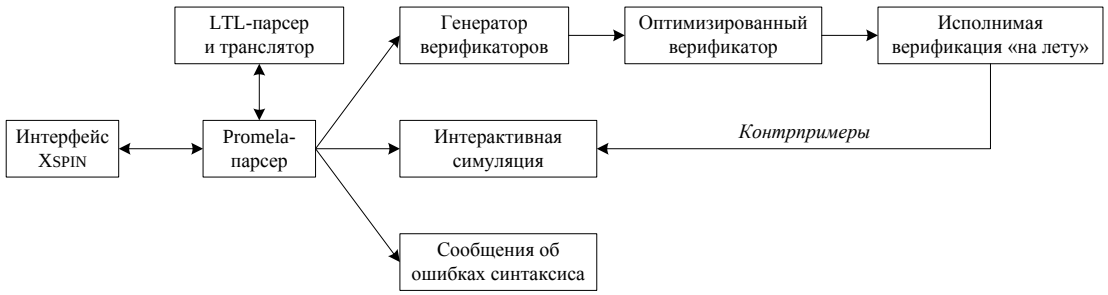


Рис. 3.1. Структура симуляции и верификации с использованием *SPIN*

Типичный способ работы с инструментальным средством *SPIN* – начать со спецификации высокоуровневой модели алгоритма. При этом обычно используют графическую оболочку *Xspin*. После исправления синтаксических ошибок выполняют интерактивную эмуляцию до тех пор, пока не наступает уверенность, что модель

ведет себя, как планировалось. После этого на третьем шаге *SPIN* используется для того, чтобы сгенерировать оптимизированную на лету программу проверки спецификации высокого уровня. Полученная программа компилируется с возможностью выбора во время компиляции алгоритмов проверки, которые будут использоваться. Если обнаруживаются какие-нибудь контрпримеры к требованиям корректности, то они могут быть возвращены в интерактивный эмулятор и рассмотрены подробно для того, чтобы можно было установить и устранить их причину.

Синтаксис языка *PROMELA* напоминает синтаксис языка *C*. Модель на языке *PROMELA* состоит из следующих элементов:

- объявления типов данных;
- объявления каналов передачи переменных;
- объявления переменных;
- объявления и определения процессов;
- процесса `init`.

Понятие «процесс» в данном случае отдаленно можно рассматривать как процедуру, выполняемую в отдельном потоке. Приведем пример объявления и определения процесса (листинг 3.1):

Листинг 3.1. Объявление и определение процесса

```
proctype proc(int a; int b)
{
  byte b; /* локальная переменная */
  /* тело процесса */
}
```

Процессы могут иметь параметры и локальные переменные. Процесс может быть запущен в нескольких экземплярах, если для него используется модификатор `active`. Запускаются процессы с помощью модификатора `run`.

*Я*зык *PROMELA* имеет пять базовых типов данных:

- *bit*;
- *bool*;
- *byte*;
- *short*;
- *int*.

Тело процесса состоит из последовательности операторов. Операторы могут быть *активными* либо *блокированными*. *Активный* оператор – это оператор, который может быть выполнен немедленно.

Блокированный оператор – оператор, который не может быть выполнен в данный момент. Такой оператор блокирует выполнение процесса до тех пор, пока он не станет активным. Например, оператор

```
x < 7
```

может быть активен тогда и только тогда, когда *x* меньше семи. В противном случае он останавливает выполнение процесса до тех пор, пока условие не выполнится. Некоторые операторы (например, оператор присваивания) активны всегда.

Язык *PROMELA* содержит также операторы ветвления и цикла, синтаксис которых (листинг 3.2) основан на охранных командах Дейкстры [47].

Листинг 3.2. Операторы ветвления и цикла

```
if                          do
  :: guard1 -> S1             :: guard1 -> S1
  :: guard2 -> S2             :: guard2 -> S2
  ...                         ...
  :: else -> Sk               :: else -> Sk
fi                          od
```

Поясним работу оператора `if`. Если условие $guard_i$ истинно, то выполняется действие S_i. Если одновременно выполняются несколько условий, то происходит недетерминированный выбор одного из них. Если все условия ложны, то выполняется действие S_k, соответствующее `else`. Конструкция `else` может отсутствовать. Тогда в случае, если все условия ложны, выполнение процесса блокируется до тех пор, пока хотя бы одно из них не начнет выполняться.

Оператор do имеет такую же форму, как оператор if, с единственной разницей, что `if` заменяется на `do`, а `fi` на `od`. Отметим, что оператор do – это в принципе всегда бесконечный цикл, так как если все охранные команды блокированы, то он ждет до тех пор, пока одна из них не станет активной. Выполнение оператора do, таким образом, может быть прекращено с помощью оператора `goto` или `break`.

Язык *PROMELA* позволяет определять каналы передачи данных. Процессы могут быть соединены через однонаправленный канал произвольной (но конечной) емкости. Каналы емкости 0 также

разрешены (примитив *rendezvous* в терминах *SPIN*). Каналы специфицируются так же, как и обычные переменные. Например,

```
chan c = [5] of byte
```

определяет канал, называемый `c`, емкостью 5 байт. Канал может быть рассмотрен как буфер в виде очереди. Посылка сообщения обозначается восклицательным знаком, прием сообщения – вопросительным знаком. Пусть `c` имеет положительную емкость. Тогда команда `c!2` активна, если `c` не полностью занят. Ее исполнение помещает элемент 2 в конец буфера. Команда `c?x` активна, если `c` не пуст. Она удаляет голову буфера `c` и присваивает ее значение переменной `x`. Если `c` не буферизована, то есть имеет размер 0, то `c!` (и `c?`) активна, если имеется команда `c?` (и `c!`, соответственно), которая может быть выполнена в то же время (и типы параметров соответствуют друг другу). Эта возможность полезна для моделирования синхронных коммуникаций между процессами.

Протокол выбора лидера

Рассмотрим использование верификатора на примере протокола выбора лидера [1, 48]. Пусть имеется N процессов ($N \geq 2$) в кольцевой топологии, соединенных неограниченными каналами. Процесс может посылать сообщения только по часовой стрелке. Первоначально каждый процесс имеет уникальный идентификатор *ident*, который ниже предполагается натуральным числом. Цель протокола выбора лидера в получении уверенности, что ровно один процесс становится лидером. Также задано дополнительное условие: процесс с наивысшим идентификатором должен быть выбран лидером. Объяснение заключается в том, что идентификатор процесса отражает его возможности, и желательно, чтобы процесс с максимальными возможностями выиграл выборы. В протоколе каждый процесс в кольце выполняет следующую задачу (листинг 3.3):

Листинг 3.3. Протокол выбора лидера

```
active:
d = ident;
while (true)
{
  send(d);
  receive(e);
  if (e == d) stop;  // Процесс d является лидером
  send(e);
  receive(f);
  if e >= max(d, f)
  {
```

```
      d = e;
    }
    else break;
  }

  relay:
  while (true)
  {
    receive(d);
    send(d)
  }
```

Из рассмотрения протокола следует, что первоначально каждый процесс активен (*active*). В течение того времени, пока он активен, процесс отвечает за определенное процессное число (сохраненное в переменной *d*). Это значение может изменяться в ходе работы протокола. Когда процесс определяет, что он не несет идентификатор текущего лидера, он начинает уступать. Если процесс уступает (*relay*), он передает сообщения слева направо прозрачным способом – без просмотра или модификации их содержимого.

Каждый активный процесс посылает свою переменную *d* своему соседу по часовой стрелке и потом ждет до тех пор, пока не получит значение (*e*) от своего ближайшего активного соседа против часовой стрелки. Если процесс получает свое собственное значение *d*, он заключает, что остался единственный активный процесс и что это *d* в действительности является идентификатором нового лидера, и прекращает работу. (Отметим, что процесс сам по себе не обязан становиться новым лидером!) В случае если принято другое значение ($e \neq d$), процесс ждет второго сообщения (*f*), содержащего значение *d*, которое несет второй ближайший активный сосед против часовой стрелки. Если значение ближайшего активного соседа против часовой стрелки наибольшее среди *e*, *f* и *d*, то процесс обновляет свое локальное значение ($d := e$). В противном случае он начинает уступать. Таким образом, из любого множества активных соседей один будет уступать в каждом раунде.

Для того чтобы проиллюстрировать, как работает протокол, рассмотрим пример его запуска для четырех процессов с идентификаторами 1 — 4, как показано на рис. 3.2. Здесь процессы обозначаются вершинами, а коммуникационные каналы – стрелками. Содержимое канала указано в метке возле стрелки, точка с запятой используется для разделения различных сообщений. Каждая вершина помечена тройкой, содержащей значения локальных переменных *d*, *e* и *f*. Черная вершина обозначает, что процесс уступает, в противном случае он активен.

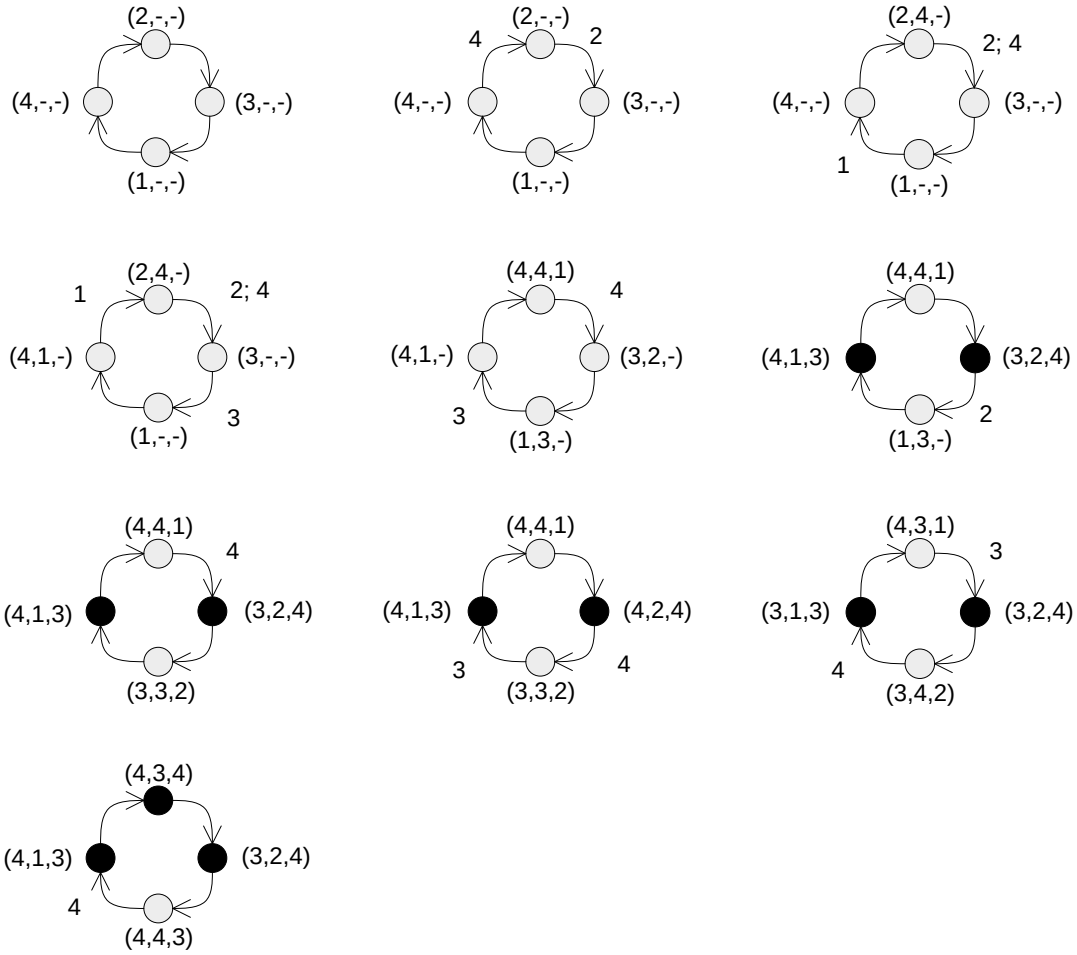


Рис. 3.2. Пример запуска протокола выбора лидера

Моделирование протокола выбора лидера на языке *PROMELA*

Спецификация *PROMELA* состоит из трех частей (листинг 3.4): определения глобальных переменных, описания поведения процесса в кольце и инициализационной секции, в которой процессы стартуют. Поведенческая часть процесса `process` в действительности получается прямым переводом описания протокола, приведенного выше. В протокол добавлены два комментария, выводящихся на экран для упрощения симуляции. Процесс соединен со своим соседом против часовой стрелки через канал `in`, и через канал `out` – со своим соседом по часовой стрелке.

Листинг 3.4. Протокол выбора лидера на языке *PROMELA*

```
proctype process (chan in, out; byte ident)
{ byte d, e, f;

  printf("%d\n", ident);
activ:
  d = ident;
  do :: true -> out!d;
                in?e;
```

```
                    if :: (e == d) ->
                          printf("%d является лидером\n", d);
                          goto stop
                       :: else -> skip
                    fi;
                    out!e;
                    in?f;
                    if :: (e >= d) && (e >= f) -> d = e
                       :: else -> goto relay
                    fi
  od;

relay:
end:
  do :: in?d -> out!d
  od;
stop:
  skip
}
```

Процесс может быть в одном из трех состояний: `activ` («active» не используется, так как это зарезервированное служебное слово в *PROMELA*), `relay` и `stop`. Процесс может достичь состояния `stop`, только если он распознал нового лидера. Процесс, который не распознал лидера, завершится в состоянии `relay`. Во избежание генерации неверного завершающего состояния (состояния, которое не является концом тела программы и которое не помечено меткой «`end:`») пометим состояние `relay` как завершающее. Если пытаться верифицировать вышеуказанную спецификацию *PROMELA* без этой метки `end:`, то *SPIN* нашел бы неверное завершающее состояние и выдал ошибку.

Отметим, что, например, оператор `in?e` выполним, только если канал `in` не пуст. Если канал пуст, то оператор блокирован, и выполнение процесса приостановлено до тех пор, пока он не станет выполнимым. Следовательно, `in?e` должно читаться следующим образом: ждать, пока сообщение не попадет в канал `in`, и затем поместить его значение в переменную `e`.

Спецификация *PROMELA* начинается с определения глобальных переменных (листинг 3.5):

Листинг 3.5. Глобальные переменные спецификации на языке *PROMELA*

```
#define N 5   /* число процессов */
#define I 3   /* процесс с наименьшим идентификатором */
#define L 10  /* размер буфера (>= 2*N) */

/* N каналов емкостью L каждый */
chan q[N] = [L] of { byte };
```

Константа `I` используется только с целью инициализации. Она определяет процесс, который будет обладать наименьшим значением `ident`. Емкость `L` соединяющих каналов должна быть как минимум $2N$, так как каждый процесс посылает максимум два сообщения в каждом раунде выборов. Так как сообщения в протоколе выбора лидера несут только один параметр (идентификатор процесса), то каналы с `q[0]` по `q[N-1]` могут содержать элементы типа `byte`.

Для того чтобы создать топологию кольца и запустить экземпляры процессов в кольце, используется конструкция `init`. Эта часть спецификации *PROMELA* имеет следующий вид (листинг 3.6):

Листинг 3.6. Запуск экземпляров процессов

```
init {
  byte i;
  atomic {
        i = 1;
        do :: i <= N -> run process (q[i-1],
                        q[i%N], (N+I-i)%N+1);
                        i = i+1
           :: i > N -> break
        od
   }
}
```

Здесь `%N` обозначает остаток по модулю *N*. Процесс запускается с определенными фактическими параметрами оператором `run process()`. Первый параметр процесса – это входящий канал, второй параметр – выходящий канал, а третий параметр – его идентификатор. Существуют, конечно, различные механизмы присвоения идентификаторов процессам, здесь просто выбран один из самых простых. Предполагается, однако, что каждый процесс получает уникальный идентификатор. Это необходимо для корректности алгоритма. С целью обеспечить всем процессам возможность стартовать одновременно, воспользуемся конструкцией `atomic`. Она защищает запуски процессов от того, чтобы исполнять тело процесса, пока другие процессы еще не созданы.

Отметим, что здесь фиксировано одно индивидуальное присвоение идентификаторов процессам. Результаты, которые получены симуляцией и верификацией, верны только для этого присвоения. С целью увеличения надежности протокола должны быть проверены другие (фактически, все возможные) присвоения. Другая возможность состоит в том, чтобы написать фрагмент на языке *PROMELA*, который присваивает процессам идентификаторы случайно.

PROMELA-спецификация, полученная таким способом, может быть теперь верифицирована. При этом не будет недостижимого кода или блокировок. Для того чтобы улучшить понимание того, как работает протокол, может быть выполнена (случайная, управляемая или интерактивная) симуляция.

Проверка моделей для спецификации на языке *PROMELA*

Самое важное свойство, которым должен обладать протокол выбора лидера, состоит в том, что он не должен позволять более чем одному процессу становиться лидером. Это выражается в LTL как

$$\mathbf{G}\,\neg[\#leaders \geq 2],$$

или, эквивалентно,

$$\mathbf{G}[\#leaders \leq 1].$$

Это свойство проверяется инструментом *SPIN* следующим образом. Определяется формула `[]p`, где `[]` соответствует **G**, а `p` определяется так:

```
#define p  (nr_leaders <= 1)
```

Здесь `nr_leaders` – вспомогательная глобальная переменная в спецификации *PROMELA*, которая отслеживает число лидеров в системе. Спецификация *PROMELA* на данном этапе должна быть расширена следующим образом (листинг 3.7):

Листинг 3.7. Новая спецификация *PROMELA*

```
byte nr_leaders = 0;

proctype process (chan in, out; byte ident)
{ .....как раньше.....

activ:
  d = ident;
  do :: true -> out!d;
                in?e;
                if :: (e == d) ->
                      printf("%d является лидером\n", d);
                      nr_leaders = nr_leaders + 1;
                      goto stop
                   :: else -> skip
                fi;

   .....как раньше.....
}
```

Свойство `[] p` автоматически конвертируется средством *SPIN* в *размеченный автомат Бюхи*. Автомат, полученный верификатором *SPIN*, имеет следующий вид (листинг 3.8):

Листинг 3.8. Автомат Бюхи

```
/*
* Введенная формула: [] p
* Ниже условие claim соответствует
* отрицанию исходной формулы !([] p)
* (формализующему нарушение свойства)
*/

never { /* !([] p) */
T0_init:
        if
        :: (1) -> goto T0_init
        :: (! ((p))) -> goto accept_all
        fi;
accept_all:
        skip
}
```

Напомним, что автомат Бюхи соответствует отрицанию проверяемой формулы – $A_{\neg\varphi}$, где φ соответствует **G**[*#leaders* ≤ 1]. Если это свойство сохранено в файле `property1`, то *SPIN* автоматически проверит его, если добавить в конец (или начало) спецификации *PROMELA* инструкцию:

```
#include  "property1"
```

Результат верификации свойства положительный (листинг 3.9).

Листинг 3.9. Результат верификации протокола выбора лидера

```
Full state space search for:
never-claim +
assertion violations + (if within scope of claim)
cycle checks - (disabled by -DSAFETY)
invalid endstates - (disabled by never-claim)

State-vector 140 byte, depth reached 155, errors: 0
   16585 states, stored
   44589 states, matched
   61174 transitions (= stored+matched)
      18 atomic steps
hash conflicts: 1766 (resolved)
(max size 2^19 states)

Stats on memory usage (in Megabytes):
2.388 equivalent memory usage for states
                        (stored*(State-vector + overhead))
2.046 actual memory usage for states (compression: 85.65%)
State-vector as stored = 119 byte + 4 byte overhead
2.097 memory used for hash-table (-w19)
0.200 memory used for DFS stack (-m10000)
4.448 total actual memory usage
```

О положительном результате свидетельствует текст `errors: 0`. Даже для этого небольшого протокола с 5 процессами (плюс соединяющие их каналы) множество состояний уже довольно велико. Поэтому *SPIN* предоставляет определенные механизмы для того, чтобы сделать множество состояний меньше. Это осуществляется или за счет хранения их компактным способом (используя редукцию частичных порядков), или путем применения хеширования (которое избегает просмотра всего множества состояний), или комбинацией этих способов.

Эффект применения редукции частичных порядков к рассмотренному примеру состоит в том, что сохраняются только 3189 состояний и 5014 переходов вместо 16585 состояний и 61172 переходов, по сравнению со случаем, когда редукция частичных порядков не применялась.

Таким же путем могут быть верифицированы следующие свойства: лидер всегда в конечном счете будет выбран; выбранным лидером будет процесс с наибольшим идентификатором; если процесс выбран лидером, то он останется лидером навсегда.

Покажем, что происходит, когда проверяется неверное свойство: проверим **G**[*#leaders* = 1]. Оно нарушается в состоянии, когда все экземпляры процессов создаются, так как среди них нет лидера. Если запустить верификатор *SPIN*, то получим ошибку, и сгенерируется контрпример, который сохранится в файл. Для анализа ошибки может быть запущена управляемая симуляция, базирующаяся на построенном контрпримере. Кратчайшая трасса, которая ведет к ошибке, может быть сгенерирована по запросу.

Верификация с использованием утверждений

Альтернативным, а часто и более эффективным путем верификации, являются так называемые *утверждения* (assertions). Утверждение – это конструкция, аргументом которой является булево выражение. Его исполнение не имеет побочных эффектов. Если спецификация на языке *PROMELA* содержит утверждение, то *SPIN* проверяет, есть ли вычисление, при котором утверждение нарушается. В этом случае отображается ошибка.

Рассмотрим пример использования утверждений. Предположим, что требуется проверить, действительно ли число лидеров всегда не превышает одного. Это может быть проверено путем добавления утверждения

```
assert(nr_leaders <= 1)
```

непосредственно в код спецификации *PROMELA* в момент, когда процесс определяет, что он выиграл выборы и увеличивает

переменную `nr_leaders`. Так получается тело процесса (листинг 3.10):

Листинг 3.10. Спецификация, использующая утверждения

```
byte nr_leaders = 0;

proctype process (chan in, out; byte ident)
{ .....как раньше.....

activ:
  d = ident;
  do :: true -> out!d;
                in?e;
                if :: (e == d) ->
                      printf("%d является лидером\n", d);
                      nr_leaders = nr_leaders + 1;
                      assert(nr_leaders <= 1);
                      goto stop
                   :: else -> skip
                fi;

   .....как раньше.....
}
```

Верификация этой спецификации *PROMELA* не возвращает ошибок и использует такое же число состояний и переходов, как и верификация соответствующей LTL-формулы **G**[*#leaders* ≤ 1].

3.2. *SMV*

Инструментальное средство *SMV* (*Symbolic Model Verifier*) [49] поддерживает верификацию кооперирующих процессов, которые взаимодействуют через разделенные переменные. Это средство было первоначально разработано для автоматической верификации синхронных аппаратных схем. Оно также было очень полезно для верификации коммуникационных протоколов. Также оно применялось к большим программным системам, например, в авиации [50]. Процессы в *SMV* могут исполняться синхронным или асинхронным способом. Этот верификатор поддерживает проверку моделей для CTL-формул. Покажем на примерах, как системы могут быть специфицированы и верифицированы с использованием *SMV*. Описание *SMV* и примеры излагаются в соответствии с курсом [1].

Спецификация на входном языке *SMV* состоит из описаний процессов, описаний (локальных и глобальных) переменных, описаний формул и спецификаций формул, которые должны быть верифицированы. Главный модуль называется `main`, как и в языке C. Глобальная структура спецификации *SMV* приведена в листинге 3.11.

Листинг 3.11. Структура спецификации *SMV*

```
MODULE main
VAR определения переменных
ASSIGN присваивания глобальным переменным

/* необязательно */
DEFINITION определение верифицируемого свойства

SPEC верифицируемая CTL-спецификация

MODULE /* подмодуль 1 */

MODULE /* подмодуль 2 */

.............
```

Основные компоненты спецификации систем при использовании *SMV* следующие:

- *Типы данных*. Единственные типы данных, предоставляемые *SMV* – это ограниченные целочисленные поддиапазоны и символические перечислимые типы.
- *Описания и инициализации процессов*. Процесс, называемый `P`, определяется следующим образом:

  ```
  MODULE P (формальные параметры)
  VAR локальные переменные
  ASSIGN стартовые присвоения переменным
  ASSIGN присвоения переменным, выполняющиеся на переходах
  ```

  Эта конструкция описывает модуль, также называемый `P`, для которого двумя способами может быть определен экземпляр:

  1) экземпляр процесса `Pasync` будет исполняться в асинхронном режиме:
     `VAR Pasync: process P(параметры)`
  2) экземпляр процесса `Psync` будет исполняться в синхронном режиме:
     `VAR Psync: P(параметры)`

  Разница между асинхронным и синхронным режимом обсуждается ниже.

- *Присваивания переменных*. В *SMV* процесс рассматривается как конечный автомат и определяется путем перечисления для каждой (локальной и глобальной) переменной начального значения (значений в начальном состоянии) и значения, которое будет присвоено переменной в следующем состоянии. Последнее

значение обычно зависит от текущих значений переменных. Например, `next(x) := x+y+2` присваивает переменной `x` значение `x+y+2` в следующем состоянии. Для переменной `x` присваивание `init(x) := 3` означает, что `x` первоначально принимает значение равное 3. Присваивание `next(x) := 0` назначает переменной `x` значение 0 в следующем состоянии. Присваивания могут быть недетерминированными. Например, оператор `next(x) := {0, 1}` означает, что следующее значение `x` равно 0 или 1. Присваивания также могут быть условными. Например, присваивание

```
next(x) := case b = 0: 2;
                b = 1: {7, 12}
           esac;
```

назначает переменной `x` значение 2, если `b` равно 0 и (недетерминированно) значение 7 или 12, если `b` равно 1. Если `x` – переменная в экземпляре процесса `Q`, то пишется `Q.x`.

Присваивания глобальным переменным разрешены, только если эти переменные встречаются в качестве параметров в экземпляре процесса.

- *Синхронный и асинхронный режимы*. В синхронном режиме все присваивания переменным выполняются за один неделимый шаг. Интуитивно это означает, что существуют одни глобальные часы и в течение каждого их такта каждый модуль выполняет шаг. В любой заданный момент времени процесс либо выполняется, либо не выполняется. Присваивание значению `next` переменной происходит только в том случае, когда процесс выполняется. Если процесс не выполняется, то значение переменной на следующем шаге остается неизмененным.

  В асинхронном режиме новое значение присваивается только переменным «активного» процесса. Следовательно, в течение каждого тика глобальных часов один процесс недетерминированно выбирается для выполнения, и один шаг этого процесса выполняется (в то время как остальные, невыбранные процессы сохраняют свое состояние). Синхронная композиция полезна, например, для моделирования синхронных аппаратных схем, тогда как асинхронная композиция полезна для моделирования коммуникационных протоколов или асинхронных аппаратных систем.

- *Спецификация CTL-формул*. Для спецификации CTL-формул *SMV* использует символы `&` – для конъюнкции, `|` – для дизъюнкции, `->`

– для импликации и ! – для отрицания. Программа проверки моделей *SMV* проверяет, что все возможные начальные состояния удовлетворяют спецификации.

Алгоритм взаимного исключения

Алгоритм взаимного исключения Петерсона и Фишера [1] используется двумя процессами, которые взаимодействуют через разделяемые переменные. Требуется лишить процессы возможности быть одновременно в критической секции. Каждый процесс i ($i = 1, 2$) имеет локальные переменные t_i и y_i, которые могут быть прочитаны другим процессом, но могут быть записаны только процессом i. Эти переменные принимают одно из значений $\{\perp, \text{false}, \text{true}\}$. Оператор $\neg$ не определен для значения $\perp$, а значения false и true он инвертирует. Это означает, что в выражении $\neg y_2 = y_1$ предполагается, что $y_1 \neq \perp$. Начальные значения переменных таковы: $y_1, y_2, t_1, t_2 := \perp, \perp, \perp, \perp$. Процесс 1 описан в листинге 3.12.

Листинг 3.12. Процесс 1

```
start1:
t1 := if y2 = false then false else true fi
y1 := t1
if y2 ≠ ⊥ then t1 := y2 fi
y1 := t1
loop while y1 = y2
critical section 1; y1, t1 := ⊥, ⊥
goto start1
```

Процесс 2 описан в листинге 3.13.

Листинг 3.13. Процесс 2

```
start2:
t2 := if y1 = true then false else true fi
y2 := t2
if y1 ≠ ⊥ then t2 := ¬y2 fi
y2 := t2
loop while (¬y2) = y1
critical section 2; y2, t2 := ⊥, ⊥
goto start2
```

Базовый метод алгоритма состоит в том, что когда оба процесса соревнуются за вход в критическую секцию, гарантированно выполняется $y_1 \neq \perp$ и $y_2 \neq \perp$. Следовательно, условия для входа в

критическую секцию никогда не будут выполнены одновременно. Отметим также, что эти два процесса не симметричны.

Моделирование алгоритма взаимного исключения в *SMV*

Перевод описанного алгоритма в спецификацию *SMV* выполняется непосредственно. Два экземпляра процесса создаются следующим образом:

```
VAR prc1 : process P(t1,t2,y1,y2);
    prc2 : process Q(t1,t2,y1,y2);
```

Поведенческая часть первого процесса из предыдущей секции приведена ниже. Присвоим каждому оператору в алгоритме метку, начиная с `l1`. Эти метки используются для того, чтобы установить следующий оператор после выполнения соответствующего оператора. Они являются аналогом счетчика инструкций. Код *SMV* для другого процесса аналогичен и здесь не приводится. Комментарии начинаются со знаков `--` и заканчиваются возвратом каретки (листинг 3.14).

Листинг 3.14. Модель процесса *P*

```
MODULE P(t1,t2,y1,y2)
VAR label : {l1,l2,l3,l4,l5,l6,l7};
            -- локальная переменная label
ASSIGN init(label) := l1;
            -- стартовое присвоение
ASSIGN      -- присвоение для переходов
   next(label) :=
       case
         label = l1              : l2;
         label = l2              : l3;
         label = l3              : l4;
         label = l4              : l5;
         label = l5 & y1 = y2    : l5;  -- loop
         label = l5 & !(y1 = y2) : l6;
         label = l6              : l7;
         label = l7              : l1;  -- goto start
       esac;
   next(t1) :=
       case
         label = l1 & y2 = false     : false;
         label = l1 & !(y2 = false)  : true;
         label = l3 & y2 = bottom    : t1;
         label = l3 & !(y2 = bottom) : y2;
         label = l6                  : bottom;
         1                           : t1;
              -- в противном случае
              -- не изменять t1
       esac;
   next(y1) :=
       case
```

```
        label = l2 | label = l4 : t1;
        label = l6              : bottom;
        1                       : y1;
             -- в противном случае
             -- не изменять y1
    esac;
```

Отметим, что метка `prc1.l6` соответствует критической секции процесса 1 и, симметрично, метка `prc2.m6` – критической секции процесса 2.

Проверка моделей для спецификации *SMV*

Укажем два главных свойства, которыми должен обладать алгоритм взаимного исключения.

Первое из них состоит в том, что в одно и то же время в критической секции должен быть максимум один процесс. Это выражается в CTL следующим образом:

$$\mathbf{AG}\ \neg(prc1.label = l6 \wedge prc2.label = m6).$$

В *SMV* это свойство определяется так:

```
DEFINE MUTEX := AG !(prc1.label = l6 & prc2.label = m6)
```

Здесь `MUTEX` – макрос. Результат верификации с использованием `SPEC MUTEX` положительный:

```
specification MUTEX is true

resources used:
user time: 1.68333 s, system time: 0.533333 s
BDD nodes allocated: 12093
Bytes allocated: 1048576
BDD nodes representing transition relation: 568 + 1
reachable states: 157 (2^7.29462) out of 3969
(2^11.9546)
```

Временно проигнорируем сообщения, касающиеся *BDD*. Верификатор *SMV* использует *BDD* (двоичные разрешающие диаграммы) для того, чтобы представлять и хранить множество состояний компактным образом с целью расширения возможностей программы проверки моделей.

Второе свойство алгоритма взаимного исключения – *отсутствие голодания*. Это означает, что не может быть ситуации, когда процесс, который хочет войти в критическую секцию, никогда не сможет это сделать:

```
NST := AG ((prc1.label in {l1, l2, l3, l4, l5} ->
                          AF prc1.label = l6) &
           (prc2.label in {m1, m2, m3, m4, m5} ->
                          AF prc2.label = m6))
```

Альтернативная формулировка этого свойства:

```
AG AF !(prc1.label in {l1, l2, l3, l4, l5}) &
AG AF !(prc2.label in {m1, m2, m3, m4, m5})
```

Из этой формулы следует, что из любого состояния в вычислении вдоль любого пути метка процесса 1 (и 2) когда-нибудь будет отличаться от меток с `l1` по `l5` (и с `m1` по `m5`). Отсюда следует, что процессы 1 и 2 регулярно находятся в критической секции.

Проверка свойства `NST` с использованием *SMV* возвращает ошибку:

```
-- specification NST is false
-- as demonstrated by the following execution sequence
-- loop starts here –
state 1.1:
NST = 0
MUTEX = 1
t1 = bottom
t2 = bottom
y1 = bottom
y2 = bottom
prc1.label = l1
prc2.label = m1

state 1.2:
[executing process prc2]

state 1.3:
[executing process prc2]
t2 = true
prc2.label = m2

state 1.4:
[executing process prc2]
y2 = true
prc2.label = m3

state 1.5:
[executing process prc2]
prc2.label = m4

state 1.6:
[executing process prc2]
prc2.label = m5
```

```
state 1.7:
[executing process prc2]
prc2.label = m6

state 1.8:
[executing process prc2]
t2 = bottom
y2 = bottom
prc2.label = m7

state 1.9:
prc2.label = m1
```

Контрпример описан как последовательность изменений переменных. Если переменная не обозначена, то ее значение не изменилось. Контрпример, который сгенерировала программа проверки моделей, указывает, что существует цикл, в котором `prc2` многократно получает доступ к критической секции, тогда как процесс `prc1` остается ждать постоянно. Это не должно удивлять читателя, так как система *SMV* может в течение каждого шага исполнения недетерминированно выбирать процесс, который будет далее выполняться. При постоянном выборе одного и того же процесса будет получено поведение, проиллюстрированное выше. Здесь можно сказать, что этот результат не особенно интересен, так как данная несправедливая диспетчеризация процессов нежелательна. Говоря иначе, хотелось бы иметь механизм диспетчеризации, в котором активные процессы (процессы, которые могут сделать переход) выбираются *справедливым* образом. Ниже объясняется, как это может быть сделано.

Проверка условий справедливости с использованием *SMV*

SMV предоставляет возможность для спецификации *требований справедливости* с помощью следующего утверждения (для каждого процесса):

```
FAIRNESS f
```

Здесь `f` – произвольная CTL-формула. Интерпретация этой команды состоит в том, что верификатор, проверяя какую-либо спецификацию (которая может отличаться от `f`), будет игнорировать любой путь, вдоль которого `f` не встречается бесконечно часто. Такие пути называются *несправедливыми* относительно свойства `f`. Используя эту конструкцию, установим справедливость для процессов следующим путем. В верификаторе *SMV* каждый процесс имеет специальную переменную `running`, которая принимает значение true, если и только если процесс в данный момент исполняется. Если добавить `FAIRNESS running` к предыдущей спецификации *SMV* для

программы взаимного исключения, результат работы верификатора покажет, что наличие голодания теперь невозможно (так как оно несправедливо). Таким образом, для данного примера получено свойство *отсутствия голодания*. Следовательно, можно заключить, что под требованием справедливости (условием, что каждый процесс исполняется бесконечно часто) алгоритм взаимного исключения не ведет к голоданию.

Рассмотрим еще один пример использования конструкции `FAIRNESS`. Для этого немного изменим алгоритм взаимного исключения, разрешив процессу оставаться в критической секции более длительное время. Вспомним, что метка `prc1.l6` соответствует тому, что процесс 1 начинает свою критическую секцию. Изменяя код модуля `MODULE P`, получим:

```
ASSIGN
   next(label) :=
     case
     .........
     label = l6 : {l6, l7};
     .........
     esac;
```

Процесс 1 теперь может находиться в своей критической секции бесконечно долго, блокируя процесс 2 от какого-либо прогресса. Отметим, что добавление `FAIRNESS running` к вышеуказанной спецификации не помогает: процесс 2 теперь будет назначен на исполнение, но, так как он блокирован, свойство *прогресса* для него выполняться не будет. Для того чтобы избежать этого, добавим ограничение справедливости:

```
FAIRNESS  !(prc1.label = l6)
```

Этот оператор означает, что при определении выполняемости формул логики CTL квантификаторы пути **A** и **E** ранжируются по справедливым путям относительно `!(prc1.label = l6)`. Другими словами, запуски, для которых `!(prc1.label = l6)` не становится верным бесконечно часто, не рассматриваются. Это обеспечивает то, что процесс 1 не остается в своей критической секции неограниченно долго.

Глава 4. Верификация автоматных программ

4.1. Автоматные программы

В этом разделе описывается технология *автоматного программирования*, так как автоматные программы, как будет

показано ниже, могут верифицироваться с помощью проверки моделей проще и эффективнее, чем программы других классов [2, 51].

В рассматриваемом подходе к программированию выделяются источники входных воздействий и автоматизированные объекты управления, каждый из которых содержит систему управления (систему взаимодействующих конечных автоматов) и объект управления. Объект управления реализует выходные воздействия, инициируемые системой управления, а также формирует входные воздействия, реализующие обратную связь объекта управления с системой управления. Входные воздействия формируются также внешней средой (пользователем или смежными системами). Входные воздействия могут быть двух видов: события, действующие кратковременно, и входные переменные, вводимые путем опроса. Входные воздействия целесообразно реализовывать в виде переменных, используя события лишь при необходимости сократить время реакции системы. При этом одно и то же входное воздействие может быть одновременно и событием (фронт воздействия), и входной переменной (потенциал воздействия). Группы входных и выходных воздействий в общем случае связываются с состояниями, выделяемыми в каждом автомате.

На рис. 4.1 изображена схема автоматизированного объекта управления. *Парадигма автоматного программирования* состоит в представлении программ как систем *автоматизированных объектов управления*.

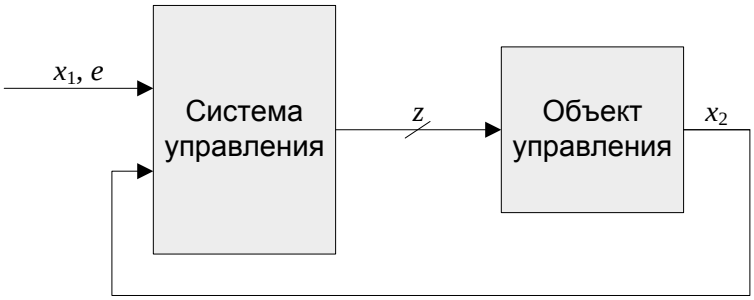


Рис. 4.1. Схема автоматизированного объекта управления

Типичной математической моделью вычислимости является машина Тьюринга (рис. 4.2). На ней можно реализовать любой алгоритм. Несмотря на это, на практике такая реализация – очень трудоемкий процесс. В автоматном подходе объединены те аспекты данной математической модели, которые подходят для практического программирования.

Базовым понятием автоматного программирования является *состояние*. Все состояния в автоматах должны быть выделены явно.

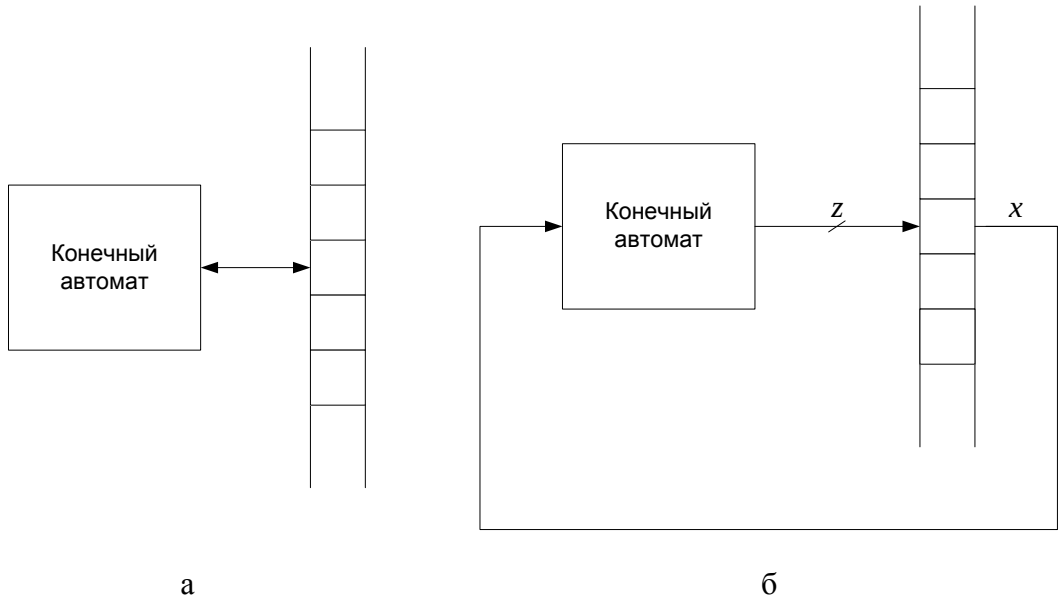


Рис. 4.2. Машина Тьюринга: классическое (а)
и модифицированное (б) изображения

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, а 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 системы, а вычислительные состояния – количественные.

В машине Тьюринга управляющее устройство с небольшим числом состояний, представляющее собой конечный автомат, может управлять бесконечным числом состояний на ленте. В автоматном программировании основное внимание уделяется управляющим состояниям. Такие состояния выделяются и перечисляются явно. В дальнейшем при использовании термина «состояние» понимается *управляющее* состояние.

В качестве документа, определяющего интерфейс автоматов в автоматном программировании, как и при автоматизации технологических процессов, используется схема связей. Применение этой схемы позволяет использовать в графах переходов и в реализующих их программах короткие символьные обозначения,

а не длинные смысловые идентификаторы, как это делается обычно. Слова в графах переходов используются только в качестве названий пронумерованных состояний. Использование символьных обозначений позволяет представлять графы переходов автоматов так, что весьма сложные графы переходов человек может «схватить» одним взглядом. При этом указанные графы могут изображаться весьма компактно и умещаться на одном экране компьютера.

Отметим, что для объектно-ориентированных программ эта схема может реализовываться диаграммой классов.

Поведение систем в рамках автоматного программирования формализуется обычно в виде графов переходов, так как в такой форме оно представляется для человека наиболее наглядно. Графы переходов отражают связи между состояниями, а также «привязку» выходных воздействий и других автоматов к состояниям и/или переходам. Дуги и петли графов переходов помечаются логическими формулами, в которых в общем случае могут содержаться символы событий и предикаты, проверяющие значения входных переменных и номера состояний других автоматов, что обеспечивает простое взаимодействие автоматов в процедурных программах. Вершины графов переходов могут содержать петли. Петли, на которых не выполняются выходные воздействия, могут умалчиваться.

В автоматном программировании принято следующее соглашение об обозначениях в автоматах. 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` или `s`, а имя выходного воздействия – с символа `z`, как указано на рис. 4.1 и 4.2 (б). После каждого из указанных символов следует номер соответствующего автомата или воздействия.

Приведем пример автоматного описания управляющих программ в виде графа переходов одного автомата (рис. 4.3).

Этот автомат управляет дверьми лифта. Работа начинается при закрытых дверях в состоянии *Closed* (*Закрыты*). При нажатии кнопки *Открыть* (событие `e11`) запускается механизм открытия дверей (`o1.z1`), и автомат переходит в состояние *Opening* (*Открываются*). При получении сообщения об открытии (`e2`) автомат переходит в состояние *Opened* (*Открыты*). Процесс закрытия происходит аналогично. При этом `e12` – нажатие кнопки *Закрыть*, `o1.z2` – запуск механизма закрытия дверей. Если какое-либо препятствие

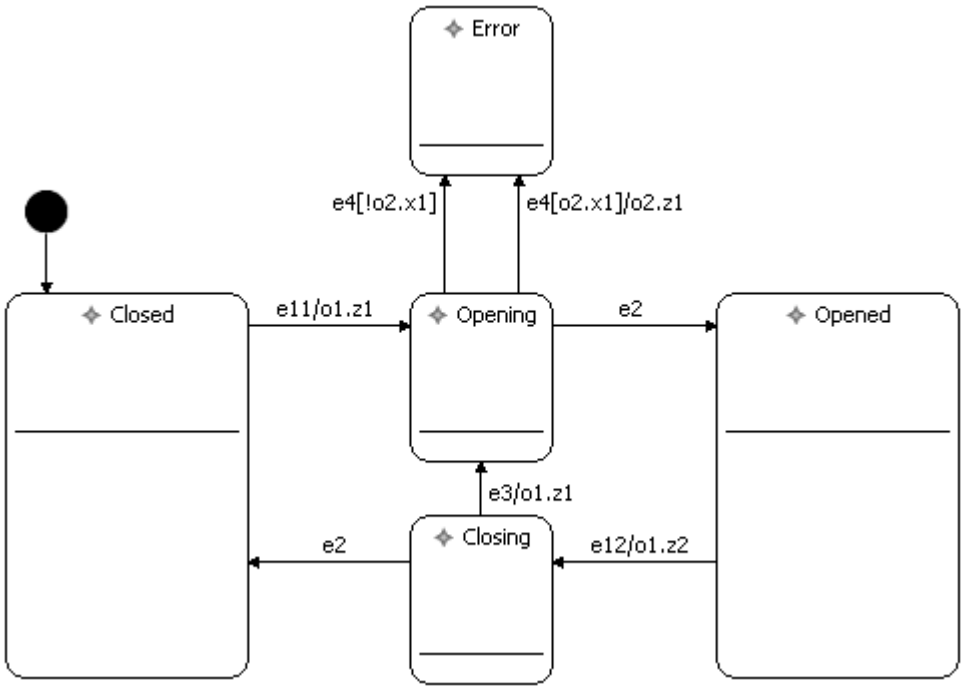


Рис. 4.3. Граф переходов автомата, управляющего дверьми лифта

мешает дверям закрыться, то происходит событие `e3`, и двери снова открываются за счет перехода автомата в состояние *Opening*. Также при открытии дверей возможно возникновение ошибки (событие `e4`), что переводит 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 в алгоритмах управления автоматы рассматриваются не изолированно, а как составные части взаимосвязанной системы – системы взаимодействующих автоматов.

Известны три способа взаимодействия автоматов между собой в процедурных программах.

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, поступающим в состояние, в которое рассматриваемый автомат вложен. 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.

В объектно-ориентированных автоматных программах взаимодействие по номерам состояний не используется.

Систему автоматов можно представить как единый автомат с помощью *прямого произведения* [52]. 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, то автомат *C* также не сможет перейти ни в одно состояние.

4.2. Обзор существующих решений

В начале исследований в рамках работ по государственному контракту по теме «Верификация автоматных программ» [53] проводился патентный поиск [54].

Темы, по которым проводился поиск:

- проверка моделей;
- темпоральная логика;
- предотвращение ошибок с помощью тестирования или отладки программного продукта.

По результатам анализа составлен краткий обзор разработок, включающий цитаты с сайтов источников, переведенные на русский язык. Основные разработки перечислены в табл. 4.1.

Таблица 4.1. Краткое описание основных разработок

Название	**Назначение, краткое описание**
Bogor	Выполняет проверку модели для собственного языка *BIR*. Поддержка большинства основных способов работы с многопоточностью. Язык *BIR* удобен для использования объектно-ориентированной парадигмы программирования. Имеет графическую оболочку на основе *Eclipse*™.
Cadena	Программа ориентирована на моделирование и верификацию CMM-систем (*CORBA Component Model*), *EJB* (*Enterprise Java Beans*). Графический интерфейс реализован как плагин для *Eclipse*™.
CADP (*Construction and Analysis of Distributed Processes*)	*CADP* – набор инструментов для разработки протоколов. Он предлагает широкий спектр функциональных возможностей: от интерактивной симуляции до самых последних техник формальной верификации. Обеспечивает эффективную компиляцию, симуляцию, формальную верификацию и тестирование описаний, записанных на языке *LOTOS* (*Language of Temporal Ordering Specifications, ISO standard 8807*). Набор инструментов, включает в себя *EVALUATOR* (программу для проведения верификации на лету) и XTL (*eXecutable Temporal Language*) – язык, реализующий различные темпоральные логики. Например, HML, CTL, ACTL и LTAC.
CBMC (*A Bounded Model Checker for C/C++ programs*)	Этот продукт выполняет проверку модели для языков ANSI C и C++. Он позволяет верифицировать выход за границы массивов, безопасность указателей, исключения и пользовательские утверждения (*assertions*). Поддерживает наиболее важные элементы ANSI C: многомерные массивы, арифметику указателей, дробную арифметику с фиксированной точкой и т. д.
GEAR (*A game based model checking tool capable of CTL, modal &-calculus and specification patterns*)	Графическая среда, которая позволяет строить модели и проводить их верификацию, используя логику CTL и μ-вычисления. Имеется много визуального материала, предназначенного для обучения. Сотрудничает с проектом *Specification patterns*, который представляет собой организацию репозитория различных требований корректности.

Java Pathfinder	Верифицирует исполняемый Java байт-код. Может находить пути выполнения, которые ведут к необработанным исключениям, блокировкам и т. п. Проверяет программы до 10 000 строк кода. Применяется в *NASA Ames Research Center*.
LTSA (*Labelled Transition System Analyser*)	Продукт предназначен для верификации распределенных и параллельных систем (concurrent systems). Они задаются с помощью *LTS* (*Labelled Transition System*) и *FSP* (*Finite State Processes*). Требования корректности задаются в виде конечных автоматов (в более ранних версия поддерживались формулы LTL). Существуют плагины, например, для верификации веб-сервисов и графический интерфейс.
MOPS (*Model Checking Programs for Security Properties*)	Верификатор моделей, извлеченных из кода программ, написанных на языке C. Требования корректности задаются в специальном виде и соответствуют утверждениям так называемого «защитного программирования» (*defensive programming*). Содержит готовую базу таких утверждений, планируется написание графического пользовательского интерфейса.
NuSMV (*A New Symbolic Model Checker*)	Это обновленная версия верификатора *SMV* – символьного верификатора моделей (*Symbolic Model Checker*). Он выполняет верификацию, комбинируя *BDD* (*Binary Decision Diagrams*) и проверку моделей, основанную на *SAT* (*SAT-based model checking*). Поддерживаемые способы задания требований корректности: CTL, LTL.
ORIS (*Uses a CTL-like temporal logic with real-time bounds, action and state based*)	Система для построения, моделирования, анализа и валидации временных сетей Петри. Выполняет также верификацию в форме проверки модели. Использует визуальную интерпретацию темпоральных формул.
SMV (*Symbolic Model Checker*)	Выполняет символьную верификацию систем, заданных в виде иерархических автоматов, для которых требования корректности задаются темпоральной логикой CTL. Утверждается, что системы можно задавать, как детализированно, так и абстрактно. Использует преобразование автоматов в модель Крипке.

SPIN	Выполняет верификацию модели для языка *PROMELA*. Этот язык поддерживает только дискретные типы данных, а также функции работы с многопоточностью. Использует логику LTL.
UPPAAL (*Uppaal Model Checker*)	Среда для моделирования и верификации систем реального времени. Использует сети временных автоматов, расширенные типами данных (ограниченные целые, массивы и т. п.). Языки моделирования – *Timed Automata*. Требования корректности задаются в подъязыке TCTL.
VIS (*Verification Interacting with Synthesis*)	Выполняет верификацию модели, используя иерархические конечные автоматы и язык Verilog. Проверяемые утверждения задаются с помощью логики CTL.
dSPIN	Расширение инструмента *SPIN*. Позволяет более эффективно верифицировать распределенные и параллельные системы. Это инструментальное средство расширяет возможности инструмента *SPIN*. Реализует некоторые новые возможности помимо алгоритмов исследования пространства состояний и сокращения числа состояний, применяемых в *SPIN*: указатели, динамическое выделение/освобождение памяти, рекурсивные функции, указатели на функцию, сборщик мусора, симметричное сокращение. Достаточно мощный входной язык (расширение языка *PROMELA*).
DBRover	Монитор времени исполнения (*runtime monitor*) для темпоральных правил, написанных на языках LTL и MTL. Это автоматическая, удаленная и графическая версия инструмента *TemporalRover*. Содержит редактор темпоральных формул, генератор и компилятор кода, симулятор. Языки моделирования: Ada, C, C++, Java, VHDL и Verilog. Поддерживаемые языки задания требований корректности: LTL, MTL (*Metric Temporal Logic*) и т. п.
Reactis Tester	Моделирует и тестирует реактивные системы. Языки моделирования: *Simulink*/*Stateflow*.
Temporal Rover	Выполняет проверку требований, сформулированных на языке LTL с ограничениями реального времени (with realtime constraints). Генерирует исполняемый код для спецификаций, записанных в виде

	комментариев. Поддерживает валидацию свойств вида «после возникновения события `e` значение переменной `x` в 5 процентах случаев не изменяется, и ее среднее значение больше 100 в течение одного часа или до того момента, когда событие `e1` произойдет дважды». Входные языки: Ada, C, C++, Java, VHDL, Verilog. Поддерживаемые способы задания требований корректности: LTL, MTL (*Metric Temporal Logic*) и т. п. Существует также проект *StateRover*, который основывается на этом инструменте, но является более новым и мощным. *StateRover* имеет более широкие возможности по сравнению с *Temporal Rover* и оперирует с подмножеством языка UML. Может генерировать код на языках Java, C, C++. Имеет мощную графическую оболочку, реализованную как плагин для *Eclipse*™.

4.3. Средства и объекты верификации

При использовании автоматного подхода в программе выделяется управляющая система, источники событий и объекты управления. Основная логика работы программы (ее поведение) моделируется в управляющей системе, тогда как объекты управления и источники событий ее практически не содержат. С одной стороны, по этой модели генерируется код (вручную или автоматически), а с другой – эта модель может использоваться для верификации, и ее, в отличие от традиционного подхода, нет необходимости строить по программе. Таким образом, в автоматной программе сразу выделяется фрагмент программы, соответствующий модели поведения, который отделен от остальной части программы. В программах, построенных традиционных путем, модель поведения априори отсутствует, и ее для верификации требуется специально строить, что обычно весьма трудоемко. При этом вопрос о корректности построенных моделей часто остается открытым. В некоторых исследованиях внимание сосредотачивается на проверке моделей для диаграмм состояний UML [55]. Основные проблемы, с которыми часто сталкиваются при решении подобных задач, связаны с тем, что в модели в явном виде рассматриваются как состояния, так и вспомогательные переменные. При этом смешиваются управляющие и вычислительные аспекты программы, поэтому работать с моделью на практике становится затруднительно. Для автоматных программ модель обычно содержит относительно немного управляющих состояний, и поэтому они

обычно могут быть верифицированы без построения более абстрактных моделей.

Таким образом, автоматные программы в значительной мере можно верифицировать автоматически. Для таких программ нет необходимости строить модель для верификации и вручную возвращаться из модели в программу при нахождении контрпримера. Это является большим их преимуществом по сравнению с программами, построенными традиционным путем. Эффективность верификации часто обратна эффективности программ, которые верифицируются. Автоматные программы обычно не являются эффективными в традиционном понимании, однако, как будет показано ниже, они весьма просто верифицируются с помощью проверки моделей. Поэтому возможно, что со временем в технических заданиях на создание ответственных систем, которые необходимо верифицировать, будет записано, что программы для них следует проектировать на основе автоматного подхода. Уже сегодня этот подход для некоторых классов таких систем постоянно применяется [56, 57].

Известно, что если схемы не контролепригодны, то их нельзя проверить, и это, например, при проектировании микропроцессоров, понимают все. Поэтому возможность проведения контроля закладывается в схему при ее проектировании. В программировании ситуация складывается иначе: программа пишется по определенным законам, обычно далеким от обеспечения возможности ее верификации. При этом на практике, даже если известно, что программу надо будет верифицировать, подходов к ее созданию не изменяют, и поэтому верификация управляющих аспектов поведения программы обычно трудоемка.

Проверка моделей после своего появления начала применяться как наиболее практичный метод из известных, но и здесь имеет место не вполне естественная ситуация: не известно ни одной области техники,, в том числе и для серийной продукции, в которой сначала объект реализуется, а затем для него разрабатывается модель. Обычно при создании практически любого образца техники все в нем спроектировано заранее. Трудно представить себе, чтобы кто-то в промышленности изготовил автомобиль, самолет или подводную лодку, а потом начал создавать модели и выпускать чертежи. Но в программировании при использовании проверки моделей обычно именно так и поступают: сначала создается программа, а потом строится ее модель. В автоматном программировании ситуация противоположная и более естественная.

Для автоматных программ существует метод создания таких программ в целом [58]. Поэтому, во-первых, автоматные программы можно проектировать, а, во-вторых, можно создавать программы в той же последовательности, как и другие образцы техники: сначала разрабатывается модель поведения, по которой формально, а часто и автоматически, она реализуется, а параллельно с реализацией по модели проводится верификация. При обнаружении ошибки модель корректируется, и вновь параллельно осуществляются реализация и верификация. В настоящее время ведутся работы по верификации автоматных программ в целом (включая функции входных воздействий и объектов управления) [59] и созданию формализмов для записи требований на языках, близких к естественному [60].

Отметим, что излагаемые в книге подходы к проверке моделей автоматных программ особенно актуальны для встраиваемых систем, для которых характерно использование около 98% процессоров и всех микроконтроллеров, выпускаемых в мире ежегодно. Значительную долю из них составляют 8-разрядные устройства, программируемые командами из одного-двух человек [61, 62].

В рамках работ по государственному контракту по теме «Верификация автоматных программ» в СПбГУ ИТМО разработано несколько верификаторов автоматных программ [63–67]. Созданные верификаторы используют разные подходы, но есть общие вопросы, которые решались при создании каждого верификатора.

Первый из них – использовать существующий верификатор программ, или реализовывать алгоритм верификации заново? Выбор существующего верификатора определяет темпоральную логику, с помощью которой можно формулировать свойства автоматной системы. При этом отметим, что наиболее известные верификаторы, например *SPIN*, работают лишь с одной разновидностью темпоральной логики. При использовании существующего верификатора возникает необходимость в разработке автоматического преобразователя автоматной программы во входной язык верификатора, что, в отличие от неавтоматных программ, осуществимо с небольшими трудозатратами.

Второй вопрос, решаемый каждым верификатором – выделение элементарных состояний автоматной программы или, другими словами, преобразование исходной программы в модель Крипке. Во время работы автоматной программы автоматы переходят из одних состояний в другие, обрабатываются события, вызываются выходные воздействия, управление передается другим автоматам. При этом

переход автомата из одного состояния в другое можно считать атомарным действием, а можно при необходимости разбить его на несколько элементарных действий, таких как вычисление условия на переходе, вызов каждого выходного воздействия, вызов вложенного автомата и т. д. Различные верификаторы могут разбивать переходы на элементарные состояния по-разному.

Следует заметить, что при использовании существующего верификатора выделение элементарных состояний происходит на этапе трансляции автоматной системы во входной язык верификатора. На входном языке верификатора программируются все элементарные состояния и переходы между ними.

Третий вопрос, который решают верификаторы автоматных программ, – это задание набора предикатов, которые могут использоваться в темпоральных формулах. Чем шире этот набор, тем более точные свойства автоматной системы можно верифицировать. Приведем примеры предикатов: «автомат *A* находится в состоянии *s1*», «произошло событие *e1*», «было вызвано действие *z1*» и т. д.

И, наконец, четвертый вопрос, который решают верификаторы автоматных программ, – это интерпретация отчета об ошибке и преобразование его в термины исходной автоматной программы. При использовании существующих верификаторов отчет об ошибке выдается в терминах модели, сформулированной на входном языке верификатора. Далее требуется преобразовать этот отчет в термины исходной автоматной программы и отобразить его пользователю верификатора. Это для автоматных программ, в отличие традиционных, эффективно выполняется автоматической процедурой.

Способы преобразования автоматной модели в модель Крипке исследуются достаточно давно. Большинство из них ставит целью построить модель так, что темпоральные формулы могли бы описать ее поведение целиком. Такое построение бывает сопряжено со следующими проблемами:

- трудности с выполнением композиции автоматов;
- неоднозначность переноса свойств модели в исходный автомат.

Например, могут возникнуть неоднозначности при интерпретации CTL-формул на автоматах Мура [68]. В следующих разделах будут рассмотрены техники, которые позволяют устранить эти неоднозначности путем выделения класса наиболее важных свойств, для которых автоматы (в том числе смешанные) могут быть преобразованы в модель Крипке без двусмысленностей.

Работы по верификации автоматных программ, кроме СПбГУ ИТМО, проводятся также и в Ярославском государственном университете им. П. Г. Демидова [69–73].

Ниже будут рассмотрены существующие верификаторы автоматных программ и инструментальные средства, их реализующие. Для демонстрации работы верификаторов будем использовать пример автоматной системы, моделирующей банкомат [74].

4.3.1. Модель банкомата

Банкомат – это устройство, автоматизирующее операции по выдаче и переводу денег, хранящихся в банке, лицу, которому они принадлежат. Идентификация каждого клиента происходит при помощи имеющейся у него карты банка и соответствующего карте секретного *pin*-кода. Банкомат осуществляет следующие операции:

- идентифицирует клиента;
- позволяет посмотреть доступные средства;
- позволяет снимать деньги;
- связывается с банком.

Модель банкомата состоит из двух автоматов. Автомат *AClient* управляет пользовательским интерфейсом, а автомат *AServer* проводит операции со счетами и связывается с банком. Кроме того, в системе работают следующие источники событий и объекты управления. Источники порождают следующие события:

- *HardwareEventProvider* – системные события, генерируемые оборудованием;
- *HumanEventProvider* – события, инициируемые пользователем;
- *ServerEventProvider* – ответы на запросы, поступающие от сервера;
- *ClientEventProvider* – запросы, поступающие на сервер.
- Объекты управления:
- *FormPainter* – обеспечивает визуализацию работы;
- *ServerQuery* – отправляет запросы на сервер;
- *ServerReply* – отвечает на клиентские запросы.

Модель банкомата построена при помощи инструментального средства *UniMod*, разработанного в СПбГУ ИТМО [58, 75, 76]. На рис. 4.4 изображена схема связей модели банкомата, на которой приведены обозначения и названия используемых событий и выходных воздействий.

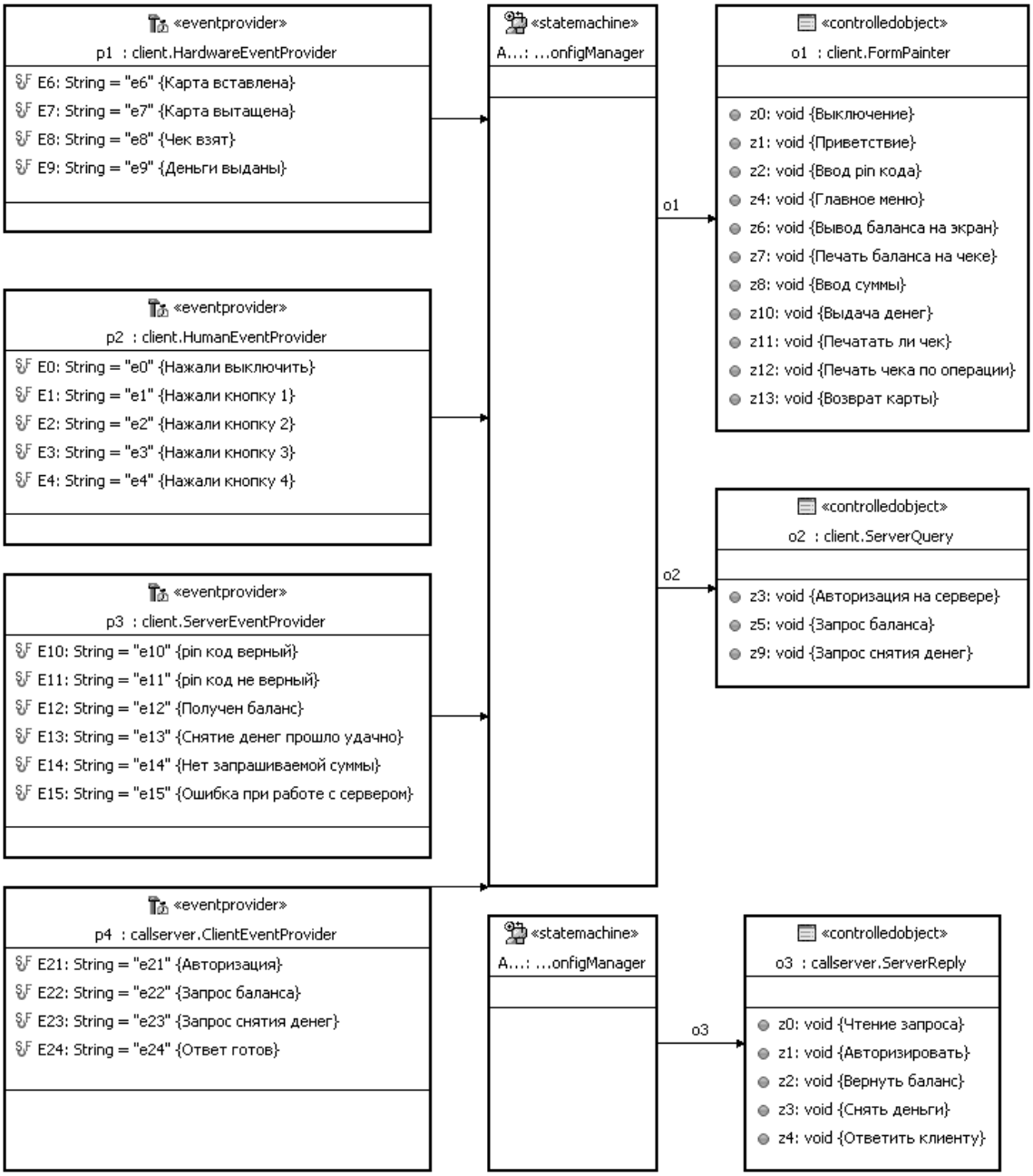


Рис. 4.4. Схема связей автоматов, источников событий и объектов управления в модели банкомата

На рис. 4.5 приведен граф переходов автомата *AClient*.

На рис. 4.6 приведен граф переходов автомата *AServer*, обрабатывающего запросы на сервер.

4.3.2. Верифицируемые свойства банкомата

Для простоты каждым верификатором будут проверяться следующие два свойства автоматной системы банкомата.

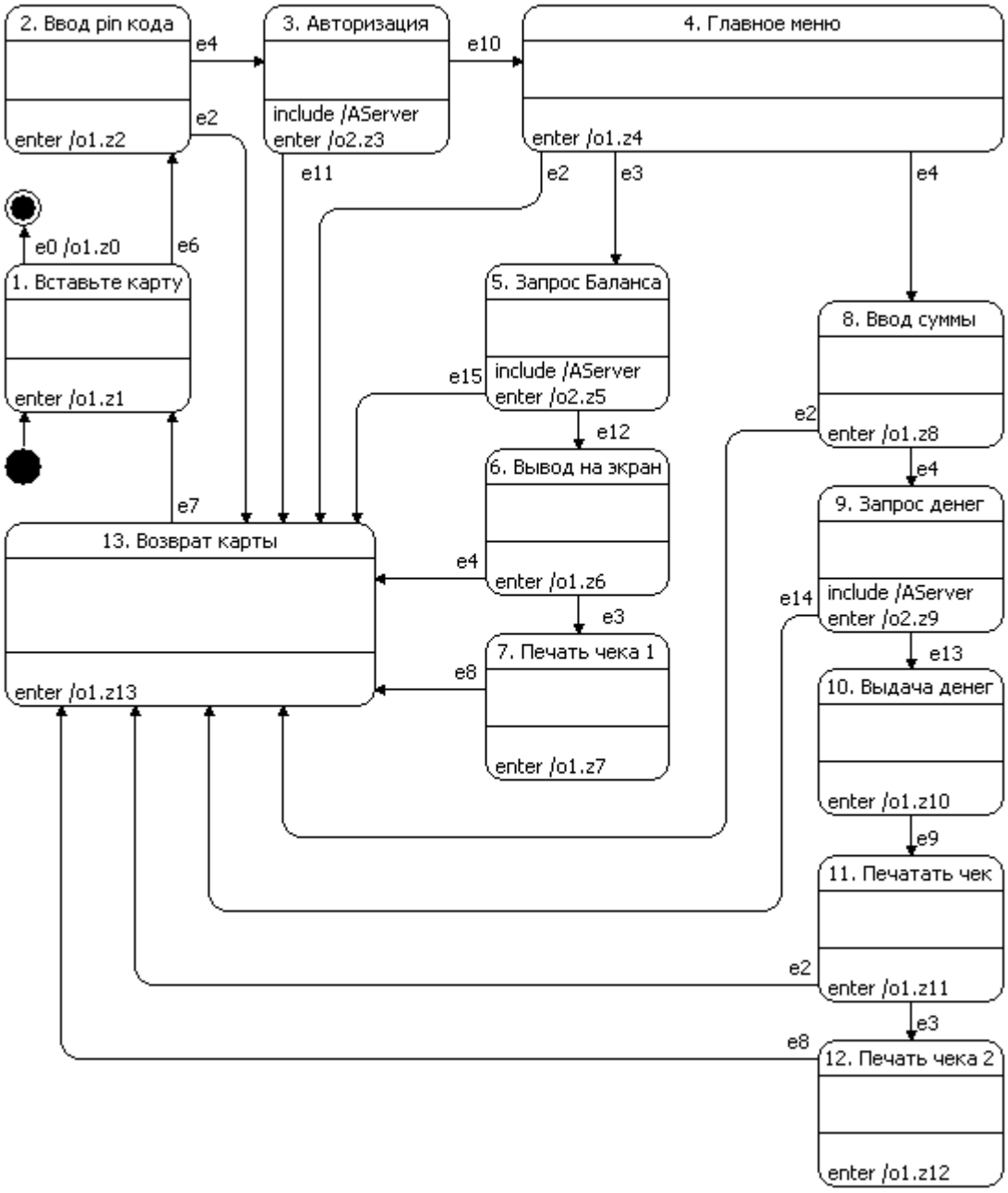


Рис. 4.5. Автомат *AClient*

Первое из них (будем обозначать его знаком Σ): «Пользователь не может получить деньги, если он не ввел правильный *pin*-код». Это свойство должно выполняться в рассматриваемом банкомате, и поэтому верификация этого свойства должна закончиться успешно. Переформулируем свойство: «Не может быть, чтобы пользователь не вводил правильный *pin*-код до того, как получил деньги». Словесная формулировка свойства напрямую переводится в темпоральную логику LTL:

$$\neg(\neg[\textit{введет правильный PIN-код}]\ \mathbf{U}\ [\textit{выдадут деньги}]).$$

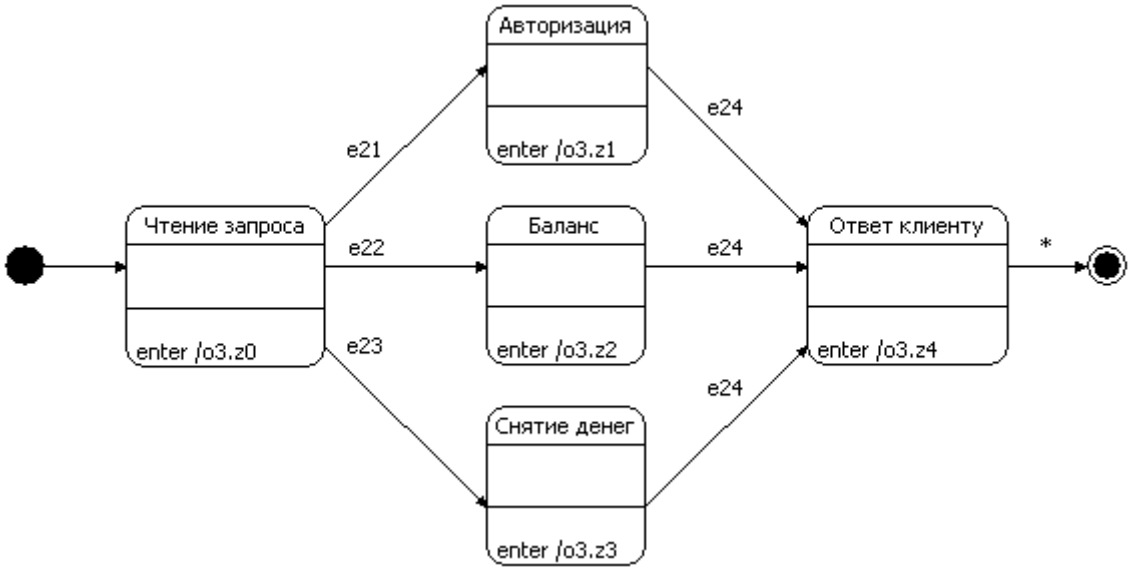


Рис. 4.6. Автомат *AServer*

Как показано на рис. 4.5, выдача денег происходит действием *o1.z10*. Поскольку один из верификаторов не позволяет проверять выполнение выходных воздействий, заметим, что действие *o1.z10* вызывается только в состоянии «*10. Выдача денег*» автомата *AClient*, и будем использовать это состояние как показатель выдачи денег. Правильно введенный *pin*-код характеризуется событием *e10*. Таким образом, формула для верификации принимает вид:

¬(¬*e*10 **U** (*AClient* in «*10. Выдача денег*»))

В логике CTL это свойство формулируется похожим образом. При этом лишь добавляется квантор «существует путь» (**E**):

¬**E**[¬*e*10 **U** (*AClient* in «*10. Выдача денег*»)].

Второе верифицируемое свойство (будем обозначать его знаком Ω): «Пользователь обязательно получит деньги». Это свойство не должно выполняться для банкомата, и верификаторы должны отобразить контрпример для этого свойства. Словесная формулировка свойства напрямую переводится в темпоральную логику LTL:

F[*выдадут деньги*].

Как обсуждалось выше, выдача денег происходит только в одном состоянии, поэтому формула принимает следующий вид:

F(*AClient* in «*10. Выдача денег*»).

Для преобразования этой формулы в логику CTL требуется лишь добавить квантор «для всех путей» (**A**):

AF(*AClient* in «*10. Выдача денег*»).

4.4. Инструменты, использующие готовые верификаторы

4.4.1. *Converter*

Общее описание

Converter [77–79] использует существующий и, пожалуй, самый известный верификатор – *SPIN* [46, 80]. Это средство предназначено для верификации формул в темпоральной логике LTL. Модель для проверки с использованием *SPIN* записывается на входном языке *Promela*.

Инструментальное средство *Converter* решает стандартные для случая использования существующего верификатора задачи: конвертация автоматной системы в язык *Promela*, определение предикатов для формулирования свойств, преобразование контрпримера в термины исходной системы. Ниже подробно описано, каким образом эти задачи автоматически решаются в верификаторе *Converter*.

Выделение атомарных состояний

Модель на языке *Promela*, полученная в результате преобразования исходной автоматной системы, содержит следующие переменные, в которых хранится состояние модели:

- `int lastEvent;` – переменная, содержащая номер последнего обработанного события;
- `int stateAi;` – такая переменная создается для каждого автомата и хранит номер текущего состояния автомата `Ai`.

Поскольку в построенной модели на языке *Promela* других переменных нет, элементарное состояние модели определяется набором значений этих переменных.

Другими словами, в данном подходе элементарное состояние автоматной системы – это набор текущих состояний всех автоматов, а также последнее обработанное событие. Переходы не разделяются на цепочки элементарных событий.

В модели на языке *Promela* для каждого автомата выделяется функция, которая недетерминированно выбирает переход из текущего состояния автомата и в зависимости от выбранного перехода обновляет значения переменных `stateAi` и `lastEvent`.

Верифицируемые свойства

Рассматриваемый верификатор проверяет формулы в темпоральной логике LTL. Поддерживаются только два типа предикатов:

- `lastEvent = e1` – выполняется, когда последнее обработанное событие `e1`;
- `stateAi = 1` – выполняется, когда номер текущего состояния автомата `Ai` в модели на языке *Promela* равен `1`. Заметим, что это не является именем соответствующего состояния в исходной автоматной системе, и поэтому пользователю нужно найти номер интересующего состояния в преобразованной модели самостоятельно.

Преобразование контрпримера

В случае если верификатор *SPIN* находит ошибку в модели, имеется возможность «воспроизвести» контрпример на модели. Это означает, что *SPIN* будет выполнять модель на языке *Promela* как настоящую программу на языке, похожем на язык программирования *C*.

Верификатор *Converter* использует эту возможность. При каждом переходе из одного состояния автомата в другое программа на языке *Promela* выводит сообщение о переходе и текущих состояниях автоматов. Таким образом, когда верификатор *SPIN* воспроизводит ошибку, в стандартный вывод печатаются эти сообщения, позволяя пользователю увидеть переходы в исходной автоматной системе. Пример вывода контрпримера представлен в листинге 4.1.

Листинг 4.1. Вывод контрпримера верификатором *Converter*

```
State Test 1 : init
Going to state Test 2 : s0
Event = e2
State Test 2 : s0
Going to state Test 33 : s1-1
Event = e3
State Test 33 : s1-1
```

Описание инструментального средства

Верификатор *Converter* работает с автоматными программами, созданными при помощи инструментального средства *UniMod*. Это средство, как отмечалось выше, позволяет визуально создавать автоматные программы, а также запускать их. При этом автоматная программа изображается на *UML*-диаграммах, а объекты управления и источники событий программируются на языке программирования *Java*.

Инструментальное средство *UniMod* позволяет сохранять автоматную программу в формате *XML*. Файл в указанном формате подается на вход верификатора *Converter*.

Дистрибутив программы *Converter* поставляется с необходимыми библиотеками и верификатором *SPIN*. Единственное, что требуется –

установить компилятор *gcc* и прописать его в переменной *PATH* для того, чтобы можно было вызвать программу *gcc* без указания ее расположения в файловой системе компьютера. Программа может запускаться только в операционных системах типа *Windows*.

Для запуска верификации требуется выполнить всего одну команду:

```
run.cmd <XML-файл системы> <файл с отчетом>
        <LTL-формула>
```

Здесь:

- `XML-файл системы` – верифицируемая автоматная программа, сохраненная с помощью *UniMod* в формате *XML*;
- `файл с отчетом` – имя файла, в который будет записан результат верификации;
- `LTL-формула` – верифицируемая формула, например: «`!(<>{lastEvent == e1})`».

Для записи формул используются следующие операторы:

- `[]` – *Globally* (всегда);
- `<>` – *Future* (когда-нибудь в будущем);
- `U` – *Until* (до тех пор пока);
- `V` – запись `p V q` эквивалентна `!(!p U !q)`;
- `!` – отрицание;
- `&&` – логическое *И*;
- `||` – логическое *ИЛИ*;
- `->` – импликация;
- `<->` – эквивалентность.

Верификация модели банкомата

Верификатор *Converter* работает с моделями *UniMod*. Поэтому дополнительных преобразований модели банкомата не требуется. Необходимо лишь использовать графический интерфейс средства *UniMod* для генерации *XML*-описания автоматной программы.

Верифицируем свойство Σ банкомата. Верификатор *Converter* создает идентификаторы для всех состояний автоматов, и требуется, чтобы верифицируемая формула использовала эти идентификаторы вместо исходных имен состояний. Поскольку заранее неизвестно, какой идентификатор будет назначен для состояния «*10. Выдача денег*», то приходится сначала запустить верификацию «вхолостую»:

```
run.cmd Bankomat.xml report.txt ""
```

Здесь `Bankomat.xml` – файл, созданный инструментальным средством *UniMod* и содержащий описание автоматной системы банкомата.

В результате выполнения команды создается файл `model.ltl`, в начале которого можно найти соответствие между идентификаторами состояний и их именами (листинг 4.2).

Листинг 4.2. Результат верификации «вхолостую»

```
...
#define STATE_10  10 /*8. Ввод суммы*/
#define STATE_11  11 /*10. Выдача денег*/
#define STATE_12  12 /*9. Запрос денег*/
...
```

Таким образом, удалось выяснить, что искомое состояние имеет идентификатор «11». Тогда верифицируемая формула принимает вид:

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

На вход верификатора *Converter* подается не верифицируемое свойство, а его отрицание. Поэтому запустить верификацию этого свойства можно следующей командой:

```
run.cmd Bankomat.xml report.txt "(!{lastEvent ==
e10} U {stateAClient == 11})"
```

В результате в консоль выводится некоторая служебная и несущественная на данный момент информация. Результат хранится в файле `report.txt` (листинг 4.3).

Листинг 4.3. Результат верификации свойства Σ банкомата

```
Converter v. 0.50
warning: for p.o. reduction to be valid the never claim
      must be stutter-invariant
(never claims generated from LTL formulae are stutter-
      invariant)
(Spin Version 4.2.8 - 6 January 2007)
      + Partial Order Reduction

Full statespace search for:
      never claim         +
      assertion violations+ (if within scope of claim)
      acceptance   cycles + (fairness disabled)
      invalid end states  - (disabled by never claim)

State-vector 32 byte, depth reached 139, errors: 0
     129 states, stored
       8 states, matched
     137 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
```

Важная информация содержится в записи «`errors: 0`» – ошибок верификации не найдено. Результат верификации ожидаемый.

Верифицируем теперь второе свойство (Ω):

```
run.cmd Bankomat.xml report.txt
        "(!<>{stateAClient == 11})"
```

В результате файл `report.txt` содержит следующую информацию (приведены только важные строки файла, листинг 4.4).

Листинг 4.4. Результат верификации свойства Ω банкомата

```
...
State-vector 28 byte, depth reached 33, errors: 1
...
Never claim moves to line 267
                [(!((stateAClient==11)))]
          State AClient 1 : s1
          Going to state AClient 13 : 1. Вставьте карту
          State AClient 13 : 1. Вставьте карту
          Going to state AClient 6 : s2
          Event = e0
          State AClient 6 : s2
spin: trail ends after 34 steps
...
```

В отчете содержится контрпример для верифицируемого свойства. Он достаточно легко читается и не требует преобразований для понимания. Банкомат начинает работу в начальном состоянии *s1*, переходит в состояние «*1. Вставьте карту*», затем нажимают кнопку *Выключить*, возникает событие *e0*, и автомат попадает в конечное состояние.

Этот контрпример соответствует следующей ситуации. Банкомат начинает работу в своем начальном состоянии, включается и предлагает вставить карту. Пользователь нажимает на кнопку *Выключить* (событие *e0*), и банкомат выключается. Попав в конечное состояние *s2*, автомат банкомата там и остается. При этом пользователь никогда не получит денег. Таким образом, найденный контрпример является действительно опровергает верифицируемое свойство.

4.4.2. *Unimod.Verifier*

Общее описание

Верификатор *UniMod.Verifier* [77, 81, 82] использует существующий верификатор *Bogor* [83]. Схема верификации в описываемом

верификаторе отличается от стандартной схемы, используемой, например, верификатором *Converter*.

Входной язык верификатора *Bogor*, который называется *BIR*, может быть расширен новыми, сложными типами данных, которые представляют собой классы, имеющие внутренние состояния. У класса можно вызывать действия, которые могут изменить внутреннее состояние, или запрашивать значения тех или иных внутренних переменных. При определенных ограничениях на новый класс средство *Bogor* может верифицировать модели, которые используют объекты нового класса.

Возможность расширить входной язык позволяет абстрагироваться от лишних деталей описания верифицируемой модели, что теоретически приводит к меньшему числу элементарных состояний модели, а следовательно, и к более простой и быстрой верификации.

В верификаторе *UniMod.Verifier* был реализован новый класс, содержащий в себе описание автоматной системы в целом. Этот класс выполняет лишь одно действие «*step*»: обработать очередное событие. При вызове этого действия класс недетерминированно выбирает событие и отдает его на обработку автоматной программе. Благодаря использованию нового класса, описание модели на входном языке верификатора *Bogor* свелось к тривиальному бесконечному циклу из одного состояния. В этом состоянии у объекта нового класса вызывается действие *step*.

Основное достоинство такого подхода состоит в том, что описание программы на входном языке верификатора стало тривиальным и одинаковым для любой автоматной программы. Логика работы модели и хранение состояний автоматов реализуется программно и один раз для всех автоматных систем. Нет необходимости генерировать новые промежуточные входные данные для каждой верифицируемой автоматной программы.

Кроме того, в верификаторе *UniMod.Verifier* применяется еще одно оригинальное решение. В других верификаторах (например, в *Converter*) используется разная реализация автоматов в рабочей программе и при верификации. В верификаторе *Converter* автоматная программа запускается и работает при помощи интерпретатора инструментального средства *UniMod*. Однако при верификации проверяется корректность автоматной программы, реализованной на языке *Promela*. При этом нет гарантий, что логика работы автоматной программы реализована одинаково в интерпретаторе инструментального средства *UniMod* и на языке *Promela*. В случае же использования *UniMod.Verifier* можно сказать, что программа *Bogor*

верифицирует автоматную систему напрямую в инструментальном средстве *UniMod*. Схема взаимодействия верификатора *Bogor* и инструментального средства *UniMod* в рамках верификатора *UniMod.Verifier* изображена на рис. 4.7.

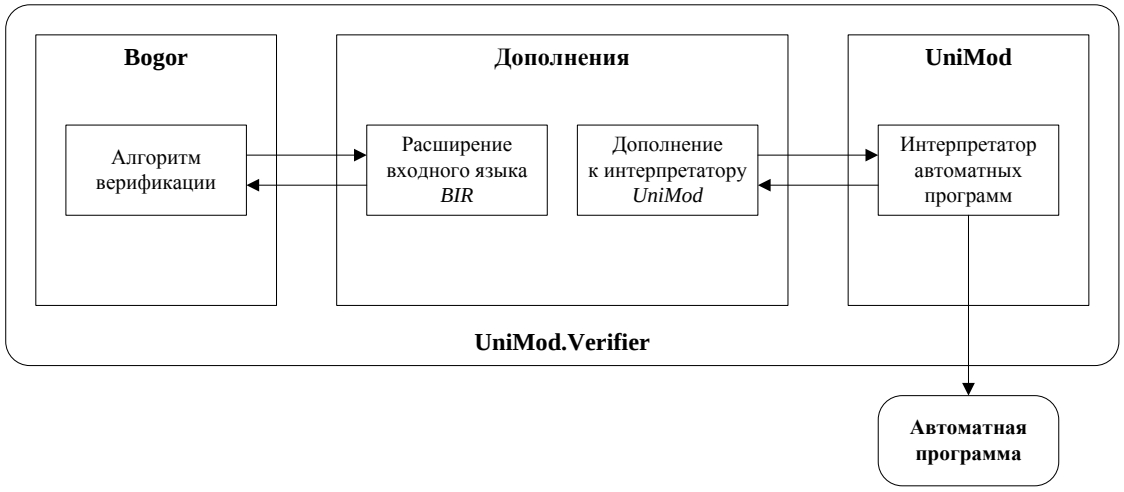


Рис. 4.7. Схема взаимодействия компонент верификатора *UniMod.Verifier*

Выделение атомарных состояний

Любой класс, расширяющий язык *BIR*, должен возвращать верификатору число или набор чисел, кодирующих его текущее состояние. Автоматное расширение, определенное в верификаторе *UniMod.Verifier*, запрограммировано возвращать набор текущих состояний автоматов в качестве состояния всей автоматной программы.

Таким образом, в данном верификаторе не производится дробления переходов автоматов на элементарные состояния.

Верифицируемые свойства

Верификатор *Bogor*, а следовательно, и *UniMod.Verifier*, позволяет верифицировать свойства, сформулированные в темпоральной логике LTL.

В качестве предикатов автоматное расширение языка *BIR* предоставляет следующие функции:

- *wasEvent(e)* – возвращает значение *True*, если в последнем шаге было выбрано для обработки событие *e*, и значение *False* – в противном случае;
- *wasInState(sm, s)* – возвращает значение *True*, если перед последним шагом автомат *sm* находился в состоянии *s*;
- *isInState(sm, s)* – возвращает значение *True*, если после совершения последнего шага автомат *sm* находится в состоянии *s*;

- *cameToState(sm, s)* – возвращает значение *True*, если после совершения последнего шага автомат *sm* изменил свое состояние на *s*. То же, что *(isInState(sm, s) && !wasInState(sm, s))*;
- *cameToFinalState()* – возвращает значение *True*, если после совершения шага головной автомат модели перешел в свое конечное состояние. Это означает, что автоматная программа завершила работу;
- *wasAction(z)* – возвращает значение *True*, если в ходе выполнения шага было вызвано выходное воздействие *z*;
- *wasFirstAction(z)* – возвращает значение *True*, если в ходе выполнения шага первым вызванным действием было *z*;
- *wasLastAction(z)* – возвращает значение *True*, если в ходе выполнения шага последним вызванным действием было *z*;
- *getActionIndex(z)* – возвращает номер действия в списке действий, вызванных в ходе выполнения последнего шага. Этот предикат предназначен для того, чтобы формулировать утверждения, задающие порядок вызова действий объектов управления в автоматной программе;
- *wasTrue(g)* – возвращает значение *True*, если в ходе выполнения последнего шага один из переходов был отмечен условием *g*, и его значение было определено как *True*. Пример условия: *g = !o1.x1 && o1.x2*;
- *wasFalse(g)* – возвращает значение *True*, если в ходе выполнения последнего шага на одном из переходов встретилось условие *g*, и его значение было определено как *False*.

Преобразование контрпримера

Поскольку верификатор работает напрямую с автоматной программой, контрпример сразу выражен в терминах исходной программы и его преобразование не требуется.

Описание инструментального средства

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

Сначала необходимо сформулировать проверяемое требование. Формула записывается в файл *Unimod.bir*, который кроме нее содержит информацию о верифицируемой модели. Для записи формулы используются следующие операторы:

- *LTL.always* – (**G**) всегда;
- *LTL.eventually* – (**F**) когда-нибудь;

- *LTL.next* – (**X**) в следующий момент во времени;
- *LTL.until* – (**U**) до тех пор, пока;
- *LTL.weakUntil* – (**W**) до тех пор, пока, или всегда;
- *LTL.release* – (**R**) освобождение: p **R** $q = \neg(\neg p$ **U** $\neg q)$;
- *LTL.negation* – отрицание;
- *LTL.equivalence* – эквивалентность;
- *LTL.implication* – импликация;
- *LTL.conjunction – И*;
- *LTL.disjunction – ИЛИ*.

Формула записывается в файл *Unimod.bir* в виде функции. Поясним на примере формат этой функции (листинг 4.5).

Листинг 4.5. Функция, описывающая формулу LTL

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

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

Эта функция выражает темпоральную формулу *!o1.z10 W e10*. В начале описания функции объявляются предикаты. Например, *"correct_pin"* – имя предиката, а *AutomataModel.wasEvent(model, "e10")* – его значение. Такой предикат верен тогда, когда последнее обработанное событие было *e10*.

После объявления предикатов записывается сама формула с использованием этих предикатов в виде вложенных вызовов функций.

Теперь вызывается верификатор. Это выполняется следующей командой:

```
verifier.cmd Bankomat.xml NoPinNoMoney
```

Здесь `Bankomat.xml` – файл, созданный инструментальным средством *UniMod* и содержащий описание автоматной системы банкомата, `NoPinNoMoney` – имя функции для верификации, описанной в файле *Unimod.bir*.

В результате вызова этой команды в ходе процесса верификации в стандартный поток вывода будет напечатана служебная информация. После окончания верификации выведется либо сообщение об успешном завершении, либо контрпример. Контрпример описывает по шагам состояния автоматов.

Верификация модели банкомата

Также как и верификатор *Converter*, верификатор *UniMod.Verifier* работает с *XML*-описанием автоматной программы, сгенерированной инструментальным средством *UniMod*.

Верифицируем первое свойство (Σ). Для того чтобы записать формулу, не требуется дополнительная информация – для ее записи можно использовать имена исходной автоматной программы. Однако такая запись формулы достаточно громоздка. Свойство Σ записывается в файле *Unimod.bir* в виде функции, приведенной в листинге 4.6.

Листинг 4.6. Запись свойства Σ банкомата для *UniMod.Verifier*

```
fun NoPinNoMoney() returns boolean =
   LTL.temporalProperty(
      Property.createObservableDictionary(
         Property.createObservableKey("correct_pin",
            AutomataModel.wasEvent(model, "e10")),
         Property.createObservableKey("give_money",
            AutomataModel.isInState(model, "/AClient",
            "10. Выдача денег"))
      ),

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

В функции сначала объявляются предикаты *correct_pin* и *give_money*, первый из которых означает, что произошло событие *e10*, а второй – что корневой автомат *AClient* находится в состоянии «*10. Выдача денег*». Затем записывается темпоральная формула с использованием этих предикатов: *!(!<correct_pin> U <give_money>)*.

Инструкция для верификации этого свойства:

```
verifier.cmd Bankomat.connectivity NoPinNoMoney
```

В результате в консоль выводится следующая информация (листинг 4.7).

Листинг 4.7. Результат верификации свойства Σ банкомата

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

Transitions: 1, States: 1, Matched States: 0, Max
  Depth: 1, Errors found: 0, Used Memory: 2MB
Transitions: 63, States: 41, Matched States: 22, Max
  Depth: 14, Errors found: 0, Used Memory: 1MB
Total memory before search: 708a688 bytes (0,68 Mb)
Total memory after search: 1a134a712 bytes (1,08 Mb)
Total search time: 688 ms (0:0:0)
States count: 41
Matched states count: 22
Max depth: 14
Done!
Verification successful!
```

В последней строке написано, что верификация завершилась успешно, как и ожидалось.

Верифицируем теперь второе свойство (Ω). Запишем его в файл *Unimod.bir* в виде функции (листинг 4.8).

Листинг 4.8. Запись свойства Ω банкомата для *UniMod.Verifier*

```
fun AlwaysMoney() returns boolean =
   LTL.temporalProperty (
      Property.createObservableDictionary (
         Property.createObservableKey("give_money",
           AutomataModel.isInState(model, "/AClient",
           "10. Выдача денег"))
      ),

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

Запускаем процесс верификации, выводя результат в файл *verifier.out*:

```
verifier.cmd Bankomat.connectivity AlwaysMoney
             > verifier.out
```

После выполнения этой команды в файле *verifier.out* оказывается следующая информация (в листинге 4.9 приведены только важные строки).

Листинг 4.9. Результат верификации свойства Ω банкомата

```
Generating error trace 0...
Done!
1 traces were found.
Replaying the trace with least states (#0).
Replaying trace by key: 0
Stack of transitions leading to the error:
```

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

Файл содержит контрпример. В каждой строке контрпримера приводится вся возможная информация: номер шага, обработанное событие, выполненные переходы и текущие состояния каждого автомата.

Найденный контрпример соответствует следующей ситуации. Банкомат начинает работу, головной автомат попадает в состояние «*1. Вставьте карту*». Карта вставляется (событие *e6*), головной автомат попадает в состояние «*2. Ввод pin-кода*». Далее пользователь нажимает на кнопку *Отмена*, автомат попадает в состояние «*13. Возврат карты*», и, когда пользователь забрал карту (событие *e7*), головной автомат вновь попадает в состояние «*1. Вставьте карту*».

Полученный результат является контрпримером, так как он представляет собой такой цикл, что автоматная программа банкомата при неограниченном прохождении по этому циклу никогда не выдаст пользователю деньги.

4.4.3. *FSM Verifier*

Общее описание

FSM Verifier [65, 77] основывается на верификаторе *NuSMV*, который верифицирует модели, описанные на языке *SMV*. Схема работы верификатора состоит из стандартных для такого подхода шагов.

1. Преобразовать автоматную систему в модель на языке *SMV*.
2. Преобразовать требования к системе в формулы темпоральной логики.
3. Запустить верификатор *NuSMV*.
4. Преобразовать контрпример к модели в контрпример к исходной системе автоматов.

Эта схема изображена на рис. 4.8.

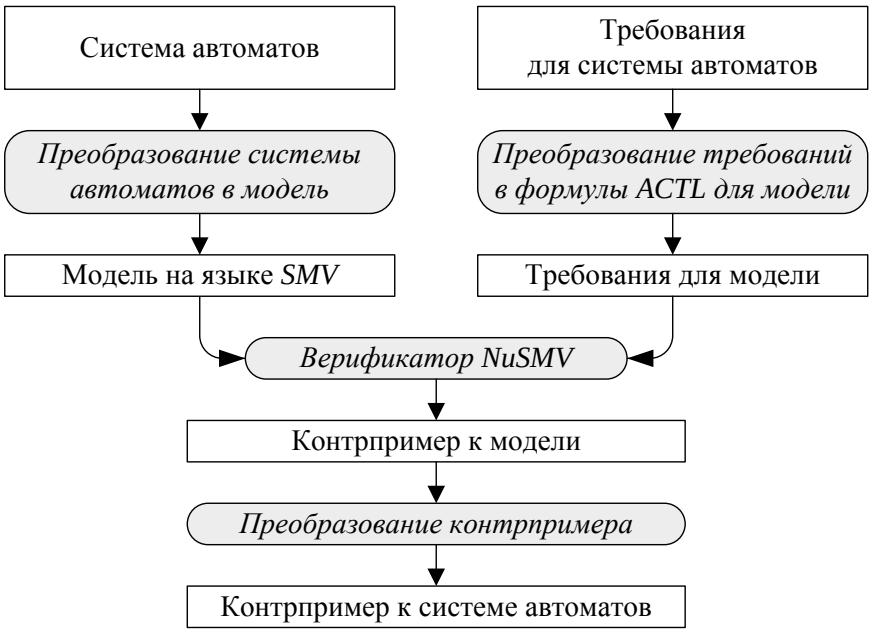


Рис. 4.8. Схема работы верификатора *FSM Verifier*

Опишем систему автоматов, которая верифицируется рассматриваемым средством. Задан набор конечных автоматов с несколькими выходными воздействиями на ребрах и выходными воздействиями при входе в состояния. Для каждого автомата задается набор событий, которые он может обрабатывать. Для каждого события задается, поступает оно от источника событий или только от автомата.

Каждый переход может содержать событие, при котором он активируется, и дополнительное условие, необходимое для его активации. Условие представляет собой логическую формулу,

содержащую в качестве предикатов входные переменные (x_1, x_2) и выражения вида A_i in s_j (верно, если автомат A_i находится в состоянии s_j). Входные переменные возвращают только булевы значения. Переход также может содержать последовательность выходных воздействий вида $o.z_i()$ и команду вида $A_i.e_j()$ для передачи управления другим автоматам.

Выделение атомарных состояний

FSM Verifier в автоматах верифицируемой программы выделяет, помимо основных состояний, множество промежуточных состояний, в которые автомат попадает во время перехода из одного состояния в другое. Промежуточное состояние фиксируется каждый раз, когда автомат выполняет одно из следующих действий:

- вызывает выходное воздействие;
- вызывает другой автомат.

На рис. 4.9 приведен пример выделения промежуточных состояний для одного перехода.

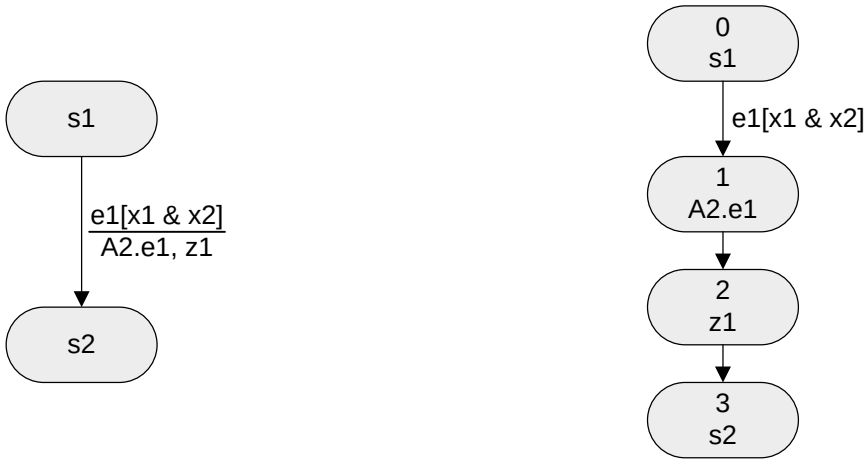


Рис. 4.9. Выделение промежуточных состояний на переходе. Исходный переход (а), преобразованный переход (б)

Верифицируемые свойства

FSM Verifier позволяет верифицировать темпоральные формулы с использованием следующих предикатов.

- автомат A_k находится в состоянии s_j;
- выполнилось выходное воздействие z_i;
- произошло событие e_i.

Верификатор *NuSMV*, а следовательно, и *FSM Verifier*, позволяет верифицировать свойства, сформулированные в темпоральной логике

CTL. В формулах возможно использование следующих темпоральных операторов: **AF** *f*, **AG** *f*, **A**[*f* **U** *g*].

Кроме того, в формулах можно использовать стандартные логические операторы.

Преобразование контрпримера

Отчет об ошибке верификации, возвращаемый верификатором *NuSMV*, содержит контрпример в терминах построенной модели автоматной системы. Контрпример содержит последовательность элементарных состояний, приводящих к ошибке. Каждое элементарное состояние однозначно определяет состояние или переход в исходной автоматной программе. Поэтому не составляет труда преобразовать контрпример в термины исходной системы автоматов. Реализация этого преобразования в методе *FSM Verifier* подробнее излагается в работе [65].

Описание инструментального средства

Дистрибутив инструментального средства содержит два файла:

- *verifier.jar* – программа, преобразующая систему автоматов в модель на языке *SMV*;
- *counterexample.jar* – программа, преобразующая контрпример, выданный верификатором *NuSMV*, в контрпример для системы автоматов.

Кроме того, для работы инструментального средства необходимы следующие программы:

- верификатор *NuSMV*;
- *Java Runtime Environment*.

Все перечисленные программы могут работать в операционных системах типа *Windows* и *Linux*.

FSM Verifier принимает на вход автоматную систему, записанную в формате *XML*. Структура входного файла была разработана специально для *FSM Verifier* и не поддерживается другими программами. Структуру входного файла можно найти в работе [77].

Формула для верификации записывается в тот же *XML*-файл в простом виде (пример записи формулы приведен в листинге 4.10).

Листинг 4.10. Запись формулы в верификаторе *FSM Verifier*

```
<specification>
  <string>AG (A0.s1 -&gt; AF A1.s1)</string>
</specification>
```

Для верификации следует выполнить три команды:

- из автоматной системы и формулы, записанных в файл *inputfile.fsm*, создается модель автоматной системы на языке *SMV* в файле *input.smv*:

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

- вызывается верификатор *NuSMV*:

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

- контрпример преобразуется в термины исходной автоматной системы:

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

В результате программа выводит в стандартный вывод контрпример в *HTML*-формате в виде таблицы. В каждой строке таблицы указана следующая информация:

- номер шага;
- имя активного автомата;
- обрабатываемое событие;
- имена состояний всех автоматов;
- выполняемое действие;
- значение входных воздействий.

Верификация модели банкомата

Автоматная система верифицируемого банкомата реализована при помощи инструментального средства *UniMod* и автоматически сохраняется в специальном формате *XML*, который отличается от формата, принимаемого на вход верификатором *FSM Verifier*. Для поддержки этого формата используется алгоритм, который конвертирует модель инструментального средства *UniMod* в модель верификатора *FSM Verifier*.

В инструментальном средстве *UniMod* имеется возможность присвоения любых имен состояниям автоматов. Однако при использовании верификатора *FSM Verifier* возникают сложности, если имена состояний содержат пробелы и другие специальные символы. Поэтому при конвертации автоматной системы из формата *UniMod* в формат *FSM Verifier* исходным состояниям присваиваются новые идентификаторы. Кроме того, изменяются имена некоторых специальных событий и всех выходных воздействий: это необходимо для работы верификатора *FSM Verifier* с автоматной системой.

В результате конвертации состояние автомата *AClient* «*10. Выдача денег*» получило идентификатор *s11*, поэтому предикат

AClient in «10. Выдача денег» принимает вид *AClient.s11*. Также в верификаторе *FSM Verifier* при записи предиката текущего события требуется записывать, каким автоматом оно обрабатывается. На рис. 4.5, 4.6 видно, что событие *e10* обрабатывается только автоматом *AClient*. Поэтому предикат *e10* принимает вид *AClient.e10*. Запись верифицируемого свойства Σ приведена в листинге 4.11.

Листинг 4.11. Запись свойства Σ банкомата

```
<specification>
  <string>!E[!AClient.e10 U AClient.s11]</string>
</specification>
```

Далее, следуя документации *FSM Verifier*, выполняем следующую команду:

```
java –jar verifyer.jar Bankomat.fsm
```

Здесь *Bankomat.fsm* – файл, сгенерированный в результате конвертации автоматной системы из формата *UniMod* в формат *FSM Verifier*. Этот файл, помимо автоматной модели, содержит проверяемую спецификацию. В результате исполнения команды в каталоге верификатора создается файл *out.smv* – выходной файл для верификатора *NuSMV*.

Следующим действием запускается верификатор *NuSMV*:

```
NuSMV out.smv
```

В результате его выполнения в консоль выводится текст, представленный в листинге 4.12.

Листинг 4.12. Результат верификации свойства Σ банкомата

```
C:\Verifiers\FSM Verifier>NuSMV out.smv
*** This is NuSMV 2.4.3 (compiled on Tue May 22 14:08:54
UTC 2007)
*** For more information on NuSMV see
<http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.

*** This version of NuSMV is linked to the MiniSat SAT
solver.
*** See
http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
*** Copyright I 2003-2005, Niklas Een, Niklas Sorensson

-- specification !E [ !AClient.e10 U AClient.s11 ]   is
true
```

В последней строке вывода указано, что спецификация была успешно проверена. Как и ожидалось, пользователь не может получить деньги до того, как введет правильный *pin*-код.

Верифицируем теперь свойство Ω банкомата. Запись этого свойства для верификатора *FSM Verifier* приведена в листинге 4.13.

Листинг 4.13. Запись свойства Ω банкомата

```
<specification>
  <string>AF AClient.s11</string>
</specification>
```

Генерируем входной файл для верификатора *NuSMV*:

```
java –jar verifyer.jar Bankomat.fsm
```

Запускаем верификатор *NuSMV*:

```
NuSMV out.smv
```

Результат выполнения этой команды представлен в листинге 4.14 (приведена только важная начальная часть).

Листинг 4.14. Результат верификации свойства Ω банкомата

```
C:\Verifiers\FSM Verifier>NuSMV out.smv
*** This is NuSMV 2.4.3 (compiled on Tue May 22
14:08:54 UTC 2007)
*** For more information on NuSMV see
<http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.

*** This version of NuSMV is linked to the MiniSat SAT
solver.
*** See
http://www.cs.chalmers.se/Cs/Research/FormalMethods/Min
iSat
*** Copyright I 2003-2005, Niklas Een, Niklas Sorensson

-- specification AF AClient.s11  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  AClient.State = 0
  AServer.State = 0
  Active = 0
  Event = 0
AClient.s9 = 0
...
```

Верификатор *NuSMV* нашел контрпример для свойства. Сохраним его вывод в файл:

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

Теперь преобразуем контрпример в термины исходной автоматной системы с помощью верификатора *FSM Verifier*:

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

В результате в папке верификатора создается файл *out.html*. Если открыть его в браузере, то отобразится контрпример, изображенный в табл. 4.2.

Таблица 4.2. Контрпример для свойства Ω банкомата, сгенерированный программой *FSM Verifier*

Step	Active	Event	AClient	AServer	Action
1	AClient	–	s1	s1	–
2	AClient	eAlways	s1	s1	o1.z1
3	AClient	–	s13	s1	–
4	AClient	e6	s13	s1	o1.z2
5	AClient	–	s9	s1	–
6	AClient	e2	s9	s1	o1.z13
7	AClient	–	s7	s1	–
8	AClient	e7	s7	s1	o1.z1
9	AClient	–	s13	s1	–

Для того чтобы разобраться в этом контрпримере, требуется сначала определить соответствие между именами состояний исходной *UniMod*-модели банкомата и именами, используемыми в модели, сконвертированной для *FSM Verifier*. В данном примере оно следующее (соответствие приводится лишь для интересующих состояний автомата *AClient*):

- *s1* – начальное состояние банкомата в *UniMod*-модели;
- *s13* – «*1. Вставьте карту*»;
- *s9* – «*2. Ввод pin-кода*»;
- *s7* – «*13. Возврат карты*».

Этот контрпример соответствует следующей ситуации. Банкомат начинает работу в начальном состоянии. Пользователь вставляет карту (событие *e6*), затем нажимает на кнопку *Отмена* (событие *e2*), и карта возвращается (событие *e7*), банкомат возвращается в прежнее состояние (*s1*). Далее все повторяется: пользователь вставляет карту и снова нажимает кнопку *Отмена*. Это может продолжаться бесконечно долго, и банкомат никогда не попадет в состояние выдачи денег. Действительно, как видно на рис. 4.5, состояния

«*1. Вставьте карту*», «*2. Ввод pin-кода*» и «*13. Возврат карты*» образуют цикл.

Вывод контрпримера завершается тогда, когда появляется уже посещенное состояние. Действительно, шаги 3 и 9 контрпримера показывают эквивалентное состояние автоматной системы. Таким образом, шаги от 4-го до 9-го образуют цикл, а шаги с 1-го по 3-й показывают путь системы из стартового состояния до этого цикла. Другими словами, следующий бесконечный сценарий работы банкомата нарушает верифицируемое свойство (по шагам контрпримера):

1 2 3 4 5 6 7 8 9 4 5 6 7 8 9 4 5 6 7 8 9 4 …

4.5. Автономные верификаторы

4.5.1. *CTL Verifier*

Общее описание

Инструментальное средство *CTL Verifier* [63, 77, 84] не использует других верификаторов, и в этом средстве реализован алгоритм верификации формул темпоральной логики CTL.

В этом верификаторе нет необходимости преобразовывать автоматную программу во входной язык другого верификатора. Тем не менее, ее требуется преобразовывать во внутреннюю модель Крипке, верифицируемую инструментальным средством. Результат верификации также будет выражен в терминах состояний модели Крипке. Поэтому в данном методе присутствует преобразование контрпримера.

Выделение атомарных состояний

Для модели Крипке $M = (S, R, Label)$ будем рассматривать маркирующее отношение $Label$ как подмножество $Label \subseteq S \times AP$. Вместо $R(s, t)$ будем писать $s \to t$.

Рассмотрим программу, модель которой задается системой автоматов, взаимодействующих по вложенности. Необходимо преобразовать эту модель в единую модель Крипке, целиком описывающую поведение данной системы.

Прежде всего, для множества автоматов выполняется топологическая сортировка по отношению вложенности. Это отношение не должно содержать циклов, иначе полученная модель будет иметь бесконечный размер. Модель Крипке строится индуктивно для каждого автомата системы, причем автоматы обрабатываются в

порядке, который был сформирован топологической сортировкой. Такой порядок означает, что при обработке внешних автоматов будут уже обработаны вложенные в них автоматы (для них будет подготовлена модель Крипке).

Метод полного графа переходов

Метод, описанный в этом разделе, является наиболее выразительным в том смысле, что с его помощью можно верифицировать свойства, охватывающие все качественные аспекты поведения автоматных программ. Он позволяет работать с автоматами, в которых в общем случае переходы могут выполняться параллельно. Будем пока предполагать этот общий случай.

В данном методе множество *AP* определяется следующим образом:

$$AP = \{Y_1, Y_2, \ldots\} \cup \{e_1, e_2, \ldots\} \cup \{x_1, x_2, \ldots\} \cup \{z_1, z_2, \ldots\} \cup$$

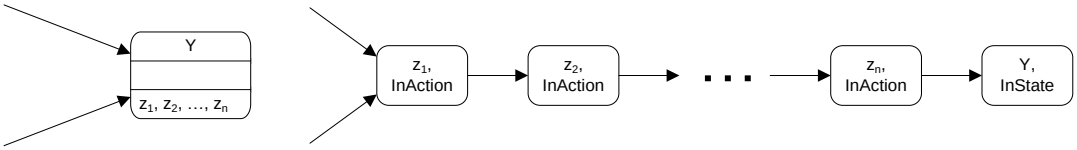
$$\cup \{InState, InEvent, InAction\} \cup Names.$$

Здесь $\{Y_1, Y_2, \ldots\}$ – множество наименований для всех состояний автоматов, $\{e_1, e_2, \ldots\}$ – указанное множество для событий, $\{x_1, x_2, \ldots\}$ – указанное множество для входных воздействий, $\{z_1, z_2, \ldots\}$ – указанное множество для выходных воздействий. *Names* – множество наименований самих автоматов, а *InState*, *InEvent* и *InAction* – управляющие атомарные предложения, предназначенные для того, чтобы было удобно различать вершины, построенные из состояний, событий и выходных воздействий.

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

После этого для каждого состояния *s* выполняем следующую операцию. Пусть *s* содержит выходные воздействия $z_{s[1]}, \ldots, z_{s[u]}$, которые выполняются при входе в состояние *s*. Добавим в модель *u* состояний $\{r_1, \ldots, r_u\}$ и *u* переходов $r_1 \to r_2, \ldots, r_{u-1} \to r_u, r_u \to s$. В отношение *Label* добавим пометки $(r_k, z_{s[k]})$, $(r_k, InAction)$ для всех *k* от 1 до *u*. Далее, при добавлении ребер в модель на следующих этапах, каждое ребро, которое идет в состояние *s*, будем перенаправлять в состояние r_1.

Пример такого преобразования проиллюстрирован на рис. 4.10.



а б

Рис. 4.10. Состояние с выходными воздействиями до преобразования (а) и после (б)

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

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

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

Пример такого преобразования приведен на рис. 4.11.

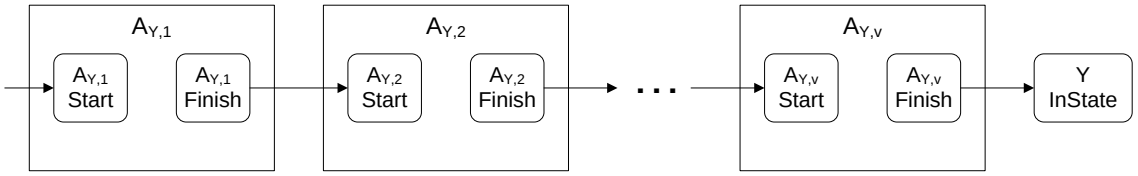


Рис. 4.11. Обработка автоматов, вложенных в состояние

Обратим внимание, что в состояния автомата A могут быть вложены различные «копии» одного и того же автомата B. В этом случае для

каждой копии создается своя модель Крипке (все эти модели изоморфны друг другу) и перенаправление ребер выполняется для нее. Размер полученной модели Крипке (число ее состояний) будет ограничен сверху произведением размеров моделей Крипке для каждого автомата в отдельности (без учета вложенных).

Приведем пример работы этого алгоритма для системы, эмулирующей работу банкомата. Система состоит из двух автоматов *AClient* и *AServer*, причем автомат *AServer* вложен в автомат *AClient*. Схема связей для соответствующей автоматной модели изображена на рис. 4.4, а графы переходов автоматов *AServer* и *AClient* – соответственно на рис. 4.5 и 4.6.

Для этой системы модель Крипке имеет большой размер. Поэтому она отображена на рис. 4.12 в упрощенном виде: во внешний автомат подставлены копии внутреннего автомата именно в тех местах, где происходит вход в него. Вершины модели Крипке представлены не отдельными блоками, а строками текста, при этом видны особенности обработки вложенных автоматов.

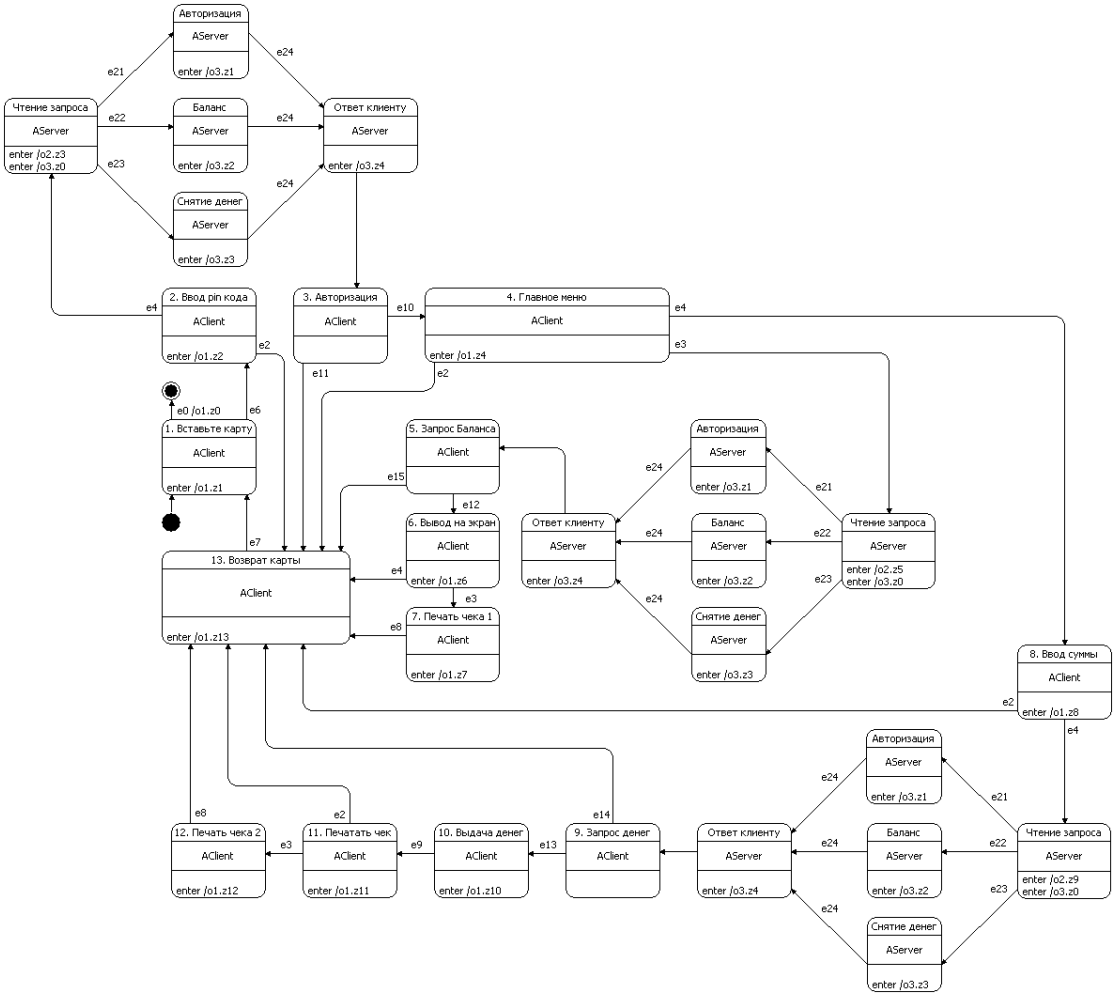


Рис. 4.12. Сгенерированный граф переходов для модели банкомата

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

Рассмотрим набор из всех булевых переменных в исходном автомате, состоящий из событий и выходных воздействий. Для каждого состояния p и каждой двоичной последовательности, которую можно присвоить переменным этого набора, определим сценарий, который должен произойти в этом состоянии при условии, что значением набора стала данная последовательность. Сценарий этот можно описать на естественном языке следующим образом. В состоянии p произошли события $e_{i[1]}, \ldots, e_{i[s]}$. При этом входные воздействия $x_{j[1]}, \ldots, x_{j[t]}$ (и только они) оказались истинными. После этого были вызваны выходные воздействия $z_{k[1]}, \ldots, z_{k[u]}$, и автомат перешел в состояние q. Для каждого такого сценария r создадим $u + 1$ дополнительное состояние $\{r_e, r_1, \ldots, r_u\}$, $u + 2$ перехода: $p \to r_e$, $r_e \to r_1$, $r_1 \to r_2$, $\ldots$, $r_{u-1} \to r_u$, $r_u \to q$, а в отношение *Label* добавим пометки $(r_e, e_{i[i^*]})$, $(r_e, x_{j[j^*]})$ $(r_e, \mathit{InEvent})$, $(r_{k^*}, z_{k[k^*]})$, $(r_{k^*}, \mathit{InAction})$ для всех i^* от 1 до s, j^* от 1 до t и k^* от 1 до u.

Если требуется построить модель для автомата, в котором на каждом шаге не происходит более одного события, то выполним тот же самый алгоритм, но с одним отличием: вместо всех возможных двоичных наборов будем перебирать только те из них, в которых истинна не более чем одна переменная e_i. Таким образом, данный перебор будет эквивалентен перебору двоичных наборов, которые могут быть присвоены входным воздействиям, и такой перебор выполняется для каждого события. При этом число состояний полученной модели ограничено снизу числом

```
(число_состояний_в_исходном_автомате + 1) ×
× число_двоичных_наборов,
```

а число соответствующих двоичных наборов равно $2^{\text{число\_переменных}}$.

Конвертация автомата в модель Крипке на основе этого метода продемонстрирована на примере автомата *ATrig*, эмулирующего работу *RS*-триггера [85] (рис. 4.13, 4.14).

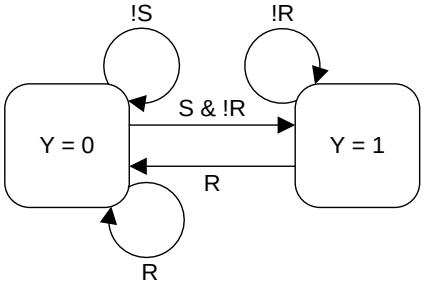


Рис. 4.13. Граф переходов автомата *ATrig*

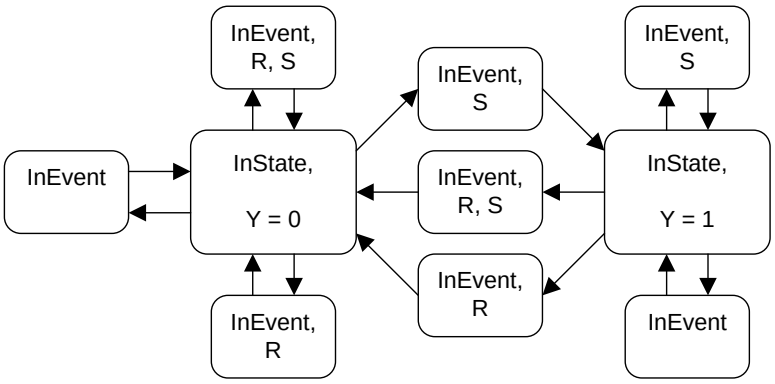


Рис. 4.14. Модель Крипке для автомата *ATrig*

В качестве примера можно проверить CTL-свойство $\neg(S \vee R) \rightarrow \mathbf{EX}\ \mathbf{EX}\ (Y = 1)$, утверждающее, что если оба входных воздействия отсутствуют, то, преодолев переход, на следующем шаге можно оказаться в состоянии `1`. Это свойство выполняется во всех состояниях, за исключением тех, которые ничем не помечены, кроме «служебного слова» *InEvent*.

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

Метод редуцированного графа переходов

В данном методе множество *AP* определяется соотношением:

$$AP = \{Y_1, Y_2, \ldots\} \cup \{e_1, e_2, \ldots\} \cup \{x_1, x_2, \ldots\} \cup \{!x_1, !x_2, \ldots\} \cup$$

$$\cup \{z_1, z_2, \ldots\} \cup \{InState, InEvent, InAction\} \cup Names.$$

Начинаем, как и в методе полного графа переходов, с того, что присвоим переменной *S* множество состояний исходного автомата и для каждого состояния *s* добавим в отношение *Label* две пометки: (*s*, *s*) и (*s*, *InState*). После этого, как и ранее, выполним разделение выходных переменных и состояний. После этого отделим состояния от вложенных в них автоматов.

Перейдем к описанию того, как обрабатываются переходы.

Рассмотрим множество следующих символов:

$$\{x_1, !x_1;\ x_2, !x_2;\ x_3, !x_3; \ldots\}.$$

Можно сказать, что это множество всех литералов, составленных из входных переменных. Следует различать смысл знаков «¬» и «!». Первый из них означает выполнение операции логического отрицания, а второй интерпретируется как символ (часть строки «!x_i»).

Тогда для каждого ребра r исходного автомата, которое ведет из состояния p в состояние q с пометкой

$$e_i \,\&\, h_{j[1]} \,\&\, h_{j[2]} \,\&\, h_{j[3]} \,\&\, \ldots \,\&\, h_{j[m]} / z_{i[1]}, \ldots, z_{i[n]}$$

(здесь $h_{j[j^*]} = x_{j[j^*]}$, либо $h_{j[j^*]} = !x_{j[j^*]}$ – это значит, что $h_{j[j^*]}$ либо входная переменная, либо ее отрицание), добавим в модель $n + 1$ состояние $\{r_e, r_1, \ldots, r_n\}$ и $n + 2$ перехода:

$$p \to r_e,\ r_e \to r_1,\ r_1 \to r_2,\ \ldots,\ r_{n-1} \to r_n,\ r_n \to q.$$

В отношение *Label* добавим пометки (r_e, e_i), $(r_e, InEvent)$, $(r_k, z_{i[k]})$, $(r_k, InAction)$ для всех k от 1 до n, а также пометки

$$(r_e, h_{i[1]}), (r_e, h_{i[2]}), \ldots, (r_e, h_{i[m]}).$$

Пример такого преобразования приведен на рис. 4.15.

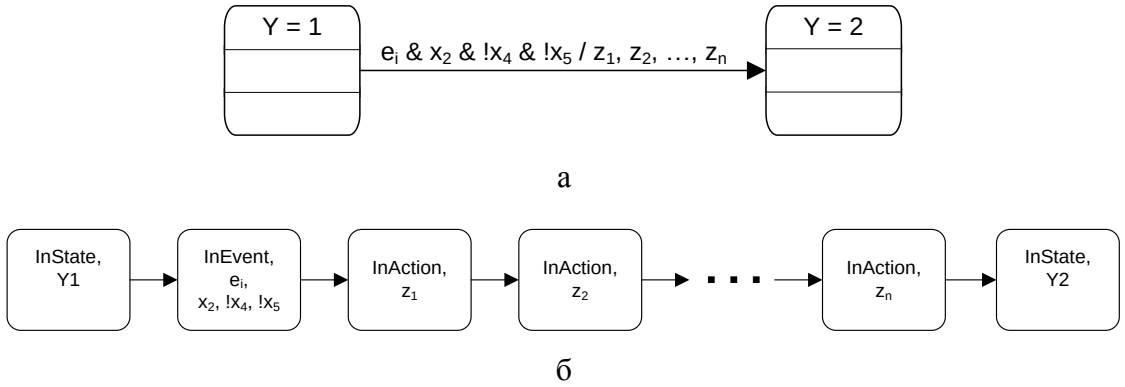


Рис. 4.15. Переход между состояниями
до преобразования по методу редукции (а) и после него (б)

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

В конце работы алгоритма так же, как и в предыдущем методе, для внешних вершин полученной модели добавим в отношение *Label* пометку с атомарным предложением, соответствующим имени обрабатываемого автомата.

Алгоритм построения модели Крипке данным методом демонстрируется на примере автомата, управляющего дверьми лифта

(рис. 4.3). На рис. 4.16 показана модель Крипке, построенная с помощью редукции графа переходов по этому автомату.

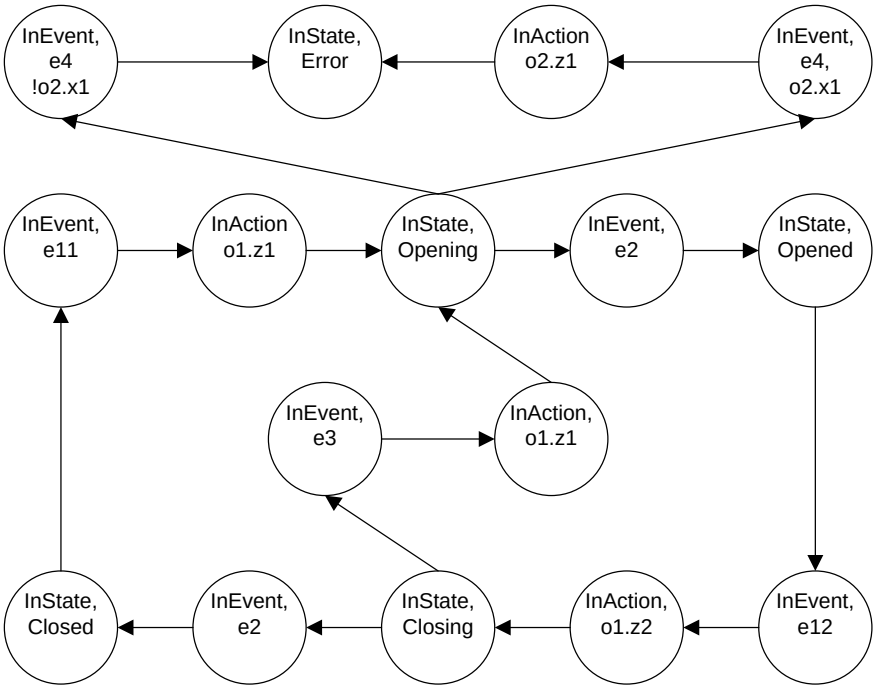


Рис. 4.16. Редукция графа переходов для автомата, управляющего дверьми лифта

Еще один пример построения модели Крипке рассмотренным методом демонстрируется на автомате *ARemote*, эмулирующем универсальный инфракрасный пульт для бытовой техники [86]. Его схема связей и граф переходов изображены на рис. 4.17 и 4.18.

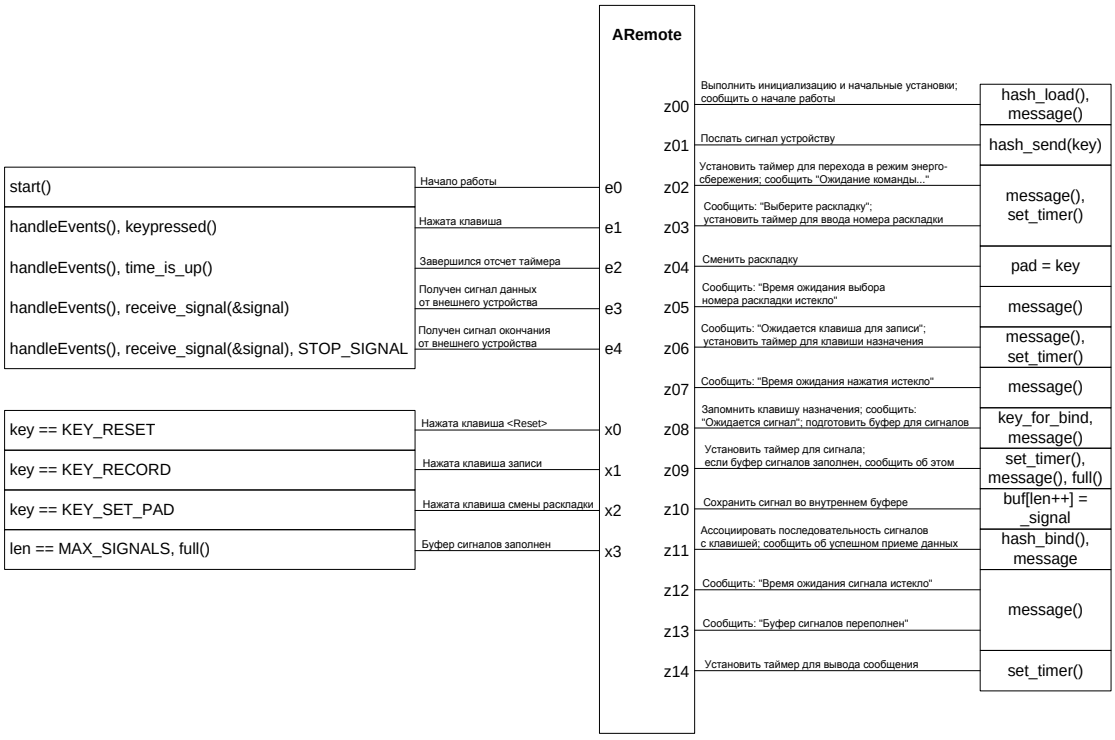


Рис. 4.17. Схема связей автомата *ARemote*

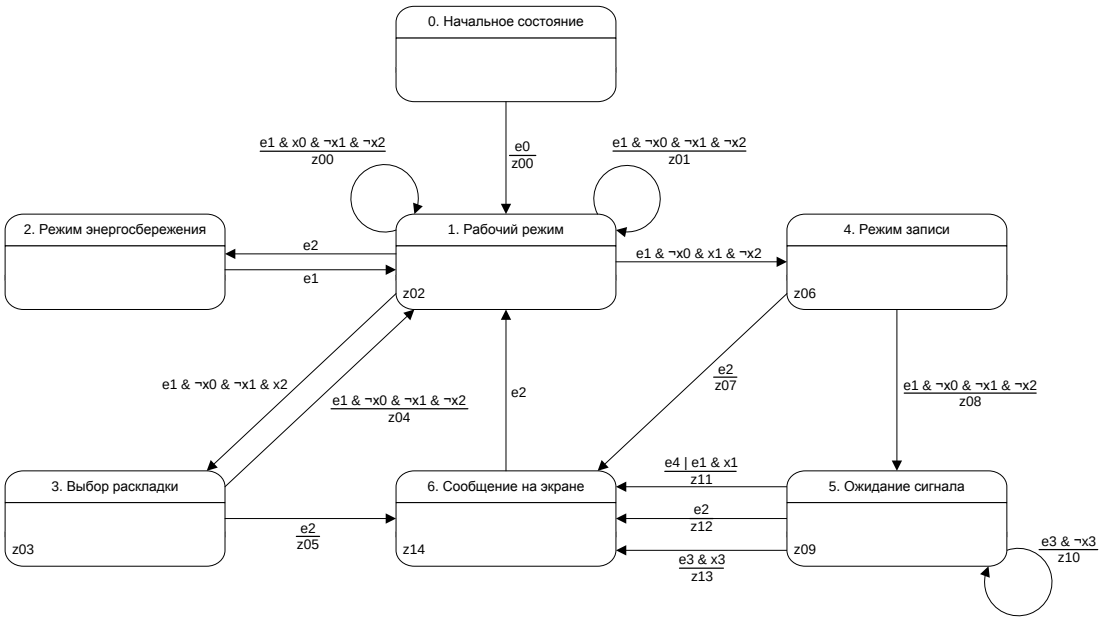


Рис. 4.18. Граф переходов автомата *ARemote*

На рис. 4.19 показана модель Крипке, построенная с помощью редукции графа переходов по автомату *ARemote*. Вершины, построенные из состояний, обозначены серым цветом.

Если переходы каких-либо автоматов зависят, помимо входных воздействий, от состояний других автоматов, то эти переходы следует добавлять в модель Крипке только в том случае, если выполняется указанная выше зависимость на состояния внешних автоматов. Эта проверка должна выполняться при обработке автоматов, вложенных в состояние. Например, если автомат `A2` вложен в `A1` и в нем некоторый переход срабатывает, только если автомат `A1` находится в состоянии 3, то этот переход следует копировать из внутренней модели во внешнюю только при обработке состояния 3 внешней модели. При обработке других состояний внешней модели этот переход следует перенаправлять в состояние, из которого он исходит. При этом в таком состоянии будет образовываться петля.

Пример построения модели Крипке по системе взаимодействующих автоматов приведен на рис. 4.20 и 4.21. Вершины модели Крипке пронумерованы в порядке, в котором они генерируются описанным алгоритмом. Пунктирами обозначены переходы, которые при копировании были заменены петлями – в действительности этих переходов в модели нет. Серым цветом отмечены стартовая и терминальная вершины.

Размер получившейся модели в методе редуцированного графа переходов линеен по отношению к размеру автомата.

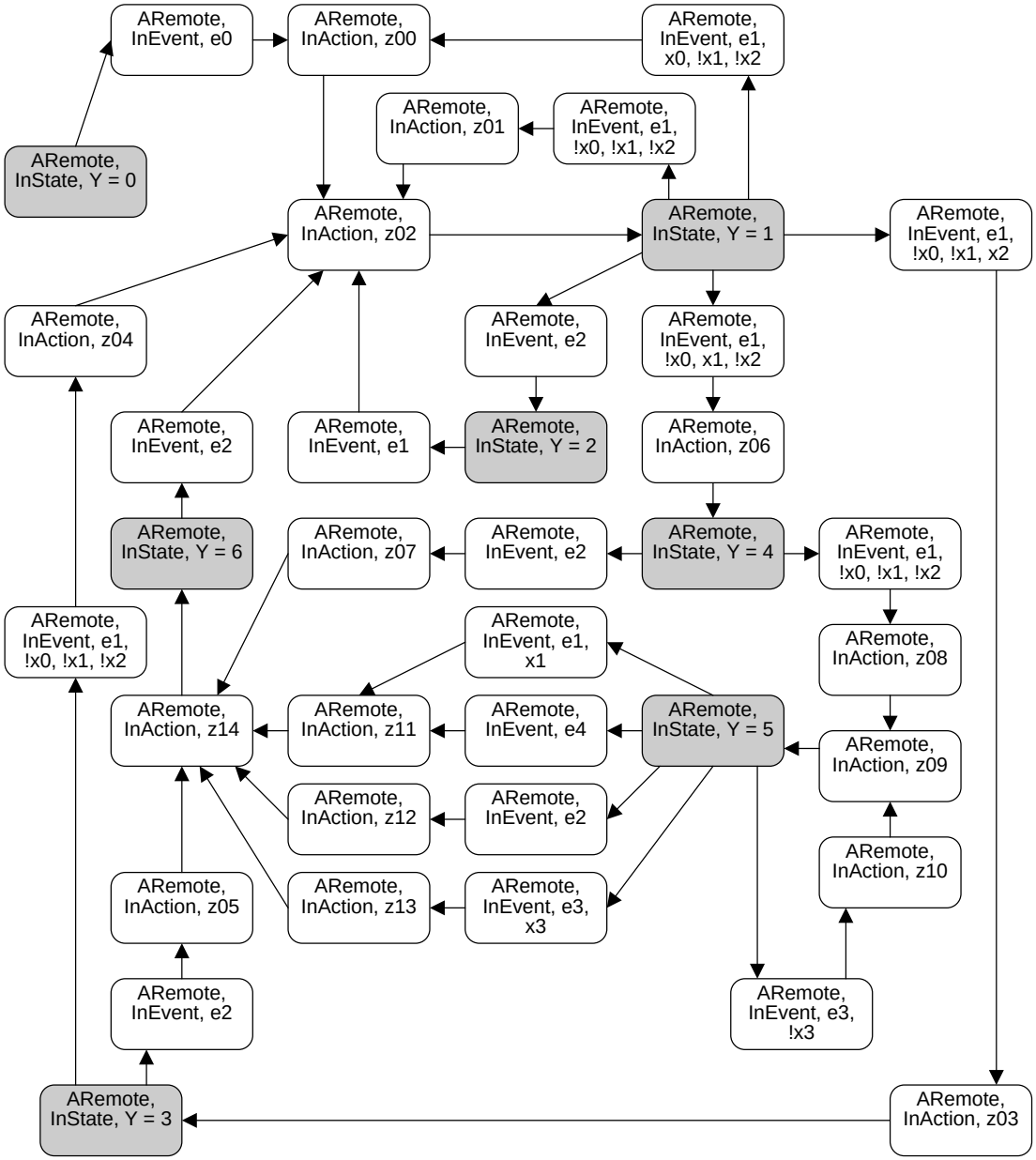


Рис. 4.19. Редукция графа переходов для автомата *ARemote*

Проверка CTL-формулы

Разберем построение и интерпретацию CTL-формул для редуцированных моделей.

CTL-семантика в этом методе будет немного отличаться от общепринятой: перед тем, как выполнять верификацию CTL-формулы, ее следует привести к определенному («каноническому») виду. Вначале в ней необходимо удалить все парные отрицания (путем замены подформул вида $\neg\neg f$ на f). После этого все входные воздействия, которые присутствуют в формуле без отрицания, следует предварить двумя отрицаниями: одно из них синтаксическое, другое

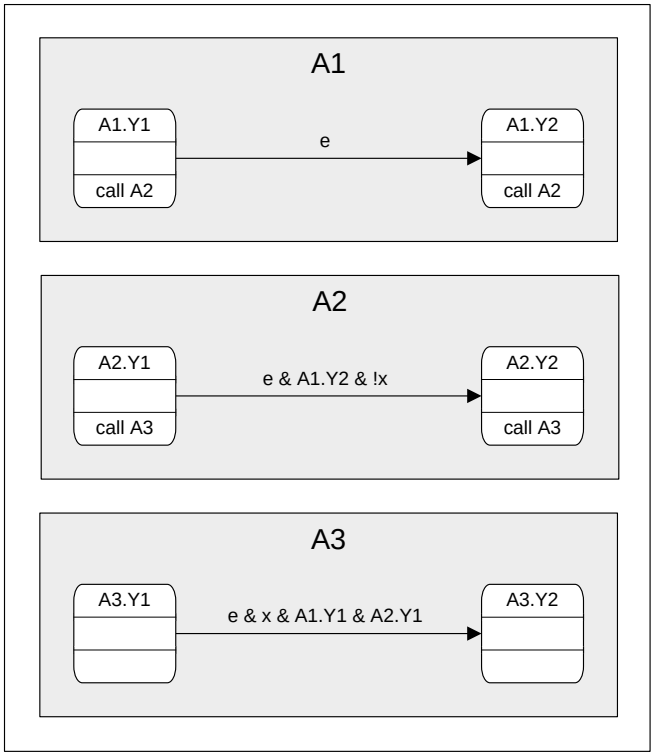


Рис. 4.20. Система автоматов, взаимодействующих по вложенности, с условиями на состояния внешних автоматов

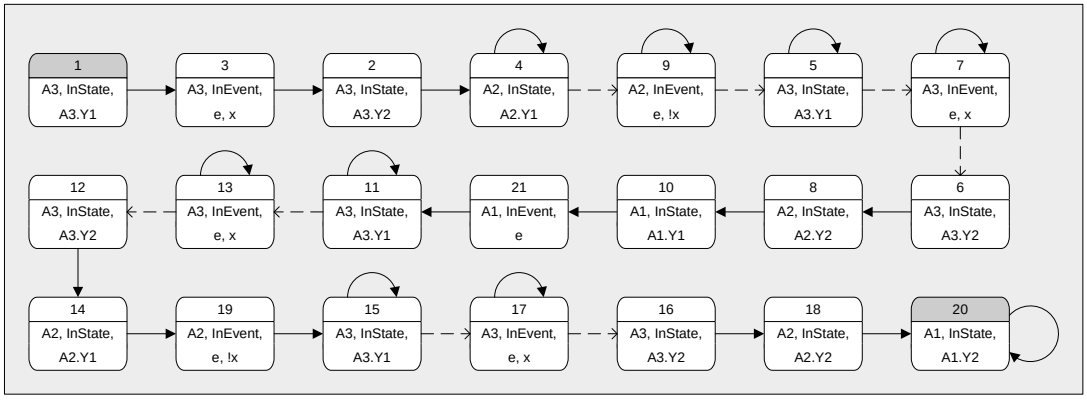


Рис. 4.21. Модель Крипке для рассматриваемой системы

логическое (это значит, что необходимо заменить литералы вида x_i на формулы $\neg!x_i$). Только после этих модификаций результирующую формулу можно верифицировать методами, предназначенными для языка CTL. Причина такого обращения с литералами состоит в следующем: требуется обеспечить, чтобы любая ссылка на несущественную переменную, которая упомянута в CTL-формуле, давала истинный результат (несущественными переменными на данном переходе называются те входные переменные исходного автомата, значение которых не проверяется на этом переходе).

Рассмотрим пример для модели банкомата. Проверим CTL-формулу: $e14 \rightarrow \neg\mathbf{E}[\neg o3.z0\ \mathbf{U}\ y10]$. Она означает: если произошло событие $e14$, то невозможно попасть в состояние 10, минуя вершину $o3.z0$. Эта формула верна во всех вершинах модели. Все пути из вершины $e14$ в состояние 10 дважды «проходят» через автомат *AServer*: в первый раз – попав в состояние 3, а второй – попав в состояние 9 автомата *AClient*. Пример такого пути, который не проходит дважды через одно состояние одного и того же экземпляра автомата, выделен на рис. 4.22. Вершина $e14$, являющаяся начальной для данного пути, выделена на рисунке жирным прямоугольником.

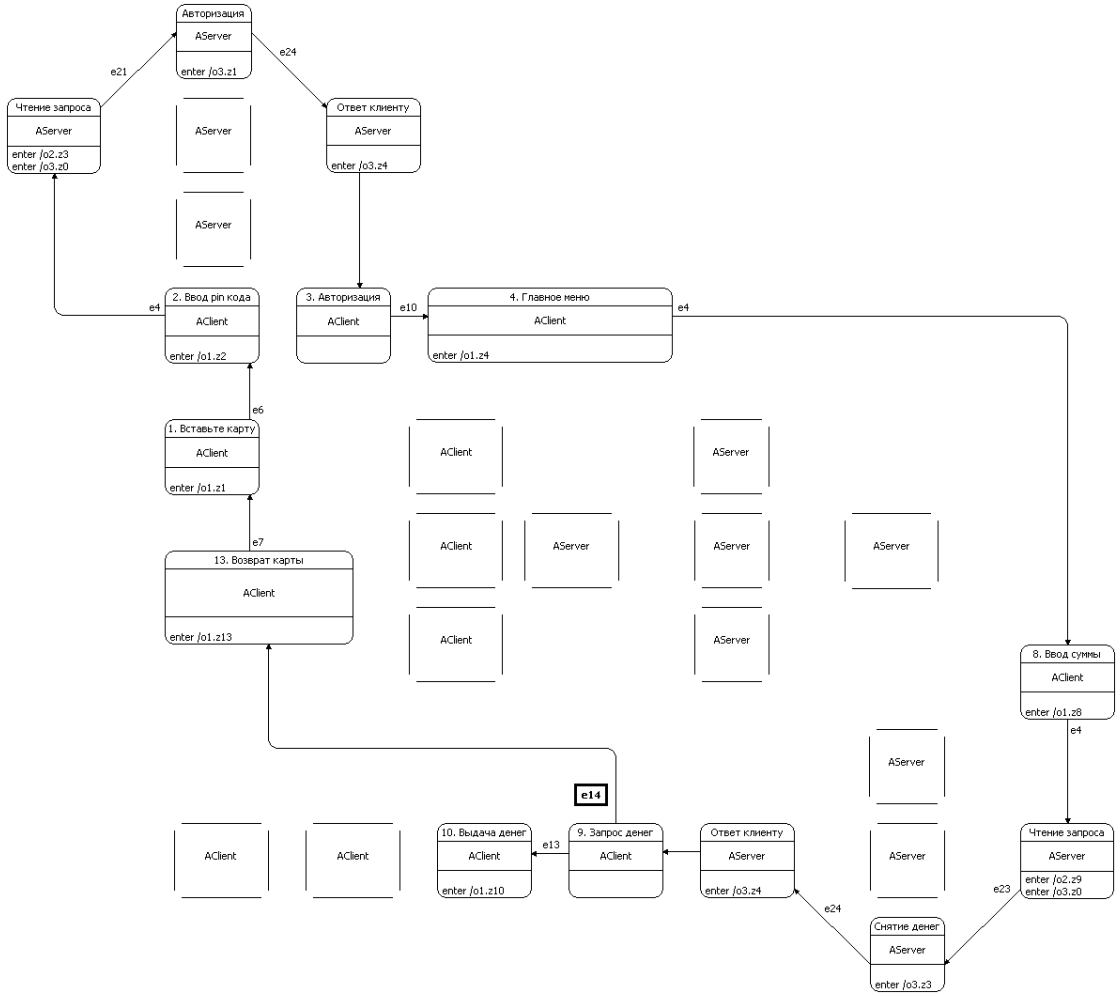


Рис. 4.22. Пример пути в модели Крипке для банкомата

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

Путь в модели Крипке для банкомата, отображенный на рис. 4.22, можно показать и в терминах исходной системы автоматов (рис. 4.23, 4.24). Стартовая вершина $e14$ выделена прямоугольником.

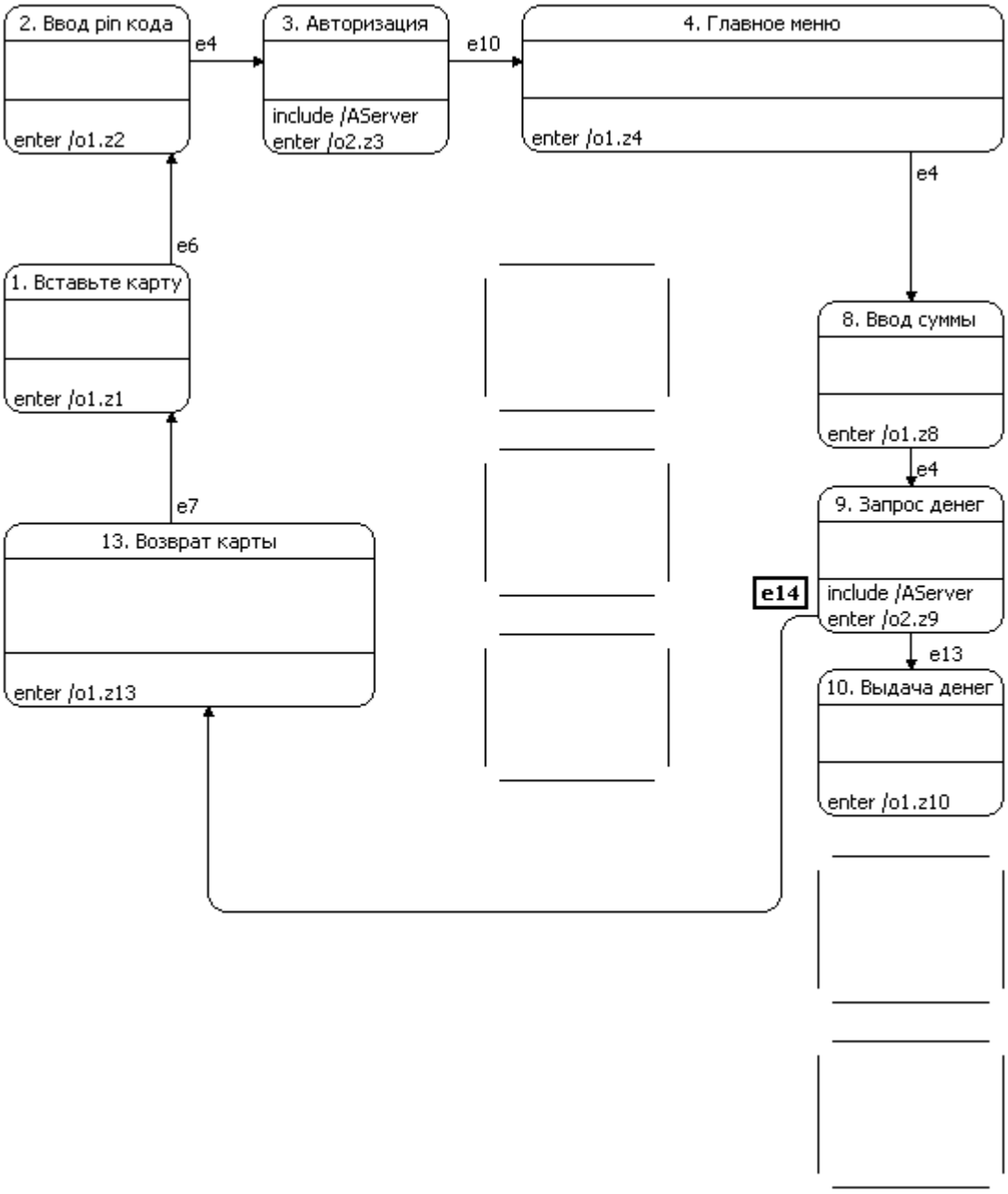


Рис. 4.23. Участок пути в автомате *AClient*

Дополнительные методы выделения атомарных состояний описаны также в работах [77, 84, 87].

Верифицируемые свойства

В качестве предикатов для верификатора *CTL Verifier* можно использовать следующие условия:

- определенный автомат находится в определенном состоянии;
- обрабатывается определенное событие;
- истинна или ложна определенная входная переменная;
- вызывается определенное выходное воздействие.

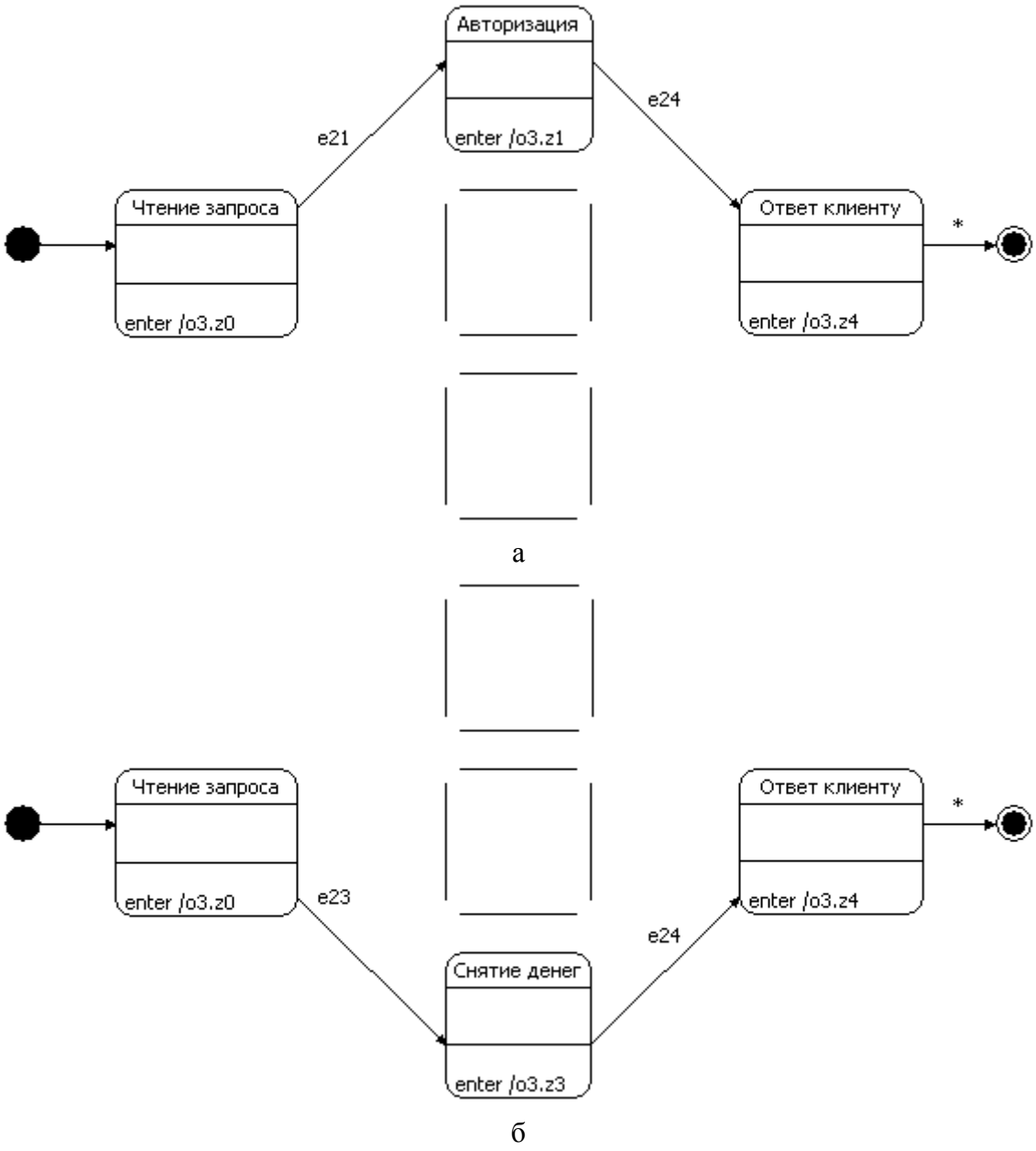


Рис. 4.24. Участок пути в автомате *AServer*
при первом прохождении (а) и при втором (б)

Инструмент *CTL Verifier* позволяет верифицировать формулы логики CTL, использующие только следующие темпоральные операторы: *EX*, *EG*, *EU*. Другие темпоральные операторы логики CTL могут быть выражены через эти по следующим правилам:

- *AX g = !EX !g*;
- *EF g = 1 EU g*;
- *AF g = !EG !g*;
- *AG g = !EF !g = !(1 EU g)*;
- *f AU g = !((!g EU !(f || g)) || EG !g)*.

Преобразование контрпримера

После моделирования и верификации требуется обработать результаты проверки модели. Программы, написанные с помощью традиционных методов, имеют достаточно сложные модели, и проводить анализ путей прямо на модели Крипке неэффективно.

При интерактивном моделировании совместно с исполнением и визуализацией автомата [58, 75, 76] процесс представления путей в модели Крипке путями в автомате можно автоматизировать.

После того как отработала программа-верификатор, требуется определить выполнимость формул, которые образуют спецификацию, на заданных участках автомата. Среди этих участков могут быть состояния, события и выходные воздействия.

Сценарий для любой подформулы спецификации представляет собой путь в модели Крипке, иллюстрирующий справедливость или ошибочность данной подформулы. Задача состоит в том, чтобы сценарий, представленный программой для модели Крипке, был представлен в исходном автомате.

Для описанного в данном разделе метода операция переноса путей из модели Крипке в автомат выполняется однозначно. Действительно, состояния модели, содержащие атомарное предложение $Y = \ldots$ или вспомогательное атомарное предложение *InState*, однозначно преобразуются в соответствующие им состояния автомата. Путь между любыми двумя соседними состояниями всегда представляет собой «змейку» из события и выходных воздействий. Любая из этих промежуточных вершин однозначно определяет то главное состояние автомата, из которого эта «змейка» исходит. Из атомарных предложений, которыми помечены состояния «змейки», однозначно восстанавливаются события. Значения существенных входных переменных (тех, которые записаны на переходе) и список несущественных определяется оттуда же (в методе редукции). Последовательным проходом по полученному пути восстанавливается информация о выполнимости литералов, соответствующих выходным воздействиям, очередности этих литералов и о том, как попасть в данное состояние.

Описание инструментального средства

CTL Verifier принимает на вход систему автоматов, описанную в текстовом формате. Этот формат достаточно простой, однако разработан специально для этого верификатора и не используется другими программами.

Во входном файле, кроме описания системы автоматов, приводится также формула для верификации. Она записывается индукцией по построению. Это значит, что все ее подформулы тоже должны быть записаны. Например, формула «$EG\ !e_1$» записывается, как указано в листинге 4.15.

Листинг 4.15. Запись формулы $EG\ !e_1$

```
[Properties]
f1 = e1
f2 = !f1
f3 = $EG f2
```

Верификация происходит путем запуска одной команды:

```
CTLVerif.exe <входной файл>
           [ <выходной файл> [<выходная папка>] ]
```

Программа *CTL Verifier* в настоящее время работает только в операционных системах типа *Windows*.

В результате верификации выводится следующая информация:

- список предикатов, используемых в построенной модели Крипке;
- модель Крипке, полученная в результате преобразования исходной автоматной системы. Все элементарные состояния пронумерованы, определены переходы между ними и списки предикатов, выполняющихся в каждом состоянии;
- для каждой формулы список номеров состояний модели Крипке, в которых она выполняется. Если формула на «верхнем уровне» содержит темпоральный оператор и доказывается циклом состояний, то выводится цепочка состояний, которая приводит к выполнению формулы. Пример:
  1 34 35 (3 38 39 5 109 110 8 91 92 15 85 86 20 82 83);
- если указана выходная папка, то для каждой верифицируемой формулы создается файл с контрпримером в терминах исходной автоматной системы.

Верификация модели банкомата

Аналогично верификатору *FSM Verifier*, верификатор *CTL Verifier* принимает на вход модель в формате, отличном от формата *UniMod*. Поэтому для верификатора *CTL Verifier* также необходимо преобразовать исходную *UniMod*-модель банкомата в подходящий формат. Для поддержки формата *UniMod* реализован алгоритм, который конвертирует модель инструментального средства *UniMod* в модель верификатора *CTL Verifier*. В процессе конвертации производятся некоторые изменения в автоматной системе, такие как

изменение имен состояний, соединение всех терминальных состояний автомата в одно терминальное состояние и т. д.

В результате конвертации модели банкомата был создан файл *Bankomat.dat*. Состояние автомата *AClient* «*10. Выдача денег*» получило имя *s10*. Секция с верифицируемым свойством Σ файла *Bankomat.dat* приведена в листинге 4.16.

Листинг 4.16. Запись свойства Σ банкомата и его подформул

```
[Properties]
; !(!e10 EU s10)
f1 = e10
f2 = !f1
f3 = s10
f4 = f2 $EU f3
f5 = !f4
```

Запускаем верификатор:

```
CTLVerif.exe Bankomat.dat out.txt out
```

Верификатор создает файл *out.txt* с информацией о модели Крипке. Кроме того, создается папка *out*, содержащая файлы *f1*, *f2*, *f3*, *f4* и *f5*. Каждый файл содержит отчет о состояниях автоматной системы, в которых выполняются соответствующие формулы. Сразу обратимся к файлу *f5*. Его содержание отражено в листинге 4.17 (приведены лишь первые 4 строки из 82).

Листинг 4.17. Результат верификации свойства Σ банкомата

```
$   1: AClient InState s0
   28: AClient InAction o1.z1
   29: AClient InState s12
   30: AClient InAction o1.z7
```

Первая же строка сообщает о том, что интересующая формула *f5* выполняется в начальном состоянии *s0* головного автомата *AClient* модели банкомата. Состояние *s0* преобразованной в формат верификатора *CTL Verifier* модели соответствует начальному состоянию *s1* исходной автоматной системы в формате *UniMod*. Таким образом, начальное состояние указанной системы удовлетворяет верифицируемой формуле. Из этого следует, что исходная автоматная система банкомата удовлетворяет тому свойству, что пользователь не получит деньги до тех пор, пока не введет правильный *pin*-код.

Верифицируем теперь свойство Ω. Поскольку *CTL Verifier* не работает с оператором *AF*, требуется преобразовать формулу, используя правила преобразования формул CTL: *AF s10 = !EG !s10*. В результате

верифицируемое свойство (листинг 4.18) записывается в файл *Bankomat.dat*.

Листинг 4.18. Свойство Ω банкомата и его подформулы

```
[Properties]
; AF s10 = ! EG !s10
f1 = s10
f2 = !f1
f3 = $EG f2
f4 = !f3
```

Запускаем верификатор:

```
CTLVerif.exe Bankomat.dat out.txt out
```

В результате формуле *f4* удовлетворяют лишь 3 состояния, среди которых нет начального (листинг 4.19).

Листинг 4.19. Результат верификации свойства Ω банкомата

```
     4: AClient InAction o1.z10
     5: AClient InState s10
   106: AClient e13 InEvent
```

Если начальное состояние не удовлетворяет этой формуле, то оно удовлетворяет ее отрицанию. Таким образом, если верифицировать отрицание рассматриваемого свойства, то верификатор должен привести доказательство этого отрицания – путь в модели Крипке. В данном случае отрицание верифицировалось формулой *f3*. Поэтому сразу можно увидеть результаты. В листинге 4.20 приведен первый из 113 контрпримеров для формулы *f3*.

Листинг 4.20. Контрпример для свойства Ω банкомата

```
[1]
$   1: AClient InState s0
   89: * AClient InEvent
   28: AClient InAction o1.z1
   29: AClient InState s12
  108: AClient e0 InEvent
  109: AClient InAction o1.z0
Cycle:
%  80: AClient InState s5
```

В листинге 4.20 отражен бесконечный путь с началом в стартовом состоянии и циклом. Это и есть контрпример для верифицируемой формулы, он должен показывать бесконечный путь в автоматной системе, который не проходит через состояние «*Выдача денег*». Для того чтобы разобраться в этом контрпримере, следует провести соответствие между именами состояний в верифицируемой автоматной системе и в исходной системе (в формате *UniMod*). Результат представлен в листинге 4.21.

Листинг 4.21. Преобразованный контрпример для свойства Ω банкомата

```
$   1: AClient InState s1
   89: * AClient InEvent
   28: AClient InAction o1.z1
   29: AClient InState "1. Вставьте карту"
  108: AClient e0 InEvent
  109: AClient InAction o1.z0
Cycle:
%  80: AClient InState s2
```

Этот контрпример соответствует следующей ситуации. Банкомат начинает работу в своем начальном состоянии, включается и предлагает вставить карту. Далее нажимается кнопка *Выключить* (событие *e0*), и банкомат выключается. Попав в конечное состояние *s2*, он там остается. Таким образом, пользователь никогда не получит денег.

4.5.2. *Automata Verificator*

Общее описание

Верификатор автоматных систем *Automata Verificator* [88] не использует каких-либо существующих верификаторов. В нем реализован алгоритм двойного поиска в глубину, позволяющий проверять формулы, описанные с помощью автомата Бюхи, а значит, и формулы темпоральной логики LTL.

Особенностью верификатора является то, что алгоритм верификации реализован с возможностью многопоточного исполнения. Таким образом, как было показано в работе [88], при реализации на многоядерных процессорах алгоритм дает лучшие результаты по сравнению с однопоточным вариантом.

Automata Verificator работает с автоматными моделями, описанными при помощи инструментального средства *UniMod*.

Выделение атомарных состояний

В рассматриваемом верификаторе атомарное состояние определяется набором текущих состояний автоматов, входящих в автоматную систему. Следовательно, дробление переходов на элементарные состояния не производится.

Верифицируемые свойства

Список предикатов, поддерживаемых верификатором *Automata Verificator*, похож на предикаты, используемые для верификатора *UniMod.Verifier*. Поддерживаются следующие предикаты:

- *wasEvent*;
- *isInState*;
- *wasInState*;
- *cameToFinalState*;
- *wasAction*;
- *wasFirstAction*.

Смысл перечисленных предикатов такой же, как в верификаторе *UniMod.Verifier*. Удобно то, что в качестве аргументов предикатов можно использовать имена состояний, событий и действий исходной автоматной системы, даже если эти имена содержат пробелы и другие специальные символы.

Кроме того, в верификаторе *Automata Verificator* можно достаточно просто создавать новые предикаты. Для этого следует разработать класс на языке *Java*, в котором создать метод с аннотацией «*@Predicate*». Например, можно создать предикат, позволяющий верифицировать свойства типа «действие $o_1.z_1$ выполняется через одно после $o_1.z_2$».

В формулах используется темпоральная логика LTL.

Преобразование контрпримера

В рассматриваемом верификаторе не используются другие верификаторы, и алгоритм верификации работает напрямую с автоматной системой. Поэтому контрпример сразу выражен в терминах исходной автоматной системы и нет необходимости его преобразовывать.

Описание инструментального средства

Верификатор *Automata Verificator* изначально был разработан в виде *Java*-классов и не представлен в виде готовой библиотеки, которую можно было бы запускать из командной строки. Поэтому был создан класс, позволяющий запускать верификацию из командной строки, и собрана библиотека со всеми классами верификатора *Automata Verificator*.

Рассматриваемый верификатор принимает на вход автоматные системы в формате *XML*, сгенерированные при помощи инструментального средства *UniMod*. Формат запуска процесса верификации следующий:

```
java –jar verifier.jar A.xml A1 "F(wasEvent(p.e1))"
```

Здесь

- *java* – команда запуска *Java*-машины (должна быть установлена программа *Java Runtime Environment* не ниже 6-й версии);
- *A.xml* – путь к файлу с описанием автоматной системы в формате *UniMod*;
- *A1* – название корневого автомата;
- *F(wasEvent(p.e1))* – верифицируемая формула (в данном примере она означает следующее: «когда-нибудь произойдет событие *e1*, генерируемое источником событий *p*»).

В результате работы такой команды сначала в консоль выводится автомат Бюхи, сгенерированный по формуле LTL, а затем результат верификации. В случае удачной верификации выводится сообщение «*Verification successful*». Вывод верификатора для случая, когда найден контрпример, приведен в листинге 4.22.

Листинг 4.22. Вывод контрпримера верификатором *Automata Verificator*

```
LTL: F(isInState(AClient, AClient["10. Выдача денег"]))
initial 0
BuchiNode 0
 →[!isInState(AClient, 10. Выдача денег)] 0
Accept set 0 [0]

DFS 2 stack:
→["<"13. Возврат карты", "s1">", 0, 0]→["<"1.
Вставьте карту", "s1">", 0, 0]
DFS 1 stack:
→["<"s1", "s1">", 0, 0]→["<"1. Вставьте карту",
"s1">", 0, 0]
→["<"2. Ввод pin-кода", "s1">", 0, 0]→["<"3.
Авторизация", "s1">", 0, 0]
→["<"4. Главное меню", "s1">", 0, 0]→["<"13. Возврат
карты", "s1">", 0, 0]
```

Автомат Бюхи выводится как набор состояний, а также указываются его начальное состояние (*initial*) и наборы допускающих состояний (*Accept set*). В данном примере автомат Бюхи состоит из одного состояния под номером *0*. Оно же является начальным и допускающим.

Далее записывается контрпример в виде набора состояний в стеке главного обхода в глубину (*DFS 1*) и вложенного обхода в глубину (*DFS 2*). В итоге контрпример получается путем продолжения последовательности *DFS 1* последовательностью *DFS 2*. Однако следует предварительно опустить первое состояние последовательности *DFS 2*, поскольку оно повторяет последнее состояние последовательности *DFS 1*.

Каждое состояние автоматной системы в контрпримере записывается внутри квадратных скобок. В них содержатся следующие три компоненты:

1. набор текущих состояний автоматов автоматной системы;
2. номер текущего состояния автомата Бюхи;
3. множество допускающих состояний автомата Бюхи.

Верификация модели банкомата

Верификатор *Automata Verificator* работает с автоматными системами в формате *UniMod*. Поэтому дополнительных преобразований исходной автоматной системы банкомата не требуется.

Верифицируем свойство Σ. Утверждение, что произошло событие *e10*, записывается следующим образом:

wasEvent(p3.e10)

Здесь *p3* – источник событий, генерирующий событие *e10* (рис. 4.5). Утверждение, что автомат *AClient* находится в состоянии «*10. Выдача денег*», записывается следующим образом:

isInState(AClient, AClient[\"10. Выдача денег\"])

Приведенная ниже команда выполняет верификацию свойства Σ банкомата:

```
java –jar verifier.jar Bankomat.xml AClient
"!U(!wasEvent(p3.e10), isInState(AClient, AClient[\"10.
Выдача денег\"]))"
```

Результат выполнения команды приведен в листинге 4.23.

Листинг 4.23. Результат верификации свойства Σ

```
LTL: !U(!wasEvent(p3.e10), isInState(AClient,
AClient["10. Выдача денег"]))
initial 1
BuchiNode 0
 →[true] 0
BuchiNode 1
 →[!wasEvent(e10)] 1
 →[isInState(AClient, 10. Выдача денег)] 0
Accept set 0 [0]

Verification successful
```

Как видно, верификация завершена успешно.

Верифицируем свойство Ω:

```
java –jar verifier.jar Bankomat.xml AClient
"F(isInState(AClient, AClient[\"10. Выдача денег\"]))"
```

Результат изображен в листинге 4.22.

В ходе верификации свойства Ω был найден контрпример. Сначала следует читать стек *DFS 1*, а затем стек *DFS 2*. Во всех приведенных глобальных состояниях автоматной системы изменяется лишь состояние автомата *AClient*.

В соответствии с контрпримером, автомат *AClient* начинает работу в своем начальном состоянии *s1*, затем переходит в состояние «*1. Вставьте карту*», после этого в состояние «*2. Ввод pin-кода*», и, наконец, в состояние «*3. Авторизация*». Авторизация проходит успешно, и автомат переходит в состояние «*4. Главное меню*». В этот момент пользователь нажимает на кнопку *Отмена*, так как автомат переходит в состояние «*13. Возврат карты*». После этого продолжается чтение стека *DFS 2*, которое завершает контрпример с переходом в уже посещенное состояние «*1. Вставьте карту*».

Найденный верификатором *Automata Verificator* контрпример является корректным, но отличается от контрпримеров, найденных другими верификаторами.

Заключение

Основная проблема в развитии программных систем состоит в том, что проверка правильности реализуемой программной системы чрезвычайно сложна. При разработке такой системы обычно основное время уходит не столько на написание кода, сколько на его анализ и отладку. Существует много методов проверки правильности программ, и они соответствуют различным их классам. В данной книге рассматривался класс систем, критичных в отношении безопасности, при проверке правильности которых недопустимо применение неполной индукции. Даже если убедиться в корректности основных сценариев работы системы, этого все равно будет недостаточно для признания системы надежной, так как самые глубинные и трудно обнаруживаемые ошибки в ней могут быть допущены на фазе проектирования. Поэтому книга посвящена методам проверки, которые гарантируют корректность любого поведения системы (или ее модели).

Тем не менее, для того чтобы получить стопроцентное доказательство правильности программы, необходимо затратить колоссальные усилия. Метод *формальной верификации* не применить к большим и сложным программам. Специалисты в разработке программного обеспечения и его верификации рассказывают, что они заканчивали

работу с 15-страничным отчетом о том, что программа на полстраницы работала корректно [89]. Поэтому следует соблюдать определенный баланс между ожидаемой достоверностью доказательства и тем, насколько удобно и эффективно оно может быть получено.

Наиболее эффективно этот баланс соблюдается в методе *проверки моделей* (model checking), который существует около 30 лет. Его популярность обусловливается, с одной стороны, высокой степенью автоматизации доказательства корректности и небольшой вовлеченностью разработчика в процесс верификации, а с другой – возможностью явно указать на ошибку в случае, когда разработчик имеет дело с ошибочным прототипом. Актуальность исследований в области проверки моделей подтверждается также и присуждением в 2007 г. премии Тьюринга основоположникам этого подхода – Эдмунду Кларку, Аллену Эмерсону и Джозефу Сифакису. Проверка моделей имеет поразительные достижения, прежде всего, в верификации аппаратных средств. Например, известно, что проверка моделей нашла бы ошибку в процессорах Pentium I и доказала бы верность ее исправления корпорацией Intel [90]. С тех пор Intel – одна из наиболее систематически применяющих эту технологию корпораций.

Основная проблема, которая возникает при использовании проверки моделей – это *комбинаторный взрыв* в пространстве состояний модели. Первые алгоритмы проверки моделей могли работать с числом состояний порядка 40 000 [89]. Со временем размеры систем переходов, допускающих эффективную верификацию, существенно возросли. Этого удалось добиться путем неявного представления модели. Для решения этой задачи была вначале создана технология *символьной проверки моделей*, основывающаяся на упорядоченных двоичных разрешающих диаграммах, а потом – *проверка моделей ограниченной глубины* (*bounded model checking*) [91], которая увеличила численные границы проверки моделей еще на несколько порядков. Значительные успехи были достигнуты также благодаря развитию методов *редукции частичных порядков*, методов использования *симметрии* в параллельных системах и методов *абстракции* проверяемых моделей. В настоящее время технологии проверки моделей позволяют верифицировать программы примерно из 10 000 строк кода [89].

Наиболее впечатляющим примером, демонстрирующим способности символьной проверки моделей, является верификация протокола когерентности кэш-памяти стандарта шины *IEEE Futurebus+* (стандарт *IEEE 896.1–1991*). Хотя разработка этого протокола была

начата в 1988 г., все прежние попытки обосновать его правильность основывались исключительно на неформальных рассуждениях. Летом 1992 г. исследовательская группа из университета Карнеги-Меллона [92] построила точную модель протокола на языке *SMV* и затем, применяя систему верификации *SMV*, показала, что полученная система переходов удовлетворяет формальной спецификации. Им удалось также обнаружить ряд ранее не замеченных ошибок и локализовать потенциально ошибочные участки в проекте протокола.

Особое значение верификация имеет для программных систем со сложным поведением. Для этого класса систем технологию создания и верификации следует выбирать уже на этапе проектирования.

Удачным выбором для таких задач является технология *автоматного программирования* [2, 51]. В первую очередь, автоматное программирование дает возможность успешно выполнить декомпозицию управляющей логики и вычислений на этапе создания программной системы. Оно позволяет использовать наглядную графическую нотацию для описания сложного поведения, сделав тем самым систему менее подверженной ошибкам. С другой стороны, при применении model checking к автоматным программам построение модели программы по ее спецификации значительно упрощается по сравнению с традиционными подходами к программированию. При использовании автоматного подхода модель программы, пригодная к верификации, строится уже на этапе проектирования. Наконец, автоматное программирование, как уже отмечалось, позволяет декомпозировать процесс верификации программы на верификацию ее поведенческой (управляющей) модели, которая выполняется методом model checking, и независимую верификацию атомарных вычислительных воздействий. Если система успешно спроектирована на основе автоматного подхода, и входные и выходные воздействия представляют собой небольшие относительно простые участки кода, то их можно проверить и формальной верификацией, оперирующей зависимостями выходных данных алгоритма от его входных данных. Выходные воздействия в автоматных программах могут также быть проверены на основе подхода, изложенного в работе [59].

Примеры использования инструментальных средств верификации более подробно рассмотрены в книгах [93, 94]. Вопросы верификации программ общего вида рассматриваются в книгах [17, 18].

Первые работы по верификации автоматных программ появились в 2006 г. – сначала [87], а затем – [70]. Исследования по верификации автоматных программ продолжаются в Санкт-Петербургском государственном университете информационных технологий,

механики и оптики [53, 63, 77] и в Ярославском государственном университете им. П. Г. Демидова [95, 96]. По этой тематике начинают защищаться диссертации [97, 98], в том числе, докторские [99]. При этом один из выводов последней работы сформулирован следующим образом: «автоматные программы являются исключительно удобным объектом для верификации методом проверки модели».

Авторы предполагают, что данная книга привлечет внимание к автоматному программированию, как подходу, ориентированному на верификацию.

Список источников

1. *Katoen J.-P.* Concepts, Algorithms, and Tools for Model Checking. Lehrstuhl für Informatik VII, Friedrich-Alexander Universität Erlangen-Nürnberg. Lecture Notes of the Course “Mechanised Validation of Parallel Systems” (course number 10359). 1998/1999. http://fmt.isti.cnr.it/~gnesi/matdid/katoen.pdf
2. *Шалыто А. А.* Switch-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. http://is.ifmo.ru/books/switch/1
3. *Liggesmeyer P., Rothfelder M., Rettelbach M., Ackermann T.* Qualitätssicherung Software-basierter technischer Systeme – Problembereiche und Lösungsansätze // Informatik Spektrum. 21: 249–258, 1998.
4. *Baier C., Katoen J.-P.* Principles of Model Checking. The MIT Press, 2008. http://is.ifmo.ru/books/_principles_of_model_checking.pdf
5. *Harel D., Pnueli A.* On the Development of Reactive Systems // Logics and Models of Concurrent Systems. V. F-13 of NATO ASI Series. NY, Springer-Verlag, 1985. http://www.wisdom.weizmann.ac.il/~dharel/SCANNED.PAPERS/ReactiveSystems.pdf
6. *Синицын С. В., Налютин Н. Ю.* Верификация программного обеспечения. М.: БИНОМ, 2008.
7. *ISO/ITU-T.* Formal Methods in Conformance Testing. Draft International Standard, 1996.
8. *Бейзер Б.* Тестирование черного ящика. Технологии функционального тестирования программного обеспечения и систем. СПб.: Питер, 2004.
9. *Бек К.* Экстремальное программирование: разработка через тестирование. СПб.: Питер, 2003.
10. *Freeman S., Pryce N., Mackinnon T., Walnes J.* Mock Roles, not Objects. http://www.jmock.org/oopsla2004.pdf
11. *Umrigar Z., Pitchumani V.* Formal verification of a real-time hardware design / Proceedings of the 20th Design Automation Conference, 1983. http://portal.acm.org/ft_gateway.cfm?id=800667&type=pdf&CFID=112534228&CFTOKEN=12780503
12. *Гуц А. К.* Математическая логика и теория алгоритмов. Омск: Наследие, Диалог-Сибирь, 2003.

13. *Hoare C. A. R.* An axiomatic basis for computer programming // Communications of the ACM. 1969/12, pp. 576–583. http://se.ethz.ch/teaching/ss2005/0250/readings/Axiomatic_Basis.pdf

14. *Owicki S., Gries D.* An axiomatic proof technique for parallel programs // Acta Informatica. 1976/6, pp. 319–340. http://www.springerlink.com/content/x12541v1q15570n2/

15. *Pnueli A.* The temporal logic of programs / 18th IEEE Symposium on Foundations of Computer Science. 1977, pp. 46–57. http://www.inf.ethz.ch/personal/kroening/classes/fv/f2007/readings/focs77.pdf

16. *Тейз А., Грибомон П., Юлен Г., Пирот А., Ролан Д., Снайерс Д., Воклер М., Гоше П., Вольпер П., Грегуар Э., Дельсарт Ф.* Логический подход к искусственному интеллекту: от модальной логики к логике баз данных. М.: Мир, 1998.

17. *Кларк Э., Грамберг О., Пелед Д.* Верификация моделей программ: Model Checking. М.: МЦНМО, 2002.

18. *Карпов Ю. Г.* Model Checking: верификация параллельных и распределенных программных систем. СПб.: БХВ-Петербург, 2010.

19. *West C. H.* Applications and limitations of automated protocol validation / 2nd Symposium on Protocol Specification, Testing and Verification. 1982, pp. 361–371.

20. *Clarke E. M., Emerson E. A.* Synthesis of synchronization skeletons for branching time logic // Logic of Programs. LNCS 131. 1981, pp. 52–71. http://www.springerlink.com/content/w1778u28166t2677/

21. *Apt K. R., Kozen D. C.* Limits for the automatic verification of finite-state concurrent systems // Information Processing Letters. 1986/22, pp. 307–309.

22. *Конев Б. Ю.* Введение в моделирование и верификацию аппаратных и программных систем. http://logic.pdmi.ras.ru/~kulikov/verification/10.pdf

23. *Lichtenstein O., Pnueli A., Zuck L.* The glory of the past // Logics of Programs. LNCS 193. 1985, pp. 196–218. http://www.springerlink.com/content/7681m36026888082/

24. *Смелянский Р. Л.* Применение темпоральной логики для спецификации поведения программных систем // Программирование. 1993. № 1, с. 3–28.

25. *Sistla A. P., Clarke E. M.* The complexity of propositional linear temporal logics // Journal of the ACM. 32(3). 1985, pp. 733–749.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.94.178&rep=rep1&type=pdf

26. *Clarke E. M., Draghicescu I. A.* Expressibility results for linear time and branching time logics // Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency. LNCS 354. 1988, pp. 428–437.
http://www.springerlink.com/content/5n2702u432119wx8/

27. *Kripke S. A.* Semantical considerations on modal logic // Acta Philosophica Fennica 16: 83–94, 1963.
http://condor.wesleyan.edu/courses/2007s/phil390/01/e-texts/Kripke/Kripke,%20Semantical%20Considerations%20on%20Modal%20Logic.pdf

28. *Emerson E. A., Halpern J. Y.* «Sometimes» and «not never» revisited: on branching versus linear time temporal logic // Journal of the ACM. 33(1). 1986, pp. 151–178.
http://www.cs.cmu.edu/~emc/15-820A/reading/p127-emerson.pdf

29. *Bryant R.* Graph-based algorithms for boolean function manipulation // IEEE Transactions on Computers. C-35. 1986/8, pp. 677–691.
http://www.cs.cmu.edu/~bryant/pubdir/ieeetc86.pdf

30. *Миронов А. М.* Верификация программ методом Model Checking.
http://intsys.msu.ru/staff/mironov/modelchk.pdf

31. *Миронов А. М., Жуков Д. Ю.* Математическая модель и методы верификации программных систем // Интеллектуальные системы. Т. 9. 2005. Вып. 1–4, с. 209–252.
http://www.intsys.msu.ru/magazine/archive/v9(1-4)/mironov-209-252.pdf

32. *Wegener I.* Branching Programs and Binary Decision Diagrams. SIAM monographs on discrete mathematics and applications, 2000.

33. *Clarke E. M., Grumberg O., Long D.* Verification tools for finite-state concurrent systems // A Decade of Concurrency–Reflections and Perspectives. LNCS 803. 1993, pp. 124–175.
http://www-2.cs.cmu.edu/~modelcheck/ed-papers/VTfFSCS.pdf

34. *McMillan K. L.* Symbolic Model Checking. Kluwer Academic Publishers, 1993.
http://cadence.com/cadence/cadence_labs/Documents/mcmillan_CMU_1992_Symbolic.pdf

35. *Clarke E. M., Emerson E. A., Sistla A. P.* Automatic verification of finite-state concurrent systems using temporal logic specifications // ACM Transactions on Programming Languages and Systems. 8(2). 1986, pp. 244–263. http://www.cs.cmu.edu/~modelcheck/ed-papers/AVoFSCSU.pdf

36. *Kropf T.* Hardware Verifikation. Habilitation thesis. University of Karlsruhe, 1997.

37. *Tarjan R.* Depth-first search and linear graph algorithms // SIAM Journal on Computing. Vol. 1 (1972). No. 2, pp. 146–160. http://rjlipton.files.wordpress.com/2009/10/dfs1971.pdf

38. *Eppstein D.* Design and Analysis of Algorithms. Lecture notes for 1996. http://www.ics.uci.edu/~eppstein/161/960220.html

39. *Alur R., Courcoubetis C., Dill D.* Model-checking in dense real-time // Information and Computation. 104: 2–34, 1993. http://www.cis.upenn.edu/~alur/Lics90D.ps

40. *Alur R., Henzinger T. A.* Real-time logics: Complexity and expressiveness // Information and Computation. 104: 35–77, 1993. http://www.cis.upenn.edu/~alur/Lics90H.ps

41. *Alur R., Henzinger T. A.* Back to the future: towards a theory of timed regular languages / IEEE Symp. on Foundations of Computer Science. 1992, pp. 177–186. http://www.cis.upenn.edu/~alur/Focs92.ps

42. *Yovine S.* Model checking timed automata // Embedded Systems. LNCS 1494, 1998. http://www-verimag.imag.fr/~yovine/articles/embedded98.ps.gz

43. *Petri C. A.* Kommunikation mit Automaten. Ph. D. Thesis. University of Bonn, 1962.

44. *Esparza J., Nielsen M.* Decidability issues for Petri nets – a survey // Bulletin of the EATCS, 1994. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.2.3965

45. *Boucheneb H., Hadjidj R.* Model checking of time Petri nets. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.100.6973&rep=rep1&type=pdf

46. *Holzmann G. J.* The Model Checker SPIN // IEEE Transactions on software engineering. 1997, V. 23, I. 5. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.134.7596&rep=rep1&type=pdf

47. *Dijkstra E. W.* Guarded commands, non-determinacy and formal derivation of programs // CACM. 18(8), 1975. http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD418.PDF

48. *Dolev D., Klawe M., Rodeh M.* An O(*n* log *n*) unidirectional distributed algorithm for extrema finding in a circle // Journal of Algorithms. 1982/3, pp. 245–260.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.129.7495&rep=rep1&type=pdf

49. *McMillan K. L.* The SMV System. Technical Report CS-92-131. Carnegie-Mellon University, 1992.
http://www.comp.nus.edu.sg/~cs3234/smvmanual.pdf

50. *Chan W., Anderson R. J., Beame P., Burns S., Modugno F., Notkin D., Reese J. D.* Model checking large software specifications // IEEE Transactions on Software Engineering. 24(7). 1998, pp. 498–519.
http://www.cs.washington.edu/homes/beame/papers/fse.pdf

51. *Поликарпова Н. И., Шалыто А. А.* Автоматное программирование. СПб.: Питер, 2010. http://is.ifmo.ru/books/_book.pdf

52. *Хопкрофт Д., Мотвани Р., Ульман Д.* Введение в теорию автоматов, языков и вычислений. М.: Вильямс, 2008.

53. Отчет по контракту о верификации автоматных программ. Этапы 1, 2. 2007.
http://is.ifmo.ru/verification/_2007_01_report-verification.pdf
http://is.ifmo.ru/verification/_2007_02_report-verification.pdf

54. Отчет о патентных исследованиях по 1-му этапу контракта о верификации автоматных программ.
http://is.ifmo.ru/verification/_2007_01_patent-verification.pdf

55. *Gnesi S., Mazzanti F.* A model checking verification environment for UML statecharts / Proceedings of XLIII Congresso Annuale AICA, 2005. http://fmt.isti.cnr.it/~gnesi/matdid/aica.pdf

56. *Волобуев В. Н., Калачинский А. В.* Опыт использования автоматного подхода при разработке программного обеспечения систем боевого управления // Системы управления и обработки информации. Вып. 18. 2009, с. 88–92.
http://is.ifmo.ru/works/_volobuev.pdf

57. *Ремизов А. О., Шалыто А. А.* Верификация автоматных программ / Сборник докладов научно-технической конференции «Состояние, проблемы и перспективы создания корабельных информационно-управляющих комплексов. ОАО «Концерн Моринформсистема Агат». М.: 2010, с. 90–98.
http://is.ifmo.ru/works/_2010_05_25_verific.pdf

58. *Гуров В. С., Мазин М. А., Нарвский А. С., Шалыто А. А.* Инструментальное средство для поддержки автоматного программирования // Программирование. 2007. № 6, с. 65–80.
http://is.ifmo.ru/works/_2008_01_27_gurov.pdf

59. *Zakonov A., Stepanov O., Shalyto A.* GA-Based and Design by Contract Approach to Test Generation for EFSMs / Proceedings of IEEE East-West Design & Test Symposium (EWDTS`10). St. Petersburg. 2010, pp. 152–155.
http://is.ifmo.ru/works/_ewdts_2010_zakonov.pdf

60. *Клебанов А. А., Степанов О. Г., Шалыто А. А.* Применение шаблонов требований к формальной спецификации и верификации автоматных программ / Труды семинара «Семантика, спецификация и верификация программ: теория и приложения». Казань, 2010, с. 124–130.
http://is.ifmo.ru/works/_2010-10-01_klebanov.pdf

61. *Barr M.* Real men program in C.
http://www.eetimes.com/General/DisplayPrintViewContent?contentItemId=4027479

62. *Кольский Н. И.* Язык программирования встроенных систем: свобода выбора или жесткий детерминизм? // Мир компьютерной автоматизации: встраиваемые компьютерные системы. 2010. № 4, с. 54–60.

63. *Вельдер С. Э., Шалыто А. А.* Верификация автоматных моделей методом редуцированного графа переходов // Научно-технический вестник СПбГУ ИТМО. 2009. Вып. 6(64), с. 66–77.
http://is.ifmo.ru/works/_2010_01_29_velder.pdf

64. *Егоров К. В., Шалыто А. А.* Методика верификации автоматных программ // Информационно-управляющие системы. 2008. № 5, с. 15–21. http://is.ifmo.ru/works/_egorov.pdf

65. *Курбацкий Е. А.* Верификация программ, построенных на основе автоматного подхода с использованием программного средства SMV // Научно-технический вестник СПбГУ ИТМО. Вып. 53. Автоматное программирование. 2008, с. 137–144.
http://books.ifmo.ru/ntv/ntv/53/ntv_53.pdf

66. *Лукин М. А., Шалыто А. А.* Верификация автоматных программ с использованием верификатора SPIN // Научно-технический вестник СПбГУ ИТМО. Вып. 53. Автоматное программирование. 2008, с. 145–162.
http://books.ifmo.ru/ntv/ntv/53/ntv_53.pdf

67. *Гуров В. С., Яминов Б. Р.* Верификация автоматных программ при помощи верификатора UNIMOD.VERIFIER // Научно-технический вестник СПбГУ ИТМО. Вып. 53. Автоматное программирование. 2008, с. 162–176.
http://books.ifmo.ru/ntv/ntv/53/ntv_53.pdf

68. *Roux C., Encrenaz E.* CTL May Be Ambiguous when Model Checking Moore Machines. UPMC LIP6 ASIM, CHARME, 2003.
http://sed.free.fr/cr/charme2003.ps

69. *Кузьмин Е. В.* Иерархическая модель автоматных программ // Моделирование и анализ информационных систем. 2006. № 1, с. 27–34. http://is.ifmo.ru/verification/_hamp.pdf

70. *Виноградов Р. А., Кузьмин Е. В., Соколов В. А.* Верификация автоматных программ средствами CPN/Tools // Моделирование и анализ информационных систем. 2006. № 2, с. 4–15.
http://is.ifmo.ru/verification/_cpnverif.pdf

71. *Васильева К. А., Кузьмин Е. В.* Верификация автоматных программ с использованием LTL // Моделирование и анализ информационных систем. 2007. № 1, с. 3–14.
http://is.ifmo.ru/verification/_LTL_for_Spin.pdf

72. *Васильева К. А., Кузьмин Е. В., Соколов В. А.* Верификация автоматных программ с использованием LTL.
http://is.ifmo.ru/verification/_ltl_aut_ver_1.pdf

73. *Кузьмин Е. В., Соколов В. А.* Моделирование, спецификация и верификация автоматных программ // Программирование. 2008. № 1, с. 38–60.
http://is.ifmo.ru/download/2008-03-12_verification.pdf

74. *Козлов В. А., Комалева О. А.* Моделирование работы банкомата. СПбГУ ИТМО, 2006.
http://is.ifmo.ru/unimod-projects/bankomat

75. *Сайт eVelopers Corporation.* http://www.evelopers.com

76. *Сайт проекта UniMod.* http://unimod.sf.net

77. Отчет по контракту о верификации автоматных программ. Этапы 3, 4. 2008.
http://is.ifmo.ru/verification/_2007_03_report-verification.pdf
http://is.ifmo.ru/verification/_2007_04_report-verification.pdf

78. *Лукин М. А.* Верификация автоматных программ. Бакалаврская работа. СПбГУ ИТМО, 2007.
http://is.ifmo.ru/papers/_lukin_bachelor.pdf

79. *Лукин М. А.* Верификация визуальных автоматных программ с использованием инструментального средства SPIN. Магистерская работа. СПбГУ ИТМО, 2009.
http://is.ifmo.ru/papers/_lukin_master.pdf

80. *Spin home page.* http://spinroot.com

81. *Яминов Б. Р.* Автоматизация верификации автоматных UniMod-моделей на основе инструментального средства Bogor. Бакалаврская работа. СПбГУ ИТМО, 2007.
http://is.ifmo.ru/papers/_jaminov_bachelor.pdf

82. *Яминов Б. Р.* Сравнение методов верификации UniMod-моделей. Магистерская работа. СПбГУ ИТМО, 2009.
http://is.ifmo.ru/papers/_jaminov_master.pdf

83. *Bogor home page.* http://bogor.projects.cis.ksu.edu

84. *Вельдер С. Э., Шалыто А. А.* Методы верификации моделей автоматных программ // Научно-технический вестник СПбГУ ИТМО. Вып. 53. Автоматное программирование. 2008, с. 123–136.
http://books.ifmo.ru/ntv/ntv/53/ntv_53.pdf

85. *Шалыто А. А.* Логическое управление. Методы аппаратной и программной реализации алгоритмов. СПб.: Наука, 2000.
http://is.ifmo.ru/books/log_upr/1

86. *Вельдер С. Э., Бедный Ю. Д.* Универсальный инфракрасный пульт для бытовой техники. СПбГУ ИТМО, 2005.
http://is.ifmo.ru/projects/irrc/

87. *Вельдер С. Э.* Введение в верификацию автоматных программ на основе метода model checking. Бакалаврская работа. СПбГУ ИТМО, 2006.
http://is.ifmo.ru/papers/_velder_bachelor.pdf

88. *Егоров К. В., Шалыто А. А.* Разработка верификатора автоматных программ // Научно-технический вестник СПбГУ ИТМО. Вып. 53. Автоматное программирование. 2008, с. 177–188.
http://books.ifmo.ru/ntv/ntv/53/ntv_53.pdf

89. *Хоффман Л.* Разговоры о model checking // Communications of the ACM. 2008. Vol. 51. № 07/08, pp. 110–112.
http://is.ifmo.ru/verification/_model_checking.pdf

90. *Хоффман Л.* В поисках надежного кода // Communications of the ACM. 2008. Vol. 51. № 07/08, pp. 14–16.
http://is.ifmo.ru/verification/_v_poiskax_nadejnogo_koda.pdf

91. *Biere A., Heule M., Maaren H. van, Walsh T. (eds.)* Handbook of Satisfiability. IOS Press, 2009.

92. *Long O. E.* Model Checking, Abstraction and Compositional Reasoning. PhD thesis. Carnegie Mellon University, 1993.

93. *Schnoebelen P., Bérard B., Bidoit M., Laroussinie F., Petit A.* Vérification de logiciels: techniques et outils du model-checking. Vuibert, 1999.

94. *Bérard B., Bidoit M., Finkel A., Laroussinie F., Petit A., Petrucci L., Schnoebelen P.* Systems and Software Verification. Model-Checking Techniques and Tools. Springer, 2001.

95. *Кубасов С. В., Соколов В. А.* Синхронная модель автоматной программы // Моделирование и анализ информационных систем. 2007. № 1, с. 11–18. http://mais.uniyar.ac.ru/ru/article/61

96. *Вельдер С. Э.* Применение методов снижения размерности к задачам верификации TCTL и оптимальной укладки графов. Магистерская диссертация. СПбГУ ИТМО, 2008. http://is.ifmo.ru/papers/_velder_master.pdf

97. *Кубасов С. В.* Верификация автоматных программ в контексте синхронного программирования. Диссертация на соискание ученой степени канд. техн. наук. Ярославль. ЯГУ им. П. Г. Демидова, 2008. http://is.ifmo.ru/disser/kubasov_disser.pdf

98. *Вельдер С. Э.* Верификация моделей автоматных программ. Диссертация на соискание ученой степени канд. техн. наук. СПбГУ ИТМО, 2011.

99. *Кузьмин Е. В.* Алгоритмические свойства формальных моделей параллельных и распределенных систем. Диссертация на соискание ученой степени докт. физ.-мат. наук. Ярославль. ЯГУ им. П. Г. Демидова, 2010.

Алфавитный указатель

В 2009 году Университет стал победителем многоэтапного конкурса, в результате которого определены 12 ведущих университетов России, которым присвоена категория «Национальный исследовательский университет». Министерством образования и науки Российской Федерации была утверждена Программа развития государственного образовательного учреждения высшего профессионального образования «Санкт-Петербургский государственный университет информационных технологий, механики и оптики» на 2009–2018 годы.

КАФЕДРА КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ

Кафедра компьютерных технологий была основана в 1991 году по инициативе профессоров В. Н. Васильева и В. Г. Парфенова для проведения специального образовательного проекта, имеющего целью создание сквозной школьно-вузовской системы отбора, подготовки и трудоустройства одаренных в области точных наук школьников и студентов. За прошедшие два десятилетия проект получил широкую известность и многочисленные положительные отзывы российских и зарубежных специалистов. В настоящее время на кафедре компьютерных технологий обучается порядка 200 студентов, более половины из которых являются победителями региональных олимпиад по точным наукам, а около семидесяти – победителями Международных и Всероссийских олимпиад по математике, физике и информатике.

По качеству студенческого и преподавательского состава и техническому оснащению компьютерной техникой кафедра компьютерных технологий относится к числу лучших российских специализированных компьютерных кафедр.

Сергей Эдуардович Вельдер,
Михаил Андреевич Лукин,
Анатолий Абрамович Шалыто,
Булат Равилевич Яминов

ВЕРИФИКАЦИЯ АВТОМАТНЫХ ПРОГРАММ

Учебное пособие

В авторской редакции
Дизайн и верстка С. Э. Вельдер
Редакционно-издательский отдел
Санкт-Петербургского государственного университета
информационных технологий, механики и оптики
Зав. РИО Н. Ф. Гусарова
Лицензия ИД № 00408 от 05.11.99
Подписано к печати 27.01.2011 г.
Заказ № 1527
Тираж 150 экз.

Учреждение «Университетские Телекоммуникации»

«Типография на Биржевой»
199034, СПб, В.О., Биржевая линия, д. 14–16
тел.: +7 (812) 915-14-54, e-mail: zakaz@TiBir.ru
www.TiBir.ru

Редакционно-издательский отдел
Санкт-Петербургского государственного
университета информационных технологий,
механики и оптики
197101, Санкт-Петербург, Кронверкский пр., 49