Ю. Г. Карпов

MODEL CHECKING

Верификация параллельных и распределенных программных систем

«Автор будет считать свою задачу выполненной, если его книга хотя бы в малой степени изменит существующее положение в отечественной науке и образовании в области верификации программ. Следует признаться, однако, что забота об отечественной науке составила лишь долю мотивации написания этой книги. Намного более сильным было желание автора поделиться с читателем тем восхищением, которое вызывают изящество формальных моделей и их удивительная мощь при решении важнейших насущных практических проблем информатики».

Юрий Карпов

+cd

Ю. Г. Карпов

MODEL CHECKING

Верификация параллельных и распределенных программных систем

Санкт-Петербург
«БХВ-Петербург»
2010

УДК 681.3.06
ББК 32.973.26-018.2
К26

Карпов Ю. Г.

К26 MODEL CHECKING. Верификация параллельных и распределенных программных систем. — СПб.: БХВ-Петербург, 2010. — 560 с.: ил. + CD-ROM

ISBN 978-5-9775-0404-1

В книге рассказывается о новых результатах в области верификации с помощью метода *model checking* и приводятся примеры приложений этого метода в самых разных областях. Рассматриваются проблема верификации, темпоральные логики, алгоритмы *model checking* для CTL и LTL, структуры Крипке как модели реагирующих систем, спецификация свойств реагирующих систем формулами темпоральной логики, бинарные решающие диаграммы, символьная верификация, количественный анализ систем и системы реального времени, а также применение фундаментальных идей алгоритма *model checking* в различных приложениях.

На прилагаемом компакт-диске находятся: инструкция по установке системы верификации Spin, методическое пособие по этой системе и описание курсовой работы по верификации нетривиальной системы логического управления с несколькими вариантами заданий.

Для студентов вузов, программистов
и исследователей в области информатики

УДК 681.3.06
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Татьяна Лапина*
Зав. редакцией	*Григорий Добин*
Редактор	*Наталья Сержантова*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 01.10.09.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 45,15.
Тираж 1000 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.60.953.Д.005770.05.09 от 26.05.2009 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0404-1

© Карпов Ю. Г., 2009
© Оформление, издательство "БХВ-Петербург", 2009

Оглавление

Предисловие 5

Структура книги 8
Сопровождающий компакт-диск 10
Благодарности 10

Глава 1. Проблема верификации 13

1.1. Ошибки в программах и их последствия 13
1.2. Примеры ошибочных программ 18
1.3. Общая схема верификации. Проверка моделей (model checking) 24
1.4. Тестирование и верификация 30
1.5. Примеры успешной верификации систем 33
1.6. Инструменты верификации 35
1.7. Заключение 36
1.8. Замечания 37
1.9. Задачи к главе 1 38

Глава 2. Темпоральные логики 41

2.1. Утверждения, истинность которых зависит от времени 41
2.2. Модальные и временные логики. Tense Logic 45
2.3. Темпоральная логика линейного времени (LTL) 51
2.4. Реагирующие системы (reactive systems) 58
2.5. Формальное определение LTL 62
2.6. Примеры использования формул LTL 64
2.7. Соотношения между операторами LTL 66
2.8. Структура Крипке 68
2.9. Расширенная темпоральная логика ветвящегося времени CTL* 73
2.10. Сравнение логик LTL и CTL 80
2.11. Темпоральные логики прошлого 82
2.12. Model checking 82

2.13. Заключение ... 83
2.14. Замечания ... 84
2.15. Задачи к главе 2 ... 85

Глава 3. Алгоритм model checking для CTL ... 91

3.1. Темпоральная логика ветвящегося времени CTL ... 91
3.2. Семантика CTL на деревьях вычислений ... 92
3.3. Формальная семантика CTL ... 96
3.4. Проверка формул CTL на развертке структуры Крипке ... 98
3.5. Базисы CTL ... 99
3.6. Алгоритм model checking для CTL ... 103
3.7. Пример выполнения алгоритма model checking ... 111
3.8. Заключение ... 113
3.9. Замечания ... 113
3.10. Задачи к главе 3 ... 114

Глава 4. Алгоритм model checking для LTL ... 117

4.1. Проверка выполнимости LTL-формул на вычислениях ... 118
Поиск контрпримеров ... 119
Контрольный автомат ... 120
4.2. Пересечение языков в теории конечных автоматов ... 123
Конечные автоматы и автоматные языки ... 123
Синхронная композиция автоматов и пересечение языков ... 125
4.3. Теоретико-автоматный метод проверки выполнимости LTL-формулы ... 128
4.4. Автоматы Бюхи — модели для задания ω-языков ... 130
ω-языки и поведение реагирующих систем ... 131
Автомат Бюхи как контрольный автомат для реагирующей системы ... 134
4.5. Операции над автоматами Бюхи ... 135
4.6. Автоматы Бюхи и LTL-формулы ... 141
4.7. Структуры Крипке и автоматы Бюхи ... 144
4.8. Проверка модели для формул LTL ... 146
4.9. Построение автомата Бюхи по формуле LTL ... 147
Разметка состояний вычисления формулами LTL ... 148
Атомы темпоральной формулы логики LTL ... 150
Переходы между состояниями искомого автомата Бюхи ... 155
Допускающие состояния искомого автомата Бюхи ... 157
Построение автомата Бюхи по формуле LTL ... 157
4.10. Заключение ... 164
4.11. Замечания ... 166
4.12. Задачи к главе 4 ... 167

Глава 5. Структуры Крипке как модели реагирующих систем ... 171

5.1. Состояния и переходы реагирующей системы. Гранулярность переходов ... 171
5.2. Реагирующие системы, специфицированные в виде систем переходов ... 175

5.3. Реагирующие системы, специфицированные в виде нескольких взаимодействующих систем переходов ... 177
5.4. Представление программы с конечным числом состояний структурой Крипке ... 186
5.5. Представление параллельных программ структурой Крипке ... 189
5.6. Пакеты верификации и структура Крипке. Пакет верификации Spin ... 197
5.7. Построение структуры Крипке для программ высокого уровня ... 204
5.8. Заключение ... 206
5.9. Замечания ... 207
5.10. Задачи к главе 5 ... 209

Глава 6. Спецификация свойств реагирующих систем формулами темпоральной логики ... 215

6.1. Примеры спецификации свойств технических систем ... 216
Причинно-следственная связь событий ... 216
Примеры спецификации свойств систем формулами LTL ... 218
Спецификация формулами LTL выполнения события на вычислениях ... 219
Спецификация формулами LTL отсутствия события на вычислениях ... 219
Примеры спецификации свойств систем формулами CTL ... 219
6.2. Свойства достижимости (reachability) ... 221
6.3. Свойства безопасности (safety) ... 222
6.4. Свойства живости (liveness) ... 227
6.5. Свойства справедливости (fairness) ... 229
6.6. Спецификация справедливости в логике CTL ... 233
6.7. Справедливая CTL (fair CTL) ... 237
6.8. Заключение ... 244
6.9. Замечания ... 244
6.10. Задачи к главе 6 ... 245

Глава 7. Примеры верификации ... 247

7.1. Протокол передачи данных W.C.Lynch ... 247
7.2. Протокол PAR ... 250
7.3. Протокол двухфазной фиксации ... 253
7.4. Заключение ... 255
7.5. Замечания ... 256
7.6. Задачи к главе 7 ... 257

Глава 8. Применения алгоритма model checking ... 259

8.1. Анализ бизнес-процессов ... 259
8.2. Проблема планирования как model checking ... 262
8.3. Логические головоломки и model checking ... 267
8.4. Верификация криптографических протоколов ... 279
8.5. Заключение ... 290
8.6. Замечания ... 291
8.7. Задачи к главе 8 ... 292

Глава 9. Бинарные решающие диаграммы 295

9.1. Проблемы с представлением булевых функций 296
9.2. Бинарные решающие диаграммы 301
9.3. Свойства бинарных решающих диаграмм 308
9.4. Операции над BDD 312
9.5. Проблема булевой выполнимости (SAT) и бинарные решающие диаграммы 328
9.6. Применения BDD к решению логических и комбинаторных задач 330
9.7. Использование BDD в анализе и синтезе логических схем 340
9.8. Конечные структуры данных и BDD 345
Представление отношений с помощью характеристических булевых функций 347
Операции над отношениями 349
9.9. Символьные и неявные алгоритмы обработки данных 355
Пример символьного алгоритма: анализ достижимости на графе 356
9.10. Бинарные решающие диаграммы с подавлением нулей 358
9.11. Заключение 363
9.12. Замечания 364
9.13. Задачи к главе 9 366

Глава 10. Символьная верификация 367

10.1. Необходимость символьных методов в верификации 368
10.2. Представление структуры Крипке булевыми функциями 370
10.3. Представление структуры Крипке реагирующих систем булевыми функциями без предварительного явного построения структуры Крипке 372
10.4. Представление булевыми функциями структуры Крипке для программ 375
10.5. Символьный алгоритм model checking для CTL 380
10.6. Операторы на множествах и неподвижные точки 385
10.7. Теорема Тарского о неподвижной точке 390
Решетки 392
10.8. Символьный алгоритм верификации для формулы EFφ 395
10.9. Определение темпоральных формул CTL через неподвижные точки операторов 403
10.10. Заключение 410
10.11. Замечания 411
10.12. Задачи к главе 10 411

Глава 11. Количественный анализ систем 415

11.1. Вероятностный метод model checking 416
11.2. Вероятностная логика ветвящегося времени (Probabilistic CTL, PCTL) и верификация вероятностных моделей 427
11.3. Проверка свойств, зависящих от дискретного времени и вероятностного выбора 432
11.4. Примеры свойств, выражаемых в PCTL 437
11.5. Заключение 440
11.6. Замечания 441
11.7. Задачи к главе 11 442

Глава 12. Системы реального времени 445

12.1. Модель систем реального времени. Временнòй автомат и его поведение 446
12.2. Сети временных автоматов: параллельная композиция 453
12.3. Операционная семантика модели временнòго автомата — система переходов 457
12.4. Нереалистичные поведения временнòго автомата 462
12.5. Проблема анализа системы переходов временного автомата 465
12.6. Временные регионы: один таймер 468
12.7. Граф регионов временного автомата 472
12.8. Временные регионы для нескольких локальных часов 476
12.9. Анализ временных автоматов: логика CTL 482
12.10. Временная темпоральная логика ветвящегося времени (Timed CTL) 489
Интерпретация формул TCTL 491
Формальная семантика TCTL 494
12.11. Анализ временных автоматов: сведение проверки TCTL-формул к проверке CTL-формул 496
12.12. Сложность алгоритмов model checking для логики TCTL 506
12.13. Другие проблемы верификации временных автоматов 507
Допускаемый язык 507
Бисимуляция 508
Достижимость 510
12.14. Временные зоны 511
Временные регионы и временные зоны 511
Операции над зонами 515
Пересечение зоны с ограничениями на показания часов 515
Приращение времени 515
Достижимость за один шаг 516
Структуры данных для представления временных зон 521
Матрица разностей границ (Difference Bound Matrix, DBM) 522
Разностная решающая диаграмма (Difference Decision Diagram, DDD) 524
12.15. Инструменты верификации систем реального времени 525
Uppaal 525
Kronos 526
12.16. Заключение 527
12.17. Замечания 529
12.18. Задачи к главе 12 530

Заключение 533

Список литературы 535

Иностранные издания 535
Литература на русском языке 544
Материалы из Интернета 545

Предметный указатель 547

Анализ корректности программных систем — тема, которая не теряет своей актуальности. Поэтому неудивительно, что тысячи исследователей и инженеров во всем мире пытаются изобрести новые методы и разработать новые инструменты проверки программ. Специалисты по теоретической информатике даже создали отдельную ветвь этой дисциплины, которая называется "формальные методы разработки и анализа программ". Однако, несмотря на массированные атаки на проблему, до последнего времени не удавалось добиться реальных результатов, которые можно было внедрять в широкую практику.

Model checking оказался одним из первых формальных методов, который вышел из лабораторий и университетов в мир промышленной разработки программных и аппаратных систем. Это одновременно радует и удивляет специалистов в области верификации программ — ведь для того, чтобы тонкий математический инструмент стал простым и понятным для обычных программистов, нужно было решить не только многие научные и технические проблемы, но и преодолеть привычный консерватизм, традиционную практику не рисковать и не полагаться в серьезных проектах на слишком новые и еще неопробованные технологии.

Такое "преодоление" невозможно без энтузиазма людей увлеченных и авторитетных. Лишь они могут сломать предубеждение по отношению к новым технологиям. Юрий Глебович Карпов — один из них. Он сочетает в себе высокую компетенцию и юношескую увлеченность изяществом теории, о которой он пишет. В результате читатель получил, пожалуй, первую книгу на русском языке, которая раскрывает потенциал нового поколения инструментов разработки программ, построенных на основе использования формальных методов.

В. П. Иванников

Академик РАН, директор Института системного программирования РАН, заведующий кафедрами системного программирования на факультете вычислительной математики и кибернетики МГУ и в МФТИ, главный редактор журнала "Программирование"

Это необычная книга. Особенность ей придают три обстоятельства:

- Актуальность и прикладная значимость ее тематики: сейчас уже смело можно сказать, что метод model checking проверки правильности функционирования компьютерных систем и устройств и их программного обеспечения вышел за пределы лабораторий и активно стал использоваться в промышленности.
- Личность автора: известного специалиста и одного из основных подвижников в исследованиях применения темпоральных логик для верификации программ и систем у нас в стране. Дар преподавателя помог ему найти и построить свою методику изложения, отличную от таких известных публикаций, например, как E. M. Clarke, O. Grumberg, D. Peled. Model checking // MIT Press, 1999. Русский перевод: Э. М. Кларк, О. Грамберг, Д. Пелед. Верификация моделей программ: Model Checking // М., 2002.
- На сегодня эта книга, пожалуй, наиболее полное изложение метода model checking и многочисленных его приложений.

Р. Л. Смелянский

Заведующий лабораторией вычислительных комплексов факультета ВМиК МГУ, доктор физико-математических наук, профессор, академик РАЕН

Предисловие

21 июня 2008 года на торжественном банкете Ассоциации по вычислительной технике (ACM) в Сан-Франциско, Калифорния, трем ученым была вручена высшая награда Ассоциации — премия Тьюринга 2007 г. — "*за их роль в превращении метода Model checking (по-русски — "проверка модели") в высокоэффективную технологию верификации, широко используемую в индустрии разработки программного обеспечения и аппаратных средств*".

Model checking — это формальная проверка того, выполняется ли заданная логическая формула на данной структуре. Структура представляет собой модель разрабатываемой системы, логическая формула описывает требования к поведению системы. Этот метод уже сегодня широко используется во всем мире для проверки сложных объектов — как программного обеспечения, так и аппаратуры. Он позволяет существенно повысить степень уверенности разработчиков в правильности функционирования интегральных схем, протоколов коммуникации, драйверов устройств, бортовых систем управления для автомобилей, самолетов, космических аппаратов.

Премия Тьюринга рассматривается мировым сообществом как Нобелевская премия в области информатики. Она вручается ежегодно с 1966 г. за выдающийся вклад в усовершенствование компьютерных технологий. Премия Тьюринга за 2007 г., пожалуй, впервые отметила ученых, вклад которых состоял не столько в разработке удивительно красивой теории, имеющей важное практическое применение, сколько в доведении этой теории до уровня "высокоэффективной промышленной технологии". Президент ACM Стюарт Фельдман сказал: "*Это великий пример того, как технология, изменившая промышленность, родилась из чисто теоретических исследований*".

Новые научные результаты были сразу востребованы практикой. Без концептуального прорыва в теории верификации, без использования этих новых результатов разрабатываемые сегодня программные и аппаратные продукты

беспрецедентной сложности не могли бы быть созданы с достаточным уровнем надежности, а компании-разработчики в перспективе разорились бы, выплачивая неустойки за ущерб, нанесенный ошибочным функционированием их разработок. Однако едва ли не большее значение метод model checking приобрел в последние годы в связи с возможностью применения его результатов и идей во многих областях, весьма далеких от области верификации технических систем, для которой этот метод был первоначально разработан.

Данная книга посвящена изложению этих новых идей и описанию их применений. В книге рассказывается о новых результатах в области верификации, полученных с помощью метода model checking, и приводятся примеры использования этого метода в самых разных областях. Эта область теоретической информатики была разработана сравнительно недавно, но она уже привела к "прорывным" практическим результатам и вызвала значительный интерес из-за своей изящности и эффективности.

Верификация программных систем — фундаментальный раздел информатики, основанный на формальных методах. Трудность изучения и восприятия формальных методов часто является следствием оторванности таких формализмов от практики. Среди программистов широко распространено мнение, что формальные методы не имеют никакого реального применения к решению практических проблем, что это просто необязательное приложение к рутинным приемам техники программирования, которые одни только и нужны программисту. Подход *model checking* опровергает это мнение. Несмотря на то, что полученные здесь результаты базируются на таких абстракциях, как темпоральные логики и структуры Крипке, бесконечные языки и автоматы Бюхи, неподвижные точки операторов, а также на других "страшных" формализмах, материализация этих абстрактных понятий дала удивительные практические результаты. На их основе созданы и работают автоматические системы верификации, позволяющие разработчикам систем методом "*нажатия кнопки*" (push button) проверять бортовые системы управления космических аппаратов, встроенные программы военной техники, мобильных телефонов, медицинской аппаратуры. Путь от этих абстракций к практическим задачам информатики удивительно прям и очевиден. Некоторые ученые называют это направление "*реабилитацией формальных методов*": здесь формальные модели позволили получить результаты, без применения которых невозможно выполнить любую практическую разработку программных и аппаратных систем для серьезных, критических применений.

Материал книги будет полезен трем категориям читателей.

Во-первых, это студенты, обучающиеся по направлению "Информатика". Изучение теории и методов, объединенных под общим термином model checking, включено в учебные планы многих иностранных университетов.

Такой интерес вызван тем, что этот раздел представляет наиболее значительные результаты в области верификации с тех пор, как эта проблема впервые была поставлена 40 лет назад. Интерес вызван также простотой и логической ясностью этих результатов, а также возможностью их практического применения. Этот новый раздел теоретической информатики станет, несомненно, обязательной частью базового фундаментального образования в области информатики. "Формальные методы должны стать частью образования каждого исследователя в области информатики и каждого разработчика ПО так же, как другие ветви прикладной математики являются необходимой частью образования в других инженерных областях", — говорится в отчете NASA. Хотя техника применения этого подхода проще, чем теория, верификация систем требует достаточно высокой квалификации исполнителей. Специалисты в этой области весьма востребованы.

Во-вторых, это научные работники. В отличие от многих теоретических разделов вычислительной науки, которые стали уже классикой, данное направление является "горячей" областью исследований, здесь продолжается активная исследовательская работа, которая направлена на расширение возможностей верификации технических систем и на применение этого подхода к широкому кругу других задач. Богатство применений этой новой техники в совершенно разных областях, огромное число открытых проблем не могут оставить равнодушным ни одного исследователя. В книге приводятся примеры того, как эти новые идеи используются для решения многих проблем в различных областях.

Третьей категорией читателей, которые могут извлечь пользу из данной книги, являются инженеры-разработчики новых систем. "Формальные методы обеспечивают интеллектуальную основу нашей области, они помогают оформить наши мысли и направить их на плодотворный путь при решении проблем; они обеспечивают нотацию при документировании требований и разработки, позволяют выразить мысли при общении разработчиков в точной и ясной манере, они обеспечивают нас инструментальными средствами для анализа свойств систем", — говорится в упомянутом выше отчете NASA. Раздел формальной верификации, основанный на методе *model checking*, достиг уровня зрелости. Он предоставляет разработчику программных и аппаратных систем теоретические основы, ясную методику, простую нотацию и эффективный инструментарий для верификации систем. Материал данной книги достаточен для того, чтобы любой программист, заботящийся о качестве результатов своего труда, смог немедленно применить изложенные здесь методы для проверки своей разработки и повышения степени доверия к ней, используя доступный инструментарий и технику "push button".

Для понимания книги необходимы только начальные знания в области дискретной математики: логики высказываний и теории автоматов в объеме

вводных университетских курсов, а также первоначальный опыт программирования.

В одной вводной книге невозможно изложить все, или даже большую часть вопросов, относящихся к данной области. Автор постарался по возможности просто и компактно раскрыть основные идеи и принципы этого направления. Интересующиеся читатели без труда смогут найти литературу, относящуюся к другим разделам этой области, и после прочтения данной книги разобраться в ней.

К настоящему времени формальная верификация уже стала этапом технологического цикла при создании сложных встроенных систем во многих фирмах-разработчиках программно-аппаратных систем для критических применений. Но среди названий этих фирм трудно найти российские. Курсы по формальным моделям, темпоральным логикам и верификации стали традиционными во всех ведущих университетах мира — но их читают только в ничтожно малом числе вузов России. Существует обширная литература, посвященная исследованиям в этой области, но на русском языке такой литературы почти нет. Исследования проводятся во всех развитых странах — кроме России, где этими проблемами занимаются только в двух-трех научных группах. Результаты этого очевидны. Например, компания "Сухой" с гордостью рапортует о заключенных контрактах на поставку ее истребителей за рубеж, но на экспортируемые самолеты ставится электроника, сделанная не в России, а во Франции. Именно во Франции работает один из трех лауреатов премии Тьюринга 2007 г., в нескольких лабораториях Франции ведутся интенсивные исследования в области верификации и разработаны технологии создания надежных бортовых систем. Очевидно, что зарубежная авионика может иметь лишь очень ограниченное применение в отечественной боевой авиации и в других критических областях.

Автор будет считать свою задачу выполненной, если его книга хотя бы в малой степени изменит существующее положение в отечественной науке и образовании в этой области.

Следует признаться, однако, что забота об отечественной науке составила лишь долю мотивации для написания этой книги. Намного более сильным было желание автора поделиться с читателем тем восхищением, которое вызывают изящество формальных моделей и их удивительная мощь при решении важнейших насущных практических проблем информатики.

Структура книги

Представленный в книге материал может составить основу одно- или двухсеместрового университетского курса по верификации параллельных и распределенных программ.

Структура представленного материала следующая.

Глава 1 является вводной. В ней описываются некоторые примеры ошибок в программных системах и их последствия. Показывается, что именно параллельные и распределенные программы являются сложными для их понимания и анализа, что формальная верификация является важнейшим средством выявления тонких ошибок в параллельных системах. Приводится общая схема верификации и рассматривается метод model checking как один из подходов к формальной верификации. Здесь же анализируются сильные и слабые стороны верификации и тестирования, этих двух классических методов повышения уровня доверия к разработкам. Приводятся примеры успешного применения верификации к практическим разработкам программного обеспечения и аппаратуры: коммуникационным протоколам, бортовым системам управления космическими аппаратами и т. п.

В *главе 2* вводятся темпоральные логики LTL, CTL и CTL* и рассматривается их использование для спецификации свойств поведения реагирующих систем.

Главы 3 и *4* посвящены изложению алгоритмов проверки модели для случаев, когда проверяемые свойства системы выражены формулами логик CTL и LTL.

В *главе 5* изучаются системы переходов как модели реагирующих систем.

Глава 6 посвящена рассмотрению свойств реагирующих систем и их классификации: безопасности, живости и справедливости.

В *главе 7* описываются некоторые примеры верификации с помощью представленной ранее методики проверки модели.

Глава 8 приводит многочисленные примеры применения алгоритма проверки моделей в самых разных областях — решение логических задач, задач планирования, поиск атак в криптографическом протоколе.

Одним из препятствий при использовании метода проверки модели является экспоненциальный рост числа состояний параллельных систем при увеличении числа компонентов и параметров модели. Эта проблема, называемая "*взрывом числа состояний*", рассматривается в *главах 9* и *10*. В *главе 9* рассказывается о новом методе представления булевых функций — бинарных решающих диаграммах (Binary Decision Diagrams, BDD), который позволяет эффективно представлять любые конечные дискретные структуры данных и эффективно оперировать ими, а также приводятся примеры применений этого метода. Данную главу можно изучать независимо от других глав, но этот материал является основным для рассмотрения в следующей, *10 главе*, символьных алгоритмов верификации, позволяющих анализировать системы с астрономическим числом состояний — 10^{100} и более.

В *главе 11* изучаются подходы к анализу количественных свойств дискретных систем — вероятностных систем и систем дискретного времени. Здесь рассматривается, как проверить свойства, выражаемые утверждениями типа: "с вероятностью, не меньшей 0.99, на переданный пакет будет получено подтверждение не более чем через 6 единиц времени".

Наконец, *глава 12* посвящена проблемам верификации систем реального времени.

Практическая часть курса состоит из двух компонентов. Во-первых, в каждой главе книги существует раздел с задачами и упражнениями. Во-вторых, студенты могут получить опыт формального моделирования и анализа параллельных и распределенных систем на входном языке Promela системы верификации Spin в рамках курсовой работы. Материалы этой курсовой работы содержатся на диске.

Сопровождающий компакт-диск

На диске, приложенном к книге, находятся:

- инструкция по установке системы верификации Spin на компьютере студента;
- методическое пособие по этой системе верификации, содержащее описание языка Promela, примеры представления на нем параллельных и распределенных программ и примеры использования системы Spin для верификации программ и протоколов;
- описание курсовой работы по верификации нетривиальной системы логического управления с несколькими вариантами заданий для группы студентов.

По запросу преподавателей им могут быть высланы ответы и решения к задачам и упражнениям, приведенным в книге, а также слайды курса лекций, которые читает автор по этому предмету.

Благодарности

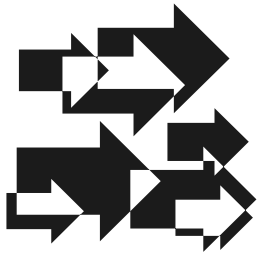
Материал книги основан, прежде всего, на фундаментальной монографии Э. Кларка, О. Грамберга и Д. Пеледа [34], которая во многих университетах мира является базовым учебником для курсов по верификации и model checking. В книге использованы также материалы из руководств, оригинальных статей и многочисленных учебных материалов, доступных в Интернете. Недавняя монография Кристела Байера и Питера Катоена по принципам model checking [19] также была использована при подготовке книги.

Автор выражает огромную благодарность всем людям, которые предоставили в Интернете в открытом доступе свои статьи, курсы лекций, слайды презентаций на эту тему. Всех их трудно перечислить, но особую благодарность хотелось бы выразить Марте Квятковской (Marta Kwiatkowska) из Бирмингемского университета, Борису Коневу из Ливерпульского университета, Юрию Лифшицу из Калифорнийского технологического университета, Питеру Катоену (Joost-Pieter Katoen) из университета Аахена, Киму Ларсену (Kim Guldstrand Larsen), Полю Петерсону (Paul Pettersson), Пьеру Валперу (Pierre Wolper) и многим другим.

Автор читает курс с этим названием в Санкт-Петербургском политехническом университете уже 10 лет. Изложение множества вопросов, казавшихся ясными, многократно перерабатывалось после того, как на экзаменах выяснялось, что эти вопросы вызывали затруднения у студентов. За эту обратную связь автор приносит благодарность своим студентам.

Издание книги поддержано Я. Ю. Карповым, за что автор приносит ему особую благодарность.

ГЛАВА 1



Проблема верификации

В этой главе обосновывается необходимость верификации программ. Приводятся примеры тяжелых последствий программных ошибок в системах для критических применений, от которых все больше зависят жизнь и здоровье людей, использующих технику. Дается сравнительный анализ верификации и тестирования, представлена общая схема верификации и кратко охарактеризован тот конкретный подход к верификации, которому посвящена книга, — метод *проверки модели* (model checking).

1.1. Ошибки в программах и их последствия

Компьютерные программы после их разработки очень часто содержат ошибки. С этим знаком каждый, разработавший программу длиной хотя бы в несколько десятков строк. В современных программных системах избежать ошибок разработки невозможно. Сложность промышленных программных систем все время возрастает. Например, ОС Microsoft Windows 3.1, разработанная в 1992 г., содержит около 3 млн строк кода, Microsoft Windows 98 — 18 млн строк, RedHat Linux 6.2 (2000 г.) — около 20 млн строк, RedHat Linux 7.1 (2001) — уже 30 млн строк, Microsoft Windows XP (2002) — 40 млн строк. Мысленно охватить функционирование таких систем не в состоянии ни один человек даже при использовании современных технологий и методов абстрагирования и управления сложностью.

Сложность разрабатываемого программного обеспечения подошла к границе его понимания и, следовательно, его управляемости. Число потенциальных ошибок в программных системах постоянно растет. Например, в ОС Windows 95, по оценкам Microsoft, до сих пор остается несколько тысяч оши-

бок. Но эта ОС используется в бытовых персональных компьютерах, от нее не зависят жизни людей. Только сумасшедший поставит эту ОС в бортовую систему управления самолетом или даже автомашиной.

Особенно подвержены ошибкам параллельные, распределенные и многопоточные программы, которые характерны для бортовых систем управления. Хорошо известно, что даже в тех случаях, когда функционирование каждой из параллельных взаимодействующих компонентов системы абсолютно ясно, человеку трудно понять работу всей параллельной системы, процессы в которой взаимозависимы. Как пишет Лесли Лэмпорт, "очень легко ошибиться, когда делаешь утверждения о последовательности событий, происходящих в параллельных программах" [94]. Причина этого, по-видимому, в том, что человек в каждый момент может думать только об одной вещи. Параллельные системы, которые работают правильно "почти всегда", годами могут сохранять тонкие ошибки, проявляющиеся в редких, исключительных ситуациях. Такие ошибки обычно невозможно обнаружить тестированием. "Существует обширный печальный опыт того, что параллельные программы весьма упорно сопротивляются серьезным усилиям по выявлению в них ошибок", — писали Сьюзен Овицки и Лесли Лэмпорт еще 25 лет тому назад [121]. В результате все чаще компании подвергаются штрафным санкциям и вынуждены выплачивать неустойки за последствия ошибок, проявившихся после передачи готовой системы заказчику.

Человек не любит рассказывать о своих ошибках. Разработчики программ обычно не спешат уведомлять всех о своих неудачах. Ни одна компания, производящая программы и компьютерную аппаратуру, не склонна без особой надобности объявлять об ошибках в своих продуктах и их последствиях. Это одна из причин того, что ошибки в программных и аппаратных системах происходят значительно чаще, чем мы об этом узнаем. Только в случаях, имеющих серьезные последствия ошибочного поведения программ, проводится анализ причин ошибок и возникших из-за них потерь. Те результаты такого анализа, которые становятся широко известными, показывают, что материальные последствия ошибок в программных и аппаратных системах управления могут быть весьма значительными, порой приводящими к полному провалу дорогостоящих проектов и даже к разорению компаний-разработчиков ПО и аппаратуры. Вот некоторые наиболее впечатляющие примеры последствий ошибок в программах.

- 4 июня 1996 г. при первом запуске на 39-й секунде после старта взорвалась ракета "Ариан 5". "Ариан 5" — это аналог российской ракеты-носителя "Протон", которая разработана и производится Европейским космическим агентством. Бортовая система ракеты "Ариан 5" использовала ПО предыдущей версии, "Ариан 4", но архитектура аппаратной части

была модифицирована. После проведенного тщательного расследования было установлено, что причина аварии — ошибка в навигационной программе бортового компьютера. Потери от аварии были огромными. Кроме прямых потерь — утраты ракеты (затраты на ее разработку превышали 7 млрд долларов) и установленного на борту научного оборудования стоимостью более полумиллиарда долларов, ущерб составили и упущенная выгода от несостоявшихся запланированных запусков, и потеря на долгие годы репутации ракеты как надежного средства выведения полезной нагрузки на околоземную орбиту. Непрямые потери составили около 2 млрд долларов.

- 25 декабря 2004 г. после семилетнего полета космический зонд Гюйгенс отделился от автоматической международной межпланетной станции "Кассини" и начал спуск в атмосферу Титана — спутника Сатурна. 15 января 2005 г. зонд впервые в истории начал передавать информацию с поверхности Титана. Ошибка в бортовой программе зонда привела к тому, что половина переданной информации была потеряна. Затраты на разработку зонда составили около 2,5 млрд долларов.
- 28 мая 2008 г. Сообщение Ассошиэйтед Пресс: "NASA потеряло связь с марсоходом "Феникс". Выполнение задач, запланированных на второй день нахождения на Марсе американского космического аппарата "Феникс", отложено из-за возникших неполадок. Связь с "Фениксом" утеряна". Стоимость проекта — 420 млн долларов.
- В 1994 г. при разработке процессора Intel Pentium была допущена ошибка во встроенной программе деления: несколько констант вспомогательного массива не были инициализированы. Ошибка была обнаружена уже после того, как дефектные процессоры поступили в продажу, и компанией "Интел" все они были заменены. Потери компании составили сотни миллионов долларов.
- В декабре 2007 г. в новых четырехъядерных процессорах AMD Phenom (Barcelona) и Opteron, произведенных по новой, 65-нанометровой технологии, выявлена ошибка реализации буфера быстрого преобразования адреса кэш-памяти третьего уровня. Ошибка может приводить к зависанию системы.
- 26 февраля 2009 г. Агентство Bloomberg сообщает: "Японское подразделение крупнейшего швейцарского банка UBS из-за компьютерной ошибки чуть было не потратило 3 триллиона иен (31 млрд долларов) на выкуп облигаций компании Capcom. Финансовая организация объяснила, что собиралась разместить заказ на покупку облигаций на сумму в 30 млн иен (310 000 долларов), но из-за системной ошибки к заказу добавились пять нулей".

Финансовые потери от ошибок в программах только в США оцениваются в десятки миллиардов долларов в год. Например, в 2002 г. эти потери оценены в 59,6 млрд долларов [52]. Необходимость более тщательной проверки программ и аппаратных систем приводит к задержкам в выполнении проектов, увеличению времени вывода продукта на рынок (time-to-market), а это грозит потерями из-за конкуренции и в связи со срывами сроков поставки.

Иногда ошибки в автоматических системах приводят не только к материальным потерям, но и к потерям человеческих жизней.

- В 1982 г. канадской компанией Atomic Energy of Canada Ltd. был запущен в серию медицинский аппарат Therac-25, предназначенный для облучения раковой опухоли потоком электронов. В приборе использовалось новое программное обеспечение встроенной системы управления процессом облучения. Несмотря на "тщательное тестирование", которому подверглось ПО, в нем остались ошибки: в редко встречающихся режимах, в которые прибор мог войти при некоторых входных управляющих последовательностях, интенсивность облучения возрастала на 2 порядка. Прибор эксплуатировался в разных клиниках несколько лет. За время эксплуатации прибора некоторые больные получили передозировки облучения, двое пациентов умерли, несколько человек остались инвалидами.
- 20 декабря 1995 г. в катастрофе самолета "Боинг-757" (Колумбия, рейс из Майами в Кали) погибли 159 человек. Расследование показало, что причиной катастрофы была ошибка в одном символе программной системы управления полетом. Изготовитель бортового компьютера, компания Honeywell Air Transport Systems и разработчик ПО компания Jeppesen Sanderson of Englewood были признаны виновными в гибели людей. Родственникам жертв аварии было выплачено 300 млн долларов.
- Февраль 2008 г. Сообщение СМИ: "Американский спутник-шпион вышел из-под контроля и может в конце февраля упасть на Землю. Обломки спутника, по информации источника в правительстве США, могут содержать опасные материалы, а точно установить, где именно упадет аппарат, пока не представляется возможным".
- 23 марта 2003 г. система "Patriot" ошибочно идентифицировала британский бомбардировщик "Tornado" как приближающуюся вражескую ракету и автоматически произвела залп. Погибли два пилота. 2 апреля 2003 г. системой "Patriot" уничтожен американский истребитель F-16, пилот погиб. В обоих случаях причиной явились ошибки в бортовой системе автоматического обнаружения цели и наведения.
- 11 апреля 2008 г., сообщение СМИ: "В результате операции во время патрульного рейда вертолет США, "спутав цель", нанес удар по бронемаши-

нам, принадлежащим ВВС США. Ранения получили пять человек — двое американских военнослужащих и трое мирных иракцев".

Во время войны в Ираке в США появился термин "дружественный огонь" (friendly fire). До 24 % всех потерь живой силы в 1-й Иракской войне произошло из-за этой причины. Выводы многочисленных комиссий по расследованию подобных инцидентов однозначны: главная причина "дружественного огня" — все возрастающая сложность технологий XXI века, используемых в военных системах, и ошибки в программном и аппаратном обеспечении автоматического управления в военных системах.

К этой же теме относится заметка в PC Week от 28.04.2008 г., озаглавленная так: "*Боевой робот взбунтовался против хозяев*": "В Ираке проводился эксперимент по эксплуатации усовершенствованных моделей боевых роботов "Talon", оснащённых пулемётом M249 и управляемых дистанционно. Один из ожидавших приказа аппаратов по непонятной причине "ожил" и стал самостоятельно шевелить манипулятором, в котором было установлено заряженное оружие".

- 2 сентября 1988 г. на борт советской межпланетной станции "Фобос-1", отправленной на Марс, была передана с Земли неверная команда. Бортовым компьютером эта команда была воспринята как команда на отключение системы ориентации и стабилизации. Эта ошибка бортовой системы управления, которая обязана выявлять некорректные внешние команды и должным образом реагировать на них, привела к тому, что станция "Фобос-1" потерялась в космосе. Через полгода была потеряна связь и с ее дублером — станцией "Фобос-2". Наиболее вероятная причина этой второй неудачи — также программная ошибка. В результате просторы Вселенной сейчас бороздят безмолвные, бесполезные для человечества советские станции "Фобос-1" и "Фобос-2".

Приведенные примеры показывают, что ошибки в сложных программных и аппаратных системах не являются чем-то исключительным, они повторяются регулярно в разработках сложных систем. Их непосредственными причинами являются и некорректные спецификации, и неправильное понимание спецификаций разработчиками, взаимное непонимание в коллективе разработчиков, непредвиденные события, режимы и условия работы, несогласованность параллельных ветвей программ и многое другое.

В наше время, когда все чаще человеческая жизнь зависит от функционирования автоматических систем, проблема гарантии правильности программных и аппаратных компонентов этих систем приобретает первостепенное значение. Надежность и предсказуемость поведения таких систем являются более важными свойствами, чем производительность, модифицируемость и т. п. Верификация, как один из основных методов повышения качества систем, становится важнейшей областью информатики.

1.2. Примеры ошибочных программ

Параллельные и распределенные программы становятся все более распространенными. В современном автомобиле существует целая сеть связанных микропроцессоров, в новых "мерседесах" их около 50. Производители соревнуются в выпуске многоядерных процессоров для персональных компьютеров, но реально их возросшая вычислительная мощность может быть использована только при программировании параллельно выполняемых процессов. В [136] происходящий сейчас поворот программирования в сторону параллелизма назван "революцией" с лозунгом: "Добро пожаловать в мир параллельной обработки". Но именно этот мир чреват проблемами, возникающими вследствие необходимости синхронизации параллельных процессов.

В мире параллельной обработки разработчик должен контролировать не последовательности возможных событий, как в последовательном программировании, а возможные комбинации частично упорядоченных событий, что значительно сложнее. Многочисленные примеры показывают, что нетривиальные многопоточные программы непостижимы для человеческого мозга. С ошибками, возможными в параллельных программах, может сегодня столкнуться каждый. В качестве примера трудновыявляемых ошибок в параллельных программах приведем несколько простых программ.

Пример 1.1

В период обучения в вузе студенты часто работают. Студент A поступил на работу в компанию SoftTech и разработал бухгалтерскую программу БухСофт, которую его компания использует сама и продает. Студент открыл депозитный банковский счет X_A, с начальным капиталом 0, на который бухгалтерская программа $БухСофт1$ компании SoftTech должна перевести ему 3000 долларов за эту работу.

Поскольку студент еще и учится, ему выплачивается стипендия. Как троечнику, ему полагается стипендия всего в 15 долларов. Бухгалтерская программа БухСофт2 университета в конце месяца должна увеличить счет этого студента на 15 долларов.

Таким образом, в двух независимых процессорах два параллельных процесса независимо выполняют операции над одним элементом общей памяти:

$$БухСофт1 :: X_A := X_A + 3000 \qquad БухСофт2 :: X_A := X_A + 15$$

К концу месяца студент рассчитывает, что после двух выплат у него на счете будет 3015 долларов, достаточные для покупки подержанной иномарки. К его

разочарованию, на счете оказалось только 15 долларов, хотя обе программы отработали и, по распечаткам, все деньги ему переведены.

Причиной некорректности является ошибка в разработанной студентом программе: отсутствие синхронизации двух параллельных процессов, выполняющих одновременный доступ к общим данным. Подобные ошибки типичны в программах, разработанных студентами-недоучками, пропускавшими лекции по теории параллельных процессов. Хотя операция начисления зарплаты записывается как $X_A := X_A + C$, реально в каждом процессе она выполняется как три неделимых операции:

1. Чтение в регистр значения из памяти.
2. Увеличение значения регистра.
3. Запись значения из регистра в память.

Если процесс *БухСофт*1 выполняет все три свои операции либо ДО, либо ПОСЛЕ того, как процесс *БухСофт*2 выполнит все свои операции, то значение X_A в результате увеличится на $C_1 + C_2$, т. е. программы будут работать правильно. Существует, однако, очень малая вероятность того, что эти две группы операций будут выполняться независимо работающими процессами приблизительно в одно время. В этом случае асинхронное параллельное выполнение независимых процессов может привести к тому, что неделимые микрокоманды разных процессов будут выполняться по очереди в некотором порядке. В итоге мы можем получить неожиданный результат: одно из изменений значения общей переменной, которое выполняет либо один, либо другой процесс, будет потеряно.

Таким образом, бухгалтерская программа, разработанная студентом, работает корректно "почти" все время, но иногда, очень редко, программа будет работать неправильно. Выявить подобные тонкие ошибки синхронизации параллельных процессов никаким тестированием невозможно, они зависят от относительных скоростей процессов, времени начала выполнения операций и т. п.

Пример 1.2

Программа разделения множеств [165]. Эта простая программа состоит из двух параллельно функционирующих взаимодействующих процессов, `Small` и `Large`. Процесс `Small` оперирует на конечном множестве чисел `S`, процесс `Large` оперирует на конечном множестве `L`. Процессы `Small` и `Large` взаимодействуют посредством посылки сообщений, обмениваясь числами из этих множеств по каналу `c1` от `Small` к `Large` и по каналу `c2` от `Large` к `Small` до тех пор, пока в `S` не соберутся все минимальные значения обоих множеств, а в `L` не соберутся максимальные значения множеств.

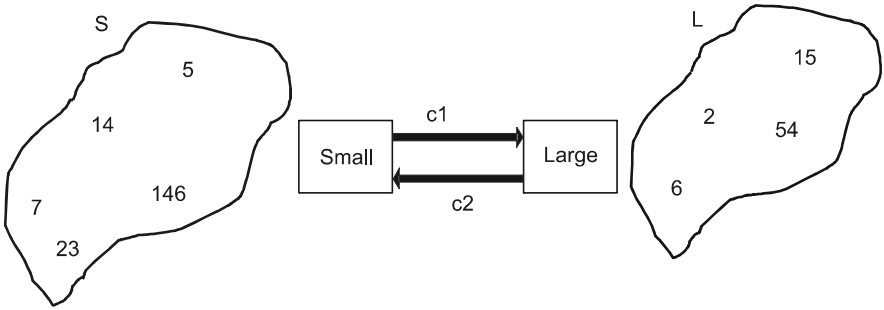


Рис. 1.1. Параллельная программа разделения множеств

Процесс `Small` на каждом шаге ищет максимальное значение в множестве `S` и посылает его процессу `Large` по каналу `c1`, затем принимает от `Large` найденное этим процессом минимальное значение `L`, посланное по каналу `c2`:

```
Small::
begin
  mx:=max(S); c1!mx; S:=S-{max(S)};
  c2?x; S:=S∪{x}; mx:=max(S);
  *[mx>x →    // цикл выполняется, пока mx>x
          c1!mx; S:=S-{max(S)};
          c2?x; S:=S∪{x}; mx:=max(S);
   ]
end
```

Процесс `Large` на каждом шаге принимает от `Small` найденное им максимальное значение `S`, посланное по каналу `c2`, сам ищет минимальное значение во множестве `L` и посылает его процессу `Small` по каналу `c1`:

```
Large::
begin
  c1?y; L:=L∪{y}; mn:=min(L);
  c2!mn; L:=L-{mn}; mn:=min(L);
  *[mn<y →    // цикл выполняется, пока mn<y
          c1?y; L:=L∪{y}; mn:=min(L);
          c2!mn; L:=L-{mn}; mn:=min(L)
   ]
end
```

Взаимодействие по каналам `c1` и `c2` очень простое — это синхронная коммуникация (хэндшейк, рандеву), при которой посылка одним процессом данных в канал может быть выполнена в том и только в том случае, если другой процесс готов прочесть эти данные.

Почти на всех вариантах исходных данных эта параллельная программа работает верно: процессы завершаются, по их завершении все данные сохраняются, число значений в каждом множестве не изменяется, каждый элемент S не больше каждого элемента L. Очень редко при некотором вполне определенном соотношении между значениями элементов множеств S и L эта параллельная программа работает некорректно: один из процессов будет бесконечно ожидать коммуникации с другим, который до этого благополучно завершился. Правильность этой программы нельзя доказать тестированием: почти на всех наборах исходных данных эта программа работает правильно. Только строгий формальный анализ, верификация программы может обнаружить ошибку.

Пример 1.3

Протокол W.C.Lynch [105]. Этот протокол был приведен Уильямом Линчем как пример убедительно выглядящей, но некорректной распределенной программы, опубликованной "одним из основных производителей вычислительной техники". Цель протокола — полнодуплексная передача (т. е. одновременная передача в обоих направлениях) последовательностей символов через ненадежную среду между пользователями A и B.

Каждое сообщение имеет две части: информационный символ и управляющую часть. Управляющая часть принятого сообщения всегда содержит один из трех управляющих символов: ack, nack или err. Если сообщение принято без ошибок, оно содержит ack или nack в управляющей части. Канал может искажать сообщения, но не может терять, дублировать или вставлять новые. Предполагается, что все ошибки передачи обнаруживаются нижним уровнем протокола, верхний уровень при этом получает символ err в управляющей части сообщения.

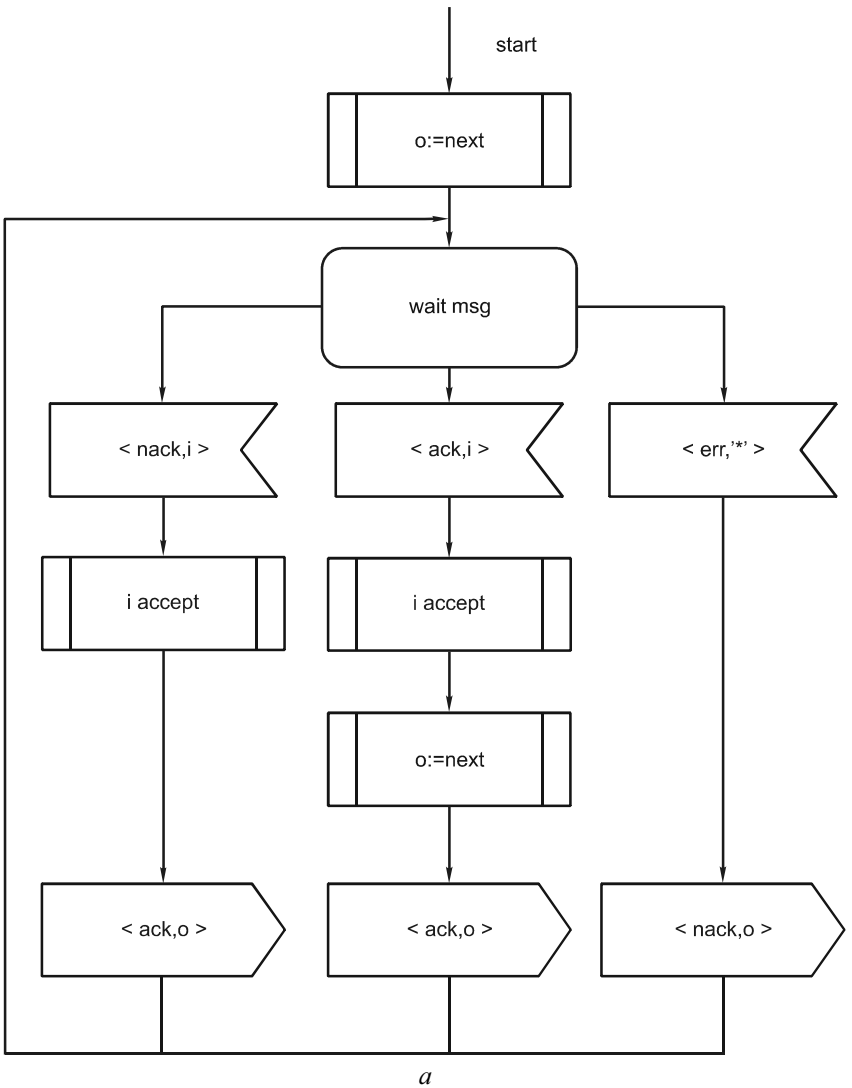
Правила функционирования протокола описываются его разработчиком так:

"В ответном сообщении в управляющей его части посылается положительное подтверждение ack, если полученное перед этим сообщение не содержало ошибок, в ответ посылается отрицательное подтверждение nack, если было принято ошибочное сообщение.

Если полученное сообщение несло отрицательное подтверждение или само оказалось ошибочным, то в ответ посылаются старые данные, в противном случае посылаются следующие данные".

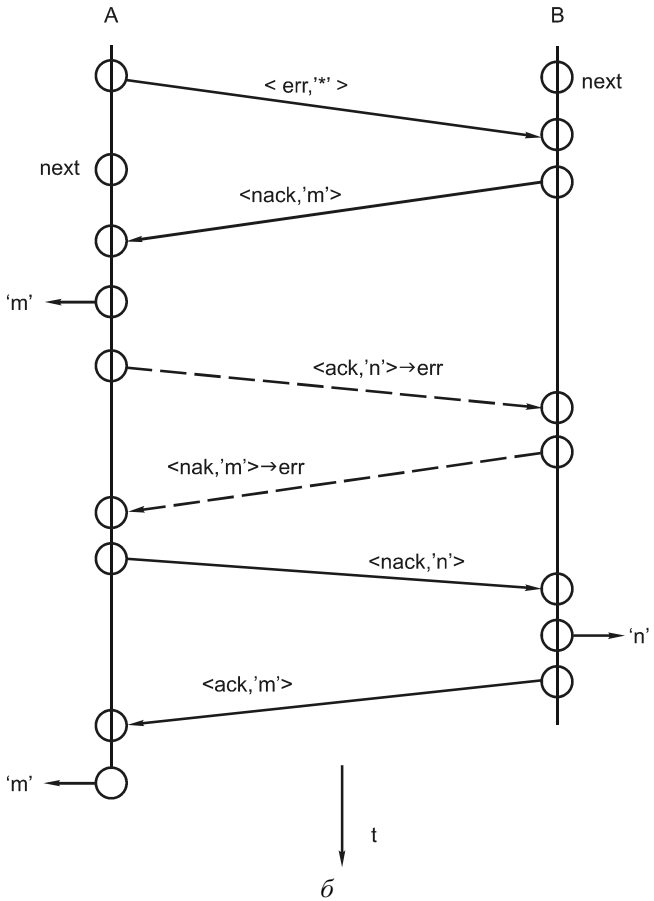
Формализованное представление этих правил в стиле SDL дано на рис. 1.2, *а*. Схема полностью соответствует правилам: полученное новое сообщение msg может иметь три возможных типа (три разных значения в управляющей части). Если пришло неискаженное сообщение, которое содержит "nack" в управляющей части, то в ответном сообщении посылается положительное

подтверждение и повторяется ранее переданный символ "o". Если принятое сообщение не искажено и оно содержит 'ack' в управляющей части, то ответ в "o" выбирается новый символ для передачи, и он посылается с положительным подтверждением "ack". Если полученное сообщение искажено, то в ответном сообщении повторно посылаются старые данные (старый информационный символ) и отрицательное подтверждение "nack" получения информации в предыдущем сообщении.



а

Рис. 1.2, *а*. Ошибка в протоколе W.C.Lynch



б

Рис. 1.2, *б*. Ошибка в протоколе W.C.Lynch

Правила функционирования выглядят абсолютно корректно, однако этот протокол содержит несколько ошибок.

Во-первых, если нет данных для передачи в одном из направлений, то передача и в другом направлении останавливается. Эту ошибку можно преодолеть, передавая какую-нибудь пустую информацию, если нет содержательной информации.

Во-вторых, протокол имеет проблемы с инициализацией: после старта на обоих концах процессы будут ожидать сообщения от партнера, т. е. с первого шага протокол блокируется. Аналогичные проблемы существуют с завершением передачи: как оба процесса могут договориться о прекращении обмена? Очевидно, что необходимы специальные правила инициализации и завершения передачи, которые сами могут оказаться непростыми и содержать ошибки.

В-третьих, протокол содержит серьезную принципиальную ошибку: при некоторых редко встречающихся сценариях протокол может ошибочно принять дубликат предыдущих данных как очередную информацию. Один из таких сценариев приведен на рис. 1.2, *б*: процесс A передает процессу B сообщение n, в противоположном направлении процесс B передает процессу A сообщение m. Двойная повторяющаяся ошибка в канале, искажающая как положительное подтверждение от A к B, так и следующее сообщение, содержащее отрицательное подтверждение от B к A, приводит к дублированию принятия сообщения m процессом A. Таким образом, протокол работает правильно "почти всегда", эта логическая ошибка проявляется при редких сочетаниях условий передачи. Исправление этой ошибки требует пересмотра всей логики протокола.

Существуют ли другие ошибочные сценарии поведения этого протокола? Ни тщательные просмотры текста, ни тестирование не могут гарантировать отсутствие ошибок в этом протоколе.

Итак, несмотря на простоту и очевидность правил передачи сообщений в этом протоколе, в нем содержатся тонкие логические ошибки, которые очень непросто обнаружить, и не совсем понятно, как их исправлять. Как указывает У. Линч, "подобные ошибки хотя и очень редко, но случаются, и именно их редкость делает их обнаружение в работающей системе чрезвычайно трудным. Такой неадекватный алгоритм будет работать правильно “почти все время”".

Поскольку подобные ошибки в параллельных системах выявить тестированием нельзя, единственным средством их обнаружения является верификация. Все серьезные разработчики следуют правилу: "любая параллельная программа должна рассматриваться как некорректная до тех пор, *пока не доказано обратное*".

1.3. Общая схема верификации. Проверка моделей (model checking)

Стандарт ИСО 9000:2000 определяет процессы верификации и валидации изготавливаемого продукта следующим образом:

- *Верификация* — подтверждение на основе представления объективных свидетельств того, что установленные требования были выполнены.
- *Валидация* — подтверждение на основе представления объективных свидетельств того, что требования, предназначенные для конкретного использования или применения, выполнены.

Хотя эти два определения, как и сами термины, кажутся почти одинаковыми, в них есть существенное различие:

- *Верификация* (от лат. *verus* — верный) — это проверка того, что продукт удовлетворяет сформулированным требованиям.
- *Валидация* (от лат. *validus* — здравый) — это проверка того, что разработано именно то, что требуется заказчику. Обычно тестирование и моделирование разработанной системы рассматривается как часть валидации.

В частности, если для анализа продукта мы используем формальные модели, то заключение о том, что выбранная модель адекватно представляет проверяемый продукт (разработанную систему) относится к процессу валидации. Заключение о том, что набор неформальных требований достаточно полно отражает желаемый уровень доверия к системе, и что формально определенные требования к поведению системы действительно отражают свойства системы, важные с точки зрения ее применения, также относится к процессу валидации. Сама же *процедура проверки* выполнения формальных требований на модели системы является процессом верификации.

Мы будем заниматься формальной верификацией программных систем.

> ❖ ***Определение 1.1***
>
> *Формальная верификация программ* — это приемы и методы формального доказательства (или опровержения) того, что модель программной системы удовлетворяет заданной формальной спецификации.

Поскольку что-то доказать формально можно только относительно формальной модели, анализируемая система (реализация) должна быть представлена для верификации формальной моделью. Программы обычно пишутся на языках с формально определенным синтаксисом и кажутся полностью формализованными. Однако с точки зрения семантики это не так. Программная система обычно состоит из программных модулей, написанных на языках с неформализованной семантикой. Использование указателей, обработка вещественных чисел с ограниченной точностью, сложные структуры данных, динамическое порождение потоков и их взаимодействие — все это приводит к тому, что из текста программы не следует непосредственно полное формальное описание ее поведения. В частности, не существует алгоритма, позволяющего проверить эквивалентность двух программ на языке Паскаль или С. Формальная модель системы обычно строится вручную. Такая модель, как правило, значительно проще самой проверяемой системы, это абстракция, в которой должны быть отражены наиболее существенные характеристики системы.

Спецификация системы также должна быть представлена формально, набором формальных требований к ее функционированию. Но спецификации программы обычно не являются формальными. Требования к конечному продукту чаще всего расплывчаты, двусмысленны, неполны. Для выполнения верификации спецификация программы должна состоять из набора формальных утверждений о желаемых свойствах поведения системы. Для формальной спецификации используется язык логики, каждое утверждение которого может быть либо истинно, либо ложно для проверяемой программной системы. Верификатор проверяет выполнение заданных формальных утверждений на заданной формальной модели.

На рис. 1.3 представлена общая схема верификации. Компоненты этой схемы должны удовлетворять очевидным требованиям. Желательно, чтобы модель, представляющая систему, была достаточно выразительной, чтобы с ее помощью можно было представить поведение параллельных процессов и их

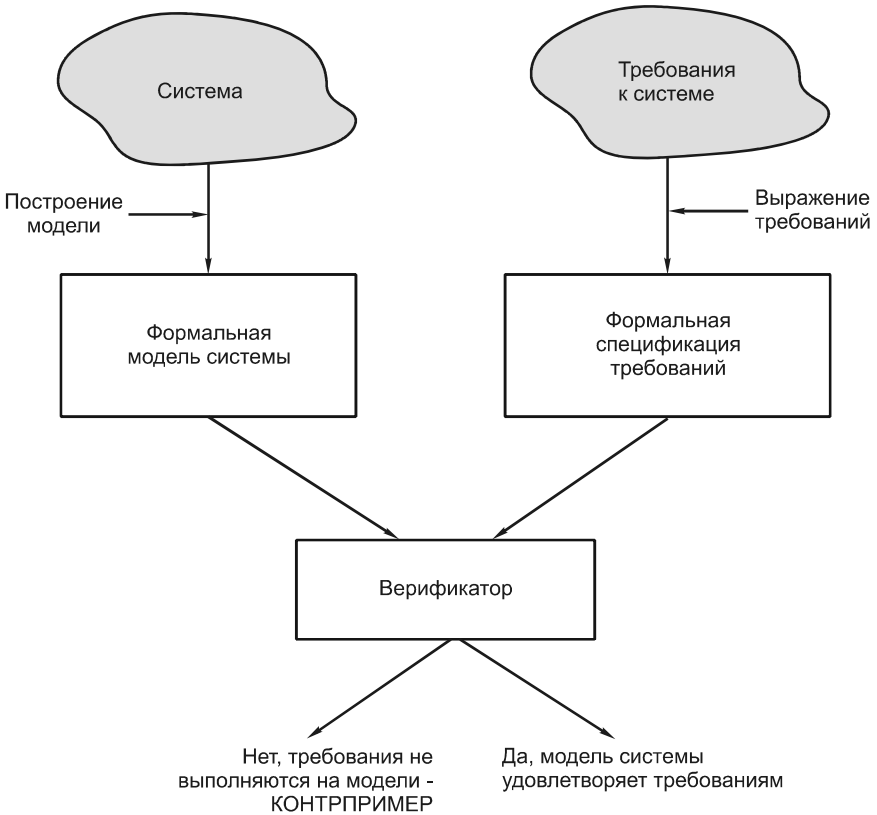


Рис. 1.3. Общая схема верификации

взаимодействие. Для систем реального времени важным моментом является возможность включения в модель временных ограничений, для стохастических систем — возможность учета вероятностей событий. Язык спецификации для формулировки требований к поведению системы должен быть выразительным, мощным, интуитивно понятным, позволяющим достаточно просто выразить широкий класс важных свойств поведения системы. Желательно, чтобы сам алгоритм верификации был эффективным, позволяющим автоматически выполнять проверку свойств систем, разрабатываемых для практического применения. Далее, если система удовлетворяет спецификации, верификатор должен выдать подтверждение этого. Если система не удовлетворяет спецификации, то получение простого ответа "нет" малопродуктивно. Желательно, чтобы верификатор в этом случае предоставлял контрпример — ту траекторию функционирования системы, на которой проверяемое свойство не выполняется. Такая информация чрезвычайно важна для исправления возможных ошибок.

В области верификации программ в течение нескольких десятилетий предпринимались серьезные усилия по разработке теории, алгоритмов и техники верификации. В настоящее время эти методы разрабатываются в трех основных направлениях:

- дедуктивная верификация;
- проверка эквивалентности;
- model checking (проверка модели).

Дедуктивная верификация — это проверка правильности программы, которая сводится к доказательству теорем в подходящей логической системе. Это направление разрабатывается уже более 40 лет, с тех пор, как Роберт Флойд и Энтони Хоар впервые формально поставили проблему доказательства корректности программ. Верификация программ здесь проводится дедукцией на основе аксиом и правил вывода. Эта весьма сложная процедура не может быть полностью автоматизирована, она требует участия человека, действующего на основе предположений и догадок, использующего интуицию при построении инвариантов и нетривиальном выборе альтернатив.

Проверка эквивалентности связана с разработкой формальных моделей взаимодействующих процессов, выражением в рамках таких моделей как спецификации, так и реализации, и проверкой эквивалентности поведений формально определенных моделей поведения. Это направление было начато работами Робина Милнера разработкой им алгебры процессов *Исчисление взаимодействующих систем* (Calculus of Communicating Systems, CCS) около 30 лет назад.

Метод model checking связан с формальной проверкой выполнения на модели реализации свойств поведения, специфицированных на языке формальной логики. Этот метод разрабатывался более 20 лет. Метод model checking может быть полностью автоматизирован.

В каждом из этих направлений в течение десятилетий исследований были получены многообещающие теоретические результаты, которые явились серьезным вкладом в теорию вычислений. Однако еще совсем недавно уровень развития верификации не позволял использовать ее как этап технологии для разрабатываемых на практике систем. Разработанные методы не могли быть применены даже к системам средней сложности, инструментальная поддержка процесса верификации была часто неадекватна и сложна в использовании.

В этой книге рассказывается о методе формальной верификации model checking, исследования которого привели в последнее время к разработке очень эффективных алгоритмов верификации, позволяющих проверять реальные, разрабатываемые промышленностью программно-аппаратные системы. *Model checking (проверка модели)* — это метод проверки того, что на данной формальной модели системы заданная логическая формула выполняется (принимает истинное значение). Хотя это определение справедливо для любой логики и любого класса формальных моделей, исторически впервые эта техника была разработана для моделей систем переходов и так называемых "темпоральных" логик, которые мы будем подробно изучать в следующей главе.

Для верификации этим методом реальная система представляется простейшей моделью ее вычислений — системой переходов с конечным числом состояний, некоторой разновидностью конечного автомата. Требуемые свойства поведения реальной системы выражаются точно и недвусмысленно формулами темпоральной логики. Проверка модели сводится к исчерпывающему анализу всего пространства состояний модели системы, она не требует дедуктивных доказательств теорем, связанных с логическим выводом на основе аксиом и правил вывода, характерных для классических подходов к верификации. Поэтому этот процесс может быть полностью автоматизирован (т. е. может выполняться без участия человека, с использованием так называемой "*технологии нажатия кнопки*", push-button technology).

Обсудим происхождение термина model checking.

В формальной логике *интерпретацией* логической формулы Φ называется объект, на котором формула принимает либо истинное, либо ложное значение. Те интерпретации формулы Φ, на которых Φ принимает истинное значение, называются МОДЕЛЯМИ логической формулы Φ. Например, в *логике высказываний* множеством интерпретаций формулы $\Phi(p,q,r)$ являются

наборы истинностных значений атомарных (элементарных) высказываний p,q,r, например, $< p = true, q = false, r = true >$. Те наборы истинностных значений $< p,q,r >$, на которых Φ истинна, являются МОДЕЛЯМИ формулы Φ.

Для темпоральной логики, используемой для спецификации свойств поведения систем, множеством интерпретаций ее формул является множество систем переходов: на каждой системе переходов M любая темпоральная формула Φ принимает либо истинное, либо ложное значение. Те системы переходов, для которых темпоральная формула Φ истинна, являются МОДЕЛЯМИ формулы Φ.

❖ *Определение 1.2*

Model checking (буквально: проверка модели) — это алгоритм, проверяющий, будет ли заданная логическая формула Φ истинна для данной системы переходов M, т. е. будет ли M моделью Φ (рис. 1.4).

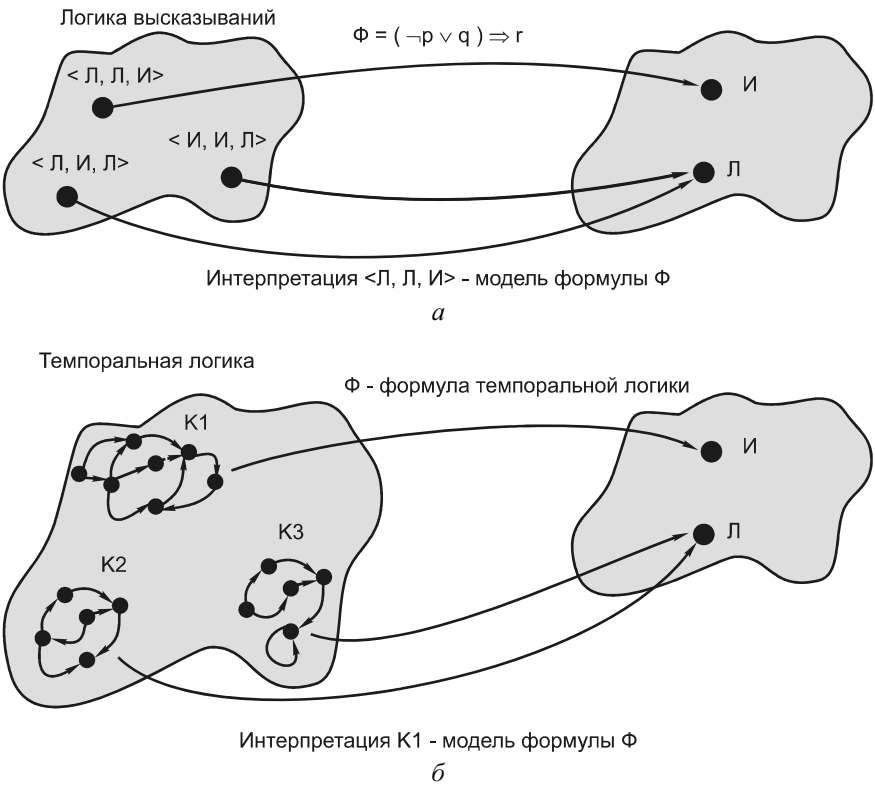


Рис. 1.4. Системы переходов как модели темпоральных формул

Поскольку система переходов является *моделью* верифицируемой системы (протокола, программы, схемы и т. п.), то часто термин model checking понимают как "*проверка на модели*". В литературе можно встретить как ту, так и другую трактовку, хотя исследователи, впервые разработавшие алгоритм model checking, имели в виду первую.

Для выполнения верификации методом model checking нужно построить для программной системы адекватную модель с конечным числом состояний, представить ее на входном языке автоматизированной системы верификации, а также выразить подлежащие проверке свойства формулами темпоральной логики. Поэтому, несмотря на то, что сам процесс верификации полностью автоматизирован, выполнение верификации программных и аппаратных систем требует работы специалиста достаточно высокой квалификации.

Одной из главных проблем этого подхода является огромное число состояний моделей реальных систем. Реальные системы обычно параллельны, а число состояний моделей параллельной системы растет экспоненциально с ростом числа компонентов. Возникающая при этом проблема называется проблемой *взрыва числа состояний* (state explosion problem). То, что метод model checking сегодня успешно применяется к разрабатываемым реальным системам, является следствием того, что были созданы методы эффективного представления данных (так называемые символьные методы) и соответствующие алгоритмы верификации, позволяющие в значительной степени решить проблему взрыва числа состояний и выполнять методом model checking верификацию систем с громадным числом состояний — 10^{100} и более.

Всем этим вопросам посвящены следующие разделы книги.

1.4. Тестирование и верификация

Наиболее очевидным и широко распространенным методом проверки правильности систем является тестирование — проверка работы построенной системы в различных реальных ситуациях, при различных исходных данных. Тестирование имеет множество преимуществ:

- проверяется реальная система, а не ее модель;
- проверка может быть выполнена в реальной среде с реальными интерфейсами;
- проверять можно реальные наиболее опасные или часто используемые режимы работы системы.

В то же время у тестирования есть и недостатки:

- тестирование очень трудоемкий процесс;

- тестирование выполняется на поздних этапах разработки системы, когда модули системы уже реализованы, поэтому исправление найденных ошибок очень дорого, при исправлении часто вносятся другие ошибки;
- все реакции системы при выполнении тестов должны быть заранее зафиксированы;
- тестированием можно проверить лишь очень немногие траектории вычисления системы (а их обычно бесконечное число);
- тестирование сложных систем трудно автоматизируется, обычно тестирование выполняется с помощью низкоквалифицированных программистов-тестеров, сопоставляющих реакцию тестируемой системы с желаемой реакцией;
- тестирование не может гарантировать правильность системы: знаменитый тезис Э. Дейкстры "*тестированием можно доказать только наличие ошибок*" говорит о том, что никаким количеством тестов нельзя установить правильность работы программной системы в любых условиях;
- тестированием невозможно выявить редко проявляющиеся ошибки, особенно ошибки в системах реального времени и ошибки в параллельных системах, которые могут зависеть от соотношения скоростей процессов, редко встречающихся ситуаций и поэтому невоспроизводимы.

В отличие от тестирования, верификация позволяет проверить выполнение требований спецификации на всех возможных траекториях функционирования системы. Верификация проводится на самых ранних этапах проектирования по заранее разработанной модели, что существенно экономит время и затраты на исправление ошибок. Существующие методы верификации достаточно эффективны, они поддержаны системами автоматического выполнения. Однако и у верификации есть ряд недостатков:

- главным недостатком верификации является то, что проверяется не реальная система, а ее абстрактная модель. Такая модель может быть неадекватной, не сохранять существенных черт исходной системы, и в этом случае проверка спецификации для модели не выявит ошибок, имеющихся в реальной системе. С другой стороны, в модели могут появиться свойства, которых в реальной системе не существует. Таким образом, ценность результатов, полученных при верификации, напрямую зависит от адекватности модели и, следовательно, от уровня профессионализма человека, строящего модель;
- язык спецификации свойств системы может быть неполным, недостаточным для формулировки всех желаемых требований к поведению системы. Выполнение любого множества проверок системы в этом случае не гарантирует ее правильности;

- сама автоматизированная система верификации может содержать ошибки;
- поскольку невозможно формально определить, что означает "*полное отсутствие ошибок*", верификация так же не может гарантировать абсолютной правильности системы, как и тестирование;
- несмотря на то, что существенная часть процесса верификации обычно автоматизирована, выполнение верификации требует достаточно высокой квалификации персонала. Построение адекватных моделей реальных систем является весьма непростой задачей. Иногда модель, для которой проводится верификация, допускает такие траектории вычисления, которые не могут произойти в реальной системе. Исключение таких траекторий из процесса анализа системы требует глубокого понимания процесса абстрагирования. Формулировка требований на языке спецификаций требует знаний логики (обычно это темпоральная логика), свойств параллельных систем и способов выражения этих свойств на языке логики.

Общепризнано, что как тестирование, так и верификация по отдельности не могут гарантировать достаточного уровня правильности разрабатываемых систем. Существует большое число примеров, когда в тщательно проверенных и оттестированных реализациях (в частности, в стандартах протоколов) с помощью верификации впоследствии обнаруживались тонкие ошибки. Например, никакое тестирование приведенной выше некорректной параллельной программы разделения множеств не может гарантировать выявление ошибки. Формальный анализ, проведенный в [165], не только позволил обнаружить в этой программе ошибку, но и понять ее причину.

С другой стороны, нельзя надеяться только на верификацию. Например, в работе [164] представлен анализ с помощью тестирования и имитационного моделирования распределенного алгоритма обнаружения дедлоков в системе управления транзакциями распределенной базы данных. Этот алгоритм оказался ошибочным, хотя его корректность была "доказана" в работе [76]. Причина ошибок в доказательстве состояла в том, что и при разработке, и при доказательстве этого алгоритма неявно делались неправильные предположения о характере его работы, т. е. использовались неадекватные формальные модели.

Таким образом, как тестирование, так и верификация обладают своими преимуществами и недостатками, поэтому эти подходы можно считать взаимно дополняющими. Для повышения надежности реализаций при разработке систем управления, программных и аппаратных систем должны применяться оба подхода.

1.5. Примеры успешной верификации систем

Методы верификации, объединенные под общим названием model checking, быстро прогрессируют. Еще совсем недавно, в книге C. Ghezzi, M. Jazayeri, D. Mandrioli "Fundamentals of Software Engineering", изданной в 2003 г., метод model checking характеризовался как "новый метод верификации, который позволяет получить доказательство корректности в некоторых простых случаях". Однако уже сегодня этот подход к верификации изменил технологию промышленной разработки ПО.

Применение методов формальной верификации как этапа в промышленной технологии разработки программных систем началось в конце 90-х годов прошлого века именно с метода model checking, который оказался чрезвычайно удобным и эффективным (cost effective) на практике. Приведем несколько примеров.

В работе [21] на основе опыта использования верификации в двух разработках аппаратуры (FIFO модуль памяти и маркерный контроллер) был сделан вывод о возможности промышленного использования этой техники для повышения уровня доверия к разработанным продуктам.

В работе [80] Дж. Холзманн (Gerard Holzmann) перечисляет ряд значительных результатов, которые к тому времени (более 10 лет назад) были достигнуты в области верификации реальных систем с помощью системы верификации Spin. Среди представленных примеров — Cambridge ring protocol, IEEE Logical Link Control protocol, LLC 802.2, фрагменты больших протоколов XTP и TCP/IP. Другие примеры успешной верификации — контроллеры, отказоустойчивые системы, протоколы доступа к шинам, протоколы контроля ошибок в аппаратуре, криптографические протоколы, протокол Ethernet Collision Avoidance, протоколы самостабилизации. Монография [113] приводит примеры верификации систем с более чем 10^{100} состояний с помощью символьных алгоритмов системы верификации SMV. После завершения одного из проектов в компании Lockheed Martin было отмечено: "Формальная верификация позволила найти несколько тонких ошибок, которые, скорее всего, были бы упущены в процессе традиционного тестирования".

В отчете 2002 г. компании Lucent Technologies рассказывается об опыте использования корпоративного верификатора VeriSoft для систематической верификации библиотеки обработки вызовов протокола CDMA для беспроводных коммуникаций. Вывод отчета заключается в том, что традиционное тестирование приносит мало пользы при разработке такого сорта ПО, позволяя проверить только ничтожную долю поведений системы. В отличие от тестирования, верификация позволила здесь систематически проанализировать

полное пространство состояний системы, найти существенные ошибки разработки в нескольких версиях продукта. Возможность нахождения ошибок на ранних стадиях разработки позволила компании сэкономить значительные средства. Авторы пишут, что стоимость исправления одной ошибки, оставшейся в их продукте после разработки, составляла бы сотни тысяч долларов. В [144] приведены результаты верификации стандартного арбитра шины Futurebus.

Путь новых методов верификации к признанию в промышленности не был прост. Например, для того чтобы убедить скептиков-разработчиков бортовых систем в необходимости использования верификации как важнейшего этапа технологии разработки, группа из Исследовательского центра Ames корпорации NASA выполнила специальное исследование [67]. В его рамках был проведен формальный анализ с помощью системы верификации Spin многопроцессного ПО бортовой системы управления космическим аппаратом в проекте DeepSpace 1 для полета к Марсу. Было обнаружено множество ошибок разработки, не выявленных методами, использующимися в стандартной технологии разработки ПО. По признанию разработчиков, четыре из этих ошибок оказались критическими. Они могли проявляться очень редко, но их возникновение привело бы к отказу системы (mission-killer errors). Все они были не примитивными опиской или логическими ошибками, характерными для последовательных программ. Эти ошибки проявлялись лишь в результате непредвиденных разработчиками перекрытий операций параллельно протекающих процессов. В центре Ames продолжаются интенсивные исследования в этой области [55].

Результаты подобного исследования были представлены недавно фирмой Rockwell Collins, мировым лидером в разработке авиационной электроники, мобильных и бортовых коммуникационных систем для коммерческих и военных применений [108]. Годовой объем продаж компанией этих систем составляет несколько миллиардов долларов. Проект, целью которого был анализ эффективности формальной верификации в промышленных разработках компании, закончился в 2007 г., и его результатом стали революционные изменения в подходе компании к созданию сложных встроенных систем. В Центре передовых технологий компании был организован отдел, использующий формальные инструменты и методы верификации при разработке продуктов, требующих высокого уровня уверенности в их правильном функционировании. В этой компании этап верификации стал частью промышленной технологии разработки бортовых систем для критических применений.

Множество компаний убедилось в невозможности разработки качественных программ без использования верификации и включило этап верификации в технологический процесс разработки систем. Все основные производители программных и аппаратных систем: Intel, IBM, Nortel, Motorola, HP, Cadence,

NEC, Bell Labs, Siemens, Lucent Technologies, Sun и многие другие имеют в своем составе группы, выполняющие верификацию проектов. Несколько лет назад компания Intel даже объявляла о приеме на работу каждого, закончившего обучение по программе Ph.D. в области формальной верификации.

1.6. Инструменты верификации

Многие крупные компании-производители схем и встроенных систем разработали свои системы верификации, которые встроены в технологический производственный цикл. На рынке существует несколько коммерческих систем верификации, поддерживаемых разработчиками, например, Chrysalis, Synopsys, Verysys, Cadence. Разработанный фирмой Microsoft верификатор Static Driver Verifier поставляется вместе с системой поддержки разработки драйверов для ОС Windows (Windows Driver Development Kit). Построенный верификатор может использоваться каждым разработчиком драйверов для этой ОС. Эта работа была отмечена Биллом Гейтсом в его торжественной речи на открытии конференции OOPSLA в 2002 г. как серьезное достижение компании.

Существуют также системы верификации, распространяемые свободно. Они разработаны с исследовательскими целями, главным образом, в университетах. Это система SMV, разработанная в университете Карнеги-Меллон, система UPPAAL, разработанная и совместно поддерживаемая университетами Уппсала и Аалборга, система PRISM, разработанная в Бирмингемском университете, система Spin, разработанная в Bell Labs, и многие другие. Примеры верификации в данной книге будут приведены на языке Promela — входном языке инструментальной системы верификации Spin, разработанной Джерардом Холзманном, работавшем в то время в Bell Labs. Система Spin достаточно проста, она используется во многих университетах мира для обучения студентов анализу параллельных процессов и практическим методам верификации. В то же время с помощью этой системы были верифицированы используемые на практике протоколы коммуникации, стандарты протоколов, модули операционных систем, бортовые системы управления.

Приведем в заключение ссылки на некоторые свободно распространяемые инструменты верификации.

- SPIN — Bell Labs — (параллельные взаимодействующие процессы)

  http://cm.bell-labs.com/cm/cs/what/spin/

- UPPAAL — Uppsala University и Aalborg University (системы реального времени)

  http://www.docs.uu.se/docs/rtmv/uppaal/

- VIS — Uni Berkeley, Uni Colorado
  http://vlsi.colorado.edu/~vis/
- KRONOS — INRIA (системы реального времени)
  http://www.inrialpes.fr/vasy/cadp/software/99-c-kronos.html
- HYTECH — Cornell University (линейные гибридные системы)
  http://www.henzinger.com/monika/hytech.html
- SMV — Carnegie Mellon University (Symbolic Model Verificator)
  http://www.cs.cmu.edu/~modelcheck/smv.html
- STeP — Stanford University (Stanford Temporal Prover)
  http://www-step.stanford.edu/

1.7. Заключение

Существует два основных подхода к проверке правильности систем: тестирование и верификация. Тестирование заключается в проверке реальной разработанной системы в ее реальных режимах работы. Верификация заключается в формальной проверке того, что для модели системы выполняются формально определенные свойства. Тестирование исследует *некоторые поведения реальной системы,* верификация анализирует *все поведения абстрактной модели системы.*

Основным преимуществом тестирования является то, что тестируется реальная система в реальных (обычно критических) режимах ее работы. Основным недостатком тестирования является то, что с помощью тестирования могут быть проверены только некоторые из бесконечного числа режимов работы системы, и отсутствия ошибок в сложной системе никаким тестированием доказать нельзя. Тестирование в принципе не может *доказать* наличие какого-либо свойства у системы.

Преимуществом верификации является то, что проводится *исчерпывающая* проверка всех возможных режимов работы модели системы, в том числе и тех, которые в реальном функционировании маловероятны, а потому и не могут быть проверены тестированием. Этот анализ проводится на ранних этапах разработки. Результатом верификации может явиться вывод о том, что система не имеет ошибок некоторого типа или обладает некоторым определенным свойством. Основным недостатком верификации является то, что проверяется не реальная система, а ее модель, которая может и не отражать существенных свойств поведения реальной системы. Далее, поскольку невозможно формально определить, что означает "полное отсутствие ошибок", верификация также не может гарантировать этого свойства у системы.

Однако, при всех успехах верификации, ее использование не может дать абсолютной гарантии правильности разрабатываемых систем, она не может заменить тестирование готовой системы. Апологетам чистой верификации можно предложить полет на самолете, бортовая система управления которого полностью верифицирована, но ни разу не проверялась ни в одном реальном режиме.

Тестирование и верификация являются взаимно дополняющими подходами к повышению степени доверия к системе.

1.8. Замечания

На форуме ACM (http://catless.ncl.ac.uk/Risks/) собрана информация о тысячах случаев отказа техники по причине ошибок, допущенных при разработке автоматизированных систем, от отказа системы бронирования билетов Syber до гибели ребенка в автоматизированном туалете.

Подробный анализ причины аварии ракеты "Ариан" и обсуждение возможных действий по выявлению в программах подобной ошибки приводятся в [172]. Описание неполадок бортовой системы зонда Гюйгенс приведено в [178]. Проблемы с советскими станциями "Фобос-1" и "Фобос-2" описаны в [175]. Анализ ошибок в процессорах AMD Phenon (Barcelona) и Opteron и их возможных причин приведен в [176]. Ссылка [181] указывает на текст, посвященный изложению проблем с системой управления в аппарате Therac-25. Информацию об ошибке в программе наведения ракеты Patriot можно найти в [171]. Выводы комиссии по расследованию причин "дружественного огня" изложены в [177]. Некорректность параллельной программы разделения множеств доказана в [165], там же определены классы исходных множеств, на которых программа работает некорректно. Протокол W.C.Lynch, рассматриваемый в этой главе, приведен в [105], анализ его некорректностей описан в книге [79].

Впервые задача верификации программ была поставлена Робертом Флойдом (Robert Floyd) в [50] и формализована как проблема доказательства теорем на основании аксиом и правил вывода Энтони Хоаром (C.A.R.Hoare) в [68]. Метод model checking для верификации параллельных систем с конечным числом состояний был впервые предложен в 1981 г. Эдмундом Кларком (Edmund Clarke), профессором университета Карнеги-Меллона, и его аспирантом Алленом Эмерсоном (Allen Emerson). В работе [29] они писали: "Глобальный граф переходов параллельной системы может рассматриваться как конечная структура Крипке, и может быть построен эффективный алгоритм для определения того, является ли эта структура моделью заданной формулы (т. е. определения того, соответствует ли программа ее спецификации)". Ими же

был введен и термин model checking. Джозеф Сифакис (Joseph Sifakis), основатель и директор лаборатории VERIMAG, французского исследовательского центра по разработке встроенных систем, был руководителем научной группы, разработавшей первую систему автоматической верификации методом проверки модели [129]. Эти работы в немалой степени способствовали тому, что именно во Франции существует лучшая в мире авионика и технологии разработки бортовых информационных и управляющих систем. Недавно в технологию фирмы Thales, крупнейшей компании по производству авиационной электроники, внедрен инструментарий для разработки систем на основе моделей, так называемые MDD tools (Model Driven Development tools). Именно с этой компанией Российский авиапром заключил многомиллионные контракты на поставку авионики [95]. Э. Кларку, А. Эмерсону и Дж. Сифакису вручена премия Тьюринга Ассоциации ACM за 2007 г. за их вклад в доведении метода model checking до производственной технологии, позволяющей осуществлять верификацию моделей реальных программных и аппаратных систем. Во многих работах приводятся результаты использования этой технологии для верификации программно-аппаратных систем для критических применений (см., например, [69], [124] и др.).

Теории и технике верификации, объединенной названием model checking, посвящена огромная литература, от монографий и вводных обзоров до многочисленных университетских курсов, представленных презентациями в Интернете. Довольно полное изложение материала этой области содержится в монографиях [34] и [19]. Укажем здесь также некоторые доступные вводные тексты: [106], [149], [117].

На русском языке, кроме перевода основополагающей монографии [34], нет книг и почти нет статей, дающих общее введение в предмет. Исключением являются слайды лекций Б. Конева, прочитанных им в ПОМИ АН РФ [193], и лекции Ю. Лифшица [194]. В сборнике [168] несколько статей посвящено вопросам верификации программ в рамках разрабатываемого под руководством проф. А. А. Шалыто "Автоматного подхода" к программированию, т. е. к программированию с явным выделением состояний.

1.9. Задачи к главе 1

1.1. Рассмотрите параллельную программу с двумя процессами, которые выполняются асинхронно:

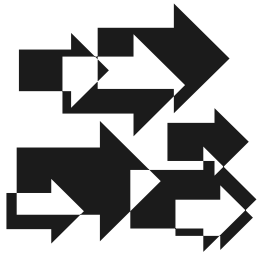
$$P1::\quad a_1 : A := 1\ ; \qquad\qquad P2::\quad a_2 : B := A\ ;$$
$$\qquad\quad b_1 : A := 2\ ; \qquad\qquad\quad\ b_2 : C := A + B\ ;$$

Значение A вначале равно 0. Рассмотрите все возможные варианты значений, которые могут иметь переменные B и C по завершении программы.

1.2. Протестируйте программу разделения множеств для различных наборов значений в множествах S и L. Докажите, что эта программа всегда работает правильно в случае одноэлементных множеств S и/или L.

1.3. Протестируйте с помощью ручной прокрутки протокол W.C.Lynch при различных ситуациях в каналах передачи данных.

ГЛАВА 2



Темпоральные логики

Для верификации технических систем свойства их поведения должны быть выражены формально логическими утверждениями, которые обеспечат простую, лаконичную и недвусмысленную их запись. В этой главе показано, что обычная логика высказываний является неадекватной для формулировки утверждений о поведении технических систем, т. е. об изменении их состояний во времени. Для спецификации таких свойств необходимы логические утверждения, истинность которых зависит от времени. Соответствующие логики называются темпоральными. Достаточно мощные, выразительные и в то же время простые темпоральные логики были построены как простые расширения обычной логики высказываний.

В этой главе вводятся две темпоральные логики: LTL — темпоральная логика линейного времени, и CTL — темпоральная логика ветвящегося времени. Эти логики очень просты, что отличает действительно полезные математические модели. Формулы этих логик являются удобным языком для выражения свойств параллельных программных систем и, как частный случай, встроенных систем.

2.1. Утверждения, истинность которых зависит от времени

При всей широте использования классической логики в науке, технике и в обычной жизни очевидны ее ограничения. Классическая логика основывается на самой примитивной модели истины. Она не позволяет выразить степень уверенности или неуверенности в истинности высказывания. Формулы классической логики могут принимать только значения "да" и "нет" на подходящей интерпретации, но не могут определить интервал возможных значений в

некоторой области. Формулы обычной логики истинны или ложны независимо от времени, в статическом мире.

Аппарат классической логики оказался недостаточно выразительным во многих областях применения. Поэтому неудивительно, что предпринимались многочисленные попытки расширений классической логики в самых различных направлениях, и некоторые из этих попыток оказались весьма успешными. В этой главе мы рассмотрим одно из таких расширений, *темпоральные логики*, которые являются очень удобными при формулировке утверждений о поведении дискретных систем, развивающихся во времени.

Если утверждения естественного языка явно или неявно включают зависимость высказываний от времени или от порядка событий во времени, то формализация их в классической логике высказываний обычно неадекватна. Например, коммутативность операции конъюнкции (перестановочность ее аргументов $A \wedge B \equiv B \wedge A$) не выполняется для следующих высказываний:

Джону стало страшно, и он убил не эквивалентно предложению *Джон убил, и ему стало страшно*.

Джон умер, и его похоронили не эквивалентно предложению *Джона похоронили, и он умер*.

Джейн вышла замуж и родила ребенка не эквивалентно предложению *Джейн родила ребенка и вышла замуж*.

Сообщение послано в канал, и на него пришло подтверждение не эквивалентно предложению *На сообщение пришло подтверждение, и оно послано в канал*.

Анализ этих утверждений в рамках обычной логики высказываний невозможен. Для адекватного формального выражения таких предложений нужна логика, позволяющая отразить соотношения моментов времени наступления событий, определяемые в естественном языке такими словами, как "*случилось после*", "*случается иногда*", "*случается всегда*".

Необходимость оперировать высказываниями, истинность которых меняется со временем, возникает часто. Например, высказывание: "*Путин — президент России"* истинно только в определенный период времени. Высказывание "*Светит солнце*", ложное сегодня, может стать истинным завтра. Утверждение "*Я голоден*" станет ложным после того, как я поем. Многие утверждения, в которых вводятся причинно-следственные отношения, также связаны со временем:

Если я видел ее раньше, то я узнаю ее при встрече.

Раз Персил — всегда Персил.

Мы не друзья, пока ты не извинишься.

Утверждение естественного языка:

Вчера он сказал, что придет завтра, значит, он сказал, что придет сегодня,

несомненно, истинно. Но в обычной логике высказываний формально доказать его истинность невозможно.

Большую долю утверждений, нуждающихся для их проверки в формализации логической теорией, составляют свойства технических систем, обладающих динамикой, т. е. поведением во времени, изменяющим некоторые параметры систем. Например:

С1: *Посланный запрос когда-нибудь в будущем будет обработан*.

С2: *Лифт никогда не пройдет мимо этажа, вызов от которого поступил, но еще не обслужен*.

Элементарные (атомарные) утверждения в этих высказываниях могут быть истинны в один момент времени и ложны в другой. Поэтому мы не можем адекватно представить утверждение С1 с помощью формулы логики высказываний:

$Послан_R \Rightarrow Обработан_R$ (если запрос R послан, то он обработан).

Действительно, в классической логике утверждения обычно понимаются как истинные либо ложные независимо от времени, а утверждение С1 явно различает разные моменты времени. Оно утверждает, что если *в некоторый момент времени* t запрос R послан, то в *какой-то будущий момент времени* $t' > t$ он будет обработан.

Попытаемся выразить эти свойства с помощью логики предикатов I порядка, вводя в употребление такие выражения, как "*событие* p *наступило в момент* t". Утверждение С1 при этом будет выглядеть так:

$$(\forall t \geq 0)\left[Послан_R(t) \Rightarrow (\exists t' > t)\, Обработан_R(t') \right].$$

Утверждение С2 в логике предикатов выглядит так (здесь $Лифт_n(t)$ — утверждение: "*Лифт в момент* t *проходит мимо этажа* n"):

$$(\forall t \geq 0)(\forall t' \geq t)\left[Вызов_n(t) \wedge (\forall t1 : t \leq t1 \leq t')\neg Обслуживается_n(t1) \Rightarrow \right.$$
$$\left. \neg(\exists t2 : t \leq t2 \leq t')\, Лифт_n(t2) \right]$$

Такая формализация трудно читается, и, как известно, доказательства в логике предикатов проводить довольно сложно. Этот путь формализации зависимых от времени утверждений пока не дал существенных результатов для практики верификации технических систем.

Попытка построения формальной модели, неявно учитывающей время в высказываниях, была предпринята философом и логиком Гансом Райхенбахом (Hans Reichenbach) для изучения глагольных времен английского языка. В теории Райхенбаха время глагола в предложении характеризуется соотношением моментов наступления событий, о которых говорится и которые подразумеваются в предложении. Используя соотношения во времени двух моментов: момента S, в который сделано высказывание (*Speech time*), и момента E наступления события, о котором говорится в высказывании (*Event time*), можно образовать только три простых времени глагола — настоящее, прошедшее и будущее с соотношениями соответственно $E = S$, $E < S$ и $E > S$. Для того чтобы покрыть бóльшее число различных глагольных времен, Райхенбах ввел третий момент — момент R точки референции (*Reference time*), т. е. момент, на который ссылается высказывание. В Future Perfect, когда мы говорим, например, "*I will have seen John*" (буквально "*Я буду иметь Джона увиденным*"), это высказывание отсылает нас не к тому моменту, когда я увидел Джона, но к моменту, по отношению к которому (with reference to) мое видение Джона уже произошло, т. е. соотношение между этими моментами времени есть $S < E < R$.

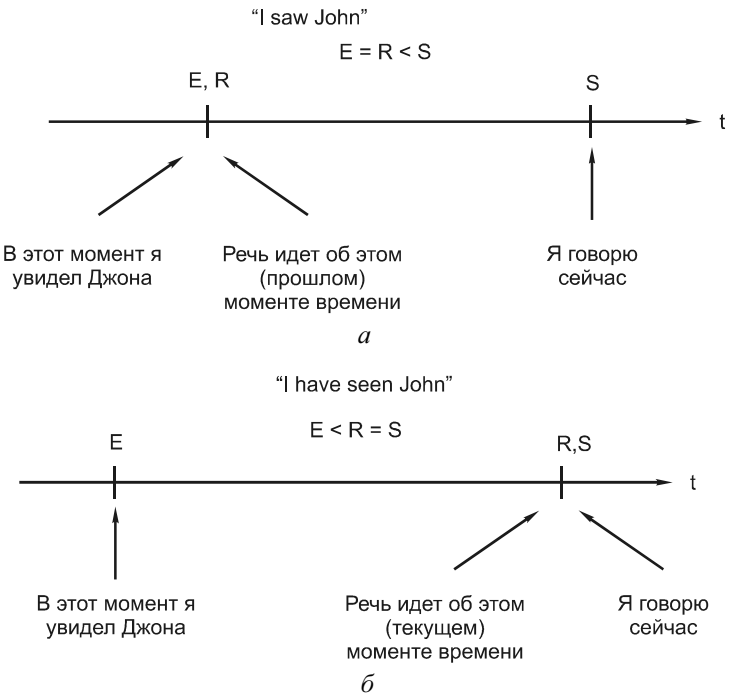


Рис. 2.1. Времена глаголов английского языка Past Indefinite и Present Perfect моделируются соотношениями моментов наступления событий

Формальная модель Райхенбаха позволяет прояснить различия многих времен глаголов английского языка. Например, в предложении "*I saw John*" соотношение между этими моментами $R = E < S$; а в предложении "*I have seen John*" это соотношение $E < R = S$ (рис. 2.1).

Однако возможность использования этого формализма для верификации дискретных систем сомнительна. Прогресс в области верификации был достигнут на пути введения специальных модальностей в обычную логику высказываний.

2.2. Модальные и временные логики. Tense Logic

В приведенных выше примерах высказываний время не присутствует явно. И нам действительно часто неважно, при каких конкретных значениях времени наступали те или иные события, важно только выразить порядок событий, *отношение* (раньше/позже) между моментами времени, в которых эти события наступали (вспомним тезис Лесли Лэмпорта (Leslie Lamport): "Время есть способ упорядочения событий"). Характеристики временных свойств систем используют категории вида "никогда не будет верно, что...", или "в будущем обязательно случится, что..." и тому подобные, характеризующие истинность суждения с учетом отношений между моментами наступления различных событий во времени. Можно расширить классическую логику, разрешив использовать такие категории перед утверждениями логики, например, выражение $\mathbf{F}\,Получено(m)$ можно понимать так: "Когда-нибудь в будущем сообщение m обязательно будет получено". Такие категории называются *модальностями*, они стоят перед высказываниями и каким-либо образом характеризуют высказывания.

Модальность в логике (от лат. *modus* — способ, наклонение) вводится дополнительным оператором, предваряющим высказывание. Модальный оператор характеризует высказывание, являющееся его операндом. В общем случае модальный оператор не обязательно связан со временем. Например, пусть q — некоторое высказывание. Можно выразить неполную уверенность в истинности утверждения q, сказав: "Возможно, что q истинно". Для формализации этого введем модальный оператор $\mathbf{M}$. Тогда $\mathbf{M}q$ имеет смысл "возможно, что q", или "может быть, q истинно".

Формальные теории устанавливают соотношения между формулами. Например, введем модальный оператор $\mathbf{L}$, имеющий смысл "с необходимостью верно, что...". Тогда можно следующим образом определить отношение между модальностями $\mathbf{M}$ и $\mathbf{L}$: $\mathbf{L}q \equiv \neg\mathbf{M}\neg q$, что согласуется с интуитивно ис-

тинным утверждением, например: "то, что q обязательно наступит, это то же самое, что неверно, что q, возможно, не наступит". Другой пример. Пусть p означает утверждение "*писатель пишет*". Тогда соотношение $\mathbf{L}p \equiv \neg\mathbf{M}\neg p$ интерпретируется совершенно правильной фразой "*Писатель должен писать тогда и только тогда, когда он не может не писать*".

Классическая логика, расширенная какими-нибудь операторами модальности, называется модальной логикой. Такое расширение можно определить поразному, вводя разного типа модальности и даже разную их семантику (формальное определение смысла модальностей). Возможность использования различной семантики проистекает из того, что в разных применениях трактовки одних и тех же модальных слов естественного языка часто различны, и даже в одной области применения они обычно смутны, расплывчаты, неоднозначны. Поэтому существует большое число различных модальных логик, каждая из которых удобна в конкретной области применения.

Модальности, которыми можно расширить классическую логику, могут быть связаны и с характеристикой момента времени, в котором утверждается истинность высказывания, зависящего от времени. Логики, расширенные такими модальностями, называются темпоральными, или временными логиками. Мы будем использовать термин *темпоральные логики*.

> ❖ ***Определение 2.1 (темпоральные логики)***
>
> Темпоральные логики — это логики, в которых истинностное значение логических формул зависит от момента времени, в котором вычисляются значения этих формул.

Темпоральные логики давно использовались в философии для изучения таких схем рассуждений, которые включают ссылки на время. Основная идея темпоральной логики состоит в том, чтобы фиксировать только *относительный порядок событий* — фактически, текущее, будущее и прошедшее время. Конечно, в некоторых приложениях, например в области верификации систем *реального времени*, явные значения времени и численные ограничения на время должны учитываться, и там мы введем другие формализмы (*см. гл. 12*).

В одной из предшественниц современных темпоральных логик — Tense Logic, разработанной английским логиком Артуром Прайором (Arthur Prior) в середине прошлого века, были впервые введены две модальности: **F** (от Future): "когда-нибудь в будущем будет истинно..." и **P** (от Past): "когда-то в прошлом было истинно...".

На рис. 2.2 показано, при каких значениях времени истинны утверждения $\mathbf{F}q$ и $\mathbf{P}q$, если заданы отрезки времени, в которых истинно утверждение q. Обо-

значим $t \models \Phi$ утверждение: "В момент времени t истинно утверждение Φ". Тогда (рис. 2.2):

- утверждение q истинно в момент времени t $\left(t \models q\right)$, если утверждение q истинно в момент t (что является, конечно, тавтологией);
- утверждение $\mathbf{F}q$ истинно в момент времени t $\left(t \models \mathbf{F}q\right)$, если для какого-то момента времени t' в будущем q станет истинным;
- утверждение $\mathbf{P}q$ истинно в момент времени t $\left(t \models \mathbf{P}q\right)$, если для какого-то момента времени t' в прошлом утверждение q было истинным.

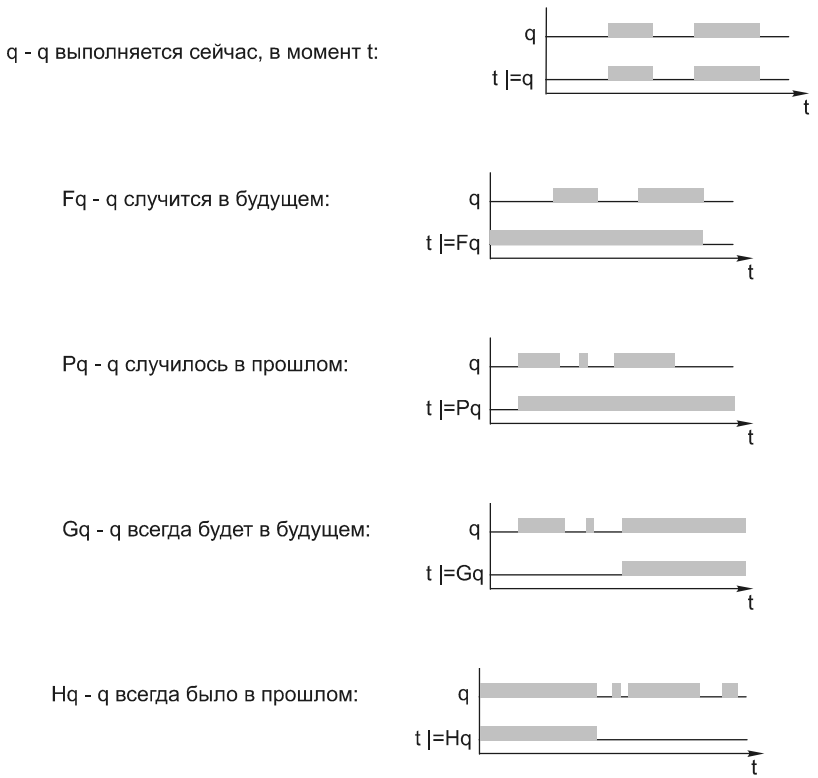


Рис. 2.2. Темпоральные операторы Tense Logic

В Tense Logic введены и два дуальных темпоральных оператора: **G** (от слова Globally) и **H** (от слова History), с соотношениями: $\mathbf{G}q \equiv \neg\mathbf{F}\neg q$, $\mathbf{H}q \equiv \neg\mathbf{P}\neg q$. Справедливость этих соотношений очевидна. Например, первого: "Утвер-

ждать, что в будущем q будет всегда истинно, это то же самое, что утверждать, что неверно, что когда-нибудь в будущем q станет ложным". Эти операторы можно определить и независимо от **F** и **P** (см. рис. 2.2):

- утверждение $\mathbf{G}q$ истинно в момент времени t $\left(t \models \mathbf{G}q\right)$, если для любого момента времени t' в будущем утверждение q истинно;
- утверждение $\mathbf{H}q$ истинно в момент времени t $\left(t \models \mathbf{H}q\right)$, если для любого момента времени t' в прошлом утверждение q истинно.

Уже с помощью двух темпоральных операторов **F** и **G** можно выразить сложные свойства, зависящие от времени. Например:

Утверждение q будет истинным:

- всегда в будущем: $\mathbf{G}q$
- хотя бы раз в будущем: $\mathbf{F}q$
- никогда в будущем: $\neg\mathbf{F}q$
- бесконечно много раз в будущем: $\mathbf{GF}q$
- с какого-то момента постоянно: $\mathbf{FG}q$

Рассмотрим, как с помощью этих операторов формально записать некоторые утверждения.

- События p и q никогда не произойдут одновременно:

  $\mathbf{G}\neg\left(p \wedge q\right)$

- Любое посланное сообщение когда-нибудь в будущем будет получено:

  $\mathbf{G}\left(Послано_m \Rightarrow \mathbf{F}\, Получено_m\right)$

- Джейн вышла замуж и родила ребенка:

  $\mathbf{P}\left(Джейн_выходит_замуж \wedge \mathbf{F}\, Джейн_рожает_ребенка\right)$

- Джейн родила ребенка и вышла замуж:

  $\mathbf{P}\left(Джейн_рожает_ребенка \wedge \mathbf{F}\, Джейн_выходит_замуж\right)$

- Ленин жил, Ленин жив, Ленин будет жить:

  $\mathbf{PG}\, Ленин_жив$

- Пока ключ зажигания не вставлен, машина не поедет:

  $\mathbf{G}\left(\neg\mathbf{P}\, Зажигание \Rightarrow \neg Старт\right)$

❒ В системах передачи информации формула

$$\mathbf{G}\big((m \in Receiver.in) \Rightarrow \mathbf{P}(m \in Sender.out)\big)$$

выражает следующее свойство: канал передачи информации не может спонтанно порождать свои собственные сообщения. А именно, всегда ($\mathbf{G}$), если сообщение m появилось во входном буфере приемника $(m \in Receiver.in)$, то когда-то ранее ($\mathbf{P}$) сообщение m находилось в выходном буфере передатчика $(m \in Sender.out)$.

В темпоральную логику можно ввести еще два оператора: *NextTime* ($\mathbf{X}$) и *Until* ($\mathbf{U}$).

Оператор *NextTime*: утверждение $\mathbf{X}q$ истинно в момент времени t, если q истинно в следующий момент $t+1$:

❒ Джон убил, и ему стало страшно:

$$\mathbf{P}(Джон_убивает \wedge \mathbf{XG}Джону_страшно).$$

❒ Если я видел ее раньше, то я ее узнаю при встрече:

$$\mathbf{G}\big(\mathbf{P}Увидел \Rightarrow \mathbf{G}\big(Встретил \Rightarrow \mathbf{X}Узнал\big)\big).$$

❒ Джон умер и его похоронили:

$$\mathbf{P}(Джон_умирает \wedge \mathbf{XF}Джона_хоронят).$$

Оператор *Until* (до тех пор, пока не наступит нечто) требует двух аргументов — утверждений. Утверждение $p\mathbf{U}q$ истинно в момент времени t, если q истинно в некоторый будущий момент времени, а во всем промежутке от момента t до t' истинно p : *когда-нибудь* q, *а до тех пор* p. Пример развертки во времени, показывающей, когда будет истинно утверждение $p\mathbf{U}q$ при различных истинностных значениях p и q, показан на рис. 2.3.

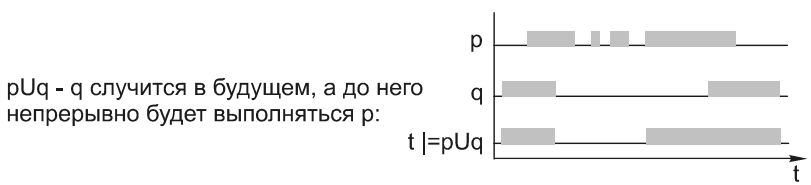


Рис. 2.3. Темпоральный оператор *Until*

С оператором *Until* многие сложные утверждения легко представимы в формальной записи, например:

- Мы не друзья, пока ты не извинишься:

  $(\neg Мы_друзья)\,\mathbf{U}\,Ты_извиняешься$

- Лифт никогда не пройдет мимо этажа, вызов от которого поступил, но еще не обслужен:

  $\mathbf{G}\left(Вызов(n) \Rightarrow \left(\neg Лифт(n)\right)\mathbf{U}\,Обслуживается(n)\right)$

Через оператор *Until* легко выражается оператор $\mathbf{F}$:

$$\mathbf{F}q = true\,\mathbf{U}\,q$$

а следовательно, и оператор $\mathbf{G}$:

$$\mathbf{G}q = \neg\left(\mathbf{F}\neg q\right) = \neg\left(true\,\mathbf{U}\neg q\right).$$

Подобно тому, как оператор $\mathbf{H}$ является аналогом оператора $\mathbf{G}$ в прошлом, в этой логике и для операторов $\mathbf{X}$ и $\mathbf{U}$ существуют их аналоги в прошлом: $\mathbf{X}^{-1}$ (*в предыдущий момент времени*, "*вчера*") и $\mathbf{S}$ (since, "*с тех пор, как*").

Tense Logic позволяет формально представить некоторые философские мысли о времени.

Например, обозначим атомарное утверждение "*нечто есть*" символом q. Тогда следующее глубокомысленное абсолютно истинное заключение (тавтологию):

Будет, что нечто было, если и только если оно или есть сейчас, или будет, или уже было

можно формализовать так:

$$\mathbf{FP}q \equiv q \vee \mathbf{F}q \vee \mathbf{P}\,q.$$

Тождественно истинными являются и такие формулы:

$p \Rightarrow \mathbf{HF}p$ — *если нечто есть, то в прошлом оно всегда собиралось наступить*.

$p \Rightarrow \mathbf{GP}p$ — *если нечто есть, то в будущем оно всегда будет наступившим*.

Философская мысль о времени:

Любое "вчера" было когда-то "завтра", любое "завтра" когда-нибудь станет "вчера"

формально запишется так:

$$\left(\mathbf{PX}^{-1}q \Rightarrow \mathbf{PX}q\right) \wedge \left(\mathbf{FX}q \Rightarrow \mathbf{FX}^{-1}q\right).$$

Тождественно истинной формулой $q \equiv \mathbf{X}^{-1}\mathbf{X}q$ можно формализовать мысль Иосифа Бродского:

Настоящему, чтобы обрести будущее, требуется вчера.

Временная логика помогает точно и однозначно выразить временные соотношения между событиями, о которых повествуют (часто неоднозначные) фразы естественного языка. Обозначим:

$Д(x)$: *x — мой друг*, $П(x)$: *x — президент.*

Используя эти обозначения, можно формально записать несколько совершенно различных интерпретаций предложения "*Мой друг NN будет президентом*":

1. $Д(NN) \wedge \mathbf{FX}\, П(NN)$:

   сейчас NN — мой друг, а в будущем он станет президентом.

2. $\mathbf{FX}\big(Д(NN) \wedge\ П(NN)\big)$:

   в будущем NN станет моим другом, и в это время он уже будет президентом.

3. $\mathbf{F}\big(Д(NN) \wedge \mathbf{FX}\, П(NN)\big)$:

   в будущем NN станет моим другом, и потом он станет президентом.

2.3. Темпоральная логика линейного времени (LTL)

Современная темпоральная логика линейного времени (LTL, Linear Time Logic) является наследницей временной логики Tense Logic Артура Прайора. LTL разработана израильским ученым Амиром Пнуэли (Amir Pnueli) для спецификации свойств параллельных технических систем. В LTL максимально упрощены все включенные в нее концепции и строго формально определена интерпретация ее формул.

Во-первых, в LTL темпоральными операторами расширена простейшая логика высказываний, формулы которой строятся из конечного числа атомарных предикатов и булевых операций. "Предикат" — это высказывание, принимающее значение либо "*истина*", либо "*ложь*" в зависимости от значений его аргументов. Например, истинность высказывания "$x > 5$" зависит от значения аргумента x. В логике линейного времени оперируют атомарными предикатами, т. е. здесь нас интересует только истинность либо ложность элементар-

ных высказываний, а не то, как эти высказывания построены или как определяется их истинностное значение.

> ❖ ***Определение 2.2 (атомарный предикат, atomic predicate)***
>
> Атомарный предикат (атомарное утверждение) — это утверждение, которое может принимать истинное или ложное значение и от структуры которого мы абстрагируемся.

Во-вторых, в новую логику включено минимальное число темпоральных операторов, которые определяют характеристики истинности высказываний, упорядоченных во времени. Таких операторов только два, **U** и **X**. Логика LTL не включает темпоральных операторов прошлого. Основания для этого очевидны: прошлое для технических систем менее важно, чем их будущее поведение, начинающееся с момента их включения, запуска.

В качестве интерпретации формул темпоральной логики рассматривается дискретная во времени бесконечная линейная направленная в будущее последовательность "миров", в каждом из которых существует своя интерпретация атомарных утверждений (рис. 2.4). Иными словами, формулы LTL принимают истинное или ложное значение на последовательности миров, и в каждом из миров для всех введенных атомарных утверждений определены свои конкретные истинностные значения. Такой взгляд на изменчивость во времени значения истинности атомарных утверждений (атомарных предикатов) имел еще Аристотель, который говорил, что "утверждения и мнения" могут иметь разные значения истинности в зависимости от моментов, в которые они делаются, отражая изменения в объектах, свойства которых они представляют.

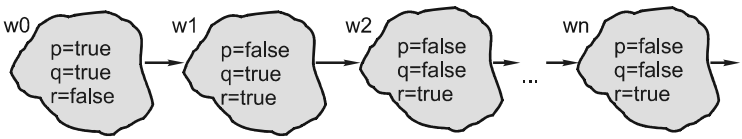


Рис. 2.4. Интерпретация темпоральной логики LTL — бесконечная последовательность миров

Линейная темпоральная логика является расширением обычной логики высказываний для рассуждений о бесконечной последовательности "миров" (например, дней или лет нашей жизни, как у Лермонтова: "*в раскаянье бесплодном влачил я цепь тяжелых лет...*"). В последовательности миров время можно считать изоморфным натуральному ряду 0, 1, 2, ..., все миры в цепочке можно считать пронумерованными, и в каждом мире логические переменные (атомарные предикаты) принимают конкретные истинностные значения —

либо истину, либо ложь. Например, сегодня я числюсь студентом и Путин — президент, а завтра я отчислен, Путин уже не президент, а Джейн родила ребенка. Направленность последовательности миров от прошлого к будущему позволяет проводить рассуждения об относительном времени, в терминах "до" и "после". На рис. 2.4 дан пример такой интерпретации. На последовательности миров $w0, w1, w2, \ldots, wn, \ldots$ задано множество атомарных предикатов $\{p, q, r\}$. В мире $w0$ атомарные предикаты p и q истинны, а r — ложен, в мире $w1$ утверждения q и r истинны, а p — ложно, и т. п. Иными словами, каждый мир может быть не похожим на другой:

День конченый — не повторится.
День кончился. Что было в нем?
Не знаю, пролетел, как птица.
Он был обыкновенным днем.
А все-таки не повторится.

Зинаида Гиппиус

Будем считать, что в этом примере, начиная с $w2$, все истинностные значения p, q и r во всех мирах этой конкретной бесконечной последовательности (рис. 2.4) одинаковы и совпадают с их значениями в $w2$.

Как происходит переключение от одного мира к следующему, теория не рассматривает, от этого она абстрагируется.

Поскольку в каждом мире каждый атомарный предикат имеет свое стабильное и неизменное в этом мире истинностное значение — истину или ложь, *true* или *false*, то и формулы обычной логики высказываний в разных мирах могут иметь различные истинностные значения, вычисленные по этим интерпретациям своих аргументов. Например, в мире $w0$ формула $q \Rightarrow r$ ложна, а во всех следующих мирах данной цепочки миров она истинна. Формула $r \wedge \neg q$ в мире $w0$ ложна, а в $w2$ — истинна. Рис. 2.5 показывает, в каких мирах упомянутые формулы выполняются, а в каких — нет (они представлены тогда со знаком отрицания "¬").

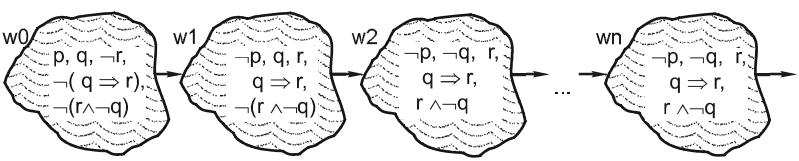


Рис. 2.5. В каждом мире некоторые пропозициональные формулы истинны, другие — нет

Очевидно, что на конкретной цепочке миров можно, в принципе, подсчитать и истинность формул, включающих темпоральные операторы. Например, формула $r \wedge \neg q$ истинна, начиная с $w2$, поэтому в $w0$ истинна формула $q\mathbf{U}(r \wedge \neg q)$: действительно, существует такое $i \geq 0$ (равное 2), что до wi истинно q, а начиная с wi истинно $r \wedge \neg q$ (ниже дано формальное определение всех темпоральных операторов).

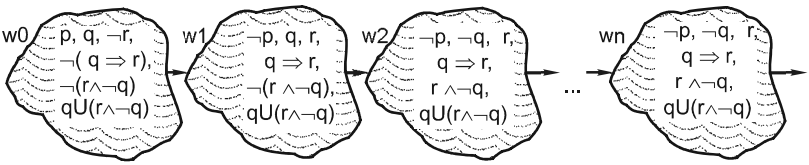


Рис. 2.6. В каждом мире некоторые темпоральные формулы истинны, другие — нет

❖ ***Определение 2.3***

Формула линейной темпоральной логики Ф выполняется на последовательности миров $w0, w1, w2, \ldots$, если Ф истинна в начальном мире $w0$ этой последовательности.

Такая привязка к начальному моменту времени важна для технических систем: для нас обычно важно знать, будет ли некоторое свойство *будущего* поведения выполняться в системе, стартующей из *начального* состояния. Это означает, что для приведенной цепочки миров выполняются формулы p, q, $\neg r$, $\neg(q \Rightarrow r)$, $\neg(r \wedge \neg q)$, $q\mathbf{U}(r \wedge \neg q)$ и т. д. — именно они истинны в $w0$.

Отметим, что существуют трактовки, которые считают формулу темпоральной логики истинной на цепочке миров только тогда, когда она истинна *на всех* мирах этой цепочки. Первая точка зрения (в английской терминологии она называется "*anchored*", *привязанная к началу*) кажется более предпочтительной, чем вторая (которая в английской терминологии называется "*drifting*", *дрейфующая*), потому что вторую можно легко выразить через первую:

$$drifting \equiv \mathbf{G}\ anchored$$

Формулы LTL построены из атомарных предикатов с помощью булевых операций и темпоральных операторов. Значения истинности обычных булевых формул в каждом конкретном мире определяется очевидным образом по значениям истинности, которые имеют атомарные предикаты в этом мире. Для темпоральных формул их истинностные значения в каждом мире определя-

ются по значениям истинности их аргументов в цепочке миров, начинающейся с этого мира.

Темпоральные формулы характеризуют развитие ситуации во времени, они могут использоваться для задания свойств бесконечных процессов. Так, формула $\mathbf{X}p$ истинна, если в следующем мире (например, на следующий день) утверждение p будет истинным. Формула $p\mathbf{U}q$ истинна в текущем мире, если q уже истинно в этом мире, или же в этом мире истинно p, и оно будет истинным во всех будущих "мирах" до тех пор, пока не станет истинным q. В логике LTL через темпоральный оператор $\mathbf{U}$ можно определить темпоральные операторы $\mathbf{F}$ и $\mathbf{G}$. Поэтому операторы $\mathbf{F}$ и $\mathbf{G}$ называются "выводимыми" в логике LTL.

Например, формула $\mathbf{F}p$ истинна в текущем мире данной цепочки миров, если когда-нибудь в будущем в этой цепочке станет истинным p (возможно, неоднократно). Можно строить и композицию темпоральных операторов. Например, формула $\mathbf{GF}p$ истинна в текущем мире, если в любом будущем мире верно, что в дальнейшем когда-нибудь станет истинным p. В мирах, которые идут после тех, в которых p стало истинным, тоже выполняется $\mathbf{F}p$, поэтому $\mathbf{GF}p$ определяет, что в будущем p будет истинным бесконечно много раз. Формула $\mathbf{FG}p$ истинна, если когда-нибудь в будущем утверждение p станет истинным и останется истинным навсегда.

При записи формул темпоральной логики будем считать, что все темпоральные операторы имеют одинаковый приоритет, больший, чем приоритет любой логической операции. Поэтому, например, формулу $p\mathbf{U}q \vee \mathbf{G}q$ будем понимать, как $(p\mathbf{U}q) \vee (\mathbf{G}q)$. Для более ясной записи и выделения структуры формул рекомендуется всегда использовать скобки.

Пример 2.1

Рассмотрим несколько примеров формализации высказываний естественного языка. Такую формализацию зачастую можно выполнить по-разному, поскольку высказывания естественного языка не всегда однозначны.

а) Формализуем известное латинское изречение "*Dum spiro, spero*" (пока живу, надеюсь). Его нельзя представить простой импликацией "*Если я жив, то я надеюсь*" (формально: *Я_жив* $\Rightarrow$ *Я_надеюсь*), поскольку формула $p \Rightarrow q$ показывает связь двух атомарных утверждений p и q только в текущий момент. Утверждение "*Пока живу, надеюсь*" говорит, что я буду надеяться *всегда*, пока буду жив. Поэтому более точно его смысл передает такая формула LTL:

$\mathbf{G}(Я_жив \Rightarrow Я_надеюсь)$: "*Сейчас и всегда в будущем, если я буду жив, я буду надеяться*".

Подобно этому, утверждение "*Пока сердце бьется, я буду бороться!*" формализуется так:

$\mathbf{G}(Сердце_бьется \Rightarrow Я_борюсь)$.

б) Несколько другой смысл имеет утверждение "*Мы будем бороться, пока не победим*". Его можно формализовать так:

$Мы_боремся\ \mathbf{U}\ Мы_победили$.

Эта формализация предполагает уверенность говорящего в будущей победе. Если такой уверенности нет, то утверждение следует формализовать так:

$(Мы_боремся\ \mathbf{U}\ Мы_победили) \vee \mathbf{G}\ Мы_боремся$.

В этой формуле нашла отражение мысль: мы все равно будем бороться, даже если и не победим (причем не исключено, что борьба продолжится и после победы). В темпоральной логике иногда используется оператор **W**, так называемый "слабый Until" (по-английски "Unless"), имеющий именно эту семантику:

$p\mathbf{W}q \equiv p\,\mathbf{U}\,q \vee \mathbf{G}\,p$.

С использованием слабого *Until* утверждение "*Мы будем бороться, пока не победим*" более точно формализуется так:

$Мы_боремся\ \mathbf{W}\ Мы_победили$.

Эта формула утверждает, что мы боремся сейчас, будем бороться и дальше, пока не победим, и даже если победы не будет, мы все равно будем бороться.

в) Англоязычные документы часто снабжаются припиской: "Not valued until signed" — "не действителен до тех пор, пока не подписан". Одна из возможных формализаций этой фразы:

$Документ_не_действителен\ \mathbf{W}\ Документ_подписан$.

Эта формула, однако, ничего не говорит о том, что случится с момента подписания документа. По умолчанию эта фраза подразумевает, что с момента подписания документ сразу же становится действительным (вступает в силу). Поэтому более точно смысл этой фразы передает следующая формула:

$$(\neg Документ_действителен)\ \mathbf{W}\ (Документ_подписан \wedge \mathbf{G}\ Документ_действителен)$$

Здесь используется слабый *Until* , поскольку нет гарантий, будет ли документ действительно подписан. Если гарантировано, что документ в будущем будет подписан (например, паспорт обязательно подписывается владельцем при получении и сразу становится валидным), то эту фразу можно формально представить оператором *Until* :

$$\left(\neg \textit{Документ_действителен}\right)\mathbf{U}\left(\textit{Документ_подписан} \wedge \mathbf{G}\,\textit{Документ_действителен}\right).$$

г) Если обозначить p утверждение "*Я твоя!*", то $\mathbf{G}p$ означает "*Я твоя навеки!*".

д) Утверждение "*Мы придем к победе коммунистического труда!*" формально запишется так:

$\mathbf{F}\,\textit{Коммунистический_труд_победил}!$

е) Строфе из песни В. Бахнова "*Сегодня он играет джаз, а завтра Родину продаст*", если ее понимать буквально, соответствует формула:

$\textit{Он_играет_джаз} \Rightarrow \mathbf{X}\,\textit{Он_продает_Родину}$.

Но, по-видимому, поэт имел в виду более общий смысл:

$\mathbf{G}\left(\textit{Он_играет_джаз} \Rightarrow \mathbf{X}\,\textit{Он_продает_Родину}\right)$ —

"*если когда-нибудь парень начнет играть джаз, то на следующий день он пойдет продавать Родину*".

Однако, возможно, поэт хотел этой фразой передать еще более общую мысль: "*если когда-нибудь парень начнет играть джаз, то потом, рано или поздно, он обязательно продаст Родину*". В этом случае он мог бы сказать точно и недвусмысленно (хотя и не совсем в рифму):

$\mathbf{G}\left(\textit{Он_играет_джаз} \Rightarrow \mathbf{XF}\,\textit{Он_продает_Родину}\right)$.

ж) Пусть p означает "*Я люблю Машу*", а q — "*Я люблю Дашу*". Тогда:

$\mathbf{F}p$ —"*Я когда-нибудь обязательно полюблю Машу*";

$\mathbf{G}q$ — "*Я люблю Дашу и буду любить ее всегда*";

$\mathbf{G}(\neg p \vee \neg q)$ — "*Я однолюб, и никогда не буду любить и Машу, и Дашу одновременно*";

$q\mathbf{U}p$ — "*В будущем я полюблю Машу, а до той поры я буду любить Дашу*";

$\mathbf{FG}p$ — "*Когда-нибудь я полюблю Машу навечно*";

$\mathbf{GF}q$ — "*Я буду бесконечно влюбляться в Дашу*".

з) Рекламный слоган "*Раз Персил — всегда Персил*" не совсем ясен простому человеку. Что значит — "*Раз Персил*"?

По-видимому, рекламодатели хотели сказать, что если некто когда-нибудь попробует Персил, то с этого момента он уже не сможет от Персила отказаться и будет им пользоваться всегда. Но тогда абсолютно точно смысл этого слогана выражает следующая формула логики LTL:

$$\mathbf{G}\left(\textit{Персил} \Rightarrow \mathbf{G}\,\textit{Персил}\right).$$

Подобными точными формулами и надо рекламировать товары.

Итак, темпоральные формулы могут конечным образом характеризовать свойства бесконечных процессов, имеющих поведение по времени, разворачивающихся во времени в последовательности миров, или ситуаций, если снабдить миры (каждую статическую ситуацию в дискретной последовательности таких ситуаций) конечным набором атомарных утверждений, принимающих одно из двух значений — истина или ложь — в каждом из миров. Атомарные утверждения сами не включают время, в каждом мире их истинностное значение статично, неизменно. Но в другом мире эти значения могут быть другими.

2.4. Реагирующие системы (reactive systems)

❖ ***Определение 2.4 (реагирующие системы, reactive systems)***

Реагирующие системы — это класс информационных систем, основной функцией которых является поддержание взаимодействия с окружением, а не преобразование информации.

Для этого класса систем — программ, аппаратуры и т. п., которые в англоязычной литературе называются reactive systems, мы будем использовать термин *реагирующие*, потому что буквальный перевод слова *reactive* — термин *реактивные* — имеет прочную устоявшуюся семантику: он характеризует устройство, имеющее в хвосте сопло с вырывающимися клубами огня.

Реагирующие системы отличны от обычных программ преобразования данных (их часто называют *трансформационными программами*). Трансформационные программы принимают на вход исходные данные, производят вычисления и выдают результат на выход, после чего работу завершают. Программы обращения матрицы, поиска экстремума функции, решения дифференциального уравнения численным методом — все это трансформационные программы. Фактически программы преобразования данных являются функциями, осуществляющими отображение входных данных в выходные, и верификация такой програмы состоит в проверке правильности преоб-

разования возможных начальных состояний программы в заключительные ее состояния.

Реагирующие системы, напротив, функционируют бесконечно, они контролируют окружение, реагируя на внешние события — получение сообщения, щелчок кнопкой мыши, нажатие клавиши <Enter> и т. п. Останов таких систем обычно связан с поломкой или коллизией (блокировкой) и является ошибкой. Как правило, реагирующие системы являются частями бóльших систем, с которыми они взаимодействуют. Операционные системы, протоколы коммуникации, планировщики, контроллеры, параллельные взаимодействующие программы, системы логического управления, драйверы — все это примеры реагирующих систем. Верификация реагирующей системы не может быть сведена к проверке отношения между входом и выходом после прихода системы в заключительное состояние: такие системы имеют начальное состояние, но не имеют заключительного состояния, они вообще не должны останавливаться. Для задания требуемых свойств поведения реагирующих систем наиболее удобным средством оказались формулы темпоральной логики.

Сведем в таблицу сравнительные характеристики трансформационных и реагирующих систем (табл. 2.1).

Таблица 2.1. Сравнительные характеристики трансформационных и реагирующих систем

	Трансформационные программные системы	**Реагирующие программные системы**
Примеры	Обработка данных, численные методы, вычисление функции, распознавание образов и т. д.	Протоколы, драйверы, системы логического управления, ОС и т. д.
Цель	Получение выходного значения как функции исходных значений	Обеспечение взаимодействия с окружением по определенным правилам
Вычисления	Всегда конечны с получением результата по их завершении	Всегда бесконечны; никогда не завершаются
Семантика	Функция: выход программы является функцией от входных данных	Поведение: последовательность состояний и реакций системы на внешние события
Спецификация (требования к системе)	Обычно в виде логических пред- и постусловий, например $\{x \geq 0\}\, S\, \{y = \sqrt{x}\}$: если до запуска программы S переменная x не меньше 0, то после завершения S переменная y будет иметь значение $\sqrt{x}$	Обычно, в виде формул темпоральной логики, например: $\mathbf{G}(p \Rightarrow \mathbf{XF}\, q)$: для всех состояний программы справедливо следующее: если выполнилось утверждение p, то потом, в каком-то следующем ее состоянии, обязательно выполнится q

Введенные выше последовательности миров можно считать моделями функционирования технических реагирующих систем. Каждый мир может представлять состояние реагирующей системы, а переходы — изменения состояний, дискретные переходы системы. Как внешнее воздействие, так и реакцию системы можно включить в ее состояние. Теперь можно формально определить вычисление реагирующей системы и ее поведение:

> ❖ ***Определение 2.5 (вычисление и поведение реагирующей системы)***
>
> Вычисление реагирующей системы — это бесконечная последовательность состояний, которые система проходит во времени. Поведение реагирующей системы — это все возможные ее вычисления.

Будем обозначать вычисления греческими буквами π, σ и т. п., например: $\sigma = s_0 s_1 s_2 s_3 \ldots$. Из одного состояния в другое система переходит в некоторые дискретные моменты времени $t_0, t_1, t_2, \ldots$, выполняя операции. Начальное состояние вычисления часто помечается стрелкой (рис. 2.7).

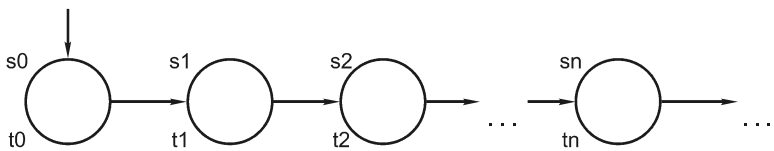


Рис. 2.7. Бесконечная цепочка миров как вычисление дискретной системы

В каждом состоянии может быть задан набор истинных в этом состоянии атомарных предикатов — тех базовых свойств, которые интересуют нас при анализе системы (рис. 2.8). Если атомарный предикат не выполняется в состоянии, то его в этом состоянии не будем указывать (т. е. состояния помечаются теми атомарными предикатами, которые в этом состоянии истинны). По значениям этих элементарных, неделимых утверждений можно строить как булевы формулы, характеризующие конкретное состояние, так и темпоральные формулы, характеризующие целое вычисление, начинающееся в этом состоянии.

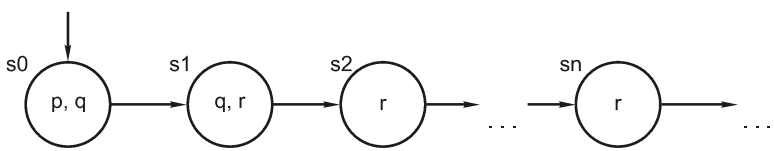


Рис. 2.8. Атомарные предикаты — базисные свойства системы.
В каждом состоянии вычисления истинны некоторые атомарные предикаты из заданного конечного их множества

Имея в каждом состоянии вычисления известные истинностные значения атомарных предикатов, мы можем в каждом состоянии дискретной системы подсчитать истинность или ложность любых логических и (в принципе) любых темпоральных формул. Темпоральные формулы, истинные в некотором состоянии вычисления, — это свойства динамического процесса, характеризующие будущее поведение системы — вычисление, начинающееся в этом состоянии. Если некоторая темпоральная формула истинна в состоянии s_0 вычисления, то говорят, что она истинна для всего вычисления, начинающегося в этом состоянии (рис. 2.9).

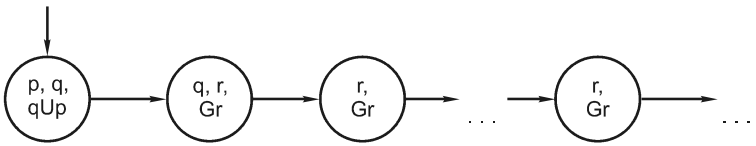


Рис. 2.9. Темпоральные формулы в каждом состоянии характеризуют свойства вычисления, начинающегося в этом состоянии

На рис. 2.9 формула $q\mathbf{U}\,p$ выполняется на всем представленном там вычислении, а формула $\mathbf{G}\,r$ на нем не выполняется. Формула $\mathbf{G}\,r$ выполняется на вычислении, начинающемся с s_1. Если темпоральная формула не выполняется в состоянии, то ее, как и ложный в этом состоянии атомарный предикат, не будем указывать в этом состоянии.

В качестве атомарных предикатов в реагирующих системах могут использоваться любые утверждения, принимающие значение "истина" или "ложь" в состояниях системы, например: "*переменная x положительна*" $(x > 0)$, "*очередь заявок к ресурсу пуста*" $(q = 0)$, "*подтверждение посылки сообщения получено*" $(request = true)$. Имя состояния, в котором может находиться дискретная система (например, метка оператора), также может быть атомарным предикатом. Атомарный предикат, принимающий истинное значение, когда модуль M находится в состоянии s, можно записать как $'M@s'$, или, если имя модуля очевидно, то $'@s'$, или же просто $'s'$. Например: свойство "принтер занят (находится в состоянии или режиме занятости)" запишестя как $'Printer@busy'$, а свойство "процесс M находится в своем критическом интервале" — как $'M@crint'$ и т. п.

Темпоральная логика оказалось очень эффективной в качестве языка спецификации требований к поведению реагирующих систем, особенно для параллельных и распределенных систем, в которых именно упорядоченность собы-

тий выражает свойства их корректного поведения. Например, требование взаимного исключения параллельных процессов формулируется просто: $\mathbf{G}\neg(@cr\,\mathrm{int}_1 \wedge @cr\,\mathrm{int}_2)$ — "*никогда два процесса не будут находиться в своих критических интервалах одновременно*".

Реагирующие системы обычно строятся из нескольких компонент, взаимодействующих друг с другом, и проверка свойств таких систем весьма важна: как было указано выше, даже если поведение каждой компоненты абсолютно ясно, свойства поведения нескольких таких компонент, функционирующих параллельно и взаимодействующих между собой, предсказать почти невозможно. Темпоральная логика и метод model checking используются, в основном, для спецификации свойств и верификации именно реагирующих систем. Свойства, которые интересуют разработчиков таких систем, обычно выражают возможные последовательности событий, возникающие при функционировании систем во времени, например: можно ли гарантировать, что запрос сервиса никогда не останется без ответа, всегда ли посланные сообщения будут доставляться коммуникационным протоколом без пропусков и дублирования в порядке их посылки, может ли в распределенной системе комментарий на новость прийти раньше, чем новость сама, и т. п.

2.5. Формальное определение LTL

Формальное определение темпоральной логики линейного времени состоит из определения синтаксиса — правил построения формул логики, и определения семантики, т. е. правил интерпретации этих формул.

> ❖ ***Определение 2.6 (формулы LTL)***
>
> Формула линейной темпоральной логики (Linear Temporal Logic) LTL это:
>
> - атомарное утверждение (атомарный предикат) $p, q, \ldots$;
> - или формулы LTL, связанные логическими операторами $\neg, \vee$;
> - или формулы LTL, связанные темпоральными операторами $\mathbf{X}$, $\mathbf{U}$.
>
> Прошлое в логике LTL не рассматривается.

Более формально структура всех возможных правильно построенных формул LTL задается грамматикой:

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

При необходимости в формулах LTL используются скобки.

Таким образом, в логике LTL дополнительно к обычной логике высказываний (пропозициональной логике) добавлены только два темпоральных оператора: унарный оператор **X** (*NextTime*) и бинарный **U** (*Until*). Два булевых оператора — отрицание и дизъюнкция — образуют базис (дизъюнктивный базис) логики высказываний: через эти операторы могут быть выражены любые другие булевы операторы. Можно считать, что все другие логические операторы выводимые, они используются для сокращения записи формул:

- *true* можно определить, как сокращение для $p \vee \neg p$;
- *false* — как сокращение для $\neg true$;
- $\varphi_1 \wedge \varphi_2$ (конъюнкция) — как сокращение для $\neg(\neg\varphi_1 \vee \neg\varphi_2)$;
- $\varphi_1 \Rightarrow \varphi_2$ (импликация) — как сокращение для $\neg\varphi_1 \vee \varphi_2$, и т. д.

Пара темпоральных операторов **X** и **U** образует темпоральный базис LTL: другие темпоральные операторы можно выразить через них. Чаще всего в LTL используются выводимые операторы **F** и **G**:

- **F**φ можно считать сокращением для *true***U**φ;
- **G**φ можно считать сокращением для $\neg$**F**$\neg\varphi$.

Семантика формул LTL определяется на вычислениях. Это значит, что формулы LTL интерпретируются (принимают истинное или ложное значение) на бесконечных цепочках состояний, в каждом из которых атомарные предикаты имеют конкретное истинностное значение — *true* или *false*. Любая формула φ логики LTL на конкретном вычислении $\sigma = s_0 s_1 s_2 \ldots$ будет либо истинной, либо ложной.

Обозначим $\sigma(i)$ состояние s_i вычисления σ, а σ^i — i-й суффикс цепочки σ, т. е. $\sigma^i = s_i s_{i+1} s_{i+2} \ldots$ Тогда $s_i \models \varphi$ обозначает утверждение: "*В состоянии* s_i *выполняется (истинна) формула* φ", а $\sigma \models \varphi$ — утверждение "*На вычислении* σ *выполняется формула* φ". В соответствии с определением 2.3:

$$\sigma \models \varphi \equiv \sigma^0 \models \varphi$$

т. е. формула истинна на вычислении, если и только если она истинна в начальном состоянии этого вычисления.

❖ ***Определение 2.7 (семантика формул LTL)***

Истинность формулы φ логики LTL на вычислении $\sigma = s_0 s_1 s_2 \ldots$ (обозначается $\sigma \models \varphi$) определяется индуктивно по структуре формулы φ:

- $\sigma \models p$, если атомарный предикат p истинен в начальном состоянии вычисления σ;
- $\sigma \models \neg\varphi \equiv \sigma \not\models \varphi$ — если формула φ не выполняется на σ;
- $\sigma \models \varphi_1 \vee \varphi_2 \equiv \sigma \models \varphi_1 \vee \sigma \models \varphi_2$ — если на σ выполняется или формула φ_1, или формула φ_2;
- $\sigma \models \mathbf{X}\varphi \equiv \sigma^1 \models \varphi$ — если φ выполняется на вычислении $\sigma^1 = s_1 s_2 s_3 \ldots$;
- $\sigma \models \varphi_1 \mathbf{U} \varphi_2 \equiv (\exists k \geq 0)\left(\sigma^k \models \varphi_2 \wedge (\forall j : 0 \leq j < k)\, \sigma^j \models \varphi_1\right)$ — если когда-то в будущем состоянии вычисления σ (включая настоящее) выполнится формула φ_2, а до ее выполнения во всех состояниях вычисления σ будет выполняться φ_1.

Значение квантора '$\forall$' на пустом множестве равно *true*, поэтому определение оператора *Until* является "рефлексивным" в том смысле, что настоящее является частью будущего. То же относится и к выводимым темпоральным операторам **F** и **G**:

- $\sigma \models \mathbf{F}\varphi \equiv (\exists k \geq 0)\, \sigma^k \models \varphi$ — если найдется такое k, что φ выполняется на вычислении $\sigma^k = s_k s_{k+1} s_{k+2} \ldots$;
- $\sigma \models \mathbf{G}\varphi \equiv (\forall k \geq 0)\, \sigma^k \models \varphi$ — если φ выполняется на всех состояниях вычисления σ.

2.6. Примеры использования формул LTL

Напомним, что для уменьшения числа необходимых скобок при записи формул LTL часто используют следующее соглашение о приоритетах операций: все темпоральные операторы имеют равный приоритет, более старший, чем приоритет любой логической операции. Например, формула

$$\mathbf{G}p \Rightarrow \mathbf{F}r\mathbf{X}q$$

должна пониматься так:

$$(\mathbf{G}p) \Rightarrow \big((\mathbf{F}r) \wedge (\mathbf{X}q)\big).$$

При необходимости или в случае сомнений в темпоральной формуле следует расставлять скобки, чтобы указать порядок выполнения операций.

Иногда вместо значков $\mathbf{F}$, $\mathbf{G}$, $\mathbf{X}$ пишут ◊, [] и O. В этой нотации формула

$$\mathbf{G}p \Rightarrow \mathbf{F}r\mathbf{X}q$$

может выглядеть как

$$[]p \Rightarrow \langle\rangle r\,\mathrm{O}q\ .$$

Пример 2.2

Рассмотрим примеры записи некоторых свойств бесконечных вычислений с помощью формул LTL.

а) $\mathbf{G}\left(q \Rightarrow \mathbf{XG}\neg q\right)$ — q встретится в будущем не более одного раза;

б) $\mathbf{F}q \wedge \mathbf{G}\left(q \Rightarrow \mathbf{XG}\neg q\right)$ — q встретится в будущем точно один раз;

в) $p \Rightarrow \mathbf{F}q$ — на p, наступившем в начальном состоянии, когда-нибудь в будущем будет реакция q;

г) $\mathbf{G}\left(p \Rightarrow \mathbf{F}q\right)$ — на любое встретившееся в вычислении p всегда в будущем будет реакция q;

д) $\mathbf{G}\left(p \Rightarrow p\,\mathbf{U}q\right)$ — всегда если запрос p будет подан, реакция q на него обязательно будет получена, а до ее получения запрос p не сбросится;

е) $\mathbf{F}q \Rightarrow \left(\neg p\right)\mathbf{U}q$ — если событие q наступит, то до его наступления событие p не наступит.

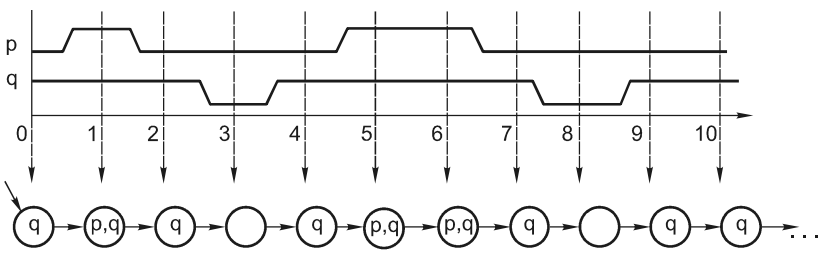


Рис. 2.10. Пример временной диаграммы и ее модели-вычисления

Пример 2.3

Свойства поведения дискретных логических схем тоже могут быть описаны с помощью формул логики LTL. На рис. 2.10 приведена временная диаграмма — график значений двух сигналов на выходе электронной схемы. Состояния схемы фиксируются в каждом такте.

Если единичное значение потенциала принять за логическое значение *true*, а низкое — за логическое значение *false*, то этой временной диаграмме можно сопоставить ее модель — вычисление, в каждом состоянии которого заданы истинностные значения атомарных предикатов p и q.

Считая, что сигналы на выходе этой схемы стабилизировались, на этом вычислении (назовем его σ) можно интерпретировать (определить истинностное значение) любую формулу LTL с атомарными предикатами p и q. В частности, для этой схемы выполняются следующие свойства:

- $\sigma \models \mathbf{FG}\neg p$: на вычислении σ когда-нибудь в будущем сигнал p сбросится и останется в этом сброшенном состоянии;
- $\sigma \models \mathbf{FG}q$: когда-нибудь в будущем сигнал q установится и останется в этом состоянии постоянно;
- $\sigma \models \mathbf{G}(p \Rightarrow q)$: в любом состоянии если установлен сигнал p, то и сигнал q установлен;
- $\sigma \models q\,\mathbf{U}(\neg p \wedge \neg q)$: оба сигнала, p и q, когда-нибудь в будущем будут сброшены, а до этого сигнал q будет все время установлен.

Последнее свойство выполняется на вычислении σ, потому что оно выполняется в начальном состоянии $\sigma(0)$ этого вычисления. Действительно, оба сигнала, p и q, в состоянии $\sigma(3)$ будут сброшены, а во всех предыдущих состояниях, $\sigma(2)$ $\sigma(1)$ и $\sigma(0)$, сигнал q установлен.

Формулы LTL могут служить для задания множеств бесконечных цепочек (так называемых ω-языков). Например, формула

$$(a \vee b)\mathbf{U}(\mathbf{G}b)$$

задает все бесконечные цепочки из a и b, содержащие только конечное число вхождений символа a (подробнее этот вопрос рассматривается в *гл. 4*).

2.7. Соотношения между операторами LTL

Будем говорить, что две темпоральные формулы φ и ψ логики LTL эквивалентны, и писать $\varphi \equiv \psi$, если они выражают одно и то же свойство бесконечных вычислений. Иными словами, $\varphi \equiv \psi$ означает, что для любого вычисления σ утверждение $\sigma \models \varphi$ верно тогда и только тогда, когда $\sigma \models \psi$ верно.

Операторы $\mathbf{U}$, $\mathbf{F}$ и $\mathbf{G}$ можно определить бесконечными формулами с помощью оператора $\mathbf{X}$:

$$p\,\mathbf{U}q \equiv q \vee \left(p \wedge \mathbf{X}q\right) \vee \left(p \wedge \mathbf{X}p \wedge \mathbf{XX}q\right) \vee \ldots$$

$$\mathbf{F}q \equiv q \vee \mathbf{X}q \vee \mathbf{XX}q \vee \mathbf{XXX}q \ \ldots$$

$$\mathbf{G}q \equiv q \wedge \mathbf{X}q \wedge \mathbf{XX}q \wedge \mathbf{XXX}q \ \ldots$$

Можно определить эти операторы рекурсивно, сами через себя (рис. 2.11):

$$p\,\mathbf{U}q \equiv q \vee \left(p \wedge \mathbf{X}\left(p\,\mathbf{U}q\right)\right)$$

$$\mathbf{F}q \equiv q \vee \mathbf{XF}q$$

$$\mathbf{G}q \equiv q \wedge \mathbf{XG}q$$

Эти определения ясны и без формального доказательства.

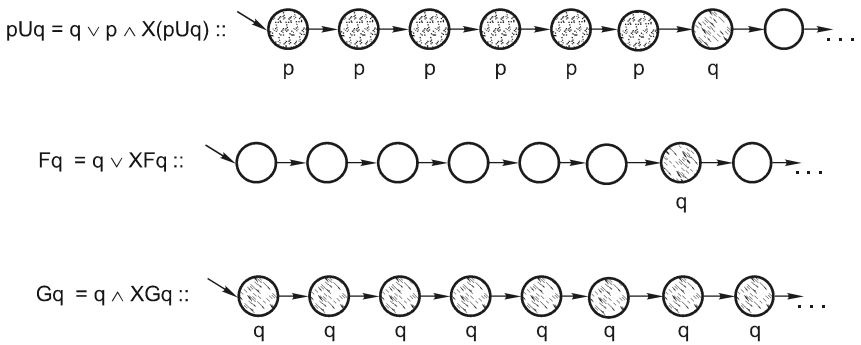


Рис. 2.11. Рекурсивное определение темпоральных операторов

Приведем несколько соотношений, которые легко могут быть доказаны на основании формального определения семантики формул LTL.

Комбинации темпоральных операторов и булевых операций:

$$\mathbf{X}\left(p \vee q\right) \equiv \mathbf{X}p \vee \mathbf{X}q$$

$$\mathbf{X}\left(p \wedge q\right) \equiv \mathbf{X}p \wedge \mathbf{X}q$$

$$\neg\mathbf{X}p \equiv \mathbf{X}\neg p$$

$$\mathbf{F}\left(p \vee q\right) \equiv \mathbf{F}p \vee \mathbf{F}q$$

$$\mathbf{G}\left(p \wedge q\right) \equiv \mathbf{G}p \wedge \mathbf{G}q$$

$(p \wedge q)\mathbf{U}r \equiv p\mathbf{U}r \wedge q\mathbf{U}r$

$p\,\mathbf{U}(q \vee r) \equiv p\mathbf{U}q \vee p\mathbf{U}r$

Законы поглощения (идемпотентность):

$\mathbf{FF}\,p \equiv \mathbf{F}p$

$\mathbf{GG}\,p \equiv \mathbf{G}p$

$p\,\mathbf{U}(p\mathbf{U}q) \equiv p\,\mathbf{U}\,q$

$(p\,\mathbf{U}q)\mathbf{U}q \equiv p\,\mathbf{U}\,q$

$\mathbf{FGF}\,p \equiv \mathbf{GF}p$

$\mathbf{GFG}\,p \equiv \mathbf{FG}p$

Существует множество других соотношений между операторами LTL. Например, общезначимые (всегда истинные) соотношения:

$\mathbf{G}p \Rightarrow \mathbf{F}p$ — "*то, что всегда будет, то когда-нибудь наступит*".

$p \Rightarrow \mathbf{F}p$ — "*то, что есть, то когда-нибудь наступит*".

$(\mathbf{G}p \vee \mathbf{G}q) \Rightarrow \mathbf{G}(p \vee q)$ — "*если всегда будет истинно p или всегда будет истинно q, то всегда будет истинно $p \vee q$*".

$\mathbf{G}(p \Rightarrow q) \Rightarrow (\mathbf{G}p \Rightarrow \mathbf{G}q)$ — "*если наступление p всегда влечет наступление q, то если p будет истинно всегда, то и q будет истинно всегда*".

2.8. Структура Крипке

Выбор подходящей математической модели является важнейшим этапом в исследовании объектов и явлений реального мира. Для исследования поведения реагирующих систем такой моделью является структура Крипке.

По заданному вычислению — бесконечной цепочке состояний с определенным в каждом состоянии набором атомарных предикатов, истинных в этом состоянии, мы определяли значения истинности булевых и темпоральных формул. Как такие вычисления представить конечным образом?

В качестве модели, которая конечным образом представляет бесконечные поведения, удобно выбрать систему переходов. Даже если система переходов имеет конечное число состояний, в общем случае количество возможных вычислений в ней бесконечно и каждое вычисление также бесконечно. Например, одно бесконечное вычисление (см. рис. 2.8) может быть представлено

конечной системой переходов (рис. 2.12, *б*), поскольку последнее состояние вычисления бесконечно повторяется.

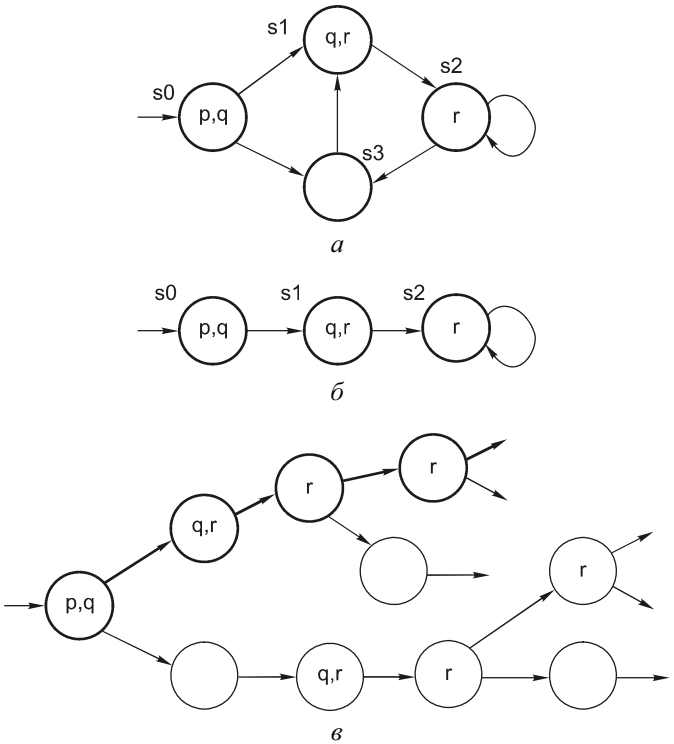


Рис. 2.12. Структура Крипке (*а*), одно из ее вычислений (*б*) и ее развертка — дерево вычислений (*в*)

Мы живем в линейном мире: после сегодняшнего дня будет завтра, потом послезавтра и т. п. В логике LTL формализован именно этот взгляд на время, как на линейную последовательность дискретных возрастающих значений (в частности, состояния любого вычисления можно пронумеровать натуральным рядом). Однако в жизни каждого человека встречаются точки ветвления, когда внешние события, принятые решения или выполненные действия могут направить жизнь либо по одному, либо по другому пути. Хотя у каждого есть возможности таких изменений, но реализуется только одна конкретная линия жизни из многих возможных.

Поведения реальных информационных систем также имеют альтернативы, выбор которых осуществляется на основе внешних событий, различий в содержании принятых сообщений и т. п. Поэтому в общем случае конкретное вычисление технической дискретной системы — лишь одно из многих воз-

можных ее вычислений. На рис. 2.12, *а* представлена модель системы переходов с конечным числом состояний. Одно из ее вычислений показано на рис. 2.4 и 2.8. На рис. 2.12, *в* показано дерево всех возможных вычислений модели 2.12, *а* — ее развертка.

Для формального задания таких систем переходов используется модель, которая называется "*структурой Крипке*". Структура Крипке имеет конечное множество состояний и (непомеченных) переходов между ними, причем каждое состояние помечено множеством истинных в этом состоянии атомарных предикатов.

❖ ***Определение 2.8 (структура Крипке)***

Структура Крипке M — это пятерка $M = (S, S_0, R, AP, L)$, где:

- S — конечное непустое множество состояний;
- $S_0 \subseteq S$ — непустое множество начальных состояний;
- $R \subseteq S \times S$ — тотальное отношение на S, т. е. множество переходов, удовлетворяющее требованию: $(\forall s \in S)(\exists s' \in S)(s, s') \in R$ (из любого состояния есть переход);
- AP — конечное множество атомарных предикатов;
- $L: S \to 2^{AP}$ — функция пометок (каждому состоянию отображение L сопоставляет множество истинных в нем атомарных предикатов).

Рис. 2.12, *а* задает структуру Крипке с четырьмя состояниями $\{s0, s1, s2, s3\}$, одним начальным состоянием $s0$, множеством атомарных предикатов $\{p, q, r\}$ и функцией пометок L: $L(s0) = \{p, q\}$, $L(s1) = \{q, r\}$, $L(s2) = \{r\}$, $L(s3) = \{\ \}$.

❖ ***Определение 2.9 (вычисление структуры Крипке)***

Вычислением структуры Крипке M называется любая бесконечная цепочка $\sigma = q_0\, q_1\, q_2\, q_3 \ldots$, такая, что $q_0 \in S_0$ и $(q_i, q_{i+1}) \in R$. Формально вычисление структуры Крипке M можно представить как отображение $\sigma: N \to S$, где N — множество натуральных чисел, т. е. $\sigma(i)$ — это некоторое состояние структуры Крипке.

Одно из вычислений структуры Крипке рис. 2.12, *а* — $s0\ s3\ s1\ s2\ s2\ s3 \ldots$.

❖ ***Определение 2.10 (траектория структуры Крипке)***

Траекторией структуры Крипке M, индуцированной вычислением $\sigma = q_0\, q_1\, q_2\, q_3 \ldots$ называется бесконечная цепочка $L(\sigma) = L(q_0)L(q_1)L(q_2)L(q_3)\ldots$, т. е. бесконечная цепочка подмножеств атомарных предикатов, истинных в соответствующих состояниях вычисления σ. Формально, траектория — это отображение натурального ряда в множество 2^{AP}.

Траекторией структуры Крипке рис. 2.12, *а* при вычислении $s0\ s3\ s1 s2\ s2 s3 \ldots$ является следующая цепочка: $\{p,q\}\{\ \}\{q,r\}\{r\}\{r\}\ldots$.

Траектории структуры Крипке иногда называют *трассами*. Траектории являются последовательностями подмножеств атомарных предикатов — наблюдаемых свойств (событий, соотношений между переменными и т. п.), которые могут выполниться (произойти) в анализируемых системах. Именно эти последовательности являются важными для анализа. Формулы темпоральной логики описывают свойства траекторий.

В структуре Крипке в каждом состоянии указаны атомарные предикаты из конечного множества AP атомарных предикатов, которые истинны (выполняются) в этом состоянии. Каждое такое подмножество атомарных предикатов можно считать символом, помечающим состояние структуры Крипке. В теории формальных языков конечные цепочки, построенные над конечным словарем, называются словами, а бесконечные цепочки называются ω-словами. Каждая траектория структуры Крипке является бесконечной. Поэтому траектории структуры Крипке называются также ее ω-словами.

Пример 2.4

Вычисления, показанные на рис. 2.12, *в* индуцируют следующие траектории (ω-слова) — бесконечные цепочки подмножеств множества атомарных предикатов, которые выполняются в последовательно проходимых состояниях:

$\pi_1 = \{p,q\}\{q,r\}\{r\}\{\ \}\ldots;$

$\pi_2 = \{p,q\}\{q,r\}\{r\}\{r\}\ldots;$

$\pi_3 = \{p,q\}\{\ \}\{q,r\}\{r\}\{\ \}\ldots;$

$\pi_4 = \{p,q\}\{\ \}\{q,r\}\{r\}\{r\}\ldots.$

Множество всех ω-слов структуры Крипке M называется ω-языком, допускаемым M.

Итак, в отличие от конечного автомата, каждое состояние структуры Крипке помечено некоторым множеством атомарных утверждений, истинных в этом состоянии. Структуру Крипке можно считать расширением конечного автомата, в котором отсутствуют пометки на переходах, поскольку цель структуры Крипке — задать конечным образом *бесконечные последовательности* состояний (вычисления) при произвольных входах. В структуре Крипке мы не рассматриваем причины переходов — конкретные внешние события, существенные в модели конечного автомата. Такие события легко могут быть учтены в модели Крипке, но при верификации нам чаще всего важно проверить, существует ли некорректное поведение системы при любых последовательностях внешних событий, и только когда такое некорректное поведение найдено, мы можем искать и его причину. Например, верификация системы управления медицинского прибора облучения Therac-25 должна была гарантировать, что при всех возможных нажатиях его кнопок медицинским персоналом прибор не сможет войти в режим, в котором интенсивность облучения пациента превышала бы заданную границу. Другой пример: разработчики бортовой системы управления советских межпланетных станций "Фобос-1" и "Фобос-2" должны были гарантировать, что любые случайные последовательности команд с Земли не приведут систему в неуправляемое состояние, в котором отключены системы стабилизации, ориентации и связи.

Тотальность множества переходов означает, что из каждого состояния существует хотя бы один переход. Если моделируемая система останавливается в каком-либо завершающем состоянии, то в соответствующей ей модели — структуре Крипке — указывается переход в то же самое состояние. Таким образом, структура Крипке, все вычисления которой бесконечны, может служить моделью и завершающихся вычислений.

Структура Крипке является семантической моделью реагирующих систем. Логические системы управления, протоколы коммуникации и параллельные взаимодействующие программы — все эти системы достаточно адекватно могут быть представлены структурой Крипке (*см. гл. 5*).

Поскольку при анализе реагирующих систем нас интересуют бесконечные вычисления и соответствующие им траектории (ω-слова), то все состояния структуры Крипке можно считать принимающими состояниями в том смысле, что любое ω-слово, которое индуцируется каким-либо вычислением структуры Крипке M, будет допускаться M.

Главным достоинством этой простой модели — структуры Крипке — является возможность конечным образом задавать поведения дискретных систем: бесконечное число бесконечных вычислений и индуцированных ими траекторий.

2.9. Расширенная темпоральная логика ветвящегося времени CTL*

Очевидно, что формулы линейной темпоральной логики могут описать свойства конкретного вычисления — одного из возможных путей, начинающегося из любого состояния структуры Крипке. Поэтому формулы LTL называют *формулами пути*. Структура Крипке представляет конечным образом бесконечные вычисления, но в процессе вычислений заставляет делать альтернативный выбор. У каждой альтернативной ветви будут истинны свои свойства, свои темпоральные LTL-формулы. Например, формула $\mathbf{XG}\,r$ будет истинна не для всех вычислений структуры Крипке (рис. 2.12, *а*), а только для одной из них, выделенной на рис. 2.12, *в*. Поэтому для каждой формулы пути на структуре Крипке желательно указать, для какого именно вычисления (для какого пути) проверяется истинность этой формулы.

Каждое состояние структуры Крипке является началом бесконечного числа вычислений (корнем дерева вычислений), поэтому указать точно конкретную ветвь, для которой данная формула истинна, невозможно. Можно указать, однако, что некоторая формула пути, характеризующая вычисление, истинна *на каком-то из путей* или, например, что данная формула пути истинна *на всех путях*, начинающихся из данного состояния. Пусть символ **E** (от *Exist*) перед темпоральной формулой пути φ утверждает, что "***существует*** вычисление, на котором формула φ истинна". Если формула $\mathbf{E}\varphi$ истинна в начальном состоянии структуры Крипке M, то будем говорить, что она выполняется для M и писать $M \models \mathbf{E}\varphi$ со следующей интерпретацией: "на структуре Крипке M из ее начального состояния существует вычисление, на котором выполняется формула φ".

Пример 2.5

Из рис. 2.12, *в* можно видеть, что формула $\mathbf{EXG}\,r$ истинна для структуры Крипке M, т. е. $M \models \mathbf{EXG}r$. Действительно, из *начального* состояния структуры Крипке M существует (**E**) такой путь (а именно $s0s1s2s2s2...$), что со следующего состояния этого пути (**X**) во всех состояниях этого пути (**G**) выполняется атомарный предикат *r*.

Очевидно, что темпоральная формула $\mathbf{E}\varphi$ с квантором пути **E** характеризует уже не вычисление, а *состояние* структуры Крипке, из которого *существует* вычисление (начинается ветвь вычисления), удовлетворяющее темпоральной формуле пути φ. Формально возможность спецификации всех таких свойств дается расширенной темпоральной логикой ветвящегося времени

CTL* (Extended Computational Tree Logic). В этой логике, в дополнение к темпоральным операторам *Until* и *NextTime*, введен квантор пути $\mathbf{E}$. В CTL* могут присутствовать как формулы пути (характеризующие некоторое вычисление), так и формулы состояния, характеризующие состояние.

❖ ***Определение 2.11 (логика CTL*)***

Формулы CTL* — это формулы состояний, они задаются следующей грамматикой:

$\Phi ::=$ /* формулами CTL* являются следующие формулы состояний: */

- $p \mid q \mid \ldots \mid$ /* все атомарные предикаты; */
- $\neg\Phi \mid \Phi \vee \Phi \mid$ /* формулы состояний, связанные булевыми операциями; */
- $\mathbf{E}\varphi$ /* формулы пути φ, связанные квантором пути $\mathbf{E}$. */

Возможные формулы пути логики CTL* задаются следующей грамматикой:

$\varphi ::=$ /* формулами пути логики CTL* являются следующие формулы: */

- $\Phi \mid \ldots$ /* формулы состояний этой логики (они характеризуют пути, начинающиеся в этом состоянии); */
- $\neg\varphi \mid \varphi \vee \varphi \mid$ /* формулы пути, связанные булевыми операциями; */
- $\mathbf{X}\varphi \mid \varphi \mathbf{U} \varphi$ */ формулы пути, связанные темпоральными операторами $\mathbf{X}$ и $\mathbf{U}$. */

В дополнение к определению формул логики LTL, в логике CTL* добавляется только один квантор $\mathbf{E}$ — "*существует такой путь, что...* ". Для удобства вводится также квантор пути $\mathbf{A}$ (от *All*) как сокращенное обозначение: $\mathbf{A}\varphi = \neg\mathbf{E}\neg\varphi$. Формула $\mathbf{A}\varphi$ понимается так: "*на **всех** путях из данного состояния формула пути* φ *истинна*". Формулы $\mathbf{E}\varphi$ и $\mathbf{A}\varphi$ являются формулами состояния, они характеризуют состояние структуры Крипке. Как и в логике LTL, в качестве формул пути кроме $\mathbf{X}\varphi$ и $\varphi\mathbf{U}\varphi$ также используются формулы $\mathbf{F}\varphi$ и $\mathbf{G}\varphi$ как сокращения:

- $\mathbf{F}\varphi$ можно считать сокращением для $true\,\mathbf{U}\,\varphi$;
- $\mathbf{G}\varphi$ можно считать сокращением для $\neg\mathbf{F}\neg\varphi$.

Квантор пути **E** понимается как "*возможное*" (possibly) выполнение следующей за ним темпоральной формулы пути: эта формула пути выполняется, по крайней мере, на одном из вычислений, начинающихся из данного состояния. Квантор пути **A** можно понимать как "*неизбежное*" (inevitably) выполнение следующей за ним темпоральной формулы пути при всех вариантах развития событий из данного состояния.

В определении CTL* формула состояния может считаться формулой пути в следующем смысле: формула состояния выполняется на некотором пути, если она выполняется в начальном состоянии этого пути. Для каждой формулы пути в состоянии необходимо указывать, какому вычислению, начинающемуся в этом состоянии, соответствует формула пути. Если явного указания нет, то предполагается, что формула пути относится ко всем возможным вычислениям, начинающимся из данного состояния.

Формула Φ логики CTL* выполняется на структуре Крипке M (обозначается $M \models \Phi$), если Φ выполняется в начальном состоянии структуры M. Две формулы Φ_1 и Φ_2 логики CTL* эквивалентны (записывается $\Phi_1 \equiv \Phi_2$), когда для всякой структуры Крипке M утверждение $M \models \Phi_1$ верно тогда и только тогда, когда верно утверждение $M \models \Phi_2$.

В практике верификации распространение получили два подмножества логики CTL*, определенные ограничениями на использование комбинаций темпоральных операторов и кванторов.

Одно из них — это логика линейного времени LTL, все формулы которой — это формулы пути, которые интерпретируются на вычислениях, имеющих единственное будущее. Недетерминизм моделируемой системы, существование альтернатив в поведении структуры Крипке учитывается только тем, что при верификации анализируются ***все возможные*** вычисления структуры Крипке. Если на структуре Крипке M проверяется выполнение формулы LTL φ, то это значит, что проверяется утверждение: "*На **всех** вычислениях M выполняется формула* φ ". Поэтому любую формулу пути φ логики LTL можно представить в логике CTL* формулой состояния $\mathbf{A}\varphi$ (на всех путях из начального состояния формула φ выполняется) явным добавлением перед φ квантора пути **A**. Именно в этом смысле логика LTL может рассматриваться как подмножество CTL*, хотя природа времени в этих логиках различна: в логике LTL — линейное время, в логике CTL* — ветвящееся время. Логика LTL и в своей нотации, и в концепциях очень проста, что делает ее весьма популярной в применениях.

Другим подмножеством логики CTL* является логика CTL (Computational Tree Logic) — *логика ветвящегося времени*. В этой логике допустимы только

формулы, в которых каждый темпоральный оператор **X**, **U**, **F** и **G** (характеризующий некоторое вычисление) предваряется квантором пути — **A** или **E**, что превращает любую темпоральную формулу пути в формулу, характеризующую состояние. Формулы этой логики, как и формулы расширенной логики CTL*, интерпретируются на деревьях вычислений структуры Крипке, в каждом узле которых существует не одно-единственное будущее, а несколько альтернатив будущего. Следовательно, время в этих логиках представляется ветвящейся структурой в каждом текущем состоянии.

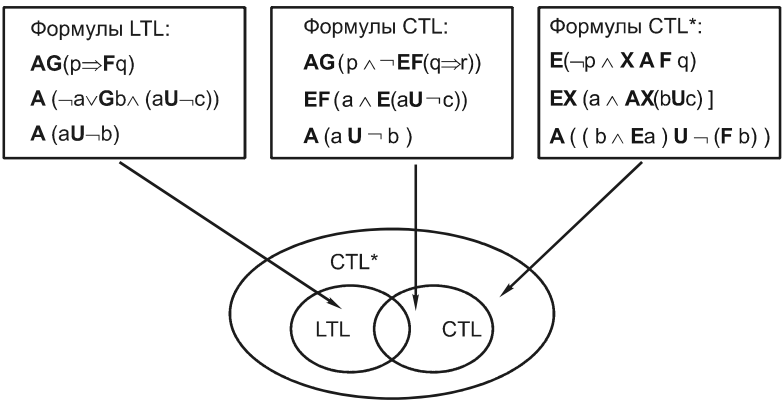


Рис. 2.13. Соотношения темпоральных логик ([34])

На рис. 2.13 представлены соотношения темпоральных логик LTL, CTL и CTL*. Логики LTL и CTL являются пересекающимися подклассами общей темпоральной логики ветвящегося времени CTL*. Все формулы логики LTL являются формулами путей, все формулы CTL являются формулами состояний. Логики LTL и CTL являются собственными подмножествами логики CTL*, причем обе эти логики между собой несравнимы. Действительно, синтаксически формула CTL* может быть построена как произвольная комбинация темпоральных операторов и кванторов пути, формула LTL имеет вид $\mathbf{A}\varphi$, где φ — формула, не содержащая кванторов пути, но может содержать произвольную комбинацию темпоральных операторов, а логика CTL включает только формулы, в которых каждому темпоральному оператору непосредственно предшествует квантор пути. Ниже приводится синтаксис (грамматические правила построения формул) всех трех логик (в логиках LTL и CTL* возможно также использование выводимых операторов).

LTL формулы состояний $\Phi ::= \mathbf{A}\varphi$

формулы путей $\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{X}\varphi \mid \varphi \mathbf{U} \varphi$

==========

CTL формулы состояний $\Phi ::= p \mid \neg\Phi \mid \Phi \vee \Phi \mid \mathbf{E}\varphi \mid \mathbf{A}\varphi$

формулы путей $\varphi ::= \mathbf{X}\Phi \mid \Phi\,\mathbf{U}\,\Phi \mid \mathbf{F}\Phi \mid \mathbf{G}\Phi$

==========

CTL* формулы состояний $\Phi ::= p \mid \neg\Phi \mid \Phi \vee \Phi \mid \mathbf{E}\varphi$

формулы путей $\varphi ::= \Phi \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{X}\varphi \mid \varphi\,\mathbf{U}\,\varphi$

==========

Некоторые формулы относятся к обоим классам. Например, формула $\mathbf{A}(a\,\mathbf{U}\neg b)$ является как формулой LTL, так и формулой CTL. Формула $\mathbf{AG}(p \Rightarrow \mathbf{F}q)$ является формулой LTL, она не является формулой CTL. Формула $\mathbf{AG}(p \Rightarrow \mathbf{EF}q)$ является формулой CTL, она не является формулой LTL. Все эти формулы являются формулами логики CTL*.

Формулы логики CTL* интерпретируются на вычислениях структуры Крипке, которые можно представить бесконечным деревом (рис. 2.12, *в*). Логика CTL* более мощная, чем и логика LTL, и логика CTL. Формулы CTL* могут различать не только вычисления (линейные пути поведений систем), но и разветвления, деревья поведений.

Пример 2.6

Строки А. С. Пушкина

Летят за днями дни, и каждый час уносит
Частичку бытия, а мы с тобой вдвоем
Предполагаем жить, и глядь — как раз — умрем

как и многие другие высказывания естественного языка, могут быть формализованы в рамках темпоральной логики по-разному, хотя, несомненно, здесь требуется использование именно логики ветвящегося времени. Одна из возможных формализаций такая:

$$\varphi_{br} = \mathbf{A}(\mathbf{G}\,life \Rightarrow \mathbf{GEX}\,death)$$

Эта формула логики CTL* [133] утверждает, что вдоль всех вычислений ($\mathbf{A}$) с вечной жизнью ($\mathbf{G}\ life$) всегда ($\mathbf{G}$) возможно ($\mathbf{E}$) встретить смерть в следующий момент ($\mathbf{X}\ death$). Строфа Пушкина как раз и говорит о возможности в любой момент переключения нашего бытия на другую ветвь, на которой нас (к сожалению) ожидает смерть. Структура Крипке, которая соответствует строкам Пушкина, представлена на рис. 2.14, *а*: в каждый момент (в каждом состоянии) перед нами альтернатива (которую обычно выбираем

не мы!): или жить дальше, или умереть. Особенно четко это показывает развертка структуры Крипке, показанная на рис. 2.14, *б*. Формула φ_{br} на этой структуре Крипке выполняется.

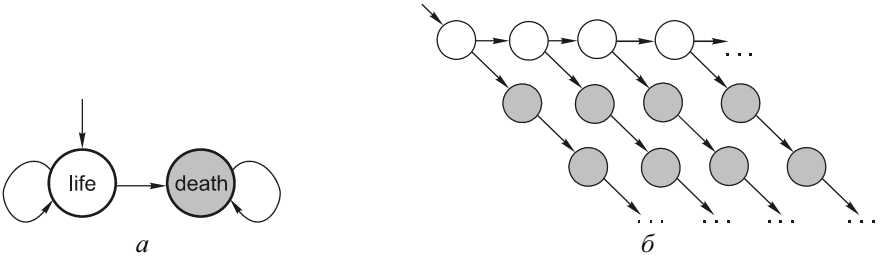


Рис. 2.14. Структура Крипке и ее развертка (дерево вычислений)

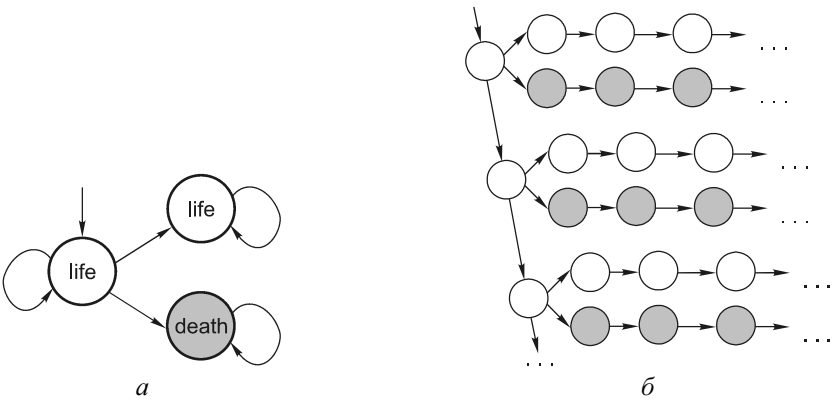


Рис. 2.15. Структура Крипке (*а*) и ее дерево поведений (*б*)

Отметим, что на структуре Крипке (рис. 2.15) эта формула не выполняется: ее развертка имеет вычисления бесконечной жизни, на которых нет ветвления на состояния, помеченные *death*. Структуры Крипке, показанные на рис. 2.14 и 2.15, имеют одно и то же множество вычислений (рис. 2.16), которое может быть выражено ω-регулярным выражением $\left\{life^{\omega} \cup life^{+}death^{\omega}\right\}$: одна бесконечная цепочка *life* и множество цепочек с любым (ненулевым) конечным числом *life*, за которыми следует бесконечное число *death*. Следствием того, что обе структуры Крипке имеют одно и то же множество линейных вычислений, является то, что никакая формула LTL не может их различить. Этот пример, в частности, показывает, что логика CTL* более мощная, чем логика LTL: формулы CTL* различают эти две структуры Крипке.

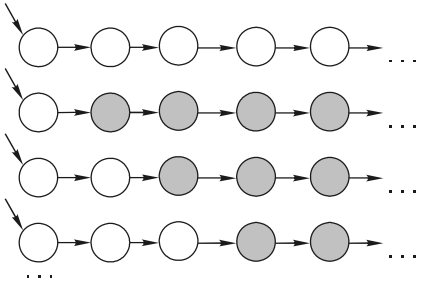


Рис. 2.16. Множество вычислений структур Крипке рис. 2.14 и 2.15

Пример 2.7

Комбинации кванторов путей **A** и **E** с темпоральными операторами **F** и **G** имеют достаточно очевидный смысл:

- **AG** φ — утверждение φ — инвариант (φ выполняется всегда, в любом состоянии);
- **EG** φ — потенциально всегда φ (возможно, что φ будет выполняться все время);
- **AF** φ — когда-нибудь φ обязательно выполнится;
- **EF** φ — когда-нибудь, возможно, φ выполнится.

Пример 2.8

Пусть p означает "*Я люблю Машу*", q означает "*Я люблю Дашу*". Тогда:

- формула $\mathbf{AG}\,p$ является формулой CTL, она формализует следующее высказывание: "*Я люблю Машу, и, что бы ни случилось, я буду любить ее всегда*";
- формула $\mathbf{AFG}\left(p \wedge \neg q\right)$ является формулой LTL, она формализует следующее высказывание: "*Что бы ни случилось, в будущем я полюблю Машу навсегда, а Дашу не буду любить!*";
- формула $\mathbf{E}\left(p\,\mathbf{U}\left(\mathbf{AG}q\right)\right)$ — формула CTL, она формализует высказывание: "*Я не исключаю такого развития событий* (**E**), *что я буду любить Машу до тех пор* (**U**), *пока, наконец, я не полюблю Дашу навечно* (**AG**)".

2.10. Сравнение логик LTL и CTL

Для обеих логик, LTL и CTL, разработаны эффективные алгоритмы проверки выполнимости их формул на структурах Крипке. Эти алгоритмы реализованы в нескольких системах верификации, как свободно распространяемых, так и используемых внутри компаний — производителей ПО и аппаратуры, и они используются в рамках корпоративных технологических циклов разработки систем. В следующих двух главах мы рассмотрим технику верификации для каждой из этих двух логик.

Существует множество черт, которые отличают эти логики друг от друга, но главные различия необходимо подчеркнуть особо:

- Формулы этих двух логик характеризуют свойства разных типов:
  - формулы LTL являются формулами пути; в логике LTL вообще нет *кванторов* пути. При проверке выполнения свойств структуры Крипке единственный квантор пути **A** перед формулой LTL говорит, что эта формула должна выполняться на *всех возможных* путях этой структуры Крипке. Логика LTL более проста и интуитивна, более понятна, чем CTL;
  - формулы CTL являются формулами состояний; в логике CTL любой темпоральный оператор предварен квантором пути, который определяет, на всех (квантор **A**) или только на некоторых (квантор **E**) путях из текущего состояния структуры Крипке утверждается истинность соответствующей темпоральной формулы пути.
- Формулы обеих логик интерпретируются (т. е. принимают истинное или ложное значение) на структурах Крипке, но:
  - формулы логики LTL интерпретируются на всех вычислениях структуры Крипке, начинающихся в начальном состоянии. Они принимают истинное или ложное значение на каждом вычислении структуры Крипке. Формула φ LTL выполняется на структуре Крипке M, если φ выполняется на *всех* траекториях вычислений M (в *начальном состоянии* каждого бесконечного вычисления этой структуры). Вычисления эти бесконечны, и их бесконечное число. Этим определяются бóльшие трудности проверки выполнения формулы LTL на структуре Крипке, чем проверка формул CTL;
  - формулы логики CTL интерпретируются на деревьях вычислений структуры Крипке. Дерево вычислений — это развертка структуры Крипке с корнем в данном состоянии. Любая формула CTL принимает истинное или ложное значение в каждом состоянии структуры Крипке,

а таких состояний конечное число. Этим определяется удивительная эффективность алгоритма *model checking* для формул CTL. Формула φ CTL выполняется на структуре Крипке M, если φ выполняется на *начальном состоянии* структуры Крипке, в корне дерева вычислений M.

- Техника верификации для LTL и CTL совершенно различна как по применяемым алгоритмам, так и по их сложности:
  - существуют инструменты верификации, которые реализуют алгоритмы Model checking для LTL, например, Spin. Сложность алгоритмов верификации линейна относительно числа состояний структуры Крипке и экспоненциальна относительно сложности (числа подформул) формулы LTL, но поскольку проверяемые формулы LTL обычно малы, возможно использование этих алгоритмов для верификации достаточно сложных систем;
  - существуют другие инструменты, реализующие алгоритмы верификации для формул CTL, например, SMV. Сложность алгоритмов верификации для CTL пропорциональна числу состояний структуры Крипке и сложности (числу подформул) формулы CTL.
- Многие свойства технических систем могут быть выражены как формулами логики CTL, так и формулами логики LTL. Например:
  - "*каждое сообщение когда-нибудь в будущем будет подтверждено*":
    - CTL: $\mathbf{AG}\left(send \Rightarrow \mathbf{AF}\, req\right)$
    - LTL: $\mathbf{G}\left(send \Rightarrow \mathbf{F}\, req\right)$

  Однако *неверно* утверждение, что если в формуле LTL каждый темпоральный оператор будет предварен квантором пути $\mathbf{A}$, то построенная так формула CTL будет эквивалентна исходной формуле (см. задачу 2.7).
- Выразительная мощность этих двух логик несравнима, т. е. некоторые свойства могут быть выражены в CTL, но не могут быть выражены в LTL, и наоборот:
  - например, формула $\mathbf{AGEF}\varphi$ логики CTL, выражающая "*достижимость свойства* φ", не выразима в LTL;
  - например, формула $\mathbf{FG}\varphi$ логики LTL, выражающая "*стабилизацию свойства* φ *когда-нибудь в будущем*", не выразима в CTL.

2.11. Темпоральные логики прошлого

В философии, лингвистике и теории искусственного интеллекта анализ утверждений, зависящих от времени, обычно проводится как для будущего времени, так и для прошлого. В технике рассмотрение формул темпоральных логик почти всегда связано с будущим временем. Это обусловлено тем, что основное применение темпоральных логик в технике — анализ будущего поведения вновь создаваемых технических систем. Однако в некоторых случаях и для технических систем удобно выражать темпоральные свойства их прошлого поведения. В частности, причинные зависимости иногда легче транслировать в формулы темпоральных логик с модальностями прошлого. Естественно для этого ввести так называемые "темпоральные операторы прошлого", например, $\mathbf{F}^{-1}$ и $\mathbf{X}^{-1}$. Формула $\mathbf{F}^{-1}\psi$ понимается так: "когда-то в прошлом утверждение ψ было истинным" и эквивалентна формуле Tense Logic $\mathbf{P}\psi$, а формула $\mathbf{X}^{-1}\psi$ понимается так: "непосредственно в предыдущем состоянии утверждение ψ было истинным". Зеркальным отображением оператора $\mathbf{U}$ является оператор $\mathbf{S}$ (*since*). Темпоральная формула $\varphi\mathbf{S}\psi$ понимается так: "формула ψ выполнилась когда-то в прошлом, и с тех пор выполняется формула φ".

Используя темпоральные операторы прошлого, свойство:

"*Пока ключ зажигания не вставлен, машина не поедет*"

наиболее естественно формализовать так:

$$\mathbf{AG}\left(start \Rightarrow \mathbf{F}^{-1} ключ_зажигания_вставлен\right)$$

— всегда верно ($\mathbf{AG}$), что если машина стартовала (*start*), то предварительно ($\mathbf{F}^{-1}$) ключ зажигания был вставлен.

В *гл. 6* мы увидим, что операторы прошлого не увеличивают экспрессивности темпоральной логики, и формулы с модальными операторами прошлого для структур Крипке можно выразить через операторы будущего, введя дополнительные атомарные предикаты.

2.12. Model checking

Model checking — это совокупность моделей, приемов и алгоритмов, позволяющих проверить, что формула темпоральной логики (CTL*, CTL или LTL), выражающая некоторое свойство поведения динамической системы во времени, выполняется (является истинной) на модели системы с конечным числом состояний (структуре Крипке).

За последние десятилетия структуры Крипке стали общепризнанной моделью реагирующих систем, с помощью которой можно адекватно представить поведение реагирующих систем: дискретных систем управления, параллельных и распределенных алгоритмов, протоколов и т. п., а темпоральные логики признаны очень эффективным формализмом для выражения свойств таких систем. Стандартными шагами доказательства того, что поведение реагирующей системы обладает некоторым свойством, являются следующие:

1. Для верифицируемой системы строится адекватная модель Крипке M, т. е. система переходов с конечным числом состояний. Поведения реальной системы представляются разверткой (деревом вычислений) построенной структуры Крипке.
2. С помощью переменных и параметров верифицируемой системы выражаются интересующие разработчика атомарные предикаты структуры Крипке — логические выражения, которые могут принимать значения "истина" или "ложь" в каждом состоянии системы.
3. Проверяемое свойство выражается формулой φ темпоральной логики с использованием атомарных утверждений, темпоральных операторов и кванторов пути.
4. Проверяется истинность утверждения $M \models \varphi$ (т. е. утверждения, что структура Крипке является моделью формулы φ) с помощью полностью автоматизированной процедуры.

Метод верификации, реализующий эти шаги, и называется *model checking*.

2.13. Заключение

Задачей временной логики является построение формализованных языков, способных сделать рассуждения о предметах и явлениях, развивающихся во времени, более ясными и точными, и потому более удобными для формального анализа.

Темпоральные логики позволяют формулировать утверждения о порядке наступления событий во времени без явного указания времени. Наиболее удобными эти логики оказались в применении к спецификации свойств параллельных программ, в которых важны именно порядки событий, а не явные значения времени наступления этих событий. Л. Лэмпорт даже определяет темпоральную логику как "формальную систему для спецификации свойств и доказательства их выполнения в параллельных программах на всех уровнях их абстрактного описания" [94].

Существует несколько вариантов темпоральных логик. В области верификации технических систем используются, в основном, две такие логики — ло-

гика линейного времени LTL и логика ветвящегося времени CTL. Обе логики интерпретируются на структуре Крипке — недетерминированной конечной системе переходов, удобной для представления динамики поведений дискретных систем. Логика линейного времени рассматривает все вычисления структуры Крипке как множество бесконечных траекторий поведения системы (трасс). Формулы этой логики построены из атомарных утверждений, связанных логическими операциями и темпоральными операторами. Логика ветвящегося времени рассматривает возможные варианты развития вычислений в каждом состоянии бесконечных траекторий.

Развитие этих двух типов темпоральных логик частично объясняется существованием двух разных школ, каждая из которых развивала свою версию темпоральной логики. Между этими школами существовало даже некоторое соперничество (см., например, название статьи [46]: "*Модальности для model checking: логика ветвящегося времени наносит ответный удар*"). До сих пор остается открытым вопрос о том, какая из логик более удобна и выразительна для спецификации свойств реагирующих систем. Обе логики могут быть использованы для формулировки утверждений о поведении технических систем. Существуют свойства систем, выразимые в одной логике и невыразимые в другой. Достаточно широко распространено мнение, что свойства систем легче выражать в логике LTL, в то время как алгоритмы верификации проще для формул, выраженных в CTL [140].

2.14. Замечания

Первые попытки учесть роль временного фактора в логических утверждениях относятся еще к античности. Идея введения временных понятий в логику восходит к греческому философу Diodorus Cronus, который жил в IV веке до нашей эры и был современником Аристотеля. Сам Аристотель писал, что конкретное значение истинности высказываний не является раз и навсегда заданным, а зависит от ситуации, в которой эти высказывания делаются. Но только в 60-х годах прошлого века английским логиком Артуром Прайором (Arthur Prior) была впервые выполнена формализация темпоральной логики, которая была им названа Tense Logic [128], [184]. Во временной логике А. Прайора были впервые введены темпоральные операторы **F** и **P**. Эти операторы (и выводимые из них операторы **G** и **H**) вошли в так называемую Minimal Tense Logic, которая впоследствии была расширена во многих направлениях. Различные расширения Tense Logic изучаются и в наше время.

В лингвистике одну из первых попыток формального анализа предложений, ссылающихся на время, сделал Ганс Райхенбах, современник Эйнштейна и Макса Планка. Он писал научные работы по квантовой механике, о природе времени, по вероятности, логике и философии науки. Его идея о возможности

формализации времен английских глаголов с помощью соотношения моментов наступления трех событий S, R и E была высказана в его книге [130].

Использование темпоральной логики для целей спецификации свойств и верификации технических систем связано с именем Амира Пнуэли (Amir Pnueli), профессора Weizmann Institute of Science, Израиль. До него темпоральные логики использовались только в лингвистике и философии для анализа высказываний и рассуждений естественного языка о событиях, происходящих во времени. В 70-х годах прошлого века А. Пнуэли разработал темпоральную логику линейного времени (Linear Temporal Logic, [125]) как развитие логики Артура Прайора для спецификации параллельных вычислительных систем. С тех пор темпоральные логики стали основным инструментом выражения свойств таких систем. А. Пнуэли также выделил в отдельный класс "реагирующие системы" (*reactive systems*) как особые программные и аппаратные системы, для которых необходимы свои формальные модели и свои особые методы анализа. В 1996 г. Амир Пнуэли был награжден Тьюринговской премией ACM "*за основополагающую работу, которая ввела темпоральную логику в информатику, и за выдающийся вклад в верификацию программ и систем*".

Формальная модель "*структура Крипке*" введена в [90] Саулом Крипке (Saul Kripke), американским философом и логиком, профессором Принстонского университета. Темпоральная логика ветвящегося времени была введена и исследована Эдмундом Кларком (Edmund M. Clarke), профессором университета Карнеги-Меллон, и Алленом Эмерсоном (E. Allen Emerson) в работах [31], [30] и многих других. Полная темпоральная логика CTL* введена Эмерсоном и Халперном в 1986 г. Сравнение различных темпоральных логик можно найти в [48].

2.15. Задачи к главе 2

2.1. Постройте в модели Г. Райхенбаха соотношение моментов наступления событий E, R и S для следующих предложений английского языка и расположите эти события на временной оси:

а) Simple Present: *I see John*;

б) Past Perfect: *I had seen John*;

в) Simple Future: *I shall see John*;

г) Future Perfect: *I shall have seen John*.

2.2. Пусть p означает "*Я люблю Машу*", а q — "*Я люблю Дашу*". Каким высказываниям соответствуют следующие формулы LTL:

а) $\mathbf{F}(p \Rightarrow \neg q)$

б) $p\mathbf{UG}q$

2.3. Представьте в виде формулы LTL высказывание: "Канал может терять сообщение только конечное число раз".

2.4. Постройте LTL-формулу, которая будет задавать вычисления со следующими свойствами:

а) если произойдет p, то q никогда не случится;

б) если произойдет p, то когда-нибудь в будущем выполнится q, а сразу после этого произойдет r;

в) если произойдет p, то затем произойдет q, но между ними не произойдет r;

г) предикаты p и q выполняются попеременно (т. е. после p не встретится p, пока не встретится q);

д) если свойство p становится истинным после того, как было ложным, то оно будет истинным точно 2 шага.

2.5. Переведите формулу $\neg(p\mathbf{U}q)$ в отрицательную нормальную форму, т. е. в форму, в которой отрицания стоят только непосредственно у литералов.

2.6. Какие из следующих формул являются CTL-формулами? Какие из них LTL-формулы?

1. $p \vee \neg q$
2. $\mathbf{AF}(p \vee \mathbf{AGX}\neg q)$
3. $\mathbf{A}(p\mathbf{U}(\mathbf{AX}q))$
4. $\mathbf{EGAF}(p\mathbf{U}q)$
5. $\mathbf{EF}(p \wedge \neg\mathbf{AG}q)$
6. $\mathbf{A}(p \vee \mathbf{X}q \vee \mathbf{XX}(p\mathbf{U}\neg q))$

2.7. Известно, что если в формуле Φ LTL каждый темпоральный оператор будет предварен квантором пути $\mathbf{A}$, то такая формула CTL не всегда будет эквивалентна исходной формуле Φ.

а) Проверьте, выполняются ли LTL-формула $\mathbf{FG}\,p$ и CTL-формула $\mathbf{AFAG}\,p$ на структуре Крипке $M1 = (S, S_0, R, AP, L)$, где:

$S=\{s_0,s_1,s_2\}$, $S_0=\{s_0\}$, $R=\{(s_0,s_0),(s_0,s_1),(s_1,s_2),(s_2,s_2)\}$, $AP=\{p\}$, $L(s_0)=L(s_2)=\{p\}$,

$L(s_1)=\varnothing$.

б) Проверьте, выполняются ли LTL-формула $\mathbf{GF}\,p$ и CTL-формула $\mathbf{AGEF}\,p$ на структуре Крипке $M2=(S,S_0,R,AP,L)$, где:

$S=\{s_0, s_1\}$, $S_0=\{s_0\}$, $R=\{(s_0,s_0),(s_0,s_1),(s_1,s_1)\}$, $AP=\{p\}$, $L(s_0)=\varnothing$, $L(s_1)=\{p\}$

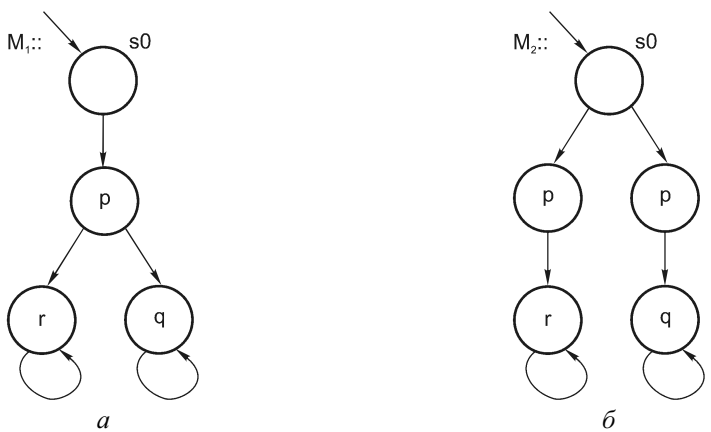
2.8. Можно ли выразить любую формулу CTL* с помощью операторов $\neg$, $\vee$, **X**, **U**, **E**? Если да, то выразить с их помощью формулу $\mathbf{AF}(p\Rightarrow\mathbf{X}(q\mathbf{U}r))$. Постройте структуру Крипке, на которой эта формула выполняется.

2.9. Логики CTL и LTL несравнимы по своей выразительной мощности. Существуют свойства поведений, которые могут быть выражены в одной из логик и не могут быть выражены в другой. Для того чтобы в этом убедиться, выполните следующее.

а) Докажите теорему: Существуют свойства поведений, выражаемые в CTL и не выражаемые в LTL.

Для этого для двух разверток структур Крипке (рис. 2.17):

i) покажите, что никакая формула LTL не может различить эти две структуры (***Указание***: рассмотрите возможные вычисления M_1 и M_2);



а *б*

Рис. 2.17. К задаче 2.9

ii) постройте формулу CTL, которая истинна на одной из структур Крипке и ложна на другой.

❖ ***Указание***

Эта формула должна отразить, что в M_1 только одно следующее состояние с двумя альтернативами, а в M_2 — два следующих состояния, каждое без альтернатив).

б) Попробуйте выразить в логике CTL свойства поведения, представленные LTL-формулами $\mathbf{AGF}\,p$ и $\mathbf{A}(\mathbf{X}\,p \vee \mathbf{XX}r)$.

2.10. Выразите формулу $\mathbf{A}(p\mathbf{U}q)$ логики CTL с помощью других формул этой логики.

2.11. Пусть p означает "*Я люблю Машу*", а q — "*Я люблю Дашу*". Каким высказываниям соответствуют следующие формулы CTL:

а) $\mathbf{AFEG}\,p$

б) $\mathbf{EFAG}\,p$

в) $\mathbf{A}(p\mathbf{U}q)$

г) $\mathbf{E}((\mathbf{EX}p)\mathbf{U}(\mathbf{AG}\,q))$

2.12. Докажите, что формула $\mathbf{G}(p \vee q) \Rightarrow (\mathbf{GF}p \vee \mathbf{GF}q)$ общезначима (т. е. истинна на всех ее интерпретациях).

2.13. Формализуйте утверждения в логике LTL:

а) "Я выйду замуж не менее двух раз"

б) "Я выйду замуж не более двух раз"

в) "Я выйду замуж точно два раза"

г) "Я выйду замуж не менее одного раза"

д) "Я выйду замуж не более одного раза"

е) "Я выйду замуж точно один раз"

ж) "Я никогда не выйду замуж"

в предположении, что говорящая сейчас не замужем и никогда замужем не была. Используйте атомарный предикат p, обозначающий событие "*Я выхожу замуж*".

2.14. Постройте структуру Крипке, на которой выполняется формула $\mathbf{EGEF}p$ и не выполняется формула $\mathbf{EGF}p$.

2.15. Выразите в LTL свойство: "Между каждой парой состояний, удовлетворяющих свойству p, обязательно встретится состояние, удовлетворяющее свойству q ".

2.16. Для темпоральных логик CTL и CTL* определены подклассы: логики ACTL, ACTL*, ECTL и ECTL*.

Формула CTL является формулой логики ACTL, если у нее используется только универсальный квантор пути **A** (т. е. все пары: *<квантор пути, темпоральный оператор>* являются парами **AX**, **AF**, **AG**, **AU**) и все отрицания применяются только к атомарным предикатам. Логика ACTL называется "*универсальной логикой CTL*".

Формула CTL является формулой логики ECTL, если у нее используется только квантор существования пути **E** и все отрицания применяются только к атомарным предикатам. Логика ACTL называется "*экзистенциальной логикой CTL*". Аналогично определяются универсальная и экзистенциальная логики CTL*.

Какой — универсальной или экзистенциальной — логике CTL (или CTL*) принадлежат формулы:

а) $\mathbf{AX}\,p \Rightarrow \mathbf{A}\left(q\mathbf{U}\neg r\right)$

б) $\mathbf{AF}p \Rightarrow \mathbf{A}\neg\left(q\mathbf{U}r\right)$

в) $\mathbf{EX}\,p \Rightarrow \left(q\mathbf{U}\neg r\right)$

г) $\mathbf{EG}p \vee \mathbf{EF}\left(q \wedge \mathbf{E}\left(\neg p\mathbf{U}r\right)\right)$

2.17. Кроме темпоральных операторов **F** и **G**, которые определяются через оператор **U**, иногда удобно применять оператор **R**, который называется "*оператором разблокировки*" (release) с двумя аргументами. Оператор $\varphi_1 R \varphi_2$ имеет следующую семантику: он утверждает, что свойство φ_2 выполняется на всем пути вплоть до того состояния, в котором истинно φ_1, хотя φ_1 может и не выполниться на пути никогда. Оператор **R** является двойственным оператору **U**: $\varphi_1 R \varphi_2 \equiv \neg(\neg\varphi_1 U \neg\varphi_2)$.

Определите формальную семантику оператора разблокировки **R**.

2.18. Докажите или опровергните:

а) $\left(\mathbf{F}p\right)\mathbf{U}\left(\mathbf{F}q\right) \equiv \mathbf{F}\left(p\mathbf{U}q\right)$

б) $\mathbf{G}\left(p \Rightarrow \mathbf{X}p\right) \equiv \mathbf{G}p \Rightarrow \mathbf{GX}p$

в) $\mathbf{G}\left(p \Rightarrow \mathbf{X}p\right) \equiv \mathbf{G}\neg p \vee \mathbf{FG}p$

г) $\mathbf{EGEX}\,p \equiv \mathbf{EXEG}\,p$

д) $p\,\mathbf{U}q \equiv (p \vee q)\,\mathbf{U}q$

2.19. Постройте несколько вычислений, на которых формула:

$\mathbf{G}(\textit{Персил} \Rightarrow \mathbf{G}\textit{Персил})$

будет истинной.

2.20. Постройте структуры Крипке, на которых выполняются следующие формулы:

а) $\mathbf{AXEX}\,p$

б) $\mathbf{EXAX}\,p \wedge \mathbf{AXEX}\,q$

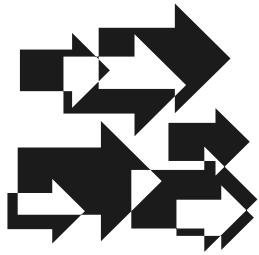
в) $\mathbf{EF}(p \wedge \mathbf{AF}\,q)$

г) $\mathbf{EXEG}\,p$

д) $\mathbf{AFG}\,p$

е) $\mathbf{AGF}\,p \wedge \mathbf{EFG}\,q$

ГЛАВА 3



Алгоритм model checking для CTL

В этой главе рассматривается алгоритм проверки моделей для одного из подклассов темпоральных логик — логики CTL (Computational Tree Logic). Формулы этой логики позволяют выразить достаточно сложные свойства поведения реагирующих систем. Алгоритм проверки выполнимости формулы CTL на модели реагирующих систем — структуре Крипке — удивительно прост и эффективен.

3.1. Темпоральная логика ветвящегося времени CTL

Рис. 3.1 уточняет рис. 1.1, представляя схему верификации на основе метода *model checking*. В качестве формальной модели системы для верификации используется структура Крипке, а формальная спецификация требований задается формулой темпоральной логики

В общей темпоральной логике ветвящегося времени CTL* возможны любые комбинации кванторов пути (**A** и **E**) и темпоральных операторов (**X** и **U**). Темпоральные операторы **F** и **G** (как и квантор всеобщности) являются выводимыми в этой логике. *Логика ветвящегося времени CTL* (Computational Tree Logic) является ограничением CTL*: в формулах CTL каждый темпоральный оператор должен быть предварен квантором пути. Иными словами, в формулах CTL эти модальности могут появиться только парами: **AF** , **EG** , **AU** и т. д.

Формулы логики CTL интерпретируются на множестве структур Крипке, т. е. модели вычислений с конечным числом состояний и возможными альтернативами. Тем самым моделируется возможность "разветвления" времени:

в каждом своем состоянии модель Крипке предоставляет несколько вариантов дальнейшего развития вычислений, и формула CTL позволяет специфицировать свойства поведения на различных ветвях дерева вычислений. Именно поэтому логика CTL названа логикой *ветвящегося времени*.

Грамматика формул CTL включает все 8 возможных пар:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{EX}\varphi \mid \mathbf{EF}\varphi \mid \mathbf{EG}\varphi \mid \mathbf{E}(\varphi\mathbf{U}\varphi) \mid \mathbf{AX}\varphi \mid \mathbf{AF}\varphi \mid \mathbf{AG}\varphi \mid \mathbf{A}(\varphi\mathbf{U}\varphi)$$

При написании формул, как обычно, можно использовать скобки для выделения порядка выполнения операторов.

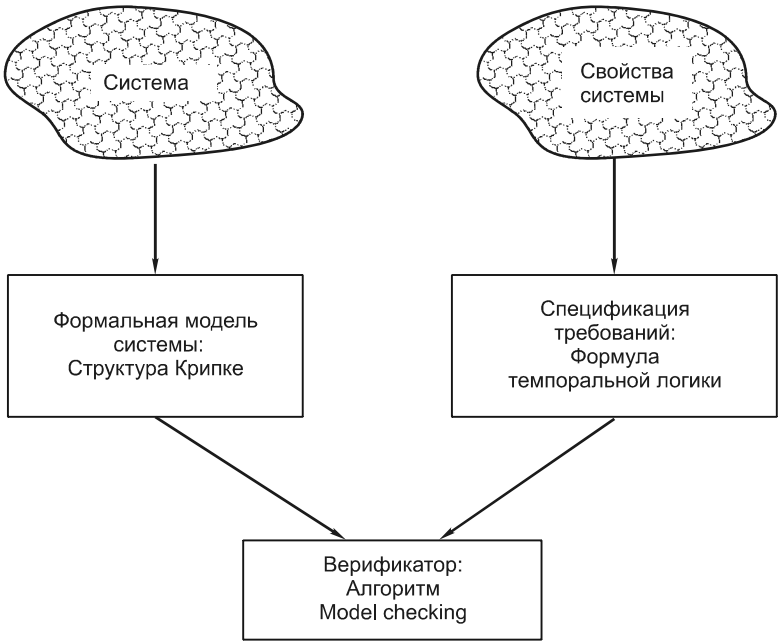


Рис. 3.1. Схема процесса верификации на основе алгоритма проверки модели (model checking)

3.2. Семантика CTL на деревьях вычислений

Все формулы CTL — это формулы состояний, т. е. любая формула CTL в каждом состоянии структуры Крипке имеет конкретное истинностное значение: она либо истинна, либо ложна. Любая формула CTL на данной структуре Крипке M либо выполняется (если она истинна в начальном состоянии M), либо не выполняется (если она ложна в начальном состоянии M).

Повторим неформальное определение этих формул. На данной структуре Крипке:

- **AX**φ выполняется в состоянии s, если формула φ выполняется во всех состояниях, непосредственно следующих за s;
- **EX**φ выполняется в состоянии s, если формула φ выполняется хотя бы в одном состоянии, непосредственно следующем за s;
- **AF**φ (*неизбежно* φ) выполняется в состоянии s, если на всех путях, начинающихся из состояния s, формула φ когда-нибудь выполнится;
- **EF**φ (*возможно* φ) выполняется в состоянии s, если из этого состояния существует путь, на котором формула φ когда-нибудь выполнится;
- **AG**φ (*глобально* φ) выполняется в состоянии s, если на всех путях, начинающихся из состояния s, во всех состояниях этих путей формула φ выполняется;
- **EG**φ выполняется в состоянии s, если существует путь из s, во всех состояниях которого формула φ выполняется;
- $\mathbf{A}(\varphi_1 \mathbf{U} \varphi_2)$ выполняется в состоянии s, если на всех путях, начинающихся в s, когда-нибудь в будущем выполнится формула φ_2, а до этого во всех состояниях этих путей выполняется формула φ_1;
- $\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2)$ выполняется в состоянии s, если из s существует путь, на котором когда-нибудь в будущем выполнится φ_2, а до этого во всех состояниях этого пути выполняется φ_1.

Визуально эту семантику удобно представить деревом вычислений с корнем в текущем состоянии. На рис. 3.2 изображены такие деревья, в корнях которых истинны указанные формулы.

Визуальное представление семантики позволяет понять рекурсивное выражение для формул CTL:

- $\mathbf{AF}\varphi = \varphi \vee \mathbf{AX}\,\mathbf{AF}\varphi$

  на всех путях из текущего состояния формула φ когда-нибудь выполнится — это то же самое, что или φ выполняется в текущем состоянии, или же формула **AF**φ выполняется во всех следующих состояниях;

- $\mathbf{EF}\varphi = \varphi \vee \mathbf{EX}\,\mathbf{EF}\varphi$

  на каком-нибудь пути из текущего состояния когда-нибудь выполнится формула φ — это то же самое, что или φ выполняется в текущем состоя-

нии, или существует такой путь, что формула $\mathbf{EF}\varphi$ выполнится в следующем его состоянии;

- $\mathbf{AG}\varphi = \varphi \wedge \mathbf{AX}\,\mathbf{AG}\,\varphi$

  во всех состояниях на всех путях из текущего состояния выполняется формула φ — это то же самое, что φ выполняется в текущем состоянии, и во всех следующих состояниях всех путей, начинающихся в текущем состоянии, выполняется формула $\mathbf{AG}\varphi$;

- $\mathbf{EG}\varphi = \varphi \wedge \mathbf{EX}\,\mathbf{EG}\,\varphi$

  существует путь из текущего состояния, во всех состояниях которого φ выполняется — это то же самое, что φ выполняется в текущем состоянии

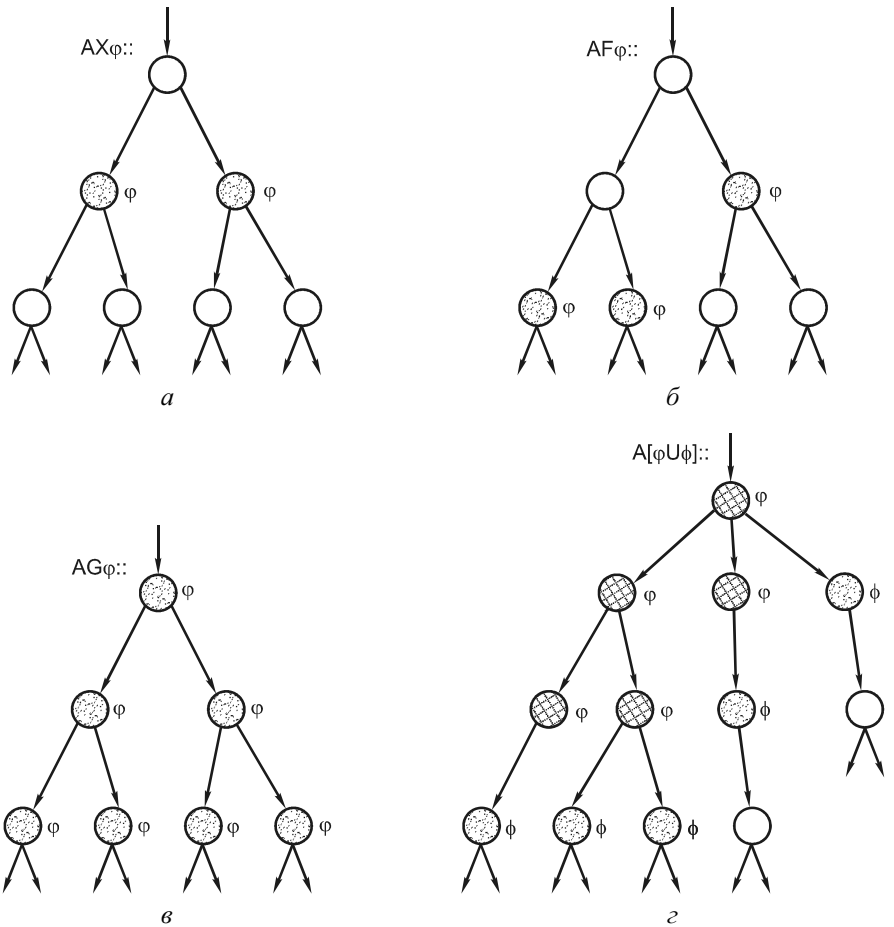
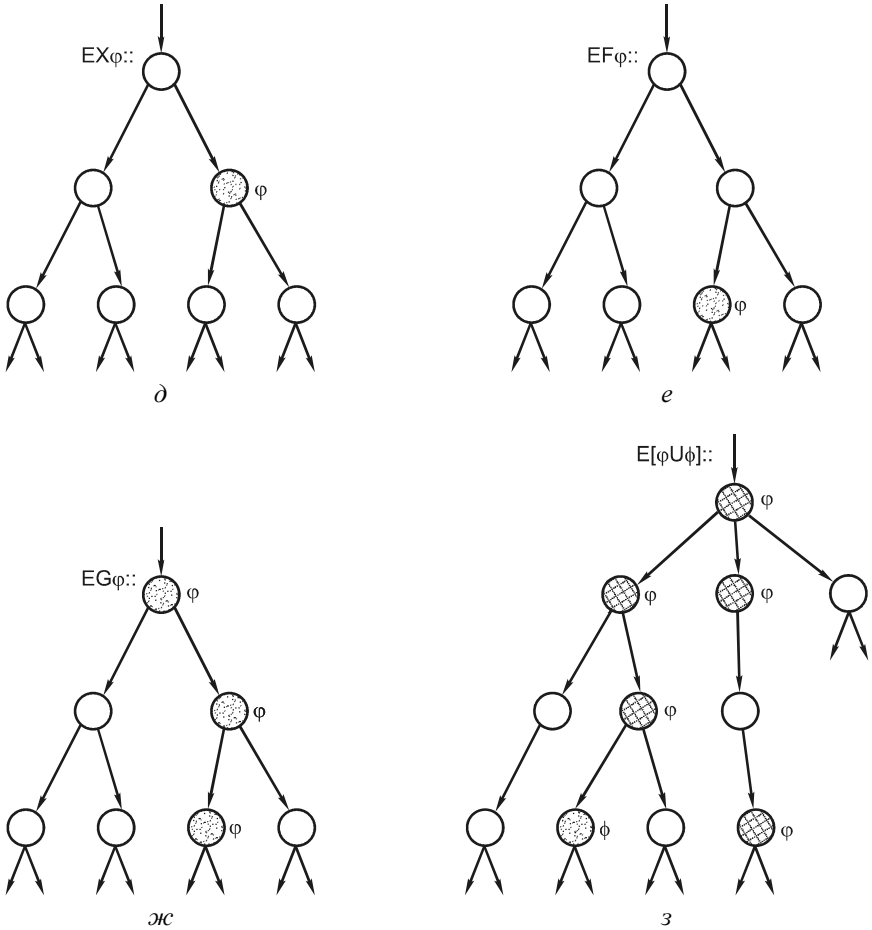


Рис. 3.2, *а—г*. Визуальное представление семантики формул CTL



д *е* *ж* *з*

Рис. 3.2, *д*—*з*. Визуальное представление семантики формул CTL

и из этого состояния существует такой путь, в следующем состоянии которого формула $\mathbf{EG}\,\varphi$ выполняется;

- $\mathbf{A}(\varphi_1 \mathbf{U} \varphi_2) = \varphi_2 \vee (\varphi_1 \wedge \mathbf{AX}\, \mathbf{A}(\varphi_1 \mathbf{U} \varphi_2))$

на всех путях из текущего состояния когда-нибудь выполнится φ_2, а до этого во всех состояниях выполняется φ_1 — это то же самое, что или φ_2 выполняется в текущем состоянии, или же в текущем состоянии выполняется φ_1, а во всех следующих состояниях всех путей, идущих из текущего состояния, выполняется формула $\mathbf{A}(\varphi_1 \mathbf{U} \varphi_2)$;

- $\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2) = \varphi_2 \vee (\varphi_1 \wedge \mathbf{EX}\, \mathbf{E}(\varphi_1 \mathbf{U} \varphi_2))$

  из текущего состояния существует путь, на котором когда-нибудь выполнится φ_2, а до этого во всех состояниях этого пути выполняется φ_1 — это то же самое, что или φ_2 выполняется в текущем состоянии, или же в текущем состоянии выполняется φ_1 и из него существует путь, в следующем состоянии которого формула $\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2)$ выполняется.

3.3. Формальная семантика CTL

Пусть M — структура Крипке $(S, S_0, \rightarrow, AP, L)$ (стрелкой "$\rightarrow$" здесь обозначено отношение перехода в M: если из состояния s_i в состояние s_j идет дуга, то $(s_i, s_j) \in \rightarrow$, или, что то же, $(s_i \rightarrow s_j)$). Утверждение "формула φ истинна в состоянии s структуры Крипке M" записывается так: $M, s \models \varphi$. Если структура Крипке M, о которой идет речь, очевидна, M может опускаться.

Истинность формулы φ логики CTL в состоянии s структуры M определяется по индукции по структуре формулы φ:

- $s \models p \equiv p \in L(s)$
- $s \models \neg\varphi \equiv s \not\models \varphi$
- $s \models \varphi_1 \vee \varphi_2 \equiv s \models \varphi_1 \vee s \models \varphi_2$
- $s \models \mathbf{AX}\varphi \equiv (\forall s_1 : s \rightarrow s_1)\ s_1 \models \varphi$
- $s \models \mathbf{EX}\varphi \equiv (\exists s_1 : s \rightarrow s_1)\ s_1 \models \varphi$
- $s \models \mathbf{AG}\varphi \equiv (\forall (s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots) : (s = s_0))(\forall i)\ s_i \models \varphi$
- $s \models \mathbf{EG}\varphi \equiv (\exists (s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots) : (s = s_0))\ (\forall i) s_i \models \varphi$
- $s \models \mathbf{AF}\varphi \equiv (\forall (s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots) : (s = s_0))\ (\exists i) s_i \models \varphi$
- $s \models \mathbf{EF}\varphi \equiv (\exists (s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots) : (s = s_0))\ (\exists i) s_i \models \varphi$
- $s \models \mathbf{A}(\varphi_1 \mathbf{U} \varphi_2) \equiv (\forall (s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots) : (s = s_0))$
  $(\exists j \geq 0)\ (s_j \models \varphi_2 \wedge (\forall k : 0 \leq k < j) s_k \models \varphi_1)$

$$s \models \mathbf{E}(\varphi_1 \mathbf{U} \varphi_2) \equiv \left(\exists (s_0 \to s_1 \to s_2 \to \ldots) : (s = s_0) \right)$$
$$(\exists j \geq 0) \left(s_j \models \varphi_2 \wedge (\forall k : 0 \leq k < j) s_k \models \varphi_1 \right)$$

Хотя эти определения выглядят довольно сложно, они совершенно понятны. Начнем с первого. Если формула есть просто атомарный предикат p, то в состоянии s структуры Крипке M эта формула истинна (выполняется) тогда и только тогда, когда p входит в число атомарных предикатов, помечающих состояние s. Второе правило говорит о том, что в состоянии s структуры Крипке M формула φ не истинна тогда и только тогда, когда в s эта формула не выполняется.

Рассмотрим семантическое правило для формулы $\mathbf{AX}\varphi$. Смысл формулы прост: она истинна в состоянии s, если во всех состояниях, непосредственно следующих за s, формула φ выполняется. Правило формально устанавливает именно это: оно говорит о том, что в состоянии s структуры Крипке M формула $\mathbf{AX}\varphi$ истинна тогда и только тогда, когда для любого состояния s_1, в которое есть переход из s, эта формула выполняется.

Формула $\mathbf{EF}\varphi$ более сложна: из текущего состояния существует путь, на котором когда-нибудь в будущем φ выполнится. Рассмотрим ее формальное определение:

$$s \models \mathbf{EF}\varphi \equiv \left(\exists (s_0 \to s_1 \to s_2 \to \ldots) : (s = s_0) \right) (\exists i) s_i \models \varphi;$$

Эта формула истинна в состоянии s тогда и только тогда, когда существует такое вычисление из s (последовательность состояний $s_0, s_1, \ldots$, в каждое из которых есть переход из предыдущего состояния), начальное состояние s_0 которого совпадает с s, что в каком-нибудь состоянии этого вычисления (включая и начальное) формула φ выполняется. Здесь $(\exists x : R(x)) Q(x)$ — это ограниченный предикат, который нужно понимать как $(\exists x)(R(x) \wedge Q(x))$. Аналогичный ограниченный предикат с квантором всеобщности $(\forall x : R(x)) Q(x)$ нужно понимать как $(\forall x)(R(x) \Rightarrow Q(x))$. Семантика остальных формул трактуется так же просто.

3.4. Проверка формул CTL на развертке структуры Крипке

Рассмотрим, как можно вычислить истинность или ложность CTL-формул на структуре Крипке. На рис. 3.3 показана структура Крипке M и ее развертка — начало бесконечного дерева возможных вычислений M, начинающихся в начальном состоянии M.

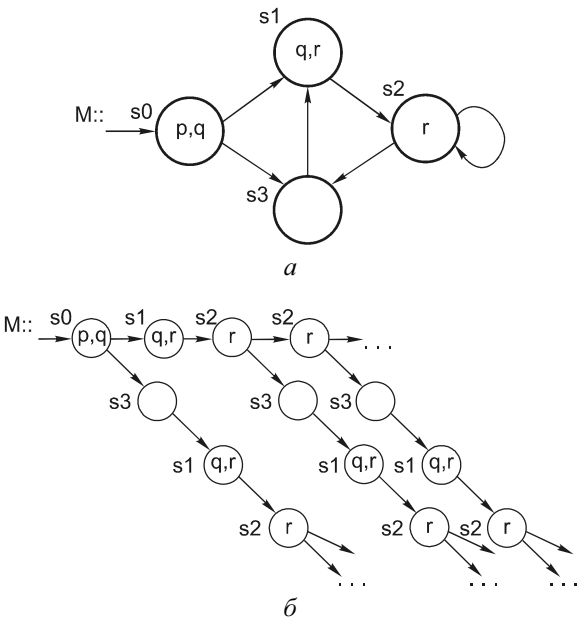


Рис. 3.3. Структура Крипке и ее развертка

Формула темпоральной логики выполняется на M, если она выполняется в начальном состоянии M. Для нашего примера на рис. 3.3:

- Формула $p \wedge q$ выполняется на M (т. е. $M \models p \wedge q$), поскольку эта логическая формула истинна в начальном состоянии M: в нем истинны оба атомарных предиката, p и q.
- Формула $\mathbf{EG}(r \vee p)$ выполняется на M (т. е. $M \models \mathbf{EG}(r \vee p)$), поскольку из начального состояния структуры Крипке M существует путь, в каждом состоянии которого или истинно r, или истинно p.
- Формула $\mathbf{EX}\,\mathbf{EG}\,r$ выполняется на M (т. е. $M \models \mathbf{EX}\,\mathbf{EG}\,r$), поскольку из начального состояния структуры Крипке M существует путь, такой, что

из следующего его состояния существует путь, на котором все время будет выполняться r.

- Формула $\mathbf{AG}\,\mathbf{EF}(\neg p \wedge \neg q)$ выполняется на M (т. е. $M \models \mathbf{AG}\,\mathbf{EF}(\neg p \wedge \neg q)$), поскольку для всех путей из начального состояния из всех состояний этих путей существует путь, такой, что в будущем на этом пути не встретится ни p, ни q.

Такой неформальный анализ можно провести лишь для небольших структур Крипке. Нашей задачей является рассмотрение алгоритма model checking, который может быть использован для произвольной структуры Крипке и формулы CTL любой сложности.

3.5. Базисы CTL

Две формулы φ и ψ темпоральной логики называются семантически эквивалентными (обозначается $\varphi \equiv \psi$), если они принимают одинаковые истинностные значения на каждой возможной интерпретации (структуре Крипке). Используя следующие очевидные соотношения:

$$\mathbf{A}\psi \equiv \neg\mathbf{E}\neg\psi$$

$$\mathbf{F}\varphi \equiv \mathit{True}\mathbf{U}\varphi$$

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

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

$$\mathbf{A}(\varphi_1\mathbf{U}\varphi_2) \equiv \mathbf{AF}\varphi_2 \wedge \neg\mathbf{E}(\neg\varphi_2\mathbf{U}(\neg\varphi_1 \wedge \neg\varphi_2))$$

можно выразить одни комбинации — "квантор пути, темпоральный оператор", через другие. Например, для CTL справедливы следующие аналоги законов Де Моргана:

$$\mathbf{AX}\varphi \equiv \neg\mathbf{EX}\neg\varphi$$

$$\mathbf{AG}\varphi \equiv \neg\mathbf{EF}\neg\varphi$$

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

Минимальное множество операторов, с помощью которых можно выразить любой оператор CTL, как обычно, называется базисом. Используя приведенные выше соотношения, можно построить несколько базисов CTL. Рассмотрим, как строить базисы CTL.

На рис. 3.4 представлены все возможные парные комбинации CTL: **EX**, **AX**, **EG**, **AG**, **EF**, **AF**, **EU** и **AU**. Стрелками показаны все перечисленные

выше возможности выражения одних комбинаций CTL через другие. А именно, комбинации **EX** и **AX**, **EG** и **AF**, **EF** и **AG** можно выразить друг через друга. Далее, поскольку $\mathbf{F}\varphi \equiv \mathit{True}\mathbf{U}\varphi$, можно выразить **EF** через **EU** и **AF** через **AU** (но не наоборот). Наконец, в соответствии с формулой:

$$\mathbf{A}(\varphi_1\mathbf{U}\varphi_2) \equiv \mathbf{AF}\varphi_2 \wedge \neg\mathbf{E}(\neg\varphi_2\mathbf{U}(\neg\varphi_1 \wedge \neg\varphi_2))$$

AU можно выразить парой комбинаций **AF** и **EU**.

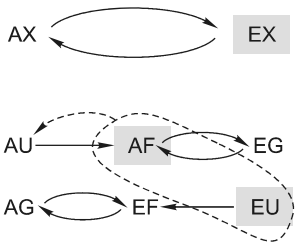


Рис. 3.4. Зависимости между комбинаторами CTL

Для того чтобы построить базис, нужно выбрать минимальное число таких комбинаций "квантор, темпоральный оператор", из которых можно получить все остальные. Например, выберем три пары: $\{\mathbf{EX}, \mathbf{AF}, \mathbf{EU}\}$ (они выделены на рис. 3.4). Это множество комбинаций — базис, поскольку по стрелкам зависимостей на рис. 3.4 видно, что оставшиеся пять комбинаций можно выразить через эти три:

$$\mathbf{AX}\varphi \equiv \neg\mathbf{EX}\neg\varphi$$

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

$$\mathbf{A}(\varphi_1\mathbf{U}\varphi_2) \equiv \mathbf{AF}\varphi_2 \wedge \neg\mathbf{E}(\neg\varphi_2\mathbf{U}(\neg\varphi_1 \wedge \neg\varphi_2))$$

$$\mathbf{EF}\varphi \equiv \mathbf{E}(\mathit{True}\mathbf{U}\varphi)$$

$$\mathbf{AG}\varphi \equiv \neg\mathbf{EF}\neg\varphi$$

Основываясь на зависимостях рис. 3.4, можно построить всего шесть базисов CTL из трех комбинаций "квантор, темпоральный оператор": из этого рисунка видно, что вместо пары **AF** можно выбрать пару **AU** или **EG**, и каждая из этих трех возможностей может быть либо с парой **EX**, либо с парой **AX**. Все базисы CTL, которые можно построить, исходя из перечисленных выше зависимостей, таковы:

$$\{\mathbf{EX}, \mathbf{AF}, \mathbf{EU}\}, \{\mathbf{EX}, \mathbf{AU}, \mathbf{EU}\}, \{\mathbf{EX}, \mathbf{EG}, \mathbf{EU}\}$$

$$\{\mathbf{AX}, \mathbf{AF}, \mathbf{EU}\}, \{\mathbf{AX}, \mathbf{AU}, \mathbf{EU}\}, \{\mathbf{AX}, \mathbf{EG}, \mathbf{EU}\}$$

Поставим вопрос: существуют ли в логике CTL другие базисы, кроме перечисленных шести?

Формула, в которой комбинация **AU** выражается через комбинации **AF** и **EU**, наводит на мысль, что существует и симметричное соотношение, выражающее **EU** через **EF** и **AU**. Наличие такого соотношения могло бы позволить построить и другие базисы CTL. Как пишет Francois Laroussinie в [51], многие думают, что это возможно. То, что это не так, F.Laroussinie называет "совершенно удивительным". Приведем, следуя [51], доказательство этого результата, которое является весьма простым. Это доказательство показывает "кухню" исследований в этой области.

ТЕОРЕМА 3.1

EU не может быть выражено с помощью **EF** и **AU**.

Доказательство

Предположим противное, т. е. что это возможно. Тогда из доказанных выше соотношений (см. рис. 3.4) следует, что множество комбинаций $\{\mathbf{EX}, \mathbf{EF}, \mathbf{AU}\}$ тоже является базисом CTL. Покажем, что это не так, т. е. что язык формул CTL строго более выразительный, чем язык L_A, в котором позволены (допустимы) только три комбинации: **EX**, **EF** и **AU**. Как следствие, мы получим доказательство утверждения теоремы.

Язык формул L_A, построенный из комбинаций $\{\mathbf{EX}, \mathbf{EF}, \mathbf{AU}\}$, определяется следующей грамматикой:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{EX}\varphi \mid \mathbf{EF}\varphi \mid \mathbf{A}(\varphi \mathbf{U} \varphi).$$

Для доказательства теоремы 3.1 покажем, что для произвольной формулы ψ, построенной из конструкций языка L_A, можно построить такую структуру Крипке, на которой для двух выбранных состояний этой структуры формула ψ будет одновременно истинной или одновременно ложной, а некоторая формула логики CTL (фактически, формула **EU**) для одного из этих состояний будет истинной, а для другого состояния — ложной. Тем самым мы покажем, что язык L_A слабее языка логики CTL: в языке L_A нельзя выразить некоторые свойства, которые выразимы в логике CTL.

Рассмотрим структуру Крипке на рис. 3.5. Покажем, что для любого $j \geq 0$, любая формула ψ языка L_A глубиной $|f| = k$ не большей, чем j (т. е. $0 \leq k \leq j$), будет одновременно истинна или ложна в состояниях s_j и r_j этой

структуры Крипке, т. е. $s_j \models \psi \equiv r_j \models \psi$. Доказательство утверждения проведем по индукции.

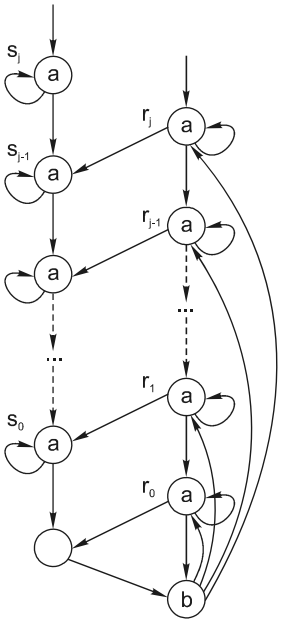


Рис. 3.5. Структура Крипке к доказательству теоремы 3.1

При $k = 0$ это соотношение очевидно. Формула ψ глубиной 0 — это атомарные предикаты. Их истинностные значения совпадают на всех состояниях s_j и r_j при $0 \le k \le j$.

Пусть теперь $k = k'+1$, и по индуктивной гипотезе предположим, что любые формулы языка L_A, глубиной k' или меньшей, одновременно истинны или ложны в состояниях s_j и r_j структуры Крипке на рис. 3.5 при $0 \le k' \le j$. Докажем, что формулы языка L_A глубиной k или меньшей одновременно истинны или ложны в состояниях s_j и r_j при $0 \le k \le j$.

Рассмотрим возможные типы формул языка L_A.

Формулы $\psi = \neg\varphi$ **или** $\psi = \varphi_1 \vee \varphi_2$. Поскольку φ, φ_1 и φ_2 — формулы глубиной k' или меньшей, по индуктивному предположению истинностные значения этих формул в состояниях s_j и r_j совпадают. Поэтому и формулы $\neg\varphi$ и $\varphi_1 \vee \varphi_2$ будут одновременно истинны или ложны в этих состояниях.

Формула $\psi = \mathbf{EX}\varphi$. По индуктивному предположению, истинностные значения любой формулы φ глубиной k' или меньшей, в состояниях s_j и r_j, так же, как и в состояниях s_{j-1} и r_{j-1}, совпадают: в этих состояниях они или одновременно истинны, или одновременно ложны. При истинности $\psi = \mathbf{EX}\varphi$ в состоянии r_j, формула φ истинна или в r_{j-1}, или в s_{j-1}, или в r_j (вследствие цикла $r_j \to r_j$). Если она истинна в r_{j-1}, то, по индуктивной гипотезе, она истинна и в s_{j-1}; если она истинна в r_j, то, по индуктивной гипотезе, она истинна и в s_j. В соответствии с рис. 3.5, ψ истинна и в s_j, т. е. $r_j \models \mathbf{EX}\varphi \Rightarrow s_j \models \mathbf{EX}\varphi$. Доказательство того, что $s_j \models \mathbf{EX}\varphi \Rightarrow r_j \models \mathbf{EX}\varphi$ аналогично.

Формула $\psi = \mathbf{EF}\varphi$. Все состояния, достижимые из s_j, достижимы также и из r_j, и наоборот. Поэтому $s_j \models \mathbf{EF}\varphi \equiv r_j \models \mathbf{EF}\varphi$.

Формула $\psi = \mathbf{A}(\varphi_1 \mathbf{U} \varphi_2)$. Если $s_j \models \mathbf{A}(\varphi_1 \mathbf{U} \varphi_2)$, то формула φ_2 в состоянии s_j выполняется (из s_j есть вычисление, состоящее только из состояний s_j). По индуктивной гипотезе, тогда φ_2 истинно и в r_j.

Теперь возьмем любую формулу f языка L_A. Пусть $|f|$ равно k. На структуре Крипке (рис. 3.5) в соответствии с доказанным выше, если $j \geq k$, то $s_j \models f$ тогда и только тогда, когда $r_j \models f$. Однако очевидно, что CTL-формула $\mathbf{E}(a \mathbf{U} b)$ выполняется в r_j и не выполняется в s_j. Следовательно, выразительная мощность языка L_A строго меньше выразительной мощности CTL: никакие формулы языка L_A не различают состояний s_j и r_j в построенной нами структуре Крипке, а формула CTL их различает.

На основании этой теоремы можно утверждать, что рис. 3.4 отражает все возможные зависимости между комбинациями "квантор пути, темпоральная формула" логики CTL. Поэтому все базисы CTL можно построить из зависимостей, показанных на рис. 3.4. Иными словами, CTL имеет только шесть базисов, перечисленных выше.

3.6. Алгоритм model checking для CTL

Основная идея этого алгоритма — для каждой подформулы заданной формулы CTL определить, в каких состояниях заданной структуры Крипке M эта подформула выполняется. Этот алгоритм называется *алгоритмом маркиров-*

ки: для каждой подформулы заданной формулы CTL, начиная с простейших подформул, этот алгоритм помечает все те состояния структуры M, в которых эта подформула выполняется.

Пусть задана произвольная формула Φ логики CTL и структура Крипке M. Для каждой подформулы ψ формулы Φ алгоритм маркировки выполняет следующие шаги:

1. Вычисляется множество Sat_ψ состояний структуры Крипке M, в которых выполняется ψ.
2. Вводится новый атомарный предикат p_ψ.
3. Этим атомарным предикатом помечаются все состояния M из множества Sat_ψ.

Поскольку обработка каждой формулы ψ заканчивается введением нового атомарного предиката p_ψ и маркировкой этим предикатом всех состояний, на которых выполняется ψ, то можно считать, что на каждом шаге алгоритма маркировки элементами подформул являются только атомарные предикаты. По завершении алгоритма, если начальное состояние структуры Крипке M помечено атомарным предикатом p_Φ, то формула Φ выполняется на M. Очевидно, что достаточно рассмотреть алгоритмы разметки только для формул какого-нибудь (любого) базиса CTL: все другие формулы могут быть выражены через формулы базиса. Выберем, например, базис $\{\mathbf{EX}, \mathbf{AF}, \mathbf{EU}\}$.

Следующий алгоритм находит множество Sat_ψ таких состояний структуры Крипке $M = (S, S_0, R, AP, L)$, в которых истинна каждая подформула ψ формулы Φ, и помечает эти состояния атомарным предикатом p_ψ.

```
for all 0< i ≤ |Φ| do
  for all Ψ∈Sub(Φ) with |Ψ| = i do   /* Для всех подформул Ψ глубиной i */
 /* все такие подформулы Ψ выделяются алгоритмом синтаксического анализа
формулы Φ */
    switch(Ψ):
       p        :     Satψ := { s∈S| p∈L(s) };
       ¬p       :     Satψ := { s∈S| p∉L(s) };
       p∨q      :     Satψ := { s∈S| p∈L(s) или q∈L(s) };
       EXp      :     Satψ := Sat_EX(p);
       AFp      :     Satψ := Sat_AF(p);
       E(pUq)   :     Satψ := Sat_EU(p,q);
    end switch
```

```
/* Sat_EX, Sat_AF и Sat_EU- это функции, определенные ниже */
/* вводим новый атомарный предикат pψ и помечаем им все состояния
множества Satψ */
  for all s∈ Satψ do L(s):=L(s)∪{pψ} od
od
```

Поясним работу алгоритма. Алгоритм неявно выполняет синтаксический анализ формулы CTL, который на каждом шаге выделяет очередную подформулу исходной формулы Φ, начиная с самых внутренних (здесь $|\Phi|$ — число подформул формулы Φ). После выделения очередной подформулы управление передается переключателю `switch`, который обрабатывает различные типы подформул Φ в зависимости от вида этой подформулы следующим образом.

Если подформула ψ — это атомарный предикат, или отрицание атомарного предиката, или дизъюнкция двух атомарных предикатов, то построение множества Sat_ψ состояний, на которых эти подформулы выполняются, совершенно очевидно.

Если подформулой ψ является **EX**, **AF** или **EU**, то алгоритм обращается к соответствующим функциям, которые возвращают множество состояний, на которых эти подформулы выполняются. Рассмотрим эти функции по порядку.

Функция $Sat_EX(p)$. Множество состояний, на которых выполняется $\mathbf{EX}p$, строится из тех состояний, хотя бы один из преемников которых уже помечен p. Графически процедура нахождения этого множества показана на рис. 3.6: если хотя бы один из преемников состояния s уже помечен p, то в этом состоянии выполняется формула $\mathbf{EX}p$.

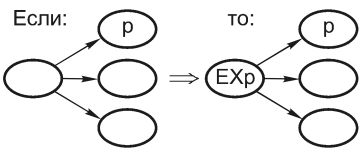


Рис. 3.6. Графическое представление правила разметки множества состояний структуры Крипке для формулы $\mathbf{EX}p$

Функция, возвращающая множество соответствующих состояний, приведена ниже.

```
function Sat_EX(p)
/* возвращает множество состояний, на которых выполняется EXp */
      local var Y;
```

```
begin
        Y:= { s| (∃s1: s→s1) p∈L(s1)};
        /* в Y включаются состояния,из которых есть переход
        в состояние, помеченное p */
        return Y
end function
```

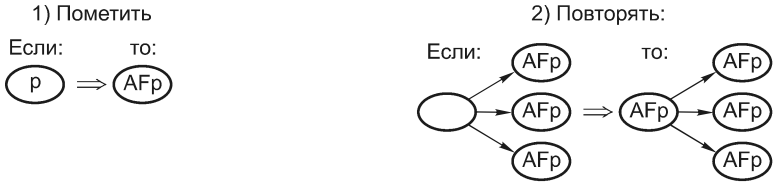


Рис. 3.7. Графическое представление алгоритма нахождения множества состояний, на которых выполняется **AF** p

Функция $Sat_AF(p)$. Множество состояний Y, на которых выполняется **AF** p, строится в два этапа. Сначала в множество Y собираются те состояния, которые уже помечены p. Затем в Y добавляются все состояния, у которых все преемники уже принадлежат Y. Этот шаг повторяется многократно до тех пор, пока в Y можно добавить хотя бы одно новое состояние. Графически эта процедура показана на рис. 3.7.

```
function Sat_AF(p)
/* возвращает множество состояний, на которых выполняется AFp) */
        local var X, Y;
begin
        X:= S;
        Y:= { s∈S| p∈L(s) }  /* в Y включаются те состояния, которые
                                                помечены p */
        repeat until X=Y;     /* пока в Y появляются новые состояния */
        begin
                X:=Y;
                Y:= Y ∪ { s | (∀s1:s→s1) s1∈Y }
                /* на каждом шаге в Y добавляются состояния, все преемники
                которых уже находятся в Y */
        end;
        return Y
end function
```

Функция $Sat_EF(p)$. Множество состояний Y, на которых выполняется **EF** p, также работает в два этапа. Сначала в множество Y собираются те со-

стояния, которые уже помечены p. Затем в Y добавляются все состояния, у которых есть хотя бы один преемник из множества Y. Этот шаг повторяется до тех пор, пока в Y появляются новые состояния.

```
function Sat_EF(p)
/* возвращает множество состояний, на которых выполняется EFp */
       local var X, Y;
begin
       X:= S;
       Y:= { s∈S| p∈L(s) } /* в Y включаются состояния, помеченные p */
       repeat until X=Y; . /* пока в Y появляются новые состояния */
       begin
              X:=Y;
              Y:= Y ∪ { s | (∃s1:s→s1) s1∈Y }
              /* на каждом шаге в Y добавляются такие состояния s,
              из которых существует переход хотя бы в одно состояние,
              уже находящееся в Y*/
       end;
       return Y
end function
```

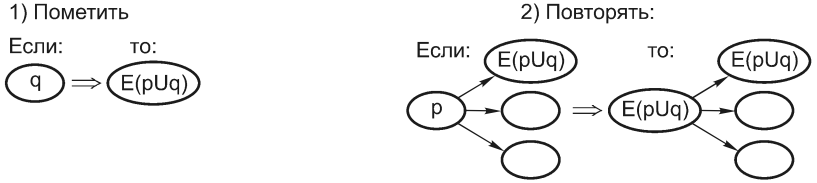


Рис. 3.8. Графическое представление алгоритма нахождения множества состояний, на которых выполняется $\mathbf{E}(p\mathbf{U}q)$

Функция $Sat_EU(p,q)$. Множество состояний Y, на которых выполняется формула $\mathbf{E}(p\mathbf{U}q)$, также выполняется в два этапа. Сначала в множество Y собираются те состояния, которые помечены q: в этих состояниях формула $\mathbf{E}(p\mathbf{U}q)$ выполняется, поскольку в них истинно q. Затем в Y добавляются те состояния, помеченные p, у которых хотя бы один преемник уже принадлежит Y. Этот шаг повторяется до тех пор, пока в Y появляются новые состояния. Графически эта процедура показана на рис. 3.8.

```
function Sat_EU(p,q)
/* функция возвращает множество состояний, на которых выполняется E(pUq) */
       local var W, X, Y;
begin
       W:= { s∈S| p∈L(s) } /* в W включаются состояния, помеченные p */
       X:= S;
       Y:= { s∈S| q∈L(s) } /* в Y включаются состояния, помеченные q */
repeat until X=Y;
       begin
              X:=Y;
              Y:= Y ∪ (W ∩{ s | (∃s1: s→s1) s1∈Y } )
              /* в Y собираются такие состояния, которые помечены p
              и из которых есть переход в состояние, уже находящееся в Y
              */
       end;
       return Y
end function
```

Сложность алгоритма model checking мала: $O\left(|\Phi|*|M|^2\right)$: алгоритм линеен относительно сложности (числа подформул) формулы Φ и квадратичен относительно сложности структуры Крипке M (числа ее состояний и числа ее переходов). Действительно, алгоритм нужно выполнять для каждой подформулы Φ, а для формулы $\mathbf{AF}\,p$ нужно для каждого из состояний M проверять, помечены ли все преемники этого состояния формулой $\mathbf{AF}\,p$.

Сложность этого алгоритма можно уменьшить, выбрав другой базис CTL. Поскольку формулу **AF** можно выразить через **EG**, то вместо базиса $\{\mathbf{EX}, \mathbf{AF}, \mathbf{EU}\}$ можно выбрать базис $\{\mathbf{EX}, \mathbf{EG}, \mathbf{EU}\}$. Напомним, что в состоянии s структуры Крипке M выполняется формула $\mathbf{EG}\,p$, если в M существует бесконечный путь, начинающийся в s, все состояния которого помечены p. Рассмотрим алгоритм построения множества состояний структуры Крипке, на которых выполняется формула $\mathbf{EG}\,p$.

Функция $Sat_EG(p)$. Множество состояний Y, на которых выполняется формула $\mathbf{EG}\,p$, работает также в два этапа. Сначала в множество Y собираются те состояния, которые помечены p. На втором этапе из этого множества Y выбрасываются все состояния, у которых нет ни одного преемника в Y (т. е. помеченных p). Этот второй шаг повторяется до тех пор, пока в Y уменьшается число состояний.

```
function Sat_EG(p)
/* возвращает множество состояний, на которых выполняется EGp */
        local var X, Y;
begin
        X:= ∅;
        Y:= { s∈S| p∈L(s) } /* в Y — состояния, помеченные p  */
        repeat until X=Y;
        begin
                X:=Y;
                Y:= Y \ { s | (∀s1:s→s1) s1∉Y }
                /* из Y выбрасываются такие состояния, из которых нет
                перехода в состояния, которые уже принадлежат Y */
        end;
        return Y
end function
```

Можно построить и другой алгоритм пометок для формулы $\mathbf{EG}\,p$, сложность которого линейна относительно сложности структуры Крипке. Этот алгоритм выполняется в три этапа (рис. 3.9).

1. Строится структура Крипке M_p, являющаяся ограничением структуры Крипке M на множестве тех состояний, которые уже помечены атомарным предикатом p.

2. На множестве M_p строятся все ССК (сильно связные компоненты).

3. Искомое множество состояний, на которых выполняется формула $\mathbf{EG}\,p$, образуют все состояния из этих ССК и те состояния из M_p, из которых существует путь хотя бы в один сильно связный компонент.

Каждый из этих шагов имеет линейную сложность. Нахождение сильно связных компонентов производится с помощью алгоритма Тарьяна.

Сложность алгоритма проверки CTL выполнения формулы Φ на структуре Крипке $M=(S, S_0, R, AP, L)$ для базиса $\{\mathbf{EX}, \mathbf{EG}, \mathbf{EU}\}$ линейна как относительно сложности формулы Φ (числа ее подформул), так и сложности структуры Крипке (числа состояний и числа переходов). Она равна $O(|\Phi|*(|S|+|R|))$.

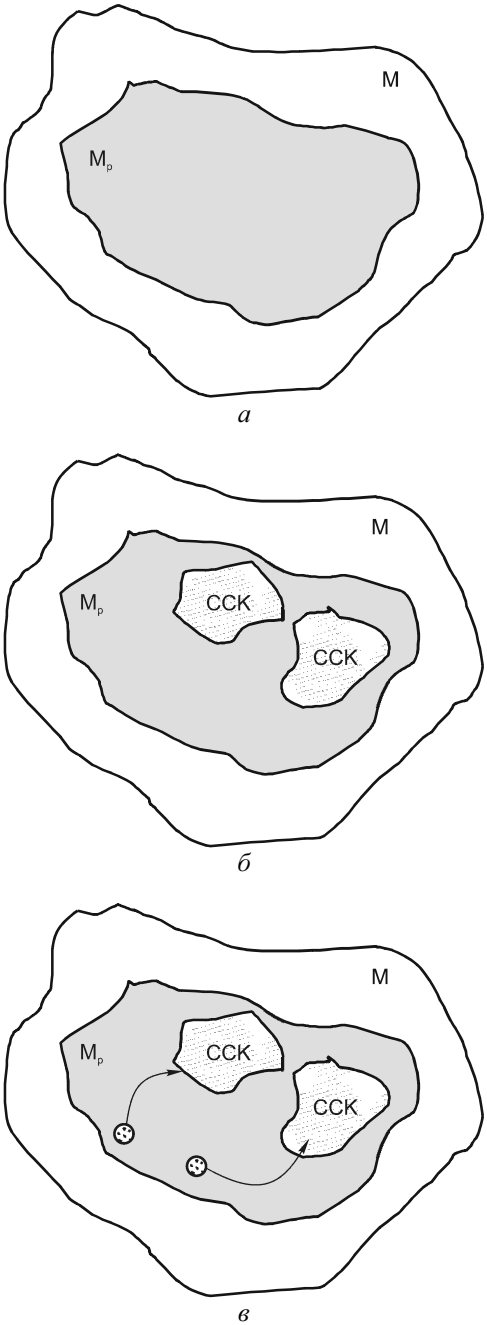


Рис. 3.9. Три этапа выполнения алгоритма вычисления состояний, удовлетворяющих **EG** *p*

3.7. Пример выполнения алгоритма model checking

Проверим, выполняется ли формула $\Phi = \mathbf{E}\left(\left(\mathbf{EX}\neg p\right)\mathbf{U}\,\mathbf{AF}\left(q \vee r\right)\right)$ на структуре Крипке M, представленной на рис. 3.10.

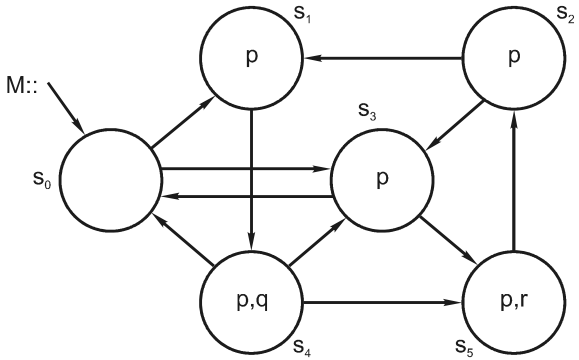


Рис. 3.10. Пример структуры Крипке для анализа

Для удобства выделим все подформулы формулы Φ с помощью обычного алгоритма синтаксического анализа [160]. Дерево синтаксического анализа этой формулы представлено на рис. 3.11.

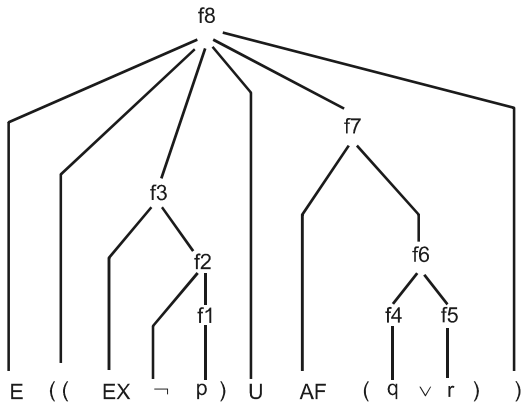


Рис. 3.11. Синтаксический анализ формулы логики CTL

Будем последовательно выполнять алгоритм маркировки для структуры Крипке M (рис. 3.10) всеми подформулами формулы Φ, начиная с простейших. Структура Крипке M с помеченными состояниями представлена на рис. 3.12.

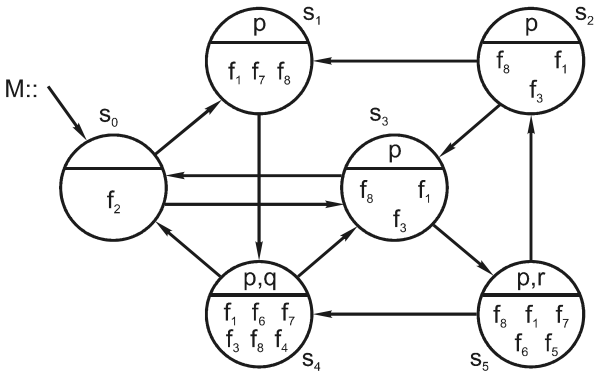


Рис. 3.12. Результат работы алгоритма model checking на структуре Крипке M

Последовательно строим:

- $f_1 = p$ $\quad Sat_{f1} = \{s_1, s_2, s_3, s_4, s_5\}$
- $f_2 = \neg f_1$ $\quad Sat_{f2} = \{s_0\}$
- $f_3 = \mathbf{EX} f_2$ $\quad Sat_{f3} = \{s_3, s_4\}$
- $f_4 = q$ $\quad Sat_{f4} = \{s_4\}$
- $f_5 = r$ $\quad Sat_{f5} = \{s_5\}$
- $f_6 = f_4 \vee f_5$ $\quad Sat_{f6} = \{s_4, s_5\}$
- $f_7 = \mathbf{AF} f_6$. Формулой f_7 сначала помечаем те состояния, которые уже помечены f_6. Это состояния $\{s_4, s_5\}$. Затем в цикле ищем те состояния, из которых все переходы ведут в состояния, уже помеченные f_7. Это только состояние s_1. Таким образом, $Sat_{f7} = \{s_1, s_4, s_5\}$.
- $\Phi = f_8 = \mathbf{E}(f_3 \mathbf{U} f_7)$. Формулой f_8 сначала помечаем те состояния, которые уже помечены f_7. Это состояния $\{s_1, s_4, s_5\}$. Затем в цикле добавляем к ним те состояния, помеченные f_3, из которых хотя бы один переход ведет в состояние, помеченное f_8. Это состояния s_2 и s_3. Таким образом, $Sat_{f8} = \{s_1, s_2, s_3, s_4, s_5\}$.

Поскольку начальное состояние s_0 не принадлежит множеству тех состояний, в которых выполняется Φ, то формула Φ не выполняется на структуре Крипке M.

3.8. Заключение

Темпоральная логика ветвящегося времени (CTL) — это подкласс обобщенной темпоральной логики ветвящегося времени CTL*. В обеих этих логиках принимается "ветвящаяся" модель времени: будущее не определено однозначно, в каждом состоянии вычислений существуют различные альтернативы развития вычисления в будущем, хотя только одна из этих альтернатив действительно реализуется. Такой взгляд на время широко используется для спецификации свойств дискретных динамических систем.

В CTL каждому темпоральному оператору (**X**, **G**, **F**, **U**) должен непосредственно предшествовать один из кванторов пути (**A** или **E**). Вследствие этого любая формула CTL является формулой состояния, т. е. принимает истинное или ложное значение на каждом состоянии структуры Крипке. Такое ограничение существенно упрощает алгоритм проверки выполнимости спецификаций на структуре Крипке. Наиболее удобным для проверки выполнения формулы логики CTL на структуре Крипке является так называемый алгоритм разметки состояний: все состояния структуры Крипке помечаются теми подформулами формулы CTL, которые истинны в этих состояниях. Такую разметку можно выполнять для различных базисов логики CTL.

Алгоритм model checking для формулы CTL линеен как относительно сложности этой формулы (числа ее подформул), так и относительно сложности структуры Крипке (числа состояний и переходов). Такая удивительная простота алгоритмов проверки выполнимости формул CTL привела к "феноменальному" [140] успеху методов верификации, основанных на CTL.

3.9. Замечания

Алгоритм model checking для формул CTL был разработан в начале 80-х годов прошлого века независимо двумя группами: группой профессора Э. Кларка (E. M. Clarke) из университета Карнеги-Меллона, США, и группой Дж. Сифакиса (J. Sifakis) из исследовательского центра VERIMAG, Франция. Этот алгоритм оказался очень полезным для анализа тонких редко проявляющихся ошибок в параллельных системах, поскольку обследует все пространство состояний системы, представленной моделью с конечным числом состояний.

Приведенные в этой главе алгоритмы для обработки подформул CTL представляют собой модификацию алгоритмов, представленных в монографии [34]. Оценки сложности алгоритмов проверки модели для CTL были сделаны в работе [31].

3.10. Задачи к главе 3

3.1. Докажите, что следующие множества комбинаций "квантор пути, темпоральный оператор" *не* являются базисами CTL:

- ❒ $\{\mathbf{EX}, \mathbf{AG}, \mathbf{EU}\}$
- ❒ $\{\mathbf{AX}, \mathbf{AF}, \mathbf{AU}\}$

3.2. Выразите все комбинации "квантор пути, темпоральный оператор" логики CTL

а) через базис $\{\mathbf{EX}, \mathbf{AU}, \mathbf{EU}\}$

б) через базис $\{\mathbf{EX}, \mathbf{EG}, \mathbf{EU}\}$

3.3. Следующие формулы CTL представьте с помощью операторов базиса $\{\mathbf{EX}, \mathbf{EG}, \mathbf{EU}\}$:

а) $A\big(p\mathbf{U}\big(\mathbf{EX}q\big)\big)$

б) $\mathbf{AG}\big(p \vee \mathbf{EX}q\big)$

в) $\mathbf{AX}\Big(\mathbf{EF}\, p \vee \mathbf{A}\big(\big(\mathbf{EX}q\big)\mathbf{U}\, p\big)\Big)$

3.4. Для следующих формул постройте синтаксические деревья, определяющие структуру их подформул в соответствии с грамматикой формул CTL:

а) $\mathbf{A}\Big(p\mathbf{U}\big(r \Rightarrow \big(\neg p \Rightarrow \mathbf{AG}q\big)\big)\Big)$

б) $\mathbf{AG}\big(p \vee \mathbf{EX}q\big)$

3.5. Постройте процедуры проверки модели для формул $\mathbf{AX}p$, $\mathbf{EF}q$, $\mathbf{A}\big(q\mathbf{U}r\big)$, а также графическое представление соответствующих процедур, аналогичное рис. 3.7.

3.6. Пусть $M = \big(S, S_0, R, AP, L\big)$ — такая структура Крипке:

$$S = \{s_0, s_1, s_2, s_3\},\ S_0 = \{s_1, s_3\},$$
$$R = \big\{(s_2, s_3), (s_2, s_0), (s_0, s_1), (s_1, s_2), (s_3, s_0), (s_2, s_2)\big\},$$
$$AP = \{a, b\},\ L(s_0) = \{a\},\ L(s_1) = \{a, b\},\ L(s_3) = \varnothing.$$

а) В каких состояниях M выполняется формула CTL $\mathbf{EX}\,\mathbf{AX}b$?

б) В каких состояниях M выполняется формула CTL $\mathbf{A}\big(\big(\mathbf{EX}b\big)\mathbf{U}\, a\big)$?

в) Выполняется ли формула $\mathbf{EG}\,\mathbf{A}\big(b\,\mathbf{U}\, a\big)$ на структуре Крипке M?

❖ ***Замечание***

Заданная структура Крипке имеет два начальных состояния. В такой структуре Крипке формула логики CTL выполняется, если она выполняется в каждом начальном состоянии этой структуры.

3.7. Пусть $M=(S,S_0,R,AP,L)$ — такая структура Крипке:

$S=\{s_0,s_1,s_2,s_3\}$, $S_0=\{s_0\}$,

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

Известно, что формула **EF**φ выполняется в состоянии s_3. Будет ли **EF**φ выполняться в s_0? В s_1? В s_2? В M?

3.8. Пусть $M=(S, S_0, R, AP, L)$ — такая структура Крипке:

$S=\{p, q, r, s\}$; $S_0=\{p, r\}$;

$R=\{(p, s),(q, r),(s, s),(q, s),(s, q),(r, p),(p,r)\}$; $AP=\{a, b, c\}$

$L:S\to 2^{AP}$; $L(p)=\{a,b\}$; $L(q)=\{c\}$; $L(r)=\varnothing$; $L(s)=\{b\}$.

а) постройте графическое представление структуры Крипке;

б) проверьте истинность:

$M\models$ **AFEX**$\neg a$

$M\models$ **EXAF**$(a\Rightarrow b)$

$M\models$ **AG**$(c\Rightarrow$ **EF**$b)$

3.9. Доказать, что формулы: $(\mathbf{AG}p)\Rightarrow p$, $p\Rightarrow \mathbf{A}(q\mathbf{U}p)$, $p\Rightarrow \mathbf{EF}p$ являются тавтологиями, т. е. они истинны в каждом состоянии любой структуры Крипке.

3.10. Какие из следующих пар CTL-формул являются тождественными:

а) **AXAF**φ и **AFAX**φ

б) **EXEF**φ и **EFEX**φ

в) **AXAG**φ и **AGAX**φ

г) **EXEG**φ и **EGEX**φ

д) **EFEG**φ и **EGEF**φ

е) **EG**φ и φ ∨ **EXEG**φ

ж) $\mathbf{EG}\varphi$ и $\varphi \wedge \mathbf{EXEG}\varphi$

з) $\mathbf{EF}\varphi$ и $\varphi \vee \mathbf{EXEF}\varphi$

и) $\mathbf{EF}\varphi$ и $\varphi \wedge \mathbf{EXEF}\varphi$

3.11. Дана формула CTL

$$\mathbf{EXA}(\varphi \mathbf{U} \psi)$$

Постройте структуру Крипке, на которой эта формула истинна, и другую, на которой эта формула ложна.

3.12. Постройте структуру Крипке, на которой одна из формул, указанных в перечисленных ниже парах, выполняется, а другая формула — нет:

а) $\mathbf{EF}\varphi$ и $\mathbf{EG}\varphi$

б) $\mathbf{EF}(\varphi \wedge \psi)$ и $\mathbf{EF}\varphi \wedge \mathbf{EF}\psi$

в) $\mathbf{EG}(\varphi \vee \psi)$ и $\mathbf{EG}\varphi \vee \mathbf{EG}\psi$

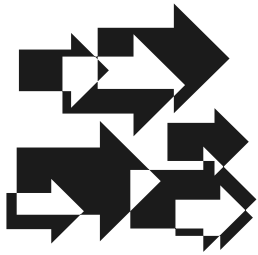
г) $\mathbf{AG}(\varphi \wedge \psi)$ и $\mathbf{AG}\varphi \wedge \mathbf{AG}\psi$

3.13. Объясните, что выражает формула CTL: $\mathbf{AG}(p \Rightarrow \mathbf{AF}(s \wedge \mathbf{AX}t))$ в терминах порядка наступления событий p, s и t.

3.14. Возможна ли ситуация, когда для одной и той же структуры Крипке M и одной и той же CTL-формулы Φ справедливо, что на M не выполняется ни Φ, ни ее отрицание, т. е. справедливо как $M \not\models \Phi$, так и $M \not\models \neg\Phi$?

3.15. Известно, что LTL-формула $\mathbf{GF}\,@s$ (любое вычисление проходит состояние s неопределенно часто) не является формулой CTL. Можно ли подобное свойство выразить формулой логики CTL?

ГЛАВА 4



Алгоритм model checking для LTL

В этой главе рассматривается алгоритм проверки моделей для еще одного подкласса темпоральных логик — логики линейного времени LTL. Эта логика интуитивно более понятна: время в ней рассматривается как текущее в одном направлении, без "разветвлений", с единственным будущим для каждого состояния вычисления. Формулами логики LTL выражаются важные свойства поведения реагирующих систем, причем некоторые из этих свойств выразить формулами логики ветвящегося времени CTL нельзя (впрочем, как было указано в *гл. 2*, справедливо и обратное: некоторые свойства, выражаемые в логике CTL, нельзя выразить в логике LTL). Разработано несколько подходов к проверке выполнимости LTL-формул на структуре Крипке. Наиболее практичный подход основывается на теории автоматов, его мы и рассмотрим здесь.

Алгоритм проверки выполнимости формулы LTL на структуре Крипке, построенный на автоматном подходе, не совсем прост. Во-первых, он использует непростые формальные конструкции (языки со словами бесконечной длины и автоматы, задающие такие языки, — автоматы Бюхи). Во-вторых, сложность этого алгоритма экспоненциальна относительно сложности формулы LTL. Тем не менее, поскольку обычно формулы, которые проверяются для реагирующих систем, являются небольшими, алгоритм проверки моделей для LTL может быть применен и для практических систем. Именно этот подход реализован в нескольких системах верификации, наибольшую известность среди которых получила система Spin. С помощью системы Spin верифицированы многие практические разработки.

4.1. Проверка выполнимости LTL-формул на вычислениях

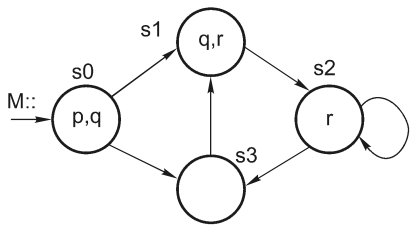
Как мы видели в *гл. 2*, все формулы линейной темпоральной логики LTL задаются грамматикой:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{X}\varphi \mid \varphi\mathbf{U}\varphi$$

Дополнительными (выводимыми) в этой логике являются все булевы операции (константы *true* и *false*, операции конъюнкции, импликации и т. п.), а также темпоральные операторы **F** и **G**.

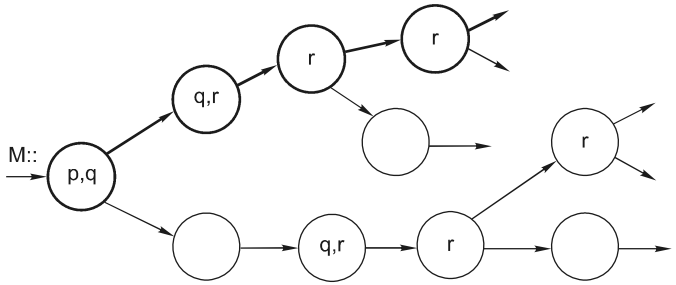
Алгоритм верификации для формул LTL в корне отличается от рассмотренного в предыдущей главе алгоритма проверки выполнимости формул CTL. Отличие объясняется фундаментальным различием семантик формул этих двух логик. Формулы логики CTL интерпретируются на деревьях вычислений, и все подформулы формулы CTL — это формулы состояний. Формулы логики LTL интерпретируются на вычислениях — бесконечных последовательностях состояний с определенными в них множествами атомарных предикатов. Все подформулы формулы LTL — это формулы пути. Для формул CTL алгоритмом проверки моделей является представленный в *гл. 3* удобный и ясный алгоритм маркировки состояний структуры Крипке подформулами исходной формулы. Алгоритм проверки моделей для формул LTL не может напрямую использовать аналогичный подход: истинность каждой подформулы формулы LTL может требовать анализа бесконечных вычислений, начинающихся в этом состоянии.

Алгоритм проверки модели для формулы φ логики LTL должен убедиться в выполнении свойства, выраженного формулой φ, на *каждом* вычислении, начинающемся в начальном состоянии структуры Крипке. Например, для проверки того, выполняется ли на структуре Крипке M (рис. 4.1, *а*) свойство, выраженное формулой $\varphi = \mathbf{GF}(r \Rightarrow q)$, нужно проверить, что на всех возможных вычислениях M (а их бесконечное число!) формула $r \Rightarrow q$ будет истинной бесконечное число раз!



а

Рис. 4.1. Структура Крипке (*а*)



б

Рис. 4.1. Дерево вычислений структуры Крипке (*б*)

Поиск контрпримеров

Говоря более точно, выполнение формулы LTL должно проверяться не на вычислениях структуры Крипке M, а на траекториях M, т. е. на последовательностях подмножеств атомарных предикатов, истинных в последовательно проходимых состояниях вычисления. Например, выделенному на рис. 4.1, *б* вычислению $s_0 s_1 s_2 s_2 \ldots$ соответствует траектория $\{p, q\}\{q, r\}\{r\}\{r\}\ldots$.

Заметим, что для формулы LTL φ и некоторой структуры Крипке M может оказаться, что на M не выполняются ни φ, ни $\neg\varphi$. Это будет в том случае, когда на некоторых вычислениях M формула φ выполняется, а на других вычислениях M она не выполняется (т. е. на этих вычислениях выполняется $\neg\varphi$).

Если на структуре Крипке M *не* выполняется формула φ, то M имеет *хотя бы одно* вычисление, удовлетворяющее $\neg\varphi$. Такое вычисление назовем контрпримером. Например, на рис. 4.1, *б* видно, что некоторые траектории M удовлетворяют формуле $\varphi = \mathbf{GF}(r \Rightarrow q)$ (например, траектория $\{p, q\}\{q, r\}\{r\}\{\ \}\{q, r\}\{r\}\{\ \}\ldots$), а другие — не удовлетворяют ей (например, траектория $\{p, q\}\{q, r\}\{r\}\{r\}\{r\}\{r\}\ldots$). Эта последняя траектория может послужить контрпримером, подтверждающим, что формула $\varphi = \mathbf{GF}(r \Rightarrow q)$ не выполняется на структуре Крипке M. Очевидно, что даже одного контрпримера достаточно, чтобы опровергнуть утверждение $M \models \varphi$. Если контрпримеров нет, то формула выполняется на данной структуре Крипке. Именно метод поиска контрпримеров лежит в основе теоретико-автоматного подхода — одного из наиболее удобных и ясных подходов к разработке алгоритма проверки моделей для формул LTL.

Очевидно, что бессмысленно искать контрпримеры перебором всех траекторий структуры Крипке. Мало того, что самих траекторий бесконечное число, каждая такая траектория бесконечна. Поэтому решение этой проблемы нетривиально.

ω-слова и ω-языки

> ❖ ***Определение 4.1 (ω-слова и ω-языки)***
>
> Любое конечное множество Σ будем называть словарем или алфавитом. Бесконечные цепочки, составленные из элементов (символов) словаря Σ, называются ω-словами над Σ. Любое множество ω-слов над словарем Σ называется ω-языком над Σ.

Множество всех бесконечных цепочек из символов словаря Σ обозначается Σ^ω по аналогии с обозначением Σ^* множества всех конечных цепочек из символов Σ. Например, если $\sum = \{a, b, c\}$, то Σ^ω составят все бесконечные цепочки из этих символов, т. е. $\Sigma^\omega = \{aaaaa..., abababab..., baccbaccbacc...,...\}$, в то время, как Σ^* составят все конечные цепочки: $\Sigma^* = \{\varepsilon, a, b, c, ab, aa, ba, abac, ...\}$. Символом ε обозначается пустая цепочка, вовсе не содержащая символов. Примером ω-языка над алфавитом $\Sigma = \{a, b\}$ является язык $b^* a^\omega$, состоящий из всех таких цепочек, у которых конечный (возможно, пустой) начальный фрагмент состоит только из символов b, а после него идет бесконечная последовательность символов a. Этот язык в точности описывается LTL-формулой $(b \wedge \neg a)\mathbf{U}\mathbf{G}(a \wedge \neg b)$. Таким образом, темпоральные формулы LTL могут задавать ω-языки — множества тех бесконечных цепочек, которые удовлетворяют этой формуле. Оказывается, что ω-языки могут быть заданы и автоматами.

Контрольный автомат

Идея, лежащая в основе алгоритма model checking для заданной LTL-формулы, близка идее использования контрольного автомата (*watchdog*, буквально — *сторожевого пса*), проверяющего правильность функционирования основного устройства. Контрольный автомат (рис. 4.2, *а*) строится по спецификации требований к проверяемому устройству и работает параллельно и синхронно с основным устройством. Синхронное выполнение означает, что контрольный автомат перехватывает все входящие и выходящие сигналы, и с каждым шагом вычисления системы контрольный автомат также делает

один шаг, если он в нем возможен. Если последовательность этих сигналов не соответствует требованиям спецификации, то контрольный автомат переходит в свое состояние "*ошибки*". Например, на рис. 4.2, *б* представлен контрольный автомат, переходящий в ошибочное состояние (E) и остающийся там навсегда в том случае, если хотя бы раз на полученный запрос (request) в течение 2 сек. основным устройством не будет выдан ответ (response). Достаточно найти только одну неправильную траекторию поведения системы, чтобы доказать некорректность работы основного устройства. Сама проверяемая система может быть очень сложной, поэтому во многих случаях такую проверку осуществить более просто, чем проверять, что *все* вычисления из бесконечного числа вычислений системы правильны.

Контрольный автомат не обязательно использовать для проверки *реализации* основного устройства, такую проверку можно выполнить еще на этапе проектирования системы, используя формальные описания проверяемой системы и контрольного автомата. В частности, это может быть сделано, если формальная модель реальной системы представлена структурой Крипке, а все "правильные" поведения заданы формулами логики LTL.

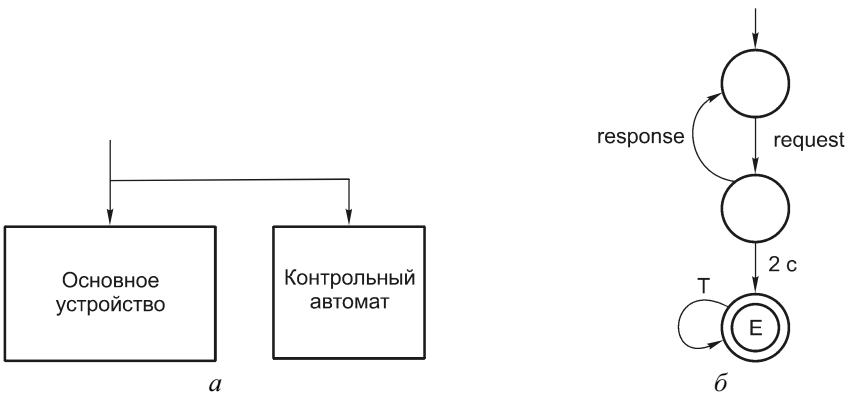


Рис. 4.2. *а*) Контроль правильности функционирования с помощью контрольного автомата; *б*) пример контрольного автомата (T означает True, переход при любом входном символе)

Проблему построения контрольного автомата для проверки правильности функционирования дискретной системы в общем случае для любой LTL-формулы φ, задающей "правильные" поведения, и для любой структуры Крипке M можно формализовать в терминах формальных языков. Если мы сможем задать конечной моделью два ω-языка: ω-язык L_M, допускаемый M, и ω-язык $L_{\neg\varphi}$, все цепочки которого "неправильны", потому что они удовлетворяют LTL-формуле $\neg\varphi$, и затем сможем найти пересечение

$L_M \cap L_{\neg\varphi}$ этих ω-языков, то наша проблема будет решена. Действительно, если это пересечение окажется пустым, то на структуре Крипке M формула φ выполняется, поскольку у M нет ни одного вычисления, на котором выполняется формула $\neg\varphi$. Если пересечение $L_M \cap L_{\neg\varphi}$ окажется непустым, то M допускает какие-нибудь вычисления, удовлетворяющие $\neg\varphi$, следовательно, неверно, что *все* вычисления M удовлетворяют формуле φ (рис. 4.3).

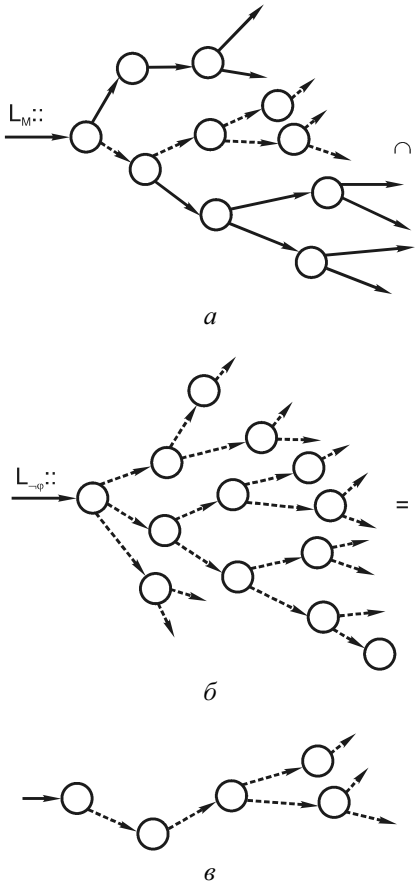


Рис. 4.3. Основная идея проверки выполнимости LTL-формул

Рассмотрим, как эта идея формализуется и реализуется в более простом случае в теории конечных автоматов и автоматных языков.

4.2. Пересечение языков в теории конечных автоматов

Конечные автоматы и автоматные языки

В теории конечных автоматов и автоматных языков эта идея может быть формализована следующим образом. Языком над конечным словарем называется любое (обычно бесконечное) множество конечных цепочек, построенных из символов словаря. Конечные автоматы являются удобным формализмом, позволяющим задавать некоторый класс языков и выполнять операции над ними.

Языки, которые можно задать конечными автоматами, называются автоматными языками. Для двух автоматных языков, L_A и L_B, которые задаются автоматами A и B, проверка непустоты их пересечения осуществляется простым алгоритмом, анализирующим конструкцию, которая называется *синхронной композицией* автоматов A и B. Синхронная композиция автоматов — это просто два автомата, стоящие рядом и синхронно обрабатывающие одну и ту же входную цепочку.

❖ ***Определение 4.2 (конечный автомат)***

Конечным автоматом называется пятерка $(S, \Sigma, s_0, \delta, F)$, где:

- S — конечное множество состояний;
- Σ — конечное множество символов (входной словарь);
- $s_0 \in S$ — начальное состояние (для недетерминированного случая множество S_0);
- $\delta : S \times \Sigma \to S$ — функция переходов (для недетерминированного автомата $\delta : S \times \Sigma \to 2^S$);
- $F \subseteq S$ — множество допускающих (финальных, заключительных) состояний.

Любая конечная цепочка, построенная из символов словаря Σ, называется словом над Σ. Множество всех слов над Σ обозначается Σ^*. Хотя множество Σ^* состоит только из конечных слов, само оно является бесконечным (счетным).

Динамика (поведение) конечного автомата определяется его переходами из состояния в состояние под воздействием очередного входного символа. Ко-

нечный автомат допускает (распознает) слово $w \in \Sigma^*$, если w переводит автомат из начального в одно из допускающих состояний. Множество всех слов, которые допускает автомат A, называется языком, допускаемым (распознаваемым) A, и обозначается L_A. Множество слов $L \in \Sigma^*$ называется автоматным языком, если существует конечный автомат, допускающий L. В общем случае язык содержит бесконечное число слов, и возможность задавать такие бесконечные множества с помощью конечной формальной модели — конечного автомата — является весьма важной.

Пример 4.1

На рис. 4.4 приведены графы переходов двух конечных автоматов, A и B. Начальные состояния автоматов на рис. 4.4 указаны стрелками, допускающие состояния обозначены двойными окружностями. Обозначим L_A язык, допускаемый автоматом A. Слова языка, допускаемого конечным автоматом, — это конечные цепочки из символов словаря $\{a, b, c\}$, переводящие автомат из начального состояния в любое из допускающих состояний. Оба автомата допускают языки, содержащие бесконечное число конечных слов. Например, автомат A, получив на вход цепочку bab, перейдет в состояние 2, которое не является допускающим, и, следовательно, эта цепочка НЕ принадлежит L_A — языку, допускаемому автоматом A. С другой стороны, цепочка acc допускается автоматом A, следовательно, $acc \in L_A$. Ни одно слово, начинающееся с символа c, не допускается автоматом A, потому что из начального состояния вообще нет перехода, помеченного c. Если в автомате из состояния s не показан переход под воздействием некоторого символа $a \in \Sigma$, то будем считать, что этот переход ведет в новое состояние, неявно представленное в графе переходов, из которого любое допускающее состояние недостижимо.

Можно видеть, что $L_A = \{ba, acc, baba, baccba, ...\}$, $L_B = \{a, ba, baba, babbba, ...\}$. Если для множества слов L существует допускающий это множество конечный автомат, то L называется автоматным языком или регулярным множеством. Такой язык можно также задать регулярным выражением.

Таким образом, конечные автоматы являются удобным конечным формализмом, позволяющим задавать бесконечные множества слов (языки) над конечным словарем.

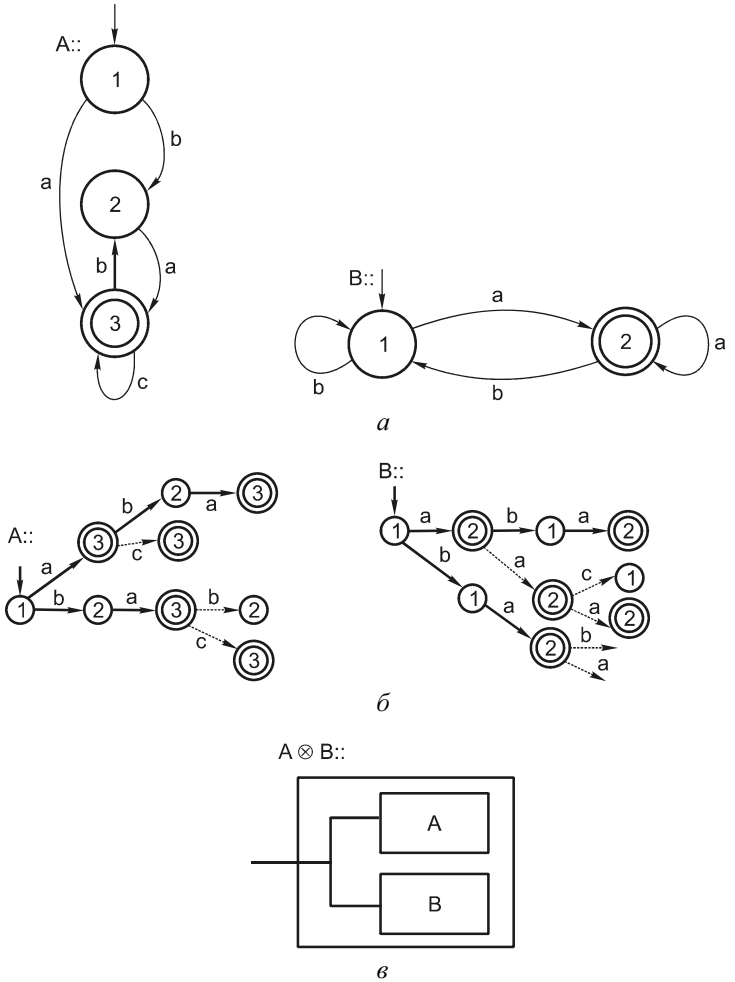


Рис. 4.4. Два конечных автомата (*а*) и развертки их поведений (*б*). Выделены цепочки, допускаемые обоими автоматами.
в) Синхронная композиция автоматов — два автомата, работающие одновременно

Синхронная композиция автоматов и пересечение языков

Из рис. 4.4 видно, что языки L_A и L_B пересекаются: некоторые цепочки их общего входного алфавита принадлежат как L_A, так и L_B. Теория конечных автоматов позволяет решить проблему пустоты пересечения языков L_A и L_B. Для этого строится конечный автомат $A \otimes B$ — так называемая синхронная

композиция конечных автоматов A и B (иногда и эту операцию над автоматами, и ее результат называют декартовым или синхронным произведением автоматов). Синхронная композиция конечных автоматов A и B — это конечный автомат, моделирующий работу двух конечных автоматов, которые одновременно, синхронно обрабатывают один и тот же вход (рис. 4.4, *в*).

Множеством состояний автомата $A \otimes B$ являются все пары состояний A и B. Будем считать, что автомат $A \otimes B$ находится в состоянии $\langle s_A, s_B \rangle$, если автомат A находится в состоянии s_A, а автомат B находится в состоянии s_B. Переход автомата $A \otimes B$ из текущего состояния $\langle s_A, s_B \rangle$ в следующее состояние $\langle s'_A, s'_B \rangle$ под воздействием входного символа a возможен, если каждый из автоматов под воздействием a может осуществить переход: A из s_A в s'_A, а B из s_B в s'_B.

Начальными состояниями $A \otimes B$ являются пары, оба элемента которых являются начальными состояниями соответствующих автоматов. Допускающие состояния композиции автоматов A и B — это такие пары $\langle s_A, s_B \rangle$ состояний, что s_A — допускающее состояние автомата A, а s_B — допускающее состояние автомата B.

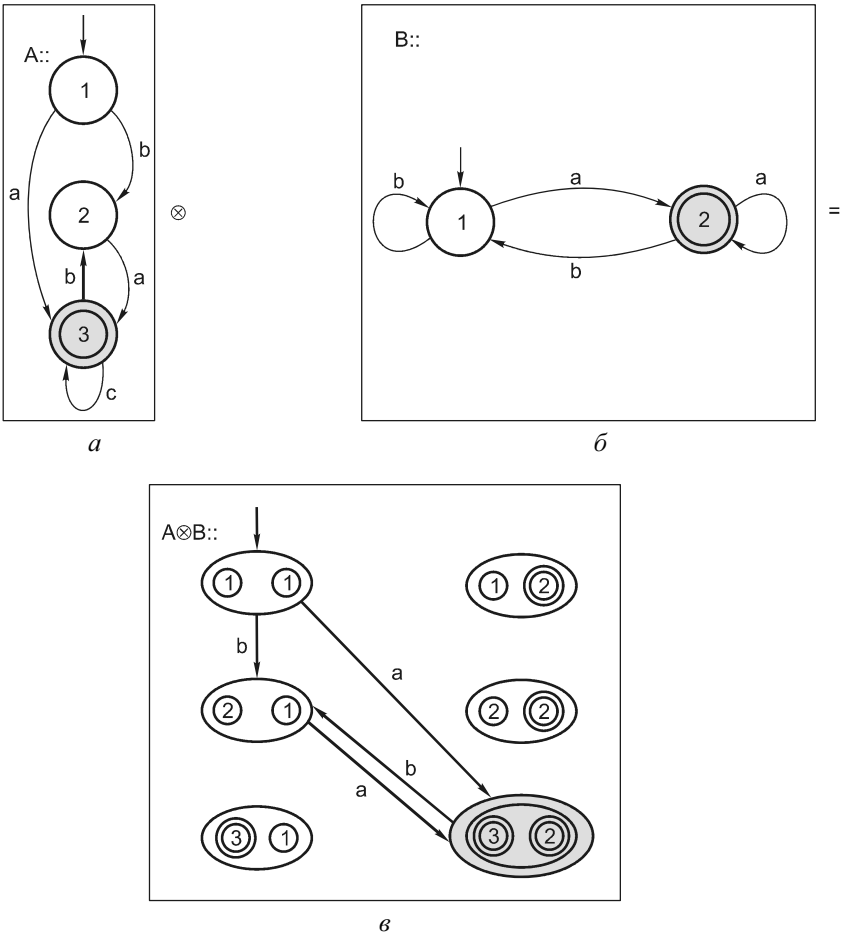
Пример 4.2

На рис. 4.5 построен конечный автомат $A \otimes B$ по автоматам, представленным на рис. 4.4. Начальным состоянием автомата $A \otimes B$ является пара $\langle 1, 1 \rangle$ начальных состояний A и B. Допускающее состояние у автомата $A \otimes B$ только одно: $\langle 3, 2 \rangle$, поскольку состояние 3 является допускающим в автомате A, а состояние 2 является допускающим в автомате B. В автомате $A \otimes B$ из начального состояния $\langle 1, 1 \rangle$ есть переход в состояние $\langle 2, 1 \rangle$, помеченный b, потому что и в автомате A, и в автомате B есть соответствующие переходы, помеченные b. Цепочка ba переводит автомат $A \otimes B$ в его допускающее состояние, следовательно, она допускается этим автоматом. Очевидно, что эта цепочка допускается как автоматом A, так и автоматом B — слово ba переводит каждый автомат из его начального в его допускающее состояние.

Справедлива следующая теорема.

ТЕОРЕМА 4.1 (о пересечении автоматных языков)

Пересечение языков, допускаемых конечными автоматами A и B, совпадает с языком, допускаемым синхронной композицией этих автоматов, т. е. $L_A \cap L_B = L_{A \otimes B}$.



а *б* *в*

Рис. 4.5. Синхронная композиция автоматов A и B допускает язык $L_A \cap L_B$

На основании этой теоремы проверка *пустоты* пересечения двух автоматных языков выполняется просто. А именно, по автоматам A и B строится их синхронная композиция $A \otimes B$, и если в автомате $A \otimes B$ допускающие состояния не достижимы из начального состояния, то языки L_A и L_B не пересекаются.

Посмотрим на синхронную композицию автоматов A и B с другой точки зрения. Назовем автомат B *контрольным*. Пусть *контрольный* автомат определяет все такие цепочки, которые мы считаем "*плохими*", "*ошибочными*". Тогда любое слово, допускаемое синхронной композицией автоматов $A \otimes B$, будет "*ошибочным*" словом, которое присутствует среди бесконечного мно-

жества слов, допускаемых автоматом A, потому что оно допускается и контрольным автоматом B тоже. Например, любая цепочка, которая допускается синхронной композицией основного устройства и контрольного автомата рис. 4.2, является ошибочной для этого основного устройства.

Определим в примере 4.2 любую цепочку, которая состоит из a и b, но заканчивается a, как "ошибочную". Автомат B на рис. 4.5 определяет как раз все такие "ошибочные" цепочки. Поставим вопрос: существует ли среди всех цепочек, допускаемых автоматом A, хотя бы одна "ошибочная"? Для ответа на этот вопрос можно не анализировать все возможные пути, ведущие в допускающие состояния автомата A. Достаточно построить синхронную композицию автоматов A и B и *проверить пустоту* языка $L_{A\otimes B}$. Если допускаемый автоматом $A\otimes B$ язык $L_{A\otimes B}$ непуст, то среди бесконечного множества цепочек, которые допускает A, есть и ошибочные цепочки. Более того, мы можем указать и путь, контрпример, показывающий, через какие состояния автомата A проходит "ошибочная" цепочка. Для автомата A примера на рис. 4.5 одним из контрпримеров является путь $1\to 2\to 3$. Действительно, путь $1\to 2\to 3$ выполняется в A под воздействием входной цепочки ba, которая является ошибочной — она заканчивается символом a.

Таким образом, анализируя конечные модели (конечные автоматы), мы получаем ответ на вопрос о том, включает ли *бесконечное* множество слов (язык L_A) какие-нибудь слова из другого *бесконечного* множества (языка L_B) "ошибочных" слов.

Алгоритм проверки моделей для LTL-формул работает полностью аналогично, хотя оперирует несколько более сложными формальными моделями.

4.3. Теоретико-автоматный метод проверки выполнимости LTL-формулы

Для реализации этой идеи в области верификации реагирующих систем, моделью которых является структура Крипке, требуется уметь конечным образом задавать язык L_M — все цепочки, возможные в произвольной структуре Крипке M, уметь конечным образом задавать язык $L_{\neg\varphi}$ — все цепочки, не удовлетворяющие заданной формуле φ логики LTL, а также уметь строить пересечение этих языков и проверять, является ли оно пустым. При этом мы не только сможем доказать, что $M\not\models\varphi$, но и найдем контрпример — те вычисления M, которые удовлетворяют формуле $\neg\varphi$.

Как уже говорилось выше, цепочки, характеризующие вычисления реагирующих систем, являются бесконечными, они называются ω-словами. Мно-

жества ω-слов называются ω-языками. Оказывается, что некоторые ω-языки можно задать конечной моделью, которая называется автоматом Бюхи. Теоретико-автоматный подход к верификации реагирующих систем состоит в том, что строится автомат Бюхи, задающий все траектории анализируемой системы, другой автомат Бюхи, задающий все неправильные траектории, и анализируется синхронная композиция этих автоматов.

Более точно, алгоритм проверки выполнимости формулы φ LTL на структуре Крипке M состоит из следующих шагов (рис. 4.6):

1. По структуре M строится автомат Бюхи B_M. Этот автомат допускает все возможные траектории (ω-слова) структуры Крипке M.

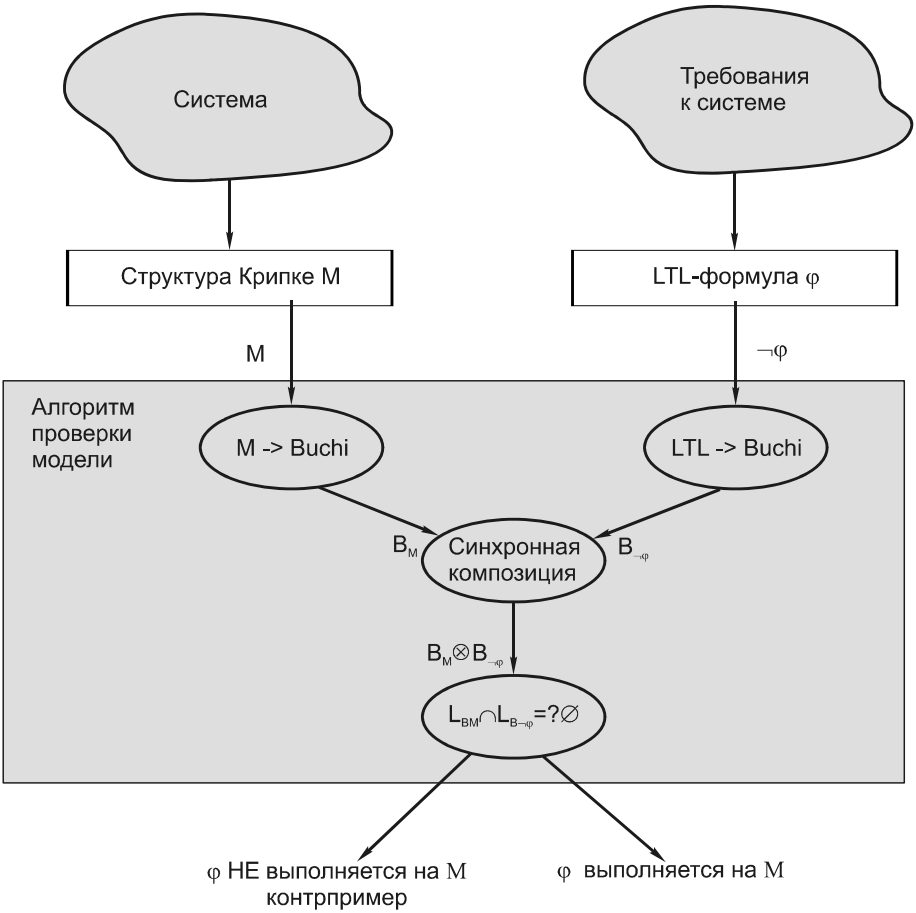


Рис. 4.6. Последовательность шагов алгоритма model checking для LTL-формулы

2. По формуле φ строится ее отрицание $\neg\varphi$ и затем по формуле $\neg\varphi$ строится автомат Бюхи $B_{\neg\varphi}$, допускающий множество ω-слов, которые удовлетворяют $\neg\varphi$. Этот автомат будет "контрольным" автоматом, допускающим все ошибочные траектории.
3. Строится автомат $B_M \otimes B_{\neg\varphi}$ — синхронная композиция автомата B_M и "контрольного" автомата $B_{\neg\varphi}$. Этот автомат допускает пересечение языков, допускаемых компонентными автоматами, т. е. все цепочки, допускаемые как автоматом B_M, так и автоматом $B_{\neg\varphi}$.
4. Проверяется пустота языка $L_{B_M \otimes B_{\neg\varphi}}$. Формула φ выполняется для M, если и только если этот язык пуст. Если этот язык непуст, то формула φ не выполняется для M, структура Крипке имеет ошибочную траекторию поведения, которая удовлетворяет $\neg\varphi$, и можно построить *контрпример* (ошибочный путь), т. е. путь в M, нарушающий свойство φ.

Все введенные понятия формально определяются в следующем разделе. Мы начнем с обоснования необходимости введения нового типа формальных моделей — автомата Бюхи, рассмотрим свойства таких автоматов, правила построения их синхронной композиции. Далее обсудим связь формул LTL и автоматов Бюхи как моделей для задания конечным образом свойств бесконечных вычислений. Затем рассмотрим алгоритм построения синхронной композиции двух автоматов Бюхи и алгоритм определения того, что автомат Бюхи допускает пустой язык. В конце главы рассмотрим правила построения автомата Бюхи, допускающего все такие цепочки, на которых заданная LTL-формула выполняется.

4.4. Автоматы Бюхи — модели для задания ω-языков

Классическая теория формальных языков и конечных автоматов построена для анализа цепочек символов, которые могут иметь КОНЕЧНУЮ, хотя и произвольную длину. Формальные языки являются формальными моделями естественных и искусственных языков, все слова которых конечны (например, сколь бы длинную фразу я ни начал, я когда-нибудь замолчу, какой бы длинный роман ни был, он когда-нибудь закончится, программист может написать только конечную программу, все сообщения имеют только конечную длину и т. п.). Теория формальных языков и конечных автоматов имеет разнообразные применения в трансляции языков программирования, анализе текстов и многих других областях. Все эти приложения имеют дело только

с конечными цепочками символов, построенными из символов конечного словаря. Еще раз подчеркнем, что число таких конечных цепочек может быть бесконечным.

ω-языки и поведение реагирующих систем

Цепочки, порождаемые вычислениями реагирующих систем, являются бесконечными, поскольку эти системы после запуска должны функционировать БЕСКОНЕЧНО долго. Формальные модели, используемые при верификации таких систем, должны иметь дело с бесконечными цепочками — ω-словами, и их множествами — ω-языками. Поэтому первой нашей задачей является следующая: как с помощью автомата описать ω-язык?

Конечные автоматы являются удобным формализмом, который конечным образом задает обычные формальные языки — бесконечные множества цепочек конечной длины. Нам нужен новый тип автомата, задающего ω-языки — бесконечные множества бесконечных цепочек. Можно ли задать хотя бы некоторые ω-языки конечными моделями, аналогичными конечным автоматам?

Существует несколько возможных путей обобщения модели конечного автомата таким образом, чтобы он мог допускать бесконечные цепочки. Мы рассмотрим два таких формализма, которые были созданы в середине прошлого века: автоматы Бюхи и обобщенные автоматы Бюхи.

Автомат Бюхи можно считать недетерминированным конечным автоматом, получающим на вход бесконечные слова. В отличие от конечного автомата, условие допустимости входной цепочки в автомате Бюхи изменено так, чтобы можно было определять некоторые бесконечные цепочки (ω-слова) как допустимые.

❖ ***Определение 4.3 (автомат Бюхи)***

Автомат Бюхи — это пятерка $(S, \Sigma, S_0, \delta, F)$, где:

- Σ — конечное множество символов (алфавит);
- S — множество состояний;
- $S_0 \subseteq S$ — множество начальных состояний (содержащее только один элемент для детерминированного случая);
- $\delta : S \times \Sigma \to 2^S$ — отношение переходов (для детерминированного автомата $\delta : S \times \Sigma \to S$);
- $F \subseteq S$ — множество допускающих (финальных) состояний.

Любое ω-слово w над алфавитом Σ можно считать функцией $w: \mathbf{N} \to \Sigma$ (где $\mathbf{N}$ — натуральные числа), понимая под $w(i)$ i-ю букву слова w. Аналогично, если $\pi: \mathbf{N} \to S$ — бесконечная последовательность состояний автомата A, то $\pi(i)$ — i-е состояние в цепочке π. Обозначим $inf(\pi)$ множество элементов из S, которые встречаются в цепочке π бесконечно много раз.

❖ ***Определение 4.4***

ω-слово w над алфавитом Σ допускается автоматом Бюхи $A = (S, \Sigma, S_0, \delta, F)$, если существует такое ω-слово π над множеством состояний S (бесконечная цепочка состояний автомата A), что:

- $\pi(0) \in S_0$ — цепочка состояний начинается одним из начальных состояний;
- $\forall i \geq 0 : \pi(i+1) \in \delta(\pi(i), w(i))$ — переход в очередное состояние вызван очередной буквой ω-слова w;
- $\inf(\pi) \cap F \neq \varnothing$ — среди бесконечно повторяющихся состояний цепочки π существует хотя бы одно допускающее.

Таким образом, автомат Бюхи допускает (распознает, принимает) ω-слово, если при чтении этого бесконечного слова автомат бесконечно много раз проходит хотя бы одно допускающее состояние. Множество ω-слов, допускаемых автоматом Бюхи A, называется ω-языком, допускаемым A. ω-язык $L \subseteq \Sigma^{\omega}$ называется Бюхи-допускаемым (или ω-регулярным), если существует автомат Бюхи, допускающий L. Автоматы Бюхи эквивалентны, если допускаемые ими языки совпадают. Поскольку автоматы Бюхи используются для задания ω-языков, их также называют ω-автоматами.

Пример 4.3

Два автомата, A и B, приведенные на рис. 4.7, можно считать как конечными автоматами, так и автоматами Бюхи. Автомат A, рассматриваемый как автомат Бюхи, допускает любую цепочку из символов a и b, содержащую бесконечное число вхождений b. Число символов a в этих цепочках может быть произвольным. Формально это множество описывается ω-регулярным выражением $(a^* b)^{\omega}$ — любые цепочки, состоящие из бесконечного числа символов b, между которыми могут находиться конечные группы из символов a. Точно те же ω-слова допускает и автомат B, если его рассматривать как автомат Бюхи. Как конечный автомат, A допускает любую конечную це-

почку из символов a и b, оканчивающуюся на b. Все цепочки, допускаемые автоматом A, можно описать регулярным выражением $(a+b)^* b$. В дополнение к тем цепочкам, которые допускает A, конечный автомат B допускает еще и пустую цепочку: $L_B = L_A \cup \{\varepsilon\}$. Таким образом, A и B эквивалентны, если их рассматривать как автоматы Бюхи; как конечные автоматы на конечных словах они различаются. Заметим, что автоматы Бюхи A и B на рис. 4.7 допускают ω-слова, которые могут включать как конечное, так и бесконечное число символов a наряду с бесконечным числом символов b.

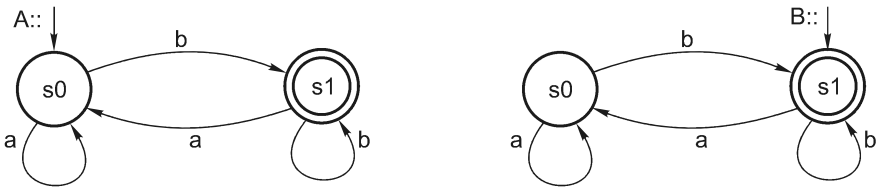


Рис. 4.7. Пример эквивалентных автоматов Бюхи

Пример 4.4

На рис. 4.8 представлен автомат Бюхи, который допускает все цепочки из a и b, содержащие только конечное число вхождений a и бесконечное число вхождений b, т. е. язык, задаваемый регулярным выражением $(a+b)^* b^{\omega}$. Этот автомат недетерминированный. Можно доказать, что не существует детерминированного автомата Бюхи, допускающего этот ω-язык (*см. задачу 4.4*). Таким образом, распознающая мощность недетерминированных автоматов Бюхи строго больше, чем детерминированных.

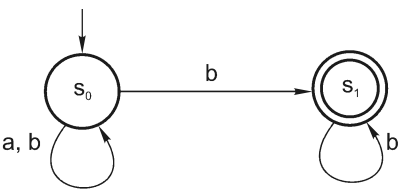


Рис. 4.8. Автомат Бюхи, допускающий цепочки с конечным числом вхождений символа a и бесконечным числом вхождений символа b

Пример 4.4 показывает, что автоматы Бюхи существенно отличаются от конечных автоматов: известно, что для каждого *недетерминированного конечного* автомата существует эквивалентный ему *детерминированный конечный* автомат, что неверно для автоматов Бюхи.

Автомат Бюхи как контрольный автомат для реагирующей системы

Итак, автомат Бюхи допускает бесконечные цепочки из символов словаря Σ, заставляющие его пройти последовательность состояний, в которой бесконечно часто встречается хотя бы одно допускающее состояние. Это значит, что автомат Бюхи допускает хотя бы одно ω-слово, если и только если существует достижимый из начального состояния цикл, проходящий через какое-либо его финальное состояние (рис. 4.9).

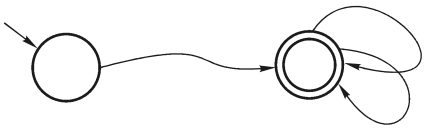


Рис. 4.9. Автомат Бюхи допускает ω-слово, если он имеет цикл с включенным допускающим состоянием

Пример 4.5

Идею использования автомата Бюхи для проверки выполнимости LTL-формулы на структуре Крипке можно пояснить простым примером. Пусть структура Крипке M представляет собой модель сложной системы управления перекрестком, в котором N_go — это разрешение движения (зеленый свет) в северном направлении, а W_go — разрешение движения в пересекающимся с ним западном направлении. Пусть LTL-формула φ выражает требование запрещения движения по двум пересекающимся направлениям одновременно, $\varphi = \neg\mathbf{F}\left(N_go \wedge W_go\right)$.

Автомат Бюхи $B_{\neg\varphi}$, допускающий все "ошибочные" цепочки, удовлетворяющие формуле $\neg\varphi = \mathbf{F}\left(N_go \wedge W_go\right)$, представлен на рис. 4.10. Поставим этот автомат рядом с автоматом Бюхи B_M, построенным по структуре Крипке M, чтобы $B_{\neg\varphi}$ перехватывал все входы и выходы B_M. Тогда $B_{\neg\varphi}$ будет выступать как "контрольный" автомат, который будет работать совместно и синхронно с B_M. Как только при функционировании B_M встретится переход, разрешающий одновременное движение по пересекающимся направлениям, $B_{\neg\varphi}$ перейдет в "ошибочное" состояние 1 и останется в нем навсегда. Состояние 1 в этом автомате допускающее, поэтому любое бесконечное входное слово, включающее в качестве символа хотя бы раз пару атомарных предикатов $\{N_go,\ W_go\}$, будет допускаться этим контрольным автоматом.

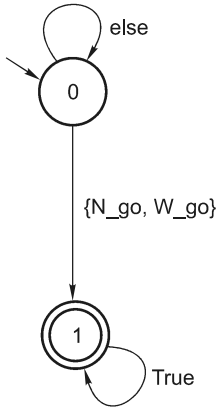


Рис. 4.10. Автомат Бюхи, допускающий все траектории, на которых выполняется формула $\mathbf{F}(N_go \wedge W_go)$

Формальный анализ системы переходов M можно провести до реализации системы управления и ее работы на реальном перекрестке. По синхронной композиции $B_M \otimes B_{\neg\varphi}$ возможность ошибочного перехода можно проверить для любых траекторий модели системы управления перекрестком на этапе ее проектирования.

4.5. Операции над автоматами Бюхи

Оказывается, что Бюхи-распознаваемые ω-языки замкнуты относительно обычных теоретико-множественных операций.

Объединение двух ω-языков L_A и L_B, распознаваемых соответственно автоматами Бюхи A и B, распознает автомат Бюхи $A \cup B$, который является объединением автоматов A и B (объединением всех элементов этих автоматов — множеств состояний, множеств начальных состояний и т. п). Можно считать, что это просто два несвязанных между собой автомата Бюхи, стоящие рядом, и входная цепочка переводит из одного состояния в другое либо один, либо другой автомат.

Пример 4.6

На рис. 4.11 построены автоматы Бюхи A и B, распознающие языки, цепочки которых содержат конечное число вхождений b и бесконечное число вхождений a (L_A), либо наоборот (L_B). Эти два автомата можно считать одним *недетерминированным* автоматом с двумя начальными состояниями, допускающим объединение языков L_A и L_B. Два начальных состояния этих автоматов можно объединить в одно.

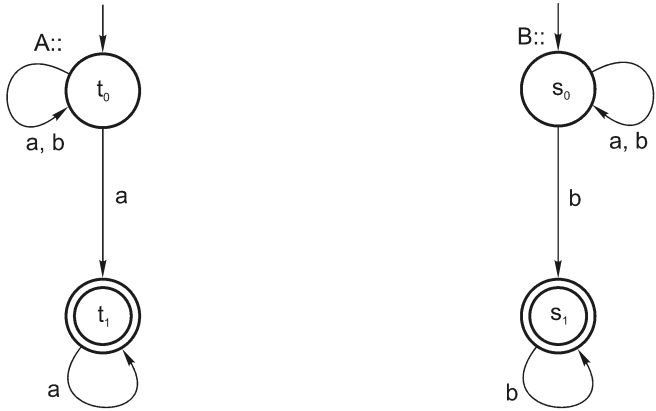


Рис. 4.11. Автоматы Бюхи, допускающие ω-слова с конечным числом вхождений одной буквы и бесконечным числом вхождений другой

Дополнение ω-языка L_A, допускаемого автоматом Бюхи A, это ω-язык $\Sigma^\omega \setminus L_A$. Оказывается, что для любого ω-регулярного языка L ω-язык $\Sigma^\omega \setminus L$ тоже ω-регулярный, т. е. существует автомат Бюхи, допускающий этот ω-язык. Он строится по исходному автомату Бюхи с помощью весьма непростого алгоритма. Результирующий автомат может быть экспоненциально сложнее исходного.

Пересечение двух ω-языков. Автомат Бюхи, распознающий пересечение двух ω-языков L_A и L_B, распознаваемых автоматами Бюхи A и B, строится как синхронная композиция $A \otimes B$ автоматов A и B. Так же, как и для конечных автоматов, синхронная композиция автоматов Бюхи — это фактически два автомата, стоящие рядом и функционирующие синхронно.

Рассмотрим построение синхронной композиции двух автоматов Бюхи более подробно.

Для КОНЕЧНЫХ автоматов принимающим состоянием их синхронной композиции является такая пара состояний компонентных автоматов, в которой оба компонента являются допускающими состояниями соответствующих автоматов (см. пример 4.2 и рис. 4.5). Для автоматов Бюхи ситуация сложнее. ω-слово допускается автоматом Бюхи, если бесконечная последовательность состояний, которые автомат проходит при приеме этого слова, содержит бесконечное число допускающих состояний. Цепочка должна допускаться автоматом $A \otimes B$, если она допускается как автоматом A, так и автоматом B. Но эта цепочка необязательно будет проходить допускающие состояния и в A, и в B *одновременно*. На рис. 4.12 приведен такой пример. Два автомата, A и

B, допускают ω-языки L_A и L_B над словарем $\{a, b\}$: L_A — состоящий из цепочек с бесконечным числом вхождений a и произвольным числом вхождений b, L_B состоит из цепочек с бесконечным числом вхождений b и произвольным числом вхождений a. Синхронная композиция этих автоматов (рис. 4.12, *б*), должна допускать все цепочки с бесконечным числом вхождений как a, так и b. Но ни одна такая цепочка не проходит через допускающие состояния A и B одновременно. Более того, в этом примере в их синхронной композиции такие состояния отсутствуют. Но в то же время, например, цепочка $(ab)^{\omega}$, которая включает бесконечное число вхождений и a, и b, проходит попеременно допускающие состояния автоматов A и B одно за другим, и каждое из них проходит бесконечное число раз. Как определить допустимость цепочки в синхронной композиции автоматов Бюхи?

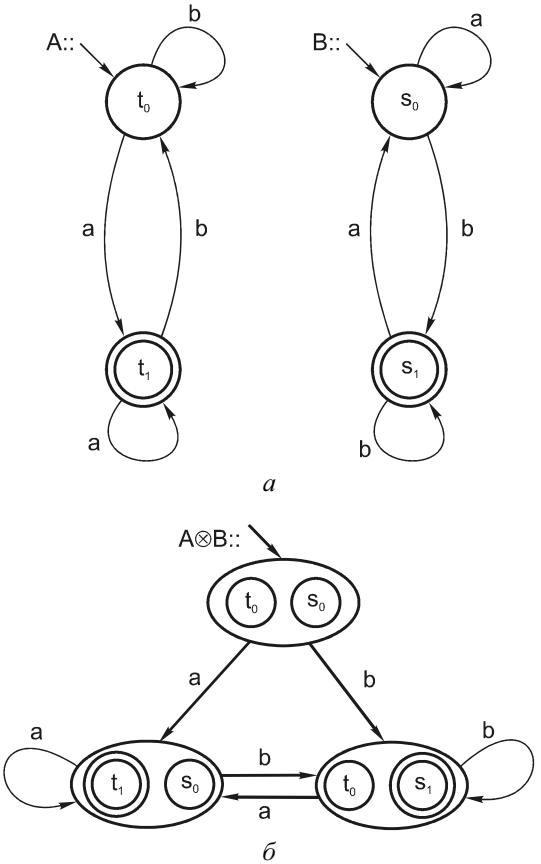


Рис. 4.12. Автоматы Бюхи и их синхронная композиция

Введем понятие *обобщенного автомата Бюхи*.

❖ ***Определение 4.5 (обобщенный автомат Бюхи)***

Этот автомат отличается от обычного автомата Бюхи тем, как в нем определяется множество допускающих состояний. А именно, множество F обобщенного автомата Бюхи состоит из конечного числа подмножеств состояний, $F = \{\mathcal{F}_1, \mathcal{F}_2, ..., \mathcal{F}_k\}$. ω-слово w допускается обобщенным автоматом Бюхи, если цепочка σ состояний, которые автомат проходит при приеме ω-слова w, содержит бесконечное число состояний из каждого множества $\mathcal{F}_i \in F$, т. е. $(\forall \mathcal{F}_i \in F)\inf(\sigma) \cap \mathcal{F}_i \neq \varnothing$.

Для нашего примера на рис. 4.12, *б* множество F обобщенного автомата Бюхи состоит из двух одноэлементных подмножеств: $\mathcal{F}_1 = \{\langle t_1, s_0 \rangle\}$, $\mathcal{F}_2 = \{\langle t_0, s_1 \rangle\}$.

Оказывается, что для любого обобщенного автомата Бюхи существует эквивалентный ему обычный автомат Бюхи. Идея построения такого автомата выражена в следующей теореме.

ТЕОРЕМА 4.2

Для обобщенного автомата Бюхи существует эквивалентный ему обычный автомат Бюхи.

Доказательство

Пусть $A = (S, \Sigma, S_0, \delta, F)$, где $F = \{\mathcal{F}_1, \mathcal{F}_2, ..., \mathcal{F}_k\}$ — обобщенный автомат Бюхи. Обычный автомат Бюхи $A' = (S', \Sigma', S_0', \delta', F')$, эквивалентный обобщенному автомату Бюхи A, строится так (рис. 4.13, *а*):

1. Строится столько копий обобщенного автомата A, сколько подмножеств допускающих состояний содержит F. Состояния каждой копии снабжаются индексами: $S' = S \times \{1, ..., k\}$;
2. $S_0' = S_0 \times \{1\}$ — начальным состоянием нового автомата является начальное состояние обобщенного автомата с индексом 1.
3. В копии i выходные ребра из состояний, принадлежащих допускающему множеству $\mathcal{F}_i$, направляются в состояния следующей по номеру копии, т. е. δ' определяется так:

если $q \in \delta(p,\ a)$, то $\langle q',\ j\rangle \in \delta'(\langle p',\ i\rangle,\ a)$ и:

$j = i$, если $p \notin \mathcal{F}_i$,

$j = (i) \bmod_k + 1$, если $p \in \mathcal{F}_i$.

Таким образом, в переходах автомата A' получается цикл, проходящий через все копии A, и каждый цикл содержит допускающие состояния из каждого допускающего множества $\mathcal{F}_i$.

4. $F' = \mathcal{F}_1 \times \{1\}$ — допускающими состояниями нового обычного автомата Бюхи являются только состояния копии 1 из множества $\mathcal{F}_1$.

Под воздействием входной цепочки w автомат A' проходит допускающие состояния первой копии бесконечно часто тогда и только тогда, когда он проходит и состояния из каждого множества $\mathcal{F}_i$ бесконечно часто в обобщенном автомате Бюхи A. Поэтому цепочка w является допустимой в A' тогда и только тогда, когда она допускается в A.

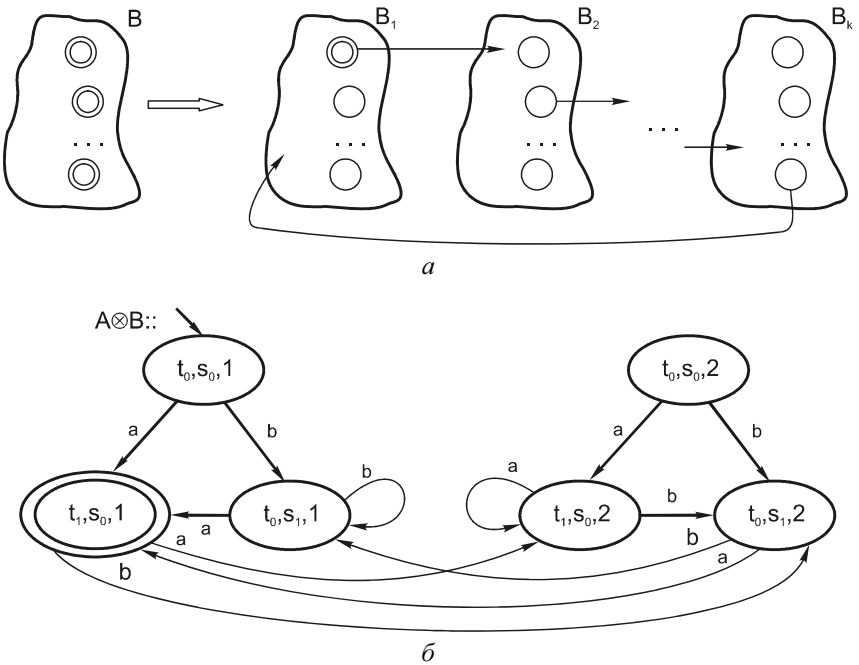
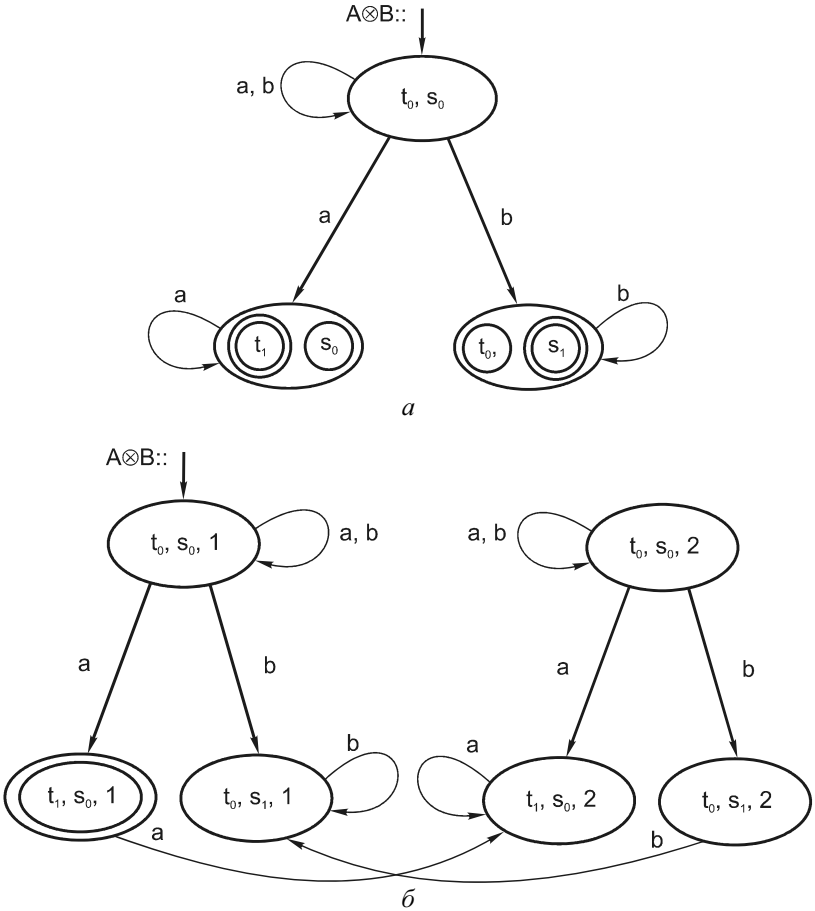


Рис. 4.13. От обобщенного автомата Бюхи к обычному автомату Бюхи

Пример 4.7

По обобщенному автомату Бюхи $A \otimes B$ (см. рис. 4.12, *б*) на рис. 4.13, *б* по вышеприведенному алгоритму построен эквивалентный обычный автомат Бюхи. Можно видеть, что этот автомат Бюхи допускает все возможные цепочки с бесконечным числом вхождений как a, так и b.



а

б

Рис. 4.14. *а*) Обобщенный автомат Бюхи — синхронная композиция автоматов Бюхи рис. 4.11; *б*) эквивалентный автомат Бюхи

Проверку *пустоты языка*, допускаемого автоматом Бюхи, можно осуществить с помощью простого алгоритма: автомат Бюхи A допускает пустой язык, если и только если в A *не существует* ни одной сильно связной компоненты, достижимой из начального состояния и включающей хотя бы одно допускающее состояние. Иными словами, *если в автомате Бюхи есть цикл*

с допускающим состоянием, достижимый из начального состояния, то автомат допускает хотя бы одно ω *-слово.*

Пример 4.8

В автомате Бюхи на рис. 4.13, *б* из начального состояния $\langle t_0, s_0, 1\rangle$ достижим цикл, включающий допускающее состояние $\langle t_1, s_0, 1\rangle$. Следовательно, этот автомат допускает ω-слова (т. е. непустой ω-язык).

Пример 4.9

Автоматы Бюхи A и B с входным алфавитом $\{a, b\}$, допускающие цепочки с конечным числом вхождений одной буквы и бесконечным числом вхождений другой, представлены на рис. 4.11. На рис. 4.14, *а* построена их синхронная композиция — обобщенный автомат Бюхи с двумя множествами допускающих состояний $F = \{\mathcal{F}_1, \mathcal{F}_2\}$, где $\mathcal{F}_1 = \{\langle t_1, s_0\rangle\}$, $\mathcal{F}_2 = \{\langle t_0, s_1\rangle\}$. По этому обобщенному автомату Бюхи легко строится обычный автомат Бюхи, показанный на рис. 4.14, *б*. В этом автомате нет цикла, включающего единственное допускающее состояние $\langle t_1, s_0, 1\rangle$. Следовательно, язык, допускаемый этим автоматом, пуст. Действительно, пересечение языков L_A и L_B включает только цепочки с конечным числом вхождений символов a и символов b, а все эти цепочки конечные, они никаким автоматом Бюхи не допускаются.

Еще раз подчеркнем, что хотя и автомат Бюхи, и обобщенный автомат Бюхи используются для задания и анализа бесконечных языков, это конечные модели с конечным числом состояний и переходов. Они отличаются от обычных конечных автоматов только правилами, определяющими допускаемые цепочки среди множества всех возможных бесконечных входных цепочек.

4.6. Автоматы Бюхи и LTL-формулы

Конечные автоматы можно считать формальными системами, задающими языки, т. е. некоторое подмножество (обычно бесконечное) конечных слов в его входном алфавите. Например, автомат A на рис. 4.7, рассматриваемый как конечный автомат, допускает (и, следовательно, задает) все цепочки из a и b, оканчивающиеся на b. Все такие цепочки определяются регулярным выражением $(a+b)^* b$. Каждый автомат Бюхи также допускает (и, следовательно, задает) некоторое множество бесконечных цепочек, составленных из символов его входного алфавита Σ, которые определяют его циклическое прохождение через одно из допускающих состояний. Иными словами, автомат Бюхи задает ω-язык.

Формулы LTL интерпретируются на вычислениях структуры Крипке. Каждое вычисление структуры Крипке — это бесконечная последовательность состояний. Ясно, однако, что не названия состояний характеризуют вычисление. Вычисление характеризуется цепочкой подмножеств тех атомарных предикатов структуры Крипке, которые истинны в соответствующих состояниях, т. е. траекторией. Каждая траектория структуры Крипке — это бесконечное слово в алфавите 2^{AP}. Например, если $AP = \{q, r\}$, то $2^{AP} = \{\varnothing, \{r\}, \{q\}, \{q, r\}\}$.

Если $\sigma = s_0 s_1 s_2 s_3 \ldots$ — вычисление структуры Крипке, то, в соответствии с *определением 2.10*, $L(\sigma)$ будем обозначать ω-слово $L(s_0)L(s_1)L(s_2)L(s_3)\ldots$, индуцированное вычислением σ. Например, траектория, соответствующая вычислению, представленному на рис. 2.8, — это бесконечная цепочка $\{p, q\}\{q, r\}\{r\}\{r\}\{r\} \ldots$. Все траектории структуры Крипке рис. 4.1, *а* — это бесконечное множество ω-слов (т. е. ω-язык), примерами которых являются:

$\{p, q\}\{q, r\}\{r\}\{\ \}\ldots;$
$\{p, q\}\{\ \}\{q, r\}\{r\}\{\ \}\{q, r\}\ldots;$
$\{p, q\}\{q, r\}\{r\}\{r\}\ldots;$

ω-слово π удовлетворяет LTL-формуле φ, если φ принимает истинное значение на π (обозначается $\pi \models \varphi$). Например, ω-слово $\pi = \{p\}\{\ \}\{p\}\{q\}\{\ \}^{\omega}$ удовлетворяет формуле $p \wedge (\neg q \mathbf{U} q)$. Любую формулу φ линейной темпоральной логики можно считать описанием (спецификацией) ω-языка, все цепочки которого удовлетворяют этой формуле.

Зададимся вопросом: можно ли конечным образом с помощью автомата Бюхи задать все те траектории вычисления, которые описываются заданной LTL-формулой φ?

Во-первых, рассмотрим, какой алфавит должен быть у такого автомата Бюхи. Пусть, например, формула φ есть $\mathbf{F}p$, т. е. эта формула логики LTL описывает такие бесконечные траектории, в которых когда-нибудь в будущем встретится p. Автомат Бюхи, допускающий все такие траектории, не может иметь в своем алфавите только символ p — ведь автомат Бюхи должен на *каждом* шаге функционирования принимать какой-то входной символ. Значит, кроме символа p во входном алфавите Σ автомата Бюхи должно быть еще что-то (например, символ, показывающий отсутствие символа p. Очевидно, что входным алфавитом Σ автомата Бюхи, допускающего ω-слова,

удовлетворяющие формуле φ LTL с множеством атомарных предикатов AP, является 2^{AP}. Для формулы Fp такой автомат будет иметь входной алфавит $\{\varnothing, \{p\}\}$, или, что то же, $\{\{\}, \{p\}\}$.

На рис. 4.10 контрольный автомат Бюхи — watchdog, отслеживающий корректность системы управления перекрестком, в качестве одного из символов входного языка имеет множество $\{N_go, W_go\}$ двух атомарных предикатов, которые не должны быть истинны одновременно ни в каком состоянии системы управления перекрестком.

Автоматы Бюхи, допускающие ω-языки, заданные некоторыми LTL-формулами, приведены на рис. 4.15.

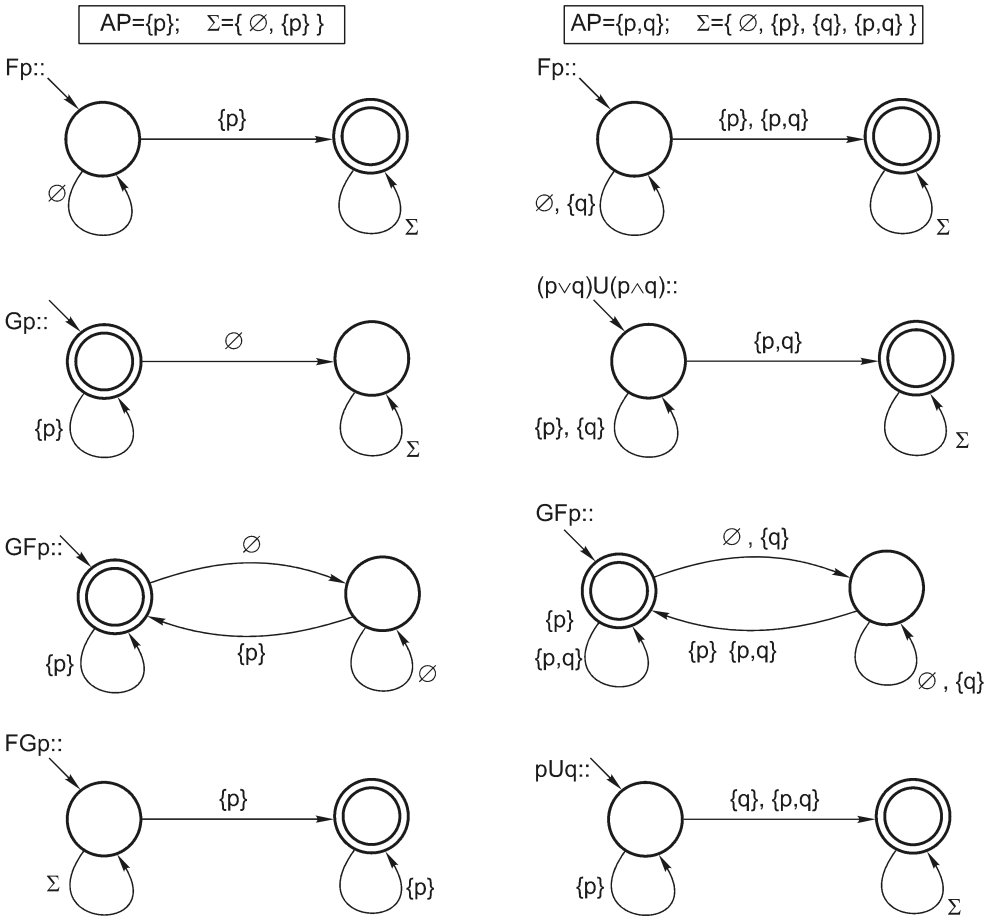


Рис. 4.15. Автоматы Бюхи для некоторых формул LTL

Существует важный результат, впервые полученный в 1959 г. [16] и впоследствии использованный для верификации в [146].

ТЕОРЕМА 4.3

Для любой LTL-формулы φ существует автомат Бюхи, допускающий все ω-слова, на которых выполняется φ, и только их.

Известно, что для некоторых автоматов Бюхи не существует LTL-формулы, которая задает все ω-слова, распознаваемые этим автоматом. Иными словами, автоматы Бюхи более выразительны, чем LTL-формулы.

ТЕОРЕМА 4.4

Существуют автоматы Бюхи, ω-языки которых не могут быть заданы никакой LTL-формулой.

В качестве примера приведем вычисления, удовлетворяющие требованию: "*p истинно на каждом четном шаге*". Они описываются ω-регулярным выражением $\left(\left(\{p\}+\varnothing\right)\{p\}\right)^{\omega}$ и определяются автоматом Бюхи (рис. 4.16). Оказывается, что LTL-формулой это свойство выразить нельзя.

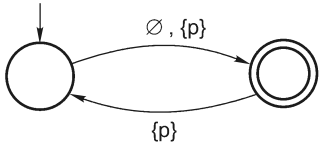


Рис. 4.16. Автомат Бюхи, допускающий траектории, в которых на каждом четном шаге выполняется *p*

4.7. Структуры Крипке и автоматы Бюхи

Структура Крипке также задает траектории — множество бесконечных цепочек над алфавитом 2^{AP}, т. е. символами цепочек являются множества атомарных предикатов. По заданной структуре Крипке M автомат Бюхи B_M, допускающий все траектории M, строится по следующему алгоритму:

- каждый переход в B_M, ведущий из состояния *s*, помечается подмножеством атомарных предикатов, истинных в *s*, и
- каждое состояние B_M является допускающим — все траектории в этом автомате Бюхи являются допускаемыми ω-словами.

Формально, пусть структура Крипке M задана так: $M = (S_M, S_{0M}, R, AP, L)$. Соответствующий автомат Бюхи $B_M = (S_B, \Sigma, S_{0B}, \delta, F)$, который допускает любые траектории структуры Крипке M, определим следующим образом:

$S_B = S_M$ — состояния M не изменяются;

$S_{0B} = S_{0M}$ — множество начальных состояний M не изменяется;

$\Sigma = 2^{AP}$ — алфавит автомата B_M составляют все подмножества атомарных предикатов M;

$F = S_M$ — все состояния автомата B_M являются допускающими;

если $(s, s') \in R$, *то* $(s, L(s), s') \in \delta$ — все переходы, ведущие из состояния s, помечаются множеством $L(s)$ атомарных предикатов, истинных в этом состоянии.

На рис. 4.17 по структуре Крипке M построен автомат Бюхи, допускающий все те траектории, которые задаются этой структурой Крипке. Очевидно, что все траектории структуры Крипке, и только они составляют ω-язык построенного так автомата Бюхи.

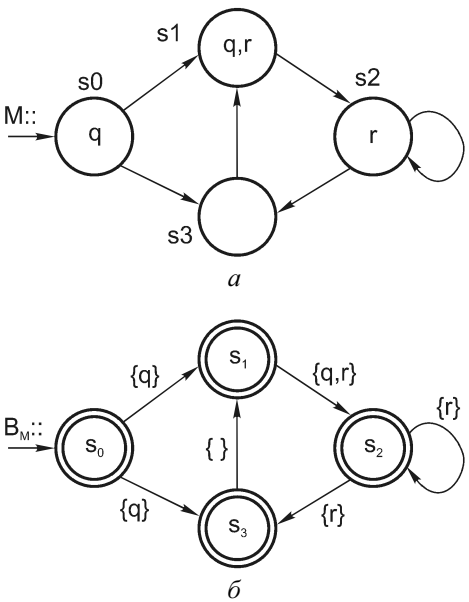


Рис. 4.17. Автомат Бюхи B_M, построенный по структуре Крипке M

4.8. Проверка модели для формул LTL

Все шаги алгоритма проверки модели для формул LTL (см. рис. 4.6) теперь ясны: по заданной структуре Крипке M строится автомат Бюхи B_M, допускающий все траектории, которые возможны в M; по LTL-формуле φ строится "контрольный" автомат Бюхи $B_{\neg\varphi}$, допускающий все ω-слова, на которых выполняется $\neg\varphi$; затем строится синхронная композиция автоматов $B_M \otimes B_{\neg\varphi}$, и этот автомат проверяется на наличие у него цикла, достижимого из начального состояния и включающего допускающее состояние. Если такой цикл существует, он выдается как контрпример — вычисление, не удовлетворяющее формуле φ.

Отметим, что построение синхронной композиции автоматов здесь упрощается по сравнению с общим случаем: у автомата Бюхи, построенного по структуре Крипке, все состояния допускающие. Таким образом, в этом частном случае в синхронной композиции автоматов Бюхи $B_M \otimes B_{\neg\varphi}$ те состояния будут допускающими, у которых допускающим является второй компонент.

Пример 4.10

Проверим, выполняется ли свойство $\varphi = \mathbf{GF}(r \Rightarrow q)$ на структуре Крипке M (см. рис. 4.17, *а*). Очевидно,

$$\neg\varphi = \neg(\mathbf{GF}(r \Rightarrow q)) = \mathbf{FG}(\neg(r \Rightarrow q)) = \mathbf{FG}(r \wedge \neg q).$$

Автомат Бюхи B_M, построенный по структуре Крипке M, представлен на рис. 4.17, *б*. На рис. 4.18, *а* этот автомат Бюхи для удобства представлен повторно, а на рис. 4.18, *б* представлен автомат Бюхи $B_{\neg\varphi}$, допускающий все вычисления, удовлетворяющие формуле $\neg\varphi = \mathbf{FG}(r \wedge \neg q)$. Синхронная композиция этих двух автоматов представлена на рис. 4.18, *в*. Найденный контрпример выделен на рис. 4.18, *в*. Он показывает одно из вычислений, на которых исходная формула $\varphi = \mathbf{GF}(r \Rightarrow q)$ не выполняется для исходной структуры Крипке. Это вычисление $s_0 s_1 s_2 s_2 s_2 \ldots$. Действительно, на этом вычислении не будет бесконечно часто повторяться формула $r \Rightarrow q$ (именно это требование определено исходной формулой φ).

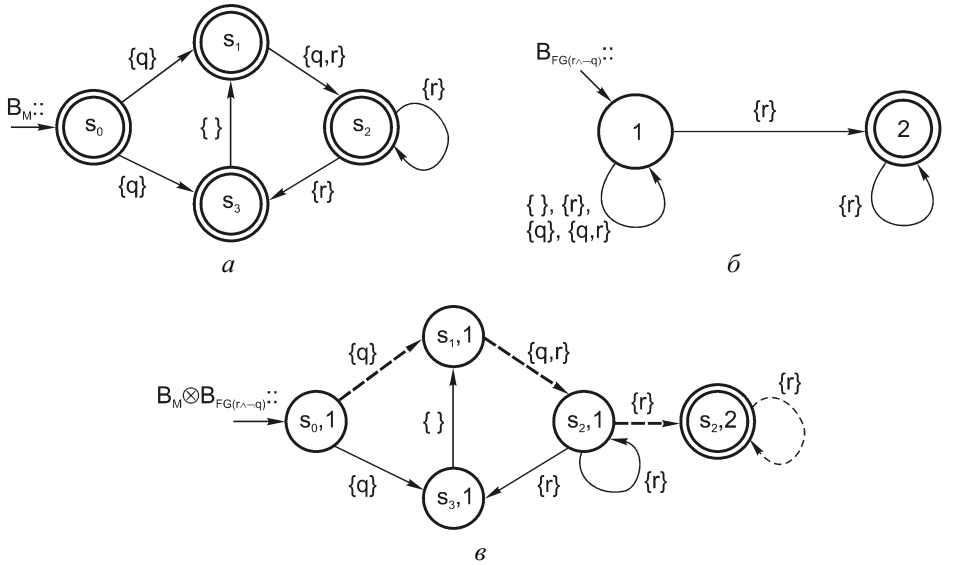


Рис. 4.18. В структуре Крипке найден контрпример: вычисление $s_0 s_1 s_2 s_2 \ldots$, на котором формула $\mathbf{GF}(r \Rightarrow q)$ не выполняется

4.9. Построение автомата Бюхи по формуле LTL

В этом разделе мы рассмотрим один оставшийся нераскрытым этап теоретико-автоматного подхода к проверке выполнения формулы LTL на структуре Крипке: построение автомата Бюхи по формуле LTL.

Существует несколько алгоритмов выполнения этого этапа, и все они весьма сложные. В общем случае все они дают в результате автомат Бюхи с экспоненциальным числом состояний относительно длины формулы φ. Здесь мы рассмотрим один из алгоритмов, который является концептуально простым, хотя он часто в результате дает не минимальный автомат Бюхи.

Формулы LTL построены над множеством атомарных предикатов AP, логических связок и темпоральных операторов. Любая формула LTL может быть выражена с помощью базиса LTL, т. е. через атомарные предикаты, булевы операции $\neg$ и $\vee$, а также темпоральные операторы $\mathbf{X}$ и $\mathbf{U}$. Все такие формулы задаются грамматикой:

$$\varphi ::= p | \neg\varphi | \varphi \vee \varphi | \mathbf{X}\varphi | \varphi \mathbf{U} \varphi .$$

Поставим следующую задачу. По заданной LTL-формуле, построенной над множеством AP атомарных предикатов, построить автомат Бюхи с входным алфавитом 2^{AP}, который допускает в точности все ω-слова, удовлетворяющие формуле φ, и только их.

Пример 4.11

Рассмотрим формулу $\varphi = (p \vee q)\mathbf{U}(p \wedge q)$. Эта формула LTL построена над алфавитом $AP = \{p, q\}$ и определяет все те вычисления, на которых когда-то в будущем встретятся и p, и q, а до этого в состояниях вычисления будут истинны или p, или q. Формула φ определена над последовательностями, построенными из символов алфавита 2^{AP}, т. е. алфавит искомого автомата Бюхи есть $\Sigma = \{\varnothing, \{p\}, \{q\}, \{p, q\}\}$. Формулы LTL интерпретируются на вычислениях — бесконечных цепочках состояний структуры Крипке (например, на цепочке $\sigma = s_0 s_1 s_2$ ... рис. 4.19, *а*), в каждом состоянии которых определены множества истинных в этом состоянии атомарных предикатов из множества AP. Последовательность подмножеств атомарных предикатов, соответствующая вычислению σ, определяется функцией пометок L. Вычислению σ на рис. 4.19, *а* соответствует цепочка множеств атомарных предикатов:

$L(\sigma) = \{p\}\{q\}\{p,q\}\{q\}\{\}\{p\}\ldots$.

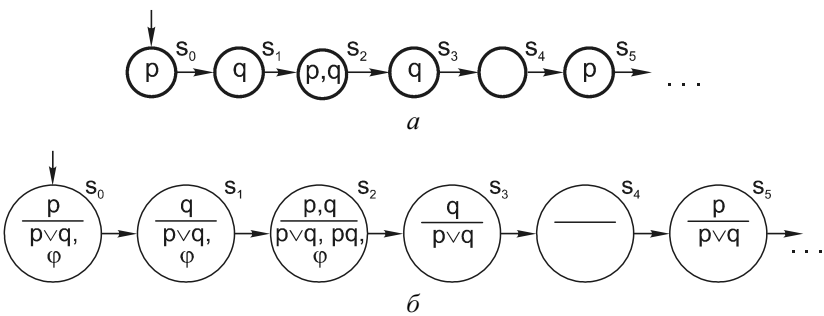


Рис. 4.19. Вычисление, на котором выполняется формула $\varphi = (p \vee q)\mathbf{U}(p \wedge q)$

Разметка состояний вычисления формулами LTL

Описываемый в этом разделе метод построения автомата Бюхи по заданной LTL-формуле основан на простой идее, близкой идее разметки состояний структуры Крипке при проверке выполнения CTL-формул. Эта идея была изложена в [145].

Следуя этой идее, рассмотрим сначала наивный метод определения того, выполняется ли формула LTL на заданном бесконечном вычислении. Например, проверим, будет ли формула $\varphi = (p \vee q)\mathbf{U}(p \wedge q)$ выполняться на вычислении, представленном на рис. 4.19, *а*.

Пометим каждое состояние вычисления $s_0 s_1 s_2$ ... рис. 4.19, *а* теми подформулами формулы φ, которые выполняются в этом состоянии. Начиная с атомарных предикатов, истинных в каждом состоянии, будем добавлять в состояние все те более сложные подформулы формулы φ, которые истинны в этом состоянии.

Очевидно, что истинность формул из атомарных предикатов, связанных любыми логическими операциями, может быть определена непосредственно в состояниях. Например, подформула $(p \vee q)$ истинна в состоянии s_0 (потому что в нем истинен атомарный предикат p), и в состоянии s_1 (в нем истинен атомарный предикат q), но ложен в состоянии s_4 (в нем ложны оба атомарных предиката, p и q).

Выполнимость подформул с темпоральными операторами определить сложнее: для этого требуется анализ подформул в последующих состояниях конкретного вычисления. Например, на вычислении $\sigma = s_0 s_1 s_2 ...$ рис. 4.19, *а* в состоянии s_1 формула $\varphi = (p \vee q)\mathbf{U}(p \wedge q)$ выполняется, а в состоянии s_3, также помеченном атомарным предикатом q, эта формула не выполняется. Обе такие разметки состояний допустимы, т. е. состояние, помеченное атомарным предикатом q, может быть при такой разметке дополнительно помечено формулой $(p \vee q)\mathbf{U}(p \wedge q)$, а может быть и не помечено этой формулой.

Для всех состояний вычисления σ такая разметка дана на рис. 4.19, *б*. Видно, что на вычислении σ формула $\varphi = (p \vee q)\mathbf{U}(p \wedge q)$ выполняется, поскольку эта формула помечает начальное состояние вычисления.

В общем случае проверить выполнимость произвольных формул LTL этим способом невозможно, поскольку вычисления бесконечны. Однако этот наивный метод разметки дает идею того, как строить автомат Бюхи, допускающий все вычисления, удовлетворяющие формуле LTL. Состояниями такого автомата будут все возможные "согласованные" разметки состояний вычислений, допускающих формулу φ, а переходы определятся по правилам возможного соседства состояний в таких вычислениях.

Начнем с формального определения тех подформул, которые *могут* помечать состояния искомого автомата Бюхи.

Множество всех возможных подформул формулы φ определяет так называемое "*замыкание*" формулы φ (обозначается $cl(\varphi)$, от слова *closure*). Фактически, $cl(\varphi)$ — это множество всех подформул формулы φ, которые могут появиться в синтаксическом дереве при анализе формулы φ. Будем считать, что для каждой формулы из $cl(\varphi)$ ее отрицание неявно входит в $cl(\varphi)$. Для формулы $(p \vee q)\mathbf{U}(p \wedge q)$ таких подформул пять:

$$cl\big((p \vee q)\mathbf{U}(p \wedge q)\big) = \{\, p,\ q,\ p \vee q,\ p \wedge q,\ (p \vee q)\mathbf{U}(p \wedge q) \,\}.$$

Конечно, не все комбинации подформул анализируемой формулы φ могут помечать одно и то же состояние любого вычисления. Подформулы, которые могут помечать состояние, должны быть непротиворечивыми, согласованными, а их множество, помечающее состояние, должно быть максимальным. Например, формула и ее отрицание не могут одновременно помечать состояние. Далее, рассмотрим рис. 4.19, *б*. Состояние s_5 помечено формулами $\{p,\ p \vee q\}$, и очевидно, что оно не может быть помечено, например, формулой $p \wedge q$, поскольку в s_0 не выполняется q. Далее, состояние s_4 помечено пустым множеством формул. В этом состоянии p и q ложны, здесь не могут встретиться ни $p \wedge q$, ни $p \vee q$, ни сама анализируемая формула φ: в этом состоянии ни одна из подформул формулы φ не выполняется (и, следовательно, выполняются отрицания их всех). Состояние s_2 помечено формулами $\{p,\ q,\ p \vee q,\ p \wedge q,\ (p \vee q)\mathbf{U}(p \wedge q)\}$ — все эти подформулы выполняются в этом состоянии вычисления рис. 4.19, *а*, а их отрицания не выполняются.

Атомы темпоральной формулы логики LTL

Легко определить формальные правила построения всех максимальных согласованных множеств подформул анализируемой формулы φ, которыми *могут* быть помечены состояния таких вычислений, *на которых выполняется* формула φ. Назовем такие множества *атомами*.

> ❖ ***Определение 4.6 (атомы темпоральной формулы логики LTL)***
>
> Множество подформул произвольной темпоральной формулы φ логики LTL назовем согласованным, если существует вычисление, на котором все они могут выполниться. Любое максимальное множество согласованных подформул формулы φ будем называть атомом этой формулы.

Атомы формулы φ можно понимать как максимальные множества LTL-формул из $cl(\varphi)$, которые могут помечать состояния вычислений. Атомы можно считать состояниями искомого автомата Бюхи.

Атомы формулы φ строятся по следующим очевидным правилам, вытекающим из семантики LTL. Для любых $\psi, \varphi_1, \varphi_2, \varphi_1 \vee \varphi_2, \varphi_1 \mathbf{U} \varphi_2 \in cl(\varphi)$ и любого атома A должны выполняться правила:

- R1. $\psi \in A \Leftrightarrow \neg\psi \notin A$;
- R2. $\varphi_1 \vee \varphi_2 \in A \Leftrightarrow \varphi_1 \in A$ или $\varphi_2 \in A$;
- R3. $\varphi_1 \mathbf{U} \varphi_2 \in A$ и $\varphi_2 \notin A \Rightarrow \varphi_1 \in A$;
- R4. $\varphi_2 \in A \Rightarrow \varphi_1 \mathbf{U} \varphi_2 \in A$.

Эти правила можно легко дополнить для выводимых логических операций и темпоральных операторов. Например, для любых $\psi, \varphi_1, \varphi_2, \varphi_1 \wedge \varphi_2, \varphi_1 \Rightarrow \Rightarrow \varphi_2, \mathbf{F}\psi, \mathbf{G}\psi \in cl(\varphi)$ и любого атома A должны выполняться правила:

- R5. $\varphi_1 \wedge \varphi_2 \in A \Leftrightarrow \varphi_1 \in A$ и $\varphi_2 \in A$;
- R6. $\varphi_1 \Rightarrow \varphi_2 \in A \Leftrightarrow \varphi_1 \notin A$ или $\varphi_2 \in A$;
- R7. $\psi \in A \Rightarrow \mathbf{F}\psi \in A$;
- R8. $\mathbf{G}\psi \in A \Rightarrow \psi \in A$.

Правила R5 и R6 очевидны. Справедливость R7 вытекает из R4, поскольку $\mathbf{F}\psi = true \mathbf{U} \psi$. Правило R8, в свою очередь, вытекает из R1 и R7. Действительно, из правила $\psi \in A \Rightarrow \mathbf{F}\psi \in A$ следует истинность его контрапозиции $\neg(\mathbf{F}\psi \in A) \Rightarrow \neg(\psi \in A)$, откуда по правилу R1 $(\neg\mathbf{F}\psi) \in A \Rightarrow (\neg\psi) \in A$ или, что то же, $(\neg\mathbf{F}\neg\psi) \in A \Rightarrow (\neg(\neg\psi)) \in A$. Правило R8 теперь вытекает из определения $\mathbf{G}\psi = \neg\mathbf{F}\neg\psi$.

В соответствии с R1, атомы включают либо саму формулу, либо ее отрицание. Для простоты отрицание формулы мы не будем указывать в атоме, т. е. если формула $\psi \in cl(\varphi)$ не входит в атом, то это означает, что в этот атом входит отрицание формулы ψ.

Правила R1—R8 определяются очевидными правилами согласования двух формул $\psi_1, \psi_2 \in cl(\varphi)$ для одного и того же произвольного состояния s любого вычисления: *если некоторая формула* ψ_1 *выполняется на вычислении, начинающемся с* s, *то должна ли на этом же вычислении выполняться*

формула ψ_2? Эти правила можно назвать *локальными ограничениями* на выполнимость формул LTL.

Пример 4.12

Построим атомы нескольких простых темпоральных формул.

1. Построим все атомы формулы $\mathbf{F}p$. Замыкание этой формулы содержит две формулы, $cl(\mathbf{F}p) = \{p, \mathbf{F}p\}$. Каждый атом содержит либо формулы, либо их отрицания. Всего возможны четыре таких комбинации: $\{\neg p, \neg\mathbf{F}p\}, \{\neg p, \mathbf{F}p\}, \{p, \neg\mathbf{F}p\}, \{p, \mathbf{F}p\}$, но некоторые из них не могут являться атомами, потому что они не согласованы. Обозначим вхождение этих двух формул двоичными переменными, $x = p \in A$, $y = \mathbf{F}p \in A$. Истинность переменной будет говорить о вхождении формулы в атом, ложность — о ее невхождении в атом. Локальные ограничения R1—R8 определяют, какие из этих комбинаций могут быть атомами. К нашей формуле имеет отношение только правило R7: $p \in A \Rightarrow \mathbf{F}p \in A$, т. е. в атомах должно соблюдаться соотношение $x \Rightarrow y$. Функция $x \Rightarrow y$ истинна на трех наборах переменных x, y: $(0, 0), (0, 1), (1, 1)$. Поэтому формула $\mathbf{F}p$ имеет три атома:

   $A_1 = \varnothing;$

   $A_2 = \{\mathbf{F}p\};$

   $A_3 = \{p, \mathbf{F}p\}.$

   При явном включении отрицаний формул эти атомы запишутся так:

   $A_1 = \{\neg p, \neg\mathbf{F}p\};$

   $A_2 = \{\neg p, \mathbf{F}p\};$

   $A_3 = \{p, \mathbf{F}p\}.$

   Среди атомов здесь отсутствует подмножество множества $cl(\mathbf{F}p)$, состоящее из одной формулы p. Для $\varphi = \mathbf{F}p$ множество формул $\{p\}$ является несогласованным, поскольку в атом не могут одновременно входить и p, и $\neg\mathbf{F}p$. Именно это и устанавливает правило R7.

2. Формула $\mathbf{GF}p$. Замыкание формулы $\mathbf{GF}p$ содержит три подформулы: $\{p, \mathbf{F}p, \mathbf{GF}p\}$. Введем три двоичных переменные: $x = p \in A$, $y = \mathbf{F}p \in A$, $z = \mathbf{GF}p \in A$. Правила вхождения формул в атомы определяются здесь формулой $(x \Rightarrow y) \wedge (z \Rightarrow y)$, построенной в соответствии с

правилами R7, R8. Эта формула истинна на пяти наборах значений переменных x, y и z: $(0,0,0)$, $(0,1,0)$, $(1,1,0)$, $(0,1,1)$ и $(1,1,1)$. Поэтому у формулы $\mathbf{GF}p$ пять атомов:

$A_1 = \varnothing;$
$A_2 = \{\mathbf{F}p\};$
$A_3 = \{p, \mathbf{F}p\};$
$A_4 = \{\mathbf{F}p, \mathbf{GF}p\};$
$A_5 = \{p, \mathbf{F}p, \mathbf{GF}p\}.$

3. Для формулы $\varphi = (p \vee q)\mathbf{U}(p \wedge q)$ все множество ее атомов встречается в состояниях вычисления, показанных на рис. 4.19, *б*:

$A_1 = \varnothing;$
$A_2 = \{p, p \vee q\};$
$A_3 = \{p, p \vee q, (p \vee q)\mathbf{U}(p \wedge q)\};$
$A_4 = \{q, p \vee q\};$
$A_5 = \{q, p \vee q, (p \vee q)\mathbf{U}(p \wedge q)\};$
$A_6 = \{p, q, p \vee q, p \wedge q, (p \vee q)\mathbf{U}(p \wedge q)\}.$

4. У формулы $p\mathbf{U}q$ пять атомов:

$A_1 = \varnothing;$
$A_2 = \{p\};$
$A_3 = \{p, p\mathbf{U}q\};$
$A_4 = \{q, p\mathbf{U}q\};$
$A_5 = \{p, q, p\mathbf{U}q\}.$

5. Формула $\mathbf{X}p$ имеет четыре атома — все подмножества $cl(\mathbf{X}p) = \{p, \mathbf{X}p\}$ являются атомами этой формулы:

$A_1 = \varnothing;$
$A_2 = \{p\};$
$A_3 = \{p, \mathbf{X}p\};$
$A_4 = \{\mathbf{X}p\}.$

6. Формула $\mathbf{FG}a$ имеет три подформулы: $\{a, \mathbf{G}a, \mathbf{FG}a\}$ и 5 атомов:

$A_1 = \varnothing;$

$A_2 = \{a\};$

$A_3 = \{a, \mathbf{FG}a\};$

$A_4 = \{\mathbf{FG}a\};$

$A_5 = \{a, \mathbf{G}a, \mathbf{FG}a\}.$

При построении атомов учтены правила R7 и R8.

7. Рассмотрим более сложный пример: формулу $\varphi = (a \vee b)\mathbf{UG}a$. Эта формула определяет все бесконечные цепочки, в которых число символов b конечно, а число символов a бесконечно. Замыкание этой формулы включает пять подформул: $cl((a \vee b)\mathbf{UG}a) = \{a, b, a \vee b, \mathbf{G}a, (a \vee b)\mathbf{UG}a\}$. Всего подмножеств этих подформул 32, но атомов среди них много меньше, поскольку они должны удовлетворять правилам R1—R8. В нашем случае при построении атомов нужно принять во внимание следующие правила:

R2. $a \vee b \in A \Leftrightarrow a \in A$ или $b \in A$;

R3. $(a \vee b)\mathbf{UG}a \in A$ и $\mathbf{G}a \notin A \Rightarrow (a \vee b) \in A$;

R4. $\mathbf{G}a \in A \Rightarrow (a \vee b)\mathbf{UG}a \in A$;

R8. $\mathbf{G}a \in A \Rightarrow a \in A$.

У этой формулы 9 атомов:

$A_1 = \varnothing;$

$A_2 = \{a, a \vee b\};$

$A_3 = \{b, a \vee b\};$

$A_4 = \{a, b, a \vee b\};$

$A_5 = \{a, a \vee b, (a \vee b)\mathbf{UG}a\};$

$A_6 = \{b, a \vee b, (a \vee b)\mathbf{UG}a\};$

$A_7 = \{a, b, a \vee b, (a \vee b)\mathbf{UG}a\};$

$A_8 = \{a, a \vee b, \mathbf{G}a, (a \vee b)\mathbf{UG}a\};$

$A_9 = \{a, b, a \vee b, \mathbf{G}a, (a \vee b)\mathbf{UG}a\}.$

Переходы между состояниями искомого автомата Бюхи

Правила, определяющие ограничения на переходах между атомами искомого автомата Бюхи, также вытекают из семантики LTL. Это правила выполнимости формул в *соседних* состояниях любого вычисления $\sigma = s_0 s_1 s_2 s_3 \ldots$. Эти правила можно назвать правилами *реализации обязательств* выполнения в будущем тех утверждений, которые выражены набором формул в каждом помеченном состоянии (атоме).

Все возможные подформулы логики LTL, которые могут встретиться в атомах, задаются грамматикой:

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

Рассмотрим обязательства, которые определяют эти формулы. Подформулы, состоящие из атомарных предикатов, не имеют обязательств их выполнения в будущем: они выполняются непосредственно в текущем состоянии. Обязательства выполнения утверждений в будущем выражаются двумя операторами, $\mathbf{X}$ и $\mathbf{U}$. Формула $\mathbf{X}\psi$, встретившаяся в текущем состоянии любого вычисления, "обязуется" выполнить формулу ψ в следующем состоянии. Формула $\varphi_1 \mathbf{U} \varphi_2$, встретившаяся в текущем состоянии любого вычисления, "обязуется", что формула φ_1 будет выполняться во всех последующих состояниях этого вычисления до тех пор, пока не встретится состояние, в котором будет выполнена формула φ_2.

Формализуем эти правила. Пусть A и A' — атомы, являющиеся соседними состояниями какого-нибудь вычисления, так что из состояния A есть переход в состояние A'. Тогда правила *реализации обязательств* имеют вид:

Для любых $\mathbf{X}\psi, \varphi_1 \mathbf{U} \varphi_2 \in cl(\varphi)$:

C1. $\mathbf{X}\psi \in A \Leftrightarrow \psi \in A'$

C2. $\varphi_1 \mathbf{U} \varphi_2 \in A \Leftrightarrow \varphi_2 \in A$ *или же* $\varphi_1 \in A$ *и* $\varphi_1 \mathbf{U} \varphi_2 \in A'$

Правило C1 вытекает из семантики формулы $\mathbf{X}\psi$: $\sigma^i \models \mathbf{X}\psi \Rightarrow \sigma^{i+1} \models \psi$. Если атом A включает формулу $\mathbf{X}\psi$, то он может иметь переход только в атом, включающий ψ. Это правило действует и в другую сторону: если в замыкание формулы φ входит $\mathbf{X}\psi$, то в атом, включающий ψ, может быть переход только из атома, в котором было обязательство $\mathbf{X}\psi$ выполнения ψ в следующем состоянии.

Правило C2 выражает рекурсивное определение семантики оператора Until:

$$\varphi_1 \mathbf{U} \varphi_2 \equiv \varphi_2 \vee \varphi_1 \wedge \mathbf{X}\left(\varphi_1 \mathbf{U} \varphi_2\right).$$

Это правило говорит о том, что если некоторый атом включает формулу $\varphi_1 \mathbf{U} \varphi_2$, то или он включает и φ_2, или он включает φ_1, а последующий атом включает формулу $\varphi_1 \mathbf{U} \varphi_2$. Это правило действует и в другую сторону: в атом, включающий $\varphi_1 \mathbf{U} \varphi_2$, может быть переход из атома, включающего φ_1, только в том случае, если он кроме φ_1 включает также и $\varphi_1 \mathbf{U} \varphi_2$. Из правила C2 вытекает, например, что не может быть переходов между атомами $\{p,\ p\mathbf{U}q\}$ и $\{p, r\}$.

Построим аналогичные правила для выводимых операторов **F** и **G**, воспользовавшись их рекурсивными определениями:

$$\mathbf{F}\psi \equiv \psi \vee \mathbf{XF}\psi,$$

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

Эти определения интерпретируются следующим образом. Для оператора **F**: если в состоянии s выполняется формула $\mathbf{F}\psi$, то это обязательство выполнения формулы ψ должно сохраняться во всей цепочке состояний вычислений, начинающейся с s до тех пор, пока не встретится состояние, в котором ψ действительно выполнится. Для оператора **G**: в состоянии s, в котором $\mathbf{G}\psi$ выполняется, должна выполняться также и формула ψ, и в любых последующих состояниях вычислений, начинающихся с s, также должна выполняться формула $\mathbf{G}\psi$.

Из этих определений легко выводятся два дополнительных правила реализации обязательств, которые должны выполнятся для любых атомов A и A', если из A есть переход в A':

C3. $\mathbf{F}\psi \in A \Leftrightarrow \psi \in A$ *или* $\mathbf{F}\psi \in A'$,

C4. $\mathbf{G}\psi \in A \Leftrightarrow \mathbf{G}\psi \in A'$.

Правило C3 говорит о том, что если атом включает формулу $\mathbf{F}\psi$, то или он также включает формулу ψ (т. е. этот атом сам реализует обязательство выполнения ψ в будущем), или каждый его переход выполняется в атом, который помечен формулой $\mathbf{F}\psi$. Обратно: в атом, помеченный формулой $\mathbf{F}\psi$, может быть переход только из атома, также помеченного $\mathbf{F}\psi$. Из правила C3 вытекает, например, что не может быть переходов между атомами $\{p, \mathbf{F}q\}$ и $\{p, r\}$.

Правило C4 говорит о том, что если атом включает формулу $\mathbf{G}\psi$, то он должен включать также и ψ, и переходы из него ведут только в атомы, включающие формулу $\mathbf{G}\psi$. Обратно: в атом, помеченный $\mathbf{G}\psi$, может быть переход только из атома, помеченного $\mathbf{G}\psi$. Из правила C4 вытекает, например, что не может быть переходов между атомами $\{p, \mathbf{G}p\}$ и $\{p, r\}$.

Допускающие состояния искомого автомата Бюхи

Условия приема входных ω-слов автоматом Бюхи должно быть определено для каждой подформулы исходной формулы φ. Это требование реализуется следующим образом: искомый автомат Бюхи будем строить как обобщенный автомат Бюхи B_φ с множеством принимающих состояний $F = \left\{ \mathcal{F}_\psi \mid \psi \in cl(\varphi) \right\}$. Каждое множество $\mathcal{F}_\psi$ будет включать все те состояния, в которых обязательства выполнения формулы ψ будут реализованы. Для атомарного предиката p обязательства его выполнения реализуются непосредственно в том состоянии, которое включает этот предикат. Поэтому в искомом автомате Бюхи не будет состояний, в которых не реализованы обязательства выполнения p. Поэтому множество принимающих состояний $\mathcal{F}_p$ автомата B_φ будет совпадать со всем множеством состояний B_φ. Аналогично строится множество принимающих состояний для $\mathbf{X}\psi$: по построению, это обязательство выполнения формулы ψ будет реализовано в каждом следующем состоянии любого вычисления и во всех последующих состояниях. Поэтому перечисленные формулы можно вообще не учитывать при построении допускающих состояний обобщенного автомата B_φ.

Более сложными случаями являются формула $\varphi_i \mathbf{U} \psi_i$ и выводимые формулы $\mathbf{F}\psi_i$, $\mathbf{G}\psi_i$. Например, для каждой формулы $\alpha_i = \varphi_i \mathbf{U} \psi_i$, которая встречается в замыкании формулы φ, в искомом обобщенном автомате Бюхи B_φ множество $\mathcal{F}_i$ допускающих состояний составляют все атомы формулы φ, кроме тех, которые определяют нереализованные обязательства выполнения формулы α_i. Множества $\mathcal{F}_i$ допускающих состояний для формул $\mathbf{F}\psi_i$ и $\mathbf{G}\psi_i$ определяются аналогично.

Построение автомата Бюхи по формуле LTL

Построение искомого автомата Бюхи выполнить теперь очень просто. Очевидно, что автомат Бюхи, допускающий все ω-слова, на которых выполняет-

ся формула φ, может иметь только те состояния, которые могут быть помечены атомами формулы φ. Поэтому все атомы формулы φ, определяемые локальными правилами R1—R8, мы и должны принять за состояния искомого автомата. Переходы этого автомата определяются правилами *реализации обязательств* C1—C4. Начальными состояниями искомого автомата Бюхи должны, очевидно, содержать обязательства выполнения этой формулы φ.

❖ ***Определение 4.7 (автомат Бюхи B_φ)***

Обобщенный автомат Бюхи B_φ, допускающий все ω-слова, на которых выполняется LTL-формула φ, определяется следующим образом:

$$B_\varphi = \left(Q, 2^{AP}, Q_0, \delta, F\right),$$

где:

- множество Q состояний B_φ составляют все атомы формулы φ: $Q \subseteq 2^{cl(\varphi)}$;
- алфавит 2^{AP} автомата B_φ состоит из подмножеств атомарных предикатов, над которыми определена формула φ;
- множество Q_0 начальных состояний B_φ составляют те атомы формулы φ, в которые входит сама формула φ;
- функция переходов δ: $Q \times 2^{AP} \to 2^Q$ согласована с семантикой формул LTL и определяется так:
  - для каждого состояния $A \in Q$ и множества $A \cap 2^{AP}$ атомарных предикатов из AP, включенных в атом A, переход $\delta\left(A,\ A \cap 2^{AP}\right)$ — это максимальное множество таких атомов $A' \in Q$, которые удовлетворяют всем правилам *реализации обязательств* C1—C4;
- $F = \left\{\left\{A \in Q \mid \varphi_i \mathbf{U} \psi_i \notin A \text{ или } \psi_i \in A\right\} \mid \varphi_i \mathbf{U} \psi_i \in cl(\varphi)\right\}$: множество $F = \left\{\mathcal{F}_1, ..., \mathcal{F}_n\right\}$ подмножеств допускающих состояний обобщенного автомата Бюхи B_φ строится так:
  - D1: Для каждой формулы $\alpha_i = \varphi_i \mathbf{U} \psi_i$, которая встречается в замыкании формулы φ, множество $\mathcal{F}_i \in F$ составляют такие атомы

формулы φ, в которые или не входит сама формула α_i, или входит ψ_i;

- D2: Для каждой подформулы $\alpha_j = \mathbf{F}\psi_j$ замыкания формулы φ множество $\mathcal{F}_j$ $\mathcal{F}_j \in F$ составляют такие атомы формулы φ, в которые или не входит сама формула α_j, или входит ψ_j;
- D3: Для каждой подформулы $\alpha_k = \mathbf{G}\psi_k$ замыкания формулы φ множество $\mathcal{F}_k \in F$ составляют такие атомы формулы φ, в которые или входит сама формула α_k, или не входит ψ_k.

Доказательство того, что построенный автомат допускает в точности все те ω-слова, на которых LTL-формула φ выполняется, приведено в [19].

Поясним построение обобщенного автомата Бюхи B_φ на примерах.

Пример 4.13

Для формулы $\varphi = \mathbf{F}p$ недетерминированный обобщенный автомат Бюхи B_φ, построенный по приведенным выше правилам, представлен на рис. 4.20. Множество его состояний составляют три атома $A_1 = \varnothing$, $A_2 = \{\mathbf{F}p\}$ и $A_3 = \{p, \mathbf{F}p\}$ (см. пример 4.12, *1*). Множеством начальных состояний является $\{A_2, A_3\}$ — именно эти состояния помечены формулой $\mathbf{F}p$.

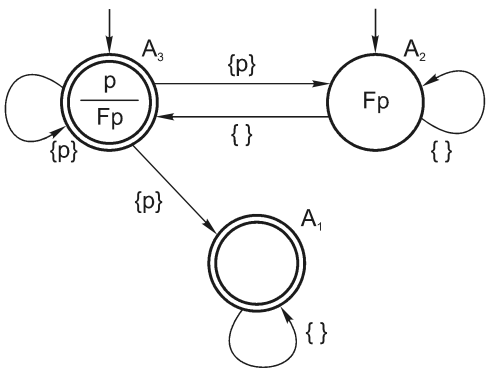


Рис. 4.20. Автомат Бюхи, допускающий все ω-слова, на которых выполняется формула $\mathbf{F}p$

Множество допускающих состояний здесь одно, $F = \{\mathcal{F}_1\}$, поскольку в формуле $\mathbf{F}p$ только одна формула вида $\mathbf{F}\varphi$. Состояние A_1 вообще не включает формулу $\mathbf{F}p$, а состояние A_3 включает p, поэтому $\mathcal{F}_1 = \{A_1, A_3\}$. Поскольку множество допускающих состояний одно, этот автомат является не обобщенным, а обыкновенным автоматом Бюхи.

Переходы автомата помечены теми наборами атомарных предикатов, которые истинны в исходящем состоянии, точно так же, как в автомате Бюхи, построенном по структуре Крипке. В автомате нет переходов из состояния A_2 в состояние A_1 и переходов из A_1 в A_2 и A_3 (они противоречат правилу С3 реализации обязательств).

Пример 4.14

Автомат Бюхи B_φ, допускающий все бесконечные входные цепочки, на которых выполняется формула $\varphi = \mathbf{GF}p$ (см. пример 4.12, *2*), показан на рис. 4.21. Состояниями этого автомата являются пять атомов формулы $\mathbf{GF}p$. Начальные состояния этого автомата — A_4 и A_5, те, которые помечены формулой $\mathbf{GF}p$. Три состояния недостижимы из двух начальных состояний (по правилу С4 *реализации обязательств*). Множество допускающих состояний автомата состоит из двух множеств, $\mathcal{F}_1$ для формулы $\mathbf{F}p$, и $\mathcal{F}_2$ для формулы $\mathbf{GF}p$. $\mathcal{F}_1 = \{A_5\}$ в соответствии с правилом D2, $\mathcal{F}_2 = \{A_4, A_5\}$ в соответствии с правилом D3.

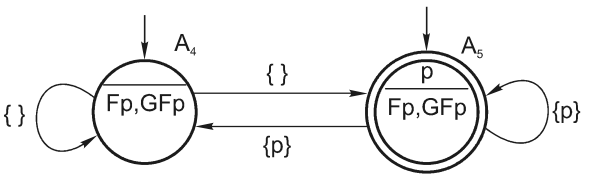


Рис. 4.21. Автомат Бюхи, допускающий все ω-слова, на которых выполняется формула $\mathbf{GF}p$

Автомат B_φ — обобщенный автомат Бюхи. Допустимые цепочки автомата B_{GF_p} — это все цепочки, которые бесконечное число раз проходят через состояние A_5 и через одно из состояний множества $\{A_4, A_5\}$. Поэтому этот автомат эквивалентен обычному автомату Бюхи с единственным допускающим состоянием A_5.

Пример 4.15

Автомат Бюхи B_φ, допускающий все входные цепочки, на которых выполняется формула $(p \vee q)\mathbf{U}(p \wedge q)$ (см. пример 4.12, *3*), показан на рис. 4.22. Допускающие состояния образуют одно множество, поскольку в формуле только одна подформула вида $\varphi_1\mathbf{U}\varphi_2$. Поэтому автомат B_φ — обычный автомат Бюхи. В соответствии с правилом D1, допускающие состояния здесь — все те, которые не включают формулу $\varphi_1\mathbf{U}\varphi_2$, и те, которые наряду с формулой $\varphi_1\mathbf{U}\varphi_2$ включают формулу φ_2. Переходы определены правилом *реализации обязательств* C2. Пометки на переходах опущены для простоты: каждый переход из состояния A помечен множеством тех атомарных предикатов, которые истинны в A.

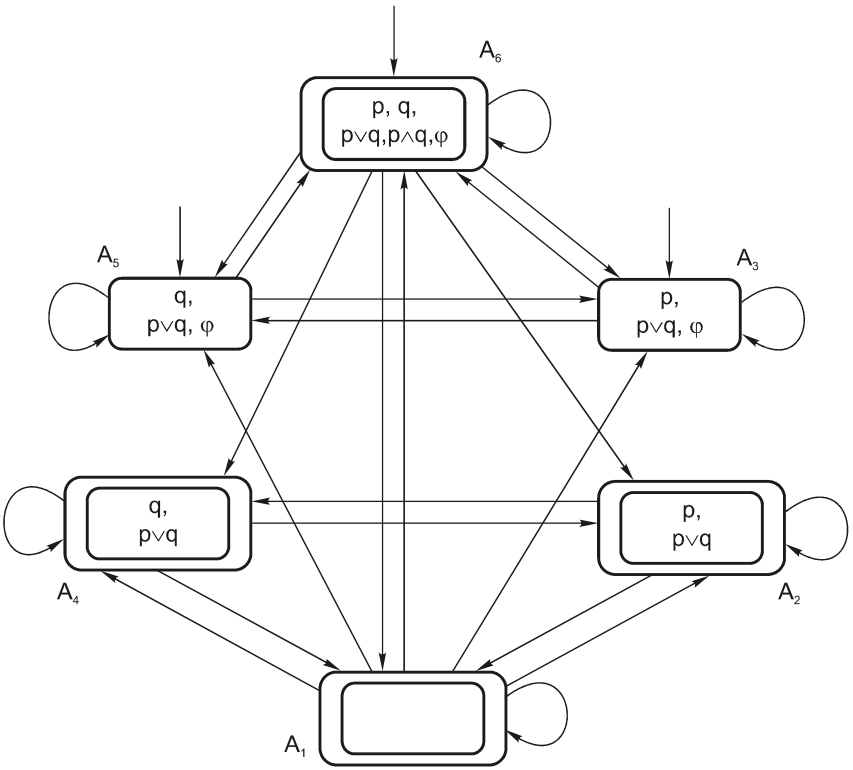


Рис. 4.22. Автомат Бюхи, допускающий все ω-слова, на которых выполняется формула $\varphi = (p \vee q)\mathbf{U}(p \wedge q)$

Пример 4.16

Построим автомат Бюхи для формулы $\varphi = \mathbf{X}p$. Множество состояний этого автомата — все подмножества множества $\{p, \mathbf{X}p\}$: $A_1 = \varnothing$, $A_2 = \{p\}$, $A_3 = \{p, \mathbf{X}p\}$, $A_4 = \{\mathbf{X}p\}$ (см. пример 4.12, *5*).

Множество начальных состояний автомата $S_0 = \{A_3, A_4\}$ — именно эти два состояния помечены формулой $\mathbf{X}p$. Функция переходов ограничена единственным правилом выполнения обязательств C1. Множество переходов этого автомата:

$$\delta(A_1, \varnothing) = \{A_1, A_4\}; \quad \delta(A_2, \{p\}) = \{A_1, A_4\};$$
$$\delta(A_3, \{p\}) = \{A_2, A_3\}; \quad \delta(A_4, \varnothing) = \{A_2, A_3\}.$$

Поскольку в формуле φ нет ни одной подформулы вида $\varphi_1 \mathbf{U} \varphi_2$, $\mathbf{F}\psi$ или $\mathbf{G}\psi$, то в построенном обобщенном автомате Бюхи нет ни одного подмножества допускающих состояний. По определению, бесконечное вычисление σ допускается в обобщенном автомате Бюхи тогда и только тогда, когда

$$\left(\left(\forall \mathcal{F}_i \in F\right) \inf(\sigma) \cap \mathcal{F}_i \neq \varnothing\right)$$

При пустом множестве F квантор $\forall \mathcal{F}_i \in F$ становится истинным. Поэтому любая бесконечная цепочка в построенном автомате Бюхи допускается.

Пример 4.17

Автомат Бюхи, допускающий ω-слова, на которых выполняется формула $\mathbf{FG}a$ (см. пример 4.12, *6*), представлен на рис. 4.23. Два состояния, A_1 и A_2,

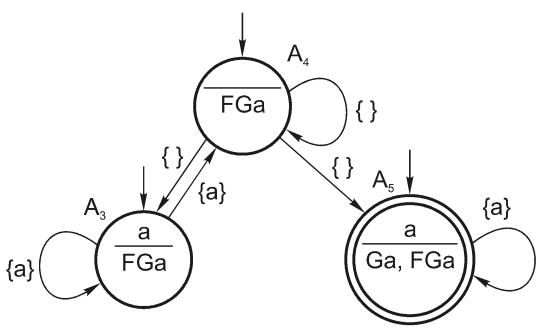


Рис. 4.23. Автомат Бюхи, допускающий все ω-слова, на которых выполняется формула $\mathbf{FG}a$

недостижимы из трех начальных состояний A_3, A_4 и A_5. По правилу D2 для формулы $\mathbf{FG}a$ допускающим состоянием является A_5 — только в A_5 входит формула $\mathbf{G}a$. Для формулы $\mathbf{G}a$ по правилу D3 допускающими состояниями являются A_4 и A_5, потому что в атом A_4 не входит a, а в атом A_5 входит $\mathbf{G}a$. Поэтому множество заключительных состояний $F = \{\mathcal{F}_1, \mathcal{F}_2\}$, где $\mathcal{F}_1 = \{A_5\}$, а $\mathcal{F}_2 = \{A_4, A_5\}$. Этот автомат можно рассматривать как обычный автомат Бюхи с одним допускающим состоянием A_5.

Пример 4.18

Автомат Бюхи B_φ, допускающий все бесконечные входные цепочки, на которых выполняется формула $\varphi = (a \vee b)\mathbf{UG}a$ (см. пример 4.12, *7*), имеет 9 состояний. Этот автомат показан на рис. 4.24. Начальными состояниями автомата являются атомы $A_5 — A_9$, потому что только они включают формулу φ. Переходы автомата определяются ограничениями, диктуемыми правилами C2, C4. Из состояний A_5, A_6 переходы могут вести только в состояния A_5, A_6 и A_7, из состояния A_6 — в состояния $A_5 — A_9$, а из состояний A_8 и A_9 переходы могут вести только в состояния A_8 и A_9. Поэтому состояния $A_1 — A_4$ являются недостижимыми. Множество F содержит два множества допускающих состояний: для формулы $(a \vee b)\mathbf{UG}a$ это множество $\{A_8, A_9\}$, а для формулы $\mathbf{G}a$ — это множество $\{A_6, A_8, A_9\}$. Отсюда следует, что этот автомат можно рассматривать как обычный автомат Бюхи с множеством допускающих состояний $\{A_8, A_9\}$.

Размер автомата Бюхи, построенного приведенным выше алгоритмом, в некоторых случаях не очень велик. Так, в примере 4.14 нами построен минимальный автомат Бюхи для формулы $\mathbf{GF}p$. Однако почти всегда размер автомата Бюхи, построенного этим методом, экспоненциален относительно размера (числа подформул) формулы φ. Верхняя граница сложности такого автомата $2^{O(|\varphi|)}$. Минимальный автомат Бюхи, допускающий тот же язык, может быть значительно меньшим. Например, автомат Бюхи, допускающий цепочки с бесконечным числом вхождений a и конечным числом вхождений b, показан на рис. 4.11. Он имеет только два состояния. Эквивалентный автомат Бюхи, построенный по приведенному здесь алгоритму, имеет 5 состояний (рис. 4.24). На рис. 4.15 приведен автомат Бюхи с двумя состояниями, допускающий все цепочки, удовлетворяющие формуле $(p \vee q)\mathbf{U}(p \wedge q)$, для

которой в примере 4.15 построен сложный автомат с шестью состояниями (см. рис. 4.22). Существуют примеры автоматов Бюхи, для которых граница сложности $2^{O(|\varphi|)}$ не может быть улучшена.

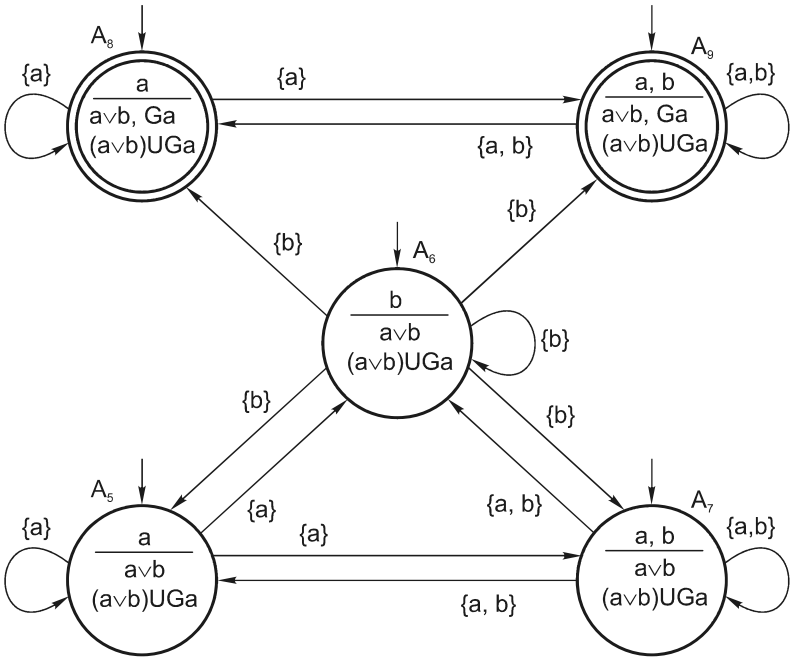


Рис. 4.24. Автомат Бюхи, допускающий все возможные ω-слова, на которых выполняется формула $(a \vee b)\mathbf{UG}a$

4.10. Заключение

Алгоритм *model checking* для формул LTL принимает на вход структуру Крипке M и формулу φ логики LTL, и в том случае, когда $M \not\models \varphi$, алгоритм выдает контрпример — одно из вычислений M, на котором формула φ не выполняется. Построенный автоматически контрпример является ценной информацией, которая позволяет найти ошибки в разработке, представленной моделью M.

Основой этого подхода является формальная модель — автомат Бюхи, позволяющий конечным образом задавать бесконечные языки. Автомат Бюхи отличается от обычного конечного автомата тем, что его семантика определена на ω-словах — бесконечных цепочках символов. ω-слово допускается авто-

матом Бюхи, если существует допускающее состояние автомата, которое проходится бесконечное число раз при приеме этого слова. В этой главе было показано, что с помощью автомата Бюхи, т. е. конечной моделью, можно представить все бесконечные траектории поведения, удовлетворяющие заданной формуле линейной темпоральной логики.

Следует ясно понимать, что автомат Бюхи — это не операционное устройство, которое "действует", "функционирует" под воздействием входных сигналов, как-то выбирает дальнейший путь движения, если выбор недетерминирован, подсчитывает число попаданий в допускающие состояния и на основе подсчета допускает или отвергает бесконечные цепочки. Автомат Бюхи — это не реальное устройство, это просто абстракция, формальная модель, удобная для задания и анализа ω-языков. С этой моделью можно выполнять абстрактные операции. Многие проблемы для автомата Бюхи неразрешимы (например, проверка эквивалентности двух автоматов Бюхи). Но некоторые проблемы разрешимы, и для их решения можно построить алгоритмы. Оказалось, что анализ этой формальной модели позволяет решать важные практические проблемы.

В частности, для автоматов Бюхи разрешима проблема пустоты допускаемого языка. Поэтому если заданы два автомата Бюхи, один из которых, B, описывает возможное функционирование проектируемой системы (бесконечное число бесконечных траекторий поведения), а другой, "контрольный" автомат K, описывает все те поведения, которые не удовлетворяют некоторому желаемому свойству (т. е. допускает все "неправильные" траектории), то, выполнив операцию синхронной композиции над этими моделями, мы строим новый автомат Бюхи $B \otimes K$, по которому можем установить, возможны ли "некорректные" траектории в будущем поведении нашей системы, проверив непустоту языка, допускаемого автоматом $B \otimes K$.

Именно эта техника и описана в данной главе. Одним из этапов этой техники является построение автомата Бюхи, допускающего все цепочки, удовлетворяющие заданной LTL-формуле (отрицанию формулы, представляющей проверяемое требование). Этот этап является одним из самых трудоемких в теоретико-автоматном подходе к верификации дискретных систем. Существует несколько подходов к процедуре такой трансляции, все они концептуально сложны и обычно проводятся в два шага. Первый шаг — построение автомата Бюхи, гарантированно удовлетворяющего поставленным требованиям. На этом шаге обычно получается автомат, сложность которого экспоненциальна относительно рассматриваемой формулы. Для того чтобы сделать эффективными следующие этапы процесса верификации (построение синхронной композиции автоматов и поиск в этой композиции циклов, содержащих принимающие состояния), обычно выполняется второй шаг, который заключает-

ся в минимизации полученного автомата. Выше был приведен концептуально наиболее простой из алгоритмов, выполняющих первый шаг.

К настоящему времени неизвестны эффективные общие процедуры получения по формуле LTL минимального автомата Бюхи; наилучшие известные методы основаны на использовании эвристик для такой минимизации.

На основе разработанной теории были построены мощные системы верификации, в частности, Cospan (Robert P. Kurshan) и Spin (Gerard J. Holzmann), которые в настоящее время широко используются для верификации крупных программно-аппаратных проектов. Система верификации Spin будет подробно рассматриваться нами далее.

За разработку теоретико-автоматного метода и практическую реализацию мощных инструментов для формальной верификации реагирующих систем, основанных на этом методе, четверо ученых, Gerard J. Holzmann, Robert P. Kurshan, Moshe Y. Vardi и Pierre Wolper, в 2005 г. были награждены премией ACM "*P.Kannelakis Theory and Practice Award*". В пресс-релизе ACM, посвященном этой награде, говорится:

"*Ключевой проблемой компьютерной науки является поиск методов проверки того, что разработанное аппаратное и программное обеспечение соответствует своей спецификации. Эта проблема особенно актуальна для реагирующих систем, которые взаимодействуют со своим окружением, таких, как цифровые системы и коммуникационные протоколы. Четыре лауреата премии показали, как вопрос проверки корректности реагирующих систем относительно выразительной спецификации может быть сведен к проблемам, связанным с конечными автоматами на бесконечных входных последовательностях. Они показали, что на основе этого подхода могут быть построены эффективные алгоритмы, позволяющие автоматизировать доказательство корректности на практике. Разработка теоретико-автоматного подхода к верификации потребовала новых результатов как в теории автоматов, так и в области программных логик*".

4.11. Замечания

Теоретико-автоматный подход к проверке выполнения формул LTL предложен Moshe Vardi и Pierre Wolper в [143], [145], [142]. Как говорилось выше, существуют разные пути обобщения модели конечного автомата, чтобы он мог допускать бесконечные цепочки. Формальную модель рассмотренного нами автомата Бюхи — распознавателя ω-языков — ввел Richard Büchi [26].

Проблема построения автомата Бюхи по заданной формуле LTL вызвала большой интерес, который не пропал и в настоящее время. Существует не-

сколько подходов к ее решению, однако все они в худшем случае дают автомат Бюхи, экспоненциально сложный относительно длины LTL-формулы. Приведенный в этой главе алгоритм построения автомата Бюхи основан на результатах, изложенных в работе (*M. Mukund. Linear-Time Temporal Logic and Buchi Automata,* 1997), а также в [145] и [86]. В [19] приводится доказательство сложности построенного этим методом автомата Бюхи. О конечных автоматах и распознавании ими языков можно прочитать в [158]. Наибольшее распространение среди систем верификации, основанных на описанной здесь технике, получила система Spin [80], [79]. Она описывается в *гл. 7*.

Существует несколько разработок, авторы которых пытаются сделать еще более выразительными и удобными для практической верификации возможности формальной спецификации на языке LTL. Например, язык FTL (ForSpec Temporal Logic, [6]), разработанный в 2002 г. компанией Intel (Интел), для более удобной спецификации свойств систем расширяет логику LTL введением логических и арифметических операций на битовых векторах. В этом языке используется также возможность описания регулярных событий и отношений между этими событиями, хотя основой его остается язык формул логики LTL.

4.12. Задачи к главе 4

4.1. Какое свойство структуры Крипке с n состояниями $\{s_1,...s_n\}$ выражается формулой:

$$\mathbf{E}\left[\mathbf{F}p_1 \wedge \mathbf{G}\left(p_1 \Rightarrow \mathbf{XG}\neg p_1\right) \wedge ... \wedge \mathbf{F}p_n \wedge \mathbf{G}\left(p_n \Rightarrow \mathbf{XG}\neg p_n\right)\right]$$

где $p_1,...p_n$ — атомарные предикаты, причем предикат p_k истинен только в состоянии s_k. Постройте структуру Крипке, удовлетворяющую этому свойству.

4.2. Какое свойство структуры Крипке выражается формулой:

$$\mathbf{E}\left[p_0 \wedge \mathbf{X}p_1 \wedge \mathbf{XX}p_2 \wedge ... \wedge \mathbf{X}^n p_n\right],$$

где $p_1,...p_n$ — атомарные предикаты, причем предикат p_k истинен только в состоянии s_k структуры Крипке. Постройте структуру Крипке, удовлетворяющую этому свойству.

4.3. Пусть задан автомат $A=(S,\Sigma,S_0,\delta,F)$, где $S=\{s_0,s_1,s_2\}$, $\Sigma=\{a,b\}$, $S_0=\{s_0\}$, $\delta=\{(s_0,a,s_1),(s_0,b,s_2),(s_1,a,s_1),(s_2,b,s_1),(s_1,b,s_2)\}$, $F=\{s_1\}$.

Приведите примеры цепочек, которые допускает A, если его рассматривать а) как конечный автомат; б) как автомат Бюхи.

4.4. Докажите, что не существует детерминированного автомата Бюхи, допускающего все цепочки с конечным числом вхождений a и бесконечным числом вхождений b (недетерминированный автомат Бюхи, допускающий этот язык, представлен на рис. 4.8).

Этот пример показывает, что недетерминированные автоматы Бюхи имеют более широкие возможности, чем детерминированные.

> ❖ ***Указание***
>
> Докажите, что детерминированный автомат, чтобы допустить любую цепочку с конечным числом вхождений a и бесконечным числом вхождений b при подаче на его вход символа a, а затем конечной цепочки из символов b, должен посетить допускающее состояние после конечного числа символов b. Покажите, что эту процедуру можно повторить.

4.5. На рис. 4.7 представлен автомат Бюхи, распознающий множество L таких бесконечных цепочек над $\Sigma=\{a,b\}$, которые включают бесконечное число символов b. Дополнение этого ω-языка — ω-язык Σ/L, цепочки которого включают только конечное число символов b. Постройте автомат Бюхи, который распознает язык Σ/L.

4.6. Постройте формулу LTL, выражающую свойство: "*состояние, удовлетворяющее p, встречается самое большее один раз на каждом вычислении*". Постройте структуру Крипке, на которой эта формула выполняется.

4.7. Постройте минимальный конечный автомат, распознающий язык $b*a*$. Какой язык допускает этот автомат, если его рассматривать как автомат Бюхи?

4.8. По произвольно заданной структуре Крипке K и формуле Ф LTL проверьте вручную, удовлетворяется ли формула Ф на структуре Крипке K. Для этого постройте автоматы Бюхи A_K и $B_{\neg\Phi}$; постройте их композицию и проверьте, пусто ли пересечение языков L_K и $L_{\neg\Phi}$.

4.9. Постройте автомат Бюхи, распознающий все цепочки, задаваемые LTL-формулой $\mathbf{G}p$ для входных алфавитов $\Sigma=\{p\}$ и $\Sigma=\{p,q\}$.

4.10. Постройте автоматы Бюхи, распознающие все цепочки, задаваемые LTL-формулами $\mathbf{G}p$,

$$p\wedge\mathbf{XG}\neg q,\ \mathbf{G}\left(p\Rightarrow p\mathbf{U}q\right),\ \mathbf{FG}p,\ \mathbf{GF}\neg p,\ \mathbf{F}\left(p\neg\wedge\left(q\mathbf{U}r\right)\right),$$

$$p\mathbf{U}q,\ \left(\mathbf{FG}p\right)\Rightarrow\left(\mathbf{GF}q\right),\ \mathbf{XX}p$$

4.11. По обобщенному автомату Бюхи $A=(S,\Sigma,S_0,\delta,F)$, где:

$$S=\{s_0,s_1\},\ \Sigma=\{a,b\},\ S_0=\{s_0\},$$

$$\delta=\{(s_0,a,s_0),(s_0,b,s_1),(s_1,b,s_1),(s_1,a,s_0)\},\ F=\{\{s_0\},\{s_1\}\}$$

постройте эквивалентный автомат Бюхи.

4.12. Постройте LTL-формулы, задающие следующие свойства:

- ❒ если сообщение m посылается неопределенно много раз передатчиком, оно будет принято неопределенно много раз приемником;
- ❒ канал теряет только конечное число сообщений;
- ❒ если процесс попал в критическую секцию, то он выйдет из нее через конечное число временных шагов;
- ❒ если сообщение получено приемником, то оно было передано передатчиком по крайней мере на 1 единицу времени раньше;
- ❒ если в ALU загружена операция, то результат операции будет на выходе ALU точно через 4 временных шага (используйте для определения шага оператор $\mathbf{X}$);
- ❒ ни одно состояние, в котором переменная x равна 1, не будет достигнуто в процессе вычислений.

4.13. Постройте LTL-формулы, описывающие следующие свойства:

- ❒ если случится p, то потом никогда не выполнится q;
- ❒ всегда если p выполнится, то в будущем обязательно выполнится q, а за ним сразу выполнится r;
- ❒ всегда после выполнения p когда-нибудь выполнится q, а до q не будет выполнено r.

4.14. Постройте структуру Крипке, на которой формула CTL $\mathbf{AGEF}p$ выполняется, а формула LTL $\mathbf{GF}p$ — не выполняется.

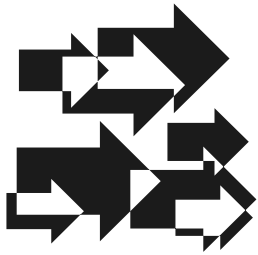
4.15. Постройте автомат Бюхи, допускающий цепочку $(ab)^{\omega}$.

4.16. Постройте автоматы Бюхи для формул $p\mathbf{U}q$, $p\mathbf{UX}q$, $\mathbf{GF}\neg p$, $\mathbf{FG}(p\wedge q)$ по правилам, изложенным в п. 4.9.

4.17. Постройте автомат Бюхи для формулы $\mathbf{G}((p\wedge\neg q)\mathbf{U}(\neg p\wedge q))$ по правилам, изложенным в п. 4.9.

4.18. Постройте автомат Бюхи для формулы $\mathbf{G}(p\wedge\neg q)\mathbf{UG}(\neg p\wedge q)$ по правилам, изложенным в п. 4.9.

ГЛАВА 5



Структуры Крипке как модели реагирующих систем

В этой главе показано, как реагирующие системы могут быть представлены моделями с конечным числом состояний для последующего их анализа с помощью техники model checking. Используемая здесь модель имеет ясную, простую и недвусмысленную семантику. Модель всегда проще реальной системы, она строится с помощью абстрагирования, отбрасывания части параметров, характеристик, особенностей реальной системы. При построении модели важно, чтобы она сохранила существенные черты исходной реальной системы.

В этой главе приведены примеры представления реагирующих систем в виде структуры Крипке. Для класса реагирующих систем, в которых основную роль играют потоки управления, а не потоки данных, формализация в виде структуры Крипке зачастую полностью адекватна, т. е. она отражает возможные порядки событий в исходной системе. Протоколы, операционные системы, логические системы управления — все эти программные системы могут естественным образом моделироваться структурой Крипке или ее вариантами.

5.1. Состояния и переходы реагирующей системы. Гранулярность переходов

Формальному анализу при верификации с помощью метода *model checking* подвергаются не сами реагирующие системы, а их модели с конечным числом состояний — структуры Крипке. Структура Крипке — это абстракция, формальная модель, и построение ее по реальной системе является важнейшим этапом процесса верификации. Структура Крипке должна адекватно

представлять систему: если существенные свойства системы не представлены в модели, то верификация бессмысленна. Рассмотрим, как основные элементы модели Крипке — состояния и переходы — соотносятся с состояниями и их изменениями в реальных системах.

Понятие *состояния* существует в любой реальной системе. Состояние представляет собой мгновенный снимок (snapshot) всей информации о системе в каждый конкретный момент времени. Каждое состояние имеет конечное описание. Обычно это вектор значений переменных и точка управления в программе. Реальные системы часто имеют бесконечное число состояний из-за бесконечной области определений параметров. В тех случаях, когда такие параметры не влияют на логику поведения программы, от них можно абстрагироваться и для верификации строить модель системы с конечным числом состояний. В частности, в коммуникационных протоколах содержимое передаваемых сообщений не влияет на логику протокола. Во многих случаях для верификации программ область значений переменных можно ограничить. Например, потенциально бесконечный объем буфера или значения счетчиков можно ограничить величиной 2—3, область значений вещественных переменных можно разделить на несколько классов эквивалентности. Возникающие при таком абстрагировании дополнительные ошибки, выявленные верификатором, можно игнорировать после проверки найденных контрпримеров тестированием реальной системы.

Мы рассматриваем дискретные системы, динамика которых представляется сменой состояний при функционировании системы в дискретные моменты времени. Смена состояний реальной системы вызывается выполнением возможных событий. Иными словами, дискретная система находится в некотором стабильном состоянии, и под влиянием некоторого события (приема или посылки сообщения, выполнения операции, или внешнего события — например, нажатия кнопки мыши или прихода тактового синхронизирующего импульса) она переходит в другое состояние, тоже стабильное (рис. 5.1).

Задачей верификации является анализ *поведения* дискретной системы, динамики ее развития во времени. Поведение дискретной системы — это бесконечная цепочка (последовательность) ее состояний, сменяющих друг друга. При верификации мы проверяем выполнение формул темпоральной логики, истинность которых зависит от времени, хотя время явно в них не присутствует. Но как можно говорить о поведении системы во времени, не упоминая время? Удобство модели переходов в том и состоит, что она абстрагируется от значений и понятий времени, учитывая только возможные последовательности наступления событий и переходы системы из одного состояния в другое под воздействием этих событий.

Переходы модели представляют изменения состояний реальной системы. Как реальная система переходит из одного своего состояния в другое? Какова

протяженность перехода системы во времени? Что происходит в реальной системе в процессе перехода?

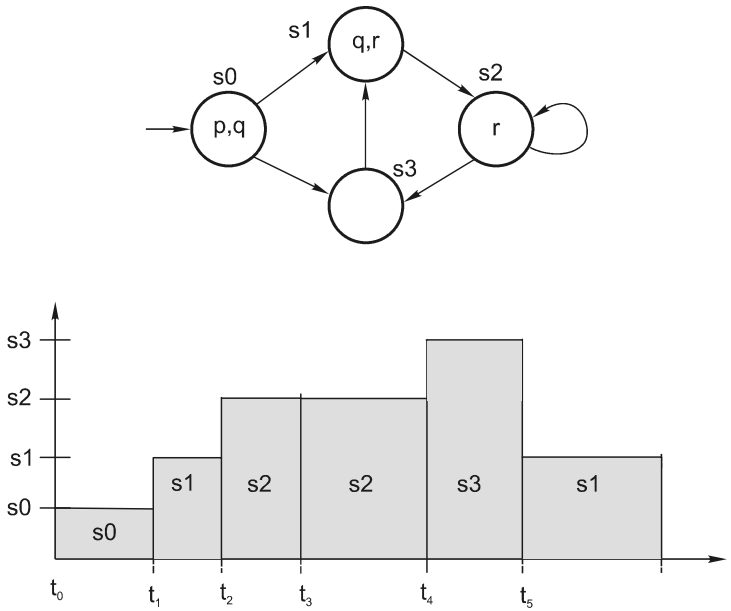


Рис. 5.1. Смена состояний реагирующей системы во времени

Модель состояний и переходов абстрагируется от всех этих вопросов. При переходе модели из состояния в состояние предполагается, что модель не проходит промежуточных состояний в том смысле, что не существует "*части*" перехода, система выполняет переход как бы "*мгновенно*".

В действительности, мы не можем понимать термин *мгновенный* буквально: мгновенных процессов не бывает. Кроме того, в нашем анализе мы всеми способами стараемся уйти от явного выражения времени, а термин "мгновенный" связан именно с понятием реального времени. В реальных дискретных системах вместо понятия *мгновенности* обычно используется понятие "*атомарность*" или "*неделимость*" изменения состояния.

❖ ***Определение 5.1 (атомарность)***

Атомарность изменения состояния в реальной системе состоит в том, что при этом изменении реальная система не проходит промежуточные этапы в следующем смысле: извне система видна (наблюдаема) только либо в одном, либо в другом стабильном состоянии, но не в процессе изменения состояний.

Иными словами, переходы системы будут атомарными, если пользователь системы и другие активные процессы могут наблюдать ее всегда в каком-то ее состоянии либо до начала изменения состояния, либо после его завершения, но никогда во время изменения.

Например, изменение значения переменной в результате выполнения оператора присваивания в реальной системе:

$$x = 5$$

не может быть частичным, мы можем *наблюдать* значение переменной x либо до, либо после выполнения операции присваивания, что обеспечивается в реальных процессорах аппаратно. Таким образом, с точки зрения *других процессов* этот переход неделим, мгновенен.

Изменение значения той же переменной:

$$x = x + 5$$

уже не так просто. Чаще всего оно реализуется тремя неделимыми операциями: чтением старого значения x в регистр, изменением содержимого регистра и затем записью значения регистра в область памяти, отведенную для x. Если переменная x — локальная, и другие процессы не имеют к ней доступа, то в модели мы можем представить выполнение этого оператора одним переходом, изменяющим состояние. Если переменная x — разделяемая, то другие процессы могут наблюдать (и изменять) промежуточные состояния, если они реально существуют при выполнении этого оператора. Поэтому в модели этот оператор следует представлять несколькими переходами с промежуточными состояниями. Если этот оператор в модели представить одним переходом, то построенная модель будет неадекватной и скрывать ошибки, существующие в системе. Этот оператор можно представить в модели одним переходом в том случае, если этот оператор явно "закрывается" с помощью специальных программных средств: семафоров, операций *test-and-set* и т. п.

Излишняя детализация при построении модели также вредна. Если оператор в действительности атомарный, а в модели он представлен несколькими переходами и промежуточными состояниями, которые в реальности не видны другим процессам, то при анализе могут обнаружиться ошибки, которых в действительности система не имеет. Кроме того, число состояний модели может стать слишком большим.

Отметим еще раз: в реальности все переходы являются протяженными во времени, их представление в модели как мгновенных неделимых актов является абстракцией, которая нуждается в обосновании. Представление реальной системы моделью состояний и неделимых переходов будет адекватным в следующих случаях:

- если неделимость (атомарность) операции (или группы операций) в реальных дискретных системах обеспечивается либо аппаратно, либо программно с помощью, например, аппаратного запрета прерывания процесса или примитивов синхронизации. В частности, в модели не нужно и вредно явно представлять каждое изменение напряжения на входе (выходе) регистров, хранящих значения переменных программы — неделимость операций чтения-записи в регистры обеспечивается аппаратно;
- если промежуточные состояния, которые система в действительности проходит при переходе, не влияют на анализируемые свойства системы.

Если промежуточные состояния являются существенными для анализа, доступными для наблюдения и изменения, то они должны быть явно введены в модель, например, как состояния "перехода", в которых система находится некоторое время.

> ❖ ***Определение 5.2 (адекватность)***
>
> Модель системы переходов будет адекватна реальной системе, если пространство состояний системы переходов совпадает с пространством значащих состояний реальной системы, а переходы модели являются неделимыми изменениями состояний в реальной системе.

5.2. Реагирующие системы, специфицированные в виде систем переходов

В некоторых случаях верификация выполняется для систем, уже представленных в виде конечной модели системы переходов. Наиболее распространенные такие модели — это конечные автоматы и их разнообразные расширения. В этом формализме часто описывают задание на проектирование встроенных систем управления, контроллеров, протоколов и т. п. Такое описание обычно является техническим заданием на реализацию, но оно же может служить и для проверки свойств разрабатываемой системы с помощью техники model checking.

Рассмотрим спецификацию для разработки и анализа системы управления микроволновой печью из [34], представленной состояниями и переходами. Множество атомарных предикатов, которые необходимы для анализа свойств поведения системы: $\{Start, Close, Heat, Error\}$ (рис. 5.2). В каждом состоянии здесь указаны атомарные предикаты, истинные в этом состоянии. *Close* указывает, что дверца печи открыта, *Heat* — что нагрев включен. Если в неко-

тором состоянии атомарный предикат ложен, он в этом состоянии не указывается. Например, в состоянии 2 атомарные предикаты *Start* и *Error* истинны, а *Close* и *Heat* ложны. Переходы помечены действиями — внешними и внутренними событиями, при которых осуществляются переходы. Как говорилось выше, в модели переходов мы абстрагируемся от "протяженных" переходов: не учитываем, например, того, что дверца печи может быть только частично приоткрыта. В этой модели дверца может быть только в двух состояниях: либо "*открыта*", либо "*закрыта*". Адекватность этой модели реальной микроволновой печи обеспечивается устройством печи: в состоянии 6, например, внешняя команда "*открыть дверцу*" игнорируется, датчик "*дверца закрыта*" выдает только два значения, *true* и *false*, он установится в *true* только при полностью закрытой дверце. Состояние 4 представляет протяженный переход из состояния 7 от запуска нагревателя (событие "*начать стряпню*") до состояния 3 готовности.

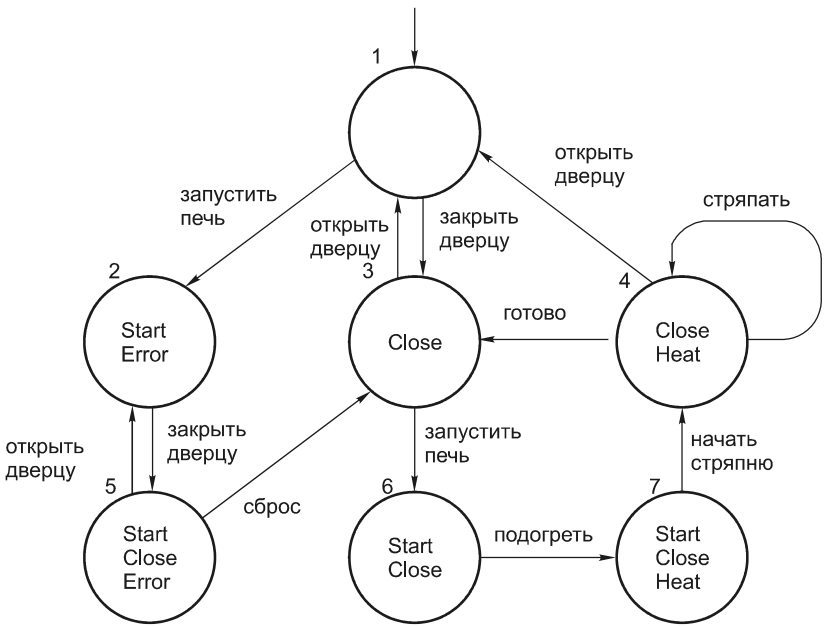


Рис. 5.2. Описание алгоритма функционирования микроволновой печи

Соответствующая структура Крипке получается из этого описания отбрасыванием действий на переходах. Проверка требований к системе производится с игнорированием конкретных действий окружения, которые приведут к нарушению этих требований. Например, требование $\mathbf{AG}\left(\neg Close \Rightarrow \neg Heat\right)$ — "*В любом состоянии всегда при открытой дверце нагревание не происхо-*

дит", должно выполняться *при любых поведениях окружения*. Если требование нарушается, то ведущие к ошибке действия окружения можно легко восстановить по контрпримеру, выданному при верификации. Тем самым появляется возможность анализа причины ошибки.

Эта же спецификация может быть и заданием на реализацию — давно используемые в проектировании методы структурного синтеза систем логического управления используют подобную спецификацию для разработки функциональной схемы управляющего автомата. Например, в системе автоматизированного проектирования Active-HDL фирмы Aldec включен модуль FSM (Finite State Machine, конечный автомат). Этот модуль обладает многофункциональным графическим интерфейсом для описания управляющих автоматов.

5.3. Реагирующие системы, специфицированные в виде нескольких взаимодействующих систем переходов

Более сложный случай — это описание параллельных и распределенных систем, каждый модуль которых задается как система переходов. Рассмотрим в качестве примера анализ протокола передачи данных. Формальное описание протоколов с помощью модели конечного автомата уже много лет используется для их недвусмысленного задания. Протокол, который мы будем анализировать, носит название PAR (Positive Acknowledgements with Retransmission) — *протокол с положительными подтверждениями и повторением передачи*. Архитектура протокола представлена на рис. 5.3. Пользователю A необходимо направить пользователю B поток сообщений через ненадежную среду, теряющую или искажающую сообщения. Для того чтобы передать поток сообщений к B (пунктирная стрелка на рис. 5.3), пользователь A использует для передачи специальный сервис, задача которого — обеспечить доставку данных без потерь и дублирования в правильном порядке через ненадежный канал. Сервис доставки сообщений использует коммуникационный протокол, который состоит из двух протокольных объектов, передатчика S и приемника R. Передатчик принимает от A сообщения, нумерует их и передает приемнику через полудуплексный (однонаправленный) канал $Ch1$. Получив сообщение, приемник передает его пользователю B и посылает подтверждение получения сообщения через полудуплексный канал $Ch2$. Каналы $Ch1$ и $Ch2$ ненадежны: как сообщение, так и подтверждение могут потеряться или исказиться. Искажение данных обнаруживается нижним уровнем протокола (кодом, обнаруживающим ошибки), и такие данные отбрасываются, поэтому искажение эквивалентно потере сообщения. Цифра-

ми в прямоугольниках на схеме рис. 5.3 обозначены коммуникационные порты, связывающие объекты протокола.

Не так просто обеспечить основное требование к протоколу передачи данных, а именно, чтобы посланные по ненадежному каналу сообщения были доставлены от пользователя A пользователю B без потерь и без дублирования. Например, в канале $Ch1$ может быть потеряно само сообщение. При этом передатчику не остается ничего, кроме как послать это сообщение повторно. Но передатчик здесь имеет ту же информацию (вернее, отсутствие информации о доставке сообщения), как и в случае, если в канале $Ch2$ потеряно подтверждение получения сообщения, посланное приемником. Поэтому передатчик должен повторно передать сообщение в обоих случаях. Приемник должен различать две ситуации, в одной из которых приходит дублированное сообщение, а в другой — нет. Для исключения дублирования, сообщения последовательно нумеруются по модулю 2 однобитным номером $0,1,0,1,\ldots$.

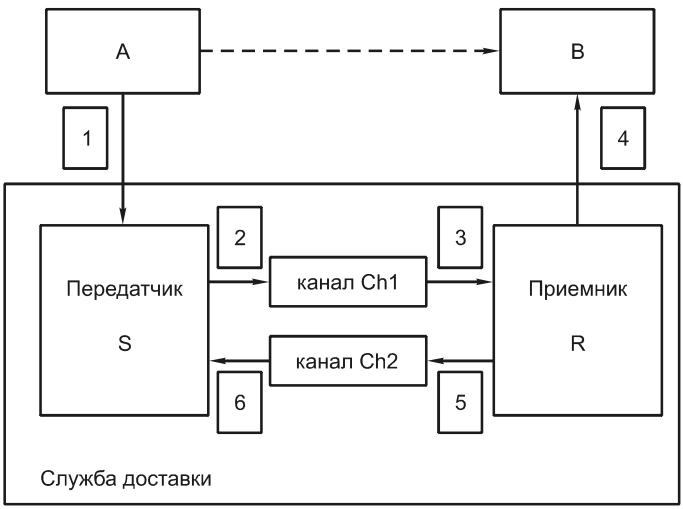


Рис. 5.3. Архитектура протокола PAR

Протокол PAR — это модификация хорошо известного классического протокола альтернирующего бита (ABP), в котором осуществляется однобитная нумерация не только передаваемых сообщений, но и подтверждений на эти сообщения. Известно, что если подтверждения не несут номер подтверждаемого сообщения, как в протоколе PAR, то протокол некорректен. Наша задача — выявить эту некорректность с помощью построенной модели переходов.

На рис. 5.4 все объекты протокола представлены конечно-автоматными моделями. Рассмотрим их по порядку.

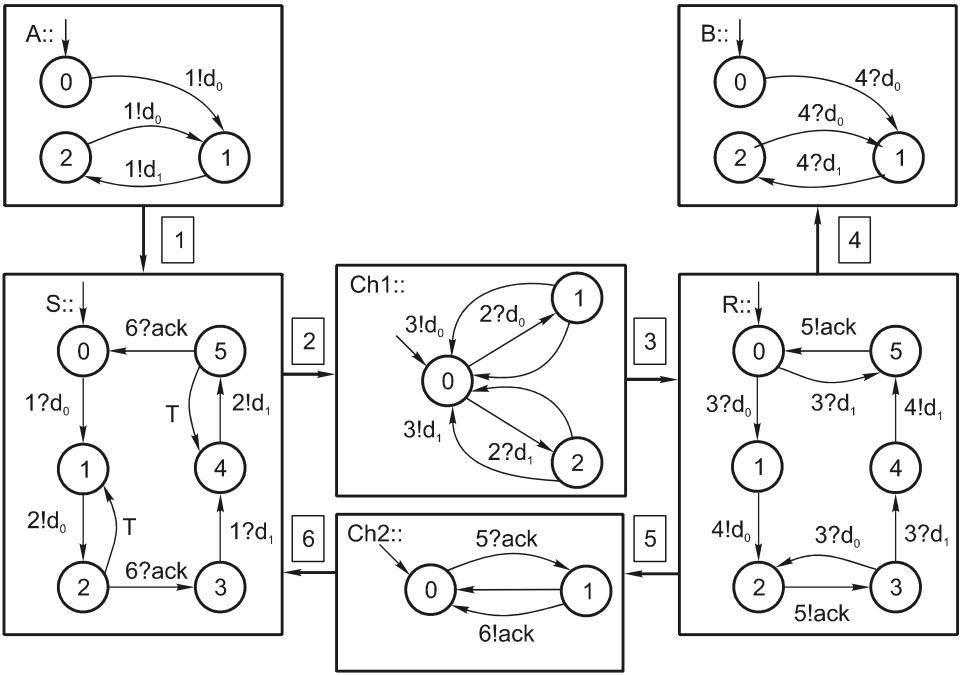


Рис. 5.4. Взаимодействующие автоматы, реализующие протокол PAR

Пользователь A последовательно передает данные, занумерованные 0 или 1, через порт 1, выполняя операцию $1!<данные>$, т. е. операции $1!d_0$ или $1!d_1$. Для упрощения модели сообщения нумеруются пользователем A, а не передатчиком, что, конечно, несущественно. Когда передатчик готов, он принимает сообщение через этот порт, выполняя операцию $1?<данные>$. Синхронизация обеспечивается тем, что в порту посылка данных одним процессом выполняется тогда и только тогда, когда другой процесс эти данные принимает в этом же порту — это взаимодействие рандеву (хэндшейк). Передатчик после чтения данных передает их в канал $Ch2$ через порт 2 (операция $2!<данные>$), после чего ждет подтверждения ack, которое должно прийти через порт 6. Поскольку данные могут потеряться, через некоторое время (тайм-аут) передатчик повторяет посылку сообщения в канал (переход, помеченный T).

Полудуплексный канал $Ch1$ принимает в порту 2 сообщение, помеченное либо 0, либо 1, и передает его приемнику в порту 3. Канал может принять не более одного сообщения. Чтобы принять следующее, канал должен вернуться в состояние 0. В канале сообщение может быть потеряно. Это моделируется спонтанным переходом канала в состояние 0. Остальные автоматы построе-

ны аналогично. Все процессы работают параллельно, синхронизируясь операциями коммуникации.

Структура Крипке, представляющая модель протокола PAR — это асинхронное произведение (композиция) компонентных автоматов. Глобальные состояния этого произведения — это наборы состояний всех компонентов. В соответствии с семантикой асинхронной композиции параллельных процессов, в каждый момент времени в системе может выполниться любое одно из возможных действий. Если два автомата готовы выполнить свои действия в некотором глобальном состоянии, то выполняется либо только одно, либо только другое действие, но не оба вместе. Таким образом, асинхронная композиция порождает недетерминизм, даже если каждый из параллельно функционирующих процессов полностью детерминированный. Недетерминизм здесь обусловлен не вероятностью наступления событий, а возможностью выполнения тех или иных действий. При верификации методом model checking мы как раз проверяем выполнение заданных требований на всех возможных путях, а не оцениваем вероятность каких-либо траекторий.

Такая семантика параллелизма (она называется *чередующимся исполнением*, или *интерливингом*, от английского interleaving) является следствием основного правила теории параллельных процессов: *при анализе параллельных процессов нельзя делать никаких предположений об относительных скоростях выполнения процессов или стратегии планировщика процессов*. Только в этом случае результаты анализа будут справедливы при любых условиях прерываний процессов, при любых скоростях процессоров и т. п. Таким образом, поведение двух параллельных процессов представляется недетерминированным интерливингом некоторого абстрактного последовательного процесса: *эффект от параллельного выполнения двух неделимых операций* α *и* β *равен эффекту от последовательного выполнения этих операций в любом порядке, т .е.* $\alpha;\beta$ *или* $\beta;\alpha$.

Формально асинхронная композиция параллельных процессов, взаимодействующих с помощью хэндшейка (рандеву), определяется следующим образом. Пусть

$$P_i = \left(S_i, \Sigma_i \rightarrow_i, S_{0i}\right), i = 1, 2$$

два параллельных процесса с множествами состояний S_i, множествами операций (пометок на переходах) Σ_i, множествами переходов $\rightarrow_i$ и множествами начальных состояний S_{0i}. Пусть операции взаимодействия рандеву обозначаются одинаково в обоих процессах. Обозначим $H = \Sigma_1 \cap \Sigma_2$ — множество этих операций (алфавит или словарь взаимодействия).

Результат операции асинхронной параллельной композиции процессов с операциями взаимодействия рандеву из алфавита H (обозначается $P_1 \| P_2$) определяется как следующий последовательный процесс $\mathcal{P}$:

$$\mathcal{P} = P_1 \| P_2 = \left(S_1 \times S_2, \Sigma_1 \cup \Sigma_2, \rightarrow, S_{01} \times S_{02}\right),$$

где множество $\rightarrow$ переходов процесса $\mathcal{P}$ определяется следующим образом:

(I) для возможных действий процесса P_1 в состоянии s_1:

$$\frac{s_1 - \alpha \rightarrow_1 s_1'}{\langle s_1, s_2 \rangle - \alpha \rightarrow \langle s_1', s_2 \rangle} \quad \text{при } \alpha \notin H,$$

если в процессе P_1 есть переход из s_1 в s_1' с выполнением операции α, не являющейся операцией рандеву, то в процессе $\mathcal{P}$ — результате параллельной композиции — этот переход может быть выполнен независимо от возможных действий процесса P_2;

(II) то же для возможных действий процесса P_1 в состоянии s_2:

$$\frac{s_2 - \alpha \rightarrow_2 s_2'}{\langle s_1, s_2 \rangle - \alpha \rightarrow \langle s_1, s_2' \rangle} \quad \text{при } \alpha \notin H$$

если в процессе P_2 есть переход из s_2 в s_2' с выполнением операции α, не являющейся операцией рандеву, то в процессе $\mathcal{P}$ этот переход может быть выполнен независимо от возможных действий процесса P_1;

(III) общее действие процессов P_1 и P_2, если каждый из них готов к этому действию:

$$\frac{s_1 - \alpha \rightarrow_1 s_1' \wedge \; s_2 - \alpha \rightarrow_2 s_2'}{\langle s_1, s_2 \rangle - \alpha \rightarrow \langle s_1', s_2' \rangle} \quad \text{при } \alpha \in H,$$

если в обоих процессах есть переход с выполнением операции α, являющейся операцией рандеву, то в параллельной композиции это действие выполняется обоими процессами одновременно.

Два первых правила перехода параллельной композиции определяют интерливинг, третье правило определяет взаимодействие параллельных процессов.

Если процессы параллельной программы взаимодействуют только с помощью общих переменных, то правило III взаимодействия отсутствует в их композиции.

Пример 5.1

Рассмотрим параллельную программу, состоящую из двух последовательных процессов, взаимодействующих через общую переменную x:

```
P1::                    P2::
  k0: x++;                n0: print(x);
  k1: x++;                n1: end
  k2: end
```

Пусть вначале x=0. Какое значение будет напечатано?

Для завершения параллельной программы оба процесса должны завершиться, выполнив все свои операторы. Однако эффект от выполнения операторов будет различным (может быть напечатано 0, 1 или 2) в зависимости от того, в какой последовательности эти операторы будут выполняться. Рис. 5.5 схематично показывает все возможные последовательности.

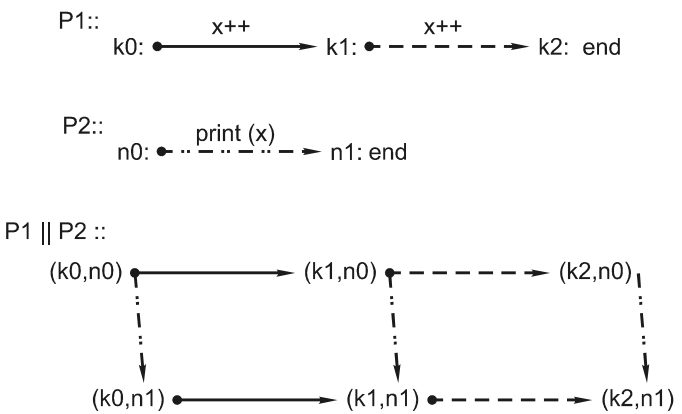


Рис. 5.5. Эффект интерливинга — возможные траектории выполнения параллельной программы

Вернемся к примеру построения модели протокола PAR. Возможные действия в параллельной композиции автоматов нашего примера, $\mathcal{P} = A \times S \times Ch1 \times Ch2 \times R \times B$ — это независимый переход любого компонентного автомата (интерливинг) или взаимодействие двух автоматов, готовых к этому взаимодействию в соответствии с семантикой рандеву. Для протокола PAR параллельная композиция представляется автоматом $\mathcal{P}$, глобальное состояние которого — это шестерка состояний. Всего в этом автомате около 2000 состояний (3×6×3×2×6×3), хотя и не все они достижимы из начального состояния. Первый компонент состояния автомата $\mathcal{P}$ — состояние автомата A, второй — состояние передатчика S, третий — состояние канала $Ch1$

и т. д. Начальное состояние произведения автоматов $\mathcal{P}$ — состояние $\langle 0,0,0,0,0,0\rangle$ (будем такую шестерку представлять просто 000000). Единственное возможное действие $\mathcal{P}$ в этом состоянии — это синхронное взаимодействие компонентов A и S с одновременным переходом их в следующие состояния, что в автомате $\mathcal{P}$ соответствует переходу в состояние 110000. На рис. 5.6 показан начальный фрагмент графа переходов $\mathcal{P}$.

Автомат $\mathcal{P}$ можно рассматривать как структуру Крипке. Некоторые возможные вычисления (последовательности состояний) системы переходов $\mathcal{P}$:

$$000000 \to 110000 \to 121000 \to 120010 \to 110010 \to 110021 \to 121131 \to ...$$

$$000000 \to 110000 \to 121000 \to 120000 \to 110000 \to 121000 \to 111000 \to 110010 \to ...$$

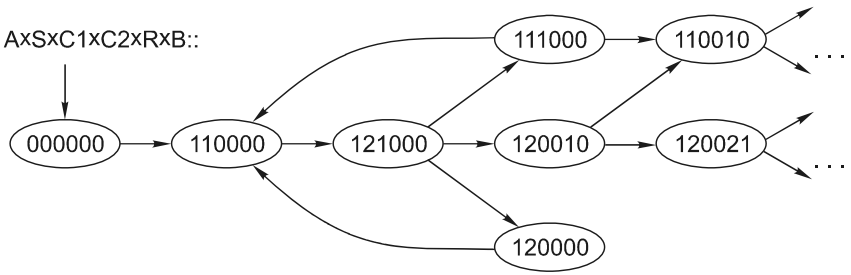


Рис. 5.6. Начальный фрагмент графа переходов протокола PAR

Атомарные предикаты системы переходов $\mathcal{P}$, рассматриваемой как структура Крипке, определяются теми свойствами, которые мы хотим проверить в протоколе PAR. Спецификация правильного поведения протокола вытекает из его задачи:

Spec1: *Все сообщения, посланные пользователем A, должны быть получены пользователем B без потерь и дублирования, причем получены в том же порядке, в каком были посланы.*

Spec2: *Если A хочет послать сообщение, он сможет это сделать (может быть, не сразу).*

Для доказательства того, что бесконечный поток сообщений, передаваемый с помощью коммуникационного протокола через ненадежный канал, сохраняет свой порядок, требуется всего три различных элемента данных. Это впервые было показано в [148]. Мы будем использовать здесь следующий прием. Введем следующие атомарные предикаты:

s_k : A послал сообщение, занумерованное k;

r_k : B принял сообщение, занумерованное k.

Поскольку различных нумераций сообщений всего два, 0 и 1, то в нашей структуре Крипке будет четыре атомарных предиката: s_0, s_1, r_0, r_1. Очевидно, что предикат s_0 истинен в состоянии 1 автомата A, s_1 истинен в его состоянии 2. Предикат r_0 истинен в состоянии 1 автомата B, r_1 истинен в его состоянии 2. В каждом состоянии структуры Крипке, моделирующей протокол, какие-то из этих атомарных предикатов будут истинны, другие нет. Приведенное выше вычисление структуры Крипке $\mathcal{P}$ снабдим спецификацией — множеством истинных в каждом состоянии атомарных предикатов:

$$000000\{\ \} \to 110000\{s_0\} \to 121000\{s_0\} \to 120010\{s_0\} \to 110010\{s_0\} \to$$
$$110021\{s_0, r_0\}\ \ldots$$

Для верификации протокола по этому вычислению построим траекторию — цепочку подмножеств атомарных предикатов, истинных в состояниях вычисления: $\{\ \}\{s_0\}\{s_0\}\{s_0\}\{s_0\}\{s_0, r_0\}\ldots$. Именно такие траектории будут проверяться на выполнение требований спецификации.

Формально требования спецификации протокола PAR представим формулами LTL:

Spec1:

$$\mathbf{G}\left(s_0 \Rightarrow \left(\neg s_1 \mathbf{U} r_0\right)\right) \wedge \mathbf{G}\left(r_0 \Rightarrow \left(\neg r_1 \mathbf{U} s_1\right)\right) \wedge \mathbf{G}\left(s_1 \Rightarrow \left(\neg s_0 \mathbf{U} r_1\right)\right) \wedge \mathbf{G}\left(r_1 \Rightarrow \left(\neg r_0 \mathbf{U} s_0\right)\right):$$

"*если A послал сообщение, то пока B не примет его, A не посылает следующее, и если B принял сообщение, то он не принимает никакого другого сообщения, пока A не пошлет следующее*".

Spec2: $\mathbf{G}\left(\left(s_0 \Rightarrow \mathbf{F} s_1\right) \wedge \left(s_1 \Rightarrow \mathbf{F} s_0\right)\right)$:

"*если A послал сообщение, то когда-нибудь в будущем он сможет послать следующее*".

Анализ композиции показывает, что не все вычисления системы переходов $\mathcal{P}$ удовлетворяют спецификации. Рассмотрим, например, вычисление (в скобках указываются атомарные предикаты, истинные в состоянии):

$000000\{\ \}$ → пользователь A посылает сообщение d_0 передатчику S →

$110000\{s_0\}$ → S посылает d_0 в канал $Ch1$ →

$121000\{s_0\}$ → $Ch1$ доставляет d_0 приемнику R →

$120010\{s_0\}$ → "нетерпеливый передатчик" не дождался подтверждения →

$110010\{s_0\}$ → приемник R доставляет d_0 пользователю B →

$110021\{s_0, r_0\} \rightarrow$ "нетерпеливый передатчик" повторно посылает d_0 в канал $Ch1 \rightarrow$

$121021\{s_0, r_0\} \rightarrow$ приемник R посылает в канал $Ch2$ подтверждение приема $d_0 \rightarrow$

$121131\{s_0, r_0\} \rightarrow$ передатчик S получает из $Ch2$ подтверждение приема $d_0 \rightarrow$

$131031\{s_0, r_0\} \rightarrow$ канал $Ch1$ повторно доставляет d_0 приемнику $R \rightarrow$

$130021\{s_0, r_0\} \rightarrow$ передатчик S принимает от A новые данные $d_1 \rightarrow$

$240021\{s_1, r_0\} \rightarrow$ S передает эти данные в канал $Ch1 \rightarrow$

$252021\{s_1, r_0\} \rightarrow$ R передает в $Ch2$ подтверждение повторного получения $d_0 \rightarrow$

$252131\{s_1, r_0\} \rightarrow$ канал $Ch1$ теряет сообщение $d_1 \rightarrow$

$250131\{s_1, r_0\} \rightarrow$ S получает подтверждение от *Ch2* (считает, что d_1 дошло) $\rightarrow$

$200031\{s_1, r_0\} \rightarrow$ S принимает от A новые данные $d_0 \rightarrow$

$110031\{s_0, r_0\} \rightarrow$ S посылает d_0 в канал $Ch1$; здесь нарушение свойства $\mathbf{G}\left(s_1 \Rightarrow \left(\neg s_0 \mathbf{U} r_1\right)\right) \rightarrow$

$121031\{s_0, r_0\} \rightarrow$ *Ch1* доставляет d_0 приемнику $R \rightarrow$

$120021\{s_0, r_0\} \rightarrow$ R ожидал d_1, он посылает в $Ch2$ подтверждение получения $d_0 \rightarrow$

$120131\{s_0, r_0\} \rightarrow$ S получает подтверждение доставки d_0, готов принять $d_1 \rightarrow$

$130031\{s_0, r_0\} \rightarrow$ A посылает передатчику S четвертое сообщение, $d_1 \rightarrow$

$240031\{s_1, r_0\}$ — ...

В результате A послал три сообщения, два из них пропали. Эта пропажа не обнаруживается протоколом, но выявляется по невыполнению спецификации (требуемого свойства Spec1). Читателю рекомендуется тщательно проверить это и другие вычисления протокола, чтобы убедиться, насколько сложными и

непредсказуемыми могут быть последовательности событий в простой параллельной системе процессов, даже если в ней поведение каждой компоненты предельно понятно.

Ручное построение асинхронного произведения автоматов для подобных систем чрезвычайно трудоемко. Как видно из примера протокола, в асинхронном произведении систем переходов рост числа состояний экспоненциальный, что означает следующее: добавление к системе каждого нового компонента с N состояниями увеличивает число состояний в асинхронной композиции в N раз. Этот эффект называется "*взрывом числа состояний*" (state explosion problem), и он составляет одну из главных проблем верификации. В реальных системах количество возможных состояний достигает астрономических величин — 10^{200} и более [108].

Процесс построения асинхронной композиции систем переходов можно автоматизировать, т. е. инструментальному пакету верификации задавать верифицируемую модель не в виде структуры Крипке, а как набор отдельных взаимодействующих модулей. Операцию асинхронной композиции этих модулей с получением структуры Крипке обычно выполняет сам инструмент верификации до того, как выполнить верификацию.

5.4. Представление программы с конечным числом состояний структурой Крипке

Известно, что многие синхронизационные и коммуникационные протоколы могут быть достаточно точно смоделированы программами с конечным числом состояний, в которых каждое состояние имеет конечное описание. Вследствие этого они могут быть адекватно представлены структурой Крипке.

В некоторых случаях программу также удобно задавать визуально как систему переходов. Существует несколько областей, где такой вид задания алгоритма активно используется: системы управления, алгоритмы поведения активных агентов, драйверы внешних устройств и т. п. В частности, диаграммы активности языка UML (Unified Modeling Language) — это, фактически, модифицированные карты состояний, т. е. системы переходов.

Программы, написанные в традиционном стиле на обычном языке высокого уровня, тоже, конечно, характеризуются своим состоянием. Например, в случае прерывания программы ее полное состояние запоминается для того, чтобы в последующем продолжить ее выполнение. Состояние здесь — это вся информация, необходимая для корректного продолжения выполнения про-

граммы. Состояние программы составляют значения всех ее переменных (структур данных), содержимое регистров, содержимое стека. Кроме того, к состоянию относится и текущая позиция в коде программы (значение счетчика команд). Для параллельной программы с взаимодействием модулей в модель необходимо добавить состояния буферов каналов (значения сообщений в них).

Переходы между состояниями программы происходят при выполнении операторов, изменяющих значения переменных. Рассмотрим случай, когда каждая переменная программы может принимать конечное множество значений, а каналы могут хранить конечное число сообщений. В этом случае программу можно описать конечной системой переходов. В литературе такие программы называются *finite-state programs* — программы с конечным числом состояний.

Пример 5.2

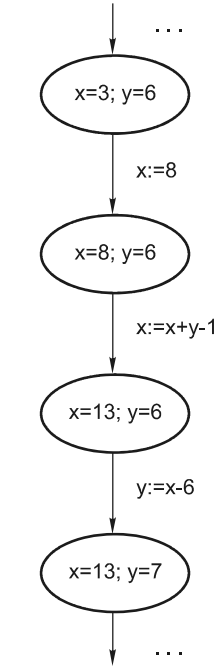
Рассмотрим простой пример:

```
...
  x := 8;
  x := x+y-1;
  y := x-6;
...
```

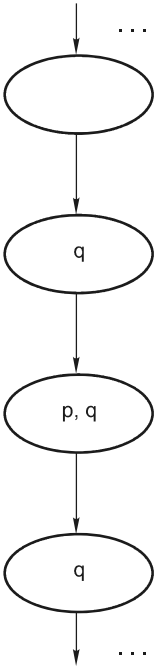
Пусть значения переменных до выполнения этого фрагмента таковы: `x=3`; `y=6` (начальное состояние фрагмента — пара ⟨`x=3, y=6`⟩). Первый оператор `x:=8` переводит это состояние в состояние ⟨`x=8, y=6`⟩, второй оператор `x:=x+y-1` — в состояние ⟨`x=13, y=6`⟩ и т. д. На рис. 5.7, *а* представлен фрагмент системы переходов для этой программы. Для того чтобы из этой системы переходов получить структуру Крипке для верификации, нужно отбросить пометки на переходах и в состояния добавить атомарные предикаты, истинные в этих состояниях.

Откуда берутся атомарные предикаты?

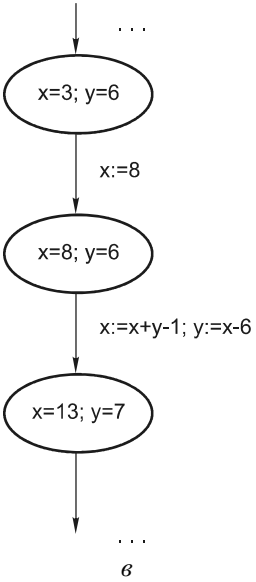
Атомарные предикаты для программы — это логические соотношения, интересующие разработчика с точки зрения поведения программы. Они назначаются программистом при формулировке спецификаций (требований к поведению программы). Пусть, например, для нас важными являются следующие соотношения в этом фрагменте: `p=x>2*y` и `q=y<x`. Следующий фрагмент структуры Крипке соответствует этой программе (рис. 5.7, *б*). Фрагментом траектории, соответствующей нашему фрагменту программы, будет цепочка подмножеств атомарных предикатов: ... $\{\ \}\{q\}\{p,q\}\{q\}\ldots$.



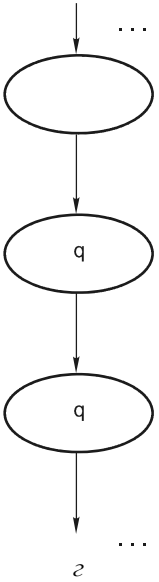
а



б



в



г

Рис. 5.7. Системы переходов и структуры Крипке фрагментов программ

Запишем теперь нашу программу в следующем виде:

```
...
  x := 8;
  x := x+y-1; y := x-6;
...
```

Этот фрагмент отличается от предыдущего тем, что в одной строке стоят два оператора (рис. 5.7, *в*). Адекватна ли соответствующая этой программе структура Крипке (рис. 5.7, *г*)?

Подобные вопросы можно продолжить. Является ли неделимым сложный оператор `x:=(y+4)mod2`, оператор `x:=x+y-1`, оператор `x++`, оператор условного присваивания `max:= a>b? a : b, ...` ? Или модель должна явно представлять операции с регистрами?

Вопрос не совсем глупый. Сформулированный по-другому, вопрос этот звучит так: насколько сложными могут (должны) быть операторы, определяющие переход в модели программы и изменяющие состояния модели, чтобы модель адекватно отражала поведение программы? Ответ на него такой: *переходы в модели должны представлять те изменения состояний программы, которые, как это было сказано в разделе 5.1, являются "неделимыми", мгновенными с точки зрения внешних процессов, внешней среды.* Например, если `x` — разделяемая переменная, то переход, помеченный оператором `x:=x+y-1` на рис. 5.7, *а* не является неделимым (см. примеры некорректных программ в *гл. 1*).

Неделимыми могут быть операции только относительно другого процесса или внешнего наблюдателя. Поэтому если программа в некотором фрагменте вообще не взаимодействует со средой (т. е. к ее переменным нет обращения извне, программа не посылает и не принимает сообщения, не вводит и не выводит значения и не работает с разделяемыми переменными), то весь фрагмент, каким бы он ни был длинным и долгим в выполнении, может быть представлен в модели одним неделимым переходом.

Проблемы, возникающие при представлении с помощью структуры Крипке параллельно функционирующих программ, рассмотрим в следующем разделе.

5.5. Представление параллельных программ структурой Крипке

Параллельные программы состоят из нескольких последовательных модулей (процессов), каждый из которых функционирует независимо, при необходи-

мости (для решения общей задачи) взаимодействуя с другими процессами через общие переменные или посылкой сообщений по каналам связи.

Рассмотрим два процесса, P_1 и P_2, параллельно и независимо выполняющие последовательности простых операций присваивания:

```
P1::                    P2::
  k0: x:= 3;              n0: y:= 2;
  k1: x:= y;              n1: y:= x ;
  k2: end                 n2: end
```

С какими значениями переменных `x` и `y` закончится выполнение этой программы, если начальные значения `x` и `y` равны 0? Подобный анализ проводится в примере 5.3, но там мы не рассматриваем состояний процессов. Несмотря на элементарность этих процессов, ответить на этот вопрос не так просто.

Анализ этой параллельной программы удобно выполнить с помощью модели. Модель параллельной программы $P_1 \| P_2$ будем строить как асинхронную композицию моделей процессов P_1 и P_2. Пусть состояние $P_1 \| P_2$ состоит из четверок значений $\langle x, y, pc_1, pc_2 \rangle$. Каждая операция этих процессов выполняется процессорами неделимо. Поскольку мы не можем делать никаких предположений о скоростях выполнения программ, естественно считать, что в каждом глобальном состоянии любая из программ может выполнить свой шаг — очередную операцию.

Рис. 5.8 показывает возможные варианты развития вычислений. По окончании параллельной программы переменные `x` и `y` могут принять три разные значения. Какой вариант реализуется при реальном запуске программ, зависит от трудно учитываемых соотношений скоростей процессов. Большинство ошибок в разрабатываемых параллельных программах как раз и являются следствием непредвиденных перекрытий операций процессов. В частности, ошибки, обнаруженные в системе управления аппаратом DeepSpace1 с помощью верификации методом model checking, относятся именно к этому типу.

Рассмотрим теперь знаменитую классическую проблему взаимного исключения в параллельном программировании. Заметим, что англоязычный термин этой проблемы — *mutual exclusion* — почти прижился в русскоязычной среде программистов под именем *мьютекс*.

Программы в своей работе могут использовать общий разделяемый ресурс, например принтер или структуру данных (файл для записи в него новой информации). Фрагмент программы, в котором этот общий ресурс используется, называется критической секцией (или критическим интервалом, *crint*). Очевидно, что не более чем одна программа может выполнять команды в

своей критической секции, иначе, например, в печатаемом документе могут появляться строки, посылаемые на печать другой программой, если критическая секция — это обращение к печати на общем принтере. Так в мультфильме "Каникулы в Простоквашино" в письме дяди Федора появились строки о лохматости.

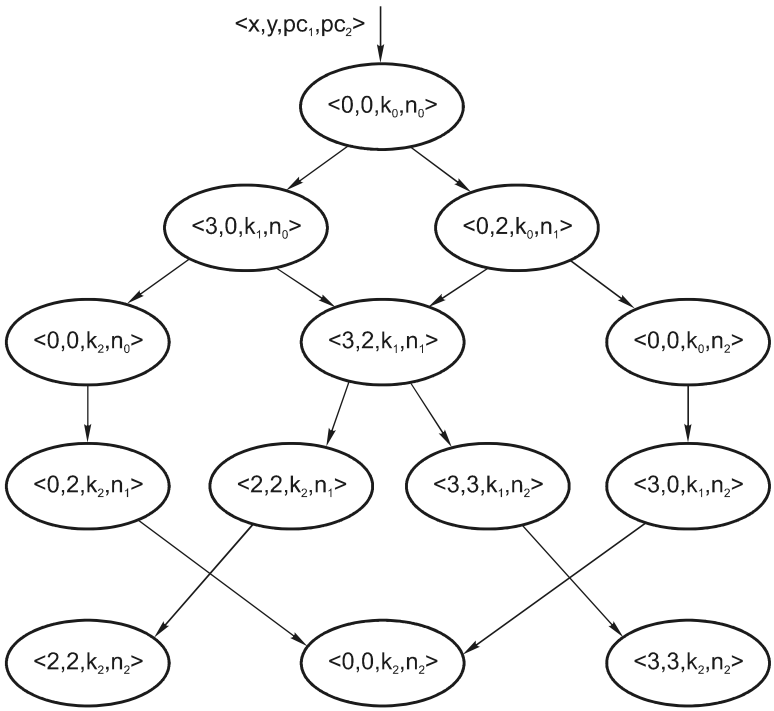


Рис. 5.8. Возможные результаты выполнения простой параллельной программы

Для выполнения требования взаимного исключения программы как-то должны синхронизироваться, т. е. учитывать, что делают другие программы. Один из вариантов решения проблемы взаимного исключения представлен ниже.

Пример 5.3

Два параллельно функционирующих процесса P_1 и P_2 имеют свои критические секции C_1 и C_2. Действия NC_1 и NC_2 — это операции, находящиеся в некритических секциях процессов. Синхронизация процессов осуществляется с помощью общей разделяемой переменной `f`, значение 1 которой определяет, что в критической секции не находится ни одного процесса:

```
P1::
  m1: NC1
  m2: if(f!=1) goto m2;
  m3: f:=0;
  m4: C1;
  m5: f:=f+1;
  m6: goto m1;
```

```
P2::
  n1: NC2
  n2: if(f!=1) goto n2;
  n3: f:=0;
  n4: C2;
  n5: f:=f+1;
  n6: goto n1;
```

Рассмотрим работу процесса P_1. Вначале значение `f` равно 1. После выполнения в строке с меткой m_1 операторов некритической секции NC_1, в следующей строке m_2 процесс проверяет значение `f`. Если другой процесс находится в своей критической секции (это показывает значение `f`, равное 0), то процесс P_1 ждет освобождения критической секции, выполняя так называемое "активное ожидание" — передает управление опять на проверку `f`, бессмысленно загружая процессор повторением пары операций (проверка, передача управления). Если вход в критическую секцию разрешен (т. е. `f=1`), то процесс P_1 входит в свою критическую секцию (строка с меткой m_4), предварительно установив `f` равным 0 в строке с меткой m_3 (чтобы другие процессы не могли войти в свои критические секции). Закончив работу в критической секции, процесс в строке с меткой m_5 восстанавливает значение `f`. Процесс P_2 работает аналогично.

Свойство взаимного исключения, которое должно быть проверено в этой системе параллельных процессов, выражается следующей формулой LTL: $\mathbf{G}\neg(pc_1 = m_4 \ \wedge pc_2 = n_4)$ — состояние, в котором оба процесса одновременно находятся в своих критических секциях, недостижимо в их композиции. Для проверки этого свойства построим модель процесса P_1, считая неделимыми выполнения каждого оператора. Состояния P_1 — это пары значений $\langle pc_1, f \rangle$, т. е. ⟨*значение счетчика команд* P_1, *значение переменной* `f`⟩. Поскольку в начале выполнения операторов процесса P_1 значение `f` может быть уже изменено другим процессом, у этой модели два начальных состояния (рис. 5.9). Всего в системе переходов для P_1 восемь состояний, столько же у P_2.

Модель выполнения процесса $P_1 \| P_2$ строится как асинхронная композиция моделей процессов P_1 и P_2 (*см. раздел 5.3*). Состояние $P_1 \| P_2$ состоит из троек значений $\langle pc_1, pc_2, f \rangle$, т. е. ⟨*значение счетчика команд процесса* P_1, *значение счетчика команд процесса* P_2, *значение переменной* f ⟩. В соответствии с семантикой асинхронной композиции в каждом состоянии компози-

ции может быть выполнено какое-либо *одно* из действий любого процесса, если это действие возможно в текущем состоянии этого процесса.

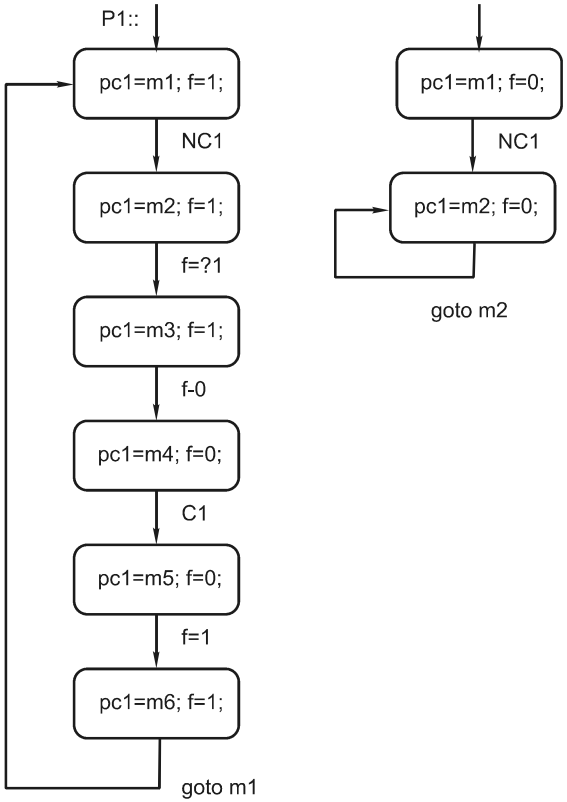


Рис. 5.9. Система переходов программы взаимного исключения

На рис. 5.10 представлен фрагмент системы переходов композиции процессов P_1 и P_2. Всего система переходов для двух программ содержит порядка 40 состояний. При росте числа процессов в параллельной программе общее число глобальных состояний всей системы растет экспоненциально.

На этой системе переходов видно, что состояние, в котором оба процесса находятся в своих критических секциях (т. е. $pc_1 = m_4$ и $pc_2 = n_4$) достижимо из начального состояния. После того как в строке m_2 процессом P_1 проверено, что значение общего флага $f = 1$, но до того, как процесс P_1 установит значение f в 0, другой процесс может проверить общую переменную f и также обнаружить, что ее значение разрешает ему войти в свою критическую секцию. Очевидно, что такие ситуации чрезвычайно редки, а их вероятность

исчезающе мала. Именно это создает огромные трудности в обнаружении подобных ошибок в параллельных системах, как уже было показано в *гл 1*. Верификация — это единственный метод, позволяющий обнаруживать редкие ошибки такого сорта.

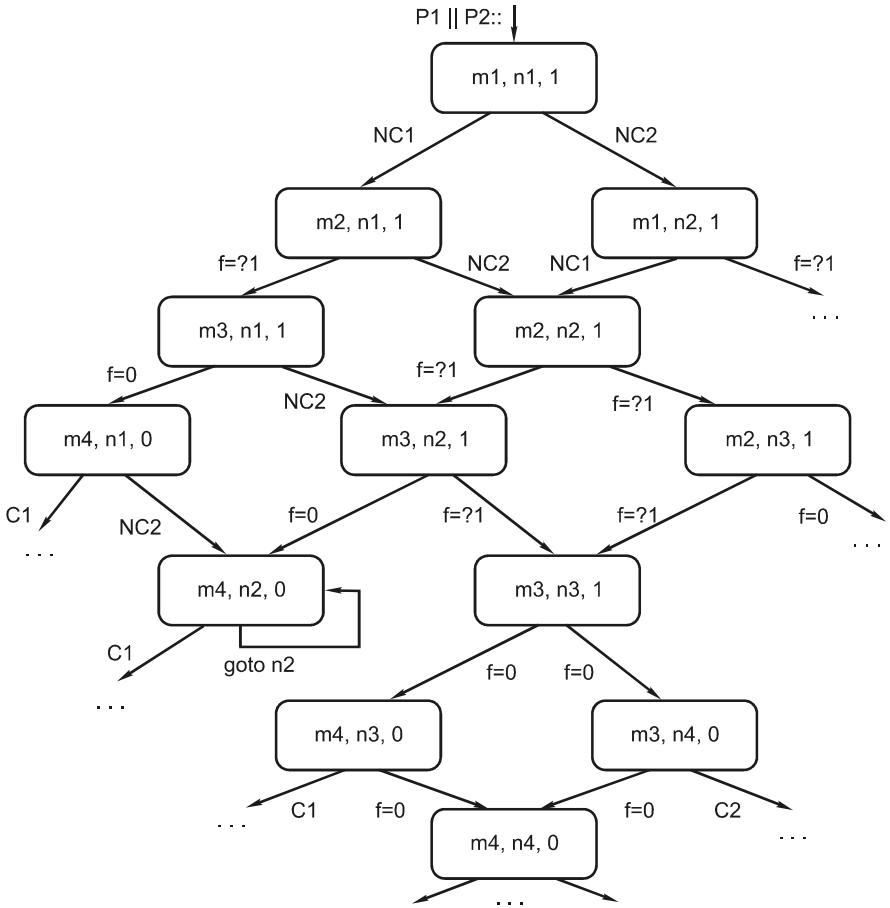


Рис. 5.10. Система переходов параллельной программы P1 || P2

Причина того, что приведенный выше алгоритм не решает проблему взаимного исключения, состоит в том, что между операциями ПРОВЕРИТЬ f (*равно ли f единице?*) и УСТАНОВИТЬ f ($f := 0$) другой процесс может проверить состояние общей переменной $f := 0$. Таким образом, для защиты критического интервала с помощью общей переменной нам необходима организация критического интервала для работы процесса с этой переменной.

Выход из этого порочного круга состоит в том, чтобы сделать выполнение пары операций ПРОВЕРИТЬ и УСТАНОВИТЬ мгновенным. Вспомним, однако, что мгновенных операций нет, да нам, в действительности, нужна не мгновенная, а только НЕДЕЛИМАЯ операция, мгновенная только *с точки зрения* других активностей системы. Иными словами, необходимо сделать так, чтобы любой процесс мог наблюдать (и изменять) состояние этой переменной только либо до, либо после того, как операция <ПРОВЕРИТЬ и УСТАНОВИТЬ> выполняется с этой переменной каким-либо другим процессом.

Пример 5.4

Включим в состав возможных операций неделимую операцию `wait_if(f=1)f:=f-1` со следующей семантикой: если значение `f` равно 1, то вычесть 1 из `f`, а если нет — то ничего не делать и ждать. Тогда новое решение проблемы взаимного исключения может быть представлено так:

```
P1::
  m1: NC1
  m2: wait_if(f=1) f:=f-1;
  m3: C1;
  m4: f:=f+1;
  m5: goto m1;
```

```
P2::
  n1: NC2
  n2: wait_if(f=1) f:=f-1;
  n3: C2;
  n4: f:=f+1;
  n5: goto n1;
```

На рис. 5.11 представлена система переходов процесса P_1, а на рис. 5.12 — система переходов параллельной композиции процессов $P_1 \| P_2$.

Из рис. 5.12 видно, что проблема взаимного исключения теперь решена корректно: среди достижимых состояний нет состояния, в котором $pc_1 = m_3$ и $pc_2 = n_3$.

Тип переменных, над которыми определены две неделимые операции:

$P(s)$ — уменьшения значения s на 1, если $s > 0$, и ждать, если $s \le 0$,

$V(s)$ — увеличение значения переменной s на 1.

называется семафором. Семафоры были введены Эдсгером Дейкстрой (Edsger Dijkstra) для решения проблем синхронизации параллельных процессов. Любой фрагмент процесса можно сделать атомарным, неделимым с точки зрения всех остальных процессов, параллельно выполняемых с данным, если поместить его между операциями P и V, выполняемыми над некоторым семафором. Заметим, что при этом решается и проблема активного ожидания — бесполезной траты времени процессора на передачи управления, если условие не выполняется. В модели весь этот фрагмент можно представить одним переходом.

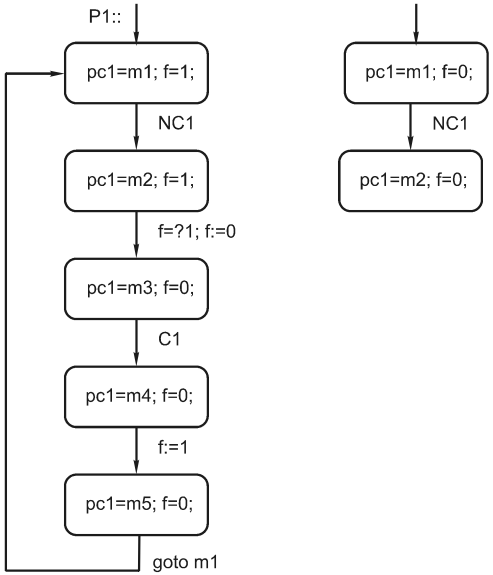


Рис. 5.11. Система переходов одного из процессов программы взаимного исключения

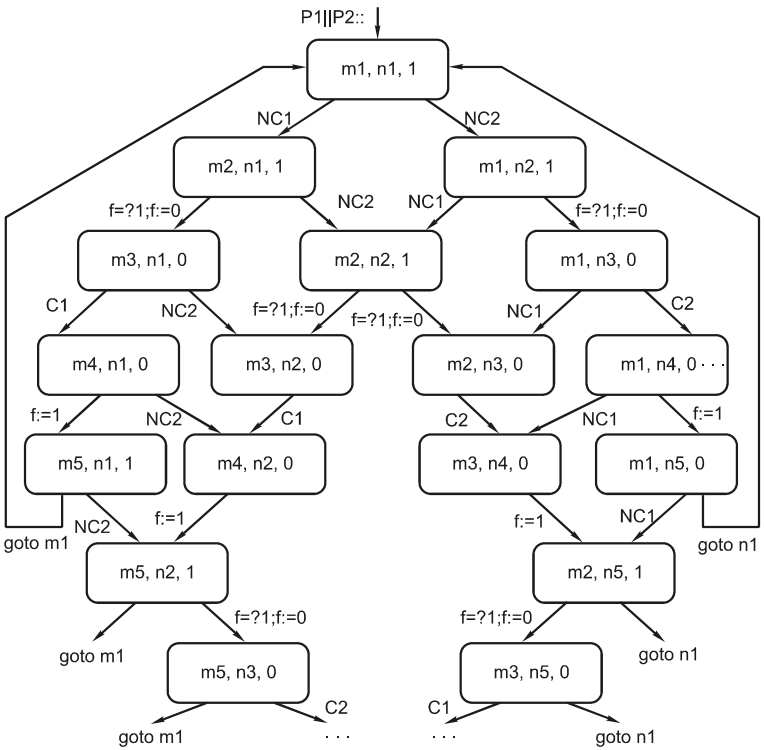


Рис. 5.12. Система переходов корректной программы взаимного исключения

5.6. Пакеты верификации и структура Крипке. Пакет верификации Spin

Пакеты верификации. Интенсивные исследования в области model checking привели к тому, что разработанная к настоящему времени техника применения этого подхода для верификации намного проще, чем его теоретический базис. Современные пакеты верификации автоматизируют все этапы верификации, в том числе они освобождают пользователя и от необходимости вручную строить структуру Крипке верифицируемой системы. Модель верифицируемой системы описывается на входном языке пакета, который представляет собой, фактически, язык программирования высокого уровня, удобный для описания множества параллельных взаимодействующих процессов, каналов их взаимодействия и работы с простыми типами данных. С помощью алгоритмов, описанных выше, пакет верификации автоматически построит структуру Крипке (систему переходов) каждого из компонентных модулей заданной системы, построит их композицию и автоматически выполнит процесс верификации — проверку выполнимости заданной темпоральной формулы на структуре Крипке. Одним из таких пакетов является Spin.

Пакет верификации Spin. Spin — это система верификации, которая поддерживает разработку и анализ корректности параллельных и распределенных систем с конечным числом состояний, спецификация свойств которых представлена формулами LTL. Основная цель использования пакета Spin — это проверка корректности взаимодействующих параллельных асинхронных процессов (рис. 5.13). Spin фокусируется именно на асинхронной модели, которая естественна для программных систем. Система Spin предоставляет пользователю:

1. Язык Promela (Protocol Meta Language) — C-подобный язык для спецификации моделей.
2. Удобные средства для выражения требований корректности формулами линейной темпоральной логики (без оператора **X** (Next Time)).

Аббревиатура Spin расшифровывается как *Simple Promela Interpreter*, т. е. простой интерпретатор систем, специфицированных на языке представления Promela. Цель языка Promela — дать возможность пользователю построить модель параллельной системы для последующей проверки в ее поведении аспектов координации и взаимодействия параллельных процессов. Поэтому в языке только три типа объектов спецификации:

- процессы;
- каналы, по которым процессы взаимодействуют;
- переменные простых типов.

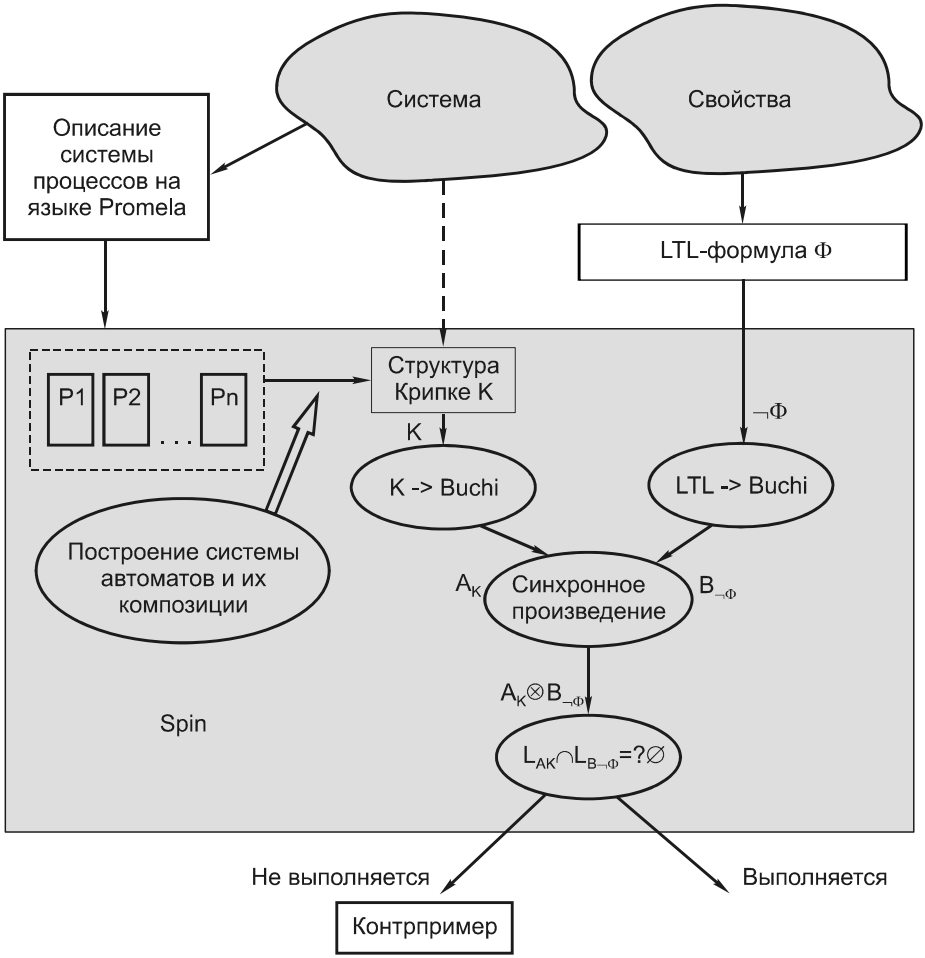


Рис. 5.13. Пакет верификации Spin

Поскольку задача нашего изложения — не изучение этого специфического языка, мы приведем здесь очень краткое введение в Promela. Цель языка Promela — предоставить программисту максимально удобные средства для описания таких программ, которые могут быть проверены, т. е. могут быть автоматически оттранслированы в представление структуры Крипке. Одним из главных ограничений поэтому является конечность числа состояний специфицируемых процессов. При этом число возможных вычислений программ, специфицированных на языке Promela, обычно бесконечно.

В параллельной программе на Promela конечное число процессов, переменные имеют конечное число возможных значений и в каналах возможно только конечное число передаваемых сообщений. Поэтому каждый процесс мо-

жет быть легко представлен моделью переходов, а по моделям переходов всех процессов может быть построена полная модель функционирования параллельной программы как асинхронное произведение компонентных моделей переходов. Такое представление выполняется автоматически системой Spin при верификации системы.

Описание системы параллельных процессов на Promela состоит из одного или более описаний типов процессов, и процесса запуска, который инициирует конечное число экземпляров процессов. Типы процессов определяют различные типы поведения взаимодействующих активностей.

Архитектура представляемых на языке Promela систем проста: это конечное число асинхронных последовательных процессов, функционирующих параллельно и взаимодействующих друг с другом. Взаимодействие процессов может быть специфицировано с помощью синхронного взаимодействия (рандеву), асинхронного взаимодействия (с буферизацией передаваемых сообщений) и с помощью доступа к общим разделяемым переменным.

Команды языка похожи на операторы языка программирования C. Команды либо выполняются, либо блокируются. Блокировка команд происходит при взаимодействии рандеву с неготовым к взаимодействию процессом, при посылке сообщения в канал, в буфере которого уже находится максимальное число допустимых сообщений, при проверке условия, которое оказалось ложным и т. п.

Допускаются переменные только с конечной областью определения: `bit`, `bool`, `byte` (8 бит), `short` (со знаком, 16 бит), `int` (со знаком, 32 бит). Можно строить одномерные массивы данных фиксированного размера. Кроме того, можно использовать записи.

В языке Promela неделимыми считаются отдельные операторы, например, операторы посылки и получения сообщения, оператор присваивания переменной выражения, каким бы сложным это выражение ни было. При необходимости сделать атомарными несколько последовательных операторов, используется ключевое слово `atomic`. Конструкция `atomic {statements}` позволяет определить всю последовательность операторов `statements` как выполняемую за один неделимый шаг. Таким образом, уровень неделимости определяется разработчиком модели. Если конструкция `atomic` отсутствует, то неделимыми считаются элементарные операторы программы.

Например, описание процесса P_1 предыдущего раздела будет выглядеть так:

```
proctype P1()
 { bool NC1, C1 = false;
   m1: NC1 = true;      /* процесс находится в некритической секции */
   m2: NC1 = false; atomic {f==1; f=f-1};
```

```
    m3: C1 = true;      /* процесс находится в критической секции */
    m4: C1 = false; f=f+1;
    m5: goto m1
   }
```

Введение атомарности для последовательности операторов может существенно уменьшить число состояний результирующей модели Крипке, поскольку уменьшается степень интерливинга при построении асинхронной композиции параллельных процессов. Для уменьшения перебора в процессе верификации следует объявлять атомарными все последовательности операторов с локальными переменными. Если в реальной системе какой-либо оператор, который оперирует с общими переменными, состоит из нескольких неделимых операторов (например, вычисление выражения в стековой машине), то сам разработчик должен это учесть при построении модели: гарантии адекватности верифицируемой модели реальной системе — это забота разработчика.

Управляющие структуры языка Promela — это так называемые защищенные команды (guarded commands), введенные Э. Дейкстрой. Таких команд две: выбор и цикл. Условный выбор `if` выполняет любую из команд `Si`, у которой истинна защита `Gi`. Оператор `if` блокируется до тех пор, пока хотя бы одна из защит не станет истинной:

```
if
  :: G1→S1
  :: G2→S2
  ...
fi
```

Оператор `do` выполняется бесконечно, если хотя бы одна защита истинна:

```
do
  :: G1→S1
  :: G2→S2
  ...
od
```

Если все защиты ложны, то оператор цикла блокируется. Если в качестве защиты используется ключевое слово `else`, то эта команда выбирается только в том случае, когда все другие защиты ложны.

Типичная последовательность работы с системой верификации Spin:

- спецификация модели распределенного алгоритма или параллельной системы на языке Promela;

- трансляция и исправление синтаксических ошибок транслятором с языка Promela;
- запуск интерактивного симулятора, который позволяет убедиться, что построенная модель действительно ведет себя так, как этого хочет пользователь (это, фактически, тестирование модели);
- верификация модели.

При симуляции Spin проверяет отсутствие блокировок процессов, неспецифицированный прием сообщений, неисполняемый код. На этапе верификации Spin транслирует каждый процессный тип в конечный автомат. Глобальное поведение всей параллельной системы строится как АСИНХРОННОЕ произведение автоматов, по одному автомату на каждый активный экземпляр процесса. В результате поведение всей системы процессов снова задается конечной системой переходов, в которой параллелизм представлен недетерминированным выбором (интерливингом).

В языке Spin есть несколько возможностей для проверки свойств построенной модели системы: конструкция `assert`, конструкция `never` и задание проверяемого свойства формулой LTL.

Оператор `assert` в программе на языке Promela задается с аргументом, который представляет проверяемое логическое условие. Если это условие при активизации оператора `assert` истинно, то выполнение `assert` эквивалентно пустому оператору. Если это условие ложно, верификатор фиксирует ошибку. Таким образом, аргумент оператора `assert` является инвариантом, который мы хотим проверить в программе.

Пример 5.5

Рассмотрим простую параллельную программу:

```
process Inc   =  while true do if x<10 then x := x+1 od
process Dec   =  while true do if x>0  then x := x-1 od
process Reset =  while true do if x=10 then x := 0   od
```

Можно ли утверждать, что всегда при выполнении процессов $0 \le x \le 10$? Проверим это, построив следующую программу на языке Promela:

```
1 x=0;
2
3 proctype Inc() {
4   do :: true -> if :: x<10 -> x = x+1 fi od }
6
7 proctype Dec() {
8   do :: true -> if :: x>0 -> x = x-1 fi od }
9
```

```
10 proctype Reset() {
11   do :: true -> if :: x==10 -> x = 0 fi od }
12
13 proctype Check() {
14   assert ( x>= 0 && x<=10 ) }
15
16 init {
17   atomic{ run Inc();run Dec();run Reset();run Check();
18 }
```

Здесь для верификации параллельной программы мы создаем процесс-монитор `Check()`, работающий параллельно с другими процессами и содержащий единственный оператор `assert`. Логическое условие, являющееся телом этого оператора, будет здесь проверяться в каждом возможном состоянии параллельной программы, поскольку процесс `Check` делает шаг после каждого шага любого другого процесса. После запуска Spin найдет ошибку — нарушение истинности условия, вычисляемого в операторе `assert`. Причина такого поведения программы состоит в том, что операторы `if` в процессах `Inc`, `Dec` и `Reset` выполняются не атомарно. Пусть `x=10`. После того как в процессе `Dec`, например, будет выполнена проверка защиты `x>0`, из-за асинхронности процессов в параллельной программе могут быть выполнены шаги процесса `Reset`, который сбросит `x` в `0`, предварительно проверив, что `x=10`.

Второй возможностью для верификации свойств систем является конструкция `never` (никогда). В конструкции `never` описываются неправильные, недопустимые траектории поведения проверяемой системы. Функция `never` задает автомат Бюхи, т. е. контрольный автомат, который работает СИНХРОННО с проверяемой системой, и если этот автомат приходит в заключительное состояние (т. е. конструкция `never` завершается), то в системе существует неправильное вычисление (см. рис. 4.2).

Например, для того чтобы проверить, что на всех вычислениях проверяемой системы *выполнится* свойство $\mathbf{GF}(x>3)$, программист может построить следующий процесс `never`, переходящий в принимающее состояние, если на каком-нибудь вычислении выполнится формула $\mathbf{FG}(x\le 3)$, которая является отрицанием формулы $\mathbf{GF}(x>3)$:

```
1 never {     /* FG(x≤3)   */
3 To_init:
4    if
5      :: (x≤3) -> goto accept
6      :: true  -> goto To_init
```

```
 7    fi;
 8 accept:
 9    if
10      :: (x≤3) -> goto accept
11    fi;
12 }
```

Spin может также проверять свойства поведения системы процессов, выраженные формулой линейной темпоральной логики. Синтаксис системы Spin позволяет записывать темпоральные операторы LTL так: **G** записывается парой квадратных скобок [], **F** — парой угловых скобок <>, оператор Until записывается обычным образом U.

Для того чтобы выполнить такую верификацию, Spin по введенной формуле LTL строит ее отрицание, которое описывает нежелательные поведения системы. По отрицанию формулы Spin строит автомат Бюхи (*см. гл. 4*), и по нему строит процесс never, представляющий поведение этого автомата. Выполнение процесса never совместно с другими процессами выполняет поведение СИНХРОННОГО произведения этого автомата и автомата, представляющего глобальное пространство состояний системы точно так, как это описано в *гл. 4*. Если язык, принимаемый этим автоматом, пуст (т. е. процесс `never` не завершится или не проходит бесконечный цикл с особой меткой `accept`), то исходное требование удовлетворяется — нежелательных поведений у анализируемой системы процессов нет. Если язык непустой, то он содержит поведения, которые в системе нежелательны. Это будет найдено в случае, если процесс `never` завершится (т. е. придет в свое завершающее состояние и в нем останется или будет проходить цикл с меткой `accept`). Выходом системы верификации является или утверждение, что исходное требование удовлетворяется или, если оно не удовлетворяется, Spin выдает примеры нежелательных поведений: запускается интерактивный симулятор, который позволяет визуально представить последовательность шагов выполнения алгоритмов модулей и посылаемых при этом сообщений, приводящих к ошибочной траектории поведения (см. рис. 5.12).

Модели, представленные на языке Promela, всегда ограниченные. Поэтому все свойства корректности автоматически становятся формально разрешимыми внутри ограничений, которые установлены размером программы и вычислительными ресурсами, доступными верификатору. Очевидно, что все системы верификации имеют физические ограничения, которые определяются объемом памяти, скоростью процессора и т. п. В системе Spin используются несколько приемов, позволяющих бороться с проблемой "взрыва числа состояний" и увеличить размер тех систем, для которых возможен исчерпывающий анализ.

Число состояний системы переходов растет экспоненциально с ростом числа процессов и ростом числа переменных. Существуют различные методы борьбы с этой проблемой. Один из них — это использование эффективных методов представления данных с помощью Бинарных решающих диаграмм (BDD) и методов символьной верификации, о чем пойдет речь в *гл. 9* и *10*. Другое решение используется в системе Spin: здесь полного преобразования системы процессов в структуру Крипке не производится: проверка выполнения свойств системы производится "на лету" (on-the-fly), только для части построенной системы переходов, которая постепенно достраивается в ограниченной области памяти.

Другой метод уменьшения числа состояний асинхронной композиции моделей при их верификации — это так называемая "редукция частичных порядков" (partial order reduction). Редукция частичных порядков состоит в том, что при верификации часть всех возможных путей (всех полных порядков частично упорядоченного множества операторов параллельной системы) отбрасывается. Например, интерливинг (перекрытие) операторов, работающих с локальными данными в каждом процессе, можно моделировать только одной последовательностью их выполнения: все остальные перестановки выполнения этих операторов будут эквивалентны с точки зрения любого проверяемого свойства. Этот прием позволяет системе верификации Spin существенно снижать сложность проверяемых моделей параллельных систем.

5.7. Построение структуры Крипке для программ высокого уровня

В общем случае, программы не являются системами с конечным числом состояний. Вещественные переменные, динамические структуры данных, динамическое создание параллельных потоков, указатели — все это требует потенциально бесконечного числа состояний. Поэтому корректность программ, написанных на языках высокого уровня, в общем случае доказать невозможно. Например, такая частная проблема верификации, как проблема останова, является алгоритмически неразрешимой.

Для верификации программ необходимо построение конечной модели системы с помощью процесса абстракции. В такой модели некоторые свойства программы, содержащей параметры с бесконечной областью определения, не сохраняются. Очевидно поэтому, что не все реальные свойства программ могут быть проверены с помощью метода model checking. Построенная для верификации модель должна быть достаточно мала, чтобы ее можно было анализировать с помощью систем верификации, но в то же время достаточно содержательна, чтобы отразить существенные свойства поведения программы.

С другой стороны, в модели всегда появляются свойства, которых нет в исходной программе. Абстрагирование обычно выполняется человеком, этот процесс требует опыта, креативности, понимания сути проблемы, которая решается с помощью алгоритма.

В последнее время ведется интенсивная работа по созданию методологии последовательного уточняющего процесса абстрагирования, который может быть автоматизирован. Например, в работе [34] описан метод автоматического построения по реальной программе S такой ее аппроксимации $A_U(S)$ (upper approximation), которая допускает большее количество поведений, чем реальная программа. Если любое свойство программы S, выраженное формулой ψ некоторого ограничения темпоральной логики, выполняется для этой аппроксимации A_U, то это свойство ψ выполняется и для исходной программы S, потому что все поведения S присутствуют и в A_U. Если же свойство ψ не выполняется на абстрактной модели, и выдается контрпример, то проверкой контрпримера на программе S можно получить один из двух результатов. Если выданный контрпример представляет ошибочную траекторию поведения программы S, т. е. проверяемое свойство не выполняется для исходной программы, то необходимо эту программу исправить. Если же выданный контрпример представляет поведение, которого не существует в исходной программе, то в этом случае необходимо построить более точную (менее абстрактную) аппроксимацию программы, чтобы это поведение в построенной аппроксимации не присутствовало. Построение более точной аппроксимации также можно выполнить автоматически и для нее повторить анализ.

Подобный подход используется в нескольких опытных системах верификации, которые могут автоматически строить абстрактное представление программ непосредственно из их записи на языках высокого уровня для последующего анализа этого абстрактного представления. Рассмотрим несколько примеров.

YASM (Yet Another Software Model checker) — академический пакет верификации, разработанный в университете г. Торонто. Он позволяет проверять свойства, выраженные в логике CTL у программ, написанных на C. Программы могут включать и рекурсивные процедуры. Верификатор использует автоматическую технику абстракции.

JavaPathFinder (JPF). Эта система была разработана в исследовательском центре Ames NASA. Система проверяет выполняемый Java-байткод анализируемой программы. Эту систему можно рассматривать как виртуальную Java машину, систематически исследующую все потенциальные траектории вычислений анализируемой программы, написанной на Java. В пакете JPF про-

веряется только ограниченный набор свойств программ — достижимость некоторых состояний, блокировки процессов и необработанные исключения. С 1999 г. система применялась для проверки свойств многих академических и промышленных разработок, и особенно полезным свойством этого пакета разработчики считают нахождение возникающих в параллельных взаимодействующих процессах тонких ошибок, не выявляемых стандартными методами тестирования и отладки.

SLAM — этот пакет разработан фирмой "Майкрософт" для верификации драйверов, написанных на C. С помощью пакета проверяются свойства безопасности, т. е. те, которые могут быть найдены на конечных траекториях программ (*см. гл. 6*). Разработчиками построено около 60 типовых шаблонов — правил — параметрически настраиваемых формул темпоральной логики, которые должны проверяться для вновь разрабатываемых драйверов, работающих под управлением ОС Windows. В этих шаблонах зафиксированы требования к сценариям взаимодействия внешних устройств с ОС Windows.

5.8. Заключение

В абстрактную модель (структуру Крипке) перевод из реальной системы обычно неформален и чаще всего неоднозначен. Структура Крипке — это система переходов с конечным числом состояний, поэтому очевидно, что в структуре Крипке многие свойства системы, содержащей параметры с бесконечной областью определения, не сохраняются. Следовательно, в общем случае не все свойства реальных систем могут быть проверены в такой модели. Поэтому эта техника верификации более удобна в применении к системам, в которых главную роль играет поток управления, а не обработка данных, поскольку обычно обрабатываемые данные принимают значения из бесконечной области определения, и построить адекватную модель таких программ обычно значительно труднее. С помощью техники model checking проверяется не правильность преобразования данных, а *свойства поведения* программ, и с этой точки зрения для большого класса систем возможно построить такую модель (обычно это так называемый *синхронизационный скелет параллельной программы*), чтобы существенные свойства поведения, интересующие разработчика, сохранялись в модели.

При построении модели Крипке M для реальной системы S необходимо решить следующий вопрос: что определяет шаг эволюции системы, т. е. переходы между состояниями в M:

- оператор языка высокого уровня;
- строка операторов;

- строка ассемблера;
- изменение напряжения на входе (выходе) регистра, хранящего значение и т. п.

Ответ на этот вопрос определяется, конечно, целью моделирования: построенная модель должна быть адекватной, т. е. те свойства, которые интересуют разработчика в системе S, должны сохраняться в модели. Если, например, разработчика интересуют проблемы, которые могут возникнуть из-за параллельной работы подсистем, то одним переходом должны быть представлены те шаги динамики системы, которые неделимы с точки зрения остальных процессов. Именно чередование таких шагов определяет свойства поведения в параллельных системах. Следует, конечно, учитывать реальную неделимость операций, поддерживаемую аппаратурой.

Для реальных систем построение соответствующей структуры Крипке часто проводится в два этапа:

- на первом все параметры ограничиваются конечными областями определения;
- на втором для полученной системы структура Крипке строится формально.

К настоящему времени разработано несколько программных пакетов, автоматически выполняющих многие этапы, необходимые при проведении верификации. Пользователь должен только описать параллельную систему на языке программирования достаточно высокого уровня и задать проверяемое свойство на языке темпоральной логики. Пакет сам строит системы переходов для компонентных процессов и выполняет операцию их асинхронной композиции. В данной главе представлен язык Promela, входной язык системы верификации Spin. Этот язык является языком программирования высокого уровня, удобным для описания множества параллельных взаимодействующих процессов, каналов взаимодействия и работы с простыми типами данных. Построение системы переходов для каждого процесса и их композиции для получения структуры Крипке осуществляется на основе этого описания автоматически, без участия программиста.

5.9. Замечания

Метод явного выделения состояний для описания систем логического управления и протоколов коммуникации широко используется много десятилетий. Этот подход давно является общепринятым и для дискретных систем управления. В нашей стране школа академика М. А. Гаврилова использовала явное выделение состояний для синтеза систем управления с начала 60-х гг. ([157]).

В области разработки протоколов Грегор Бохманн (Gregor Bochmann) более 30 лет назад в [22] писал о таком подходе как о "фольклоре", уже тогда широко известном в среде программистов. Амир Пнуэли в 1977 г. выделил так называемые "реагирующие системы" (reactive systems) как специальный класс систем, требующий своих средств для описания функционирования (состояния и переходы) и верификации (проверка логических утверждений об упорядоченности событий). Карты состояний (statecharts) — визуальное представление состояний и переходов систем, которое является расширением графического представления конечных автоматов, были предложены Дэвидом Харелом в [61] для удобного наглядного и строгого представления сложных реагирующих систем. Карты состояний уже много лет являются фактическим стандартом для визуальной спецификации реализаций бортовых и встроенных синхронных систем управления реального времени в авиационной промышленности, на транспорте, в военных применениях и энергетике [62]. Во многих областях применение этой техники состоит в использовании графического редактора для построения визуального представления взаимодействующих компонентов системы, которое является основой для верификации разработки и автоматической генерации кода реализации. Одним из самых известных представителей этого подхода является среда SCADE для разработки управляющих систем [187]. Этот коммерческий продукт фирмы Esterel Technologies с начала 90-х гг. прошлого века широко используется для разработки надежных встроенных систем управления для критических применений в космической и оборонной технике. В нашей стране метод функциональной спецификации систем с помощью явного выделения состояний и переходов был "переоткрыт" в 1991 г. и известен под названием Switch-технологии.

Построение реагирующих систем может производиться так, чтобы модель переходов, расширенная тем или иным способом, могла бы генерироваться из разработки. Например, в работе [56] описывается новый язык программирования IOA (Input/Output Automata) для моделирования и реализации распределенных систем и набор инструментов, упрощающих программирование на этом языке. Программа на языке IOA может быть использована как для реализации реагирующей системы (с помощью соответствующего транслятора), так и для формальной проверки свойств реализации, генерируя систему переходов. Другой подход состоит в том, чтобы модель системы переходов автоматически вычленить из программы на языке высокого уровня, описывающей функционирование реагирующей системы [38].

Приведенный в данной главе анализ протокола PAR выполнен по аналогии с анализом протокола альтернирующего бита, представленном в [20]. Автором системы верификации Spin является Джерард Холзманн (Gerard Holzmann). Система свободно распространяется с 1991 г. С 1995 г. проводятся семинары

SpinWorkshop, в 2008 г. состоялся 15-й семинар [189]. С помощью Spin были верифицированы многие сложные программные системы. В настоящее время Дж. Холзманн является сотрудником NASA (где проблема верификации бортовых систем космической техники чрезвычайно актуальна) и профессором Калифорнийского технологического института. Подробное описание системы верификации Spin вместе с примерами верифицированных распределенных программ и обсуждением техники абстракции и спецификации представлено в [80] и [10]. Практические примеры верификации с помощью этой системы даны в [75]. Монография Дж. Холзманна [79] содержит введение в коммуникационные протоколы и методы их формального анализа с помощью Spin. Последняя версия Spin описана в его новой книге [78]. В работе [63] описана версия Spin для многоядерных процессоров. В 2001 г. система Spin получила премию ACM System Software Award, которой ежегодно награждаются лучшие профессиональные разработки в области программирования. В работе [77] Дж. Холзманн дает методические рекомендации по построению моделей в языке Promela.

Экспериментальная система YASM описана в [60]. Ссылка [186] выводит на домашнюю страницу разработчиков этой системы. Описание возможностей пакета SLAM приведено в [12], где описывается опыт применения этого верификатора более чем для сотни драйверов ОС Windows.

5.10. Задачи к главе 5

5.1. Рассмотрите параллельную программу $P1 \| P2$ с двумя процессами, которые выполняются асинхронно:

```
P1:: { a1: x:=1; b1: x:=2; c1: x:=3 }
P2:: { a2: y:=x; b2: z:=x+y; c2: w:=x }
```

Значение переменной `x` вначале равно 0. Рассмотрите все возможные варианты значений, которые могут иметь переменные `y`, `z` и `w` по завершении программы (после завершения обоих процессов).

5.2. Рассмотрите несколько возможных вычислений по модели протокола PAR.

5.3. Прокомментируйте возможные вычисления протокола PAR, представленные в *разделе 5.3*.

5.4. Измените описание протокола PAR так, чтобы подтверждения несли информацию о номере принятого сообщения, 0 или 1. Проверьте вручную работу этого протокола на вычислениях с потерями сообщений, потерями подтверждений, с "нетерпеливым передатчиком".

5.5. Две параллельных нити (потока) запущены асинхронно. Общая переменная x вначале равна 0. Каждый оператор присваивания вычисляется атомарно. Чему будут по окончании равны значения переменных y и z?

Thread 1: `begin x:=1; x:=2 end`

Thread 2: `begin y:=x; z:=x end`

5.6. Два параллельных процесса P_1 и P_2 выполняют оператор:

```
x :=x+x;
```

при начальном значении x, равном 1. Рассмотреть возможные значения x после завершения процессов в двух случаях: а) каждый оператор выполняется атомарно; б) оператор присваивания транслируется в неделимые операции стековой машины с командами

```
LD a    // загрузка значения из ячейки памяти a
ADD     // безадресная команда сложения двух верхних элементов стека с
        // запоминанием результата в верхушке стека
ST a    // запоминание верхнего значения стека в памяти по адресу a
```

5.7. Рассмотрим параллельную программу P:

```
P = P1 || P2,
```

где:

```
P1:: while true do
        m1: a := x;
        m2: y := 1-a;
        m3: z := a;
       od

P2:: while true do
        n1: b := y;
        n2: z := 1-b;
        n3: x := b;
       od
```

Предположим, что каждый оператор выполняется неделимо (атомарно). Переменные a, b, x, y и z являются целыми с начальными значениями a=b=x=y=1, z=0.

Проверьте для программы P свойство "*если x никогда не установится в 0, то z никогда не установится в 0*", построив структуру Крипке, определив атомарные предикаты и сформулировав проверяемое свойство как формулу темпоральной логики.

5.8. Проанализируйте работу следующей параллельной программы, написанной на языке Promela (пример взят из руководства по этому языку). Здесь неделимым считается каждый оператор. Выполнение оператора задерживается до тех пор, пока он не может быть выполнен. Например, оператор `state = =1` будет задержан до тех пор, пока значение переменной `state` не станет равным 1.

```
byte state = 1;

proctype A()
{       byte tmp;
        (state==1) -> tmp = state; tmp = tmp+1; state = tmp
}

proctype B()
{       byte tmp;
        (state==1) -> tmp = state; tmp = tmp-1; state = tmp
}

init
{       run A(); run B()
}
```

5.9. Проанализируйте работу следующей программы на языке Promela, использующей ключевое слово `atomic` (пример взят из руководства по языку).

```
byte state = 1;

proctype A() {
atomic { (state==1) -> state = state+1 }
}

proctype B() {
atomic { (state==1) -> state = state-1 }
}

init {
run A(); run B()
}
```

5.10. Рассмотрите возможность реализации атомарного выполнения некоторого участка параллельной программы с помощью команд, которые аппаратно реализуются неделимыми в некоторых архитектурах процессоров:

- `Test_and_set(a)`

  (проверить и установить) — операция с одним аргументом: неделимо установить значение аргумента в 1, а предыдущее его значение возвратить как результат операции (реализована, например, в архитектуре i386);

- `Swap(a,b)`

  (атомарный свопинг) — операция с двумя аргументами: неделимо обменять значения двух своих аргументов.

5.11. Проанализируйте корректность алгоритма взаимного исключения Петерсона (Gary Peterson, 1981):

```
P1::
loop forever
  NonCritical 1;
  atomic{ b1:=true; x:=2};
  wait until (x=1 ∨ ¬b2);
  Critical1;
  b1 := false
end loop
```

```
P2::
loop forever
  NonCritical 2;
  atomic{b2:=true; x:=1};
  wait until (x=2 ∨ ¬b1);
  Critical2;
  b2 := false
end loop
```

Проанализируйте работу этого алгоритма в случае, если пара присваиваний

```
bi := true; x := 3-i
```

в каждом процессе P_i будет выполняться неатомарно. Проанализируйте также работу этого алгоритма в случае, если эта пара присваиваний будет выполнена неатомарно, но в другом порядке:

```
x := 3-i; bi := true
```

в каждом процессе P_i.

5.12. Постройте структуру Крипке параллельной композиции двух процессов $P_0 \| P_1$:

```
Pi::
begin
mi:  while true do
nci:   wait (turn :=0);
cri:   turn :=1;
     od;
ni:  end;
```

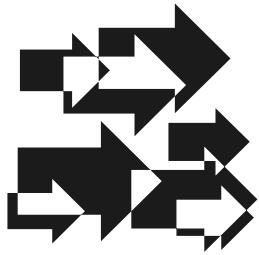
5.13. Постройте структуру Крипке параллельной композиции двух процессов $P_0 \| P_1$:

```
Pi ::  while true do
              m_i1: Noncritical section;
              m_i2: (y_i, s) := (1, i);
              m_i3: wait until ((y_i-1 =0) ∨ (s ≠ i));
              m_i4: Critical section;
              m_i5: y_i := 0
       od
```

Оператор, помеченный m_{i2}, является парой присваиваний, выполняемых атомарно.

Постройте структуру Крипке и проверьте выполнение свойства взаимного исключения.

ГЛАВА 6



Спецификация свойств реагирующих систем формулами темпоральной логики

Подход к верификации на основе метода *model checking* состоит в том, что для модели системы формально проверяется выполнение логической формулы, выражающей некоторое свойство ее "правильного" поведения. Обычно понятие "правильного поведения" включает некоторое множество свойств, которые делают систему полезной для использования, и все эти свойства должны быть проверены последовательно, одно за другим. Возникают вопросы:

- какие свойства следует проверить у системы, чтобы убедиться в ее корректности?
- когда можно прекращать верификацию, т. е. какой набор свойств гарантирует полное отсутствие ошибок в системе?

Невозможно определить формально, что означает "*полное отсутствие ошибок*" в системе. Поэтому верификация так же не может гарантировать полного отсутствия ошибок в анализируемой системе, как и тестирование. Максимум того, что может быть сделано для повышения надежности разрабатываемой системы — это доказать, что система удовлетворяет тем требованиям технического задания, которые разработчик может выразить формально, и не имеет тех ошибок, которые типичны для реагирующих систем, состоящих из параллельных взаимодействующих процессов. Иными словами, при разработке параллельных систем кроме специфичных требований, которым должно удовлетворять поведение данной конкретной системы, следует проверять еще и общие свойства, гарантирующие отсутствие некорректностей, типичных для параллельных систем. Такие свойства традиционно разбиваются на следующие классы:

- свойства достижимости (reachability), которые устанавливают, что некоторые специфические состояния системы могут быть достигнуты;

- свойства безопасности (safety), устанавливающие, что нечто плохое, нежелательное, никогда не произойдет с системой;
- свойства живости (liveness), устанавливающие, что при некоторых условиях нечто "хорошее" в конце концов обязательно произойдет при любом развитии процесса;
- свойства справедливости (fairness), устанавливающие, что нечто будет выполняться неопределенно часто.

Разбиение свойств на классы важно с двух точек зрения. Первая — это взгляд пользователя, который при верификации должен проверить систему на выполнение в ней заданных требований и отсутствие ошибок, характерных для параллельных процессов. Вторая — это взгляд разработчика алгоритмов верификации: различные классы свойств выражаются разными типами формул и, в общем, требуют разных подходов к их проверке. Мы будем здесь рассматривать эти классы с точки зрения пользователя.

В этой главе сначала рассматриваются примеры свойств, которые часто проверяются в технических системах. Затем вводятся и анализируются общие классы свойств — достижимость, безопасность, живость и справедливость.

Отметим еще раз, что выбор множества проверяемых свойств осуществляется человеком — верификатором, никто не может сказать, насколько ПОЛНО осуществлена проверка — все ли нужные свойства системы проверены.

6.1. Примеры спецификации свойств технических систем

Причинно-следственная связь событий

Выражение требований к поведению систем с помощью формул темпоральной логики не всегда является простым и очевидным делом. Рассмотрим, например, задачу формальной спецификации причинно-следственной связи событий, которую обозначим P: "*посылка сообщения send всегда, в конце концов, приведет к получению подтверждения ack* ". Построим формулу линейной темпоральной логики, отражающую это свойство. Рассмотрим несколько вариантов решения.

- $send \Rightarrow ack$

  Очевидно, что это неправильная формализация свойства P, построенная формула имеет следующий смысл: "*если в начальном состоянии send истинно, то в том же состоянии истинно и ack* ".

- $\mathbf{G}send \Rightarrow ack$

  Поскольку темпоральные операторы имеют более высокий приоритет, чем логические связки, эта формула будет пониматься как $(\mathbf{G}send) \Rightarrow ack$, т. е. она имеет следующий смысл: "*если во всех состояниях вычисления выполняется* $send$, *то в начальном состоянии должно выполняться* ack ". Очевидно, что это тоже неверная формула для выражения свойства P .

- $\mathbf{G}(send \Rightarrow ack)$

  Эта формула тоже неверна: она говорит, что в любом состоянии вычисления если выполняется $send$, то *в этом же состоянии* должно выполняться и ack .

- $\mathbf{G}(send \Rightarrow \mathbf{F}ack)$

  Эту формулу можно назвать "темпоральным следствием". Смысл ее в том, что в любом состоянии вычисления если выполняется $send$, то когда-нибудь в будущем (включая настоящее) должно выполняться и ack. Эта формула достаточно точно отражает свойство P, она часто используется для спецификации свойств параллельных систем. В системах верификации часто используется специальный знак для обозначения этого свойства. Например, в инструментальной системе UPPAAL формула $\varphi \to \psi$ означает "φ *всегда ведет к* ψ " и эквивалентна LTL-формуле $\mathbf{G}(\varphi \Rightarrow \mathbf{F}\psi)$ и CTL-формуле $\mathbf{AG}(\varphi \Rightarrow \mathbf{AF}\psi)$. Под названием "*leads-to*" эта формула была введена в [121] и получила там свой знак "~> ".

  Эта формула часто используется для выражения причинно-следственной связи событий, однако она будет истинна и в тех случаях, когда в том же состоянии, в котором истинно $send$, будет истинно и ack. Часто причинно-следственная связь предполагает некоторый временной интервал между причиной и следствием: сообщение после посылки проходит по каналу, приемник его обрабатывает и только *затем* выдает подтверждение. В этих случаях такая формула не является корректной. Здесь нужно искать другое формальное выражение для такой связи.

- $\mathbf{G}(send \Rightarrow \mathbf{XF}ack)$

  Эта формула более точно отражает причинно-следственную связь между событиями $send$ и ack, но она выполняется и в том случае, если событие $send$ вообще никогда не наступит! Если мы, например, строим спецификацию протокола, который построен для многократно выполняющейся операции посылки сообщений, то это следует учесть в формализации.

- $\mathbf{G}\big((send \Rightarrow \mathbf{XF}ack) \wedge \mathbf{F}send\big)$

  Эта формула, по-видимому, наиболее полно отражает идею причинно-следственной связи событий *send* и *ack*: она утверждает, что *send* в вычислении будет наступать неопределенно часто, и всегда после посылки сообщения *send* когда-нибудь в будущем процесс получит подтверждение *ack*.

Примеры спецификации свойств систем формулами LTL

Разработчику полезно ознакомиться с другими часто используемыми конструкциями, которые можно применять как шаблоны при верификации программ и аппаратуры. Приведем несколько примеров LTL-формул.

1. $\mathbf{GF}\,enabled$

   "Свойство *enabled* будет истинным бесконечное число раз на всех траекториях системы".

2. $\mathbf{GF}\,true$

   Свобода от дедлоков (блокировок): "из каждого достижимого состояния существует возможность продолжения функционирования системы".

3. $\mathbf{G}\big(send \Rightarrow \mathbf{X}\big(\neg send\,\mathbf{U}\,receive\big)\big)$

   На состояниях системы выполняется следующее свойство: "если выполнится *send*, то со следующего состояния в будущем обязательно выполнится *receive*, а до этого момента *send* не будет выполняться".

4. $\mathbf{G}\big(input \Rightarrow \mathbf{X}(output \vee \mathbf{X}output)\big)$

   "Как только установится сигнал *input*, по крайней мере, через два следующих шага вычисления будет установлен *output* ".

5. $\mathbf{FG}\neg p$

   Свойство стабилизации: "*когда-нибудь в будущем система придет в состояние, с которого свойство p никогда больше не наступит*". Эта формула может отражать требование конечности числа наступлений события *p*.

6. $\mathbf{G}\big(crint \Rightarrow \mathbf{F}\neg crint\big)$

   Эта формула может отражать требование конечности числа временных шагов нахождения процесса в критической секции: "*если процесс вошел в критическую секцию, когда-нибудь в будущем он из нее выйдет*".

Спецификация формулами LTL выполнения события на вычислениях

Спецификация того, что событие p выполнится

когда-нибудь в будущем	$\mathbf{F}p$
до события r	$(\neg r)\mathbf{U}(p \wedge \neg r)$
не раньше события r	$(\neg p \wedge \neg r)\mathbf{U}p$
после события q, *но до события* r	$\mathbf{G}((q \wedge \neg p \wedge \neg r) \Rightarrow (\neg r)\mathbf{U}(p \wedge \neg r \wedge \mathbf{XF}r))$

Спецификация формулами LTL отсутствия события на вычислениях

Спецификация того, что событие p не наступит:

никогда	$\mathbf{G}\neg p$
до события r	$\mathbf{F}r \Rightarrow (\neg p)\mathbf{U}r$
после наступления r	$\mathbf{G}(r \Rightarrow \mathbf{G}\neg p)$
после события q, *но до события* r	$\mathbf{G}((q \wedge \neg r) \Rightarrow (\neg p)\mathbf{W}r)$

Заметим, что словесную формулировку свойства поведения можно по разному формализовать с помощью формул темпоральной логики из-за неоднозначности семантики естественного языка.

Примеры спецификации свойств систем формулами CTL

Эти формулы учитывают возможности альтернативного поведения в каждом состоянии вычисления:

AG EF *restart*

"*Из любого достижимого состояния системы возможно достигнуть состояния рестарта*".

EG AF *restart*

"*Существует такое вычисление системы, из любого состояния которого всегда обязательно вернемся в состояние рестарта*".

$\mathbf{EF}(started \wedge \neg ready)$

"Из начального состояния можно достигнуть такого состояния, в котором выполняется started, но не выполняется ready".

Утверждение $@s$ — "*система находится в состоянии s*" — часто используется при спецификации свойств программных систем. Например, при верификации параллельных программ часто используют следующие свойства:

Частичная корректность: $(@Start \wedge \varphi) \Rightarrow \mathbf{AG}\,(@Finish \Rightarrow \psi)$ — если программа при запуске находится в начальном состоянии $(@Start)$, и в начале программные переменные удовлетворяют утверждению φ, то по окончании вычислений, по какому бы пути программа ни пришла в заключительное состояние $@Finish$, программные переменные будут удовлетворять утверждению ψ.

Локальный инвариант: $\mathbf{AG}(@s \Rightarrow \varphi)$ — в состоянии s системы утверждение φ выполняется.

Взаимное исключение: $\mathbf{AG}\left(\neg(@crint_1 \wedge @crint_2)\right)$ — в любом достижимом состоянии программы в своих критических интервалах не могут находиться две программы одновременно.

Отсутствие взаимной блокировки: $\mathbf{AG}(enabled_1 \vee \ldots \vee enabled_n)$ — при функционировании системы параллельных процессов из любого достижимого состояния хотя бы один процесс имеет возможность продолжить вычисления.

Корреляция событий: $\mathbf{AG}(\mathbf{F}p \Rightarrow \mathbf{F}q)$ — на любой траектории выполнения программы в любом состоянии этой траектории если когда-нибудь выполнится условие p, то когда-нибудь выполнится и q (если неопределенно часто p, то неопределенно часто q).

В большинстве случаев темпоральные формулы выражают те свойства поведения реагирующих систем, которые разработчик определяет для некоторых специфических режимов поведения будущей системы. Пусть, например, частью такой спецификации поведения является модуль стейтчарта (карты состояний), представленный на рис. 6.1. Проверяемым свойством системы может быть следующий инвариант:

- в логике LTL:

$$\mathbf{G}\left((mode = ready \wedge event = activate) \Rightarrow \mathbf{X}(mode = active)\right)$$

❒ в логике CTL:

$$\mathbf{AG}\big((mode = ready \wedge event = activate) \Rightarrow \mathbf{AX}(mode = active)\big)$$

Смысл этих формул ясен: "*если модуль mode находится в режиме ready и очередное событие в системе есть activate, то этот модуль перейдет в режим active*".

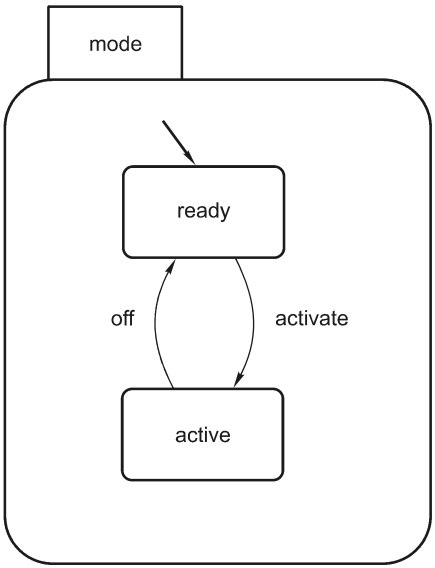


Рис. 6.1. Модуль стейтчарта (карты состояний)

6.2. Свойства достижимости (reachability)

Свойства достижимости являются одними из наиболее часто проверяемых свойств параллельных систем: *некоторая конкретная ситуация в процессе функционирования системы случится.*

Свойство достижимости естественным образом выражается формулой $\mathbf{EF}\varphi$, где φ не содержит темпоральных операторов:

$\mathbf{EF}$ *crint* — *существует такой путь, на котором критическая секция будет достигнута*;

$\neg\mathbf{EF}$ *crash* — состояние отказа никогда не будет достигаться;

$\mathbf{EF}\,(n < 0)$ — *существует путь, на котором значение n станет отрицательным*.

Достижимость из текущего (начального) состояния не означает достижимости из следующего состояния:

EF *exit* — *из начального состояния можно достичь состояния exit* ;

AG EF *exit* — *из любого* (*достижимого*) *состояния можно достичь состояния exit* .

Требование достижимости может сопровождаться условиями. Для выражения условной достижимости используется оператор Until:

$\mathbf{E}\left((n<0)\,\mathbf{U}\,crint\right)$ — *существует возможность того, что система войдет в критическую секцию без прохождения состояний, в которых* $n \geq 0$.

Проверка свойства достижимости выполняется в пространстве всех достижимых состояний простым алгоритмом поиска.

6.3. Свойства безопасности (safety)

Этот класс свойств гарантирует, что при некоторых условиях некоторая ситуация никогда не может быть достигнута (гарантия того, что *нечто плохое никогда не произойдет*). Свойство безопасности выражается формулой $\mathbf{AG}\neg\varphi$. Типичные примеры свойства безопасности — взаимное исключение, свобода от дедлоков (блокировок), сохранение инвариантов. Свобода от блокировок является одним из главных требований к параллельным системам: блокировки возникают, когда каждый процесс из группы параллельно работающих процессов ожидает некоторого события (например, каждому процессу для продолжения его работы необходим ресурс, уже захваченный другим процессом, и каждый процесс, захватив ресурс, ждет освобождения другого нужного ему ресурса, захваченного другим процессом). Примеры свойства безопасности:

- В любом режиме светофор не может разрешать движение по пересекающимся направлениям:

  $$\mathbf{AG}\neg\left(\left(E_green \vee W_green\right) \wedge \left(N_green \vee S_green\right)\right)$$

- Утверждение "*неверно, что существует путь, на котором выполнится условие* φ" можно считать и свойством достижимости, и свойством безопасности. Оно выражается формулой $\neg\mathbf{EF}\varphi = \mathbf{AG}\neg\varphi$.

Безопасность может быть сформулирована с дополнительным условием: "*пока ключ зажигания не вставлен, машина не двинется*": $\mathbf{AG}\left(\neg start\mathbf{W}key\right)$.

Подобная формула со строгим оператором Until:

$$\mathbf{AG}\left(\neg start \mathbf{U} key\right)$$

означает, что ключ обязательно в конце концов будет вставлен.

Вообще проверка свойств безопасности относительно проста: ее можно выполнить за конечное число шагов.

Проверка формул с операторами прошлого. Часто свойства безопасности выражаются с помощью темпоральных операторов прошлого: $\mathbf{X}^{-1}, \mathbf{P}, \mathbf{S}$. Эти темпоральные операторы имеют следующий смысл (*см. гл. 2*):

$\mathbf{X}^{-1}$ обратно $\mathbf{X}$,

$\mathbf{P}$ (от Past) обратно $\mathbf{F}$: ($\mathbf{P} = \mathbf{F}^{-1}$),

$\mathbf{S}$ (от Since, *с тех пор, как*) обратно $\mathbf{U}$.

Примеры:

- "если сигнал *тревога* установился, то в предыдущий момент времени был сигнал *отказ* ": $\mathbf{G}\left(тревога \Rightarrow \mathbf{X}^{-1}\, отказ\right)$
- "каждый раз, когда появляется сигнал *alarm*, ему после последнего *crash* не предшествует *reset* :

  $\mathbf{G}\left(alarm \Rightarrow \left(\neg reset\right)\mathbf{S}\, crash\right)$
- "если сообщение получено приемником, то оно было послано передатчиком по крайней мере за 1 шаг до этого":

  $\mathbf{G}\left(receive \Rightarrow \mathbf{X}^{-1}\mathbf{P}\, send\right)$

Формальное определение семантики этих новых темпоральных операторов очевидно:

- $\sigma^i \models \mathbf{X}^{-1}\varphi \equiv \sigma^{i-1} \models \varphi$ для $i > 0$ — на вычислении $\sigma^i = s_i s_{i+1} s_{i+2} s_{i+3} \ldots$ выполняется LTL-формула $\mathbf{X}^{-1}\varphi$, если φ выполняется в предыдущем состоянии этого вычисления;
- $\sigma^i \models \mathbf{P}\varphi \equiv \left(\exists k : 0 \le k \le i\right) : \sigma^k \models \varphi$ — на вычислении $\sigma^i = s_i s_{i+1} s_{i+2} s_{i+3} \ldots$ выполняется LTL-формула $\mathbf{P}\varphi$, если когда-то в прошлом состоянии вычисления σ (включая настоящее) формула φ выполнилась;

❒ $\sigma^i \models \varphi_1 \mathbf{S} \varphi_2 \equiv \left(\exists j : 0 \le j \le i\right)\left(\sigma^j \models \varphi_2\right)$ и $\left(\forall k : j < k \le i\right) : \sigma^k \models \varphi_1\right)$ — на вычислении $\sigma^i = s_i s_{i+1} s_{i+2} s_{i+3} \ldots$ выполняется LTL-формула $\mathbf{P}\varphi$, если когда-то в прошлом состоянии вычисления σ (включая настоящее) выполнилась формула φ_2, а после ее выполнения во всех состояниях вычисления σ до состояния s_i включительно выполнялась формула φ_1.

Оказывается, что можно проверить выполнение подобных утверждений в рамках уже разработанной теории, не расширяя логику темпоральными операторами прошлого. Для этого вводятся так называемые исторические переменные. Покажем эту технику на примерах.

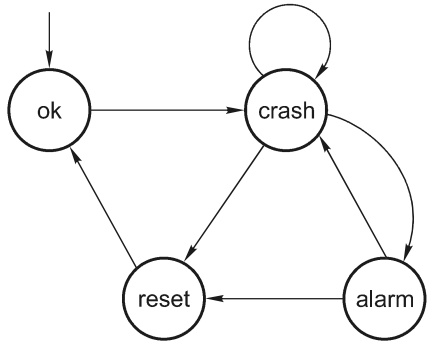
Пример 6.1

Пример структуры Крипке с атомарными предикатами *alarm* и *crash* представлен на рис. 6.2. Рассмотрим формулу $\mathbf{G}\left(alarm \Rightarrow \mathbf{X}^{-1}\, crash\right)$ с темпоральным оператором прошлого $\mathbf{X}^{-1}$. Введем новый атомарный предикат (историческую переменную) *h1* так, чтобы на развертке исходной структуры Крипке M значение *h1* было истинным в тех состояниях, в которых оператор прошлого $\mathbf{X}^{-1}\, crash$ выполняется. Если мы это сделаем, то формула $\mathbf{G}\left(alarm \Rightarrow h1\right)$ превратится просто в свойство достижимости, выраженное в обычной логике LTL, и может быть легко проверена.

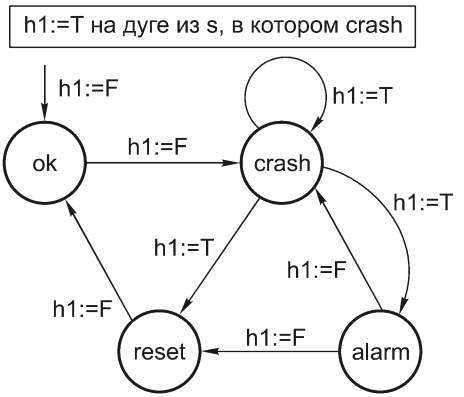
Пометим оператором $h1 := true$ все ребра, идущие из состояний, в которых атомарный предикат *crash* истинен, и оператором $h1 := false$ все ребра, идущие из состояний, в которых атомарный предикат *crash* ложен (рис. 6.2, *б*). Перейдем от системы переходов, ориентированной на сообщения, к системе переходов, ориентированной на состояния, т. е. преобразуем систему переходов так, чтобы состояния новой структуры Крипке хранили те изменения, которые вызваны выполнением операций на входящих в них переходах. Если в одно и то же состояние исходной структуры входят переходы с различными операторами, то такие состояния продублируем (рис. 6.2, *в*). В новой структуре Крипке вместо оператора прошлого $\mathbf{X}^{-1}\, crash$ можно использовать атомарный предикат $h1$.

Пример 6.2

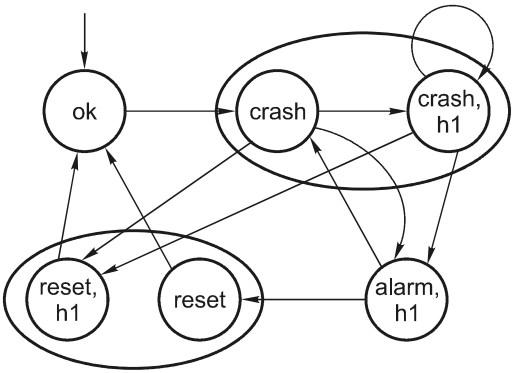
Рассмотрим теперь формулу $\mathbf{G}\left(alarm \Rightarrow \left(\neg reset\right)\mathbf{S}\, crash\right)$ с темпоральным оператором прошлого $p\mathbf{S}q$, который имеет представленную выше семантику: предикат p все время истинен с тех пор, как предикат q был истинен когда-то в прошлом.



а



б



в

Рис. 6.2. Введение исторической переменной $h1 = \mathbf{X}^{-1}crash$

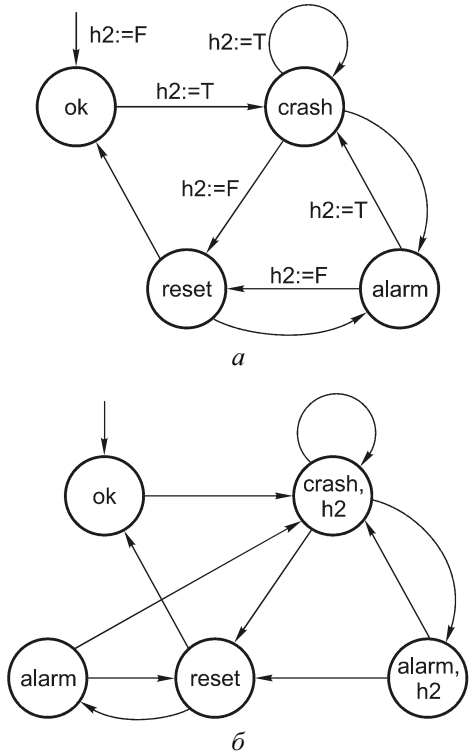


Рис. 6.3. Введение исторической переменной $h2 \equiv (\neg reset)\mathbf{S}crash$

Пример структуры Крипке с атомарными предикатами *reset* и *crash* представлен на рис. 6.3. Введем историческую переменную $h2 \equiv (\neg reset)\mathbf{S}\,crash$ со следующими правилами ее определения:

- на начальной стрелке $h2 := false$, потому что свойство *crash* до начала работы системы не выполнялось;
- на каждом переходе, ведущем в состояние *crash*, выполним присваивание $h2 := true$, поскольку в этом состоянии *crash* станет истинным;
- на каждом переходе, ведущем в состояние *reset*, установим $h2 := false$, если в нем не выполняется *crash*;
- на каждом переходе, ведущем в состояние $\neg reset$, переменная $h2$ не изменяется.

На рис. 6.3, *а* показано такое определение исторической переменной $h2$, а на рис. 6.3, *б* — преобразование структуры Крипке так, чтобы изменения $h2$ были бы зафиксированы в состояниях.

На преобразованной структуре Крипке формула $\mathbf{G}(alarm \Rightarrow h2)$ будет выражать свойство безопасности $\mathbf{G}(alarm \Rightarrow (\neg reset)\,\mathbf{S}\,crash)$ исходной структуры Крипке, которое может быть легко проверено разработанными выше алгоритмами.

Введение подобных переменных для операторов прошлого всегда можно сделать, поскольку истории на структуре Крипке конечны, а начальное состояние не удалено в $-\infty$. Следует, однако, учесть, что в общем случае введение дополнительной двоичной переменной в качестве атомарного предиката удваивает число состояний структуры Крипке. Например, в модели протокола PAR в *гл. 5* (см. рис. 5.4) начальное состояние 0 процесса A расщеплено на два состояния, 0 и 2. Эти состояния эквивалентны как состояния конечного автомата, но не эквивалентны с точки зрения истории: в состоянии 0 не выполняется атомарный предикат s_1, который имеет смысл :

$$s_1 = \mathbf{X}^{-1}\ (\text{процесс_}A\text{_ послал _ сообщение, _ }з\text{анумерованное _1}).$$

Характеризация свойства безопасности. Для формального определения класса свойств безопасности рассмотрим язык, состоящий из некоторого множества бесконечных слов (ω-слов) над алфавитом Σ, т. е. $L \subseteq \Sigma^{\omega}$. Конечная цепочка x над Σ называется "*плохим префиксом*", если $xy \notin L$ для любой бесконечной цепочки $y \subseteq \Sigma^{\omega}$. Иными словами, как бы ни продолжалась цепочка x, полученная бесконечная цепочка не принадлежит языку L. Можно считать цепочку x контрпримером, показывающим, что некоторое свойство не выполняется. Свойство называется свойством безопасности, если любое вычисление, являющееся для этого свойства контрпримером, включает "*плохой префикс*". Поэтому свойства безопасности называются свойствами "*с конечным опровержением*": любое вычисление, которое не удовлетворяет свойству безопасности φ, имеет конечный префикс, на котором φ нарушается.

6.4. Свойства живости (liveness)

Программы, которые ничего не делают, всегда удовлетворяют требованиям безопасности: в них ничего не происходит, поэтому и ничего плохого наступить не может. Очевидно, что требования безопасности в спецификации программ должны сопровождаться требованиями живости, прогресса вычислений системы в нужном направлении. Например, взаимное исключение — свойство безопасности — будет выполняться в системе управления ресурсами, в которой планировщик вообще заблокирует использование ресурса все-

ми процессами. Отсюда следует, что кроме свойств безопасности у системы необходимо обязательно проверить и возможность прогресса. Прогресс в "правильном" направлении обеспечивается свойствами живости. Свойство живости утверждает, что *нечто хорошее в будущем обязательно произойдет*, что выражается формулой CTL $\mathbf{AF}\varphi$, или *нечто хорошее обязательно в будущем будет происходить неопределенно часто*, что выражается формулой LTL $\mathbf{GF}\varphi$. Примеры таких свойств:

Пример 6.3

Любой запрос будет в конце концов обслужен:

$\mathbf{AG}(rec \Rightarrow \mathbf{AF}sat)$ в CTL,

$\mathbf{G}(rec \Rightarrow \mathbf{F}sat)$ в LTL

Система всегда вернется в свое начальное состояние:

AG EF *init* в CTL (в LTL-формулы для выражения этого свойства нет).

После дождя выглянет, наконец, солнце:

$\mathbf{AG}(дождь \Rightarrow \mathbf{AF}солнце)$ в CTL или

$\mathbf{G}(дождь \Rightarrow \mathbf{F}солнце)$ в LTL

Пример 6.4

Протокол выбора лидера состоит в том, что распределенные процессы должны выбрать любой один процесс для координации их параллельной активности. Существует множество различных алгоритмов выбора лидера, но для всех них должны быть выполнены свойства:

- число выбранных лидеров <= 1 (свойство безопасности);
- в конце концов, точно один лидер будет выбран (свойство живости).

Пример 6.5

Рассмотрим задачу построения контроллера — системы управления светофором на перекрестке (рис. 6.4). Контроллер должен обеспечивать безопасный и эффективный проезд перекрестка во всех направлениях: на север (N), восток (E), юг (S) и запад (W). Эти требования можно выразить следующим образом:

- C1: Гарантия отсутствия коллизий — невозможность ситуации, при которой разрешен проезд по пересекающимся направлениям;
- C2: Гарантия обеспечения сервиса — по каждому направлению должна быть обеспечена возможность проезда (прогресс).

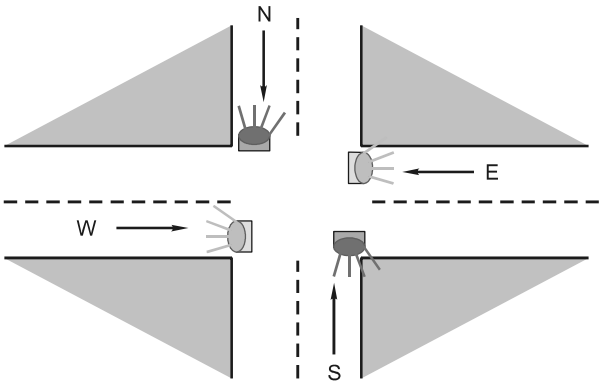


Рис. 6.4. Управление светофорами на перекрестке

Очевидно, что C1 — это свойства безопасности, C2 — свойства живости. Формально для двух пересекающихся направлений:

- C1: $\mathbf{AG}\neg\big(E_green \wedge \big(N_green \vee S_green\big)\big)$

  $\mathbf{AG}\neg\big(N_green \wedge \big(E_green \vee W_green\big)\big)$

- C2: $\mathbf{AG}\big(N_ready \Rightarrow \mathbf{AF}\big(N_green\big)\big)$

  $\mathbf{AG}\big(E_ready \Rightarrow \mathbf{AF}\big(E_green\big)\big)$

Можно считать, что свойство, выраженное в LTL, является свойством живости, если оно не является свойством безопасности. Свойства живости более сложные, чем свойства безопасности: для доказательства свойства живости необходимо анализировать бесконечные цепочки.

6.5. Свойства справедливости (fairness)

Во многих случаях выполнение свойств, определяющих корректное функционирование модели, должно проводиться только на тех вычислениях, на которых выполняется некоторое дополнительное условие, которое по разным причинам называется требованием справедливости (fairness):

$fairness \Rightarrow свойство$.

Существует две причины формулировки требования справедливости. Первая причина состоит в том, что анализируемая система будет работать в некотором окружении, к функционированию которого мы предъявляем эти требования, поскольку только с этими дополнительными требованиями к окружению наша система может выполнять полезную функциональность.

Пример 6.6

Рассмотрим задачу проектирования системы S параллельных процессов $S = \{P1, P2, ...\}$, работающих с общим ресурсом (например, с процессом, предоставляющим некоторый сервис). Свойство "удушения" (starvation) в системе параллельных процессов состоит в том, что процесс, выставивший запрос к ресурсу (или готовый войти в свою критическую секцию), не получает его никогда. Мы можем проверять свойства системы S *в предположении*, что удушения процессов нет, т. е. что планировщик ресурсов (не рассматриваемый как часть проверяемой системы) не игнорирует запросы ни одного из процессов. При этом предположении при верификации системы S можно ограничиться рассмотрением только тех траекторий ее функционирования, на которых планировщик всегда удовлетворяет запросы к ресурсу, выставленные каждым процессом. Естественно назвать планировщик, обладающий этим свойством, справедливым, а такие траектории — "*справедливыми*" (fair). Очевидно, что в РЕАЛИЗАЦИИ планировщика мы должны ТРЕБОВАТЬ выполнения этого свойства (свойство справедливости должно быть обеспечено реальным планировщиком). В то же время можно представить себе такую стратегию планирования выделения ресурса, при которой выбор процесса, которому будет выделен ресурс, всегда происходит, начиная с самого приоритетного, например, с проверки того, выставил ли запрос к ресурсу процесс $P1$, если нет, то выполняется проверка запроса процесса $P2$ и т. п. Очевидно, что эта стратегия не является справедливой: некоторые процессы могут бесконечно ожидать доступа к ресурсу, если другие, более приоритетные процессы, запрашивают его достаточно часто. С другой стороны, стратегия "round robin", т. е. стратегия проверки готовности процессов "*по кругу*", напротив, является справедливой. Стратегия, при которой выбор процесса на исполнение из множества готовых процессов производится с некоторой вероятностью, отличной от 0, также будет справедливой.

Вторая причина совершенно другая. Она состоит в том, что в той модели переходов, которая используется при верификации для представления реальных систем, могут существовать нереалистичные, нереализуемые вычисления, возникшие из-за ограниченных выразительных средств модели.

Пример 6.7

Возможность потери сообщения ненадежным каналом удобно представить в модели канала спонтанным переходом в начальное состояние без передачи сообщения приемнику (см. рис. 5.4 *гл. 5*, каналы $Ch1$ и $Ch2$). В такой модели возможна траектория поведения протокола передачи данных, на которой канал бесконечно теряет передаваемое сообщение. Очевидно, что если вероятность передачи сообщения в реальном канале отлична от нуля, при работе

протокола это вычисление не будет реализовано никогда, хотя оно и существует в модели протокола!

Причиной появления такого вычисления, на котором бесконечно теряется сообщение, является то, что в модели структуры Крипке мы не можем выразить, что некоторые бесконечные цепочки невозможны (например, что они имеют нулевую вероятность). Более формально, в модели структуры Крипке нельзя выразить, например, такое свойство: любые конечные цепочки символов a^* возможны (например, любое *конечное* повторение потерь в канале), а *бесконечная* цепочка потерь a^ω невозможна. Таким образом, дополнительным условием, при котором мы должны проверять правильность функционирования протокола передачи данных, является следующее: "*Если сообщение посылается по каналу неопределенно много раз, то это сообщение будет доставлено каналом бесконечно много раз*". Такое требование к среде передачи сообщений называется требованием "*справедливости*".

Назовем нереалистичные траектории поведения модели несправедливыми. Справедливые траектории поведения системы в этом случае не нужно специально обеспечивать при реализации: реальный канал, например, сам обеспечит ненулевую вероятность доставки сообщения, но формальную проверку свойств системы нужно выполнять при выполнении дополнительных условий справедливости — фактически, отбрасывая нереалистичные траектории поведения модели.

Причины возникновения требований справедливости, таким образом, могут быть различны. Различают три вида справедливости.

- *Безусловная справедливость* — например, "каждый процесс получает обслуживание неопределенно часто".
- *Сильная справедливость* — например, "если процесс запрашивает обслуживание неопределенно часто, он получает его неопределенно часто" или "действие не может быть все время игнорироваться, если оно готово к выполнению неопределенно часто".
- *Слабая справедливость* — например, "если процесс запрашивает обслуживание постоянно (неопределенно долго), он получает его неопределенно часто" или "действие не может бесконечно игнорироваться, если оно все время готово к выполнению".

Итак, в некоторых случаях проверку корректности системы нужно проводить только на справедливых путях вычислений. Как задать ограничение справедливости при верификации?

Введенные выше неформальные требования справедливости, которые возникают из-за многих совершенно разнородных причин, могут быть формализованы по-разному. Рассмотрим некоторые из возможных формулировок.

Пусть φ и ψ — формулы LTL (в частном случае просто атомарные утверждения). Тогда три вида справедливости можно выразить так:

- Безусловная справедливость — $\mathbf{GF}\psi$.

  Например, реалистичные траектории модели, представляющей выпадения очков игральной кости, должны удовлетворять формуле:

  $\mathbf{GF}'1' \wedge \mathbf{GF}'2' \wedge ... \wedge \mathbf{GF}'6'$.

- Сильная справедливость — $\mathbf{GF}\varphi \Rightarrow \mathbf{GF}\psi$.

  При доступе к критической секции для работы с ресурсом, при справедливом планировании доступа траектории должны удовлетворять формуле $\mathbf{GF}wait \Rightarrow \mathbf{GF}crint$.

- Слабая справедливость — $\mathbf{FG}\varphi \Rightarrow \mathbf{GF}\psi$.

  Например, траектории процесса, который включает пешеходный светофор, при постоянном нажатии кнопки разрешения перехода должны бесконечно часто переключать светофор на возможность перехода пешеходов.

Рассмотрим простейшую программу на языке Promela (пример взят из [10]):

```
 1 int n = 0;
 2 bool flag = false;
 3
 4 active proctype P() {
 4  do
 5   :: flag -> break    /* завершается, как только flag равно true */
 6   :: else -> n = 1 - n
 7  od
 9 }
10
11 active proctype Q() {
12   flag = true        /* Q все время готов установить flag в true */
13 }
```

Результаты верификации этой программы зависят от того, учитывается свойство справедливости или нет. Оператор `flag=true` в процессе Q все время готов выполниться, поэтому на всех траекториях, удовлетворяющих свойству слабой справедливости, этот оператор когда-нибудь выполнится, и оба процесса завершатся. Существует единственное вычисление этой программы, на котором готовый к выполнению оператор в строке 12 в процессе Q постоянно игнорируется, а цикл `do` в процессе P выполняется бесконечно. На этом вычислении свойство C : "*программа завершается*" не выполняется. Если

при верификации анализируются все вычисления, то для этой программы свойство C не выполняется, если при верификации слабая справедливость учитывается, то свойство C выполняется.

На практике многие верификаторы, в частности Spin, имеют режим, при котором автоматически реализуется слабая справедливость. При этом, например, если у одного из параллельных процессов есть, по крайней мере, один не блокированный оператор, этот процесс не будет постоянно игнорироваться в проверяемых траекториях. Более сложные виды справедливости необходимо при верификации выражать формулами LTL явно.

Очевидна справедливость следующей теоремы:

ТЕОРЕМА 6.1

Структура Крипке M удовлетворяет формуле ψ при условии справедливости Φ тогда и только тогда, когда $M, s_0 \models (\Phi \Rightarrow \psi)$.

Такой метод верификации при ограничениях справедливости возможен для формул LTL, поскольку ограничения справедливости выражаются формулами именно логики LTL.

6.6. Спецификация справедливости в логике CTL

Для проверки свойств систем, выраженных формулами логики CTL, требуется другой способ задания ограничений справедливости.

Определим множества $\mathfrak{F}_1, \mathfrak{F}_2, ..., \mathfrak{F}_k$ состояний структуры Крипке и будем считать, что путь является справедливым, если хотя бы один элемент из каждого множества $\mathfrak{F}_i$ встречается на нем неопределенно часто.

❖ ***Определение 6.1***

Пусть $M = (S, S_0, R, AP, L)$ — структура Крипке, $F = \{\mathfrak{F}_1, \mathfrak{F}_2, ..., \mathfrak{F}_k\}$ — условие справедливости и $\pi = s_0, s_1, s_2, ..., \pi \in S$ — путь в M (бесконечная цепочка состояний из S). Путь π называется "справедливым" путем, если в π бесконечное число раз встречается по крайней мере по одному состоянию из каждого $\mathfrak{F}_i \in F$.

Такое задание ограничений справедливости довольно эффективно, оно часто используется при верификации, если требования спецификации выражены формулами логики CTL. Рассмотрим примеры.

Пример 6.8

Система управления взаимным исключением процессов. Эта система должна быть построена так, чтобы какой-нибудь процесс (например, более приоритетный) после выхода из своей критической секции не мог сразу вновь получить разрешение на вход в нее, если другие процессы уже ожидают этого разрешения. Если управление взаимным исключением строится с помощью семафора, то *реализация* семафора должна удовлетворять требованиям справедливости (например, поддерживать FIFO очередь процессов, желающих войти в критическую секцию). В то же время АНАЛИЗ системы процессов с критическими секциями должен проводиться В ПРЕДПОЛОЖЕНИИ, что это требование выполняется. Иными словами, при анализе должны рассматриваться только справедливые траектории вычислений. Для системы процессов примера 5.3 *гл. 5* ограничения справедливости можно определить так: $F = \{\{P1@m3\}, \{P2@n3\}\}$ — т. е. каждый процесс может входить в свою критическую секцию бесконечное число раз.

Пример 6.9

На рис. 6.5 представлена модель игральной кости: каждый результат бросания кости приводит к состоянию, в котором отражено число очков, выпавших в данном акте бросания. В модели возможны нереалистичные траектории (последовательности состояний), например такие, в которых бесконечное число раз выпадает только 6. Очевидно, что в реальности могут выполняться только такие траектории, в которых каждая грань кости выпадает неопределенно часто. Если мы используем модель игральной кости как компонент в модели системы параллельных процессов, то при анализе системы мы должны ограничиться только реалистичными траекториями поведения. Назовем их "*справедливыми*". Ограничения справедливости можно в этом случае определить множеством $F = \{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}\}$: на всех вычислениях *каждая* грань должна выпадать неопределенно часто. Это же можно выразить LTL-формулой

$$\mathbf{GF}'1' \wedge \mathbf{GF}'2' \wedge \mathbf{GF}'4' \wedge \mathbf{GF}'5' \wedge \mathbf{GF}'6'$$

Хотя это довольно слабое требование, поскольку оно не совсем отражает то, что происходит в действительности (в реальности все грани выпадают с одинаковой вероятностью), однако во многих приложениях его можно считать хорошей аппроксимацией вероятностного выбора. Этот пример показывает, что с помощью понятия справедливости можно учесть реальные свойства среды, в которой функционирует система.

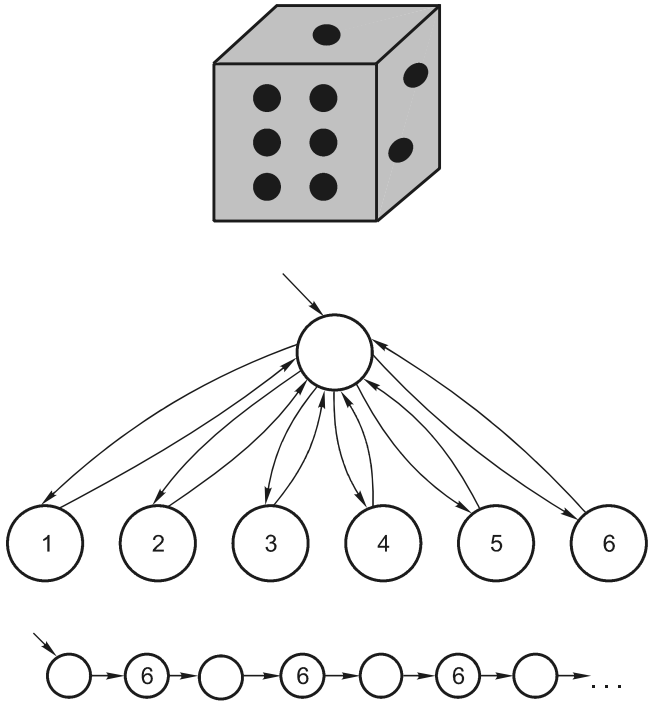


Рис. 6.5. Модель игрального кубика допускает нереалистичные вычисления

Пример 6.10

Рассмотрим композицию двух параллельных процессов (рис. 6.6). В глобальном состоянии $\langle s_1, q_0 \rangle$ композиции возможно выполнение операций как первого, так и второго процессов. Вычисление, в котором выполняется операция только первого процесса, соответствует зависанию системы в этом состоянии, хотя второй процесс готов выполнить свой переход. Понятно, что такая траектория вычислений нереальна, ее можно считать несправедливой. Таким образом, нереализуемые траектории поведения в модели системы могут возникнуть из-за семантики интерливинга. Ограничением справедливости в этом примере будет множество $F = \{\{s_1\}, \{q_1\}\}$: и состояние s_1, и состояние q_1 должны встречаться в вычислениях, на которых осуществляется анализ поведения системы, бесконечное число раз. Этот пример показывает, что с помощью понятия справедливости можно учесть ограничения модели переходов и их асинхронной композиции. Заметим, что в режиме удовлетворения свойства слабой справедливости в верификаторе (например, в языке Spin) все такие нереальные вычисления не будут анализироваться.

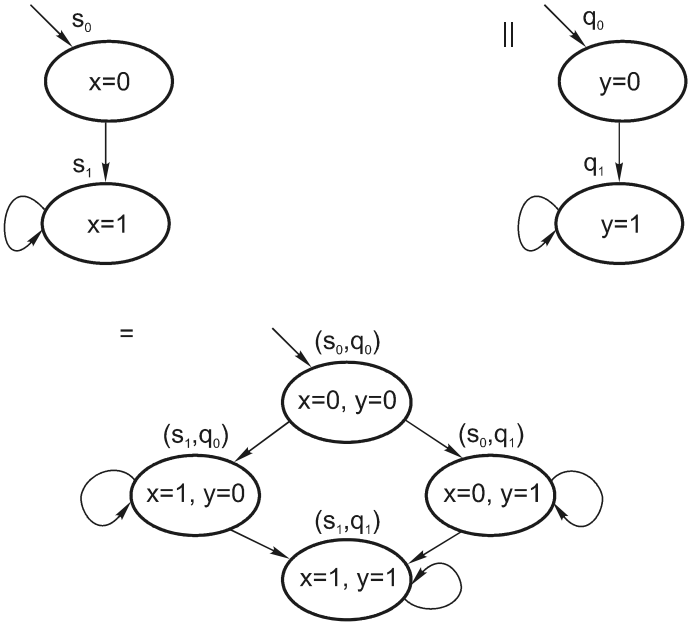


Рис. 6.6. Асинхронная композиция процессов порождает нереалистичные вычисления

Пример 6.11

Рассмотрим структуру Крипке K, представленную на рис. 6.7, и путь $\pi = s_0 (s_1, s_3)^{\omega}$ на K. В состоянии s_1 есть выбор между переходами в состояния s_2 или s_3, но на пути π возможность перехода из s_1 в s_2 всегда игнорируется. Интуитивно траектория π не является реалистичной в K: если на некотором вычислении бесконечно часто предлагается альтернативный выбор между двумя возможностями, каждая из них рано или поздно (или даже бесконечное число раз) должна быть выбрана. Определим для структуры Крипке K два ограничения справедливости: $F = \{\{s_2\}, \{s_3\}\}$, которые имеют следующий смысл: обе альтернативы в состоянии s_1 должны выбираться неопределенно часто на любом вычислении K. Вычисление π не удовлетворяет этим ограничениям справедливости: в нем вовсе не встречается состояние s_2.

Таким образом, с помощью ограничения справедливости можно исключить те вычисления модели, которых в реальной системе нет (если вычисления, на которых одна из альтернатив бесконечно игнорируется в альтернативном выборе, считать нереальными (несправедливыми)).

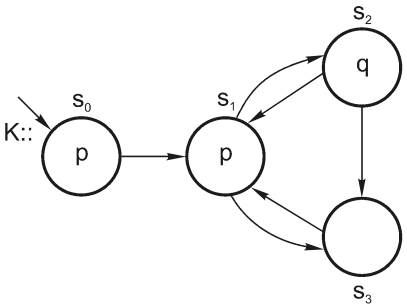


Рис. 6.7. Структура Крипке,
в которой присутствует альтернативный выбор

Как видно из этих примеров, определение ограничений справедливости как набора подмножеств состояний структуры Крипке весьма удобно. Однако, как было доказано в [45], это свойство невыразимо формулами CTL. Для того чтобы остаться в рамках CTL и использовать разработанный для CTL эффективный алгоритм верификации, в [31] было предложено несколько изменить семантику формул CTL так, чтобы выполнимость формул CTL определялось бы не на всех вычислениях структуры Крипке, а только на справедливых вычислениях. Такая семантика формул CTL называется "*справедливой*" семантикой.

Ниже мы рассмотрим, как должны интерпретироваться формулы темпоральной логики ветвящегося времени на справедливой структуре Крипке.

6.7. Справедливая CTL (fair CTL)

Итак, условия справедливости выразить в логике CTL невозможно. Однако можно изменить семантику формул CTL так, чтобы они интерпретировались с дополнительным условием справедливости таким образом, чтобы при анализе несправедливые вычисления игнорировались.

❖ ***Определение 6.2***

"Справедливой" структурой Крипке M_F назовем структуру Крипке с множеством ограничений справедливости: $M_F = (S, S_0, R, AP, L, F)$, где ограничение справедливости $F \subseteq 2^s$ есть множество подмножеств состояний структуры Крипке.

Любая формула CTL в справедливой структуре Крипке должна интерпретироваться не на всех возможных путях, которые ведут из данного состояния структуры Крипке, а только на справедливых путях.

Заметим, что это условие совпадает с условием допуска ω-слов в обобщенном автомате Бюхи (*см. гл. 4*). При пустом множестве F справедливая структура Крипке эквивалентна обычной структуре Крипке: в ней любой путь является справедливым.

Определим справедливую семантику для базиса $\{\mathbf{EX}, \mathbf{EU}, \mathbf{EG}\}$ логики CTL.

Для структуры Крипке $M_F = (S, S_0, R, AP, L, F)$, где $F = \{\mathcal{F}_1, \mathcal{F}_2, ..., \mathcal{F}_k\}$, обозначим $M_F, s \models_F \Phi$ следующее утверждение: формула Φ удовлетворяется в состоянии s структуры Крипке M_F при ограничениях справедливости F.

Введем в рассмотрение новый атомарный предикат *fair* — ему удовлетворяют все такие состояния справедливой структуры Крипке M_F, из которых *существует* вычисление, проходящее через состояния каждого из множеств $\mathcal{F}_1, \mathcal{F}_2, ..., \mathcal{F}_k$ бесконечное число раз. Рассмотрим, как построить множество состояний, в которых этот атомарный предикат истинен.

❖ ***Определение 6.3***

Справедливой ССК (сильно связной компонентой) структуры Крипке M_F будем называть такую ССК, которая пересекается со всеми множествами $\mathcal{F}_1, \mathcal{F}_2, ..., \mathcal{F}_k$.

❖ ***Утверждение ([31], лемма 4.1)***

Предикат *fair* истинен в состоянии s *тогда и только тогда*, когда в структуре Крипке M_F существует достижимая из s справедливая *сильно связная компонента* (рис. 6.8).

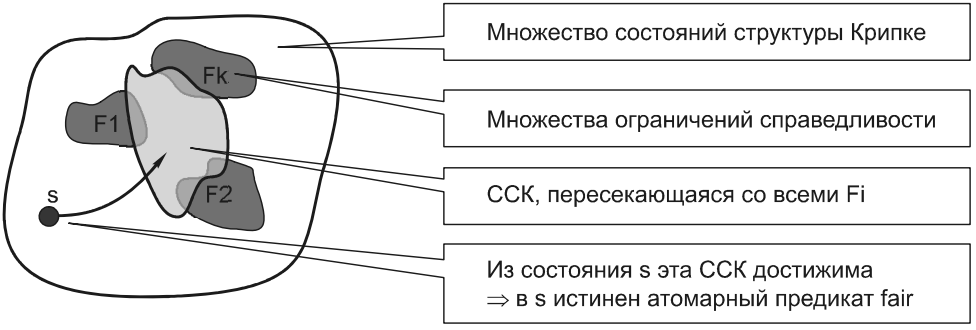


Рис. 6.8. Состояния структуры Крипке, в которых истинен атомарный предикат *fair*

В соответствии с этим утверждением, определение множества состояний S_F, в которых выполняется атомарный предикат *fair*, состоит из трех шагов:

- находятся все сильно связные компоненты M_F. Назовем их ССК$_1$, ССК$_2$, ..., ССК$_n$;
- находятся справедливые сильно связные компоненты M_F, т. е. те, которые пересекаются со всеми множествами $\mathcal{F}_i$ ограничений справедливости;
- строится множество $S_F \subseteq S$ состояний M_F — это все такие состояния, из которых достижима хотя бы одна справедливая сильно связная компонента.

Пометим все состояния из S_F атомарным предикатом *fair*, используя функцию пометок L:

```
for all s∈S_F do L(s) := L(s)∪{ fair } od
```

Алгоритм маркировки состояний M_F в соответствии со справедливой семантикой для каждой подформулы ψ формулы Φ выполняет следующие шаги:

1. Вычисляется множество состояний $Sat_{fair}(\psi)$, в которых ψ выполняется в соответствии со справедливой семантикой.
2. Вводится новый атомарный предикат p_ψ.
3. Этим атомарным предикатом помечаются все состояния из множества $Sat_{fair}(\psi)$.

Поскольку обработка каждой формулы ψ заканчивается введением нового атомарного предиката p_ψ и маркировкой этим предикатом p_ψ всех состояний, удовлетворяющих ψ, то на каждом шаге алгоритма элементами подформул являются атомарные предикаты. Заметим, что атомарный предикат p_ψ означает выполнение формулы ψ, поэтому p_ψ истинно в состоянии s тогда и только тогда, когда выполнимость ψ в s определена в соответствии со справедливой семантикой.

Следующий алгоритм вычисляет множество $Sat_{fair}(\Phi)$ таких состояний структуры Крипке, в которых формула Φ выполняется в соответствии со справедливой семантикой

```
for all 0< i ≤ |Φ| do
  for all Ψ∈ Sub(Φ) with |Ψ| = i do
```

```
    switch(Ψ):
       true    :      Sat_fair(Ψ) := S_F;
       p       :      Sat_fair(Ψ) := { s∈S_F | p∈L(s) };
       ¬p      :      Sat_fair(Ψ) := { s∈S_F | p∉L(s) };
       p∧q     :      Sat_fair(Ψ) := { s∈S_F | p,q∈L(s) };
       EXp     :      Sat_fair(Ψ) := Sat( EX(p∧fair)) );
       E(pUq):        Sat_fair(Ψ) := Sat( E(pU(q∧fair)) );
       EGp     :      compute Sat_fair(EGp);  /*  определено ниже  */
    end switch
/* состояния из Sat_fair(Ψ) помечаем новым атомарным предикатом p_Ψ */
  for all s∈ Sat_fair(Ψ) do L(s):=L(s)∪{ p_Ψ } od
od
```

Рассмотрим этот алгоритм. Алгоритм работает $|\Phi|$ раз, выделяя подформулы формулы Φ, начиная с самых внутренних (здесь $|\Phi|$ — число подформул формулы Φ). Основным оператором алгоритма является переключатель **switch**, который обрабатывает различные подформулы Φ так:

Ψ=true — в s на справедливых вычислениях M_F выполняется атомарный предикат true, если в s истинен атомарный предиката fair, т. е. если $s \in S_F$;

Ψ=p — в s на справедливых вычислениях M_F выполняется атомарный предикат p, если $s \in S_F$, и, кроме того, в s истинен атомарный предикат p;

Ψ=¬p — в s на справедливых вычислениях M_F выполняется формула $\neg p$, если $s \in S_F$, а предикат p ложен в s;

Ψ=p∧q — в s на справедливых вычислениях M_F выполняется формула $p \wedge q$, если $s \in S_F$, и в s выполняются как p, так и q;

ψ = **EX**p — в s на справедливых вычислениях M_F выполняется формула $\mathbf{EX}p$, если из s существует переход в такое состояние, из которого существует справедливое вычисление (т. е. в нем истинен предикат *fair*) и в котором истинен атомарный предикат p. Эту задачу решит нахождение обычным алгоритмом model checking для CTL-формул (с обычной CTL-семантикой) такого множества состояний, в котором истинна формула $\mathbf{EX}(p \wedge \mathit{fair})$. Здесь используется очевидное соображение: вычисление, которое после первого шага становится справедливым, само будет справедливым;

ψ = **E**(p**U**q) — в s на справедливых вычислениях структуры Крипке M_F выполняется формула $\mathbf{E}(p\mathbf{U}q)$, если из s существует конечный путь, все состояния которого помечены p, в такое состояние, из которого существует справедливое вычисление (т. е. в нем истинен предикат *fair*) и в котором истинен атомарный предикат q. Эту задачу решит нахождение обычным алгоритмом model checking для CTL-формул (с обычной CTL-семантикой) такого множества состояний, в котором истинна формула $\mathbf{E}(p\mathbf{U}(q \wedge fair))$. Здесь также используется очевидное соображение: вычисление, которое после конечного числа шагов становится справедливым, само будет справедливым;

Ψ= **EG**p — эта подформула обрабатывается специальным алгоритмом. Он похож на алгоритм нахождения состояний, удовлетворяющих атомарному предикату *fair*, и состоит из четырех шагов:

- справедливая модель M_F (т. е. модель, в каждом состоянии которой выполняется предикат *fair*), ограничивается такой моделью M_{Fp}, в каждом состоянии которой выполняется еще и атомарный предикат p (напомним, что если p=p$_\psi$ для некоторой формулы ψ, то во всех этих состояниях формула ψ выполняется справедливо, поскольку мы проводим анализ индукцией по подформулам формулы Φ);
- находятся все сильно связные компоненты на множестве состояний M_{Fp}. Назовем их ССК$_1$, ССК$_2$, ..., ССК$_n$;
- находятся все справедливые ССК, т. е. те, которые пересекаются со всеми множествами $\mathcal{F}_1,...,\mathcal{F}_k$ ограничений справедливости. Назовем их $C_1, C_2, \ldots, C_r$;
- формула **EG***p* *справедливо* выполняется в состоянии s *тогда и только тогда*, когда в M_{Fp} из s достижимо хотя бы одно из множеств C_i.

Рис. 6.9 показывает последовательность построения множества состояний, удовлетворяющих формуле **EG***p* в соответствии со справедливой семантикой.

Пример 6.12

Снова рассмотрим структуру Крипке K на рис. 6.7. Проверим наличие у K свойства, выраженного формулой Φ = **AF***q* — *на любых путях когда-нибудь в будущем встретится* q.

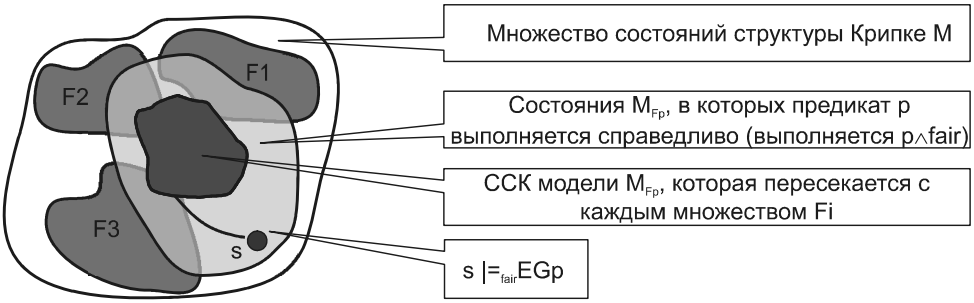


Рис. 6.9. Состояния, в которых формула **EF**p выполняется справедливо

Для проверки преобразуем эту формулу к базису $\{\mathbf{EX}, \mathbf{EU}, \mathbf{EG}\}$. Поскольку $\mathbf{A}\varphi = \neg\mathbf{E}\neg\varphi$, $\mathbf{F}\psi = \neg\mathbf{G}\neg\psi$, то $\mathbf{AF}q = \neg\mathbf{EG}\neg q$. Подформулы этой формулы таковы: $f1 = q$, $f2 = \neg f1$, $f3 = \mathbf{EG}f2$, $f4 - \neg f3$.

Проверим выполнение свойства $\Phi = \neg\mathbf{EG}\neg q$ на структуре Крипке K при обычной семантике CTL. Строим множества состояний, на которых выполняются подформулы формулы Φ:

1. $f1 = q; Sat(f1) = \{s_2\}$ — новым атомарным предикатом $f1$ помечаем состояние s_2.
2. $f2 = \neg f1; Sat(f2) = \{s_0, s_1, s_3\}$ — новым атомарным предикатом $f2$ помечаем s_0, s_1, s_3.
3. $f3 = \mathbf{EG}f2$; Строим ССК множества $\{s_0, s_1, s_3\}$ тех состояний, в которых $f2$ выполняется. Эта единственная ССК=$\{s_1, s_3\}$ достижима из всех состояний K, т. е. $Sat(f3) = \{s_0, s_1, s_2, s_3\}$.
4. $\Phi = \neg f3; Sat(\Phi) = \varnothing$.

Вывод: в K нет состояний, в которых выполняется формула $\Phi = \mathbf{AF}q$. Действительно, в K существует вычисление $\pi = s_0(s_1, s_3)^{\omega}$, которое никогда не проходит через состояние s_2.

Проверим теперь выполнение свойства $\Phi = \neg\mathbf{EG}\neg q$ при справедливой семантике. В примере 6.8 определены два ограничения справедливости: $F = \{\{s_2\}, \{s_3\}\}$, которые имеют следующий смысл: обе альтернативы в состоянии s_1 должны выбираться неопределенно часто в любой траектории функционирования K. Для справедливой структуры Крипке K_F будем рас-

сматривать только справедливые пути, т. е. те пути, на которых бесконечно часто встречаются состояния s_2 и s_3.

Введем новый атомарный предикат *fair* и им пометим состояния K, из которых существуют справедливые пути. Для того чтобы найти такие состояния, построим все ССК множества состояний $\{s_0, s_1, s_2, s_3\}$. Существует только одна ССК этого множества $C = \{s_1, s_2, s_3\}$. Множество C пересекается с каждым из ограничений справедливости — множествами $F_1 = \{s_2\}$ и $F_2 = \{s_3\}$. Поэтому C — справедливая ССК. Состояния из C достижимы из всех состояний структуры Крипке K, поэтому $S_F = S = \{s_0, s_1, s_2, s_3\}$. Все состояния из S_F пометим атомарным предикатом *fair*: $Sat(fair) = \{s_0, s_1, s_2, s_3\}$.

Проверяем выполнение свойства $\Phi = \neg\mathbf{EG}\neg q$ на структуре Крипке K_F по подформулам Φ:

1. $f1 = q$; $Sat_{fair}(f1) = \{s_2\}$.
2. $f2 = \neg f1$; $Sat_{fair}(f2) = \{s_0, s_1, s_3\}$.
3. $f3 = \mathbf{EG}f2$. Для нахождения $Sat_{fair}(f3)$ выполним следующее:
   - справедливую модель K_F ограничиваем до модели K_{Ff2}, во всех состояниях которой выполняется $f2: S_{Ff2} = \{s_0, s_1, s_3\}$;
   - строим все ССК множества $\{s_0, s_1, s_3\}$. Это единственная ССК=$\{s_1, s_3\}$;
   - находим справедливые ССК, т. е. такие, которые пересекаются с ограничениями справедливости $F1 = \{s_2\}$ и $F2 = \{s_3\}$. Таких ССК нет;
   - множество состояний K_F, из которых достижимы справедливые ССК, пусто, поэтому $Sat_{fair}(f3) = \varnothing$.
4. $\Phi = \neg f3$; $Sat(\Phi) = \{s_0, s_1, s_2, s_3\}$.

Вывод: во всех состояниях K_F формула $\Phi = \mathbf{AG}q$ выполняется при введенном ограничении справедливости (мы рассматривали только *справедливые* пути из начального состояния). Поскольку эта формула выполняется в начальном состоянии K_F, она выполняется в K_F. Заметим, что определение ограничения справедливости только одним множеством $\{s_2, s_3\}$ не решает проблемы: формула $\mathbf{AF}q$ в такой структуре Крипке не выполняется.

6.8. Заключение

Важность выделения классов свойств заключается в том, что кроме проверки специфических свойств системы важно проверить также, что в системе отсутствуют ошибки, типичные для параллельных систем. Если реагирующая система состоит из нескольких взаимодействующих компонентов, то в ней полезно проверить такие свойства, как отсутствие блокировок, прогресс вычислений ее компонент и т. п. Неформально говоря, свойства безопасности выражают, что ничего плохого не случится, свойства живости — что нечто "хорошее" случится, а свойство справедливости — что при верификации рассматриваются только те вычисления, которые реализуемы в реальной системе. Например, пусть мы строим арбитр общей шины, которая разделяется несколькими процессорами. Свойство безопасности в спецификации арбитра будет состоять в том, что в каждый момент не более одного процессора будет иметь доступ к шине. Свойство живости будет выражать то, что арбитр неопределенно часто будет находиться в состоянии разрешения использования шины каким-нибудь процессором. Свойство справедливости арбитра будет выражать, что каждый процессор будет иметь доступ к шине бесконечно часто, в частности, он может захватить шину на произвольное, но конечное время.

Метод учета ограничений справедливости состоит в том, что мы остаемся в рамках той же модели — структуры Крипке, но вводим ограничение: те вычисления, которые являются реалистичными, назовем "справедливыми" (fair), и пытаемся при анализе рассматривать только их. Это ДОПОЛНИТЕЛЬНОЕ требование к тем свойствам, которые следует проверять в дискретных системах. Вычисление будет анализироваться на предмет выполнения требований спецификации, только если на вычислении выполняется это свойство. Условие справедливости чаще всего вводится для того, чтобы иметь возможность отбросить нереалистичные поведения, являющиеся результатом использования упрощенной модели. Можно считать, что справедливая структура Крипке — это обычная структура Крипке, дополненная правилом допуска ω-слов обобщенного автомата Бюхи.

6.9. Замечания

Множество примеров спецификации свойств дискретных систем формулами темпоральной логики и классификация их в шаблоны (patterns) дано в [41]. В *разделе 6.1* использованы результаты этой работы. Выделение типов свойств (свойств безопасности и свойств живости) было впервые предложено в [92]; впоследствии эти классы свойств многократно обсуждались. Весьма подробно проблемы спецификации свойств реагирующих систем и класси-

фикации свойств рассмотрены в [11], где этому вопросу посвящен целый раздел монографии. Классификация свойств на свойства достижимости, безопасности и живости позволяет выбрать наиболее подходящий метод проверки этих свойств. В частности, для доказательства свойств безопасности не нужно учитывать справедливость вычислений — свойство безопасности выполняется, если все вычисления, как справедливые, так и не справедливые, удовлетворяют этому свойству. С другой стороны, доказательства свойств живости необходимо выполнять только для справедливых вычислений. Во многих работах была рассмотрена проблема выражения этих типов свойств в различных логиках (см., например, [135]). Пример с подбором темпоральной формулы, отражающей свойство причинной зависимости событий, приведен в лекциях К. Савенкова, прочитанных в МГУ [195].

Метод введения исторических переменных, которые используются для спецификации свойств прошлого, а также приведенные в *разделе 6.2* примеры заимствованы из [11]. Множество практических рекомендаций и примеров спецификации и верификации систем с операторами прошлого дано в [127]. Положительный ответ на старый вопрос о том, является ли темпоральная логика с модальностями прошлого экспоненциально более экономной, дан в [99]. В литературе предложено множество различных типов справедливости и методов ее обеспечения при верификации параллельных систем [58], [94], [103].

6.10. Задачи к главе 6

6.1. Постройте CTL-формулы, описывающие следующие свойства технических систем:

а) при функционировании системы свойство p никогда не выполнится между q и r;

б) если постоянно запрашивать, то в будущем войдешь в критическую секцию;

в) существует путь, на котором p выполняется до тех пор, пока в будущем на всех продолжениях вычислений будет всегда выполняться сигнал q.

6.2. Для следующих свойств живости написать формальное определение:

Если долго стараться, то в конце концов получится.

Если лифт вызван, он обязательно придет рано или поздно.

Светофор в конце концов переключится на зеленый.

В конце концов я закончу университет.

Если нажать на кнопку лифта, то он в конце концов приедет.

Солнце будет выглядывать неопределенно часто.

6.3. Рассмотрим систему управления лифтом, обслуживающим N этажей, от 0 до $N-1$. Пусть $door(i)$ — атомарный предикат, обозначающий, что двери лифта на i-м этаже открыты, $lift(i)$ обозначает, что лифт находится на i-м этаже, $req(i)$ обозначает, что на i-м этаже нажата и горит кнопка вызова, $move(i)$ обозначает, что в лифте нажата и горит кнопка посылки лифта на этаж i.

Запишите следующие требования к системе управления лифтом:

(а) Дверь на этаже никогда не будет открыта, если лифт не находится на этом этаже.

(б) Запрос с любого этажа будет когда-нибудь обслужен.

(в) Лифт возвращается на этаж 0 неопределенно часто.

(г) Лифт не двигается, если не было необслуженного вызова.

Какие из этих свойств являются свойствами безопасности, а какие — свойствами живости?

6.4. Дана структура Крипке $M=(S, S_0, R, AP, L)$, где:

$S=\{s_0, s_1, s_2, s_3, s_4, s_5\}, S_0=\{s_0\}, AP=\{p,q\}$,

$R=\{(s_0,s_1),(s_0,s_2),(s_2,s_3),(s_3,s_0),(s_2,s_4),(s_1,s_5),(s_4,s_5),(s_5,s_4)\}$,

$L=\{(s_0,\{p,q\}),(s1,\{\ \}),(s_2,\{q\}),(s_3,\{p\}),(s_4,\{q\}),(s_5,\{p\})\}$.

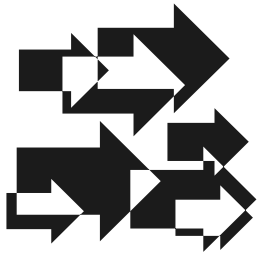
а) Проверьте выполнение CTL-формулы $\mathbf{AG}(p\vee q)$.

б) Введите условие справедливости $F=\{\{s_2,s_3\},\{s_5\}\}$, т. е. любой справедливый путь на этой структуре Крипке должен проходить через состояния s_2 или s_3, а также через состояние s_5 бесконечное число раз.

Определите множество "*справедливых*" (fair) состояний.

Проверьте выполнение формулы $\mathbf{AG}(p\vee q)$ на "с*праведливой*" (fair) структуре Крипке.

ГЛАВА 7



Примеры верификации

В этой главе техника верификации демонстрируется на нескольких примерах простых протоколов. Анализируемые протоколы специфицируются на языке Promela, являющемся входным языком описания асинхронных моделей пакета верификации Spin.

Приведенные здесь примеры не совсем тривиальны, они требуют нескольких десятков строк спецификации алгоритмов на языке Promela.

7.1. Протокол передачи данных W.C.Lynch

В этом разделе приведен пример программы на языке Promela, заимствованный из руководства по этому языку. Программа описывает простой протокол, который передает данные между двумя пользователями *A* и *B* в обоих направлениях. Фактически это протокол, описанный W.C.Lynch, который схематически представлен в *гл. 1* (рис. 1.2). Мы сопроводим пример подробными комментариями, чтобы продемонстрировать на нем возможности языка Promela. Для лучшего понимания текста программы на рис. 7.1 приведена архитектура этого протокола.

Два пользователя, *A* и *B*, передают и принимают данные по соответствующим каналам Ain, Aout, Bin, Bout, взаимодействуя со своими модулями приема-передачи, которые общаются друг с другом по ненадежным каналам `AtoB` и `BtoA`. Программа на языке Promela начинается в строке 1 описанием констант перечислимого типа `mtype`, к которым можно обращаться по именам. Этими именами будут предваряться передаваемые данные: сообщения в протоколе состоят из двух частей (см. рис. 1.2 в *гл. 1*): передаваемые по сети данные предваряются одним из служебных слов `ack` — положительное подтверждение (`acknowledge`), `nack` — отрицательное подтверждение, `err` — ошибка в

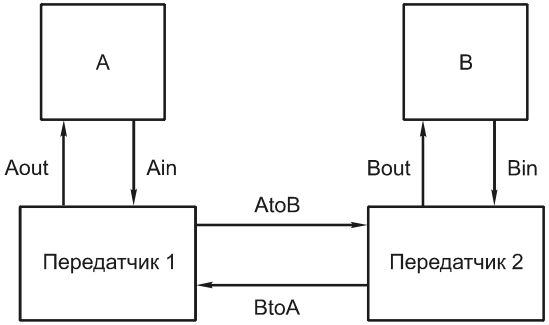


Рис. 7.1. Архитектура протокола

передаче. Полученная пара данных, например, `<ack, i>` для удобства восприятия в программе на Promela может представляться как `ack(i)`. Для единообразия структуры сообщений данные, получаемые от пользователя, предваряются служебным словом `next`, передаваемые пользователю — служебным словом `accept`. Здесь та же ситуация: вместо пары `<accept, i>` при передаче пользователю можно записать `accept(i)` — принято сообщение `i`.

```
 1 mtype = { ack, nak, err, next, accept };
 2
 3 proctype transfer(chan in,out,CHin,CHout)
 4 {  byte o, i;    /* переменные для выходных и входных сообщений */
 5    in ? next(o); /* по каналу in ждем сообщение для передачи */
 6
 7  do   /* ожидаем один из трех типов сообщений nack, ack, err */
 8  :: CHin ? nak(i) ->
 9             out ! accept(i);
10             CHout ! ack(o)
11
12  :: CHin ? ack(i) ->
13             out ! accept(i);
14             in ? next(o);
15             CHout ! ack(o)
16
17  ::     CHin ? err(i) ->
18             CHout ! nak(o)
19  od
20  }
21
22 init
23  { chan AtoB = [1] of { mtype, byte };
24    chan BtoA = [1] of { mtype, byte };
```

```
25
26    chan Ain  = [2] of { mtype, byte };
27    chan Bin  = [2] of { mtype, byte };
28
29    chan Aout = [2] of { mtype, byte };
30    chan Bout = [2] of { mtype, byte };
31
32    atomic {
33             run transfer(Ain, Aout, AtoB, BtoA);
34             run transfer(Bin, Bout, BtoA, AtoB)
35           };
36
37    AtoB!err(0)
38  }
```

В строке 3 с ключевого слова `proctype` задается описание типа процесса `transfer`. Это процесс приема-передачи, которому в качестве параметров при запуске должны быть переданы имена четырех конкретных каналов. Каналы `in` и `out` связывают этот процесс с пользователем, каналы `CHin` и `CHout` связывают его с другим процессом приема-передачи. Если процесс будет запущен с такими значениями параметров: `transfer(Ain, Aout, AtoB, BtoA)`, то он будет работать, как Передатчик 1 (рис. 7.1), если он будет запущен, как `transfer(Bin, Bout, BtoA, AtoB)`, то он будет работать, как Передатчик 2.

В строке 4 открывается тело процесса `transfer` и описываются две переменные типа `byte`: переменная `o` для выходных данных от пользователя и переменная `i` для входных данных к пользователю. Конкретные значения этого типа, предваренные константами `ack`, `nack`, `err` и т. д. перечислимого типа `mtype`, будут передаваться по каналам. Строка 5 — первый оператор процесса — прием от пользователя по каналу `in` сообщения, состоящего из двух частей: служебного слова `next`, предваряющего данное для передачи, которое помещается в переменную `o`. В строке 7 начинается бесконечный цикл приема-передачи данных. Оператор цикла `do` имеет три возможные альтернативы — прием от другого процесса сообщения, предваренного одной из констант `nack`, `ack` или `err` — в точности так, как в протоколе W.C.Lynch. Если принятое сообщение не удовлетворяет одному из трех предопределенных типов структуры (служебное слово `nack`, `ack` или `err`, за которым идет любое данное типа `byte`), то происходит блокировка процесса. Принятое данное присваивается переменной `i`, но разные префиксы принятого сообщения вызывают различные реакции, как это и представлено на рис. 1.2 в *гл. 1*.

Начиная со строки 22, описывается процесс `init`, который вводит описания шести каналов взаимодействия процессов (см. рис. 7.1): `AtoB`, `BtoA`, `Ain`, `Aout`,

Bin и Bout. В первых двух каналах возможна буферизация не более одного сообщения, в остальных четырех — не более двух сообщений. Скобка {mtype, byte} в описании каналов указывает, что каждый канал может передавать сообщения, состоящие из двух полей: первое поле — любое данное типа mtype, второе — типа byte. Данные типа mtype — символьные константы ack, nack, err — перечислены в строке 1. В строке 32 одновременно запускаются оба процесса передачи с параметрами, обсужденными выше. Заканчивается процесс init передачей ошибочного сообщения для инициализации процессов передатчиков (чтобы они не ждали бесконечно друг друга в цикле do). Необходимость этого искусственного приема объясняется некорректностью протокола, описанного W.C.Lynch — в нем инициализация процессов не описана. Заметим, что здесь не моделируются ошибки, возникающие в реальных каналах передачи данных.

Для использования этого описания в качестве модели протокола необходимо построить модели пользователей *A* и *B*, которые в каналы Ain и Bin передавали бы сообщения, содержащие пару: служебное слово next и произвольное значение типа byte (например, символ), и принимали бы по каналам Aout и Bout соответствующие сообщения.

7.2. Протокол PAR

Этот протокол был приведен в *гл. 5* как пример протокола коммуникации, работающего некорректно при редких сочетаниях ошибок в канале связи. Протокол PAR является разновидностью классического протокола альтернирующего бита (Alternating Bit Protocol, ABP). В отличие от протокола ABP, в протоколе PAR подтверждения, которые посылает приемник о получении сообщений, не нумерованы. Это и является причиной некорректности протокола PAR. Подобные некорректности легко выявляются при анализе протокола с помощью системы Spin.

Протокол ABP — это классический пример протокола. Он часто приводится в учебных курсах для демонстрации асинхронной природы взаимодействия параллельных процессов в распределенной системе. Несмотря на свою простоту, протокол не может быть записан в несколько строк: в работе [82] спецификация этого протокола на языке Estelle занимает 300 строк кода. В приводимом ниже описании мы покажем, как этот протокол может быть описан на языке Promela: это описание, фактически, повторяет спецификацию этого протокола в виде взаимодействующих систем переходов, приведенную на рис. 5.4. Цель приводимого ниже описания этого протокола — демонстрация того, что распределенная система, каждый модуль которой представлен в виде конечно-автоматного описания, практически однозначно может быть представлена системой взаимодействующих процессов на языке Promela.

```
00 /* описание протокола PAR */
01 mtype = { d0, d1, ACK }        /* возможные сообщения в каналах */
02 chan c1, c2, c3, c4, c5, c6 = [0] of { mtype }     /* 6 каналов */
03 bool s0, s1, r0, r1 = false;
```

Первая строка описания — определение сообщений в каналах (это данные, занумерованные 0 и 1, и подтверждение `ACK`), которые удобно представить в модели как константы перечислимого типа. Хотя сообщения, передаваемые по каналам, могут быть произвольными, мы моделируем их просто константами. Порты 1, ..., 6 в спецификации протокола в *гл. 5* здесь представляются синхронными каналами `c1–c6`, по которым без буферизации передаются сообщения типа `mtype`. Каналы *Ch*1 и *Ch*2, по которым эти сообщения передаются на рис. 5.4, имеют свое собственное поведение (они могут терять информацию), и поэтому представлены здесь процессами *Channel*1 и *Channel*2 соответственно.

В строке 03 описаны глобальные булевы переменные `s0`, `s1`, `r0`, `r1`. Они являются атомарными предикатами и будут использоваться при верификации протокола. Они имеют следующий смысл:

`Si` — пользователь *A* послал данное, занумерованное `i`;

`Ri` — пользователь *B* принял данное, занумерованное `i`.

Опишем процессы пользователей протокола. В этих процессах в нужных местах приписываем атомарным предикатам нужные значения: `s0=true`, когда посланы данные, занумерованные 0, `s1=true`, когда посланы данные, занумерованные 1, и т. д.

```
04 proctype UserA() {
05   q0: c1! d0; s0 = true;  s1 = false;
06   q1: c1! d1; s0 = false; s1 = true;
07   q2: c1! d0; s0 = true; s1 = false; goto q1;
08 }
09
10 proctype UserB(){
11   q0: c4? d0; r0 = true; r1 = false;
12   q1: c4? d1; r0 = false; r1 = true;
13   q2: c4? d0; r0 = true; r1 = false; goto q1;
14 }
```

Процесс передатчика строится в точности по спецификации, представленной на рис. 5.4:

```
15 proctype Sender() { mtype msg;
16  q0: c1? d0;
```

```
17  q1: c2! d0;
18  q2: if
19      :: c6? ACK -> qoto q3
20      :: true  -> goto q1 /* передатчик не дождался подтверждения */
21      fi;
22  q3: c1? d1;
23  q4: c2! d1;
24  q5: if
25      :: c6? ACK -> qoto q0
26      :: true  -> goto q4 /* передатчик не дождался подтверждения */
27      fi;
28 }
29
30 proctype Receiver() {
31 /* процесс Receiver строится полностью аналогично процессу Sender */
```

Процессы, моделирующие поведение каналов, учитывают возможность потери передаваемой информации:

```
31 proctype CHANNEL1() { mtype msg;
32   q0: c2? msg;
33   if
34   :: msg == d0; ->
35      if
36      :: c3! d0
37      :: true -> skip
38      fi
39   :: msg == d1; ->
30      if
31      :: c3! d1
32      :: true -> skip
33      fi
34   fi;
35   goto q0
36 }
37 proctype CHANNEL2() { mtype msg;
38  /* процесс Channel2 строится полностью аналогично процессу Channel1 */
39 }
```

К этой программе нужно добавить процесс инициализации `init`, запускающий все 6 процессов.

Формально требования спецификации протокола в *гл. 5* были представлены формулами LTL, в которых, в соответствии с нотацией системы Spin, темпоральные операторы **G** и **F** обозначаются [] и <>:

Spec1:

$$[]\big(s0 -> \big(!s1\mathbf{U}r0\big)\big) \wedge []\big(r0 -> \big(!r1\mathbf{U}s1\big)\big) \wedge []\big(s1 -> \big(!s0\mathbf{U}r1\big)\big) \wedge []\big(r1 -> \big(!r0\mathbf{U}s0\big)\big)$$

Spec2: $[]\big(\big(s0 -> <> s1\big) \wedge \big(s1 -> <> s0\big)\big)$

Анализ с помощью системы Spin этих требований позволяет найти контрпримеры — ошибочные траектории, на которых требования не выполняются.

7.3. Протокол двухфазной фиксации

Одним из методов обеспечения целостности данных в распределенных базах данных является подтверждение выполнения обработки транзакции в соответствии с протоколом двухфазной фиксации (2PC, Two Phase Commit Protocol). Этот протокол используется в случаях обработки транзакции несколькими серверами в распределенной базе данных (БД). Всем активным серверам нужно принять единое коллективное решение либо о том, что все они успешно завершили обработку транзакции, либо о неудаче, если хотя бы один сервер не смог довести обработку до конца и транзакция должна быть абортирована. Протокол состоит из двух фаз.

- Фаза1: подготовка

  Менеджер транзакций (например, Microsoft Distributed Transaction Coordinator, MS DTC) сначала проверяет готовность всех серверов к завершению транзакции, запрашивая их решение. Каждый сервер, участвующий в обработке транзакции, посылает свой ответ менеджеру транзакций, информируя его либо о безошибочном завершении, либо о необходимости аборта (отклонения) обрабатываемой транзакции вследствие обнаруженной им ошибки.

- Фаза2: фиксация или откат (commit or rollback)

  Если все серверы сообщили, что они завершили обработку транзакции безошибочно, менеджер транзакций отправляет всем серверам команду окончательной фиксации этой транзакции, разрешая записать в БД сделанные при обработке транзакции изменения данных. Если хотя бы один из серверов при выполнении транзакции обнаружил ошибку и направил об этом информацию менеджеру, менеджер направляет всем серверам команду об аборте транзакции. Все изменения данных, сделанные при обработке этой транзакции, каждым сервером игнорируются (не записываются в БД), и транзакция запускается на обработку повторно (используя механизм множественных поколений, или реинкарнаций).

Приведем модель протокола 2PC как программу на языке Promela. При построении модели абстрагируемся от несущественных деталей протокола, на-

пример, зафиксируем число активных процессов, решение о том, правильно или неправильно обработана транзакция будем принимать случайно, будем считать, что передача сообщений синхронна — каждое посланное сообщение должно ожидаться получателем.

```
 1 #define N 6                              /*  число активных процессов  */
 2 mtype = {START, COMMIT, ABORT, NODECISION}; /*символьные константы*/
 3 chan Mchan = [0] of {mtype},    /* канал от менеджера к процессам */
 4      Pchan = [0] of {mtype};    /* канал от процессов к менеджеру */
 5 mtype GlobalDecision = NODECISION;        /* коллективное решение */
 6
 7 proctype Manager(){ byte count = 0;
 8      /* count считает число взаимодействий с активными процессами */
 9  do  /* ШАГ1: запрос ко всем активным процессам принять решение*/
10  :: (count < N) -> Mchan ! START; count ++
11  :: (count == N) -> break
12  od;
13
14      /* ШАГ2: прием решений от всех активных процессов */
15 GlobalDecision = COMMIT; /* сначала полагаем общее решение — COMMIT
16 Это решение останется, если ни один из процессов не пришлет ABORT */
17  mtype vote;    /* vote — переменная для приема решений процессов */
18  do
19  :: (count < 2*N ) -> Pchan ? vote; count ++;
20    if
21    :: (vote == ABORT) -> GlobalDecision = ABORT
22    :: (vote == COMMIT) -> skip
23      /* если хотя бы один процесс прислал ABORT, то решение ABORT */
24    fi
25  :: (count == 2*N) -> break
26  od;
27  /* ШАГ3: рассылка общего решения всем активным процессам */
28  do
29  :: (count < 3*N) -> Mchan ! GlobalDecision; count ++
30  :: (count == 3*N) -> break
31  od
32 }
33
34 proctype Active_process (byte id) /*активный процесс с номером id */
35 { mtype decision = NODECISION;
36       /*  сначала у процесса решения нет  */
37  Mchan ? START ->  /* стартует, когда получен запуск от менеджера */
```

```
38   if
39     /*  случайно выбираем свое решение и отправляем его менеджеру */
40     /* потом ожидаем общее решение от менеджера  */
41   :: Pchan ! ABORT;  Mchan ? decision; decision = ABORT
42   :: Pchan ! COMMIT; Mchan ? decision
43   fi; assert (decision == GlobalDecision)
44   /* если мы решили ABORT, общее решение должно быть только ABORT */
45 }
46           /* Оставшаяся часть программы инициирует процессы */
47 init      /* инициатор запускает N активных процессов и менеджер */
48 { byte count = 0;
49   do
50   :: (count  < N) -> run Active_process (count); count++
51   :: (count == N) -> break
52   od;
53   run Manager();
54 }
```

Здесь для верификации использован оператор `assert(логическое выражение)` в строке 43. Этот оператор всегда вычислим. Если логическое выражение истинно на всех траекториях в процессе выполнения программы, то вычисление этого оператора не имеет никакого эффекта (см. пример 5.2). Если же на какой-нибудь траектории вычисления это выражение ложно, то Spin выдает информацию об ошибке. Использование такого оператора в данном протоколе позволяет его верифицировать: корректное поведение протокола 2PC состоит в том, что при всех возможных вариантах поведения активных процессов решение каждого процесса совпадает с общим решением, но если процесс принял решение `ABORT`, то и общее решение будет `ABORT`. Именно это проверяется здесь оператором `assert`.

7.4. Заключение

Приведенные в этой главе примеры показывают, как можно в инструментальной системе Spin верифицировать параллельные программы и протоколы. Этот инструмент, как и многие другие, реализующие алгоритмы верификации на основе техники model checking, был использован для верификации множества систем для критических применений значительной сложности и доказал свою эффективность. Существует несколько проблем, с которыми сталкивается любой профессионал, выполняющий верификацию с помощью техники проверки модели. Перечислим основные из них.

- Каждая система автоматической верификации (Spin, SMV, PRIZM и др.) имеет свой собственный язык описания поведения верифицируемых сис-

тем. Построение адекватной модели на входном языке инструмента верификации требует квалификации, искусства и опыта. Справедливо следующее утверждение: "*Верификация системы на основе метода проверки модели полезна настолько, насколько адекватна построенная для анализа модель системы*". Часто очень непросто дать гарантию того, что построенная модель действительно адекватна, что она корректно представляет анализируемую систему.

- То же относится и к спецификации требований. Каждый инструмент верификации имеет свою собственную нотацию для записи проверяемых свойств. Обычно эта нотация представляет собой язык для построения формул некоторого подмножества темпоральных логик CTL, CTL* или LTL. Свойства систем, которые желательно проверить, не всегда возможно выразить в этой нотации. Выражение многих свойств также требует достаточно высокой квалификации и опыта.
- Очень часто число состояний анализируемых моделей слишком велико, и результатом запуска инструмента верификации может быть его останов из-за нехватки памяти. В этом случае для верификации системы необходима дополнительная работа по ее абстрагированию, отбрасыванию второстепенных деталей, ограничению числа параметров или области принимаемых ими значений. В сложных случаях может оказаться необходимым уменьшить число верифицируемых модулей, или проводить верификацию модулей независимо.
- Если при верификации найдена ошибка и выдан контрпример, то часто нелегко установить причину ошибки: выдаваемые трассы могут оказаться очень длинными и сложными, их анализ может потребовать глубокого проникновения в существо проверяемых алгоритмов. Иногда причиной ошибки может явиться именно ограниченность модели, а вовсе не некорректность верифицируемой системы.

7.5. Замечания

Система верификации Spin для верификации протоколов передачи данных стала использоваться с начала 90-х годов прошлого века. С тех пор с помощью этой системы были проверены многие промышленные разработки для критических применений, в частности, модули бортовой системы управления Deep Space 1. Многие университеты используют Spin в учебном процессе не только в курсах верификации, но и в курсах, изучающих параллельные и распределенные алгоритмы для моделирования и наглядного представления взаимодействующих процессов.

7.6. Задачи к главе 7

7.1. Проверьте корректность другой формулировки протокола взаимного исключения Петерсона:

```
P1::
m0:  NC1;
m1:  y1 :=true;
m2:  t := false;
m3:  wait (¬y2∨t);
m4:  CS1;
m5:  y1 := false;
m6:  goto m0;
```

```
P2::
n0:  NC2;
n1:  y2 :=true;
n2:  t := true;
n3:  wait (¬y1∨¬t);
n4:  CS2;
n5:  y2 := false;
n6:  goto n0;
```

Спецификация этой программы:

а) Безопасность (взаимное исключение всегда выполняется):

$$\mathbf{G}\neg\left(P1@m4 \wedge P2@n4\right)$$

б) Живость (прогресс процессов):

$$\mathbf{G}\left(P1@m3 \Rightarrow \mathbf{F}P1@m4\right)$$

$$\mathbf{G}\left(P2@n3 \Rightarrow \mathbf{F}P2@n4\right)$$

7.2. Проверьте корректность алгоритма взаимного исключения, написанного на языке Promela:

```
bool  turn, flag[2];
byte  ncrit;
active proctype[2] user()
{
assert ( _pid == 0 ||  _pid ==1 );
start:
     flag[ _pid ] = 1;
     turn = _pid;
     (flag [1-_pid] == 0 || turn == 1 − _pid);
     ncrit++;
     assert (ncrit == 1);
     /* critical section */
     ncrit --;
     flag [_pid ] = 0;
     goto start;
}
```

7.3. Алгоритм взаимного исключения с названием TicketMe, приведенный в [104], описывается так:

"Разделяемая переменная содержит пару скаляров: <*next, granted*>, каждый из которых принимает значения {1,2, ..., *n*}. Вначале ее значение <1, 1>. Компонент *next* этой переменной представляет собой следующий "пропуск" в критическую секцию, компонент *granted* — это последний выданный "пропуск" в критическую секцию. Когда процесс хочет войти в свою критическую секцию, он "берет пропуск", т. е. копирует себе компонент *next* разделяемой переменной и увеличивает его по модулю *n*. Когда номер процесса равен значению компонента *granted*, процесс входит в критическую секцию. Когда процесс выходит из критической секции, он увеличивает значение компонента *granted* по модулю *n*".

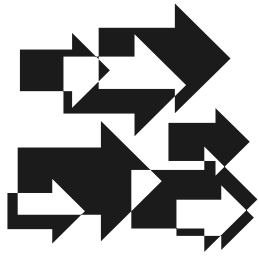
Проверьте корректность этого алгоритма.

7.4. Проверьте корректность еще одной формулировки протокола взаимного исключения Петерсона:

```
Process0::
  flag0 :=1; turn :=1;
  while (flag1 and turn = 1) wait;
  CriticalSectioon0;
flag0 :=0;
```

```
Process1::
  flag1 :=1; turn :=0;
  while (flag0 and turn = 0) wait;
  CriticalSectioon1;
flag1 :=0;
```

ГЛАВА 8



Применения алгоритма model checking

Подход к верификации на основе метода *model checking* состоит в том, что для модели системы формально проверяется выполнение логической формулы. Полученные в области верификации фундаментальные результаты удивительным образом оказались востребованы в огромном числе приложений, в которых идеи model checking применяются для решения самых разнообразных задач. В данном разделе мы рассмотрим только некоторые из этих приложений.

8.1. Анализ бизнес-процессов

Проблема формального анализа требуемых от бизнес-процессов свойств с помощью метода model checking явилась предметом исследования нескольких научных групп. Компания в этих исследованиях рассматривается не как совокупность отделов и групп, а как множество бизнес-процессов, вокруг которых реализуется деятельность компании. Бизнес-процесс можно определить как набор связанных операций, выполнение которых направлено на достижение целей компании.

Каждая компания создается для выполнения некоторых целей, поэтому в любой компании существуют бизнес-процессы, даже если они не описаны и не документированы. Внедрение формального описания бизнес-процессов позволяет построить системы управления качеством компании, решить проблему построения эффективной структуры управления, оптимизировать деятельность на основе ключевых показателей.

Бизнес-процесс — это не обязательно последовательность действий одного работника. Бизнес-процесс может реализовываться несколькими работниками в пределах одного подразделения компании, охватывать несколько ее

подразделений или даже несколько подразделений различных компаний. Поэтому бизнес-процессы представляются как параллельно протекающие активности многих исполнителей, последовательно выполняющих свои функции. Здесь возникают известные в параллельном программировании проблемы доступа к ресурсам, блокировки, ограничения живости, и поэтому результаты и решения в области верификации параллельных процессов востребованы и в области бизнес-процессов.

Приведем один из примеров исследования в этой области. Наличие большого числа взаимодействующих бизнес-процессов ставит задачу формальной проверки их правильности, что может быть выполнено методом model checking. Однако бизнес-процессы специфицируются менеджерами, которые не имеют подготовки в области формальных моделей и информатики, а для формального анализа бизнес-процессов необходимо детальное представление модели процесса на формальном языке, которое трудно построить и которое нелегко понять менеджерам.

Для разрешения этой проблемы авторы работы [83] предлагают использовать для спецификации бизнес-процессов простой графический формализм, который близок графическому изображению параллельных активностей картами состояний (statecharts) Харела и сетями Петри, а также диаграммами активностей языка UML (UML activity diagrams). Транслятор с этого языка в язык Promela — входной язык системы верификации Spin — позволяет проверить свойства бизнес-процессов.

Предлагаемый авторами графический язык, названный Amber (Architectural Modelling Box for Enterprise design), позволяет представить бизнес-процессы, агенты, участвующие в процессах, и данные, с которыми процессы оперируют. Поведение структурировано в виде процессов, взаимодействующих с помощью синхронной коммуникации (хэндшейка). Свойства процессов специфицируются с помощью формул LTL. Авторы разработали формы, позволяющие упростить такую спецификацию, поскольку формулы логики LTL, по их мнению, также являются слишком сложными для использования менеджерами.

Язык Amber позволяет выразить действия, которые выполняются в бизнес-процессах, и отношения между этими действиями. Графические элементы, связывающие элементарные шаги процессов, следующие:

- ■ — AndJoin — процесс продолжается, если все входные действия этого блока выполнены;
- □ — OrJoin — процесс продолжается, если хотя бы одно из входных действий этого блока выполнено;
- ♦ — AndSplit — все выходные процессы этого блока запускаются;

◊ — OrSplit — по крайней мере один из выходных процессов этого блока запускается.

Для структуризации процессов используется объединение действий в модули, графически представляемые прямоугольниками, включающими в себя элементы модуля. Модули могут быть связаны синхронной коммуникацией.

Пример спецификации бизнес-процессов страховой компании с помощью графического языка Amber показан на рис. 8.1. В этой страховой компании существует несколько бизнес-процессов: процесс Garage, описывающий действия и их связи по ремонту автомашины после аварии, процесс Pro-fit, моделирующий прохождение бумаг в компании, процесс клиента, подающего документы на возмещение ущерба и получающего машину после ремонта, и др. Типичные свойства, которые могут быть проверены для этих бизнес-процессов:

P1: *Машина ремонтируется только после того, как принято решение об удовлетворении иска, причем это решение может быть и не принято*:

$$\mathbf{G}\left((\neg damage_repaired)\,\mathbf{W}\,claim_approved\right)$$

где $\mathbf{W}$ — слабый Until.

P2: *Иск до 10 тысяч долларов для клиента N104 всегда будет удовлетворен:*

$$\mathbf{G}\left((claim_below_10 \wedge customer_104) \Rightarrow \mathbf{F}\,claim_approved\right)$$

P3: *Из гаража машина возвращается владельцу только после ее ремонта:*

$$\mathbf{G}\left((\neg deliver_car)\,\mathbf{W}\,damage_repaired\right)$$

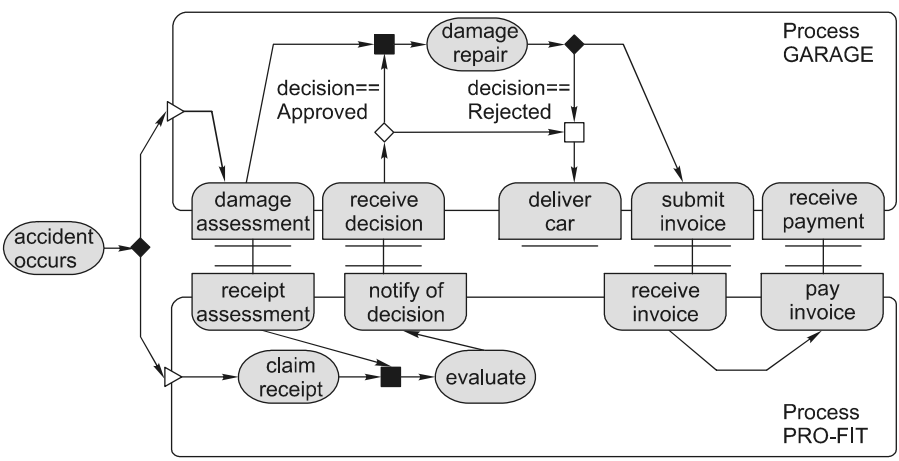


Рис. 8.1. Графическое представление бизнес-процессов страховой компании

P4: Если иск отвергнут, машину не будут ремонтировать:

$$\mathbf{G}\left((claim_rejected \Rightarrow \mathbf{G}\neg damage_repaired)\right)$$

P5: Гараж не представит счет за ремонт, если иск отвергнут:

$$\mathbf{G}\left((claim_rejected \Rightarrow \mathbf{G}\neg submit_invoice)\right)$$

8.2. Проблема планирования как model checking

Проблема планирования — это классическая проблема искусственного интеллекта. Она состоит в поиске такого плана целесообразной деятельности в некоторой среде, который приведет к заданной цели. Более точное определение обычно определяет среду как дискретное множество состояний и переходов между ними, а проблему планирования — как поиск конечной последовательности действий, которая переводит из начального состояния в одно из состояний, которые удовлетворяют поставленной цели.

В классической теории планирования приняты следующие предположения:

- состояния среды дискретны, каждое состояние полностью наблюдаемо и множество состояний конечно;
- время решения и выполнения действий не учитывается;
- возможные действия по изменению состояний известны, каждое действие приводит к единственному новому состоянию;
- начальное состояние полностью определено;
- цель определена явно как одно состояние или множество заключительных состояний;
- план — последовательность действий — строится один раз, изменение среды в процессе выполнения плана определяется только выполняемыми действиями.

Решением проблемы планирования является путь от начального состояния до целевого состояния. В некоторых случаях искомый путь должен удовлетворять некоторому набору ограничений. Часто с решением связывают его качество, т. е. некоторую функцию стоимости пути. Тогда поиск решения состоит в нахождении оптимального пути, который имеет наименьшую стоимость среди всех возможных решений.

Одной из главных причин сложности этой проблемы является огромный размер пространства возможных последовательностей шагов. Для построения алгоритма планирования, который мог бы найти лучший путь среди огромно-

го числа альтернативных путей, требуются методы для представления пространственного знания: структуры среды, связей и переходов между состояниями, а также управления перебором в пространстве вариантов.

Существуют различные методы синтеза плана из формальной модели и логической спецификации цели, которые реализуются в алгоритме планировщика. В общем виде процесс планирования представлен на рис. 8.2. Описание среды — конечного множества состояний и их связей, а также описание цели являются входом в систему планирования, которая либо не находит требуемого плана, либо находит его и выдает как путь из начального состояния в целевое состояние.

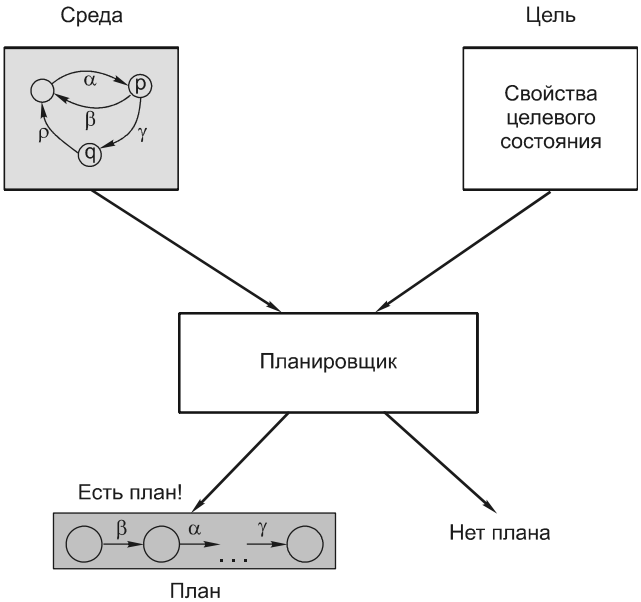


Рис. 8.2. Система планирования

Представленный таким образом процесс планирования очень похож на процесс верификации (рис. 8.3): описание системы — конечное множество состояний и переходов, а также описание проверяемого свойства являются входом в систему верификации. Верификатор либо заключает, что заданное свойство выполняется, либо, если оно не выполняется, выдает контрпример — такой путь из начального состояния, на котором свойство не выполняется.

Если среду представить как структуру Крипке M, а состояния цели описать с помощью логического условия φ, то в простейшем случае задачу планирования можно переформулировать как задачу верификации — проверку вы-

полнения свойства, выраженного формулой **AG**¬φ линейной темпоральной логики LTL на структуре M (на всех путях из начального состояния структуры M свойство φ никогда не встретится). Верификатор будет искать опровержение, т. е. путь в M, удовлетворяющий отрицанию ¬**AG**¬φ этой формулы, т. е. формуле **EF**φ. Если верификатор найдет опровержение, то это и будет решением задачи планирования. Таким образом, система верификации не только сообщит, что существует путь, на котором будет выполняться формула **EF**φ (т. е. свойство φ может выполниться когда-нибудь в будущем), но и выдаст один из возможных путей как контрпример (рис. 8.3).

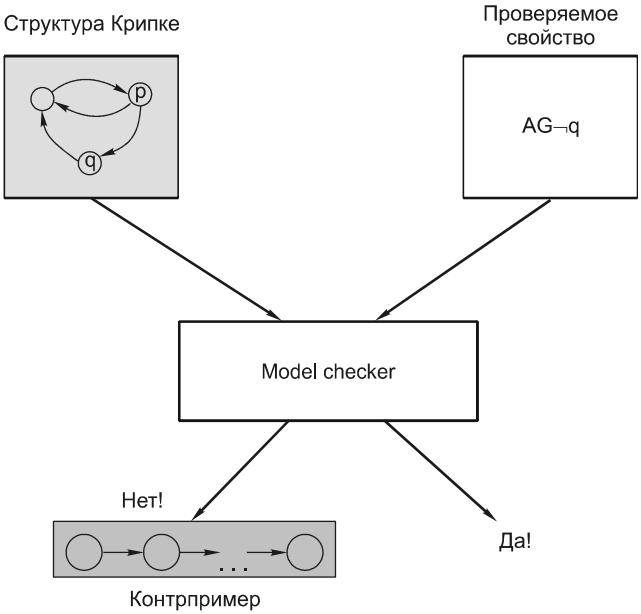


Рис. 8.3. Система верификации

Если искомый путь к цели должен удовлетворять ограничениям, запишем эти ограничения с помощью логической формулы ψ, которая должна выполняться во всех состояниях на пути до цели. Верификатор в этом случае может быть использован для проверки условия $\mathbf{A}\neg(\psi\mathbf{U}\varphi)$ (*не существует такого пути, на котором в будущем может быть достигнуто состояние, удовлетворяющее* φ*, а до этого во всех состояниях пути выполняется свойство* ψ). Верификатор будет искать путь, удовлетворяющий формуле $\neg\mathbf{A}\neg(\psi\mathbf{U}\varphi) = \mathbf{E}(\psi\mathbf{U}\varphi)$, и если такой путь будет найден, то выдаст его в качестве контрпримера.

Таким образом, идеи model checking и формулы темпоральной логики для спецификации цели и ограничений при поиске цели могут быть использованы для решения проблем планирования. Ниже мы рассмотрим пример, в котором идеи model checking применяются для решения простой проблемы этой области. В этом примере мы используем стандартный подход model checking без какой-либо оптимизации и эвристики. Вместо того чтобы представлять проблему в виде структуры Крипке, будем использовать систему верификации Spin и представлять модель проблемы на языке Promela. На этом языке легко определяется множество возможных состояний среды (как множество всех возможных наборов значений переменных, описывающих среду), можно задать начальное и целевое состояния (как наборы значений параметров среды), а также определить правила изменения состояний. Проверяемым условием является следующее: "*целевое состояние среды недостижимо из ее начального состояния*". В качестве контрпримера Spin находит и выдает план — последовательность шагов, приводящих из начального в целевое состояние, удовлетворяющее ограничениям.

Мир блоков (*blocks world*). Мир блоков — это искусственная среда планирования действий, которая из-за своей ясности и простоты является одним из наиболее часто приводимых примеров планирования в искусственном интеллекте.

Мир блоков состоит из плоской поверхности (стола), на которой расположено несколько именованных блоков $(A, B, C, ...)$. Взаимное расположение блоков (конфигурация) является состоянием среды. Задача планирования — найти план, который позволит перевести начальную конфигурацию в целевую. За один шаг может быть перемещен только один блок, на котором ничего не стоит: он может быть либо поставлен на стол, либо сразу поставлен сверху на другой блок. Оптимальный план — это план, имеющий наименьшее число шагов.

На рис. 8.4, *а* представлен пример такого задания. Рис. 8.4, *б* показывает все пространство состояний и переходы между ними для трех блоков. План — решение задачи — состоит в любой последовательности допустимых переходов из начального в целевое состояние.

Легко построить программу на языке Promela, описывающую все возможные конфигурации и допустимые их изменения для любого числа блоков и любого их расположения. Число блоков определим параметром `N`, каждый блок будет идентифицироваться уникальным номером от `0` до `N-1`. Конфигурацию будем задавать двумя массивами длины `N`: `up[i]` определяет номер блока, стоящего над `i`-м, `down[i]` определяет номер блока, стоящего под `i`-м. Специальная константа `NOTHING` пусть определяет, что над или под блоком ничего

не стоит. Целевая конфигурация определяется подобными массивами `T_up` и `T_down`.

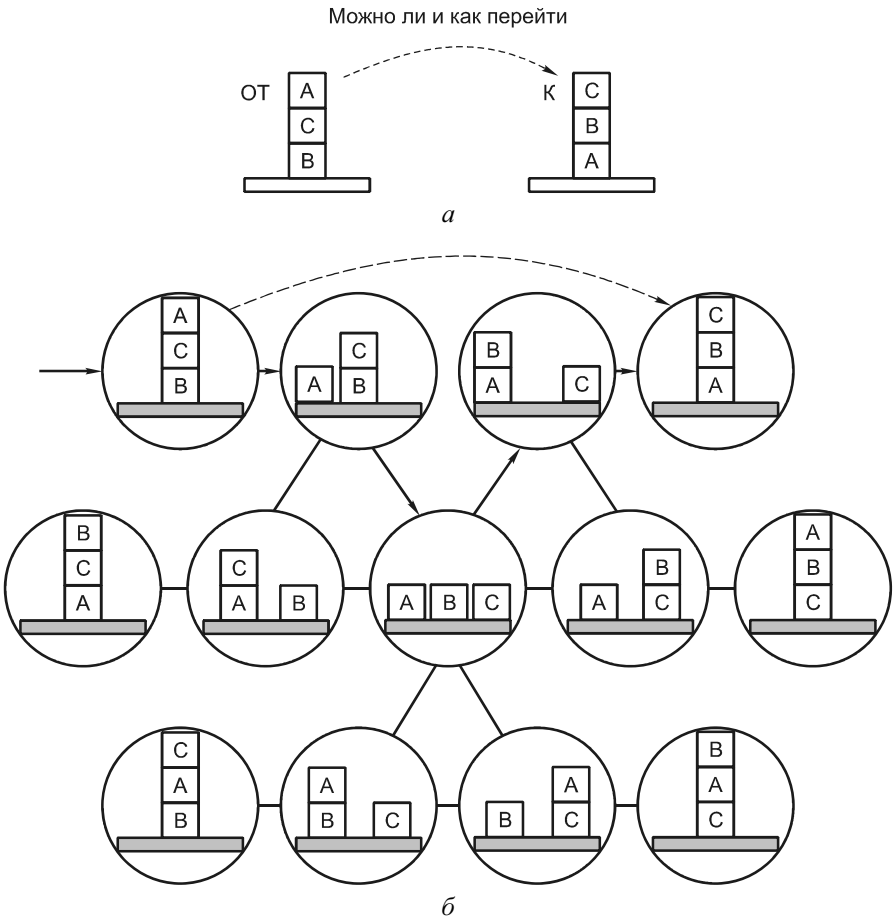


Рис. 8.4. Мир блоков: *а*) задача; *б*) пространство состояний

Исходная конфигурация на рис. 8.4 будет представлена так: `up[0]=NOTHING`, `up[1]=2`, `up[2]=0`, `down[0]=2`, `down[1]=NOTHING`, `down[2]=1`. Возможные переходы между состояниями программируются так:

1. Выбирается случайный блок `i`, стоящий сверху (т. е. если `up[i]==NOTHING`), и этот блок снимается (`up[down[i]] = NOTHING`).
2. Выбранный блок либо кладется на стол (`down[i]=NOTHING`), либо выбирается другой блок `j`, отличный от `i` (`j!=i`), который стоит сверху (`up[j]==NOTHING`), и на него помещается блок `i` (`up[j]=i; down[i] = j`).

Проверяемое условие задается как совпадение текущей и целевой конфигурации, т. е. совпадением массивов `up[]` и `T_up[]`, а также `down[]` и `T_down[]`. Программа в цикле сначала проверяет, совпали ли текущая и целевая конфигурации, и если нет, то случайно делает один из возможных переходов — изменяет текущее состояние (конфигурацию) на соседнее.

Программа на языке Promela для решения этой задачи приведена в Руководстве по языку Promela и использованию системы верификации Spin, приложенному к этой книге.

Одна из главных трудностей проблемы планирования — та, что для практических задач пространство поиска огромно. Эта трудность преодолевается использованием символьного алгоритма model checking на основе бинарных решающих диаграмм, который мы будем изучать в следующих главах.

В настоящее время model checking используется в планировании очень широко. Современные системы планирования, основанные на этих идеях, используют мощные схемы, объединяющие методы символьной верификации (*см. гл. 10*) и классические методы эвристического поиска в пространстве состояний. На основе идей model checking разработаны алгоритмы решения сложных проблем построения планов, удовлетворяющих требованиям, выраженным произвольными формулами темпоральной логики. Примером с автоматической генерацией подобного плана является приведенная в следующем разделе задача о фермере, волке, козе и капусте. По исследованиям в этом направлении проводится несколько ежегодных конференций, разработаны мощные системы планирования, интегрировавшие model checking с классическими методами эвристического поиска (BDDPlan, CIRCA, MBP, Proplan и др.). Существует множество успешных примеров практического применения этого подхода к задачам планирования. В частности, в заявлении организационного комитета Международного семинара "Planning via Model Checking ’02" говорится:

"*В настоящее время model checking является одной из наиболее “горячих” тем информатики. Идея заключается в том, что модель системы проверяется на выполнение для нее логических требований. Изначально эта идея была использована для проверки корректности схем и компьютерных протоколов. Недавно эта же идея была применена для задач планирования в области искусственного интеллекта с замечательным успехом и привела к созданию мощных систем планирования*".

8.3. Логические головоломки и model checking

Большое число логических головоломок состоит в поиске последовательности действий, которая приводит некоторую систему из заданного состояния в

некоторое требуемое состояние. Такие головоломки можно считать частным случаем проблемы планирования. Рассмотрим несколько примеров.

Китайские кольца (Chinese rings) — это старинная механическая головоломка, которая состоит в том, что заданы N колец, соединенных друг с другом и надетых на горизонтальную вытянутую проволочную петлю. К каждому кольцу прикреплен стержень (рис. 8.5, *а*). Цель головоломки — построить план действий, в результате которых все кольца будут сняты с петли.

Для решения задачи нужно учесть следующее. Первое кольцо всегда можно снять с петли или надеть на нее (рис. 8.5, *б*). Любое кольцо можно снять или надеть на петлю, только если ближайшее предыдущее находится на петле, а все кольца до этого предыдущего уже сняты (рис. 8.5, *в*).

Используем для решения головоломки систему верификации Spin. Будем рассматривать эту систему колец как дискретную динамическую систему. Мы можем описать все возможные изменения состояний системы, ее начальное и целевое состояния, и проверить выполнение у системы свойства: *целевое состояние недостижимо*. Если это условие неверно, то система верификации выдаст опровержение — контрпример, который и будет искомым планом действий.

Состояние системы "Китайские кольца" определим как набор значений булевых переменных $x_1, x_2, ..., x_n$, где x_i — состояние i-го слева кольца ($x_i = true$ — надето, $x_i = false$ — снято), n — число колец. Начальное состояние — все значения x_i равны $true$, целевое состояние — все значения x_i равны $false$. Переходы в каждом состоянии определяются так: если все $x_1, x_2, ..., x_{k-1} = false$ (эти кольца сняты), а $x_k = true$ (k-е кольцо надето), то значение переменной x_{k+1} можно изменить на противоположное (т. е. кольцо $k+1$ можно либо снять, либо надеть). *Проверяемое свойство*: $\mathbf{AG}\left(x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5\right)$ — на всех путях во всех состояниях всегда хотя бы одно кольцо останется, т. е. никакая последовательность действий НЕ приведет к целевому состоянию, в котором все кольца сняты.

Основной фрагмент программы решения этой проблемы на языке Promela для 5 колец выглядит так:

```
bool x1, x2, x3, x4, x5 = true;      /* вначале все кольца надеты */

do
:: true → x1=¬x1   /* первое кольцо можно всегда снять/надеть*/
:: x1    → x2=¬x2  /* если 1 надето, то 2 можно снять/надеть */
::¬x1&&x2 → x3=¬x3 /* если 1-снято, 2-надето, то 3 можно снять/надеть */
```

```
::¬x1&&¬x2&&x3 → x4=¬x4
       /* если 1-е и 2-е-сняты,а 3-надето, то 4 можно снять/надеть*/
::¬x1&&¬x2&&¬x3&&x4 → x5=¬x5       /* 5-е можно снять/надеть  */
::¬x1&&¬x2&&¬x3&&¬x4&&¬x5 → break /* останов, все кольца сняты */
od
```

Система Spin выдает контрпример — искомый путь к целевому состоянию из 18 шагов:

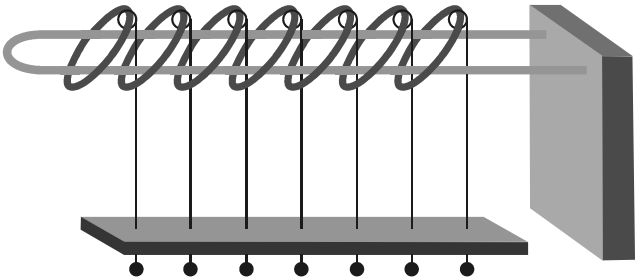
```
11111 → 01111 → 01011→ 11011 → 10011 → 00011 → 00010 → 11010 → 01010
→ 01110 → 11110 → 00110 → 00100→11100→01100→01000→11000→00000
```

Логические головоломки, связанные с переправами через реку. Такие головоломки имеют более чем тысячелетнюю историю. Они были известны еще в Древнем Египте. В наше время подобные задачи встречаются не только в школьных математических кружках, но и при приеме на работу (в частности, такие задачи задавались на собеседованиях при найме сотрудников в фирму "Майкрософт"). Мы рассмотрим здесь решение с помощью model checking двух из них; некоторые другие формулировки приведены как упражнения к этой главе.

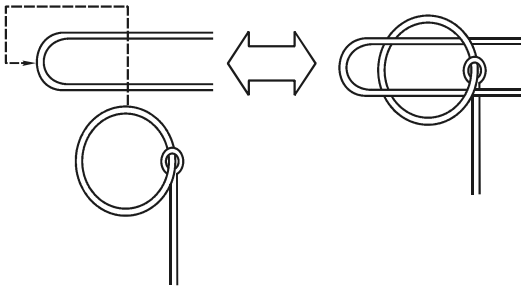
Проблема "фермер, волк, коза и капуста". Эта головоломка состоит в том, что фермер должен переправить через реку с левого берега на правый волка, козу и мешок капусты. Кроме фермера, который только один и может управлять лодкой, лодка может выдержать дополнительно что-то одно — либо волка, либо козу, либо мешок капусты. Решение проблемы осложняется тем, что без присмотра фермера нельзя козу оставлять с волком, а капусту — с козой.

Хотя среда, в которой принимаются решения, совсем не дискретная, для решения этой проблемы можно построить абстрактную дискретную модель — структуру Крипке, и исследовать на ней возможные пути. Эта абстрактная модель не будет учитывать ни времени нахождения системы в каком-либо состоянии (например, пока фермер отдыхает после очередной переправы), ни времени переправы. Переходы, представляющие собой переправы, в модели будем считать неделимыми (рис. 8.6).

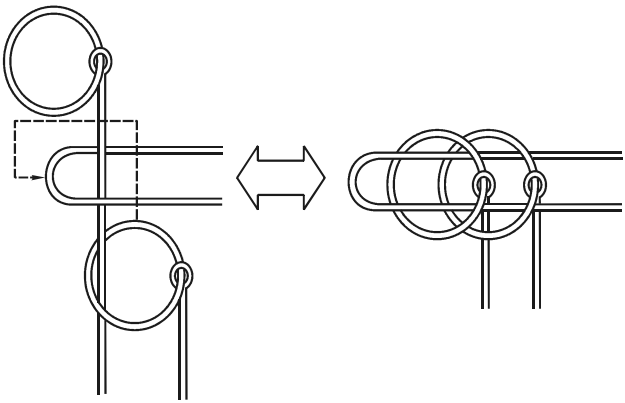
Такая абстракция не учитывает проблем с затаскиванием в лодку козы, проблем с течением реки и т. п. Для решения задачи эти факторы не важны, переходы в модели неделимы *с точки зрения проблемы: составления плана переправы*. Таким образом, абстрактная дискретная модель переходов (структура Крипке) является адекватной системе объектов *<фермер, волк, коза, капуста>* и ее динамике для решения задачи планирования переправы, хотя сама система с козой, голодным волком, течением реки и т. п. вовсе не является дискретной.



а



б



в

Рис. 8.5. *а*) Головоломка "Китайские кольца";
б) первое кольцо всегда можно снять либо надеть; *в*) если n – 1 колец сняты, а n — нет, то n + 1 кольцо можно снять либо надеть

Состояние: w ⟨⟩ fgc = <f, ¬w, g, c>

волк на левом берегу,
фермер, коза и капуста на правом берегу

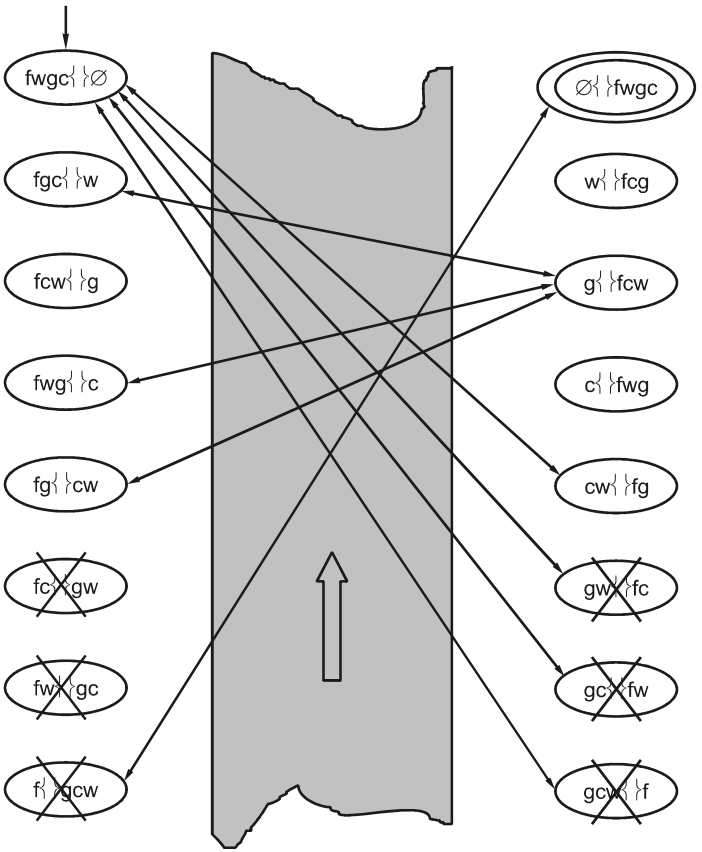


Рис. 8.6. Проблема переправы через реку фермера, волка, козы и капусты

Состоянием среды будем считать состояние всех четырех действующих объектов. Введем булевы переменные: f описывает состояние фермера, w — состояние волка, g — состояние козы и c — состояние капусты. Их значения: $f = false$ — фермер на левом берегу, $f = true$ — фермер на правом берегу; то же для волка, козы и капусты. Таким образом, состояние $\langle f = true, w = false, g = false, c = true \rangle$ говорит о том, что фермер с капустой уже находятся на правом берегу, а волк с козой ждут на левом. Будем такое состояние обозначать $\langle f, \neg w, \neg g, c \rangle$. Начальное состояние — все на левом берегу, целевое состояние — все на правом берегу.

Переходы структуры Крипке определяют возможные изменения состояний системы, т. е. переправы через реку. Эти переправы характеризуются тем, что только фермер может управлять лодкой. Поэтому, например, из состояния $\langle f, \neg w, \neg g, c \rangle$ можно перейти только в два состояния: в состояние $\langle \neg f, \neg w, \neg g, c \rangle$ (фермер, который находился на правом берегу, переехал на левый берег один) или в состояние $\langle \neg f, \neg w, \neg g, \neg c \rangle$ (фермер переехал на левый берег с капустой, которая находилась на том же берегу, что и он).

В отличие от предыдущей задачи, на искомый путь здесь должно быть наложено ограничение. Его можно выразить формулой $(g \equiv w) \vee (g \equiv c) \Rightarrow (g \equiv f)$ — если коза находится на том же берегу, что и волк или капуста, то фермер находится на этом же берегу. Наша задача — найти такой путь в структуре Крипке, на котором выполняется формула $((g \equiv w) \vee (g \equiv c) \Rightarrow (g \equiv f))\mathbf{U}(fwgc)$ — когда-нибудь в будущем все переберутся на правый берег, а до этого во всех состояниях пути соблюдается требуемое ограничение. На рис. 8.6 эти запрещенные состояния перечеркнуты.

Используем для нахождения плана переправы систему верификации Spin. Описание этой задачи на языке Promela чрезвычайно просто:

```
bool f, c, g, w = false;   /* начальное состояние */

do
  /* из возможных альтернатив недетерминировано выбирается одна    */
:: true  → f = ¬f                  /* фермер может переехать сам */
:: f == w → f = ¬f; w = ¬w         /* или с волком               */
:: f == c → f = ¬f; c = ¬c         /* или с капустой             */
:: f == g → f = ¬f; g = ¬g         /* или с козой                */
:: f && w && g && c → break /* останов, когда все переправятся */
od
```

Проверяемое условие: $\mathbf{A}\neg\Big(\big((g \equiv c) \vee (g \equiv w) \Rightarrow (g \equiv f)\big)\mathbf{U}\big(fwgc\big)\Big)$ — среди всех возможных путей не существует такого, на котором все переправятся на правый берег, и при этом во всех состояниях этого пути выполняется заданное ограничение.

Система верификации Spin выдаст контрпример, который и является искомым планом (см. рис. 8.7).

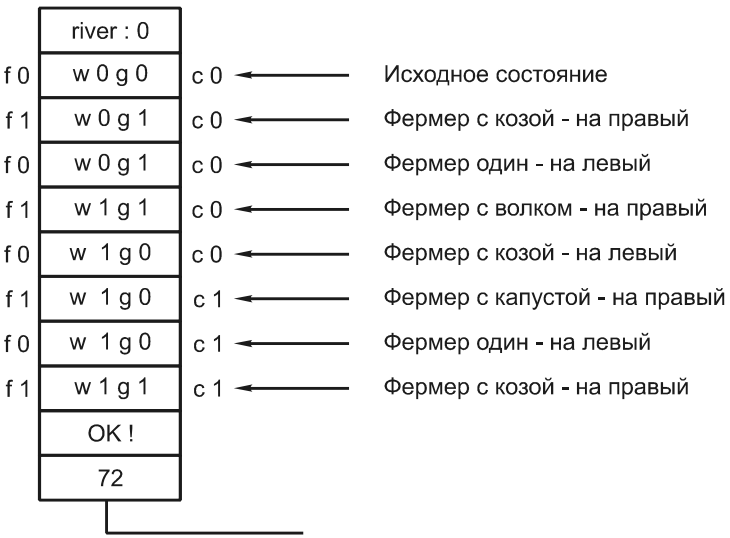


Рис. 8.7. План переправы через реку, сгенерированный системой верификации для проблемы "фермер, волк, коза, капуста"

Заметим, что, используя темпоральную логику, можно выразить многие интересные полезные свойства строящихся планов. Например, для проблемы с фермером и его командой:

- в заключительное состояние можно попасть: $\mathbf{EF}\big(fwgc\big)$;
- существует такое решение задачи, при котором, если коза переедет на правый берег, она там навсегда и останется: $\mathbf{E}\Big[\mathbf{F}\big(fwgc\big) \wedge \big((\neg g)\mathbf{UG}g\big)\Big]$;
- при любом варианте решения задачи фермер должен будет возвращаться через реку один:

$$\mathbf{A}\Big[\big((g \equiv c) \vee (g \equiv w) \Rightarrow (g \equiv f)\big)\mathbf{U}\big(fwgc\big) \Rightarrow \mathbf{F}\big((f \wedge \mathbf{X} \wedge \neg f) \wedge (w \equiv \mathbf{X}w) \wedge (g \equiv \mathbf{X}g) \wedge (c \equiv \mathbf{X}c)\big)\Big]$$

Проблема "миссионеры и каннибалы". Три миссионера и три аборигена-каннибала должны переплыть реку с левого берега на правый. Их лодка может выдержать не более двух человек. Каждый миссионер и каждый каннибал могут управлять лодкой, но если на одном берегу каннибалов окажется больше, чем миссионеров, то голодные аборигены могут позавтракать миссионерами. Проблема планирования состоит в построении такого плана переправы через реку, чтобы все миссионеры остались в живых, и при этом через реку должны переправиться и миссионеры, и аборигены.

Задача эта очень похожа на предыдущую. Состоянием здесь можно выбрать тройку: $\langle b,m,c\rangle$, где b — положение лодки ($b=1$ — лодка на левом берегу, $b=0$ — лодка на правом берегу), а m и c — число миссионеров и число каннибалов на левом берегу. Например, состояние $\langle 0,0,2\rangle$ говорит о том, что лодка на правом берегу, на левом берегу нет миссионеров и два каннибала (а три миссионера и один каннибал на правом берегу, потому что общее число миссионеров и общее число аборигенов не меняется).

Как и в предыдущей задаче, будем считать, что переходы между состояниями неделимы, они не важны для анализа поведения системы. Пример возможной траектории системы:

$\langle 1,3,3\rangle \to \langle 0,2,2\rangle \to \langle 1,3,2\rangle \to \langle 0,1,2\rangle$ — неудача: миссионера съели на левом берегу.

Запрограммируем все переходы в языке Promela, и с помощью системы Spin будем искать безопасное решение задачи. Для этого проверим в системе Spin утверждение: "*безопасного решения задачи не существует*", которое постараемся формализовать на языке формул линейной темпоральной логики.

Опишем проблему для произвольного числа N миссионеров и каннибалов:

```
# define N  ...
bool   b = false;            /* вначале лодка на левом берегу */
byte m = N;                  /* начальное число миссионеров */
byte c = N;                  /* начальное число каннибалов */

do                           /* все возможные переходы системы */
  ::  ¬b→b=¬b;       /* если лодка на левом берегу, ее можно использовать
                        так: */
    if
    :: m>0→m = m-1           /* миссионер переправился на правый берег */
    :: c>0→c = c-1           * каннибал переправился на правый берег */
    :: m>1→m = m-2           /* два миссионера переправились */
    :: c>1→c = c-2           /* два каннибала переправились */
```

```
    :: m>0&&c>0 → m=m-1;c=c-1    /* миссионер и каннибал переправились */
    fi;
:: b → b=¬b;          /* если лодка на правом берегу, ее можно
                         использовать так: */
    if
    :: m<N   → m=m+1          /* миссионер переправился на левый берег */
    :: c<N   → c=c+1          /* каннибал переправился на левый берег */
    :: m<N-1→ m=m+2           /* два миссионера переправились */
    :: c<N-1 → c=c+2          /* два каннибала переправились */
    :: m<N&&c<N → m=m+1;c=c+1 /* миссионер и каннибал переправились
                                 вдвоем */
    fi;
:: (m>0)&&(c>m) → break      /* миссионеров съели на левом берегу */
:: (m<N)&&(m>c) → break      /* миссионеров съели на правом берегу */
:: (m+c==0)     → break      /* все благополучно перебрались на правый
                                берег */
od
```

В качестве проверяемого условия выберем следующее:

$\mathbf{A}\neg\Big(\Big(\neg\big(m>0\,\&\&\,c>m\big)\,\&\&\neg\big(m<N\,\&\&\,m>c\big)\Big)\mathbf{U}\big(m+c=0\big)\Big)$ — на всех путях неверно, что можно всем переправиться $\big(m+c==0\big)$, и в процессе переправы не будет условий для съедения миссионеров.

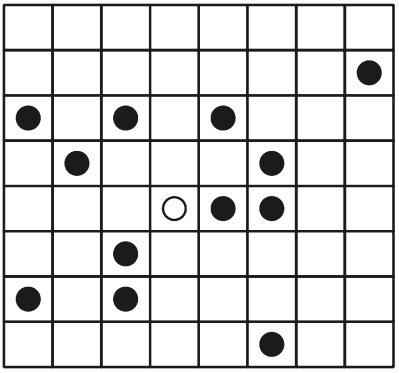
Для значения $N=3$ система верификации Spin выдает опровержение этого условия, т. е. безопасный план действий:

$$\langle 1,3,3\rangle\to\langle 0,3,1\rangle\to\langle 1,3,2\rangle\to\langle 0,3,0\rangle\to\langle 1,3,1\rangle\to\langle 0,1,1\rangle\to\langle 1,2,2\rangle\to$$
$$\langle 0,0,2\rangle\to\langle 1,0,3\rangle\to\langle 0,0,1\rangle\to\langle 1,0,2\rangle\to\langle 0,0,0\rangle$$

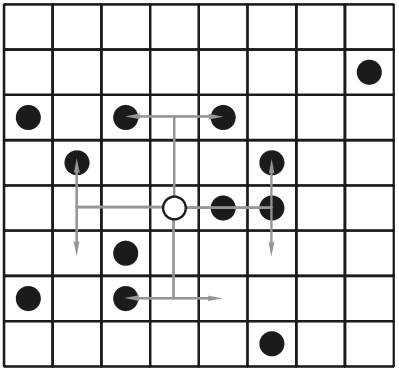
Для $N=4$ решения этой задачи нет: Spin не нашел опровержения.

Обход конем шахматной доски. Известная логическая задача состоит в том, чтобы найти план обхода конем шахматной доски размером $N\times N$ из некоторой начальной позиции (I, J), побывав на каждой клетке только один раз. Снова система верификации Spin, проведя исчерпывающий анализ всего конечного множества возможных состояний построенной модели, позволяет решить задачу.

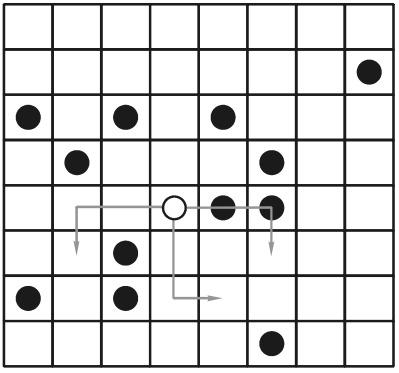
На рис. 8.8 представлена идея решения проблемы. Состояние системы запоминается в булевом массиве A размером $N\times N$, в котором, фактически, хранятся те клетки, на которых конь уже побывал при обходе.



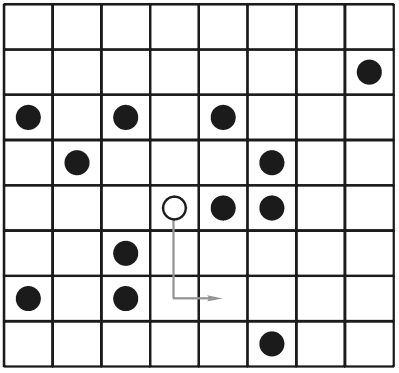
Очередное состояние



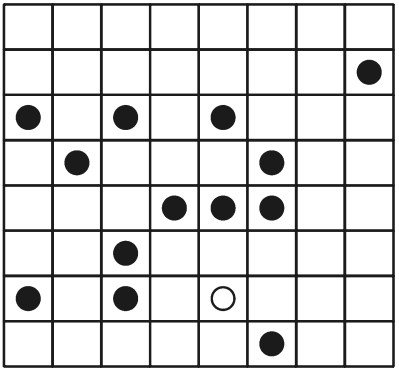
Возможные ходы коня



Где еще не были



Выбираем один ход



Следующее состояние

Рис. 8.8. Выбор хода в задаче обхода конем шахматной доски

Для простоты в представленном ниже фрагменте программы на псевдокоде будем использовать двумерный булев массив $A[N,N]$. Хотя в языке Promela напрямую возможно использование только одномерных массивов, в нем есть простой способ описать и использовать двумерные массивы. Это делается с использованием оператора определения сложных типов `typedef`. Этот оператор, например, можно использовать для задания структуры сообщения, посылаемого в канал. Мы увидим использование этого оператора в следующем разделе. Кроме того, эту конструкцию можно использовать и для задания и манипулирования многомерными массивами. Например, в следующем фрагменте показывается, как описать и использовать двумерный массив в языке Promela:

```
typedef VECTOR { int a[8] }

 VECTOR A[8];
...
 A[i].a[j] = A[j].a[i];
```

Вначале все элементы массива A имеют значение *true* за исключением начальной позиции коня, когда конь попадет на клетку (i,j), значение элемента $A[i,j]$ становится равным *false*, и далее на эту клетку ход будет запрещен. Возможная смена состояний системы определяется следующим циклом `do` программы на языке Promela. Каждая из 8 альтернатив, соответствующих возможным ходам коня, выполняется, если конь может сделать этот ход и если конь не был на этой клетке:

```
/* псевдокод */
# define K = 1; A[ , ]=true; A[I,J] = false;
do      /* проверка всех 8 вариантов хода коня и выполнение возможных
ходов */
:: i>2&&j<N && A[i-2, j+1] → i=i-2; j=j+1; A[i, j]=false; K++
:: i>2&&j>1 && A[i-2, j-1] → i=i-2; j=j-1; A[i, j]=false; K++
...
:: K==N*N →break /* прошли все клетки по одному разу */
od
```

Проверяемое свойство: $\mathbf{AG}(\neg(K == N*N))$ — на всех путях счетчик ходов K никогда не достигнет значения $N*N$.

Игра 15. Всем известная "игра 15" является классическим примером, на котором часто оценивается эффективность эвристических алгоритмов планирования, подобных известному алгоритму $A*$. Правила игры просты. Из

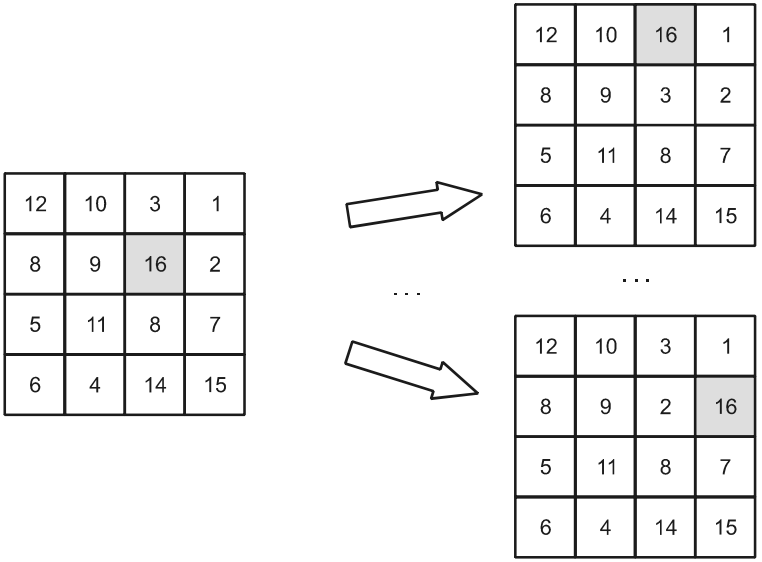


Рис. 8.9. Возможные альтернативные действия на очередном ходе в игре "15"

начального состояния, представляющего собой произвольное расположение 15 костяшек на поле 4×4, перемещением в пустое поле соседних с ним костяшек разместить их в порядке номеров (рис. 8.9). Известно, что ровно половину из 16! возможных начальных положений в игре "15" нельзя свести к искомому упорядоченному положению. Эта головоломка имеет очевидные обобщения, в которых нужно в квадрате $N \times N$ либо в прямоугольнике $N \times M$ от произвольного расположения костяшек прийти к их расположению в порядке номеров. Основной фрагмент программы на Promela этой игры для квадрата со стороной $N = 3$ представлен ниже.

```
byte N = 3;
do
 /* I,J — координаты пустого места с номером N*N */
::I>1 ⇒ Swap( a[I,J], a[I-1,J] ); I=I-1 /* пустая вверх */
::I<N ⇒ Swap( a[I,J], a[I+1,J] ); I=I+1 /* пустая вниз  */
::J>1 ⇒ Swap( a[I,J], a[I,J-1] ); J=J-1 /* пустая влево */
::J<N ⇒ Swap( a[I,J], a[I,J+1] ); J=J+1 /* пустая вправо*/
od
```

Проверяемое свойство:

$$\mathbf{AG}\neg\left(\wedge_{i,j\notin\{1,\ldots,N\}}\left(a[i,j]==(i-1)*N+j\right)\right)$$

на всех путях никогда не достигнем состояния, при котором все костяшки выстроятся по порядку. Если исходное положение костяшек возможно свести к упорядоченному, система верификации найдет план действий, приводящий к нему.

8.4. Верификация криптографических протоколов

Одной из областей приложения алгоритмов model checking является анализ корректности криптографических протоколов. Криптографический протокол — это правила, выполнение которых пользователями компьютерных сетей позволяет обеспечить безопасность обмена информацией через открытую сеть. Правила безопасного взаимодействия по сети включают базовые криптографические операции: шифрование и дешифрование, генерация псевдослучайных чисел и т. п. Проверка корректности таких протоколов должна рассматриваться с двух точек зрения. Во-первых, такие протоколы реализуются параллельными взаимодействующими процессами, поэтому их необходимо проверять на отсутствие типичных ошибок параллелизма — блокировок, ограничений живости, доступа к разделяемым ресурсам и т. п. Однако в значительно большей степени криптографические протоколы нуждаются в проверке их основных функций — обеспечения конфиденциальности (невозможности получения информации посторонним) и аутентичности (подлинности авторства) при передаче информации по сети. Несмотря на относительную простоту криптографических протоколов, многие из предложенных в этой области решений оказались некорректными, что объясняется трудностью анализа всех вариантов развития событий в системе, в которой параллельно действуют как участники обмена информацией, так и злоумышленники.

Атака на криптографический протокол — это возможные действия нечестных пользователей сети (злоумышленников, intruders), направленные на нарушение требований безопасности протокола. Верификация такого протокола состоит в выявлении всех возможных атак определенного типа, для чего необходимо построить формальную модель действий честных субъектов обмена и всех возможных действий потенциальных злоумышленников. Если проблему удастся свести к анализу системы с конечным числом состояний, а требования безопасности протокола выразить с помощью логической формулы, то для верификации можно использовать разработанные процедуры model checking, результатом работы которых будет либо гарантия того, что атака невозможна, либо контрпример — найденная атака на протокол, т. е. последовательность действий злоумышленника, на которой требования корректности протокола не выполняются.

Протокол Нидхэма — Шрёдера. Мы выполним здесь анализ на отсутствие атак известного криптографического протокола Нидхэма — Шредера. Задачей этого протокола является обеспечение взаимной аутентификации объектов в компьютерной сети. Роджер Нидхэм и Михаэль Шредер в 1978 году ([120]) впервые предложили механизм решения проблемы аутентификации в сети, который базируется на использовании так называемого "шифрования с открытым ключом". Только через 17 лет после опубликования этого протокола с помощью верификации была обнаружена атака на этот протокол. Протокол был модифицирован, и в настоящее время на основе модифицированного протокола Нидхэма — Шредера построена широко известная система аутентификации *Kerberos* (которая названа именем Цербера, пса из греческой мифологии, охранявшего ворота в ад).

Аутентификация (англ. *authentication*), это *подтверждение подлинности* — проверка соответствия *субъекта* тому, за кого он себя выдает. Протокол аутентификации решает следующую задачу. Имеется открытая компьютерная сеть, в узлах которой находятся субъекты обмена (собеседники). Необходимо обеспечить взаимную аутентификацию собеседников.

Как убедить удаленного собеседника, что ты — это ты? Как самому убедиться в том, что твой удаленный собеседник — именно тот, за кого он себя выдает?

Очевидно, что собеседника можно считать аутентифицированным, если он продемонстрировал *знание некоторого секрета, известного только ему*. Принципиальным моментом протокола Нидхэма — Шредера является удостоверение в том, что собеседник действительно знает свой личный секрет. Ясно, что посылка агентом своего секрета через сеть бессмысленна: сеть не защищена, и любой злоумышленник может перехватить секрет и затем пользоваться им под видом этого агента.

Протокол Нидхэма — Шредера решает проблему аутентификации на основе использования так называемых *публичных* и *секретных ключей*. Каждый агент R в сети обладает публичным (открытым) ключом $PK(R)$ и секретным ключом $SK(R)$. Публичный ключ $PK(R)$ агента R — это метод шифрования информации. Он известен всем другим собеседникам в сети, и каждый из них может с помощью этого ключа зашифровать посылаемое по сети сообщение для R. Секретный ключ $SK(R)$ знает только сам владелец R этого ключа: только он может дешифровать любое сообщение, зашифрованное его публичным ключом.

Протокол аутентификации Нидхэма — Шредера использует для идентификации удаленного собеседника R знание им его секретного ключа $SK(R)$, или, что то же, возможность дешифрования им сообщения, зашифрованного публичным ключом $PK(R)$. Протокол Нидхэма — Шредера позволяет двум ни-

когда не встречавшимся удаленным собеседникам не только идентифицировать друг друга в сети, но и обменяться им секретной информацией, которая может быть использована этими собеседниками для генерации их общего секретного ключа для последующего сеанса связи, закрытого от других.

Мы рассмотрим здесь несколько упрощенный вариант протокола Нидхэма — Шредера, отличающийся от оригинального тем, что все собеседники помнят *открытые* ключи друг друга, а не обращаются за ними к открытому серверу аутентификации, как это сделано в протоколе Kerberos. Это не изменяет суть протокола, но чуть упрощает модель.

Предположим, что существует несколько субъектов (например, A — *Alice*, B — *Bob* и др.), у каждого из которых есть свой известный всем другим его публичный ключ и известный только ему свой секретный ключ. Субъекты могут передавать сообщения, зашифрованные публичным ключом собеседника, но только сам собеседник может своим секретным ключом это сообщение дешифровать. Инициатором обмена будем считать субъект *Alice*. Протокол состоит из посылок по несекретному каналу следующих трех сообщений (рис. 8.10).

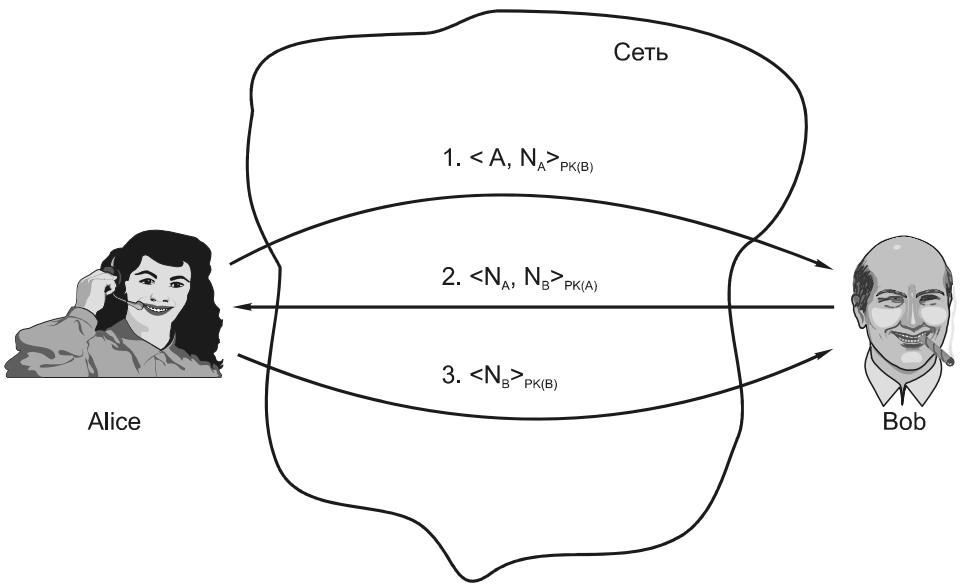


Рис. 8.10. Схема протокола аутентификации Нидхэма — Шредера

1. *Alice* генерирует случайное число (*nonce*) N_A (N_A — первая половина ее будущего общего "секрета" с *Bob*, который для надежности будет потом использован только один раз), шифрует сообщение $\langle A, N_A \rangle$, публичным

ключом $PK(B)$ агента B, и посылает по сети шифрованное сообщение $\langle A, N_A \rangle_{PK(B)}$. Здесь A — идентификация отправителя. Получить это сообщение может любой, но дешифровано оно может быть только агентом B.

2. Дешифровав это сообщение, *Bob* генерирует свое случайное непредсказуемое число (*nonce*) N_B (вторую половину их будущего общего с *Alice* "секрета"), шифрует сообщение $\langle N_A, N_B \rangle$, публичным ключом $PK(A)$, принадлежащим *Alice*, и посылает в сеть шифрованное сообщение $\langle N_A, N_B \rangle_{PK(A)}$. В этом сообщении кроме *nonce* N_A (тот секрет агента A, который B прочитал из полученного им сообщения) присутствует и *nonce* N_B — вторая часть их общего секрета. Поскольку сообщение зашифровано публичным ключом $PK(A)$, то только *Alice* может дешифровать это сообщение.

3. *Alice* получает сообщение $\langle N_A, N_B \rangle_{PK(A)}$ и дешифрует его. Наличие N_A в этом сообщении убеждает *Alice*, что код получен именно от *Bob* — только он мог дешифровать первое сообщение. Она принимает пару (N_A, N_B) как их общий секрет и уверена, что установила связь именно с B. *Alice* отсылает *Bob* сообщение $\langle N_B \rangle_{PK(B)}$ с его секретом N_B, зашифровав его публичным ключом $PK(B)$, чтобы подтвердить прием второго сообщения.

4. *Bob* принимает сообщение $\langle N_B \rangle_{PK(B)}$ и дешифрует его. Наличие в нем N_B убеждает *Bob*, что сообщение прислала *Alice* — только она могла дешифровать второе сообщение с его, *Bob*, секретом N_B. Поэтому *Bob* также принимает пару (N_A, N_B), как их общий секрет, и уверен, что он установил связь именно с A.

Таким образом, хотя все сообщения посылаются по несекретному каналу, каждый собеседник, и A, и B, продемонстрировал своему партнеру знание своих секретных ключей, и поэтому аутентифицированы друг другом: A убежден, что его собеседник B, а B убежден, что его собеседник A. Кроме того, пара случайных чисел (N_A, N_B) известна только этим двум собеседникам (поскольку эти значения посылались только зашифрованными), и эта пара теперь может быть использована для генерации по известному им обоим алгоритму нового секретного ключа для закрытого сеанса их приватной беседы.

Предположим, что компьютеры агентов защищены от вторжения, а примитивы шифрования являются стойкими (т. е. шифрование является абсолютным, и никто, кроме владельца секретного ключа, не может дешифровать посланное ему сообщение, зашифрованное его публичным ключом). В этих предположениях протокол выглядит совершенно безупречно: ведь злоумышленник не может имитировать такой обмен, т. к. не может дешифровать ни сообщения для A, ни сообщения для B.

Атака на протокол Нидхэма — Шредера. Будем считать, что в обмене сообщениями кроме *Alice* и *Bob* присутствует еще один собеседник, злоумышленник (Intruder), который не знает секретных ключей собеседников и не может получить доступ к данным в их компьютерах, но пытается провести атаку на протокол с помощью всех других доступных средств, имея такой же, как и все участники, доступ к каналам связи и зная сам протокол (рис. 8.11). Злоумышленник выступает перед другими агентами честным собеседником, поэтому с ним, как и с другими честными участниками обмена, и A и B могут устанавливать контакты.

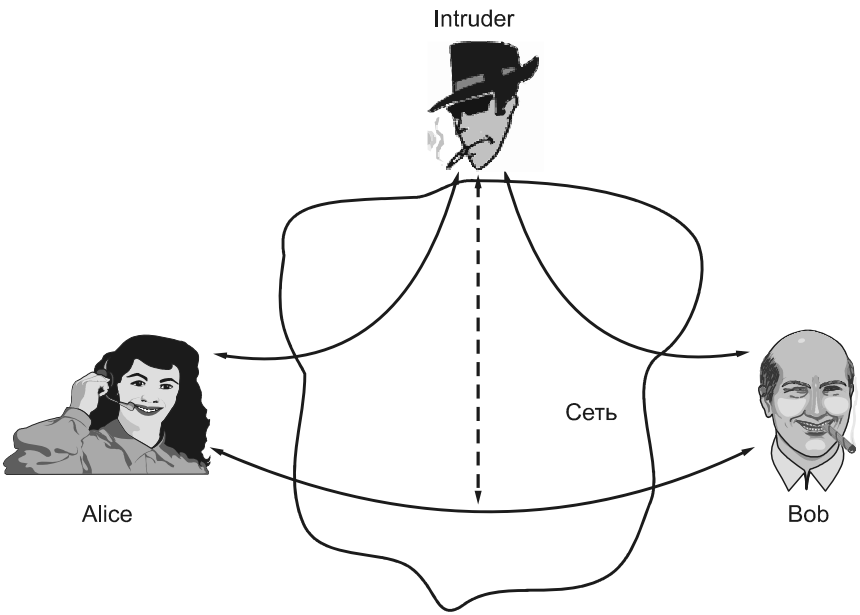


Рис. 8.11. Агенты протокола аутентификации

Наша задача — найти атаку на протокол не путем общих логических рассуждений, а использовать стандартный подход к верификации параллельных систем с помощью системы верификации Spin, которая проанализирует вы-

числения при всех действиях злоумышленника, ему доступных, и сама выберет ту последовательность действий злоумышленника, которые приведут к нарушению требований секретности протокола. Иными словами, система верификации сгенерирует атаки на протокол, если они существуют.

Будем считать, что злоумышленник может выполнять следующие очевидные действия, возможные при работе с незащищенными каналами связи:

- перехватывать сообщения, посланные по каналу;
- посылать сообщения, которые он мог перехватить ранее (даже если он не имеет ключа для их дешифрования);
- создавать собственные сообщения, используя известную ему информацию, и посылать их, маскируясь под любого из агентов в сети.

Может ли злоумышленник, выполняя такие действия, провести атаку на этот протокол? Как уже говорилось, атака — это действия, нарушающие свойства корректности криптопротокола. Неформально требования корректности данного протокола можно выразить так:

- *целостность*: когда каждый из агентов, A и B, успешно завершил обмен сообщениями, то A верит, что его собеседник B *тогда и только тогда*, когда B верит, что его собеседник A;
- *конфиденциальность*: после того, как честные собеседники завершили протокол, и B установил, что его собеседник — A, злоумышленник не должен знать ни N_A — секрета агента A, ни N_B — секрета агента B.

Проведем верификацию протокола с помощью системы Spin. Для этого в языке Promela:

- построим модель, описывающую действия честных агентов, следующих правилам протокола;
- построим недетерминированный процесс злоумышленника, в котором представлены все перечисленные выше возможные действия;
- формально с помощью формул LTL выразим требования корректности протокола;
- проверим выполнение требований корректности протокола у построенной модели, пользуясь системой Spin.

Атакой на протокол в этой постановке будет являться любая последовательность действий злоумышленника, нарушающая требования корректности протокола, выраженные формулой LTL. Поскольку Spin исчерпывающе проверяет все пути, которые возможны в модели, то Spin сгенерирует атаку

(найдет контрпример, для которого не выполняются требования корректности протокола), если такая атака существует.

Полный текст программы на Promela для протокола Нидхэма — Шредера приведен в руководстве по системе Spin, приложенном к этой книге. Здесь мы дадим только общую идею этой программы.

Переменные значения, которые могут использоваться в протоколе, представим символьными константами:

```
mtype = { msg,                          /* тип сообщения */
          alice, bob, intruder,         /* идентификаторы собеседников */
          pkA, pkB, pkI,                /* публичные ключи собеседников */
          nonceA, nonceB, nonceI,       /* случайные числа собеседников */
          ok, err};
```

Использованием конструкции `typedef` определим структуру зашифрованных сообщений в сети, состоящих из идентификации посылающего и принимающего агента, ключа, которым зашифровано сообщение, и двух данных. Некоторых элементов в реальном сообщении может и не быть. Например, сообщение типа 1 несет только информацию: отправитель, получатель, ключ получателя и секрет отправителя. Сообщение типа 3 не несет отправителя и его секрета.

```
typedef mCrypt {mtype sender, receiver, key, info1, info2}
```

Все 5 элементов сообщения будут представлены символьными константами.

Сеть моделируется как два синхронных канала, разделяемых всеми агентами:

```
chan network[2]=[0] of { mtype, /*мнемоническое имя—номер сообщения */
                         mCrypt /*само сообщение — структура данных */
                       }
```

Модель честных агентов *Alice* и *Bob* строится полностью в соответствии с протоколом:

```
active proctype Alice( )  {
    mtype pkey, pnonce;              /* публичный ключ и nonce партнера */
    mCrypt data;
/* выбор собеседника: A может захотеть установить контакт с B или I */
if
  :: partnerA=agentB; pkey=keyB      /* msg от A будет направлено B */
  :: partnerA=agentI; pkey=keyI      /* msg от A будет направлено I */
fi
/* посылка первого сообщения со своим nonce выбранному собеседнику */
network [0]! (msg1, mCrypt {Alice, partnerA, pkey, nonceA, 0 } )
```

```
  /* прием nonce собеседника из второго сообщения, если оно зашифровано
   публичным ключом keyA и содержит ранее посланный nonceA */
network [1]? (msg2, data);
(data.key==keyA)&&(data.info1 == nonceA) ⇒ pnonce = data.info2;
/* посылка принятого nonce выбранному ранее собеседнику */
network [0]! (msg3, mCrypt {Alice, partnerA, pkey, 0, pnonce } );
statusA = ok;  /* агент A завершился успешно */
}
```

Модель поведения агента *Bob* также соответствует протоколу и строится аналогично.

В модели злоумышленника опишем все действия, которые возможны:

- каждое сообщение, которое, в принципе, можно послать, посылается;
- полученное сообщение, если оно не адресовано злоумышленнику, запоминается и потом может быть послано без изменения;
- если сообщение направлено злоумышленнику, он из этого сообщения извлекает и запоминает всю информацию;
- если данные, необходимые для конструирования любого сообщения, когда-то были получены, они могут быть использованы в операторе посылки (например, в его защите).

В процесс злоумышленника введем логические переменные, необходимые для формирования требований к протоколу, например, параметр `I_knows_nonceB` станет истинным, если злоумышленник действительно получит секрет B.

```
active proctype Intruder( ) {
bool I_knows_nonceA = false, I_knows_nonceB = false;
/* вначале злоумышленник не знает секретов A и B */

do          /* цикл подслушивания и посылок сообщений */
:: network [0] ? (msg, data)->      /* если подслушал любое сообщение */
  if
   :: intercepted = data;    /* либо запоминает перехваченные данные */
   :: skip;                  /* либо нет */
  fi;
  if
   :: data.key == keyI ->    /* если в принятом сообщении — свой ключ */
   if                        /* то извлекает чужой nonce, если он есть */
    :: data.info1==nonceA||data.info2==nonceA->I_knows_nonceA= true;
    :: data.info1==nonceB||data.info2==nonceB->I_knows_nonceB= true;
    :: else -> skip;
```

```
    fi
     :: else -> skip;       /* если в принятом сообщении — не свой ключ */
  fi;
            /* посылает свое сообщение собирая его из возможных кусков */
::if        /* произвольно выбирает тип сообщения */
  :: msg = msg1;
  :: ...       /* недетерминированно делает то же для msg2 и msg3 */
  fi;
  if   /* случайно выбирает получателя, выдавая себя за A, B или I*/
   :: data.sender=agentA -> data.receiver = agentB; data.key = pkeyB;
    .../* или выдает себя за B, или за I, и выбирает получателей
  fi;
  if  /* собирает свое сообщение для посылки */
   :: data=intercepted;  /* перехваченные данные шлет без изменения */
   :: if                 /* либо собирает содержимое сообщения для A */
        :: data.info1 = agentA;
        :: I_knows_nonceA -> data.info1 = nonceA;
          ...  /* то же для B */
      fi
      ...      /* то же для data.info2  */
   :: if                    /* собирает содержимое сообщения для B */
        :: ...
        :: ...
      fi
   fi;
   network [1]! (msg, recpt, data);
od;
```

Подавляющее число возможных сообщений злоумышленника некорректны. Они распознаются агентами A и B, которые не отвечают на них, поэтому построенная программа имеет много блокировок. Но нам важно проверить, существует ли хотя бы один путь, на котором возможна атака. Это и делает Spin, проверяя выполнение требований корректности протокола и генерируя контрпример, если он есть.

Проверка корректности протокола осуществляется стандартным образом по формальной спецификации свойства "атаки нет". Это свойство выражается тремя условиями:

- C1 (целостность): "после сессии агент A верит, что его партнер B, если и только если агент B верит, что его партнер A". Его формальная спецификация формулой LTL:

$$\mathbf{AG}\big(statusA == ok \;\&\; statusB == ok \Rightarrow \big(partnerA == agentB \Leftrightarrow partnerB == agentA\big)\big)$$

- C2 (конфиденциальность): "после того, как агент A завершился и установил, что его партнер — B, агент I (intruder) не знает секрета A". Его формальная спецификация:

$$\mathbf{AG}\left(statusA == ok \;\&\; partnerA = agentB \Rightarrow \neg I_knows_nonceA\right)$$

- C3 (конфиденциальность): "после того, как агент B завершился и установил, что его партнер — A, агент I не знает секрета B". Его формальная спецификация:

$$\mathbf{AG}\left(statusB = ok \;\&\; partnerB = agentA \Rightarrow \neg I_knows_nonceB\right)$$

По этим формальным спецификациям система Spin находит атаку на протокол (рис. 8.12), т. е. последовательности состояний собеседников, нарушающие формальные требования к протоколу.

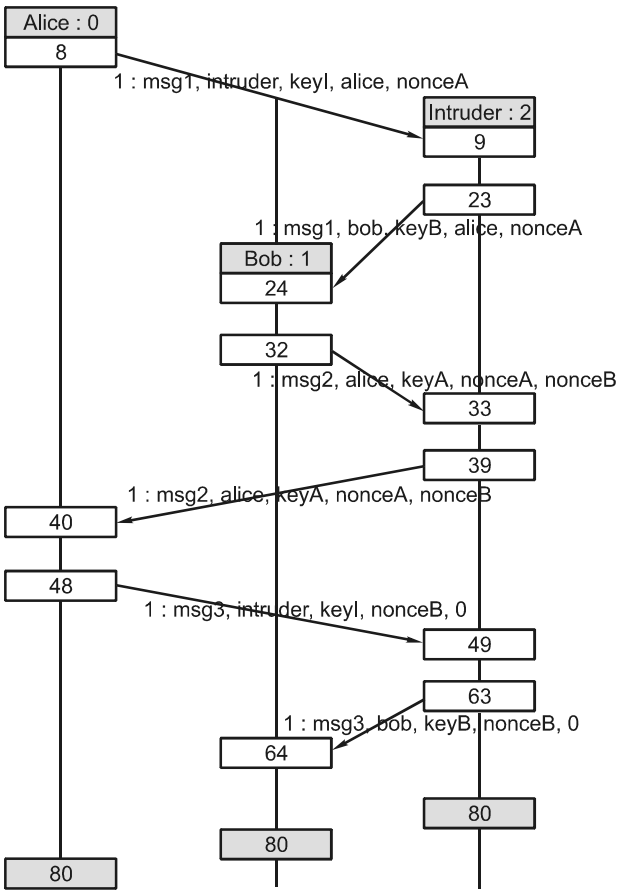


Рис. 8.12. Атака на протокол Нидхэма — Шредера

Найденная атака следующая:

1. A начинает взаимодействовать с I как с честным партнером.
2. Получив сообщение $msg1$ от A, I посылает свое сообщение $msg1$ к B, маскируясь под A: он использует $nonceA$, полученный от A. Агент B считает, что это сообщение получено от A, потому что в нем в качестве отправителя стоит A (оно подписано I, выдающим себя за A).
3. B посылает сообщение $msg2$, которое зашифровано ключом A. В это сообщение включены как $nonceA$, так и $nonceB$. I перехватывает это сообщение. Поскольку оно закодировано открытым ключом A, I не может декодировать это сообщение сам, и поэтому просто пересылает его агенту A.
4. Получив сообщение $msg2$, A видит в нем свой $nonceA$, считает, что все закончилось успешно, и посылает $msg3$ своему партнеру — агенту I, считая, что он установил связь с I.
5. Это сообщение закодировано ключом для I и содержит $nonceB$. Поэтому агент I может его декодировать, получив, в дополнение к $nonceA$, также и $nonceB$.
6. I посылает в своем сообщении $msg3$ этот $nonceB$ агенту B, который тоже удовлетворен и успешно завершается, считая, что им установлена связь с A.

В результате этой атаки Bob думает, что он говорит с $Alice$, а он говорит со злоумышленником (рис. 8.13), а сам злоумышленник имеет секреты обоих честных собеседников, A и B. Теперь злоумышленник может разговаривать с пользователем B под видом пользователя A.

Итак, при анализе протокола Нидхэма — Шредера найдена атака, т. е. такая последовательность действий одного из субъектов обмена, которая приводит к нарушению свойств корректности протокола. Анализ найденных атак позволяет изменить криптографический протокол так, чтобы свойства корректности удовлетворялись. Например, одной из причин возможной атаки здесь является то, что агент I может от своего имени послать полученное от B сообщение $msg2$, адресованное агенту A. Если в этом сообщении потребовать идентификации отправителя (т. е. если вид этого сообщения станет $\langle B, N_A, N_B \rangle_{PK(B)}$), то верификатор не находит возможной атаки: агент A обнаружит по содержимому этого сообщения, что хотя он говорит с I, в полученном от I зашифрованном сообщении стоит не его имя.

Очевидно, что в общем случае алгоритм верификации может и не выявить некорректность криптографического протокола, если модель этого протокола

слишком грубая. Модель криптографических средств защиты информации должна детально и полно описывать, какой информацией владеет на каждом возможном шаге функционирования системы каждый субъект распределенной системы, а это не всегда легко сделать. Кроме того, часто является трудным делом формально выразить такие свойства криптографических протоколов, как секретность, конфиденциальность, анонимность и т. п.

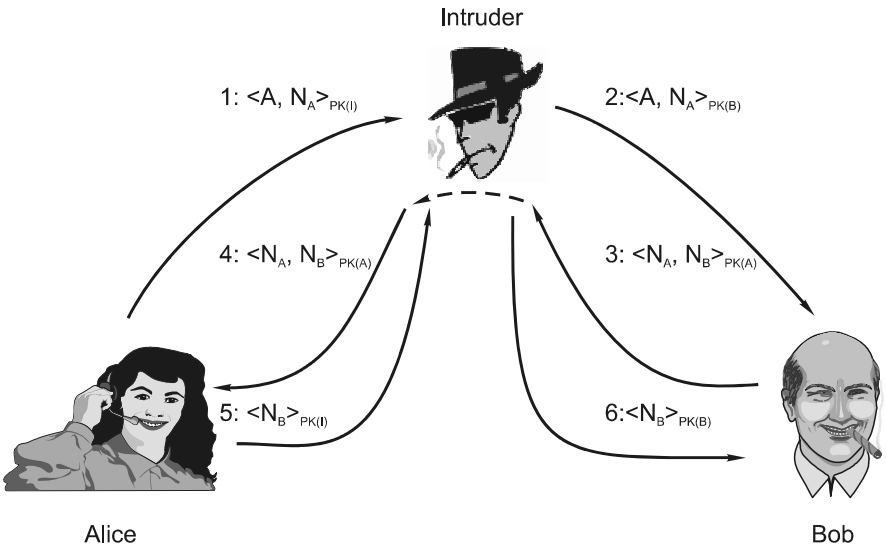


Рис. 8.13. Bob думает, что он говорит с Alice, а в действительности он говорит с Intruder

8.5. Заключение

Метод model checking используется не только для верификации систем обработки данных — программ и аппаратуры. В этой главе приведены примеры лишь некоторых из задач, которые могут быть решены с помощью этой техники. В последние годы этот метод стал широко применяться в разработке алгоритмов планирования при анализе веб-документов, анализе бизнес-процессов и протоколов электронной коммерции, в искусственном интеллекте. В 2008 г. состоялся уже пятый Международный семинар по model checking и Искусственному интеллекту (MoChArt V). Любая поисковая система выдаст сотни ссылок на сочетания слов model checking со словами: access control policy, World Wide Web, web services, web applications, systems biology, nanotechnology. Публикуются работы по применению этого метода для преобразования графов, анализа учебных планов, анализа XML-документов,

проверки непротиворечивости семантических ограничений в текстах, анализе систем представления знаний, анализе коммуникации интеллектуальных агентов и т. д. и т. п.

Одним из важных приложений метода проверки модели является верификация криптографических протоколов. Для проверки правильности таких протоколов с помощью model checking обычно строится модель протокола и всех возможностей, доступных злоумышленнику, а в качестве спецификации правильной работы формулируется утверждение, что требования корректности протокола не нарушаются. Система верификации тогда сгенерирует атаку, если она возможна в этом протоколе.

8.6. Замечания

Существует множество работ, в которых процедуру model checking предлагается использовать для верификации бизнес-процессов, например, [151], [40]. Очень похожие идеи лежат в основе использования алгоритма model checking для анализа диаграмм активности UML [49]. В отличие от языка Amber, язык диаграмм активностей UML стандартизирован [122]. Близким к этому направлению является быстро развивающаяся область верификации протоколов электронной коммерции с помощью model checking (например, [7]).

Использование model checking для решения проблем планирования обсуждается во многих публикациях. В искусственном интеллекте сформировалась парадигма "*planning as model checking*" [59]. Введение в это направление изложено в [23], эта работа использована в *разделе 7.2*. В [18] предложено использовать темпоральную логику для управления процессом поиска при решении задач планирования: в этой работе демонстрируется, как с помощью темпоральной логики можно формализовать специфические знания структуры среды поиска. Описание подходов к проблемам планирования в мире блоков можно найти в [137]. Статью в Википедии о Китайских кольцах можно найти по ссылке [191]. Старинная задача фермер-волк-коза-капуста описывается во многих сборниках логических задач, ее обсуждение можно найти в [179]. Логическая головоломка о миссионерах и каннибалах описана в статье Википедии [180].

Верификация криптографических протоколов с помощью model checking и других подходов к верификации параллельных систем превратилась в отдельную ветвь криптоанализа. Изложенный здесь материал опирается на результаты из [100], [115], [84] и [106].

8.7. Задачи к главе 8

8.1. Выразите в CTL и проверьте для задачи волк-коза-капуста следующее:

существует решение задачи, которое удовлетворяет ограничениям (т. е. что все переправятся, но при этом коза не останется без присмотра с волком, а капуста — с козой), а также и дополнительным условиям:

а) если коза переедет на ту сторону, она там навсегда и останется;

б) волк может остаться наедине с капустой (без фермера и без козы) на этом берегу реки;

в) при любом варианте решения задачи фермер должен будет хоть раз переезжать реку обратно не один.

8.2. Какие свойства в задаче волк-коза-капуста выражают формулы:

$$\mathbf{AF}\left[\neg c \Rightarrow \mathbf{EXG}c\right]$$

$$\mathbf{EF}\left[\neg c \Rightarrow \mathbf{XG}c\right]$$

$$\mathbf{G}\left[w\mathbf{U}\left(\mathbf{AF}\neg w\right)\right]$$

$$\mathbf{AG}\left[f \Rightarrow \mathbf{F}\neg f\right]$$

$$\mathbf{AF}\left[f \neg w\right]$$

$$\mathbf{EFG}\left(fc\right)$$

8.3. С помощью системы Spin решите головоломку о лягушках, которые хотят по камням пересечь ручей (см. [196]). Три зеленых лягушки сидят на трех камнях слева и хотят перебраться слева направо, три красных лягушки сидят на трех камнях справа и хотят перебраться справа налево, между ними один свободный камень. Найти последовательность действий лягушек, после которой все зеленые лягушки окажутся справа, а все красные — слева. Ограничения на движения лягушек:

а) они могут прыгать только в том направлении, куда они глядят: лягушки слева могут прыгать только к правому берегу, лягушки справа — только к левому берегу;

б) лягушки могут прыгать либо на свободный камень перед собой, либо на свободный камень, перепрыгнув через одну лягушку другого цвета перед собой.

8.4. Одна из головоломок о переправе звучит так. Муж с женой и двумя детьми хотят переправиться через реку. Единственный способ переплыть ее —

взять лодку у рыбака, находящегося на этом берегу. Но лодка может выдержать только одного взрослого или двух детей, которые умеют грести. Может ли семья переплыть реку на этой лодке и вернуть ее рыбаку? Решите головоломку с помощью системы Spin.

8.5. Еще одна головоломка о переправе. Два мальчика с лодкой согласились помочь трем солдатам переплыть реку. Их лодка так мала, что может выдержать либо одного солдата, либо двух мальчиков. Как распланировать переправу? Решите головоломку с помощью системы Spin.

8.6. Один из вариантов головоломки о переправе через реку относят к XVI веку. Задача состоит в переправе трех пар *жених-невеста* с использованием лодки, которая может выдержать только двоих. Чтобы не компрометировать невесту, она не должна оказаться с чужим женихом в лодке либо на берегу, если своего жениха нет рядом.

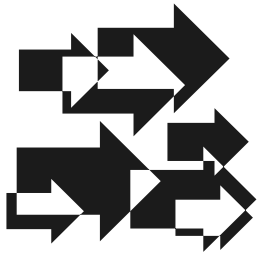
8.7. Проблема поиска в графе гамильтонова пути, т. е. пути, проходящего точно по одному разу через каждую вершину, может быть сформулирована в виде проблемы проверки модели для некоторой формулы LTL. Действительно, пусть $G = (V, R)$ — направленный граф с n вершинами: $V = \{v_1, v_2, ..., v_n\}$. Поместим в каждую вершину v_i атомарный предикат p_i, истинный только в этой вершине. Построим LTL-формулу, которая будет истинной в том и только в том случае, когда в G существует такой путь, что все атомарные предикаты на нем истинны точно по одному разу (т. е. p_i когда-нибудь в будущем будет истинным, и в любом состоянии этого пути если утверждение p_i стало истинным, то со следующего состояния оно все время будет ложным).

Поскольку нам неважно, с какой вершины начнется гамильтонов путь и в какой вершине он закончится, введем дополнительную вершину — источник, из которой направлено по ребру в каждую вершину G, и дополнительную вершину — сток, в который направлены дополнительные ребра из каждой вершины G.

Постройте такую формулу.

> ❖ ***Замечание***
>
> Проблема поиска Гамильтонова пути является NP-полной. Сведение этой проблемы к проверке истинности LTL-формулы говорит о том, что сложность алгоритма проверки модели для LTL-формулы является NP-полной проблемой.

ГЛАВА 9



Бинарные решающие диаграммы

Бинарные решающие диаграммы (Binary Decision Diagrams, BDD) — это экономная форма представления булевых функций в виде ориентированного ациклического графа. Вершины графа представляют аргументы функции, листья — ее двоичные значения. Булевы функции в форме BDD от десятков, сотен и даже тысяч двоичных переменных могут быть экономно представлены в памяти компьютера, и операции с такими функциями выполняются очень эффективно. Поэтому многие задачи, в которых используются булевы функции, могут решаться весьма эффективно.

При представлении булевых функций в форме BDD оказалось возможным решать многие проблемы, которые при традиционных представлениях структур данных являются неразрешимыми из-за большой размерности таких представлений и сложности операций над ними. Поскольку базовые структуры данных (в частности, конечные множества и отношения) могут быть закодированы булевыми функциями, а операции над ними можно определить с помощью булевых операций, то многие алгоритмы на конечных структурах данных становятся очень эффективными, если эти структуры данных представить в форме BDD. Одно из самых удивительных применений BDD, позволившее увеличить сложность анализируемых систем в миллионы миллионов раз — это верификация методом *model checking*. Этой теме посвящена *гл. 10*. Однако верификация — не единственная область применения бинарных решающих диаграмм. BDD могут с успехом применяться фактически в каждой области, в которой требуется обработка дискретных структур данных, в частности, в комбинаторике. Можно без преувеличения сказать, что BDD совершили революцию в представлении данных и алгоритмах работы с ними.

"*Бинарные решающие диаграммы (BDD) — удивительны, и чем больше я играю с ними, тем больше я их люблю. Около 15 месяцев я был как ребенок*

с новой игрушкой, получив возможность решать такие проблемы, о которых я никогда не думал, что они могут быть решены" — пишет Дональд Кнут о BDD в препринте своей новой книги [88].

В данной главе вводятся основные определения из этой области, а также кратко рассматриваются теоретические и практические аспекты, связанные с BDD: их свойства, операции над ними и примеры их применения. Эта глава может читаться независимо от остального материала книги.

9.1. Проблемы с представлением булевых функций

Формы представления булевых функций. Булевы функции играют фундаментальную роль в информатике. Они широко применяются уже более полувека, на булевых функциях основана теория синтеза аппаратных систем вычислительной техники, их свойства исследовались несколько десятилетий. Казалось бы, в этой области трудно придумать что-то радикально новое. Однако несколько лет назад была предложена новая эффективная форма представления таких функций — бинарные решающие диаграммы, которая позволяет по-новому, очень эффективно решать задачи во всех областях применения булевых функций.

Булевы функции могут быть представлены многими различными способами. Стандартное представление для любого конечного отображения — это таблица соответствия, которая каждому набору значений аргументов сопоставляет значение функции на этом наборе. Для булевых функций такая таблица называется *таблицей истинности*: каждому набору двоичных значений аргументов сопоставляется 0 или 1. Другое широко используемое представление булевых функций — бинарное решающее дерево (Binary Decision Tree).

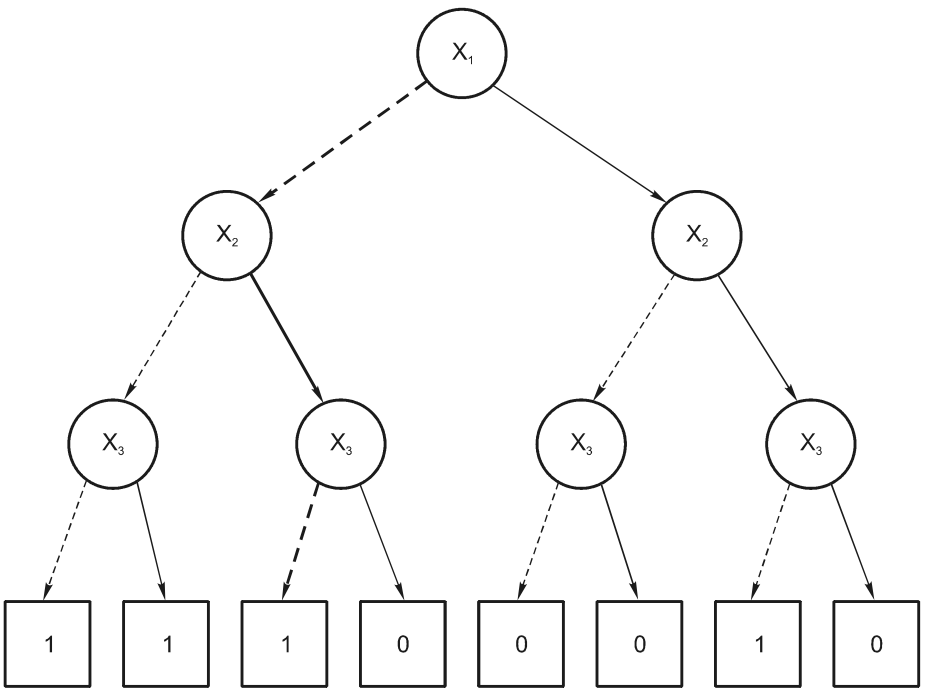
Пример 9.1

Булева функция от трех двоичных аргументов $f(x_1, x_2, x_3)$ представлена на рис. 9.1, *а* своей таблицей истинности, т. е. соответствием результата из множества $\{0,1\}$ каждому набору значений трех двоичных аргументов. На рис. 9.1, б представлено бинарное решающее дерево этой же функции.

> ❖ ***Определение 9.1 (бинарное решающее дерево)***
>
> Бинарное решающее дерево булевой функции — это представление функции в виде ориентированного дерева с одной корневой вершиной. Корень и другие внутренние вершины этого дерева помечены переменными функции, а листья (терминальные вершины) — значениями 0 или 1.

$x_1x_2x_3$	$f(x_1, x_2, x_3)$
000	1
001	1
010	1
011	0
100	0
101	0
110	1
111	0

а



б

Рис. 9.1. Формы представления булевой функции: *а*) таблица истинности и *б*) бинарное решающее дерево

Бинарное решающее дерево часто называют *семантическим деревом* [170]. На каждом ярусе такого дерева стоит одна и та же переменная. Из каждой внутренней вершины выходят два ребра — одно (штриховое) для значения переменной, равного 0 (его назовем 0-ребро), другое, сплошное, для значения этой переменной, равного 1 (его назовем 1-ребро). Направление ребер — сверху вниз, стрелки на ребрах обычно не указываются, поскольку порядок вершин определяется высотой их расположения. Любой путь от корня дерева к листу соответствует какому-либо набору значений двоичных переменных и завершается значением функции (0 или 1) на этом наборе. Поэтому вычисление значения функции может быть выполнено спуском по семантическому дереву от корня к терминальной вершине. На рис. 9.1, *б* выделен путь на бинарном решающем дереве, который соответствует следующему набору значений переменных: $x_1 = 0$, $x_2 = 1$, $x_3 = 0$. Этот путь ведет в терминальную вершину, помеченную 1. Действительно, значение функции f примера 9.1 на наборе $\langle 0, 1, 0 \rangle$ равно 1. Можно считать, что бинарное решающее дерево — это графическое представление алгоритма вычисления значения булевой функции по заданному набору значений аргументов.

Существует много других форм задания булевых функций. В частности, функцию можно задать аналитически, с помощью различных логических формул.

Пример 9.2

Функция f, представленная рис. 9.1, может быть задана аналитически различными формулами, например:

1. Произвольные логические формулы:

$$f(x_1, x_2, x_3) = \neg x_1 \vee x_2 \oplus x_3 x_2 (x_1 \vee x_3) =$$
$$= \left(\neg x_1 \oplus \left(x_2 (x_1 \oplus x_3)\right)\right) \vee x_1 x_2 \neg x_3 \vee (x_1 \Rightarrow x_2) \,\&\, \neg (x_2 x_3)$$

2. Совершенная дизъюнктивная нормальная форма (СДНФ):

$$f(x_1, x_2, x_3) = \neg x_1 \neg x_2 \neg x_3 \vee \neg x_1 \neg x_2 x_3 \vee \neg x_2 x_1 \neg x_3 \vee x_2 x_1 \neg x_3$$

3. Совершенная конъюнктивная нормальная форма (СКНФ):

$$f(x_1, x_2, x_3) = (x_1 \vee \neg x_2 \vee \neg x_3)(\neg x_1 \vee x_2 \vee x_3)(\neg x_1 \vee x_2 \vee \neg x_3)(\neg x_1 \vee \neg x_2 \vee \neg x_3)$$

4. Дизъюнктивная нормальная форма (ДНФ):

$$f(x_1, x_2, x_3) = \neg x_1 \neg x_2 \vee x_2 \neg x_3 \vee \neg x_1 \neg x_3 = \neg x_1 \neg x_2 \vee x_2 \neg x_3$$

5. Конъюнктивная нормальная форма (КНФ):

$$f(x_1, x_2, x_3) = (\neg x_1 \vee x_2)(\neg x_2 \vee \neg x_3) = (\neg x_1 \vee x_2)(x_1 \vee \neg x_2 \vee \neg x_3)(\neg x_2 \vee \neg x_3)$$

6. Полином Жегалкина:

$$f(x_1, x_2, x_3) = 1 \oplus x_1 \oplus x_1 x_2 \oplus x_2 x_3$$

> ❖ ***Определение 9.2 (невыполнимые, выполнимые, общезначимые булевы функции)***
>
> Булева функция называется невыполнимой, если на всех значениях аргументов она равна 0. Булева функция называется выполнимой, если существуют такие наборы значений аргументов, на которых она равна 1. Булева функция называется общезначимой, если на всех значениях аргументов она равна 1.

Для невыполнимой булевой функции все ее значения в таблице истинности (и пометки на терминальных вершинах в ее двоичном решающем дереве) равны 0. Для общезначимой функции все эти значения равны 1. У выполнимой функции существуют наборы ее значений, на которых значения функции равны 1.

> ❖ ***Определение 9.3 (каноническое представление)***
>
> Каноническим называется такое представление булевой функции, которое является для функции единственным.

Очень часто решение проблем информатики сводится к проверке совпадения двух объектов, и такая проверка основана на сравнении канонических представлений объектов. Две функции равны (эквивалентны), если их значения совпадают на всех наборах аргументов. Равенство двух функций можно проверить по их каноническому представлению: они равны, если их канонические представления совпадают (обычно с точностью до порядка членов или до изоморфизма). Каноническое представление булевой функции важно именно потому, что оно позволяет проверить равенство (эквивалентность) двух функций.

Легко видеть, что ДНФ, КНФ и произвольная формульная запись не являются каноническими представлениями булевой функции Любая булева функция может иметь множество различных представлений в классах ДНФ, КНФ и в классе произвольных логических формул. Для функции f примера 9.1 выше приведены совершенно различные ее представления в классах таких формул.

С другой стороны, таблица истинности и двоичное решающее дерево — это канонические представления булевой функции. Среди аналитических пред-

ставлений булевой функции каноническими являются СДНФ, СКНФ и полином Жегалкина. Недостатком всех этих канонических представлений булевой функции является их громоздкость. Сложность и СДНФ, и СКНФ, и полинома Жегалкина, а также таблицы истинности и бинарных решающих деревьев растет экспоненциально с ростом числа переменных. Например, бинарное решающее дерево булевой функции от n переменных всегда имеет $2^{n+1}-1$ вершин, таблица истинности имеет 2^n строк, сложность представления функции полиномом Жегалкина $O\left(2^n\right)$. Кроме того, недостатком аналитических канонических представлений булевой функции является то, что некоторые функции не могут быть представлены в соответствующих классах. Например, невыполнимая функция не имеет представления в виде полинома Жегалкина и в классе СДНФ. Общезначимая функция не имеет представления в СКНФ.

Задачи, решаемые с помощью булевых функций. Булевы функции используются в огромном числе приложений информатики, не только в логике и при синтезе логических схем. Для булевых функций типичными задачами являются следующие:

- ❒ найти значение функции на конкретной интерпретации (наборе значений аргументов);
- ❒ найти все интерпретации, на которых функция равна 1;
- ❒ проверить, является ли булева функция выполнимой, невыполнимой, общезначимой;
- ❒ проверить, совпадают ли две булевых функции;
- ❒ выполнить булевы операции над булевыми функциями;
- ❒ провести минимизацию функции в некотором базисе.

Если число переменных, от которых зависит булева функция, равно 3, 4, даже 5, то работать с такими функциями легко и удобно: разработаны многочисленные приемы решения упомянутых выше задач. Например, карты Карно очень удобны для минимизации булевой функции от трех и четырех переменных. Для числа переменных 5 или 6 представление функций с помощью карт Карно становится громоздким, а процесс минимизации трудоемким. При 10—20 переменных карты Карно, формулы, таблицы истинности применять бессмысленно.

При большом числе переменных все вышеперечисленные задачи решаются очень трудно. Причиной этого является экспоненциальный рост сложности представления булевых функций и соответствующий рост времени работы

любых алгоритмов с этими представлениями. Это является отражением главного положения теории сложности вычислений:

задача является практически неразрешимой, если время, требуемое для решения конкретных экземпляров этой задачи, растет экспоненциально с увеличением размеров этих экземпляров.

Таким образом, булевы функции, представленные стандартными способами, не позволяют эффективно решать проблемы вследствие экспоненциального роста сложности их представления.

9.2. Бинарные решающие диаграммы

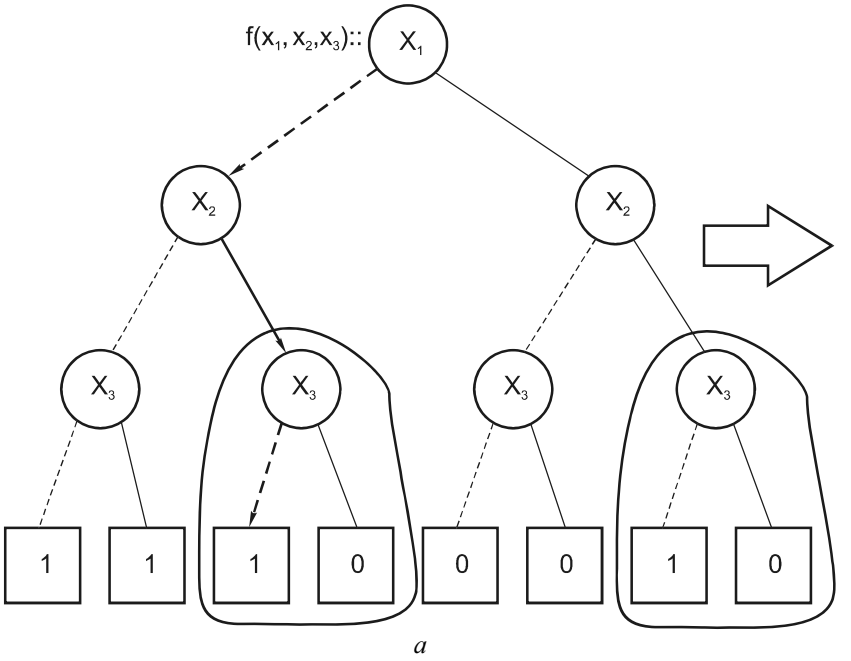
Бинарная решающая диаграмма (Binary Decision Diagram, BDD) — это новая форма представления булевых функций в виде ориентированного графа. Можно считать, что BDD получается из бинарного решающего дерева выбрасыванием всех избыточных подструктур (изоморфных подграфов). BDD представляет булеву функцию в виде минимального направленного ациклического графа (решающей диаграммы).

Пример 9.3

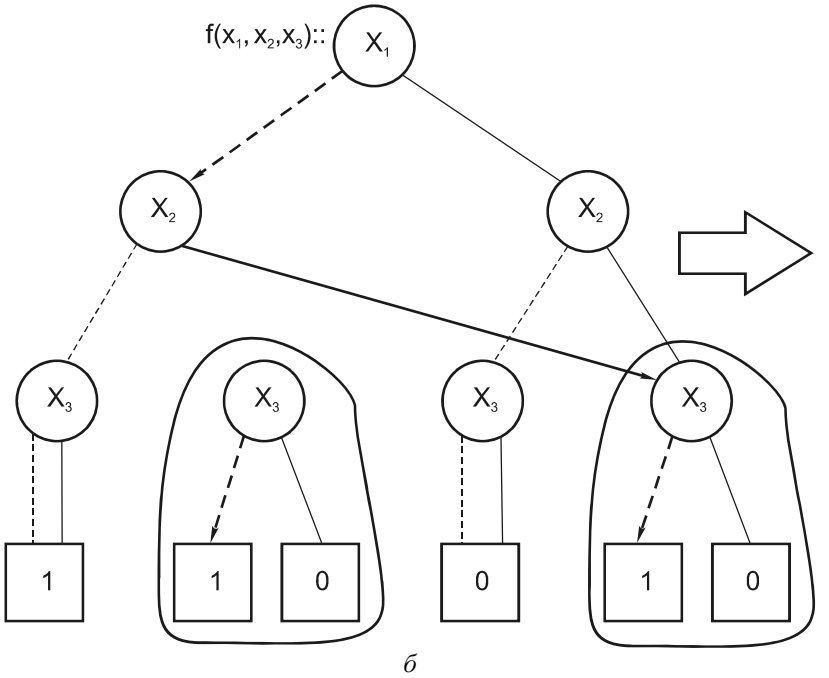
Рис. 9.2 показывает, как можно перейти к BDD от двоичного решающего дерева, которое также является ациклическим орграфом, но содержащим изоморфные (одинаковые) подграфы. Рассмотрим по шагам, как можно сделать такой переход, избавляясь от всех избыточностей в исходном дереве. Результирующая структура будет уже не деревом, а ориентированным графом.

Во-первых, в исходном двоичном решающем дереве нашего примера существует два одинаковых поддерева с вершиной, помеченной x_3. Ясно, что для вычисления значения функции по значениям аргументов можно использовать только один из одинаковых подграфов. В частности, можно объединить одинаковые терминальные вершины, оставив только две, 0 и 1. Кроме того, если из некоторой вершины оба выхода (со значениями 0 и 1 соответствующей переменной) ведут в одну и ту же подструктуру, то эта вершина может быть выброшена — значение функции на этом пути не зависит от этой переменной. Как результат этих двух преобразований, последний граф на рис. 9.2 представляет BDD исходной функции.

На рис. 9.2 можно видеть, что в результирующем графе выделенный путь, определяемый значениями переменных $\langle x_1 = 0, x_2 = 1, x_3 = 0 \rangle$, приводит в терминальную вершину со значением 1, как и в исходном двоичном решающем дереве.

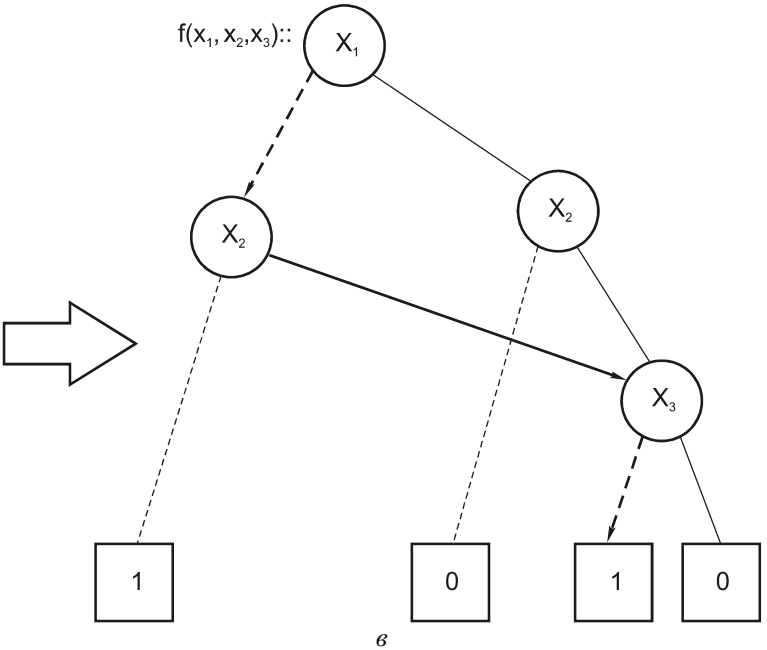


а

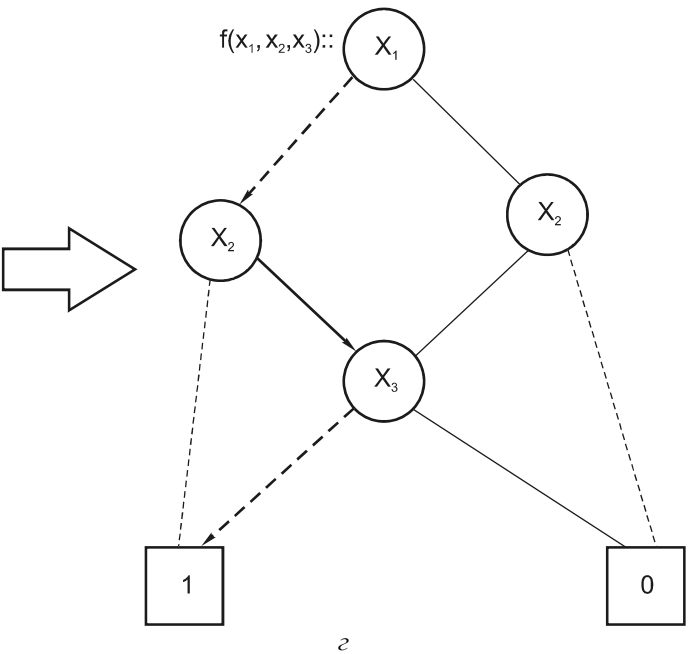


б

Рис. 9.2, *а* и *б*. Преобразование двоичного решающего дерева в BDD для $f\left(x_1, x_2, x_3\right)$



в



г

Рис. 9.2, *в* и *г*. Преобразование двоичного решающего дерева в BDD для $f\left(x_1, x_2, x_3\right)$

❖ ***Определение 9.4 (бинарная решающая диаграмма, Binary Decision Diagram, BDD)***

BDD — это двоичный направленный граф без избыточностей, т. е. редуцированный (reduced) ациклический орграф, в котором отсутствуют повторения в структуре, с одной корневой вершиной и в общем случае двумя листьями, помеченными 0 и 1. Корневая и промежуточные вершины помечены переменными, из каждой из них выходят ровно два ребра, соответствующие значениям 0 и 1 переменной, помечающей вершину.

Название Binary Decision Diagram отражает основную идею этой формы представления булевой функции:

- Binary — потому что используются двоичные переменные;
- Decision — потому что эта графическая структура визуально представляет те решения алгоритма движения по графу, которые приводят к результату (значению функции на данном наборе аргументов);
- Diagram — поскольку решение о значении функции принимается как результат движения по упорядоченному ациклическому графу (диаграмме).

Можно видеть, что BDD — это минимальное графическое представление алгоритма вычисления значения булевой функции по набору значений аргументов в заданном их порядке. BDD определяет возможные пути от корня в терминальную вершину, представляющую значение функции, в каждом промежуточном узле осуществляя решение (выбор пути) в соответствии со значением соответствующего аргумента функции. Хотя BDD не является деревом, у этого направленного графа всегда есть единственная начальная вершина (корень), это граф без циклов, у него есть в общем случае две терминальные вершины, помеченные 0 или 1.

Преобразование двоичного решающего дерева в BDD называется алгоритмом редукции (Reduce). Главная задача алгоритма редукции — устранить избыточность в графическом представлении алгоритма вычисления булевой функции. Как указывалось выше, этот алгоритм основан только на двух операциях (рис. 9.3):

- С1: *Правило слияния*. Объединить (слить) любые изоморфные подграфы (как частный случай, следует объединить терминальные вершины, помеченные одинаковыми значениями);
- С2: *Правило удаления.* Удалить из графа любую вершину, у которой оба непосредственных преемника изоморфны.

Легко показать, что повторное применение операций слияния и удаления в любом порядке к двоичному решающему дереву булевой функции или к лю-

бому полученному из него графу до тех пор, пока все избыточности не будут устранены (т. е. ни C1, ни C2 не могут быть далее применены), приводит к редуцированному представлению этой функции, которое и называется *бинарной решающей диаграммой*.

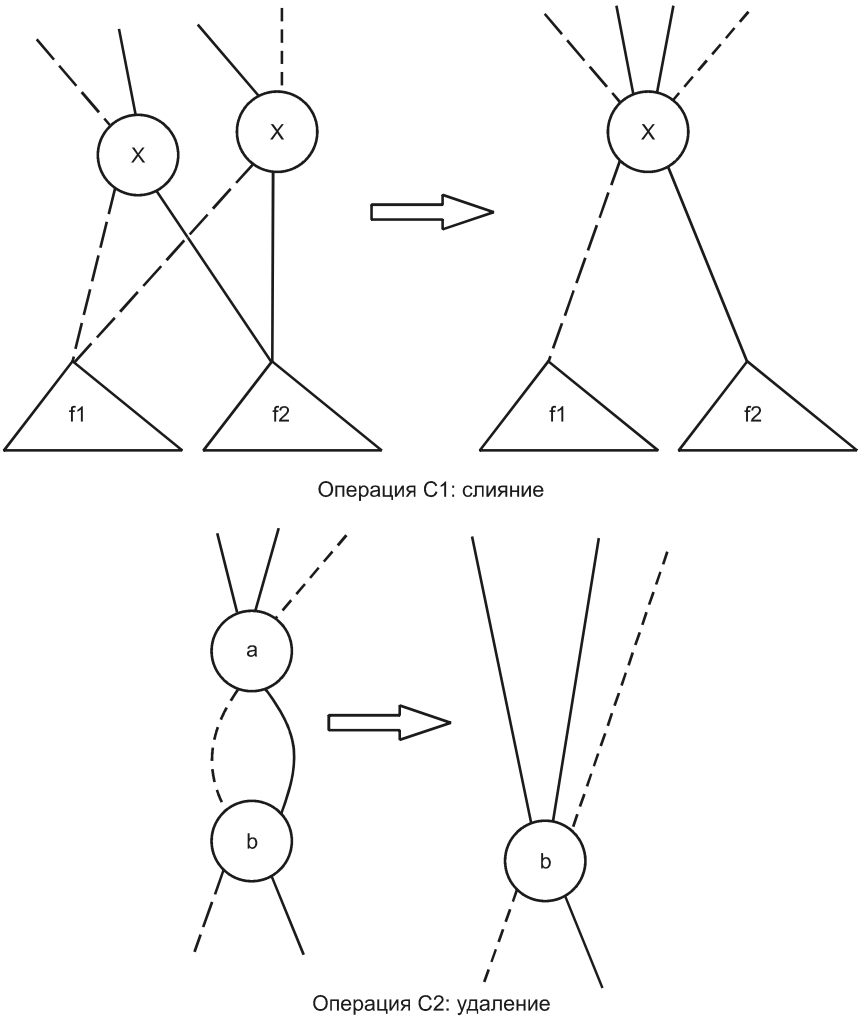


Рис. 9.3. Операции Reduce преобразования семантического дерева в BDD

Ключевым требованием к BDD является требование упорядоченности редуцированной бинарной решающей диаграммы. В редуцированной упорядоченной бинарной решающей диаграмме (Reduced Ordered Binary Decision Diagram, ROBDD) последовательность пометок переменных функции на всех

путях должна идти в любом, но одном и том же глобальном порядке, и на одном пути переменные не могут повторяться. Например, рис. 9.4 представляет бинарную решающую диаграмму, которая не упорядочена: на разных путях от корневой вершины к заключительной вершине переменные встречаются в разном порядке.

Обычно ROBDD называют просто BDD: при использовании термина BDD обычно понимается, что соответствующий граф и редуцирован, и упорядочен в соответствии с некоторым зафиксированным порядком переменных.

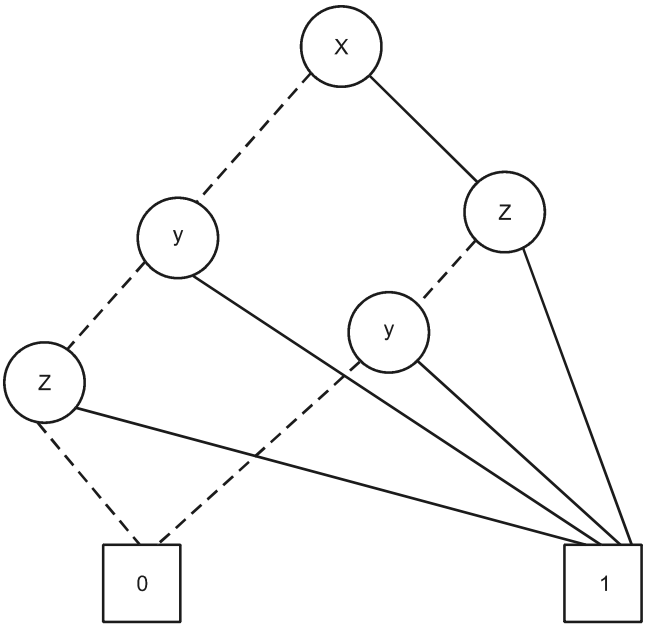


Рис. 9.4. BDD, но не ROBDD

BDD и разложение Шеннона. Двоичные решающие деревья и бинарные решающие диаграммы графически представляют разложение Шеннона:

$$f(x_1,\ldots,x_n) = x_i f|_{x_i=1} \vee \neg x_i f|_{x_i=0},$$

которое для любого аргумента x функции f может быть записано в виде

$$f = \textbf{\textit{if}}\ x\ \textbf{\textit{then}}\ f|_{x=1}\ \textbf{\textit{else}}\ f|_{x=0}.$$

Здесь, фактически, используется булева функция ***ite*** от трех булевых аргументов:

$$\textbf{\textit{ite}}(x,y,z) = \textbf{\textit{if}}\ x\ \textbf{\textit{then}}\ y\ \textbf{\textit{else}}\ z = xy \vee \neg xz,$$

которая соответствует разложению Шеннона:

$$f = \textbf{\textit{ite}}\left(x, f\,|_{x=1}, f\,|_{x=0}\right) = \textbf{\textit{if}}\ x\ \textbf{\textit{then}}\ f\,|_{x=1}\ \textbf{\textit{else}}\ f\,|_{x=0}\ .$$

Рис. 9.5 показывает, что любой узел двоичного решающего дерева представляет собой начальную вершину направленного ациклического графа, определяющего значение некоторой частично определенной булевой функции. Для корня эта функция исходная, для нашего примера это $f\left(p,q,r\right)$, для внутренних вершин соответствующие им функции — это частично определенные функции с конкретными (0 или 1) значениями некоторых переменных, для терминальных вершин — это константы 0 или 1, являющиеся значениями функции на конкретных наборах значений аргументов.

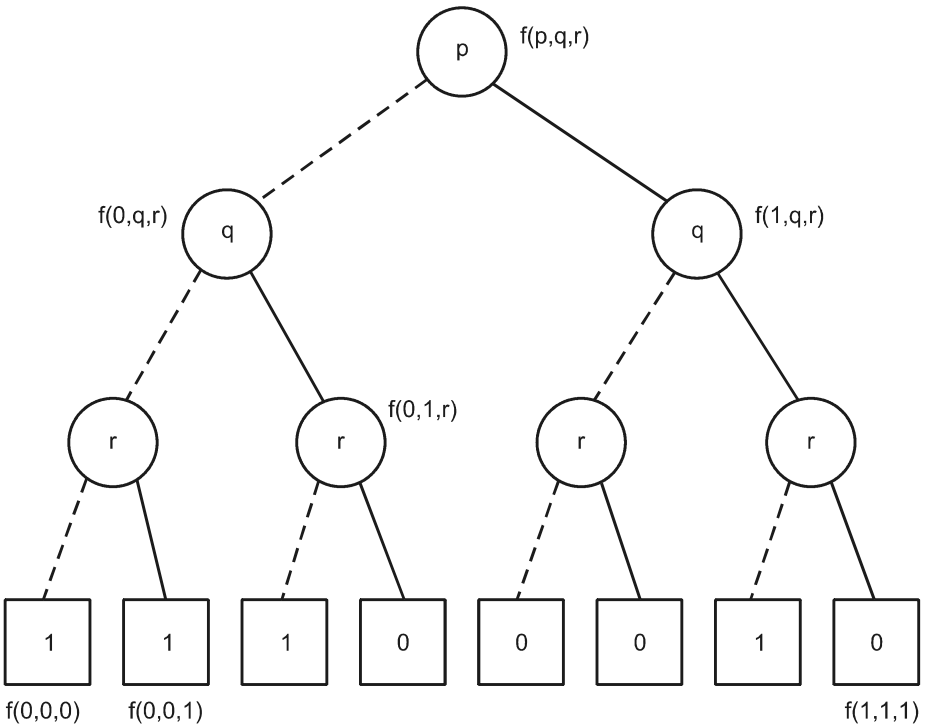


Рис. 9.5. Двоичное решающее дерево
как представление разложения Шеннона

Бинарная решающая диаграмма является представлением заданной функции в базисе $\{\,0\,,\ 1\,,\ \textit{ite}\,\}$ при фиксированном порядке переменных.

9.3. Свойства бинарных решающих диаграмм

Редуцированные упорядоченные бинарные решающие диаграммы имеют множество хороших свойств, которые делают это представление булевых функций очень удобным на практике.

BDD — каноническое представление булевых функций. Главным из свойств BDD является каноничность.

ТЕОРЕМА 9.1

Редуцированная упорядоченная бинарная решающая диаграмма является каноническим представлением булевой функции при любом фиксированном порядке переменных.

Таким образом, любая булева функция имеет единственное представление в виде BDD при выбранном порядке переменных, что дает возможность эффективного выполнения для BDD такой традиционно трудной операции, как проверка эквивалентности двух булевых функций. Проверка эквивалентности функций, представленных в BDD с совпадающим порядком переменных, сводится к проверке изоморфности (совпадения с точностью до обозначения вершин) двух графов. В общем случае алгоритм проверки изоморфности графов является NP-полным, но проверка изоморфности графов BDD очень проста именно из-за их специфики: ацикличности, единственного корня, фиксированного порядка переменных и наличия только двух направленных ребер, выходящих из каждой промежуточной вершины. Сложность этого алгоритма пропорциональна числу вершин бинарной решающей диаграммы, представляющей функцию. Если порядок переменных в двух сравниваемых BDD не совпадает, одну из них следует перестроить.

Сложность BDD мала. Еще одним важным свойством BDD является то, что сложность представления в форме BDD многих булевых функций полиномиальна или даже линейна. Например, линейный рост показывают BDD для многих аппаратных устройств: BDD для 4-разрядного сумматора имеет 31 вершину, а для 64-разрядного сумматора — всего 571 вершину. Это означает, что BDD используемых на практике функций от большого числа двоичных переменных (десятков и даже сотен) можно представить очень эффективно в памяти компьютера.

Сложность BDD зависит от порядка переменных. Разный порядок переменных дает разные представления функции в BDD. Для некоторых функций их BDD могут существенно отличаться по сложности для разных порядков переменных. На рис. 9.6 представлена BDD функции $F = a_1 b_1 \vee a_2 b_2 \vee \ldots \vee a_n b_n$,

которая при порядке переменных $a_1 < b_1 < a_2 < b_2 < a_3 < b_3 \ldots$ имеет линейную сложность (у нее $2n$ вершин), а при порядке переменных $a_1 < a_2 < a_3 < \ldots < b_1 < b_2 < b_3 \ldots$ сложность ее экспоненциальна (2^{n+1} вершин). Для заданной булевой функции можно попытаться найти порядок ее переменных, при котором ее BDD минимальна, но проблема нахождения такого оптимального порядка переменных в общем случае является NP-полной. Существуют эвристические алгоритмы для нахождения субоптимального порядка переменных в бинарных решающих диаграммах.

Следует отметить, что для некоторых классов функций их BDD имеют экспоненциальную сложность при любом порядке переменных. Например, рассмотрим функции, представляющие схему умножения двух двоичных n-разрядных чисел:

$$(F_1, F_2, \ldots, F_{2n}) = (a_1, a_2, \ldots, a_n) \times (b_1, b_2, \ldots, b_n).$$

Каждый разряд произведения представляет собой булеву функцию от $2n$ переменных:

$$F_i = F_i(a_1, \ldots, a_n, b_1, \ldots, b_n).$$

В [25] доказано, что BDD по крайней мере для одной из функций F_i имеет экспоненциальное число вершин при любом порядке переменных. В [150] представлен анализ BDD для схемы умножения двух 16-разрядных двоичных чисел. Эта BDD имеет более 40 миллионов вершин.

Однако для подавляющего большинства встречающихся на практике булевых функций сложность их представления в форме BDD линейна или полиномиальна. Наиболее экономно представляются BDD булевых функций регулярных аппаратных структур и структур, в которых есть симметрия.

Представление BDD. В памяти компьютера BDD представляются очень эффективно: их удобно представить таблицей, в каждой строке которой указываются номер вершины, номер переменной и номера вершин, в которые направлены два выхода из этого узла, 0-ребро и 1-ребро. Эти вершины, которые часто называются low и high, являются корнями решающих подграфов, определяющих значения соответствующих частично определенных функций. Явно отмечается корневая вершина. Для примера на рис. 9.7 для функции $f = \neg x_1 \vee x_2 x_3 x_4$ построена BDD и ее представление в виде таблицы и в виде списка. Сложность представления булевой функции в форме BDD в памяти компьютера пропорциональна числу вершин диаграммы. При этом обычно используется хеширование.

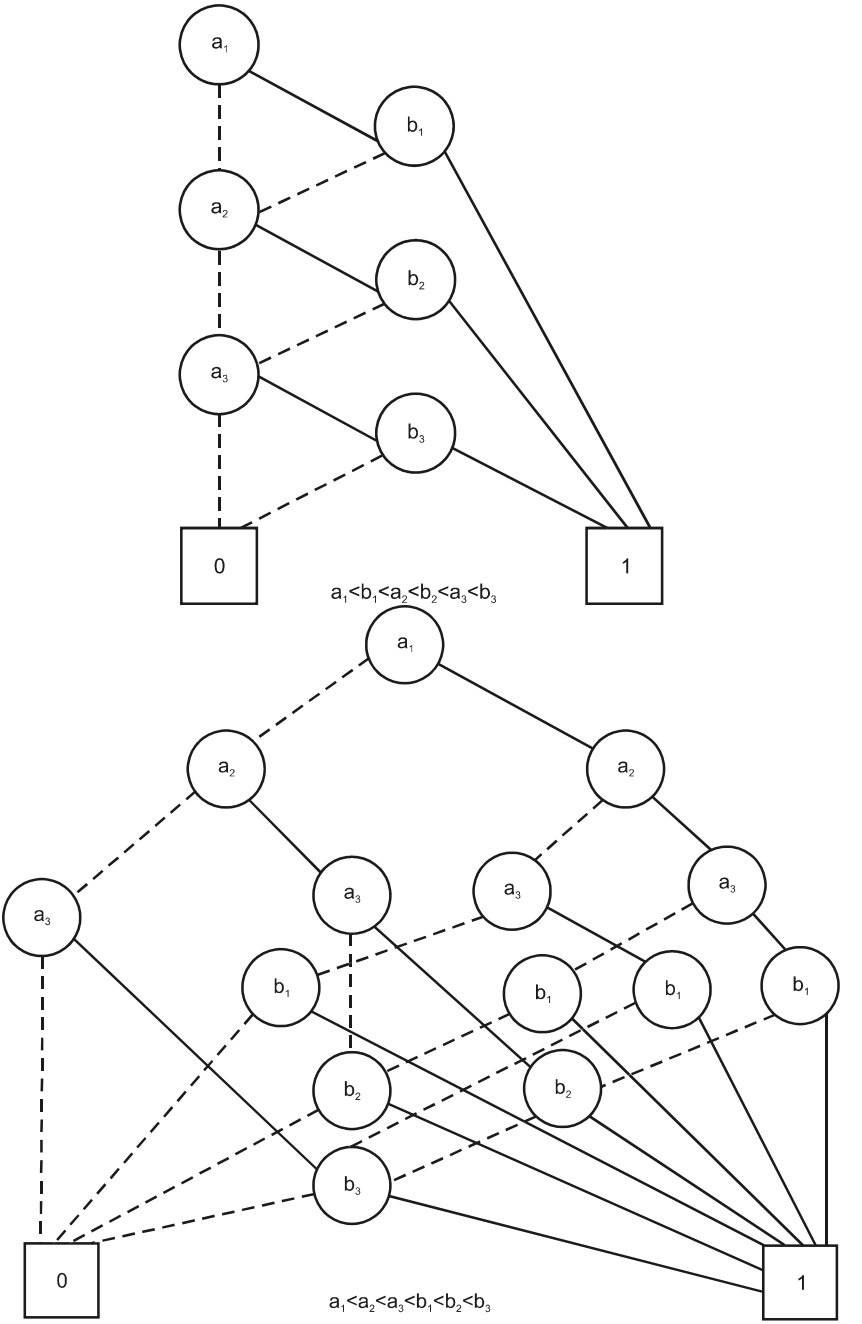
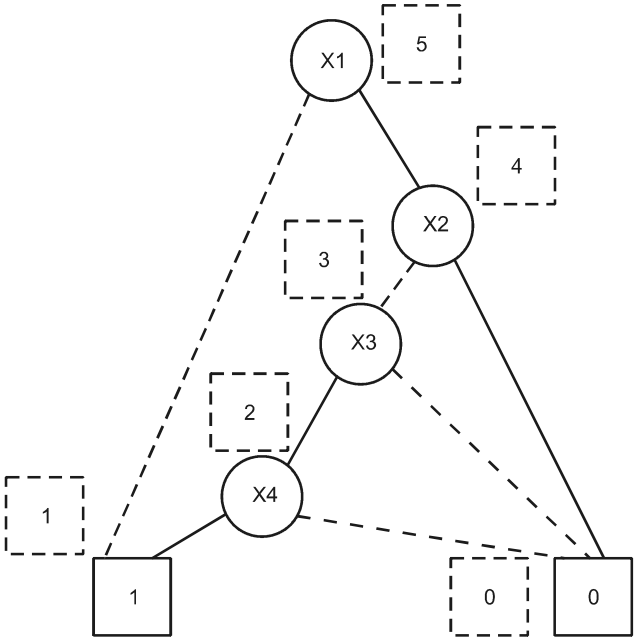


Рис. 9.6. Для функции $a_1b_1 \vee ... \vee a_nb_n$ при одном порядке переменных сложность BDD линейна, при другом — экспоненциальна

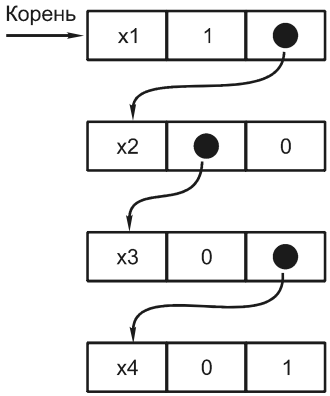


а

f - вершина 5

n	var	low	high
0			
1			
2	x4	0	1
3	x3	0	2
4	x2	3	0
5	x1	1	4

б



в

Рис. 9.7. Представление BDD таблицей и списком

9.4. Операции над BDD

Вернемся к операциям с булевыми функциями, которые перечислены в *разделе 9.1*:

- найти значение функции на конкретной интерпретации (наборе значений аргументов);
- найти все интерпретации, на котор19
- ых функция равна 1;
- проверить, является ли функция выполнимой, невыполнимой, общезначимой;
- проверить, совпадают ли две булевых функции;
- выполнить булевы операции над булевыми функциями;
- провести минимизацию функции в некотором базисе.

Все эти операции могут быть выполнены очень эффективно, если булевы функции представлены в BDD. Рассмотрим их по очереди.

Вычисление значения булевой функции. Вычисление значения булевой функции от n переменных на конкретном наборе значений аргументов по таблице истинности требует хранения таблицы с 2^n строками и манипуляциями с нею. Если функция представлена логической формулой, то такое вычисление может вылиться в сложную процедуру. Например, для функции $f(p,q,r)=(p\Rightarrow q)\wedge\neg(qr)$ это потребует значительного времени. Эта же функция может быть представлена и более сложными формулами, например:

$$f(p,q,r)=\left(\neg p\oplus\left(q\left(p\oplus r\right)\right)\right)\vee pq\neg r\vee(p\Rightarrow q)\,\&\,\neg(qr).$$

По такой формуле вычисление значения этой функции потребует очень большого времени.

Использование BDD позволяет выполнить эту операцию очень просто. Фактически, бинарная решающая диаграмма булевой функции представляет собой направленный граф для вычисления ее значения по значениям переменных. Алгоритм вычисления значения функции, представленной в BDD, — это несколько шагов (не более числа переменных) по этому графу. Любой набор значений аргументов определяет путь на графе, приводящий к терминальной вершине, 0 или 1. Например, BDD функции f представлена на рис. 9.2, и там же показано, как вычисляется значение этой функции при значениях ее аргументов $x_1=0, x_2=1, x_3=0$.

Для вычисления значений булевой функции *f*, вместо того чтобы двигаться по графу, по BDD этой функции можно сгенерировать C-подобный код, вы-

числяющий значение *f* при любых значениях аргументов. Например, по BDD, представленной на рис. 9.7, можно сгенерировать следующий текст, причем каждая строка этого кода соответствует одной вершине BDD:

```
bool f ( bool x1, x2, x3, x4 ) {
      goto m5;          // переход к головной вершине f
  m5: if ( x1==0 ) goto m1; else goto m4;
  m4: if ( x2==1 ) goto m0; else goto m3;
  m3: if ( x3==0 ) goto m0; else goto m2;
  m2: if ( x4==0 ) goto m0; else goto m1;
  m1: return (1);
  m0: return (0);
}
```

Сложность этой программы линейна: значение функции вычисляется всегда за число шагов, не большее числа переменных, от которых зависит функция.

Для вычисления функций, у которых заданы значения не всех аргументов, бинарные решающие диаграммы также удобны: на рис. 9.8 показаны BDD функции f и получение из нее BDD функции $f(p, q, 1)$.

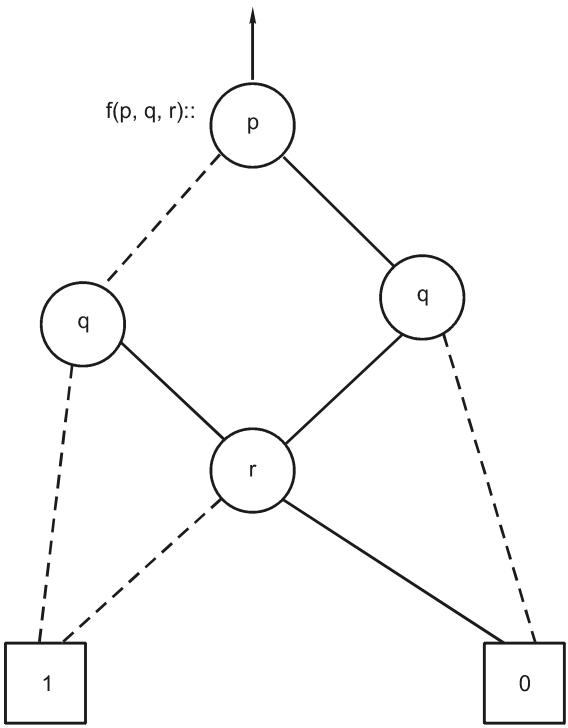


Рис. 9.8, *а*. BDD функции и BDD этой функции при задании значений некоторых аргументов

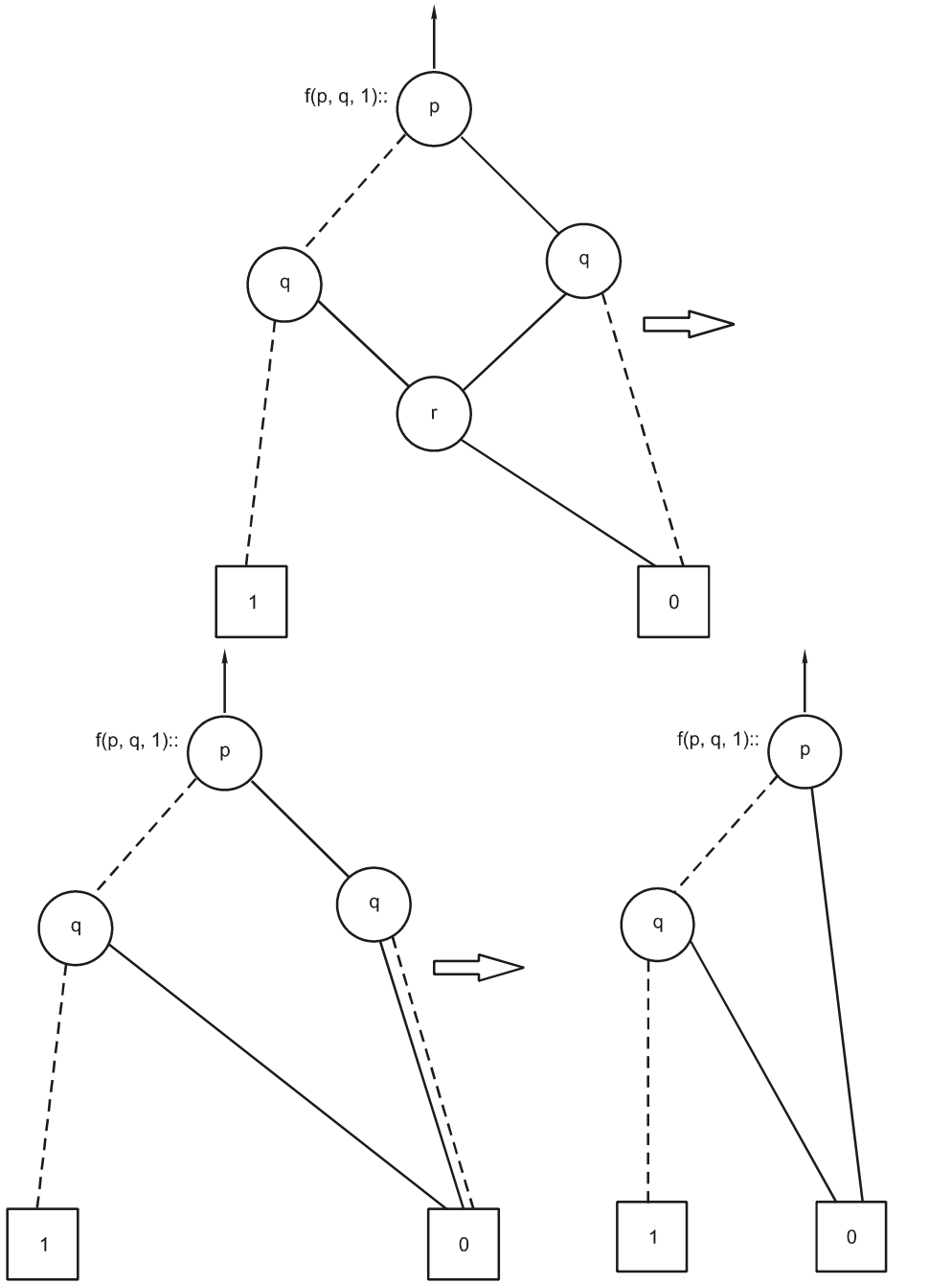


Рис. 9.8, *б*. BDD функции и BDD этой функции при задании значений некоторых аргументов

Найти все интерпретации, на которых функция равна 1. Любой путь на бинарной решающей диаграмме функции из корня в терминальную вершину, помеченную 1, определяет импликанту — конъюнкцию литералов (переменных с отрицаниями или без них), на которой значение этой функции равно 1.

Построение ДНФ и КНФ функции по BDD. BDD хранит богатую семантическую информацию о представляемой ею булевой функции f. Пути от корня к листу 1 компактно представляют все модели f (т. е. интерпретации, на которых $f = 1$). Все пути от корня к листу 0 компактно представляют все контрмодели (те интерпретации, на которых $f = 0$). Далее, если путь от корня к листу 1 рассматривать как конъюнкцию литералов и все такие конъюнкции дизъюнктивно объединить, то получим ДНФ функции f. Множество 0-путей легко интерпретировать как конъюнктивную нормальную форму функции f.

Пример 9.4

Функция $f(a,b,c)$, BDD которой представлена на рис. 9.9, может быть записана в дизъюнктивной нормальной форме (ДНФ) так: $f = ab \vee a\neg bc$. Следо-

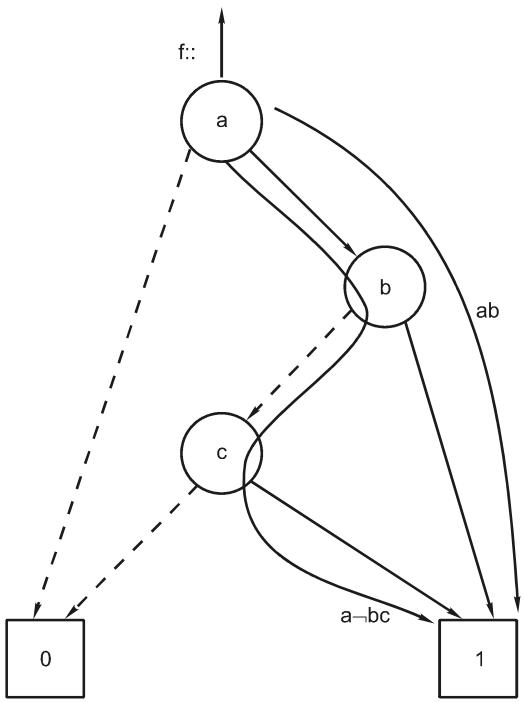


Рис. 9.9. Построение импликант функции по BDD

вательно, функция F принимает значение 1 на следующих наборах переменных: $a=1,\ b=1$ (при любых значениях переменной c) и $a=1,\ b=0,\ c=1$.

Проверка того, является ли функция выполнимой, невыполнимой или общезначимой. Такая проверка для булевой функции, представленной в форме BDD, осуществляется элементарно. BDD невыполнимой функции содержит единственную вершину, помеченную 0; BDD общезначимой функции содержит единственную вершину, помеченную 1; BDD выполнимой функции содержит вершину, помеченную 1 (рис. 9.10).

Проверка эквивалентности двух булевых функций. Проверка эквивалентности двух булевых функций, представленных в BDD, сводится к проверке изоморфности двух бинарных решающих диаграмм (при одном порядке переменных), поскольку BDD — каноническое представление функции.

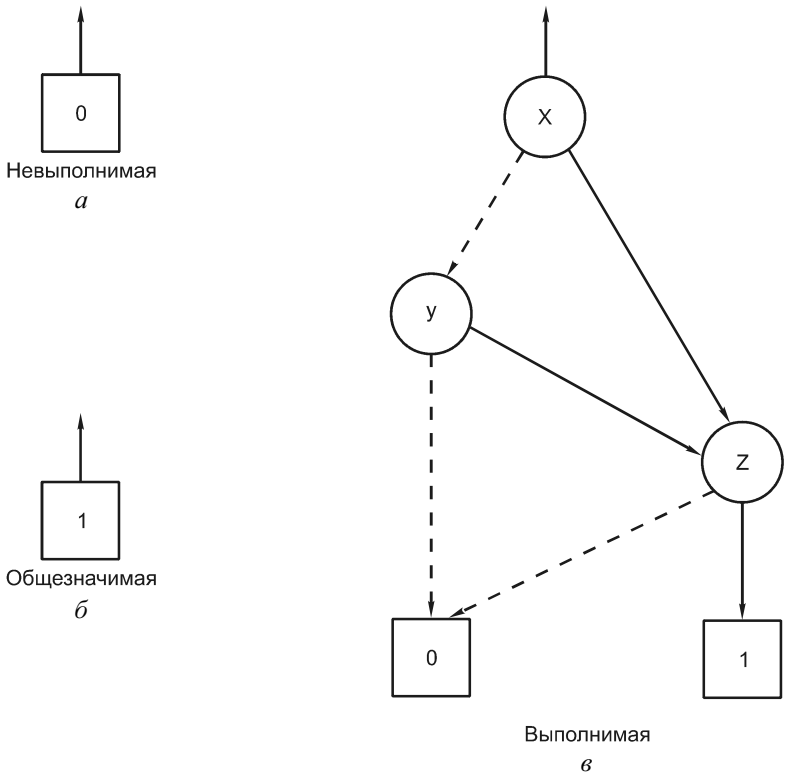


Рис. 9.10. Представление невыполнимой (*а*), общезначимой (*б*) и выполнимой (*в*) функций

Булевы операции над BDD. Булевы операции над двоичными функциями, представленными в BDD, выполняются очень просто. Операция $f|_{x=b}$ — подстановка булева значения b вместо переменной x в функции f, выполняется перенаправлением всех ребер диаграммы, входящих в каждую вершину v, помеченную x, в вершину, в которую направлено выходное b-ребро вершины v (см. рис. 9.8). Операция отрицания сводится к изменению значений 0 на 1 и 1 на 0 в терминальных вершинах диаграммы (рис. 9.11).

Бинарные логические операции над двумя BDD выполняются просто, если эти BDD построены в соответствии с одним и тем же порядком переменных.

В бинарной решающей диаграмме для любой промежуточной вершины v, помеченной переменной x, подграф, который начинается с вершины, в которую идет 0-ребро, назовем $low(v)$, а тот граф, в которую идет 1-ребро, назовем $high(v)$. Эти графы в общем случае пересекаются. Каждый из них представляет частичную функцию при различных значениях x. Алгоритм построения BDD, являющейся результатом применения бинарной булевой операции к двум заданным BDD, основан на следующих соотношениях. Очевидное равенство, которое представляет разложение Шеннона:

$$F = \textbf{\textit{if}}\ x = 0\ \textbf{\textit{then}}\ F|_{x=0}\ \textbf{\textit{else}}\ F|_{x=1}$$

запишем так (считая $F = f \otimes g$, где $\otimes$ — обозначение произвольной булевой операции):

$$f \otimes g = \textbf{\textit{if}}\ x = 0\ \textbf{\textit{then}}\ (f \otimes g)|_{x=0}\ \textbf{\textit{else}}\ (f \otimes g)|_{x=1} = \textbf{\textit{if}}\ x =$$
$$= 0\ \textbf{\textit{then}}\ (f|_{x=0} \otimes g|_{x=0})\ \textbf{\textit{else}}\ (f|_{x=1} \otimes g|_{x=1}).$$

Рекурсивное применение этого соотношения позволяет построить BDD функции $f \otimes g$ по диаграммам функций f и g. Рис. 9.12 показывает, как выполняется произвольная бинарная булева операция над двумя BDD с одним и тем же порядком переменных.

Обозначим подграф с вершиной v, помеченной переменной x, так:

$$[x \to p_0, p_1],$$

где p_0 и p_1 — подграфы, в которые из вершины v идут, соответственно, 0-ребро и 1-ребро.

Операции над диаграммами выполняются рекурсивно в соответствии со следующими правилами.

1. Если в вершинах диаграмм одна и та же переменная x, то:

$$[x \to p_0, p_1] \otimes [x \to q_0, q_1] = [x \to (p_0 \otimes q_0), (p_1 \otimes q_1)]$$

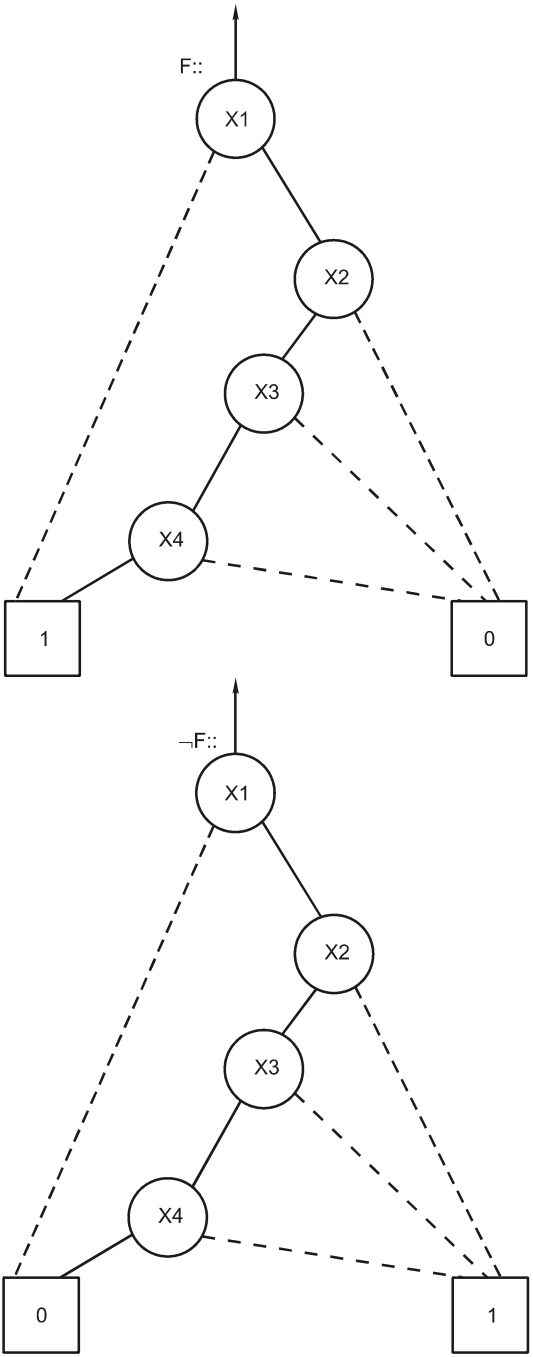


Рис. 9.11. Результат применения операции отрицания к функции, представленной в BDD

2. Если переменные x и y в вершинах диаграмм разные и $x < y$, то:

$$[x \to p_0, p_1] \otimes [y \to q_0, q_1] = \left[x \to \left(p_0 \otimes [y \to q_0, q_1] \right), \left(p_1 \otimes [y \to q_0, q_1] \right) \right]$$

Именно это показано на рис. 9.12.

После выполнения операций результирующая структура может быть не минимальна. Применение алгоритма *Reduce* приводит результат к требуемой форме BDD.

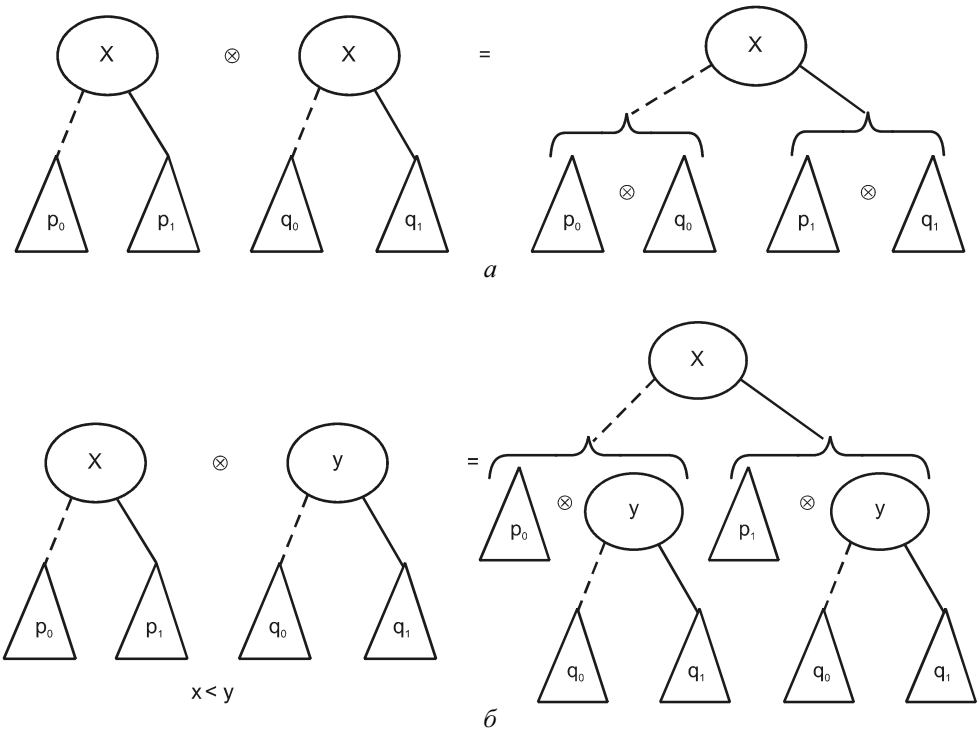


Рис. 9.12. Реализация бинарных булевых операций над двумя функциями, представленными в BDD

ТЕОРЕМА 9.2

Сложность выполнения бинарных булевых операций над двумя функциями f и g, представленными в форме BDD, полиномиальна: $O(|f| \times |g|)$.

Минимизация булевых функций, представленных в BDD. Набор функций $\{0, 1, \textbf{\textit{ite}}\}$ является базисом булевых функций. Иными словами, любая буле-

ва функция может быть представлена с помощью суперпозиции функций из этого множества. Например, функции базиса Буля (отрицание, дизъюнкция и конъюнкция) представятся так:

- отрицание $\neg x$: ***if*** x ***then*** 0 ***else*** 1
- дизъюнкция $x \vee y$: ***if*** x ***then*** 1 ***else*** y
- конъюнкция $x \wedge y$: ***if*** x ***then*** y ***else*** 0

Бинарная решающая диаграмма — это такое представление булевой функции в этом базисе, в котором все ее эквивалентные булевы подвыражения присутствуют один раз. Например, BDD на рис. 9.7 можно записать так:

$$f = f_5 = \textbf{\textit{if}}\ x_1\ \textbf{\textit{then}}\ f_4\ \textbf{\textit{else}}\ 1$$

$$f_4 = \textbf{\textit{if}}\ x_2\ \textbf{\textit{then}}\ 0\ \textbf{\textit{else}}\ 1$$

$$f_3 = \textbf{\textit{if}}\ x_3\ \textbf{\textit{then}}\ f_2\ \textbf{\textit{else}}\ 0$$

$$f_2 = \textbf{\textit{if}}\ x_4\ \textbf{\textit{then}}\ 1\ \textbf{\textit{else}}\ 0$$

Здесь исходная функция $f(x_1, x_2, x_3, x_4) = f_5$ соответствует корневой вершине, а функции f_2, f_3, f_4 соответствуют различным вершинам графа. Таким образом, BDD является минимальным графическим представлением булевой функции в базисе $\{0, 1, \textbf{\textit{ite}}\}$ при фиксированной упорядоченности переменных.

Классическая проблема построения минимальной ДНФ булевой функции, представленной в BDD, решается очень просто. Сначала из BDD можно построить все импликанты функции, перебрав все пути из корня в терминальную вершину, помеченную 1. Например, функция f, представленная BDD на рис. 9.19, имеет следующие две импликанты: ab, $a\neg bc$. Имея эти импликанты, можно далее использовать классические методы минимизации. Для нашего примера легко получается минимальная скобочная форма функции: $a(b \vee c)$. Современные алгоритмы двухуровневой минимизации используют BDD, что позволяет существенно повысить эффективность минимизации по сравнению с классическими алгоритмами [39]. Одна из первых работ, в которых исследовался эффект от применения BDD для минимизации логических функций, называлась "*Прорыв в двухуровневой минимизации логических схем*" [32].

Пример 9.5

Рассмотрим пример минимизации более сложной функции от 4 переменных:

$$F = x_1\neg x_2 x_3 x_4 \vee x_1 x_2 \neg x_3 x_4 \vee x_1 x_2 \neg x_3 \neg x_4 \vee x_1 x_2 x_3 \neg x_4 \vee \neg(\neg x_1 x_2 \neg x_3 x_4).$$

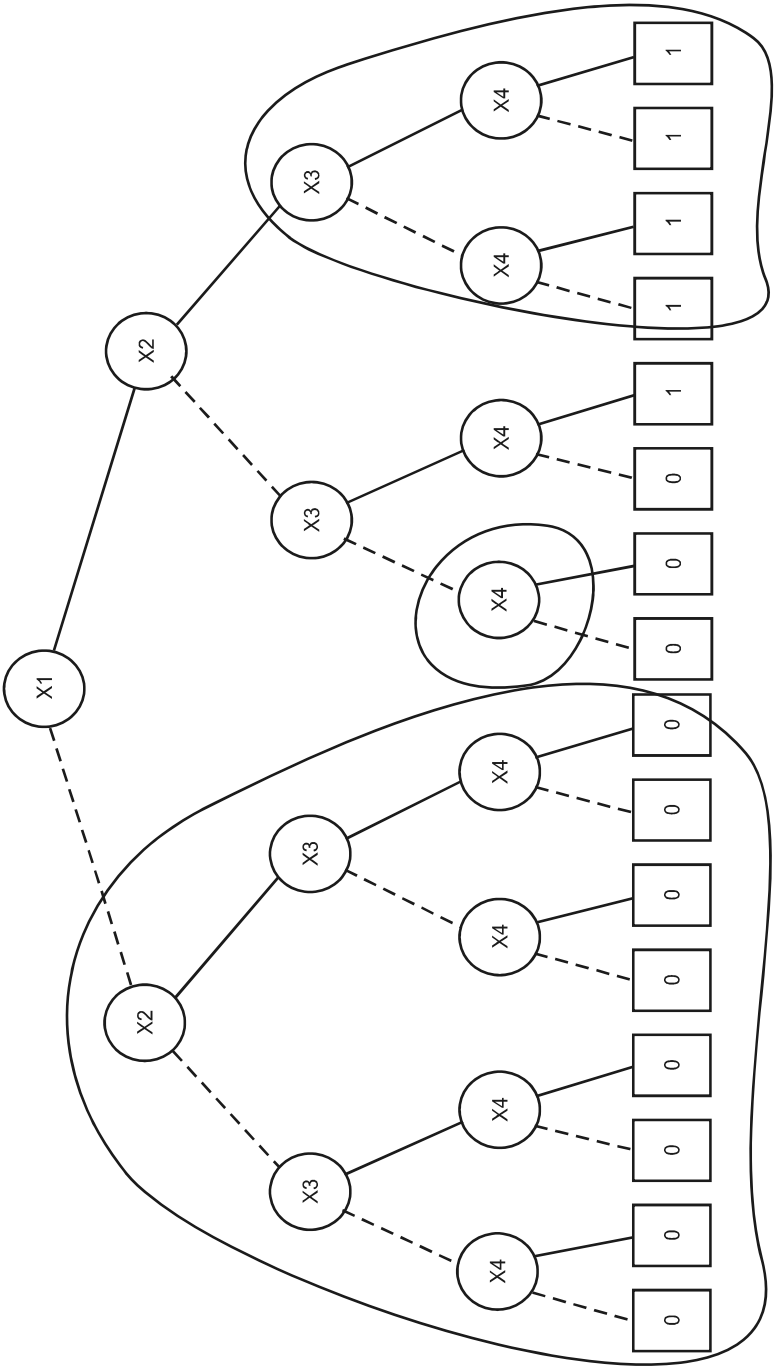


Рис. 9.13, *а*. Построение импликант булевой функции

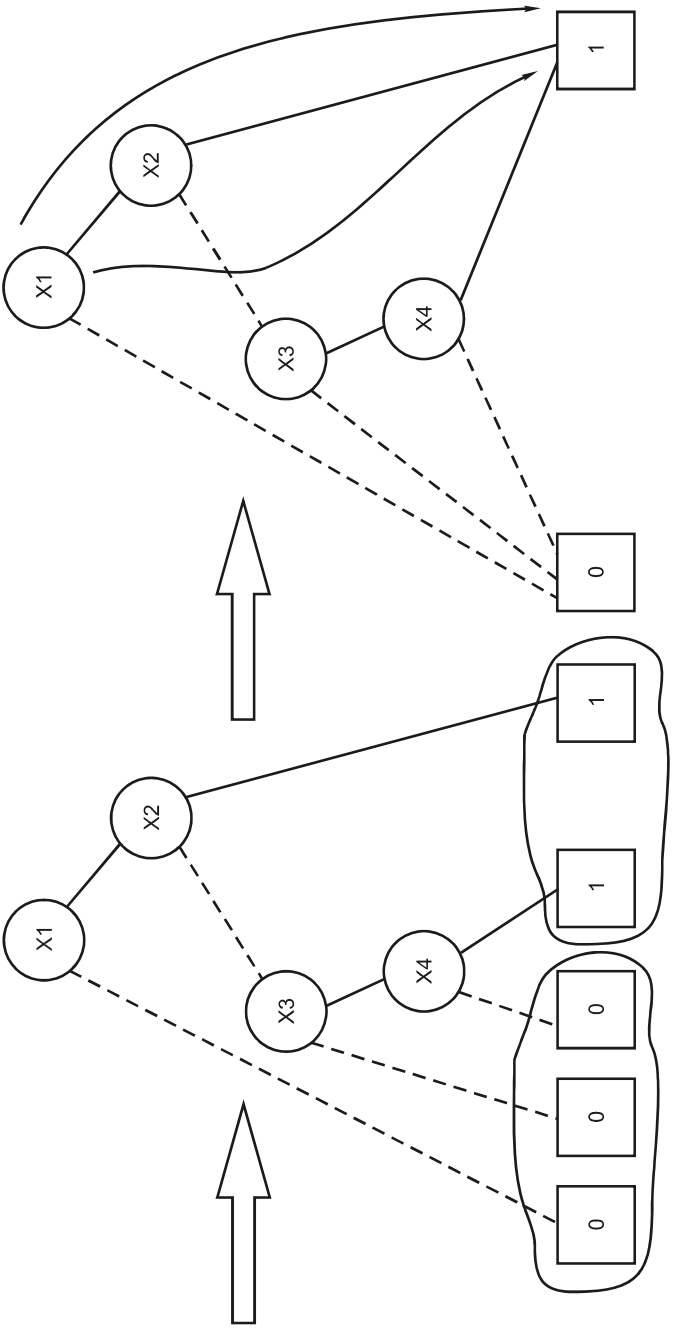


Рис. 9.13, *б*. Построение импликант булевой функции

Построим для этой функции BDD из ее двоичного решающего дерева (рис. 9.13). Из этого рисунка видно, что функция имеет две импликанты: x_1x_2 и $x_1\neg x_2x_3x_4$. Простое склеивание и вынос за скобки дает для этой функции минимальную скобочную форму $F = x_1\left(x_2 \vee x_3x_4\right)$.

Минимизация неполностью определенных булевых функций. Неполностью определенные булевы функции могут быть представлены с помощью расширений BDD. Одно из таких расширений, MBD (модифицированные BDD), предложено в работе [85]. В этой работе также изложены алгоритмы минимизации неполностью определенных булевых функций перенаправлением тех ребер графа MBD, которые ведут в терминальную вершину, помеченную X (don't care). Рис. 9.14 показывает пример такой минимизации.

Построение BDD. Построение BDD булевой функции можно выполнить после построения ее двоичного решающего дерева, как это показано на рис. 9.2 и 9.13. Но если функция представлена в формульной записи, можно применить простой алгоритм трансляции, выполняя синтаксический анализ заданной формулы (т. е. выделяя ее элементарные подформулы) и используя методы атрибутной трансляции.

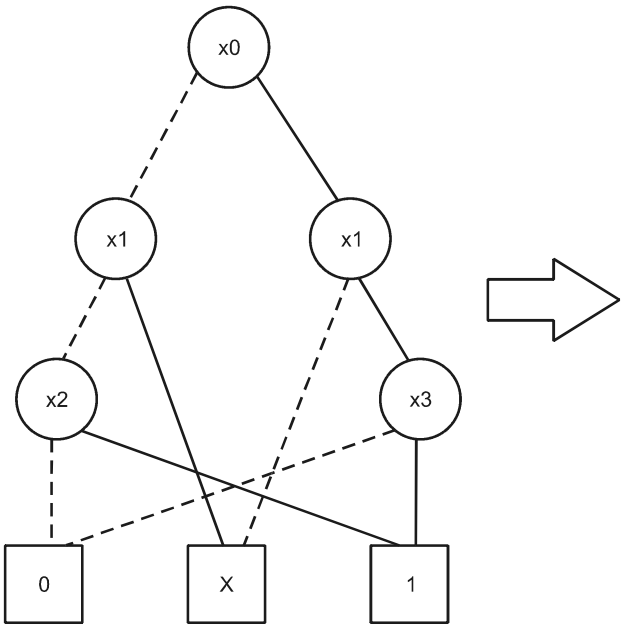
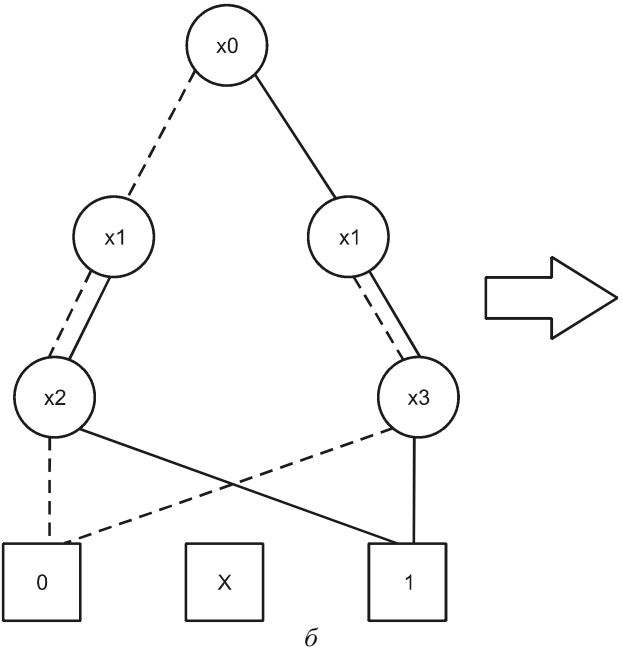
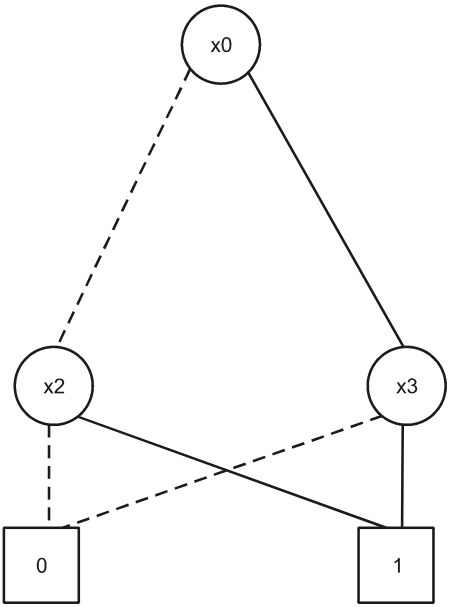


Рис. 9.14, *а*. Пример минимизации частично определенной функции



б



в

Рис. 9.14, *б* и *в*. Пример минимизации частично определенной функции

Пример 9.6

Построим BDD для булевой функции F следующего вида:

$$F = y \,\&\, (\neg z \Rightarrow x)$$

для порядка переменных $x < y < z$.

В соответствии с основной идеей синтаксически-ориентированной трансляции, такое построение должно выполняться в два этапа. На первом по грамматике языка логических формул, учитывающей приоритеты логических операций, строится синтаксическое дерево формулы F с помощью обычного восходящего алгоритма синтаксического анализа, например, алгоритма LALR(1) [159]. Синтаксическое дерево представляет собой структуру подформул данной формулы. Оно показано на рис. 9.15.

В синтаксическом дереве формулы F нашего примера шесть нетерминалов: f_1—f_6, с начальным нетерминалом f_6, которые представляют следующие подформулы формулы F:

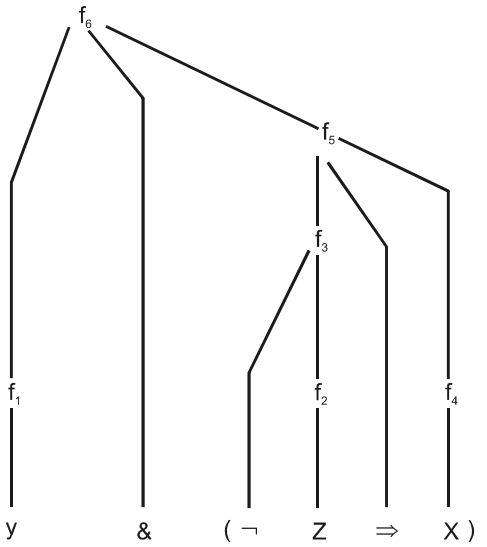
$$f_1 = y;\ f_2 = z;\ f_3 = \neg f_2;\ f_4 = x;\ f_5 = f_3 \Rightarrow f_4;\ F = f_6 = f_1 \,\&\, f_5.$$

По этому дереву для каждой подформулы, начиная с простейших, вычисляются ее BDD на основе правил атрибутной семантики.

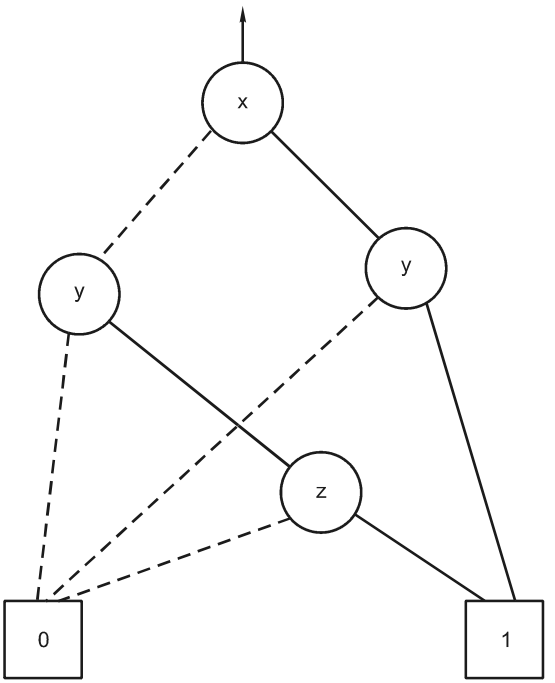
При атрибутной трансляции формулы в соответствующую BDD для терминала x семантическим атрибутом является граф $[x \to 0, 1]$, т. е. BDD с одним корнем, помеченным x, и двумя терминальными вершинами, 0 и 1.

Семантическим атрибутом любого нетерминала N при решении нашей проблемы естественно выбрать BDD той подформулы, которая выводится из N. Так, семантическим атрибутом нетерминала f_1 будет BDD подформулы y, нетерминала f_2 — BDD подформулы z, нетерминала f_3 — BDD подформулы $\neg z$ и т. д. Семантическим атрибутом нетерминала f_6 будет искомая BDD всей формулы $F = y \,\&\, (\neg z \Rightarrow x)$, которая строится на основе правила $F = f_1 \,\&\, f_5$ как результат операции $\&$, примененной к BDD подформул f_1 и f_5, построенным на предыдущих шагах.

На рис. 9.16 показано последовательное построение BDD для нетерминалов f_1—f_6 на основе приведенных выше правил выполнения унарной (отрицание) и бинарных операций над BDD. В результате применения этого алгоритма мы получим BDD нетерминала f_6, совпадающую с BDD, приведенной на рис. 9.15, *б*.



а



б

Рис. 9.15. Синтаксическое дерево формулы $y \,\&\, (\neg z \Rightarrow x)$ и BDD функции $F = y \,\&\, (\neg z \Rightarrow x)$

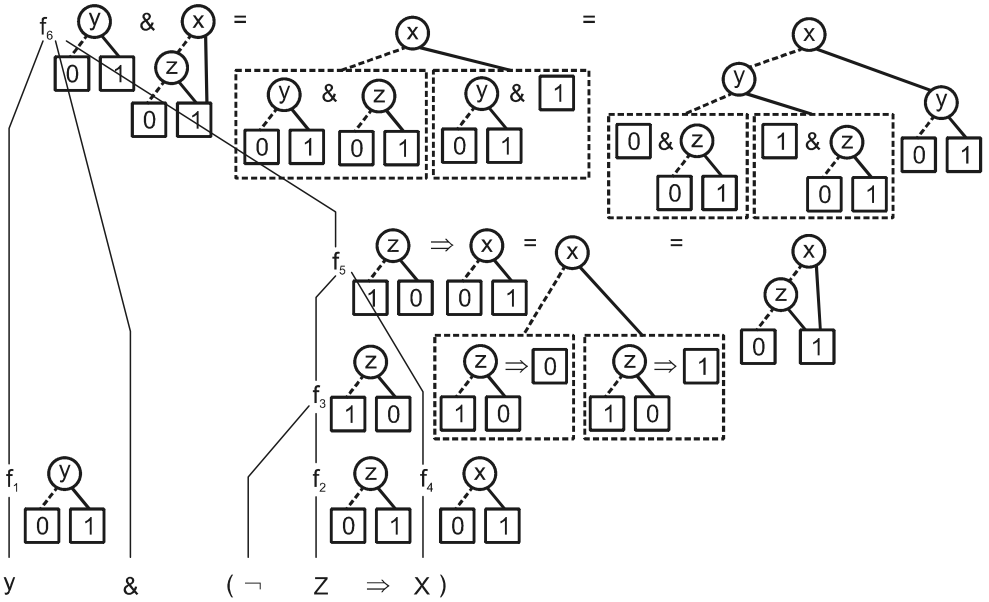


Рис. 9.16. Последовательное построение BDD для всех подфункций функции $y \& (\neg z \Rightarrow x)$

Библиотеки. Существует несколько свободно распространяемых библиотек, в которых реализованы алгоритмы манипуляций с BDD. Библиотеки включают программы для построения BDD по формульной записи булевой функции, выполнения булевых операций над BDD, вычисления значения функции для конкретного набора значений аргументов, сравнения двух булевых функций, представленных в BDD, и многие другие полезные операции. Перечислим лишь некоторые из этих библиотек:

- ABCD: The ABCD package by Armin Biere, http://fmv.jku.at/abcd/.
- BuDDy: A Package by Jørn Lind-Nielsen, http://sourceforge.net/projects/buddy/.
- CMU BDD: CMU, http://www.cs.cmu.edu/~modelcheck/bdd.html.
- CuDD: University of Colorado, http://vlsi.colorado.edu/~fabio/CUDD/.
- JavaBDD, http://javabdd.sourceforge.net/.

9.5. Проблема булевой выполнимости (SAT) и бинарные решающие диаграммы

Проблема SAT. Проблема булевой выполнимости (проблема SAT, от английского слова satisfiability) является центральной проблемой вычислительной науки с множеством применений. Проблема состоит в следующем. Дана формула логики высказываний в КНФ, например:

$$f = (b \vee \neg c \vee d)(a \vee b)(\neg a \vee b \vee c)(a \vee \neg b \vee \neg d).$$

Определить, выполнима ли эта формула.

Проблема выполнимости является фундаментальной проблемой, к которой сводится решение многих проблем из логики, машинного обучения, задач удовлетворения ограничениям, синтеза логических схем.

В 1971 г. Стивеном Куком был впервые введен класс "NP-полных задач", и SAT была первой задачей из этого класса. Класс NP-полных задач характеризуется интересным свойством универсальности: любая из задач этого класса полиномиально сводится к другой NP-полной задаче. Наиболее удобно сводить такие задачи именно к задаче SAT из-за большой гибкости и удобства кодирования параметров задач и отношений между ними булевыми переменными и булевыми функциями. После появления результатов Кука была доказана NP-полнота для множества других задач сведением их к задаче SAT. Задача коммивояжера, составление расписаний, поиск гамильтонова цикла в графе, проблема раскраски — все они являются NP-полными задачами. Вследствие этого SAT можно считать центральной проблемой среди многих вычислительно трудно решаемых проблем, и создание эффективных алгоритмов решения этой проблемы чрезвычайно важно для решения проблем из множества областей.

Любая задача из класса NP-полных задач может быть решена за полиномиальное время *недетерминированным* алгоритмом, в котором выбирается всегда верный путь решения. Разработанные на сегодняшний день *детерминированные* алгоритмы требуют в общем случае экспоненциального времени для NP-полных задач. В настоящее время неизвестно, можно ли детерминированным алгоритмом решить NP-полную задачу за полиномиальное время, т. е. верно ли соотношение $P = NP$? На практике полиномиальная сложность задачи обычно означает относительно легкое ее решение. Очевидно, что решение за полиномиальное время хотя бы одной из NP-полных задач автоматически приведет к относительно легкому решению всех NP-полных задач.

Проблема $P = NP$ является одной из центральных проблем современной информатики. Эта проблема имеет действительно важное практическое значе-

ние. Например, современная практическая криптография, использующая системы криптозащиты RSA или DES/AES, основана на том предположении, что NP-полные задачи трудноразрешимы, и появление полиномиального алгоритма, решающего NP-полные задачи, поставит под вопрос защищенность современных криптосистем. Проблема $P = NP$ указана первой среди семи важнейших проблем, стоящих перед математикой на рубеже XXI века (Millenium Problems), за решение каждой из которых Clay Mathematical Institute, США, объявил награду в 1 млн долларов. Эти проблемы определены Институтом как "*важные классические задачи, решение которых не найдено в течение многих лет*". Одна из этих проблем — проблема Пуанкаре — была решена петербургским математиком Григорием Перельманом (впрочем, от награды отказавшимся).

Проблема SAT и BDD. Для булевой функции, уже представленной в форме BDD, проблема SAT вообще не требует никакого поиска, она сводится к проверке того, есть ли в BDD терминальная вершина 1. Сложность задачи нахождения набора значений переменных, на котором булева функция, представленная в BDD, равна 1, составляет $O(n)$, где n — число переменных. Поэтому возникает вопрос: является ли это доказательством того, что $P = NP$?

В настоящее время разработаны очень эффективные решатели проблемы SAT, основанные на модифицированном алгоритме Дэвиса — Патнема, работающие с тысячами булевых переменных. Они используются во многих прикладных областях: проверка эквивалентности логических схем, диагностика отказов, планирование и анализ игровых стратегий в искусственном интеллекте и пр. В худшем случае, однако, SAT-решатели требуют экспоненциального времени, и использование алгоритма построения BDD в некоторых случаях действительно упрощает решение задачи SAT. Решатели, основывающиеся на представлении функции в форме BDD, не проводят поиск: их задача — просто привести булеву функцию в форму BDD. Если функция невыполнима, ее BDD представление при любом порядке переменных состоит из единственной вершины, помеченной 0. Алгоритмы построения BDD для некоторых классов задач работают полиномиальное время там, где SAT-решатели работают экспоненциальное время. Поэтому алгоритмы построения BDD и их варианты стали основой для разработки нескольких решателей задачи SAT, которые эффективны при решении многих практических проблем. В работе [123] показано, как необходимость решения проблемы выполнимости возникает в области проектирования асинхронных схем и описан эффективный алгоритм решения этой проблемы, объединяющий методы ее решения на основе идей Дэвиса — Патнема с использованием BDD для представления булевых функций.

Оказывается, однако, что в худшем случае построение представления булевой функции в BDD также занимает экспоненциальное время, а SAT-решатели для некоторых из таких функций работают полиномиальное время. Поэтому в одних случаях предпочтительнее использовать классические SAT-решатели, а в других — строить BDD-представление для заданной булевой функции. В работе [24] замечено"*соревнование между традиционным подходом к решению проблемы SAT и подходом, основанным на BDD, привело к прогрессу в обоих подходах*". Но известно также, что существуют задачи, на которых оба класса алгоритмов работают экспоненциальное время.

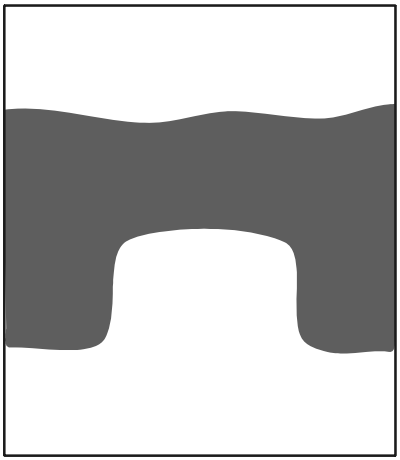
Таким образом, несмотря на существенный прорыв в решении практических задач в области обработки дискретных структур данных, который обеспечивают BDD, проблема "P=NP?" остается неразрешенной.

9.6. Применения BDD к решению логических и комбинаторных задач

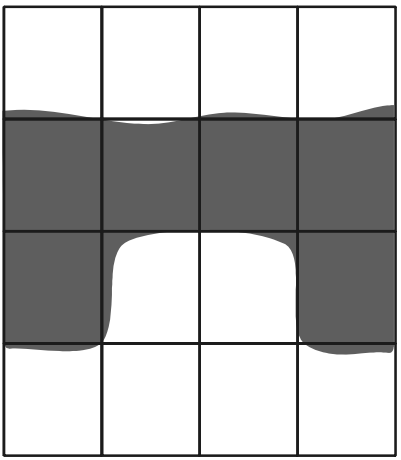
Использование представления BDD для булевых функций позволяет существенно повысить эффективность алгоритмов решения многих сложных задач. Рассмотрим некоторые из них.

Компрессия изображений. Любое двумерное изображение можно представить прямоугольной таблицей пикселов — минимальных элементов цифрового двумерного изображения в растровой форме, обладающих фиксированным цветом. Рассмотрим для простоты двумерное черно-белое изображение. Оно может быть представлено бинарной функцией, значения 0 и 1 которой соответствуют белой или черной закраске пикселов. Рис. 9.17 показывает связь изображения и булевой функции, которую для компрессии можно представить в виде BDD. Такая компрессия осуществляется без потерь, и во многих случаях, особенно когда изображение имеет регулярности, очень эффективна. Например, для представления мегабайтного изображения нужна булева функция от 23 переменных (поскольку $\left\lceil \log\left(8*10^6\right) \right\rceil = 23$, и BDD функции от такого числа переменных обычно легко представляются в памяти компьютера. Очевидно, что эту технику легко расширить и на изображения с несколькими градациями серого цвета, и на цветные.

Использование BDD для информационного поиска. Для представления информации в поисковых системах используются инвертированные списки. В работе [96] обсуждается использования BDD для сжатия информации, содержащейся в инвертированных списках. Рис. 9.18 демонстрирует идею, обсуждаемую в этой работе.



а



б

x_1x_2 \ x_3x_4	10	11	01	00
10	0	0	0	0
11	1	1	1	1
01	1	0	0	1
00	0	0	0	0

в

Рис. 9.17. Компрессия изображения: *а*) черно-белое изображение; *б*) прямоугольная таблица пикселов; *в*) булева функция, кодирующая изображение

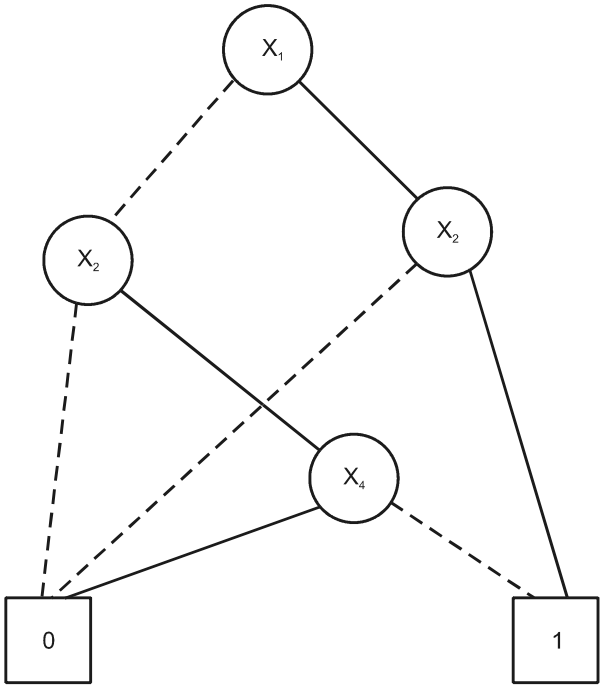


Рис. 9.17. Компрессия изображения: *г*) BDD, представляющая изображение

Для каждого термина в инвертированном файле хранится список номеров документов, в которых этот термин встречается. Так, на рис. 9.18, *а* показано, что термин "intensive" встречается в документах 0, 2, 3,..., 15, термин "inversion" в документах 2, 6, 7, ... и т. д. Пусть всего имеем 16 документов. Закодируем все документы двоичными кодами. Документ 0 закодируем кодом 0000, документ 1 — 0001 и т. д. Для каждого термина построим булеву функцию от 4 аргументов. Эта функция будет равна 1 на двоичных наборах, которыми закодированы те документы, в которых этот термин встречается. Рис. 9.18, *б* показывает такую функцию для термина "intensive". При большом числе документов хранить такую функцию удобно в форме BDD (рис. 9.18, *в*). Этот подход существенно ускоряет обработку сложных запросов в больших БД, когда требуется выполнение булевых операций И, ИЛИ, НЕ и других над инвертированными списками.

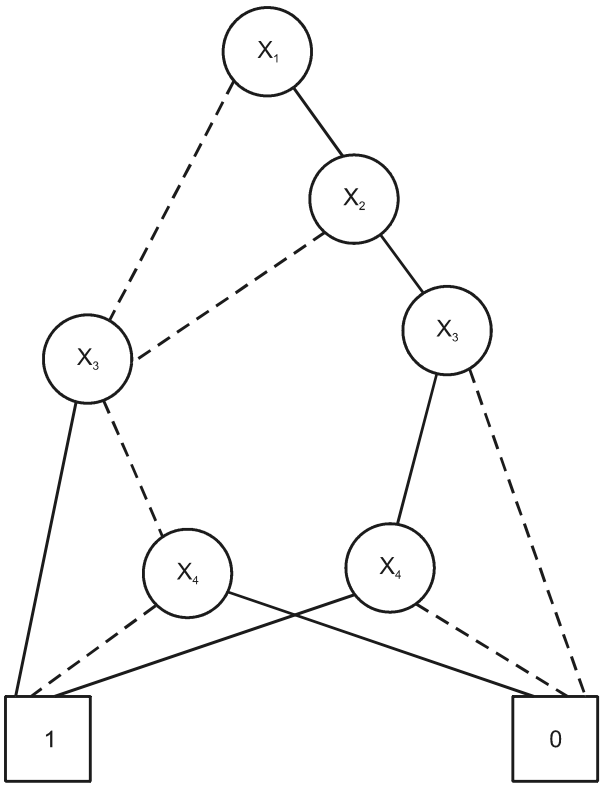
Задача о расстановке ферзей. Одной из знаменитых задач на шахматной доске является задача о восьми ферзях: расставить на шахматной доске 8 ферзей так, чтобы они не били друг друга. Известно много способов решения этой задачи, все они сводятся к организации перебора.

intensive	0, 2, 3, 4, 6, 7, 8, 10, 11,15
inversion	2, 6, 7, 8, 10, 11, 12, 14
invert	0, 1, 5, 8, 9, 11, 13, 15
inverter	3, 5, 6, 10, 11, 12
invest	1, 2, 4, 5, 7, 9, 10, 12,15

а

$x_1x_2x_3x_4$	f
0000	1
0001	0
0010	1
0011	1
0100	1
0101	0
0110	1
0111	1
1000	1
1001	0
1010	1
1011	1
1100	0
1101	0
1110	0
1111	1

б



в

Рис. 9.18. Использование BDD для представления инвертированных файлов:
а) фрагмент инвертированного файла для терминов;
б) булева функция для термина "intensive",
в) BDD этой функции

Для решения задачи без перебора построим булеву функцию f, аргументами которой являются 64 переменных $x_{i,j}, i, j \in \{1, ..., 8\}$, причем $x_{i,j} = 1$, если на i-й горизонтали и j-й вертикали стоит ферзь, и $x_{i,j} = 0$, если ферзя в этой позиции нет. Функция f строится таким образом:

ЕСЛИ ферзь стоит в позиции $\langle i, j \rangle$, ТО $\left(x_{i,j} \Rightarrow\right)$:

- на i-й горизонтали нет ферзей: $\left(\Lambda_{1 \le k \le 8,\ k \ne j} \neg x_{i,k}\right)$ И
- на j-й вертикали нет ферзей: $\left(\Lambda_{1 \le k \le 8,\ k \ne i} \neg x_{k,j}\right)$ И
- на С-З диагонали нет ферзей: $\left(\Lambda_{1 \le k \le 8,\ 1 \le j+k-i \le 8,\ k \ne i} \neg x_{k,\ j+k-i}\right)$ И
- на С-В диагонали нет ферзей: $\left(\Lambda_{1 \le k \le 8,\ 1 \le j+i-k \le 8,\ k \ne i} \neg x_{k,\ j+i-k}\right)$ И
- на каждой горизонтали есть ферзь: $\left(\Lambda_{1 \le i \le 8} \left(x_{i,1} \vee x_{i,2} \vee ... \vee x_{i,8}\right)\right)$.

Конъюнкция всех этих формул для всех $i, j \in \{1, ..., 8\}$ даст булеву функцию f от 64 переменных, такую, что любая интерпретация (набор значений двоичных переменных), на которой $f = 1$, будет решением проблемы.

Представление булевой функции от 64 двоичных переменных в любой другой форме, кроме BDD, нереально. В то же время построение бинарной решающей диаграммы этой функции с использованием одной из перечисленных выше библиотек является простым делом. Перечисление тех интерпретаций булевой функции, представленной в BDD, на которых функция равна 1, является стандартной операцией этих библиотек.

Программирование в ограничениях. Программирование в ограничениях состоит не в описании алгоритма решения задачи, а декларативной спецификации допустимых состояний среды, которые задаются набором отношений между параметрами среды.

Формально постановка задачи в парадигме программирования в ограничениях формулируется следующим образом. Пусть на множестве переменных $X^1, ..., X^n$, областями значений которых являются множества $D^1, ..., D^n$, заданы ограничения $C_k\left(X^1, ..., X^n\right)$, $k \in \{1, ..., K\}$. Ограничения выражаются уравнениями, неравенствами, логическими соотношениями и т. п. Требуется найти множество допустимых состояний среды, т. е. такие наборы

$\langle a^1, ..., a^n \rangle$ $(a^j \in D^j)$ значений переменных $X^1, ..., X^n$, которые удовлетворяли бы всем ограничениям одновременно. Подобные задачи возникают при решении задач планирования, проектирования, прогнозирования и др. В монографии [169] этот класс задач называется "*Задачи удовлетворения ограничений* (Constraint Satisfaction Problems, CSP)".

Сложность решения задач этого класса обуславливается огромным пространством поиска, в то время как вычислительные алгоритмы играют здесь второстепенную роль. В качестве основного метода решения задач CSP обычно используются эвристические алгоритмы направленного поиска, алгоритмы поиска с возвратами, методы, позволяющие сократить пространство поиска, и т. п.

Если множества D^1, ..., D^n возможных значений переменных конечны, то решение задачи CSP может быть выполнено без перебора и без эвристик с помощью построения булевых функций следующим образом. Все возможные значения в каждой области D^j закодируем двоичными наборами. Каждое ограничение C_k представим логической функцией f_k, а все множество заданных ограничений представим конъюнкцией этих функций $F = \&_k f_k$. Решение задачи определится такими наборами кодировок, на которых $F = 1$. Как правило, построенная таким образом функция F зависит от большого числа двоичных переменных, и обычные формы представления функции не позволяют решать сколько-нибудь сложные задачи CSP. Если булевы функции представить в форме BDD, то решение задач может существенно упроститься.

Пример 9.7

Рассмотрим задачу из области CSP, в которой использование булевых функций заменяет поиск. Задача эта называется SUBSET-SUM, она состоит в том, что для заданного конечного множества $S = \{m_1, m_2, ..., m_k\}$ натуральных чисел и некоторого целого M необходимо найти подмножества таких элементов из S, сумма которых равна M. Известно, что задача SUBSET-SUM является NP-полной.

Построим булеву функцию F, которая имеет k двоичных входов и равна 1 на таких двоичных наборах $\langle x_1, ..., x_k \rangle$, что $\sum_{r=1}^{k} (x_r \times m_r) = M$. Рис. 9.19 показывает, как можно построить такую функцию. Ее основной составной частью является функция $f : \{0, 1\}^k \to \{0, 1\}^n$, которая каждому двоичному набору $\langle x_1, ..., x_k \rangle$ ставит в соответствие двоичный код суммы чисел

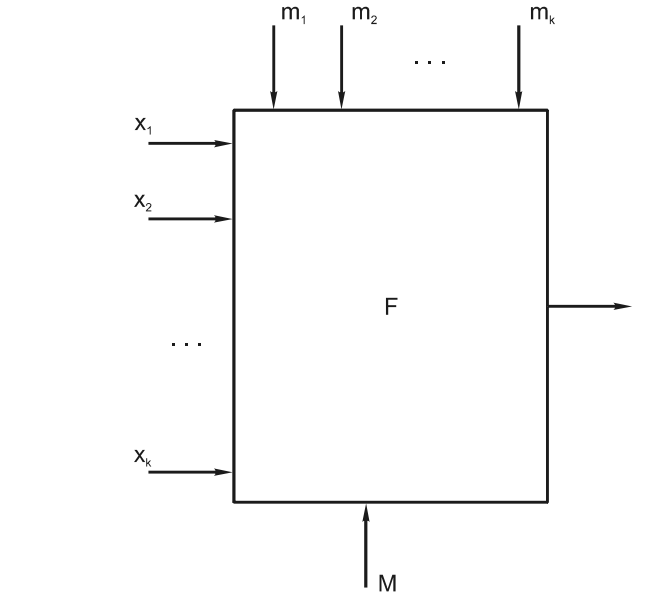
$\sum_{r=1}^{k}(x_r \times m_r)$. Иными словами, результатом функции f является вектор двоичных функций $\langle f_1, ..., f_n \rangle$ длиной n, представляющий собой двоичные разряды этой суммы. Каждая функция f_j представляет j-й разряд двоичного кода сумм соответствующих целых из множества S (здесь n — число двоичных разрядов, достаточное для представления в двоичном виде суммы всех чисел множества S). Функция F определяется соотношением $F = \&_i (f_i \equiv y_i)$, где y_i — i-й разряд двоичного представления числа M.

Пусть, например, множество S такое: $\{m_1, m_2, m_3\} = \{4, 5, 12\}$. Тогда вектор функций f_i может быть задан следующей таблицей истинности:

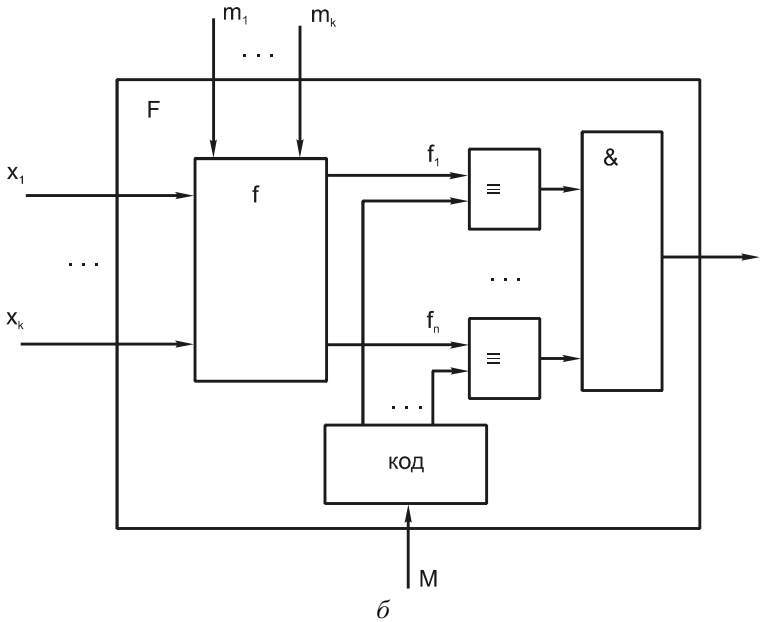
$x_3\ x_2\ x_1$	$f_5\ f_4\ f_3\ f_2\ f_1$	**Пояснение**
0 0 0	0 0 0 0 0	0
0 0 1	0 0 1 0 0	код m_1 (число 4)
0 1 0	0 0 1 0 1	код m_2 (число 5)
0 1 1	0 1 0 0 1	код m_2+m_1 (число 9)
1 0 0	0 1 1 0 0	код m_3 (число 12)
1 0 1	1 0 0 0 0	код m_3+m_1 (число 16)
1 1 0	1 0 0 0 1	код m_3+m_2 (число 17)
1 1 1	1 0 1 0 1	код $m_3+m_2+m_1$ (21)

Решение задачи SUBSET-SUM состоит в построении вектора булевых функций $\langle f_1, ..., f_n \rangle$. Функции эти можно построить, перебирая все возможные наборы двоичных переменных $x_1, ..., x_k$ и подсчитывая двоичный код сумм $\sum_{r=1}^{k}(x_r \times m_r)$. Очевидно, что сложность такого алгоритма экспоненциальна.

Более эффективный алгоритм построения вектора функций f_i состоит в построении их сразу в форме BDD с последовательным увеличением числа аргументов функций. Очевидно, что при числе аргументов 0 все функции $f_1, ..., f_n$ равны 0. Пусть все функции $f_1, ..., f_n$ уже построены для переменных $x_1, ..., x_i$. Рассмотрим, как изменятся эти функции при добавлении еще одной переменной x_{i+1}.



а



б

Рис. 9.19. Построение булевой функции для решения проблемы SUBSET-SUM

Очевидно, что добавление переменной x_{i+1} к числу переменных вектора функций $f = \langle f_1, ..., f_n \rangle$ — это, фактически, учет возможности добавления очередного числа m_{i+1} к комбинациям подмножеств множества S: все векторы значений функции f на тех наборах аргументов, в которых $x_{i+1} = 1$, должны быть увеличены на двоичный код числа m_{i+1}. Пусть $\langle z_n, ..., z_1 \rangle$ — код числа m_{i+1}. Тогда для всех разрядов $j = 1, ..., n$ последовательно выполняем следующие вычисления:

$$f_j(x_1, ..., x_{i+1}) = \neg x_{i+1} \wedge f_j(x_1, ..., x_i) \vee f_j^*(x_1, ..., x_i),$$

где функция $f_j^*(x_1, ..., x_i)$ определяется следующим образом. Сначала вычисляется булева функция переноса в следующий разряд (очевидно, что перенос $C_1 = 0$):

$$C_{j+1}(x_1, ..., x_i) = \textbf{\textit{если}}\ z_j = 0, \textbf{\textit{то}}\ f_j(x_1, ..., x_i) \wedge C_j(x_1, ..., x_i)$$
$$\textbf{\textit{иначе}}\ f_j(x_1, ..., x_i) \vee C_j(x_1, ..., x_i)$$

Затем вычисляется эта функция $f_j^*(x_1, ..., x_i)$ — значения j-го разряда суммы двух слагаемых: двоичного кода числа m_{i+1} и полученного ранее кода суммы $\sum_{r=1}^{k}(x_r \times m_r)$:

$$f_j^*(x_1, ..., x_i) = \textbf{\textit{если}}\ \ z_j = 0\ \ \textbf{\textit{то}}\ \ f_j(x_1, ..., x_i) \oplus C_j(x_1, ..., x_i)$$
$$\textbf{\textit{иначе}}\ \neg\left(f_j(x_1, ..., x_i) \oplus C_j(x_1, ..., x_i)\right)$$

Проведенные эксперименты показали, что сложность построения такой функции растет полиномиально с ростом мощности множества S до нескольких сотен. Иными словами, переборная задача SUBSET_SUM с пространством поиска порядка 10^n при использовании BDD может быть решена за линейное время при росте значения n до ~100.

Пример 9.8

Классическим примером задач "в ограничениях" являются буквенно-арифметические головоломки. Например, возьмем шутливую фразу: РЕШИ ЕСЛИ СИЛЕН. Необходимо найти такую подстановку цифр вместо букв, при которой сумма двух первых чисел равна третьему. Разные буквы обозначают разные цифры, а старшие разряды чисел не равны нулю.

Хотя задача формулируется просто, решение ее не совсем тривиально: действуя "в лоб", простым перебором, для решения этой задачи нужно рассмотреть более 60 тысяч вариантов различных подстановок, поскольку необходимо найти подстановку одной из 10 различных цифр для каждой из 7 различных букв.

Для формализации ограничений задачи расположим слова заданной в условии фразы в форме для сложения "в столбик" с явной записью переносов, которые обозначим c_i (от англ. *carry* — перенос):

```
 c4   c3   c2   c1          // ci — перенос из i-го столбца
      P    E    Ш    И
  +   E    C    Л    И
-----------------------------------------
 C    И    Л    E    H
```

Ограничения задачи можно записать в виде линейных целочисленных уравнений, связывающих буквы из одного и того же столбца с учетом переносов:

$$c_4 = C;$$
$$P + E + c_3 = И + 10 * c_4;$$
$$E + C + c_2 = Л + 10 * c_3;$$
$$Ш + Л + c_1 = E + 10 * c_2;$$
$$И + И + 0 = H + 10 * c_1.$$

Дополнительно следует формализовать следующее ограничение: значения всех указанных выше семи букв должны быть различными.

Поскольку свои возможные значения буквы принимают из конечного множества 10 цифр, можно каждую букву закодировать четырьмя двоичными переменными. Каждый из четырех переносов c_i является двоичной переменной. Таким образом, всего для решения задачи нужно 32 двоичных переменных. Конъюнкция всех ограничений, записанных как логические функции от этих переменных, даст функцию F, принимающую единичные значения на тех наборах двоичных переменных, которые являются решениями задачи. Если функция F невыполнима, решений у задачи нет.

Опыт показывает, что манипулирование двоичными функциями, представленными в форме BDD, не вызывает затруднений, даже если эти функции зависят от нескольких десятков и даже сотен двоичных переменных. Необхо-

димые операции с функциями, представленными в BDD, включены во все указанные выше библиотеки.

9.7. Использование BDD в анализе и синтезе логических схем

Современные интегральные схемы содержат многие миллионы транзисторов, и их проверка, тестирование и анализ представляют чрезвычайно трудную проблему. Если для таких схем построены BDD реализуемых схемами функций, задачи эти существенно упрощаются. В настоящее время BDD являются основными структурами данных, используемыми в САПР СБИС, манипуляции с BDD включены во многие промышленные системы автоматизированного проектирования интегральных схем. Некоторые авторы монографий по разработке интегральных схем пишут, что "*BDD — это структура данных в области автоматизированной разработки ИС, которая применяется также и в других областях*" — настолько широко и интенсивно используется эта форма представления булевых функций в автоматизированном проектировании схем.

Поскольку BDD задают булевы функции, реализуемые логическими схемами, их можно считать представлением логических схем. Если схема реализует несколько логических функций, имеющих общие подсхемы, то многовыходную схему можно представить BDD с несколькими головными вершинами, каждая из которых представляет свою схему. Такие BDD называются разделяемыми (shared). Рис. 9.20 представляет пример разделяемой BDD с двумя двоичными входами и тремя выходами. Фактически, разделяемая BDD представляет совместную минимизацию нескольких булевых функций в базисе $\{0, 1, \textbf{\textit{ite}}\}$ при заданном порядке переменных.

Некоторые области применения BDD в САПР СБИС следующие: логический синтез, верификация, генерация тестов, моделирование отказов (fault simulation).

Логический синтез — реализация схемы по BDD. Самый простой и прямой метод реализации логической схемы по представлению соответствующей функции в BDD — это использование мультиплексора (MUX) — схемы, имеющей несколько входов и один выход. MUX реализует функцию $\textbf{\textit{ite}}(x, y, z) = \textbf{\textit{if}}\ x\ \textbf{\textit{then}}\ y\ \textbf{\textit{else}}\ z$ от трех двоичных переменных. Рис. 9.21 показывает технику такой реализации. Из этого примера видно, что графическое представление BDD однозначно соответствует топологии сети мульти-

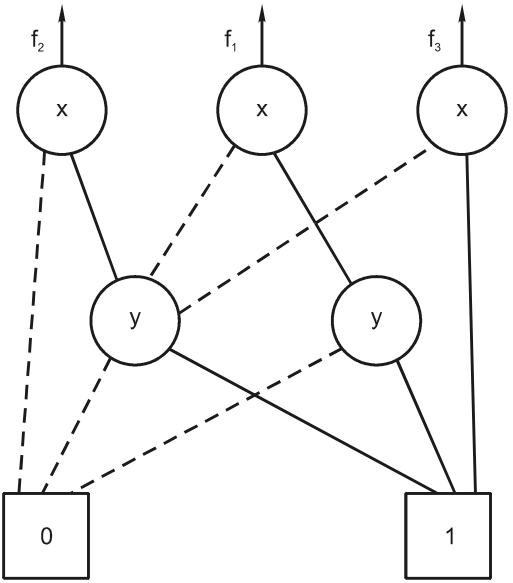
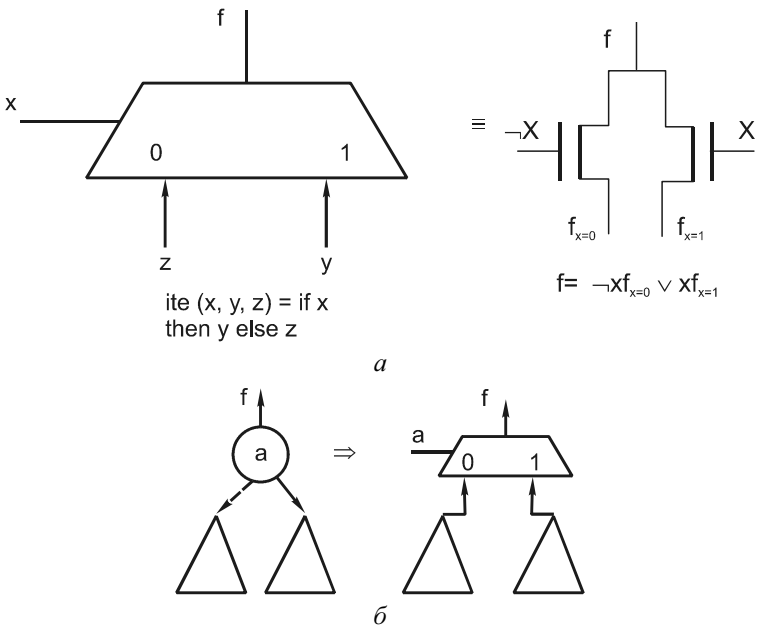


Рис. 9.20. BDD трех булевых функций: $f_1 = x \oplus y$, $f_2 = x \wedge \neg y$, $f_3 = x \vee y$



а

б

Рис. 9.21. Непосредственная реализация схемы по BDD:
а) элемент MUX, реализующий функцию $\boldsymbol{ite}(x, y, z)$;
б) соответствие фрагмента BDD и схемы MUX

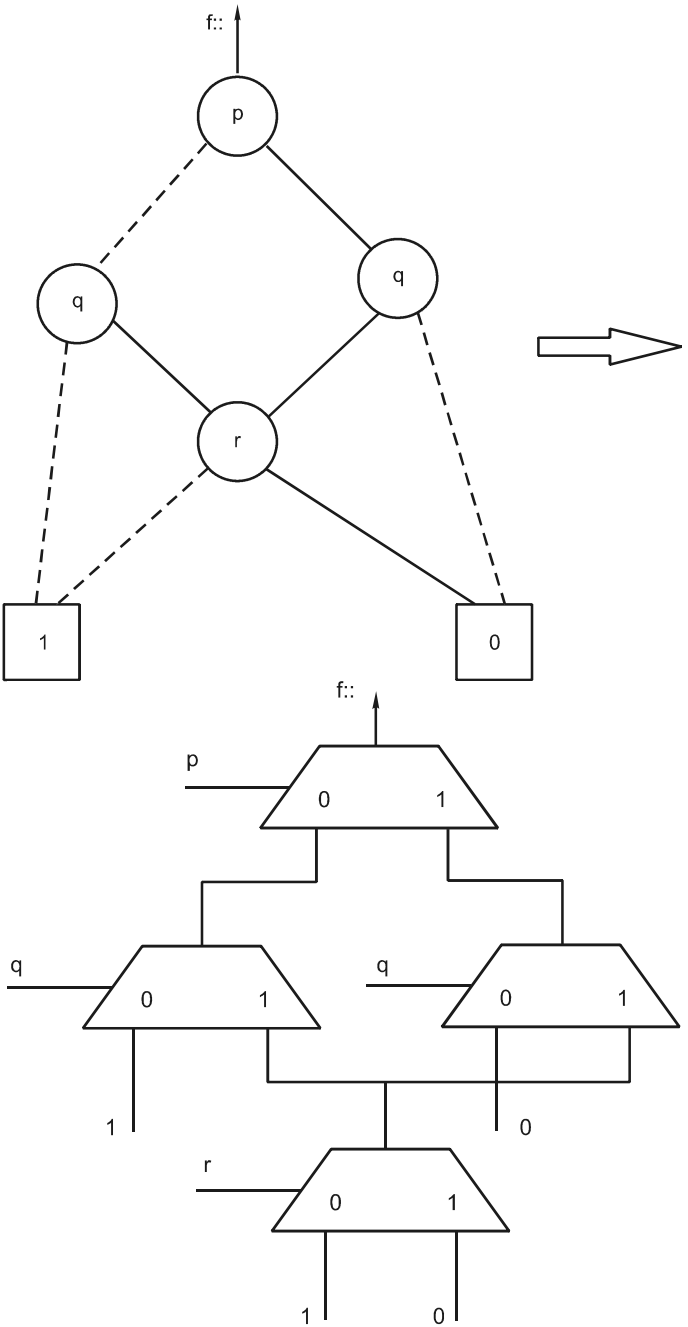


Рис. 9.21. Непосредственная реализация схемы по BDD:
в) реализация схемы по BDD

плексоров, реализующей функцию. В настоящее время это направление синтеза схем интенсивно развивается, в частности, разрабатывается технология PTL (Pass Transistor Logic) построения интегральных схем на основе MUX. Развивается также так называемое направление *BDD-Based Synthesis* в комбинации PTL с CMOS-логикой.

Анализ схемы — построение BDD по логической схеме. Построение BDD-функции, реализуемой логической схемой, осуществляется по структуре этой схемы очень просто. На рис. 9.22 это показано для схемы, реализующей функцию $z = (a \vee b) \wedge (a \vee c)$. Сначала строятся BDD всех входных переменных схемы. На рис. 9.22 это переменные a, b и c. Затем по ярусам, от входов к выходу строятся BDD всех промежуточных подфункций. На рис. 9.22 это функции $x = a \vee b$, $y = a \vee c$. Результирующая BDD представляет логическую функцию z, реализуемую схемой.

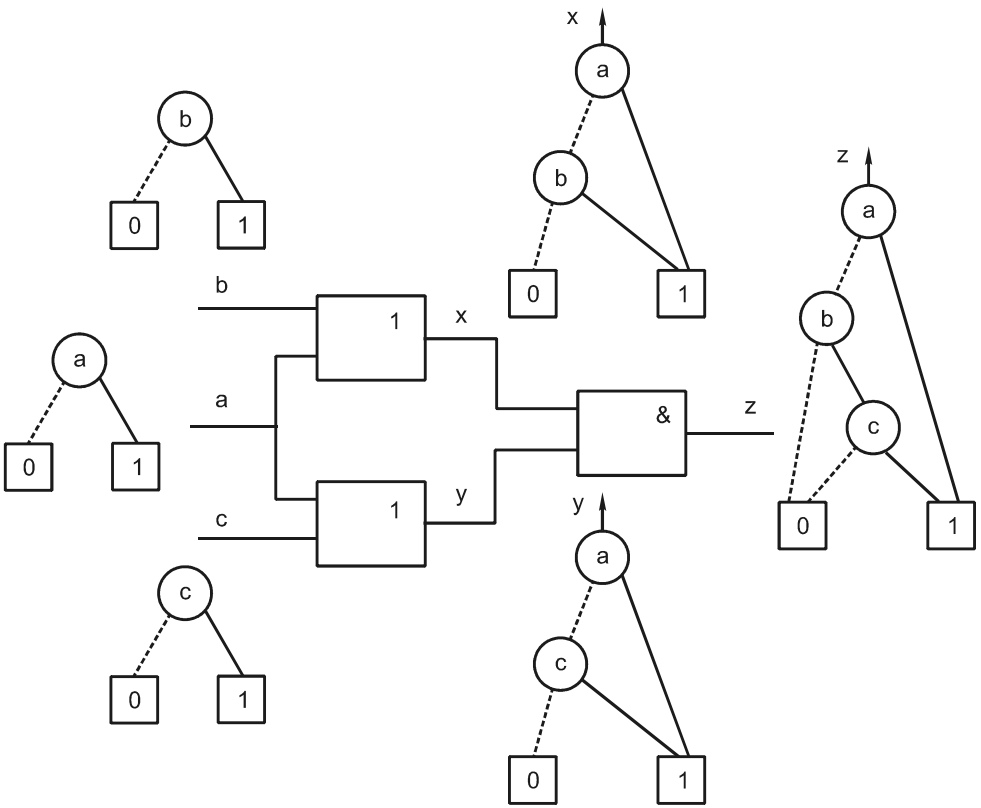


Рис. 9.22. Построение BDD по логической схеме

Верификация схемы: проверка того, удовлетворяет ли логическая схема ее спецификации. Спецификация логической схемы обычно задается булевой формулой или наборами значений аргументов, на которых функция должна быть равна 1. По этой спецификации может быть построена BDD соответствующей логической функции, а по схеме можно построить BDD той функции, которая реализуется этой схемой. Сравнение двух BDD — канонических представлений этих двух булевых функций — позволяет легко проверить, удовлетворяет ли схема ее спецификации.

Проверка эквивалентности двух логических схем. Одну и ту же булеву функцию могут реализовывать различные логические схемы (они, конечно, должны иметь одинаковое число двоичных входов и одинаковое число двоичных выходов). Проверка эквивалентности логических схем может быть выполнена проверкой эквивалентности булевых функций, реализуемых этими схемами.

Другой способ — это построение функции F, которая равна 1 при несовпадении значений хотя бы на одном выходе этих двух схем (рис. 9.23). Эта проблема, фактически, проблема SAT, решается построением BDD для логической функции, реализующей выход схемы рис. 9.23, *б*, и проверкой того, существует ли у BDD этой функции терминальная вершина 1.

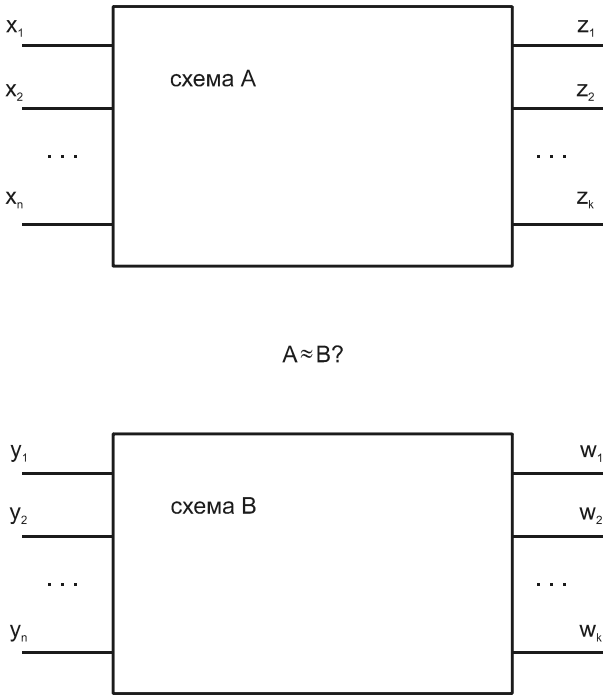


Рис. 9.23, *а*. Проверка эквивалентности двух логических схем

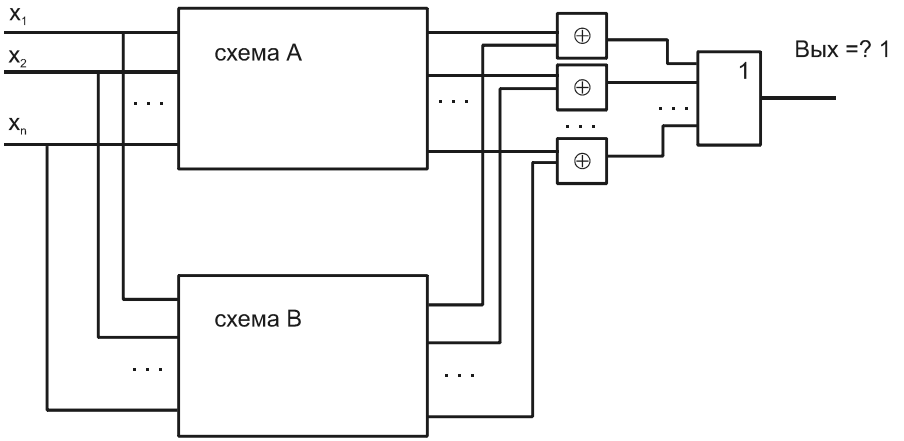


Рис. 9.23, *б*. Проверка эквивалентности двух логических схем

9.8. Конечные структуры данных и BDD

Как было показано выше, BDD успешно используются во многих областях, существенно расширяя границы возможного применения булевых функций к решению практических задач. Но наиболее широкое применение BDD получили в разработке эффективных алгоритмов для представления и оперирования конечными структурами данных. К таким структурам в математике относят множества и отношения.

Характеристическая функция множества. Пусть S — конечное множество из m элементов. Выберем натуральное $k = \lceil \log 2(m) \rceil$ и закодируем все элементы S векторами двоичных переменных $x_1, ..., x_k$ длиной k. Не уменьшая общности, можно считать, что мощность множества S является степенью 2.

Пример 9.9

Пусть S состоит из 8 элементов: $\{a_0, a_1, ..., a_7\}$. Закодируем элементы S двоичными кодами произвольным образом, например, так $(k = 3)$:

$a_0 \Leftrightarrow 000, a_1 \Leftrightarrow 001, ..., a_7 \Leftrightarrow 111$.

Пусть A — некоторое подмножество множества S, $A \subseteq S$, например, $A = \{a_0, a_1, a_2, a_6\}$. Набор двоичных кодов, соответствующих элементам из A, таков: $\{000, 001, 010, 110\}$.

Обозначим χ_A булеву функцию, которая равна 1 на наборах, кодирующих элементы из A. Для примера 9.6 функция χ_A равна 1 на наборах $\{000, 001, 010, 110\}$ и равна 0 на всех остальных наборах. Подобным образом любое подмножество конечного множества может быть задано с помощью булевой функции. Эта функция называется характеристической функцией множества.

❖ ***Определение 9.5 (характеристическая функция множества)***

Характеристической функцией множества A, подмножества некоторого конечного множества S, называется булева функция, принимающая значения 1 в точности на тех двоичных наборах, которыми закодированы элементы A при некотором фиксированном кодировании элементов множества S.

Характеристическую функцию множества, как и любую булеву функцию, можно представить в различных формах. Например, таблицей истинности, логической формулой или ее бинарной решающей диаграммой. Функция χ_A примера 9.9 может быть задана таблицей истинности рис. 9.1, *а*, или двоичным решающим деревом рис. 9.1, *б*, или любой логической формулой примера 9.3, или бинарной решающей диаграммой рис. 9.2.

Именно представление характеристических функций множеств в форме BDD оказалось очень эффективным как для хранения множеств, так и для выполнения операций над множествами.

Пусть x_1, x_2, x_3 — двоичные переменные, которые являются вспомогательными, вновь введенными символами. Тогда булева функция:

$$\chi_A = \neg x_1 \neg x_2 \neg x_3 \vee \neg x_1 \neg x_2 x_3 \vee \neg x_1 x_2 \neg x_3 \vee x_1 x_2 \neg x_3$$

задает множество $A = \{a_0, a_1, a_2, a_6\}$.

Итак, подмножества конечного множества можно задавать логической формулой, представляющей характеристическую функцию. Будем писать

$$A = \{000, 001, 010, 110\}(x_1, x_2, x_3),$$

чтобы показать закодированные элементы множества A, переменные кодировки и их порядок.

Операции над множествами. Все операции над конечными множествами могут быть определены через булевы операции над соответствующими характеристическими булевыми функциями. Пусть S — конечное множество, а P и Q — его подмножества. Рассмотрим, как операции над множествами P

и Q можно выполнить с помощью операций над характеристическими функциями этих множеств χ_P и χ_Q.

Нульарные операции (константы):

- полное множество S: $\chi_S = 1(True)$;
- пустое множество $\varnothing$: $\chi_\varnothing = 0(False)$.

Унарная операция:

- дополнение множества $Q \subseteq S$: $\chi_{S\setminus Q} = \neg\chi_Q$.

Бинарные операции:

- пересечение множеств $P \cap Q$: $\chi_{P\cap Q} = \chi_P \wedge \chi_Q$
- объединение множеств $P \cup Q$: $\chi_{P\cup Q} = \chi_P \vee \chi_Q$
- разность множеств $P \setminus Q$: $\chi_{P\setminus Q} = \chi_P \wedge \neg\chi_Q$

Таким образом, ВСЕ операции над множествами можно выразить через булевы операции над характеристическими булевыми функциями. Если характеристические булевы функции множеств представлены в форме BDD, то булевы операции над ними будут выполняться достаточно эффективно.

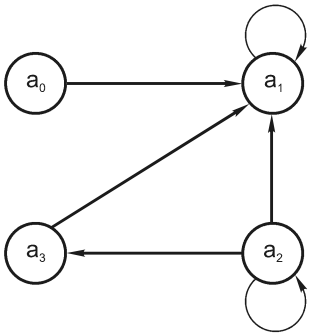
Представление отношений с помощью характеристических булевых функций

Пример 9.10

Пусть $S = \{a_0, ..., a_{n-1}\}$ — множество из n элементов. Пусть для определенности $n = 4$. Для двоичного кодирования элементов из S нужны два двоичных разряда, обозначим их x_1 и x_2. Выберем кодирование элементов S так: $a_0 \Leftrightarrow 00, a_1 \Leftrightarrow 01, ... a_3 \Leftrightarrow 11$. Любое подмножество S можно представить булевой функцией от двух переменных.

Бинарное отношение на S — это подмножество пар элементов S, например, $R = \{(a_2, a_3), (a_0, a_1), (a_1, a_1), (a_2, a_1), (a_2, a_2), (a_3, a_1)\}$. Такое бинарное отношение можно рассматривать как представление направленного графа, вершинами которого являются элементы из S, а направленные ребра соединяют соответствующие вершины. Пример такого графа представлен на рис. 9.24, *а*. Первый элемент каждой пары отношения R — это вершина, из которой идет ребро, второй элемент каждой пары бинарного отношения — вершина, в ко-

торую направлено ребро. Бинарное отношение можно представить матрицей инциденций, такая матрица для отношения R показана на рис. 9.24, *б*.



а

	a_0	a_1	a_2	a_3
a_0	0	1	0	0
a_1	0	1	0	0
a_2	0	1	1	1
a_3	0	1	0	0

б

$x_1'x_2'$ / x_1x_2	00	01	10	11
00	0	1	0	0
01	0	1	0	0
10	0	1	1	1
11	0	1	0	0

в

$x_1'x_2'$	00	01	10	11
	0	1	1	1

г

x_1x_2	
00	1
01	1
10	1
11	1

д

Рис. 9.24. *а*) Бинарное отношение R; *б*) матрица бинарного отношения *R*; *в*) таблица истинности функции χ_R, представленная матрицей; *г*) результат операции Forward Image над χ_R; *д*) результат операции Backward Image над χ_R

Введем χ_R — характеристическую булеву функцию бинарного отношения R. Эта булева функция для нашего примера определена на четырех двоичных переменных: она равна 1 на кодировках пар из R, т. е. на шести двоичных наборах

$$R = \{1011, 0001, 0101, 1001, 1010, 1101\},$$

соответствующих шести направленным ребрам графа.

Разряды кода первого элемента каждой пары бинарного отношения R обозначим v, например, (x_1, x_2). Разряды кода второго элемента каждой пары бинарного отношения обозначим v' — штрихованными переменными из v, например, (x_1', x_2'). Характеристическую функцию χ_R отношения R при кодировке (v, v') можно записать в СДНФ так:

$$\chi_R(v, v') = (x_1 \neg x_2 x_1' x_2') \vee (\neg x_1 \neg x_2 \neg x_1' x_2') \vee (\neg x_1 x_2 \neg x_1' x_2') \vee$$
$$(x_1 \neg x_2 \neg x_1' x_2') \vee (x_1 \neg x_2 x_1' \neg x_2') \vee (x_1 x_2 \neg x_1' x_2').$$

Эту булеву функцию можно задать, например, таблицей истинности в виде матрицы, в которой строки помечены кодами нештрихованных двоичных переменных, а столбцы — кодами штрихованных двоичных переменных. Такая матрица для примера отношения R представлена на рис. 9.24, *в*. Заметим, что эта матрица совпадает с матрицей инциденций отношения R.

Итак, бинарное отношение на конечных множествах может быть задано характеристической булевой функцией, которая может быть представлена в различных формах: логической формулой, таблицей истинности или же в форме BDD, которую удобно представлять с одним и тем же порядком нештрихованных и штрихованных переменных.

Операции над отношениями

Операции квантификации по переменной. Существует две операции квантификации: "$\exists$-квантификация" и "$\forall$-квантификация".

Пусть множество A, элементы которого закодированы так: $A = \{1011, 0001, 0101, 1001, 1010, 1110\}$, представлено функцией F_A от четырех булевых переменных (x_1, x_2, x_1', x_2'). Эта функция является характеристической функцией отношения R примера 9.10.

Операция квантификация функции F_A по переменной x_2 записывается так:

$$(\exists x_2)\, F_A(x_1, x_2, x_1', x_2')$$

и определяет следующее множество кодировок:

$$B = \{111, 001, 101, 110\}(x_1, x_1', x_2').$$

В этих кодировках разряд, по которому выполняется операция квантификации, просто выброшен. Квантификация булевой функции может быть определена через операцию дизъюнкции:

$$(\exists x)\, f(x, y) = f(0, y) \vee f(1, y).$$

Квантификация по нескольким переменным вычисляется последовательно в любом порядке. Например, операция

$$(\exists x_1, x_2)\, F_A(x_1, x_2, x_1', x_2') = (\exists x_1)(\exists x_2)\, F_A(x_1, x_2, x_1', x_2')$$

определяет следующее множество кодировок:

$$\{11, 01, 10\}(x_1', x_2').$$

Легко видеть, что это кодировки тех вершин, в которые входит хотя бы одно ребро отношения R примера 9.10.

Выполним теперь операцию $\exists$-квантификации по переменным x_1', x_2'. Операция

$$(\exists x_1', x_2')\, F_A(x_1, x_2, x_1', x_2')$$

определяет следующее множество кодировок:

$$\{10, 00, 01, 11\}(x_1, x_2).$$

Легко видеть, что это кодировки тех вершин, из которых выходит хотя бы одно ребро отношения R примера 9.10.

Таким образом, с помощью операций квантификации можно по характеристической булевой функции, представляющей отношение, получить характеристические функции как множества всех вторых, так и множества всех первых компонент бинарного отношения.

Операция "$\forall$-квантификации" определяется симметрично операции "$\exists$-квантификации":

$$(\forall x) f(x, y) = f(0, y) \wedge f(1, y).$$

Эта операция также полезна. Пусть с помощью бинарного отношения задан направленный граф. Если операцию "$\forall$-квантификации" применить ко всем переменным, кодирующим первые компоненты бинарного отношения, мы получим кодировки вершин графа, в которые направлены ребра из всех вершин. Применив операции "$\forall$-квантификации" ко всем переменным, кодирующим вторые элементы бинарного отношения, получим кодировки тех вершин, из которых ребра направлены во все вершины.

Пример 9.11

Для характеристической функции F_A, представляющей отношение R примера 9.10 (см. рис. 9.24):

$$(\forall x_1, x_2)\, F_A(x_1, x_2, x_1', x_2') = \neg x_1' x_2',$$

что можно интерпретировать так: из всех вершин графа есть ребро в вершину a_1.

$$(\forall x_1', x_2')\, F_A(x_1, x_2, x_1', x_2') = false,$$

что можно интерпретировать так: в графе нет вершин, из которых ребра направлены во все вершины.

Прямой образ бинарного отношения (Forward Image). Пусть характеристическая булева функция $\chi_R(v, v')$ определяет бинарное отношение R на множестве S. Характеристическая функция множества тех элементов из S, в которые направлено хотя бы одно ребро отношения R, задается операцией *Forward Image* (*Прямой образ*) отношения R. Она определяется как операция квантификации на характеристической функции отношения R: $(\exists v \chi_R)(v')$, т. е. это операция квантификации по всем переменным, кодирующим вершины, *из которых* выходят ребра отношения. Для нашего примера отношения R в результате этой операции получим характеристическую функцию множества $\{a_1, a_2, a_3\}$ — в эти элементы множества S входят ребра, направленные из каких-то элементов этого множества. Фактически, операция Forward Image просто выбрасывает из бинарного отношения все первые компоненты пар. Заметим, что в результате операции Forward Image, примененной к характеристической функции бинарного отношения с кодировкой (v, v'), получается характеристическая функция множества с кодировкой штрихованными переменными v' (см. рис. 9.24, *г*). Если необходимо получить характеристическую функцию с нештрихованными переменными, можно выполнить переименование переменных: (v / v') — подстановка нештрихованных переменных из v вместо штрихованных переменных из v'.

Обратный образ бинарного отношения (Backward Image). Характеристическая функция множества тех элементов из S, из которых направлено хотя бы одно ребро бинарного отношения R, задается операцией *Backward Image* (*Обратный образ*) отношения R. Эта операция определяется над характеристической функцией отношения R так: $(\exists v')\chi_R(v')$, т. е. это операция квантификации по всем переменным, кодирующим вершины, *в которые* входят ребра отношения. Для нашего примера отношения R в результате этой операции получим характеристическую функцию множества $\{a_0, a_1, a_2, a_3\}$ — из этих элементов S есть переходы в какие-нибудь элементы множества S. Фактически, операция Backward Image просто выбрасывает из бинарного отношения все вторые компоненты пар. Заметим, что в результате операции Backward Image, примененной к характеристической функции бинарного от-

ношения с кодированием (v, v'), получается характеристическая функция множества с кодированием v (см. рис. 9.24, *д*).

Если характеристическая булева функция бинарного отношения представлена в форме BDD, то операции квантификации могут быть выполнены довольно легко. Они включены во все библиотеки, работающие с бинарными решающими диаграммами.

Ограничения отношения на подмножестве. Пусть R — бинарное отношение на S, и A — подмножество S. Обозначим $\chi_R(v,v')$ и $\chi_A(v)$ — характеристические функции отношения R и множества A при соответствующих кодировках. Конъюнкция $\chi_A(v)\,\&\,\chi_R(v,v')$ определяет бинарное отношение $A|R$ — такое ограничение отношения R на множестве A, что все первые компоненты этого отношения принадлежат A. Конъюнкция $\chi_A(v')\,\&\,\chi_R(v,v')$ определяет бинарное отношение $R|A$ — такое ограничение отношения R на множестве A, что все вторые компоненты этого отношения принадлежат A.

Пример 9.12

Пусть бинарное отношение $R \subseteq S \times S$ задано направленным графом рис. 9.25, *а*. На рис. 9.25, *б* выделено множество $A \subseteq S$. Бинарное отношение $A|R$ — такое ограничение отношения R на множестве A, что первые компоненты этого отношения принадлежат A, представлено на рис. 9.25, *в*. Бинарное отношение $R|A$ — такое ограничение отношения R на множестве A, что вторые компоненты этого отношения принадлежат A, представлено на рис. 9.25, *г*.

Пусть задано конечное множество S и R — бинарное отношение на S, $R \subseteq S \times S$. Пусть A — подмножество S, $A \subseteq S$ (рис. 9.26). Рассмотрим, как решаются две задачи достижимости.

I) Поставим задачу найти множество B всех тех элементов S, которые *достижимы* за один шаг отношения R из элементов множества A (рис. 9.26, *а*). С использованием характеристических булевых функций $\chi_R(v,v')$ и $\chi_A(v)$ характеристическую функцию искомого множества B можно найти за две операции: конъюнкции и квантификации, а после этого выполнить переименование переменных. Формально это можно выразить следующим алгоритмом:

1. Определяем ограничение отношения R на тех первых элементах R из S, которые принадлежат A:

   $\chi_A(v)\,\&\,\chi_R(v,v')$

2. Строим множество $B = Forward\ Image(A, R)$, т. е. *Прямой образ* множества A относительно бинарного отношения R (выбрасываем все первые компоненты) и затем проводим переименование переменных:

$$\chi_B(v) = \chi_{ForwardImage(A,R)}(v) = (\exists v)\left[\chi_A(v) \,\&\, \chi_R(v, v')\right](v / v').$$

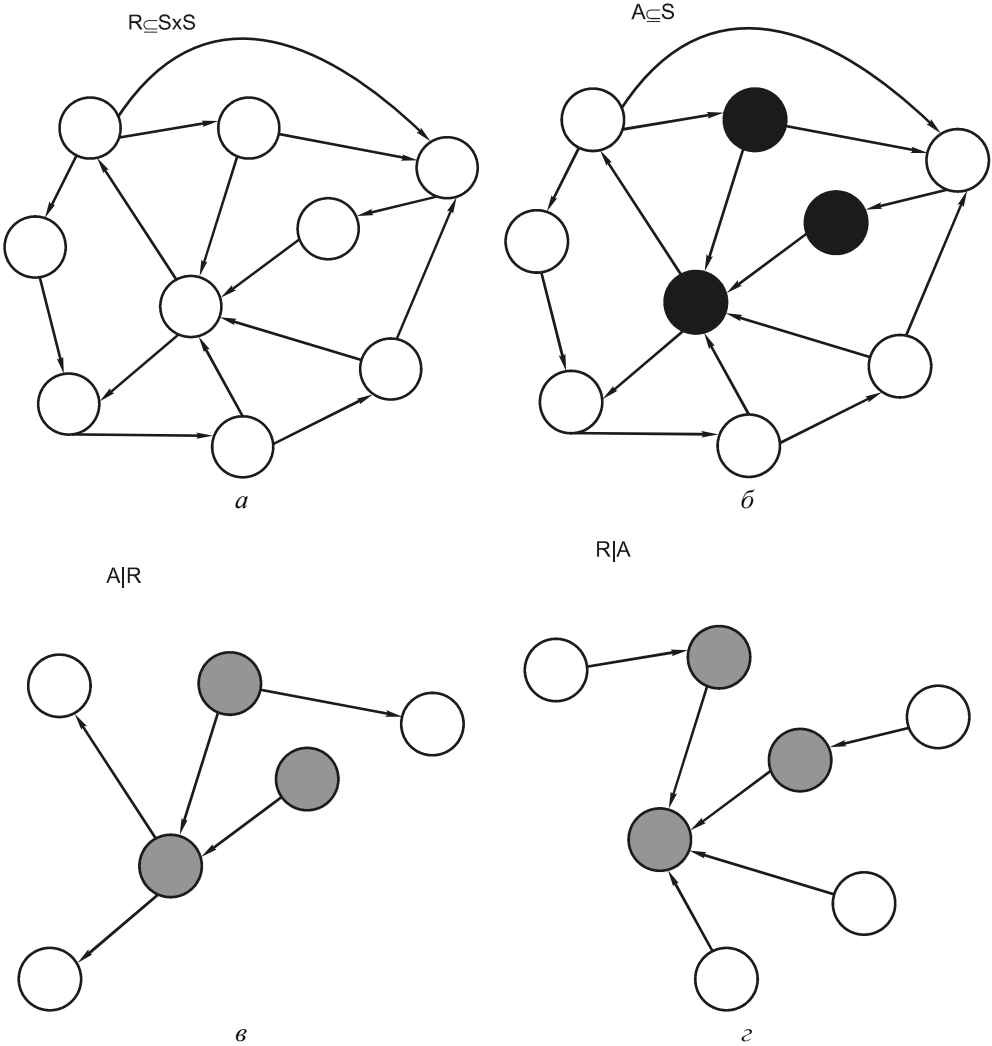
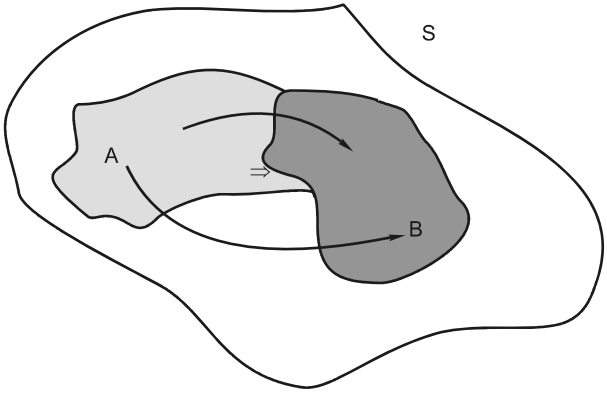
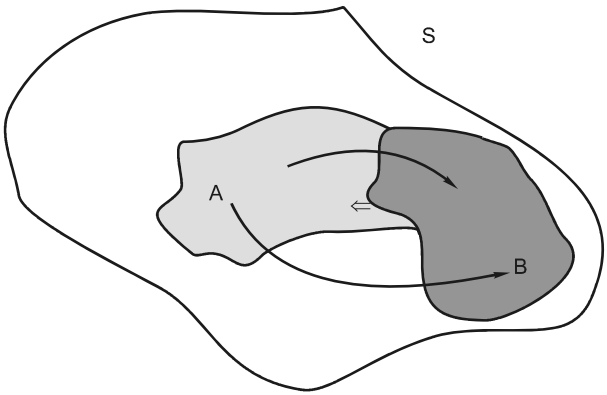


Рис. 9.25. *а*) Бинарное отношение R; *б*) множество A; *в*) бинарное отношение $A \mid R$; *г*) бинарное отношение $R \mid A$



а



б

Рис. 9.26. *а*) Множество элементов S, которые достижимы за один шаг отношения R из элементов заданного множества; *б*) множество элементов S, из которых достижимы за один шаг отношения R элементы заданного множества

II) Поставим задачу найти множество A всех тех элементов S, из которых достижимы за один шаг отношения R элементы заданного множества B (рис. 9.26, *б*). С использованием характеристических булевых функций $\chi_R(v,v')$ и $\chi_B(v)$ характеристическую функцию искомого множества A можно найти за две операции: конъюнкцию и квантификацию, но до этого нужно выполнить переименование переменных у характеристической функции $\chi_B(v)$ с v на v'. С помощью формул это можно выразить так:

1. Строим ограничение отношения R на тех вторых элементах R из S, которые принадлежат B:

   $\chi_B(v') \,\&\, \chi_R(v,v')$.

2. Строим $A = Backward\ Image(B, R)$, т. е. обратный образ B относительно R (выбрасываем все вторые элементы):

$$\chi_A(v) = \chi_{BackwardImage(B,R)}(v) = (\exists v')\left[\chi_B(v') \,\&\, \chi_R(v, v')\right](v).$$

Отношения общего вида. Отношения общего вида — это подмножества декартова произведения нескольких конечных множеств (не обязательно одинаковых), например $R \subseteq M \times N \times P$, где M, N и P — конечные множества. Представление такого отношения с помощью булевых функций очевидно: все элементы множеств M, N и P кодируются двоичными наборами, и если тройка (m, n, p) принадлежит R, то двоичному набору, соответствующему коду этой тройки, сопоставляем 1, в противном случае 0. Так, например, в форме BDD можно представить любые конечные отображения, поскольку отображение является частным случаем отношения. Примером сложной структуры данных, представимой в виде набора булевых функций (и, следовательно, в форме нескольких BDD), являются конечные автоматы. Пусть $A = (S, X, S_0, \delta, F)$ конечный автомат с множеством состояний S, входным алфавитом X, множеством начальных состояний S_0, функцией переходов δ и множеством заключительных состояний F. Этот конечный автомат можно представить в форме BDD тремя характеристическими булевыми функциями: характеристическими функциями множеств начальных и заключительных состояний χ_{S_0} и χ_F, а также характеристической функцией χ_δ, представляющей функцию переходов δ как конечное отношение $\delta \subset S \times X \times S$.

9.9. Символьные и неявные алгоритмы обработки данных

Основанные на BDD алгоритмы обработки данных называют *символьными* (symbolic) или *неявными* (implicit).

Почему *символьные*?

Для операций с объектами и отношениями вводятся булевы переменные, которые можно считать дополнительными параметрами, или *символами*.

Почему *неявные?*

Традиционные алгоритмы обычно оперируют *явным представлением* (explicit representation) дискретных объектов, перебирая их последовательно, один за другим. Алгоритмы, основанные на BDD, работают с *неявным* представлением (implicit representation) множеств объектов с помощью булевых функций, представленных в форме BDD. Одна операция на BDD, представляющей множество, заменяет все операции, которые необходимо выполнить последо-

вательно для каждого элемента множества. Поскольку размер BDD, представляющей множество объектов, с ростом числа элементов множества растет не линейно, а чаще всего логарифмически, операции с BDD выполняются очень эффективно. Два рассмотренных выше примера нахождения всех элементов множества, достижимых за один шаг из любых элементов другого множества, являются примерами неявных алгоритмов: поиск искомого множества производится с помощью только двух операций над BDD, независимо от числа элементов в представляемых BDD множествах.

Пример символьного алгоритма: анализ достижимости на графе

Пусть $\Gamma = (S, R, S_0)$ — направленный граф, где S — конечное множество, R — бинарное отношение на S, а S_0 — множество начальных вершин. Поставим задачу проверки того, достижима ли в Γ некоторая вершина $g \in S$ из какого-нибудь состояния множества S_0.

Построим символьный алгоритм анализа достижимости, работающий с характеристическими функциями множеств и отношений, представленных в BDD. Обозначим S_i — множество вершин, достижимых после i или меньшего числа шагов переходов из S_0. Очевидно, что для любого i, $S_{i+1} = S_i \cup \textit{Прямой образ}(S_i, R)$.

Алгоритм выглядит следующим образом.

1. Представляем множества S_0 и R в форме BDD, т. е. определяем множество v переменных, кодирующих вершины из S, а также характеристические функции $\chi_{S0}(v)$ и $\chi_R(v, v')$ в форме BDD. Строим также характеристическую функцию $\chi_{\{g\}}(v)$, представляющую вершину g.

2. В цикле по i полагаем $i = 0$, $S_i = S_0$, т. е.:

   $\chi_{S_i}(v) := \chi_{S_0}(v)$.

3. В теле цикла вычисляем ограничение R на S_i:

   $\chi_{S_i|R}(v, v') := \chi_{S_i}(v) \,\&\, \chi_R(v, v')$,

   и множество вершин графа Γ, в которые можем перейти из S_i за один шаг:

   $\chi_{S_{i+1}}(v) := \left[\chi_{S_i}(v) \vee (\exists v)\chi_{S_i|R}(v, v')\langle v / v'\rangle\right]$.

4. Цикл по i завершается, когда $\chi_{S_{i+1}}(v) = \chi_{S_i}(v)$.

5. Для проверки того, достижима ли вершина g из множества вершин S_0 в Γ, нужно просто проверить, будет ли выполнима характеристическая функция $\chi_{\{g\}}(v) \wedge \chi_{S_i}(v)$.

Поскольку алгоритм использует характеристические булевы функции для S_0 и R, а не работает с каждой отдельной вершиной и ребром графа Γ, то это — неявный, символьный алгоритм. Его сложность полиномиальна на множествах, растущих экспоненциально. На рис. 9.27 показана последова-

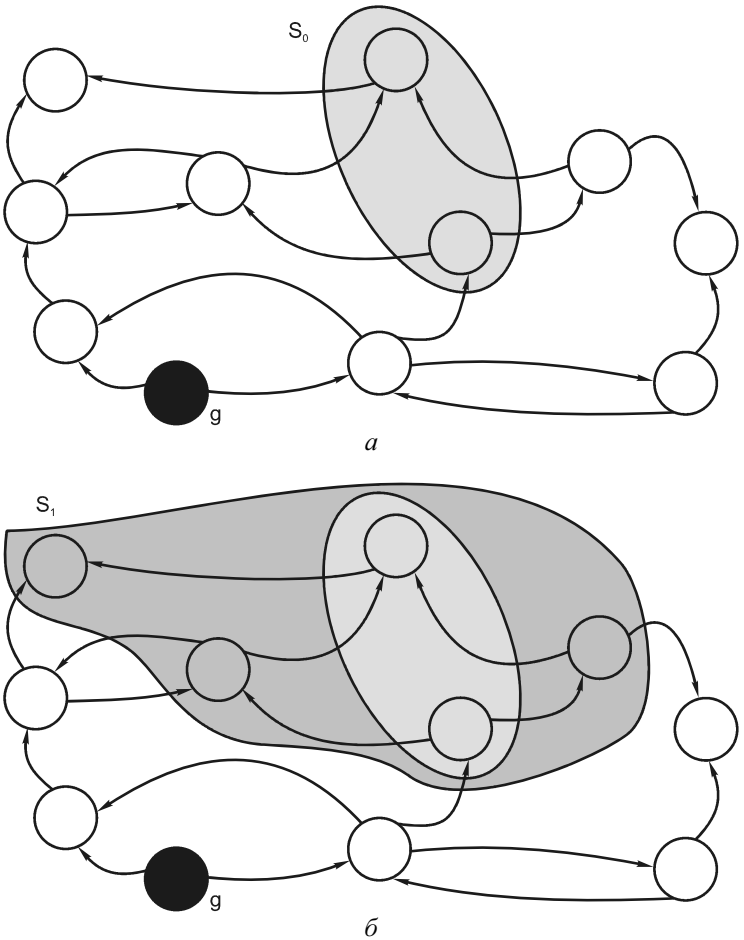


Рис. 9.27, *а* и *б*. Последовательность нахождения множеств вершин графа Г, достижимых из начального множества за 0, 1, 2, ... шагов

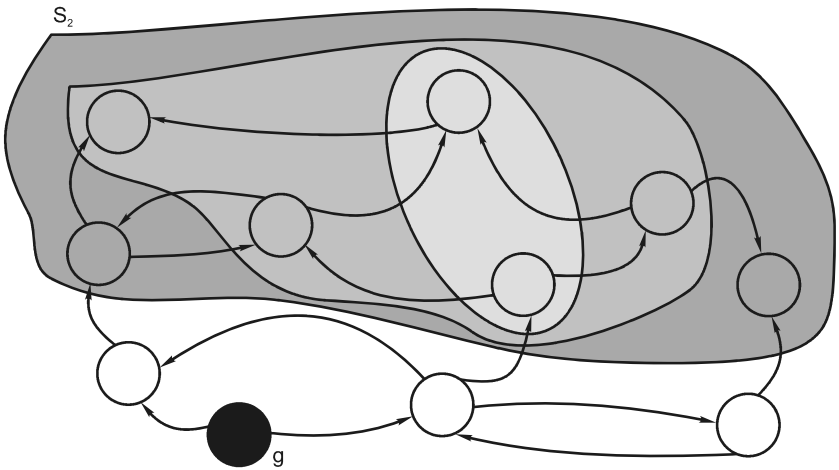


Рис. 9.27, *в*. Последовательность нахождения множеств вершин графа Г, достижимых из начального множества за 0, 1, 2, ... шагов

тельность множеств вершин, достижимых из множества S_0 за 1, 2 и т. п. шагов. Каждое множество S_i символьным алгоритмом строится за две операции над характеристическими функциями.

Таким образом, BDD можно рассматривать как каноническое компактное представление конечных множеств и отношений, которое допускает очень эффективную реализацию операций с этими данными.

9.10. Бинарные решающие диаграммы с подавлением нулей

Опыт применения BDD показал, что эта форма не всегда дает наиболее экономное представление данных для всех типов проблем. Поэтому были разработаны несколько модификаций BDD, которые в конкретных приложениях дают лучшее сжатие информации: MTBDD (Multi-Terminal BDD), FBDD (Free BDD) и др. Мы рассмотрим в этом разделе один из примеров такой модификации — бинарные решающие диаграммы с подавлением нулей (Zero-suppressed Binary Decision Diagrams, ZDD).

При работе с разреженными объектами BDD не является удобной и эффективной структурой для представления этих объектов. Множество объектов называется разреженным, если число элементов в нем много меньше того, что может в нем быть. В представлении BDD таких объектов большинство ребер указывает на терминальную вершину, помеченную 0.

Рассмотрим, например, представление множества подмножеств конечного множества. Определение 9.4 вводит характеристическую функцию множества $A \subseteq S$, как функцию, принимающую значения 1 на кодировках тех элементов из S, которые входят в A. Несколько подмножеств конечного множества при такой характеризации будут определяться несколькими булевыми функциями.

Определим другую характеризацию подмножества конечного множества так, чтобы одной двоичной функцией можно было представить сразу несколько небольших подмножеств большого конечного множества. Именно это позволяют сделать бинарные решающие диаграммы с подавлением нулей — они используются тогда, когда нужно экономно представить разреженные структуры данных. Диаграммы ZDD во многом похожи на BDD, но у них изменено одно правило редукции, что и позволяет этим модифицированным структурам быть более эффективными в приложениях, связанных с теорией множеств.

Бинарные решающие диаграммы с подавлением нулей (ZDD) строятся так. Для каждого элемента множества S введем свою двоичную переменную (двоичный разряд). Каждый двоичный код будет теперь кодировать некоторое подмножество множества S целиком, а именно, подмножеству $A \subseteq S$ ставится в соответствие следующий двоичный код. Если элемент $A \in S$ входит в A, то соответствующая переменная будет в этом коде равна 1, а переменная, соответствующая элементу, не входящему в A, будет входить в этот код со значением 0. Например, множеству подмножеств:

$$\{\{x_3, x_4\}, \{x_2, x_3, x_4\}, \{x_1, x_3, x_4\}, \{x_1, x_2\}, \{x_1, x_2, x_4\}, \{x_1, x_2, x_3\}, \{x_1, x_2, x_3, x_4\}\}$$

множества $\{x_1, x_2, x_3, x_4\}$ соответствует следующее множество двоичных наборов:

$$\{0011, 0111, 1011, 1100, 1101, 1110, 1111\}.$$

Будем считать перечисленные наборы единиц и нулей наборами, на которых некоторая булева функция F принимает значение 1. Эта функция имеет следующую формульную запись $F = x_1 x_2 \vee x_3 x_4$.

На рис. 9.28 показаны BDD и ZDD для функции $F = x_1 x_2 \vee x_3 x_4$. Хотя BDD для некоторых булевых функций более эффективны (содержат меньше вершин), чем ZDD, но эти две диаграммы имеют совершенно разную семантику. Из рисунка видно, что ZDD — это направленный граф, в котором в терминальную вершину 1 ведут семь путей, в точности соответствующих всем перечисленным выше семи наборам значений переменных $x_1, \ldots, x_4$.

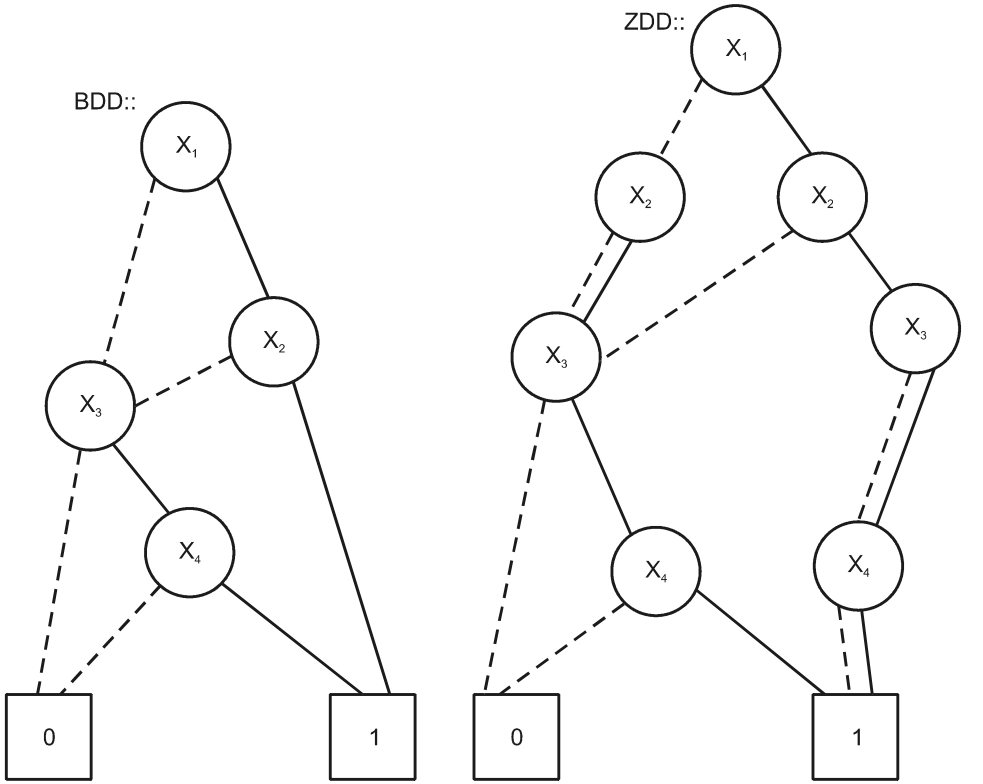


Рис. 9.28. BDD и ZDD для функции $F = x_1x_2 \vee x_3x_4$

Отличие семантик двух решающих диаграмм можно выразить так. В BDD на каждом пути из головной вершины в терминальную вершину 1 пропущенным переменным могут быть присвоены как 0, так и 1. В ZDD на каждом пути из головной вершины в терминальную вершину 1 пропущенным переменным может быть присвоено только значение 0. На рис. 9.28 по этому правилу из каждой диаграммы могут быть легко восстановлены те наборы значений переменных $\{1100, 1101, 1110, 1111, 0011, 0111, 1011\}$, на которых функция F равна 1.

BDD строятся из двоичного решающего дерева булевой функции в соответствии с двумя правилами упрощения ориентированного графа:

- C1: *Правило слияния*. Объединить (слить) любые изоморфные подграфы;
- C2: *Правило удаления.* Удалить из графа любую вершину, у которой оба непосредственных преемника совпадают, а все входящие в эту вершину

ребра перенаправить в вершину, в которую направлены выходные ребра вершины v.

ZDD строятся также из двоичного решающего дерева булевой функции по двум правилам редукции: *Правилу слияния* и *Правилу удаления*, первое из которых совпадает с С1, а второе формулируется несколько по-другому:

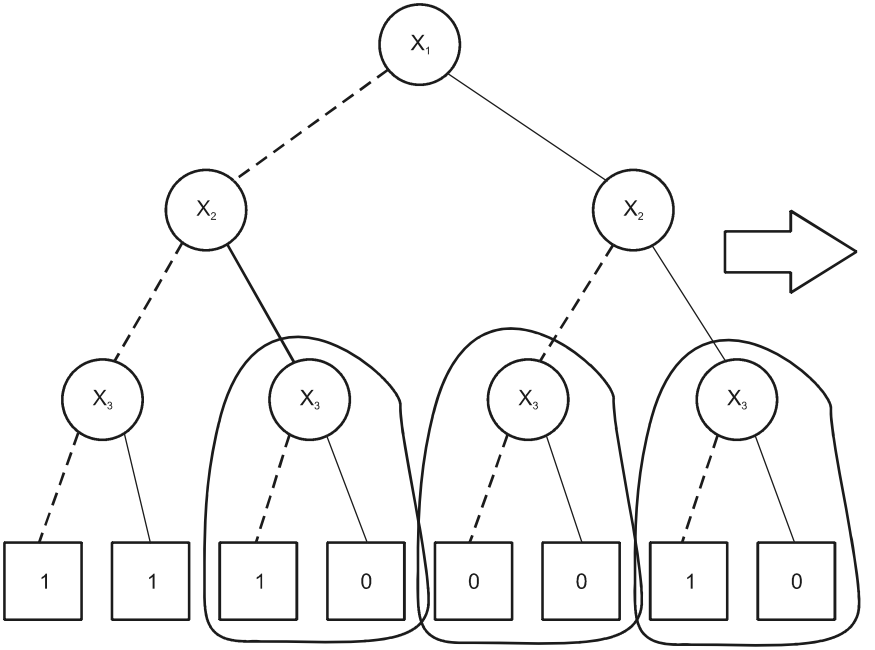
- CZ1: *Правило слияния*. Объединить (слить) любые изоморфные подграфы;
- CZ2: *Правило удаления*. Удалить из графа любую вершину v, у которой 1-ребро соединено с терминальной вершиной 0, а все входящие в эту вершину ребра перенаправить в вершину, в которую направлено выходное 0-ребро вершины v.

Таким образом, правила построения ZDD отличаются от правил построения BDD лишь в одном пункте.

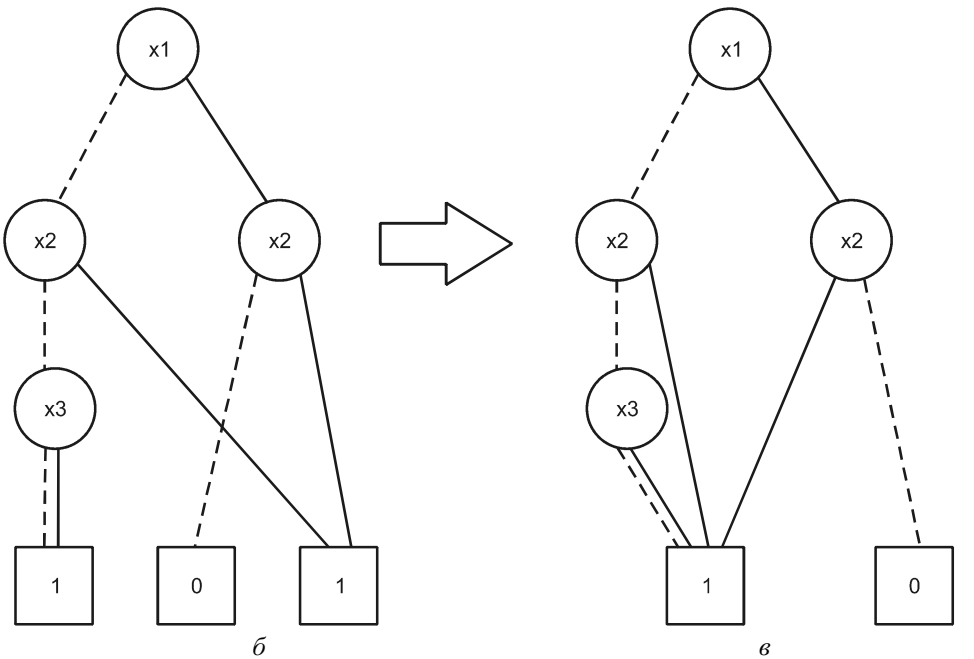
Пример 9.13

Рис. 9.1 представляет булеву функцию f от переменных x_1, x_2, x_3. Функция f равна 1 на наборах $\{000,\ 001,\ 010,\ 110\}$. Будем считать, что эта функция представляет следующие четыре подмножества множества $\{x_1, x_2, x_3\}$ из трех элементов: $\{\ \varnothing, \{x_3\}, \{x_2\}, \{x_1, x_2\}\ \}$. Для экономного представления этих четырех подмножеств построим ZDD этой функции. Рис. 9.29 показывает, как преобразуется в ZDD двоичное решающее дерево функции f. Диаграмма рис. 9.29, *в* показывает, что в терминальную вершину 1 идут ровно четыре пути: первый — $(x_1 = 0;\ x_2 = 0;\ x_3 = 0)$, второй — $(x_1 = 0;\ x_2 = 0;\ x_3 = 1)$, третий — $(x_1 = 0;\ x_2 = 1)$, четвертый — $(x_1 = 1;\ x_2 = 1)$. Эти четыре пути соответствуют четырем указанным выше подмножествам.

Оказывается, ZDD сохраняют два главных свойства BDD — каноничность (при определенном порядке переменных) и простоту выполнения булевых операций. Основная область применения ZDD — это комбинаторные задачи, в которых необходимы представление и работа с множествами подмножеств конечных множеств. Для представления множеств подмножеств при решении практических задач эта структура оказывается намного более компактной, чем представление их в BDD. Эффективность ZDD особенно велика в тех случаях, когда наборы булевых переменных, на которых булева функция равна 1, содержат значительно больше нулей, чем единиц (или, в терминах теории множеств, число подмножеств множеств из n элементов значительно меньше, чем 2^n, и каждое подмножество содержит небольшое число элементов).



а



б *в*

Рис. 9.29. Преобразование семантического дерева в ZDD

ZDD были использованы во многих областях, от минимизации булевых функций до алгоритмов на графах и представления регулярных выражений. В [88] представлены применения ZDD для решения комбинаторных проблем. В частности, Дональд Кнут утверждает, что любая проблема покрытия может быть решена построением ZDD для подходящей булевой функции. Во многих применениях алгоритмы, основанные на представлениях булевых функций в форме ZDD, оказываются эффективнее, чем алгоритмы, основанные на представлениях в форме BDD. Существуют эффективные процедуры, которые позволяют конвертировать BDD в ZDD и обратно. Свободно распространяющиеся библиотеки функций для работы с BDD обычно содержат и функции, позволяющие работать с ZDD.

9.11. Заключение

Бинарные решающие диаграммы (Binary Decision Diagrams, BDD) введены в теорию и практику информатики около 20 лет назад. Предлагаемая концепция проста: строить вычисление значения булевой функции как последовательность бинарных выборов на направленном ациклическом графе. BDD можно рассматривать как каноническую форму задания булевых функций, как, например, СДНФ, но обладающую многими удобными свойствами, в частности, компактным представлением и возможностью эффективного выполнения операций с ними.

Представление булевых функций в форме BDD можно сравнить с компрессией данных, но в отличие от других видов компрессии, с BDD все операции можно выполнять непосредственно, без предварительной декомпрессии операндов и последующей компрессии результата операции.

Бинарные решающие диаграммы рассматриваются как прорыв в области автоматизированного проектирования вследствие того, что они позволяют эффективно представить булевы функции от огромного числа переменных, встречающиеся при решении практических проблем. Приведенные выше примеры решения логических и комбинаторных проблем с помощью BDD демонстрируют удивительные возможности этого представления булевых функций.

Можно провести некоторую аналогию BDD с использованием позиционной числовой записи вместо унарного кода. Действительно, в унарной системе счисления количество N представляется N единицами, и операции над числами в ней более очевидны и понятны. Однако позиционная система счисления позволяет экономно представить огромные числа (такие представления растут логарифмически при росте чисел) и выполнять над ними операции

очень эффективно. Поэтому преимущества позиционной системы счисления очевидны, хотя, например, операция сложения для позиционной системы счисления намного сложнее, чем для унарной системы счисления.

Следует, однако, помнить, что использование BDD не всегда позволяет представить булеву функцию эффективно. Хотя известно, что BDD являются компактным представлением для большинства булевых функций, встречающихся на практике, нет гарантий того, что для вновь рассматриваемой булевой функции ее BDD будет линейна или полиномиальна относительно числа ее аргументов.

Сложность BDD зависит от порядка переменных, и хотя для большинства булевых функций представление в BDD линейно или полиномиально при любых порядках переменных, для некоторых функций число вершин в представляющих их BDD может существенно различаться при разных порядках переменных. Более того, найдены классы булевых функций, для которых представление в BDD экспоненциально для любых порядков переменных.

Одним из основных применений BDD является представление конечных структур данных на основе использования булевых характеристических функций. Это привело к новому классу алгоритмов: так называемых "*символьных*" или "*неявных*" алгоритмов. Одной из областей, в которой символьные алгоритмы используются наиболее широко и успешно, является автоматизированное проектирование больших интегральных схем. Другая область — верификация дискретной аппаратуры и программ. Этой теме посвящена следующая глава.

Некоторые исследователи называют эффект применения символьных алгоритмов революцией в обработке дискретной информации. В наше время работа с булевыми функциями, представленными в форме BDD, является фундаментальной техникой, применяемой во многих областях вычислительной науки.

9.12. Замечания

Бинарные решающие диаграммы как структуры данных для представления булевых функций были введены Рэнделом Брайантом (Randal E.Bryant) в работе [25]. По некоторым источникам, публикация [25] является самой цитируемой в информатике, что показывает огромный интерес исследователей к этой теме и широкое применение этой структуры данных. В работе [25] было показано, что при фиксированном порядке аргументов булевой функции BDD являются ее каноническим представлением. Брайант ссылается на рабо-

ту Lee [98] как на первую работу, в которой предлагается использовать структуру бинарных решающих диаграмм для представления булевых функций. Работы S. Ackers [1] и B. Moret [114] также обсуждают возможность графического представления булевых функций. Но именно введение в [25] двух дополнительных требований к бинарным диаграммам — редуцированности и упорядоченности — сделало эту форму широко используемой. Линейный алгоритм редукции был предложен в [138]. Сравнение SAT-решателей и подхода к решению проблемы выполнимости с помощью BDD обсуждается в [101] и [28].

Постановки задач в парадигме "Программирование в ограничениях" можно найти, например, в [167]. Проблема SUBSET-SUM исследуется в [97], там же доказывается, что эта проблема NP-полна. Использование BDD для решения задач программирования в ограничениях обсуждается в [134]. Компрессия изображений на основе BDD исследовалась многими авторами, например, [87], [126]. В работах [57], [147] представлены эффективные символьные алгоритмы обработки графов, основанные на BDD. В работе [42] проведено исследование эффективности представления в BDD разреженных случайно сгенерированных графов.

Использование BDD для анализа и синтеза логических схем уже стало общепринятым. По этой теме написано несколько обзоров и монографий, например, [110], [118], [119]. Существует несколько программных систем, использующих BDD для автоматизированного синтеза VLSI. Например, в университете Амхерста разработана система BDS (BDD-based Logic Synthesis system) [192]. Разработаны программные системы эффективного манипулирования отношениями на основе BDD для решения разнообразных задач, например, система CrocoPat [173].

В качестве вводного обзорного текста по BDD и их применению можно указать, например, [9]. Бинарные диаграммы с подавлением нулей ZDD были введены в рассмотрение Shin-ichi Minato в [109]. Их применение рассматривается во многих работах, в частности, для представления неполностью определенных булевых функций. Материалы работы [107], в которой подробно описываются области применения ZDD, использованы в *разделе 9.10*. Дональд Кнут в новом препринте [88] одного из разделов его очередного тома "Искусства программирования", выставленном в Интернете, представил глубокое исследование этой структуры данных. Им представлено множество теоретических результатов и применений как BDD, так и ZDD в области комбинаторики.

9.13. Задачи к главе 9

9.1. Для логических формул $f_1 = x \wedge \neg z$ и $f_2 = z \vee (x \Rightarrow y)$:

а) построить двоичные решающие деревья;

б) по этим деревьям построить BDD для каждой функции;

в) построить BDD функций $f_1 \oplus f_2$ и $f_1 \Rightarrow f_2$ по BDD для компонентных функций.

9.2. Постройте BDD для компаратора — функции сравнения двух битовых слов: $F = (x_1 \equiv y_1) \wedge (x_2 \equiv y_2) \wedge \ldots \wedge (x_n \equiv y_n)$ для n = 2 и 3 для двух разных порядков переменных $x1 < x2 < x3 < y1 < y2 < y3$ и $x1 < y1 < x2 < y2 < x3 < y3$. Заметьте, что при одном порядке переменных рост сложности BDD линеен, при другом — экспоненциален.

9.3. Минимизируйте функцию $f = (p,q,r) = \neg p \vee q \oplus rq(p \vee r)$ примера 9.1.

9.4. Постройте BDD из двоичного решающего дерева для двух функций: $f_1 = \neg x1 \vee x3$ и $f_2 = x2 \wedge x3$. Постройте BDD функции $f_1 \vee f_2$ по полученным BDD для каждой из этих функций.

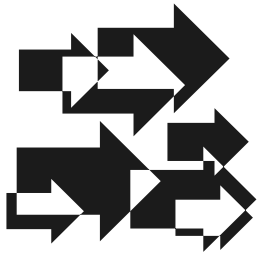
9.5. Представьте в форме BDD следующее отображение f конечного множества $X = \{a,b,c,d,e,g\}$ в конечное множество $Y = \{\alpha,\beta,\gamma,\delta\}$:

$$f(a) = \alpha, f(b) = \gamma, f(c) = \beta, f(d) = \delta, f(e) = \beta, f(g) = \alpha.$$

9.6. Представьте в форме BDD характеристические булевы функции для конечного автомата

$$A = (S, X, S_0, \delta, F) : S = \{s_0, s_1, s_2, s_3\},\ X = \{a,b\},\ S_0 = \{s_0, s_2\},$$
$$\delta = \{(s_0,a,s_0),(s_0,b,s_1),(s_1,a,s_0),(s_1,b,s_2),(s_2,a,s_3),(s_2,b,s_1),(s_3,a,s_2),(s_3,b,s_0)\},$$
$$F = \{s_3\}.$$

ГЛАВА 10



Символьная верификация

Успешность применения техники *model checking* для анализа больших и сложных систем в значительной степени обуславливается именно символьными алгоритмами верификации, которые рассматриваются в этой главе.

Алгоритмы проверки модели для логики CTL, рассмотренные в *гл. 3*, работают с явным представлением состояний структуры Крипке, и поэтому их часто называют "*явными*" алгоритмами (explicit state model checking — алгоритмы model checking с явным представлением состояний). Для каждой подформулы заданной темпоральной формулы эти алгоритмы вычисляют множество состояний структуры Крипке, на которых эта подформула выполняется. Символьные алгоритмы верификации фактически делают то же самое, но они работают с неявным представлением множеств состояний структуры Крипке булевыми функциями в форме бинарных решающих диаграмм. Символьную верификацию методом model checking часто определяют как *model checking+бинарные решающие диаграммы* (model checking+BDDs). Символьная верификация является одним из самых элегантных и самых удивительных применений BDD — этой формы представления двоичных функций, позволившей значительно повысить сложность анализируемых систем, что в конце концов привело к возможности применения верификации для анализа систем, разрабатываемых промышленностью.

Алгоритмы model checking выполняют преобразования множеств состояний структуры Крипке итеративно до тех пор, пока множество, к которому эти преобразования применяются, не перестанет изменяться. Такой объект, который является результатом многократного применения преобразования к аргументу до тех пор, пока аргумент не перестанет изменяться, называется *неподвижной точкой преобразования*. По сути, все алгоритмы model checking для CTL вычисляют неподвижные точки, представляются ли они символьно

или явно. Теория неподвижных точек преобразований дает теоретическое обоснование таких алгоритмов. Именно теория неподвижных точек операторов на конечных множествах составляет основной материал этой главы.

Другим вопросом, рассматриваемым здесь, является вопрос о том, как может быть построена для проверяемой сложной системы структура Крипке, содержащая астрономическое число состояний. Очевидно, что вручную строить такие структуры Крипке невозможно. Необходимы методы построения двоичных функций, представляющих структуру Крипке, непосредственно из текста программ.

10.1. Необходимость символьных методов в верификации

Классические алгоритмы верификации реагирующих систем, рассмотренные в *гл. 3*, являются *алгоритмами с явным представлением состояний* структуры Крипке. На современных компьютерах они могут использоваться для верификации систем, содержащих миллионы состояний. Представление каждого состояния требует обычно нескольких сотен байт, скорость обработки таких структур данных — порядка сотни состояний с секунду. Казалось бы, этого вполне достаточно для верификации систем. Однако реальные системы описываются моделями с огромным числом состояний. Хотя вычисления любой реальной системы в процессе своей жизни могут пройти лишь ничтожную долю возможных состояний, обычно заранее неизвестно, какие именно состояния и переходы необходимо проверять. Преимущество верификации в том и состоит, что она позволяет исчерпывающе проверить все возможные траектории поведения систем.

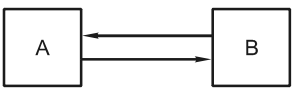


Рис. 10.1. Пример простой системы параллельных процессов

Рассмотрим пример простой системы (рис. 10.1). Система состоит из двух процессов, каждый из которых пусть имеет четыре управляющих состояния (например, значения счетчика команд) и оперирует только одной целой переменной типа `short` (1 байт). Подсистемы связаны двумя каналами, которые могут передавать целые значения также типа `short` и могут это сообщение хранить в буфере (т. е. передавать не мгновенно). Число возможных состояний этой простой системы равно $\left(4\times 2^8\right)^2 \times \left(2^8\right)^2 = 2^{36} \approx 10^{10}$, т. е. составляет десятки миллиардов! Хотя эта система очень проста, обычные методы вери-

фикации не могут быть использованы для ее проверки: число ее состояний на несколько порядков превышает то, с чем может оперировать классический явный алгоритм model checking.

При увеличении числа компонентов параллельной системы и/или увеличении числа переменных в системе происходит экспоненциальный рост числа состояний. Это явление известно под названием *проблемы взрыва числа состояний* (state explosion problem). Можно заключить, что при всей эффективности стандартных алгоритмов верификации model checking они могут быть применены только к очень простым системам или к очень абстрактным упрощенным моделям параллельных систем. Поэтому символьные алгоритмы обработки дискретной информации, рассмотренные в *гл. 9*, оказались востребованными для применения методов верификации model checking к моделям реальных, разрабатываемых на практике систем.

Символьная верификация вместо того, чтобы работать с явным представлением состояний и переходов структуры Крипке, работает с их неявным (implicit) представлением булевыми функциями в форме BDD. Этот подход предлагает решение проблемы взрыва числа состояний, поскольку символьные алгоритмы работают не с отдельными состояниями системы, а с характеристическими булевыми функциями множеств состояний.

В названии одной из первых работ [14], в которой был разработан символьный алгоритм верификации, указывается астрономическое число состояний верифицируемых этим методом систем: "*... 10^{20} и более ...*"! Явное представление подмножеств множества из 10^{20} состояний требует огромного объема памяти. Действительно, пусть для запоминания одного состояния нужно только 10 байт. Тогда запоминание 10^{20} состояний требует тысячу миллионов терабайт. Перебор такого числа состояний при выполнении алгоритмов верификации с помощью явных алгоритмов model checking потребует сотен миллиардов лет.

Рассмотрим неявное представление такого множества с помощью BDD. Любое подмножество множества из 10^{20} состояний может быть представлено булевой функцией от 70 переменных, поскольку $10^{20} = \sim 2^{70}$. Таким образом, представление структуры Крипке для системы с числом состояний 10^{20} требует:

- одну булеву функцию от ~70 двоичных переменных для представления множества начальных состояний;
- одну булеву функцию от ~140 двоичных переменных для представления множества переходов;
- для каждого атомарного предиката и каждой подформулы проверяемой темпоральной формулы одну булеву функцию от ~70 двоичных перемен-

ных, представляющую множество состояний, в которых эта формула истинна.

Если сложность BDD линейна или даже полиномиальна относительно числа переменных булевых функций, то представление в памяти этих функций и работа с ними не представляет трудности. И действительно, именно этот "символьный" подход к представлению состояний и алгоритмы работы с ними привел к возможности верификации промышленных программно-аппаратных систем.

Таким образом, использование BDD и неявных алгоритмов обработки дискретных данных революционно изменяет возможности алгоритмов верификации, основанные на методе model checking. Но для того, чтобы использовать преимущества технологии BDD в верификации, необходимо переформулировать проблему верификации в терминах манипуляций не с множествами состояний, а с булевыми функциями, представляющими такие множества. Первый шаг на этом пути — представление структуры Крипке булевыми функциями.

10.2. Представление структуры Крипке булевыми функциями

Для верификации структуры Крипке $M = (S,\ S_0,\ R,\ L)$ помощью *символьного* алгоритма model checking необходимо сначала построить булевы функции в форме BDD для:

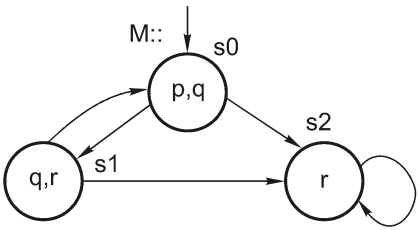
- множества начальных состояний M;
- множества переходов M;
- для каждого атомарного предиката — множеств тех состояний M, в которых он истинен.

Пример 10.1

Рассмотрим, как можно представить структуру Крипке булевыми функциями. Структура Крипке $M = (S,\ S_0,\ R,\ L)$ (рис. 10.2, *а*) имеет три состояния, $S = \{s_0,\ s_1,\ s_2\}$, множество S_0 начальных состояний $\{s_0\}$, пять переходов $R = \{(s_0,\ s_1),(s_0,\ s_2),(s_1,\ s_0),(s_1,\ s_2),(s_2,\ s_2)\}$ и три атомарных предиката $\{p,\ q,\ r\}$, такие, что $Sat_p = \{s_0\}$, $Sat_q = \{s_0, s_1\}$, $Sat_r = \{s_1, s_2\}$.

Выберем две булевы переменные $v = \langle x_1,\ x_2\rangle$ для кодирования состояний. Закодируем состояния так:

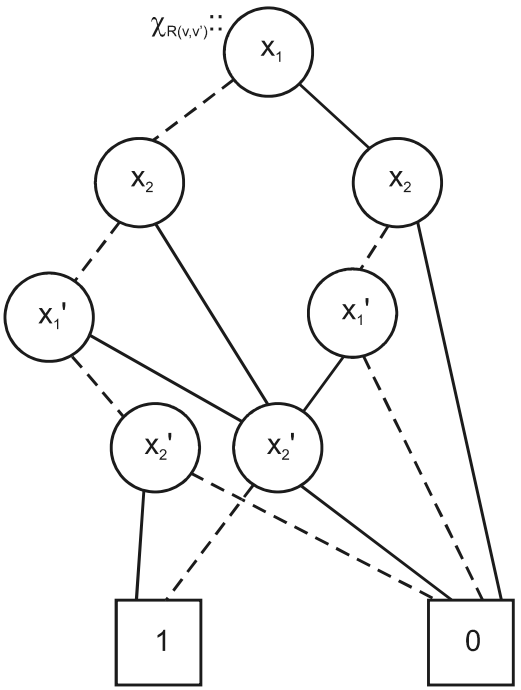
$$s_0 \Leftrightarrow 00,\ s_1 \Leftrightarrow 01,\ s_2 \Leftrightarrow 10\ (v).$$



а

x_1	x_2	x_1'	x_2'	$\chi_{R(v,v')}$
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

б



в

Рис. 10.2. Пример структуры Крипке:
а) характеристическая булева функция ее переходов;
б) таблица истинности характеристической функции переходов;
в) ее бинарная решающая диаграмма

Множество начальных состояний структуры Крипке содержит единственное состояние s_0. Его код 00, поэтому:

$$S_0 = \{s_0\} = \{00\}(v).$$

Для кодирования вторых элементов в парах, представляющих отношения, будем использовать штрихованные переменные $v' = \langle x_1', x_2' \rangle$. Отношение перехода представится так: $R = \{0001,\ 0010,\ 0100,\ 0110,\ 1010\}\ (v, v')$.

Множества состояний, в которых выполняются атомарные предикаты:

$$Sat_p = \{00\}(v)$$

$$Sat_q = \{00, 01\}(v)$$

$$Sat_r = \{01, 10\}(v)$$

Характеристические функции для S_0, R и Sat:

$$\chi_{s0} = \neg x_1 \neg x_2$$

$$\chi_{r(v,v')} = \neg x_1 \neg x_2 (x_1' \oplus x_2') \vee \neg x_1 x_2 \neg x_2' \vee x_1 \neg x_2 x_1' \neg x_2'$$

$$\chi_{Sat\ p}(v) = \neg x_1 \neg x_2$$

$$\chi_{Sat\ q}(v) = \neg x_1$$

$$\chi_{Sat\ r}(v) = x_1 \oplus x_2$$

Для выполнения символьной верификации все эти функции представляются в форме BDD. На рис. 10.2, *б*, *в* приведены таблица истинности и двоичная решающая диаграмма для характеристической булевой функции переходов структуры Крипке M.

Очевидно, что для анализа системы явно представлять в памяти ни множество всех состояний структуры Крипке, ни ее переходы не нужно, символьная верификация проводит анализ систем, используя только неявное представление множества начальных состояний и переходов структуры Крипке с помощью булевых функций.

10.3. Представление структуры Крипке реагирующих систем булевыми функциями без предварительного явного построения структуры Крипке

Приведенный выше пример показывает принципиальную возможность представления структуры Крипке булевыми функциями. Однако, поскольку обычно символьная верификация используется для верификации систем с огромным числом состояний, приведенный выше пример с ручным кодированием состояний и переходов структуры Крипке не имеет смысла: такие структуры Крипке просто невозможно явно представить.

Оказывается, что представление структур Крипке булевой функцией может быть выполнено и без предварительного явного построения структуры Крипке. Покажем это на простом примере.

Пример 10.2

Рассмотрим представление булевыми функциями структуры Крипке для решения логической головоломки о планировании переправы через реку "фермер, волк, коза и капуста", рассмотренной в *гл. 8*, без явного построения состояний структуры Крипке.

Четыре булевы переменные в этой проблеме f, w, g и c обозначают нахождение соответствующего объекта на правом берегу реки (значение переменной равно 1) или на левом берегу (значение переменной равно 0). Состояние структуры Крипке — вектор значений переменных $\langle f, w, g, c\rangle$. Например, состояние $\langle 0, 0, 1, 1\rangle$ означает, что фермер и волк находятся на левом берегу, а коза и капуста — на правом берегу.

Начальное состояние структуры Крипке — $\langle 0, 0, 0, 0\rangle$: все на левом берегу. Соответствующая характеристическая булева функция $\chi_{s0} = \neg f \neg w \neg g \neg c$.

Все возможные переходы для этой системы определяются четырьмя условиями переправы:

1. Либо только сам фермер может переправляться с одного берега на другой.

Либо фермер может перевезти с одного берега на другой:

2. только волка;

   или

3. только козу;

   или

4. только капусту.

Эти четыре правила легко задаются с помощью булевых функций, кодирующих соответствующие переходы в системе с помощью нештрихованных переменных (состояния до шага перехода) и штрихованных переменных (состояния после шага перехода). Например, первое правило говорит о том, что на каком бы берегу ни находился фермер до переправы, его положение после переправы будет другим $(f' \equiv \neg f)$, а все остальные объекты рассмотрения останутся на том же берегу, где и были: $(w' \equiv w) \wedge (g' \equiv g) \wedge (c' \equiv c)$. Обозначив $same(V)$ функцию, которая приравнивает все штрихованные и нештрихованные переменные из множества V, получим следующие характеристические функции для указанных выше правил:

$\chi_{R1}(v,v') = (f' \equiv \neg f) \wedge same(\{w,g,c\})$ // фермер переправляется сам

$\chi_{R2}(v,v') = (f' \equiv \neg f) \wedge (f \equiv w) \wedge (w' \equiv \neg w) \wedge same(\{g,c\})$ // везет только волка

$\chi_{R3}(v,v') = (f' \equiv \neg f) \wedge (f \equiv g) \wedge (g' \equiv \neg g) \wedge same(\{w,c\})$ // везет только козу

$\chi_{R4}(v,v') = (f' \equiv \neg f) \wedge (f \equiv c) \wedge (c' \equiv \neg c) \wedge same(\{w,g\})$ // везет только капусту

Все разрешенные правилами переходы возможны, поэтому булева функция, описывающая возможные переходы в этой системе, есть *дизъюнкция* всех четырех характеристических функций переходов:

$\chi_R(v,v') = \chi_{R1} \vee \chi_{R2} \vee \chi_{R3} \vee \chi_{R4}$

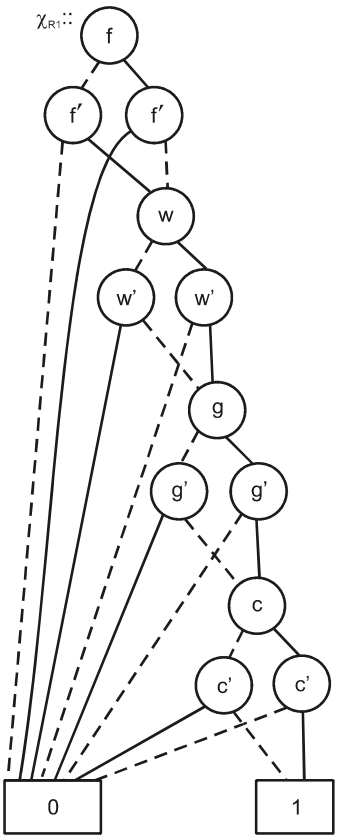


Рис. 10.3. BDD характеристической булевой функции множества переходов $\chi_{R1}(v,v') = (f' \equiv \neg f) \wedge same(\{w,g,c\})$ проблемы :"фермер-волк-коза-капуста"

Эта простая булева функция задает все 40 возможных переходов структуры Крипке, представляющей проблему. BDD характеристической функции переходов $\chi_{R1}(v,v')$ для первого правила представлена на рис. 10.3.

10.4. Представление булевыми функциями структуры Крипке для программ

Состояние программы — это мгновенный снимок, задающий множество значений всех переменных программы и значение счетчика команд pc. Пусть $V=\{v_1,\ v_2,\ v_3\}$ — множество программных переменных, принимающих значения в D_i, $v_i \in D_i$, а PC — множество всех значений программного счетчика. Состояние

$$s \in (PC,V),\ s=\langle pc \leftarrow m_5; v_1 \leftarrow 5; v_2 \leftarrow 3, v_3 \leftarrow 6\rangle$$

можно представить предикатом, т. е. функцией из некоторого множества (значений векторов программных переменных) в множество $\{true,\ false\}$:

$$(pc=m_5)\wedge(v_1=5)\wedge(v_2=3)\wedge(v_3=6).$$

Таким образом, состояние программы можно задать предикатом, истинным на том конкретном состоянии, на котором выполняются указанные соотношения. Предикатом можно задать не только одно состояние, но и множество состояний программы, например, предикат $P=(v_1>v_2)\wedge\neg(v_3=6)$ определяет множество всех таких состояний, на которых переменные v_1, v_2, v_3 удовлетворяют соотношению P. Состояние

$$s=\langle pc \leftarrow m_5; v_1 \leftarrow 5; v_2 \leftarrow 3, v_3 \leftarrow 4\rangle$$

принадлежит множеству состояний, которые удовлетворяют предикату P, а состояние:

$$s=\langle pc \leftarrow m_5; v_1 \leftarrow 3; v_2 \leftarrow 5, v_3 \leftarrow 4\rangle$$

не принадлежит этому множеству.

Операторы программы изменяют ее состояния. В структуре Крипке оператору соответствует переход — упорядоченная пара состояний $(s,\ s')$. Если нештрихованные переменные использовать для задания состояний, из которых направлен переход, а штрихованные переменные использовать для состояний, в которые направлен переход, то программу можно задать предикатами, каждый из которых соответствует своему оператору.

Пример 10.3

Рассмотрим оператор присваивания в программе:

```
...
m5: v1:=2*v1- v2;
m6: ...
...
```

Здесь мы не можем однозначно указать, какое состояние получится в результате выполнения оператора присваивания, поскольку мы не знаем начального значения переменных перед выполнением оператора. Однако мы можем точно сказать, что новое значение переменной v_1 (обозначаемое v_1') будет равно $2*v_1-v_2$ при любых начальных значениях этих переменных. Поэтому можно записать предикат, который описывает все те переходы, которые определяются оператором $v_1:=2*v_1-v_2$ с меткой m_5, если после него идет оператор с меткой m_6. Этот предикат имеет вид:

$$\left(\left(pc = m_5\right) \wedge \left(pc' = m_6\right) \wedge \left(v_1' = 2 * v_1 - v_2\right) \wedge same\left(\left\{v_2, v_3\right\}\right)\right)$$

Он задает множество переходов программы при произвольных начальных значениях переменных v_1, v_2, v_3. Здесь функция $same(V)$ определяется аналогично тому, как это было сделано выше:

$$same(V) \equiv \&_{v \in V}\left(v' = v\right)$$

Если множества D_i возможных значений программных переменных конечны, то каждый такой предикат может быть представлен булевой функцией. Эта функция может быть построена очень просто после того, как мы произвольным образом закодируем наборами двоичных переменных все возможные значения из D_i. Поскольку число операторов любой программы всегда конечно, то все значения программного счетчика также представляются различными двоичными наборами.

Таким образом, предикатом — формулой над множеством переменных из V, можно задать как множества состояний, так и множества переходов любой программы. Если области значений программных переменных конечны, то эти предикаты легко превратить в булевы функции, закодировав двоичными кодами все возможные значения каждой переменной.

В общем случае проблема представления программы с конечным числом состояний характеристическими булевыми функциями, задающими соответствующую программе структуру Крипке, — это проблема разработки компилятора, переводящего программу в формулы логики I порядка с последующим кодированием всех значений переменных программы наборами двоичных

переменных. Для этого по заданной последовательной программе нужно построить:

- формулу $S_0(V, PC)$ для множества начальных состояний;
- формулы $\Re = \Re(V, PC, V', PC')$ для всех возможных переходов в программе.

Рассмотрим алгоритм такой компиляции на примере простого языка. Будем считать, что программа состоит из операторов, каждый из которых, в соответствии с принципами структурного программирования, имеет один вход и один выход. Все возможные операторы языка определяются грамматикой:

$$P ::= \textbf{\textit{skip}} \mid v := E \mid P; P \mid \textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ P\ \textbf{\textit{else}}\ P\ \textbf{\textit{fi}} \mid \textbf{\textit{wile}}\ b\ \textbf{\textit{do}}\ P\ \textbf{\textit{od}}$$

Преобразование программы в формулу логики I порядка, производится в три шага.

Шаг 1: Разметка программы — трансляция ее в идентичный помеченный текст, в котором каждый оператор помечен меткой. Метки начала и конца всей программы считаем заданными.

При трансляции для любого оператора необходимо ввести только метку его начала. Метка конца оператора — это метка следующего оператора. Такая разметка задается атрибутной семантикой для каждого правила грамматики:

Оператор программы	**Результат трансляции — помеченный оператор**
skip	m: **skip**
v := E	m: v := E
P1; P2	m: P1; m1: P2
if b **then** P1 **else** P2 **fi**	m: **if** b **then** m1: P1 **else** m2: P2 **fi**
while b **do** P **od**	m: **while** b **do** m1: P **od**

В результате выполнения шага 1 перед каждым оператором и после каждого оператора программы появится метка.

Шаг 2: Построение булевой формулы, определяющей множество начальных состояний программы.

$S_0(V, PC) = Pre(V) \,\&\, (pc = m0)$, где $Pre(V)$ — предусловие, определяющее возможные начальные значения переменных, а $m0$ — метка начала программы. Если ограничений на возможные начальные значения не наложено, то $Pre(V) = true$.

Шаг 3: Построение предикатов для переходов. Каждый оператор программы определяет множество возможных переходов между состояниями. Эти переходы строятся для операторов программы так:

m: **skip**; m1:

$$\Re(m, skip, m1) \equiv$$
$$(pc = m) \wedge (pc' = m1) \wedge same(V)$$ // ничего не меняется, кроме pc

m: v := E; m1:

$$\Re(m, v := E, m1) \equiv$$
$$(pc = m) \wedge (pc' = m1) \wedge (v' = E) \wedge same(V \setminus \{v\})$$ // ничего не меняется, кроме pc и v

m: ***if*** b ***then*** m1: P1 ***else*** m2: P2 ***fi***; m3:

$$\Re(m, \textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ m1 : P1\ \textbf{\textit{else}}\ m2 : P2\ \textbf{\textit{fi}}, m3) \equiv$$
$$b \wedge (pc = m) \wedge (pc' = m1) \wedge same(V)$$ // если b истинно, идем на выполнение P1
$$\vee\ \neg b \wedge (pc = m) \wedge (pc' = m2) \wedge same(V)$$ // если b ложно, идем на выполнение P2
$$\vee\ \Re(m1, P1, m3)$$ // выполняем P1 и выходим
$$\vee\ \Re(m2, P2, m3)$$ // выполняем P2 и выходим

m: ***while*** b ***do*** m1: P ***od***; m2:

$$\Re(m, \textbf{\textit{while}}\ b\ \textbf{\textit{do}}\ m1 :\ P\ \textbf{\textit{od}}, m2) \equiv$$
$$b \wedge (pc = m) \wedge (pc' = m1) \wedge same(V)$$ // если b истинно, идем на выполнение P
$$\vee\ \neg b \wedge (pc = m) \wedge (pc' = m2) \wedge same(V)$$ // если b ложно, выходим
$$\vee\ \Re(m1, P, m2)$$ // выполняем P и выходим

Предикат, определяющий все возможные переходы всей программы, строится как дизъюнкция предикатов переходов для всех операторов программы:

$$\Re(Prog) = \Re_1 \vee \Re_2 \vee \ldots \vee \Re_n .$$

Параллельная программа образована множеством процессов (последовательных программ), которые исполняются параллельно.

$$P ::= \textbf{\textit{cobegin}}\ P_1 \,||\, P_2 \,||\, \ldots \,||\, P_n\ \textbf{\textit{coend}} .$$

Для асинхронных параллельных процессов к языку последовательных программ добавляется оператор параллельного запуска и операторы синхронизации. Параллельные программы выполняются асинхронно, взаимодействуя

через разделяемые переменные или посылкой сообщений. Если параллельные программы взаимодействуют через разделяемые переменные, то операторы в программах, работающие с разделяемыми переменными, должны быть атомарными.

Пример 10.4

Построим предикат, определяющий все переходы параллельной программы $A \| B$ с разделяемыми переменными, где:

```
A::                          B::
while true do                while true do
  pcA=0:  x:= 0;               pcB=0:  x:= 1;
  pcA=1:  y:= 1;               pcB=1:  y:= 0;
od                           od
```

Предикат этот строится так:

$$\Re(A \| B) \equiv$$

$$\big((pcA = 0) \wedge (pcA' = 1) \wedge (x' = 0) \wedge (y' = y) \wedge (pcB' = pcB)\big) \vee \quad \textit{//выполнение 1 оператора в A}$$

$$\big((pcA = 1) \wedge (pcA' = 0) \wedge (x' = x) \wedge (y' = 1) \wedge (pcB' = pcB)\big) \vee \quad \textit{//выполнение 2 оператора в A}$$

$$\big((pcB = 0) \wedge (pcB' = 1) \wedge (x' = 1) \wedge (y' = y) \wedge (pcA' = pcA)\big) \vee \quad \textit{//выполнение 1 оператора в B}$$

$$\big((pcB = 1) \wedge (pcB' = 0) \wedge (x' = x) \wedge (y' = 0) \wedge (pcA' = pcA)\big) \vee \quad \textit{//выполнение 2 оператора в B}$$

Рассмотрим, как задать характеристической булевой функцией множество переходов, соответствующее оператору синхронизации ***wait*** (b) в одном из параллельных процессов. Этот оператор имеет семантику периодической проверки разделяемой переменной b до тех пор, пока булева переменная b не будет установлена (другими процессами) в *true*:

$$\Re(m, \textbf{\textit{wait}}(b), m1) \equiv$$

$$\big(\neg b \wedge (pc = m) \wedge (pc' = m) \wedge same(V)\big) \quad \textit{если b не выполняется, то ждем}$$

$$\vee\big(b \wedge (pc = m) \wedge (pc' = m1) \wedge same(V)\big) \quad \textit{если b выполняется, переходим}$$

Характеристические булевы функции, представляющие начальное состояние, переходы и множества состояний программы, в которых выполняются атомарные предикаты, представляются в форме BDD. Они являются исходной информацией для проверки выполнения формул логики CTL на представленной таким образом структуре Крипке.

10.5. Символьный алгоритм model checking для CTL

Задача верификации методом model checking для CTL состоит в следующем: Даны структура Крипке M и темпоральная формула Φ логики ветвящегося времени (CTL). Найти множество S_Φ всех таких состояний структуры Крипке M, для которых выполняется Φ. Если начальное состояние принадлежит S_Φ, то Φ выполняется для M.

Символьный алгоритм model checking для каждой подформулы φ заданной формулы Φ логики CTL строит характеристическую булеву функцию f_φ, задающую то множество состояний Q_φ структуры Крипке M, на которых формула φ выполняется. Операции при этом выполняются не над множествами состояний, а над характеристическими булевыми функциями в форме BDD, представляющими эти множества состояний. Для разных подформул формулы CTL это делается по-своему. Типы формул CTL задаются их синтаксисом.

Синтаксис CTL:

$$\varphi ::= p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \mathbf{EX}\varphi \mid \mathbf{AX}\varphi \mid \mathbf{EF}\varphi \mid \mathbf{AF}\varphi \mid \mathbf{E}(\varphi_1 \mathbf{U} \varphi_2) \mid \mathbf{A}(\varphi_1 \mathbf{U} \varphi_2) \mid \mathbf{EG}\varphi \mid \mathbf{AG}\varphi$$

Рассмотрим, как для каждой CTL-формулы φ можно построить характеристическую функцию χ_{Q_φ} того множества состояний Q_φ структуры Крипке, на котором φ истинно. Рассмотрим их по порядку.

Формула p. p — это атомарный предикат. Для каждого атомарного предиката p функция χ_{Q_p}, задающая множество состояний, в которых p истинен, определяется при постановке задачи.

Формула $\neg\varphi$. Операция отрицания выполняется над булевой характеристической функцией χ_{Q_φ}, задающей то множество состояний структуры Крипке, на котором выполняется формула φ. Эта операция над BDD выполняется элементарно.

Формула $\varphi_1 \vee \varphi_2$. Для построения характеристической функции $\chi_{Q_{\varphi_1 \vee \varphi_2}}$ операция дизъюнкции выполняется над характеристическими булевыми функциями $\chi_{Q_{\varphi_1}}$ и $\chi_{Q_{\varphi_2}}$.

Формулы $\mathbf{EX}\varphi$, $\mathbf{AX}\varphi$. Поскольку $\mathbf{AX}\varphi = \neg\mathbf{EX}\neg\varphi$, то можно рассмотреть только формулу $\mathbf{EX}\varphi$. Смысл формулы $\mathbf{EX}\varphi$ такой: из данного состояния структуры Крипке существует переход в состояние, в котором выполняется формула φ. Пусть характеристическая булева функция множества состояний Q_φ структуры Крипке, на которых выполняется формула φ, уже дана. Для того чтобы построить характеристическую функцию множества тех состояний $Q_{\mathbf{EX}\varphi}$, на которых выполняется формула $\mathbf{EX}\varphi$, используем операцию "Обратный образ" над булевыми функциями, представляющими ограничение отношения R структуры Крипке на подмножестве Q_φ (рис. 10.4).

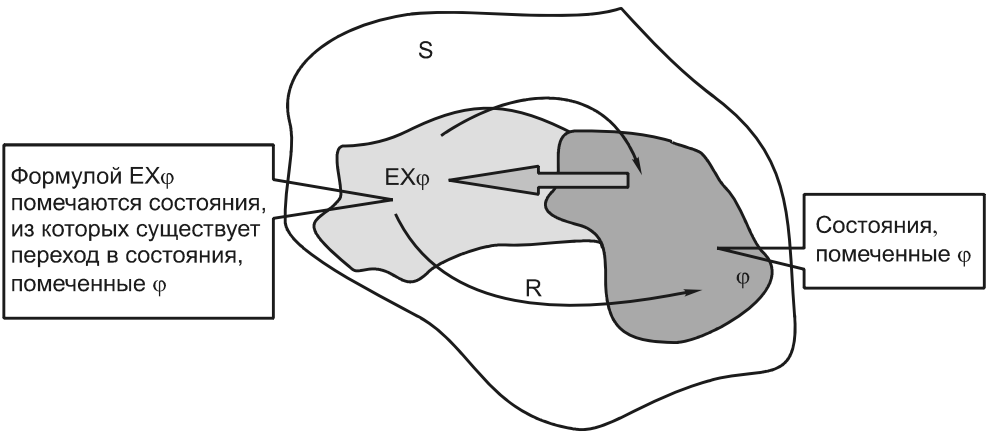


Рис. 10.4. Построение множества состояний, помеченных формулой EXφ, по отношению R и множеству состояний, помеченных формулой φ

1. Сначала у характеристической функции множества состояний Q_φ заменим переменные v кодировки на штрихованные v':

$$\chi_{Q_\varphi}(v') = \chi_{Q_\varphi}(v) < v'/v >$$

2. Затем построим характеристическую функцию, задающую такое ограничение отношения R, у которого вторые компоненты отношения находятся только во множестве Q_φ:

$$\chi_{(Q_\varphi|R)}(v,v') = \chi_{Q_\varphi}(v') \wedge \chi_R(v,v').$$

3. Над получившимся ограниченным отношением выполняем операцию "Обратный образ", получая характеристическую функцию, задающую первые компоненты построенного ограничения отношения R:

$$\chi_{Q_{\mathbf{EX}\varphi}}(v) = \chi_{BackwardImage(Q_\varphi|R)}(v) = (\exists v')\chi_{(Q_\varphi|R)(v,v')}(v).$$

Пример 10.5

Рассмотрим пример построения множества состояний структуры Крипке, удовлетворяющих CTL формуле $\mathbf{EX}\varphi$.

Дано: Q_φ и R.

Найти: $Q_{\mathbf{EX}\varphi}$.

На рис. 10.5, *а* представлен пример множества Q_φ и отношения R. Состояния, в которых выполняется формула φ, выделены. Как множество Q_φ, так и отношение R заданы характеристическими булевыми функциями и представлены в форме BDD.

На рис. 10.5, *б* показано ограничение отношения R на множестве Q_φ. Его составляют только такие переходы из R, у которых второй компонент принадлежит множеству Q_φ. Для того чтобы получить множество состояний $Q_{\mathbf{EX}\varphi}$, которые удовлетворяют формуле $\mathbf{EX}\varphi$, нужно у этих переходов взять первый компонент (рис. 10.5, *в*). Это делается с помощью операции квантификации характеристической функции по переменным второго компонента.

Таким образом, всего две операции с BDD, представляющими Q_φ и R, позволяют построить характеристическую функцию множества состояний $Q_{\mathbf{EX}\varphi}$.

Формула $\mathbf{EF}\varphi$ (состояние s удовлетворяет этой формуле, если из s существует путь, на котором когда-нибудь в будущем встретится состояние, на котором формула φ выполняется).

Дано: характеристические функции множества состояний Q_φ структуры Крипке и множества R переходов этой структуры.

Найти: характеристическую функцию множества состояний $Q_{\mathbf{EF}\varphi}$.

Это классическая задача достижимости: найти на системе переходов (с заданным отношением переходов R) то множество состояний, из которых достижимо некоторое заданное множество состояний (которые удовлетворяют условию φ).

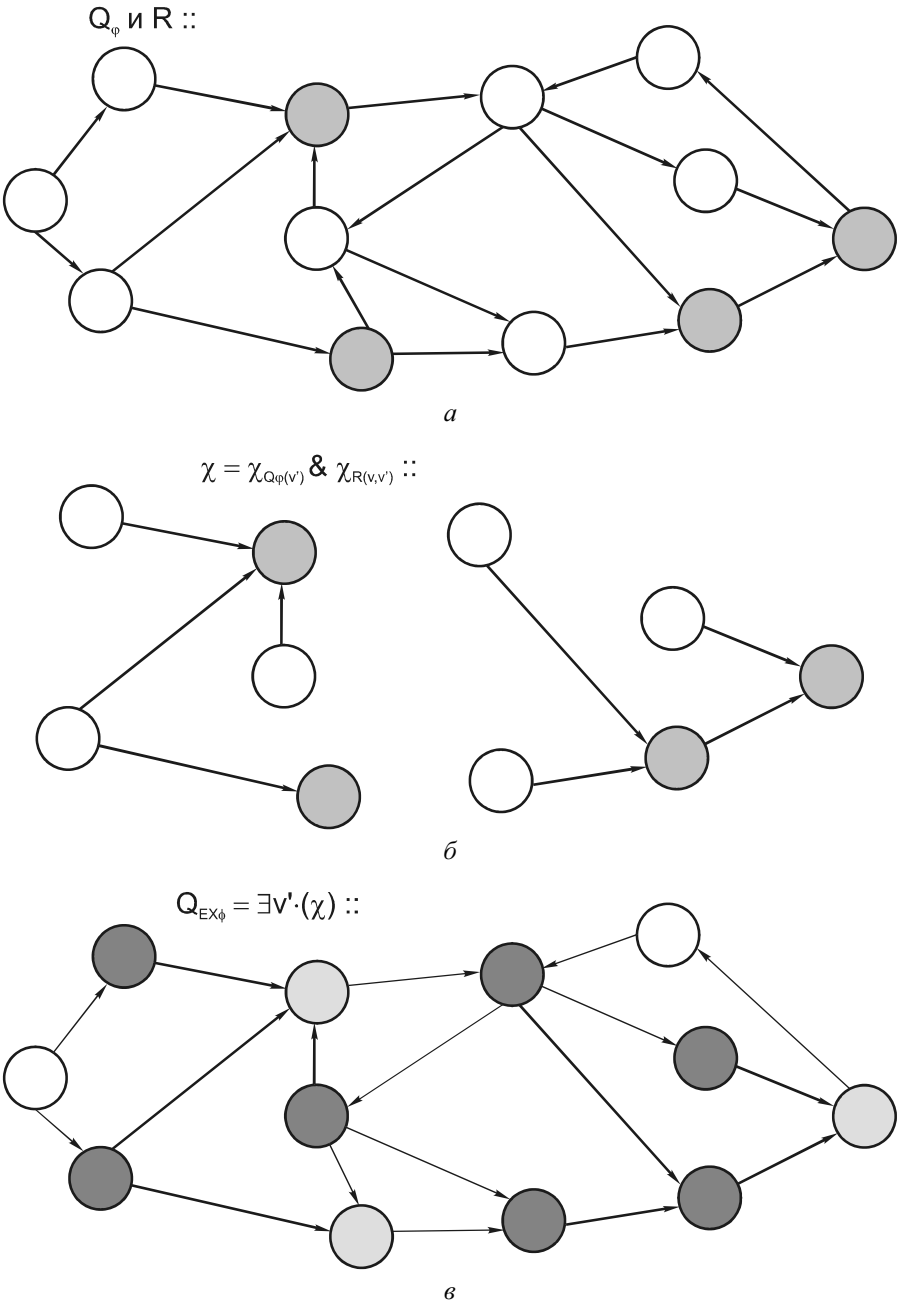


Рис. 10.5. *а*) Множество состояний Q_φ и отношение R;

б) ограничение отношения R на Q_φ; *в*) множество состояний $Q_{\mathbf{EX}\varphi}$ и отношение R

Явный алгоритм нахождения множества состояний $Q_{\mathbf{EF}\varphi}$ по заданным множеству состояний Q_φ и множеству переходов R рассматривался в *гл. 3*. Алгоритм нахождения множества таких состояний $Q_{\mathbf{EF}\varphi}$, из которых за произвольное конечное число шагов достижимы состояния из Q_φ, можно схематично представить так:

```
A:=Qφ;
do
  B:=A;
  A:=A∪Pre(A);  /* Pre(A) — это множество состояний, из которых
                   есть переход в состояния множества A) */
while (A ≠ B)
return A
```

Этот алгоритм основывается на рекурсивном определении формулы $\mathbf{EF}\varphi$:

$$\mathbf{EF}\varphi = \varphi \vee \mathbf{EX}\left(\mathbf{EF}\varphi\right),$$

откуда:

$$\mathbf{EF}\varphi = \varphi \vee \mathbf{EX}\varphi \vee \mathbf{EX}\left(\mathbf{EX}\varphi\right) \vee \mathbf{EX}\left(\mathbf{EX}\left(\mathbf{EX}\varphi\right)\right) \vee \ldots$$

и состоит в следующем. Сначала за искомое множество A состояний принимается множество Q_φ, т. е. те состояния, которые помечены формулой φ. Затем к множеству состояний A добавляются те состояния, из которых есть переход (за один шаг) в состояния из A. Такое преобразование делается многократно до тех пор, пока не получится множество A, применение преобразования к которому его не изменяет (рис. 10.6).

Аргумент, который не изменяется при применении к нему преобразования, называется *неподвижной точкой* этого преобразования. В данном алгоритме таким аргументом является полученное после нескольких шагов итераций множество состояний A, а преобразование, которое применяется к нему, это $Q_\varphi \cup Pre\left(A\right)$. Алгоритм останавливается, когда $A = Q_\varphi \cup Pre\left(A\right)$.

Все алгоритмы построения подмножеств состояний, удовлетворяющих оставшимся шести формулам $\mathbf{EF}\varphi$, $\mathbf{AF}\varphi$, $\mathbf{E}\left(\varphi_1 \mathbf{U} \varphi_2\right)$, $\mathbf{A}\left(\varphi_1 \mathbf{U} \varphi_2\right)$, $\mathbf{EG}\varphi$ и $\mathbf{AG}\varphi$ ищут именно неподвижные точки соответствующих преобразований.

Построение подобного неявного алгоритма, оперирующего не состояниями и переходами, а характеристическими булевыми функциями множеств состояний и множества переходов R, может быть выполнено по аналогии с алгоритмом, изложенным в *гл. 3*, и не представляет принципиальных трудностей.

Однако возникает вопрос: всегда ли алгоритм будет сходиться (т. е. всегда ли повторяющееся применение некоторого преобразования к подмножеству состояний структуры Крипке приведет к неподвижной точке), и будет ли полученное в результате множество состояний именно тем, которое нам нужно (все ли состояния, удовлетворяющие соответствующей формуле, войдут в это множество).

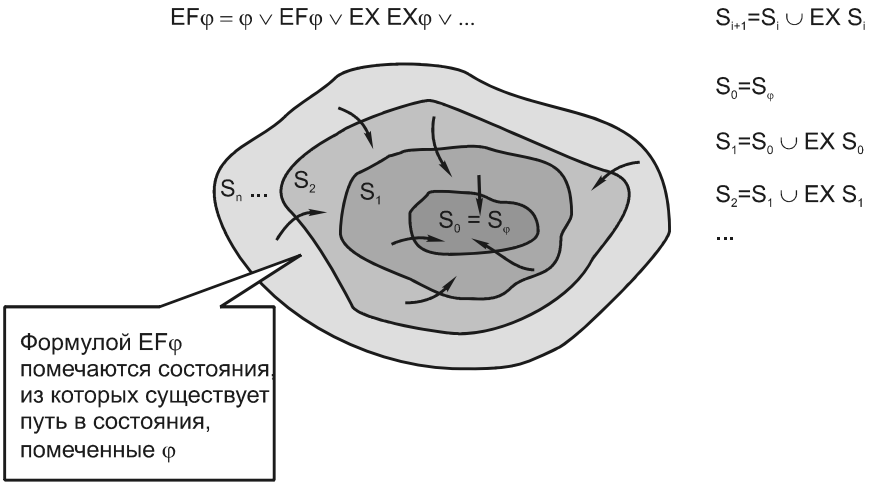


Рис. 10.6. Построение $Q_{\mathbf{EF}\varphi}$

Ответы на эти вопросы, иными словами, теоретическое обоснование применения неявных алгоритмов, которые состоят в повторяющемся применении преобразований к множествам, дает теория неподвижной точки операторов на решетках предикатов.

10.6. Операторы на множествах и неподвижные точки

Пусть S — конечное множество, 2^S — совокупность всех его подмножеств.

Подмножества конечного множества и диаграмма Хассе. Пусть $S=\{a, b, c, d\}$. Обозначим 2^S множество его подмножеств: $2^S = \{\varnothing, \{a\}, \{b\}, \{c\}, \{d\}, \{a,b\}, \ldots \{a,b,c,d\}\}$. Множество 2^S частично упорядочено, если считать, что отношением частичного порядка $\leq$ связаны два множества, находящиеся в отношении включения одного множества в другое. Некоторые пары элементов множества 2^S всех подмножеств S находят-

ся в отношении $\leq$ (например, $\{a\} \leq \{a,b\}$), другие — нет (например, $\{a,b\}$ и $\{a,c,d\}$ несравнимы). Таким образом, множество всех подмножеств любого множества частично упорядочено по включению.

Построим диаграмму Хассе — наглядное графическое представление конечного частично упорядоченного множества. В диаграмме Хассе каждому элементу сопоставляется точка плоскости таким образом, что меньший элемент всегда изображается ниже большего элемента. Два элемента x и y в диаграмме Хассе соединены ребром тогда и только тогда, когда $x \leq y$ и не существует такого элемента z, что $x \leq z \leq y$ (т. е. в диаграмме Хассе не отражена транзитивность отношения частичного порядка). Диаграмма Хассе множества 2^S для $S=\{a, b, c, d\}$ представлена на рис. 10.7.

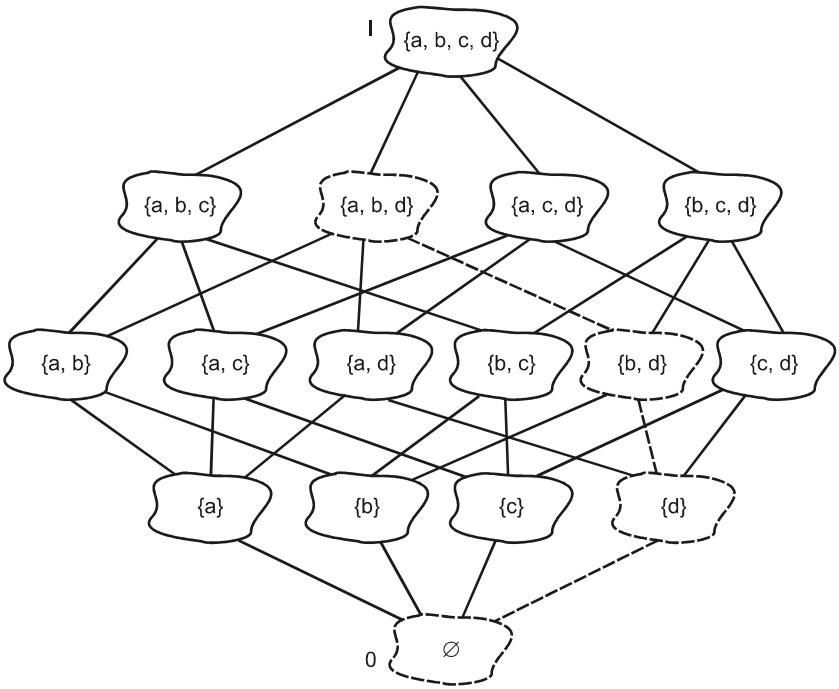


Рис. 10.7. Диаграмма Хассе подмножеств множества $\{a,b,c,d\}$

Операторы на множествах. Оператор τ на S в S — это отображение $\tau : 2^S \to 2^S$, т. е. оператор на S — это функция, которая переводит одно подмножество множества S в другое его подмножество:

$$\tau(Z_1) = Z_2$$

Что будет, если оператор τ применить к множеству Z несколько раз? Обозначим $\tau^k(Z)$ k-кратное применение оператора τ к Z (рис. 10.8), причем $\tau^0(Z) = Z$, $\tau^{k=1}(Z) = \tau\left(\tau^k(Z)\right)$.

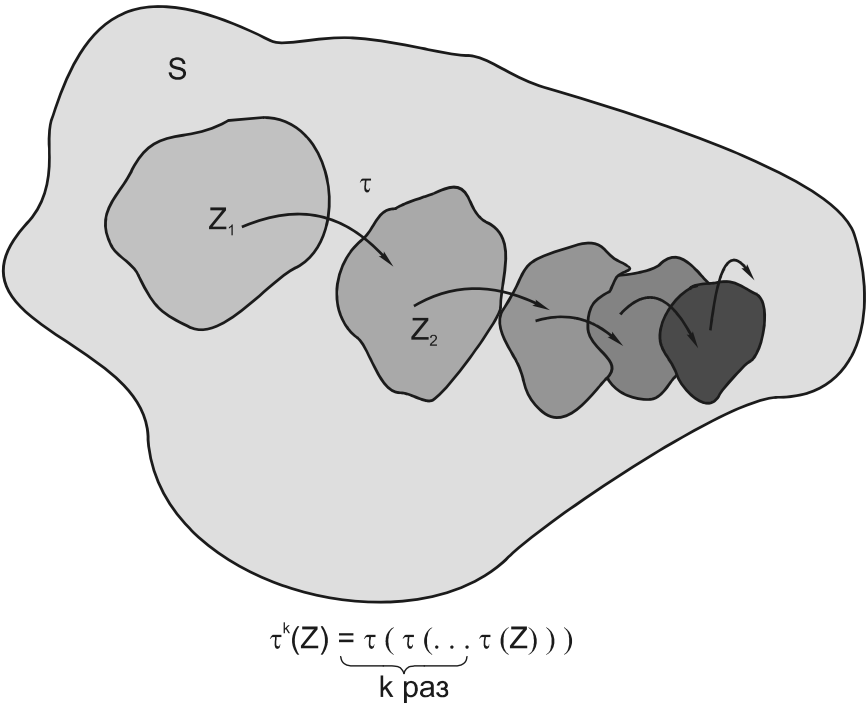


Рис. 10.8. Многократное применение оператора к множеству

Поскольку множество S конечно, многократное применение любого оператора к некоторому подмножеству S либо будет приводить в результате к единственному пределу — так называемой "*неподвижной точке оператора*", либо к конечной циклически повторяющейся последовательности множеств.

❖ ***Определение 10.1 (неподвижная точка оператора)***

Неподвижной точкой оператора τ на множестве S называется такое подмножество Z множества S, которое не изменяется при применении к нему оператора τ:

$Z \subseteq S$ — неподвижная точка оператора τ, если $\tau(Z) = Z$.

Как уже было сказано, итеративные алгоритмы построения множеств состояний, в которых выполняются формулы **EF**, **EU**, **AG** и др. строят именно неподвижные точки операторов на множестве состояний структуры Крипке, поскольку итерация в этих алгоритмах выполняется до тех пор, пока преобразование, применяемое к множеству состояний, перестанет изменять это множество. Изучение теории неподвижных точек операторов необходимо для понимания и обоснования правильности этих алгоритмов.

Пусть дано конечное множество $S = \{a, b, c, d\}$ и множество переходов R на нем (рис. 10.9). Рассмотрим несколько примеров операторов на этом множестве.

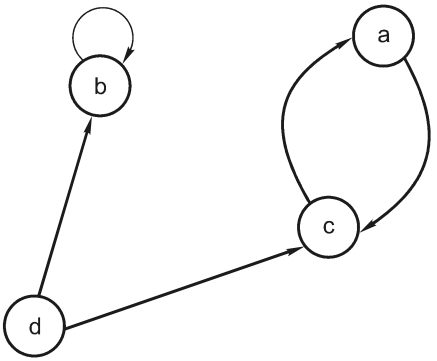


Рис. 10.9. Пример множества S с отношением перехода

Пример 10.6

$\tau_1(Q) = Q \cup \{a,b\}$

$\tau_1(\{a,d\}) = \{a,b,d\}$

$\tau_1(\{a,b,c\}) = \{a,b,c\}$ (множество $\{a,b,c\}$ — неподвижная точка).

$\tau_1(\{a,b\}) = \{a,b\}$ (неподвижная точка).

$\tau_1(S) = S$ (неподвижная точка).

$\tau_1(\varnothing) = \{a,b\}$; $\tau_1(\tau_1(\varnothing)) = \tau_1^{\,k}(\varnothing) = \{a,b\}$ при $k \geq 1$.

Таким образом, у оператора τ_1 несколько неподвижных точек.

Пример 10.7

$\tau_2(Q) = \{c\} \cup (\{a\} \cap Q)$

$\tau_2(\{a\}) = \{a,c\}$

$\tau_2(\{a,c\}) = \{a,c\}$ (неподвижная точка).

$\tau_2(S) = \{a,c\}$

$\tau_2(c) = \{c\}$ (неподвижная точка).

У оператора τ_2 две неподвижные точки: $\{c\}$, $\{a,c\}$.

Пример 10.8

$\tau_3(Q) = Q'$, где Q' — множество элементов S достижимых из элементов множества Q за один шаг отношения R (см. рис. 10.9).

$\tau_3(\{d\}) = \{b,c\}$

$\tau_3(\{b,c\}) = \{a,b\}$

$\tau_3(\{a,b\}) = \{b,c\}$

$\tau_3(\{a,c\}) = \{a,c\}$ (неподвижная точка).

Неподвижные точки оператора τ_3: $\varnothing, \{b\}, \{a,c\}, \{a,b,c\}$.

Пример 10.9

$\tau_4(Q) = Q \setminus Q'$

$\tau_4(\{d\}) = \{d\}$ (неподвижная точка).

$\tau_4(\{a,d\}) = \{a, d\}$ (неподвижная точка).

$\tau_4(\{b,c,d\}) = \{d\}$

$\tau_4(\{a,b,c,d\}) = \{d\}$

$\tau_4(\varnothing) = \varnothing$ (неподвижная точка).

Из рассмотрения примеров 10.6—10.9 можно сделать следующие выводы:

- у оператора может быть несколько неподвижных точек;
- не всегда многократное повторное применение оператора к аргументам дает в результате неподвижную точку. Например, многократное применение τ_1 и τ_2 к любому аргументу всегда дает какую-нибудь неподвижную точку, а многократное применение τ_3 к некоторым аргументам (например, $\{d\}$, $\{a,d\}$ и т. п.) приводит к циклу.

Возникают следующие вопросы:

- Существует ли какая-нибудь структуризация множества неподвижных точек операторов?
- Какая из неподвижных точек является решением проблемы, связанной с этим оператором?
- Как найти эту неподвижную точку?

Ответы на эти вопросы дает теорема Тарского.

10.7. Теорема Тарского о неподвижной точке

Теорема Тарского устанавливает структуру множества неподвижных точек для операторов, которые удовлетворяют одному дополнительному требованию: оператор должен быть монотонным.

> ❖ ***Определение 10.2 (монотонный оператор)***
>
> Оператор τ на S называется монотонным, если для любых подмножеств P, Q множества S выполняется соотношение:
>
> $$P \subseteq Q \Rightarrow \tau(P) \subseteq \tau(Q).$$

Неформально, монотонный оператор — это такой оператор, при применении которого к большему аргументу мы получим не меньший результат (рис. 10.10).

Можно проверить, что операторы $\tau_1 - \tau_3$ примеров 10.6—10.8 являются монотонными, а оператор τ_4 примера 10.9 — немонотонный. Действительно, $\tau_4(\{a\}) = \{a\}$; $\tau_4(\{a,b\}) = \varnothing$.

Очевидно, что для любого τ и любого S, $S \supseteq \tau(S)$. Применим *монотонный* оператор τ к обеим частям этого отношения. Поскольку оператор τ моно-

тонный, то $\tau(S) \supseteq \tau(\tau(S))$, $\tau(\tau(S)) \supseteq \tau(\tau(\tau(S)))$ и т. д.: для любого $k \geq 0$: $\tau^k(S) \supseteq \tau^{k+1}(S)$. Поскольку множество S конечно, то через некоторое число шагов i применения оператора τ, не большее, чем $|S|$, цепочка

$$S \supseteq \tau(S) \supseteq \tau^2(S) \supseteq \tau^3(S) \supseteq ...$$

обязательно придет к неподвижной точке, т. е. при некотором i, $\tau(\tau^i(S)) = \tau^i(S) = \tau^\infty(S)$.

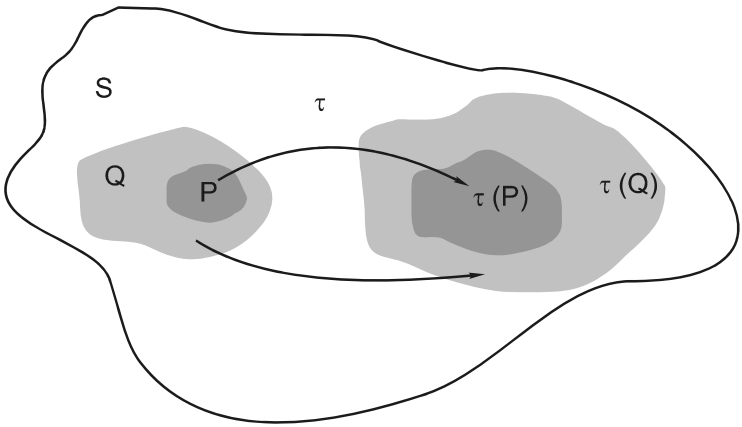


Рис. 10.10. Монотонный оператор на S

Полностью аналогично, если *монотонный* оператор τ применить к пустому множеству $\varnothing$, то через некоторое конечное число шагов k, не большее, чем $|S|$, применения оператора τ, цепочка

$$\varnothing \subseteq \tau(\varnothing) \subseteq \tau^2(\varnothing) \subseteq \tau^3(\varnothing) \subseteq ...$$

обязательно придет к неподвижной точке, т. е. при некотором k, $\tau(\tau^k(\varnothing)) = \tau^k(\varnothing) = \tau^\infty(\varnothing)$.

Таким образом, нами доказана следующая лемма.

ЛЕММА 10.1

Для монотонного оператора τ на конечном множестве S существуют такие целые i, k, не большие $|S|$, что $\tau^i(S)$ и $\tau^k(\varnothing) =$ — неподвижные точки оператора τ.

Решетки

❖ ***Определение 10.3 (решетка)***

Решетка — такой частичный порядок, в который для каждой пары элементов входит и наибольшая нижняя грань, и наименьшая верхняя грань.

Известно, что каждая решетка имеет единственный максимальный и единственный минимальный элементы. Например, множество всех подмножеств любого (не обязательно конечного) множества является решеткой. Диаграмма Хассе решетки подмножеств конечного множества (см. рис. 10.7) показывает, что для любой пары A, B подмножеств существует их *наибольшая нижняя грань* (получаемая пересечением множеств A и B) и *наименьшая верхняя грань* (получаемая объединением множеств A и B). Максимальным элементом решетки множества 2^S является множество S, минимальным элементом этой решетки является пустое множество.

Структуру множества неподвижных точек монотонных операторов определяет следующая теорема Тарского, которую мы приведем в применении к конечным множествам.

ТЕОРЕМА 10.1 (теорема Тарского для конечного множества)

Все неподвижные точки монотонного оператора на конечном множестве образуют решетку, поэтому среди них есть максимальная и минимальная неподвижные точки. Максимальная неподвижная точка монотонного оператора τ на конечном множестве S (обозначается $\nu Z.\tau(Z)$) определяется как предел применения оператора τ к множеству $S: \nu Z.\tau(Z) = \tau^{\infty}(S)$. Минимальная неподвижная точка монотонного оператора τ на конечном множестве S (обозначается $\mu Z.\tau(Z)$) определяется как предел применения оператора τ к пустому множеству: $\mu Z.\tau(Z) = \tau^{\infty}(\varnothing)$.

Выше мы уже видели, что последовательности: $S \supseteq \tau(S) \supseteq \tau^2(S) \supseteq \tau^3(S) \supseteq \ldots$ и $\varnothing \subseteq \tau(\varnothing) \subseteq \tau^2(\varnothing) \subseteq \tau^3(\varnothing) \subseteq \ldots$ для монотонного оператора τ имеют пределы — неподвижные точки (лемма 10.1). Теорема Тарского утверждает, что это соответственно наибольший и наименьший элементы решетки, которую образуют все неподвижные точки монотонного оператора τ на множестве S.

Доказательство

Докажем теорему Тарского для наименьшей неподвижной точки. Пусть S — множество с n элементами и пусть $T = \tau^{\infty}(\varnothing)$. В соответствии с лем-

мой 10.1, существует такое $k \le n$, что $\tau^k(\varnothing) = \tau\left(\tau^k(\varnothing)\right) = T$, т. е. T — неподвижная точка оператора τ. Докажем, что T — *наименьшая* неподвижная точка. Пусть $T*$ — какая-нибудь другая неподвижная точка оператора τ. Ясно, что $\varnothing \subseteq T*$. Применим оператор τ к обеим частям этого неравенства k раз. Из-за монотонности τ имеем: $\tau^k(\varnothing) \subseteq \tau^k(T*)$, откуда следует, что $T \subseteq T*$.

То, что $\tau^\infty(S)$ — *наибольшая* неподвижная точка оператора τ, доказывается аналогично.

Построим диаграммы Хассе неподвижных точек операторов $\tau_1 - \tau_4$ приведенных выше примеров 10.6—10.9 (рис. 10.11). Из этого рисунка видно, что неподвижные точки монотонных операторов $\tau_1 - \tau_3$ образуют решетки. Наибольшую неподвижную точку монотонных операторов можно получить за несколько шагов применения оператора ко всему множеству S, а наименьшую неподвижную точку этих операторов можно получить за несколько шагов применения оператора к пустому множеству $\varnothing$. Как видно из примера оператора τ_4, у немонотонных операторов неподвижные точки решетку не образуют, и наибольших и/или наименьших неподвижных точек может вовсе не быть.

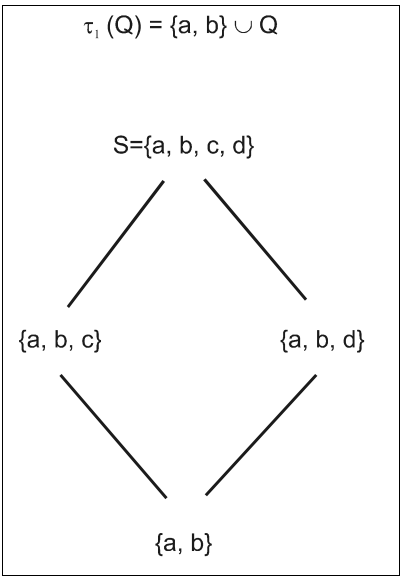
Таким образом, теорема Тарского не только определяет структуру неподвижных точек монотонных операторов на конечных множествах (все они образуют решетку), но и дает простые алгоритмы вычисления максимальной и минимальной неподвижных точек этих операторов. Важность теоремы Тарского в том, что во многих приложениях, в том числе и в символьной верификации, важны именно максимальные и/или минимальные неподвижные точки операторов.

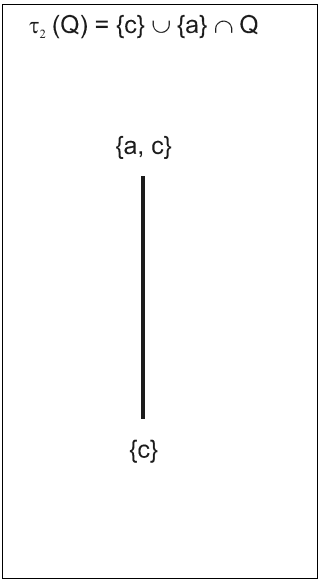
Теорема Тарского имеет непосредственное практическое применение. Пусть `Tau` — монотонный оператор на множестве S. Алгоритм вычисления *максимальной* неподвижной точки оператора `Tau`:

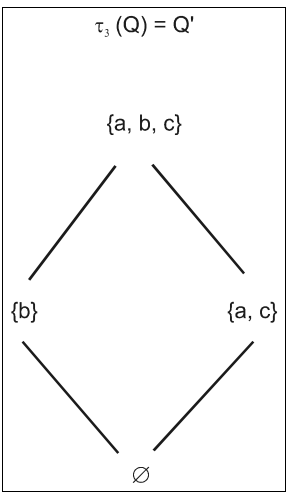
```
 A := S;
do
  B:=A;
  A:=Tau ( B );
while (A ≠ B);
return A
```

Алгоритм вычисления *минимальной* неподвижной точки оператора `Tau`:

```
 A := ∅;
do
  B:=A;
  A:=Tau( B );
while (A ≠ B);
return A
```







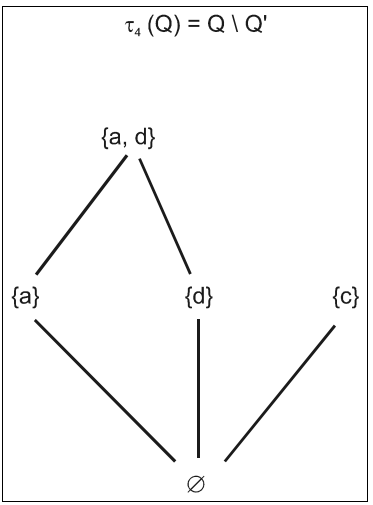


Рис. 10.11. Диаграммы Хассе неподвижных точек операторов примеров 10.6—10.9

Теорема Тарского гарантирует завершение этих алгоритмов за конечное число шагов, не большее мощности S. На рис. 10.12 приведен пример некоторой конечной решетки и множества неподвижных точек монотонного оператора на этой решетке (затемненные узлы), которое само образует решетку. Стрелками на рисунке показаны пути достижения наименьшей и наибольшей неподвижных точек этого оператора по вышеприведенным алгоритмам.

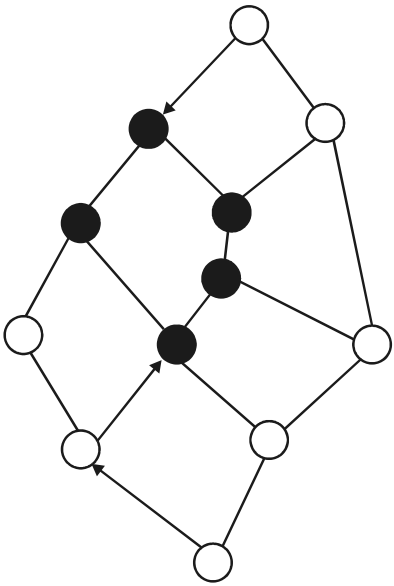


Рис. 10.12. Пример решетки неподвижных точек монотонного оператора

10.8. Символьный алгоритм верификации для формулы $\mathbf{EF}\varphi$

Вернемся к анализу темпоральных формул CTL: $\mathbf{EF}\varphi$, $\mathbf{AF}\varphi$, $\mathbf{E}(\varphi_1\mathbf{U}\varphi_2)$, $\mathbf{A}(\varphi_1\mathbf{U}\varphi_2)$, $\mathbf{EG}\varphi$ и $\mathbf{AG}\varphi$. Теоретический материал предыдущего раздела имеет непосредственное отношение к разработке алгоритмов нахождения множеств состояний структуры Крипке, удовлетворяющих этим формулам: оказывается, все эти множества являются наименьшими или наибольшими неподвижными точками некоторых операторов.

Рассмотрим сначала формулу $\mathbf{EF}\varphi$. Обозначим Q_ψ множество состояний структуры Крипке, на которых выполняется формула ψ. Для формулы $\mathbf{EF}\varphi$

постановка задачи верификации следующая. Пусть известно множество состояний Q_φ. Необходимо найти множество состояний $Q_{\mathbf{EF}\varphi}$.

Вспомним рекурсивное определение формулы $\mathbf{EF}\varphi$ (*см. раздел 3.2*):

$$\mathbf{EF}\varphi = \varphi \vee \mathbf{EX}\left(\mathbf{EF}\ \varphi\right)$$

"существует путь из текущего состояния, на котором формула φ когда-нибудь выполнится ($\mathbf{EF}\varphi$) — это то же самое, что или φ выполняется в текущем состоянии (будущее включает настоящее), или существует такой путь, что в следующем состоянии этого пути формула $\mathbf{EF}\varphi$ выполнится ($\mathbf{EX}\,\mathbf{EF}\ \varphi$)".

Аналогичное рекурсивное определение имеют остальные формулы CTL.

По рекурсивному определению формулы $\mathbf{EF}\varphi$ можно записать определение множества $Q_{\mathbf{EF}\varphi}$ состояний, на которых выполняется $\mathbf{EF}\varphi$:

$$Q_{\mathbf{EF}\varphi} = Q_\varphi \cup \mathbf{EX}\left(Q_{\mathbf{EF}\varphi}\right)$$

Рассмотрим следующий оператор на множестве состояний структуры Крипке:

$$\tau_{\mathbf{EF}\varphi}\left(Z\right) = Q_\varphi \cup \mathbf{EX}\left(Z\right)$$

Оператор $\tau_{\mathbf{EF}\varphi}$ по множеству Z строит объединение двух множеств: множества Q_φ и множества тех состояний, из каждого из которых есть переход в состояния множества Z, являющегося аргументом оператора $\tau_{\mathbf{EF}\varphi}$. Подставим вместо аргумента Z искомое множество $Q_{\mathbf{EF}\varphi}$. В соответствии с рекурсивным определением множества $Q_{\mathbf{EF}\varphi}$ можно записать $Q_{\mathbf{EF}\varphi} = \tau_{\mathbf{EF}\varphi}\left(Q_{\mathbf{EF}\varphi}\right)$. Это значит, что искомое множество $Q_{\mathbf{EF}\varphi}$ есть неподвижная точка монотонного оператора $\tau_{\mathbf{EF}\varphi}$.

Но неподвижных точек у оператора может быть много. Вопрос в том, какая из неподвижных точек нам нужна?

Вспомним два определения.

(I) Неподвижная точка оператора $\tau_{\mathbf{EF}\varphi}\left(Z\right) = Q_\varphi \cup \mathbf{EX}\left(Z\right)$ определяется так: это любое множество состояний A, включающее только такие состояния из множества Q_φ, из которых *за один шаг* можно достичь состояний из A.

(II) Семантика формулы EFφ определяется так: это все такие состояния, из которых существует путь в состояние, на котором формула φ выполнится (*гл. 3, раздел 3.2*).

Видно, что *определение неподвижной точки* оператора $\tau_{\mathbf{EF}\varphi}(Z)$ более слабое, чем семантическое определение: в нем отсутствует информация о семантике формулы **EF**φ и следующем из нее *алгоритме* построения множества $Q_{\mathbf{EF}\varphi}$. Поэтому неподвижной точкой оператора $\tau_{\mathbf{EF}\varphi}(Z)$ может быть не только искомое множество $Q_{\mathbf{EF}\varphi}$, но и те множества состояний, которые не являются решением нашей проблемы. В частности, все множество S удовлетворяет определению (I): действительно, S включает и Q_φ, и все те состояния, из которых за один шаг можно достичь состояний из S. Очевидно, что S — наибольшая неподвижная точка оператора $\tau_{\mathbf{EF}\varphi}(Z) = Q_\varphi \cup \mathbf{EX}(Z)$. Ясно поэтому, что искомое множество $Q_{\mathbf{EF}\varphi}$ — это *наименьшая* неподвижная точка оператора $\tau_{\mathbf{EF}\varphi}(Z)$. Докажем это. Докажем сначала, что оператор $\tau_{\mathbf{EF}\varphi}$ монотонный.

ЛЕММА 10.2

Оператор $\tau_{\mathbf{EF}\varphi}$ монотонный.

Доказательство

Пусть $Z \subseteq Z1$. Тогда множество состояний, из которых есть переход в Z, является подмножеством состояний, у которых есть переход в $Z1$. Поскольку операция объединения множеств монотонна, то оператор $\tau_{\mathbf{EF}\varphi}$, как композиция монотонных операторов, монотонен.

ЛЕММА 10.3

Множество $Q_{\mathbf{EF}\varphi}$ — это наименьшая неподвижная точка оператора $\tau_{\mathbf{EF}\varphi}(Z) = Q_\varphi \cup \mathbf{EF}(Z)$.

Доказательство

Доказательство леммы 10.3 проводится по индукции. Докажем сначала, что $\tau^i{}_{\mathbf{EF}\varphi}(\varnothing) \subseteq Q_{\mathbf{EF}\varphi}$ при любом i. Поскольку $\tau^0{}_{\mathbf{EF}\varphi}(\varnothing) \subseteq \varnothing$, это соотношение верно для $i = 0$. Предположим теперь, что $\tau^n{}_{\mathbf{EF}\varphi}(\varnothing) \subseteq Q_{\mathbf{EF}\varphi}$ при некотором

натуральном n. Поскольку, согласно Лемме 10.2, оператор $\tau_{\mathbf{EF}\varphi}$ монотонный, применив его к левой и правой части соотношения $\tau^n{}_{\mathbf{EF}\varphi}(\varnothing) \subseteq Q_{\mathbf{EF}\varphi}$, получим $\tau_{\mathbf{EF}\varphi}\left(\tau^n{}_{\mathbf{EF}\varphi}(\varnothing)\right) \subseteq \tau_{\mathbf{EF}\varphi}\left(Q_{\mathbf{EF}\varphi}\right)$. Но $Q_{\mathbf{EF}\varphi}$ — неподвижная точка, т. е. $\tau_{\mathbf{EF}\varphi}\left(Q_{\mathbf{EF}\varphi}\right) = Q_{\mathbf{EF}\varphi}$. Отсюда получаем справедливость индуктивного шага: $\tau^{n+1}{}_{\mathbf{EF}\varphi}(\varnothing) \subseteq Q_{\mathbf{EF}\varphi}$.

Докажем теперь, что $Q_{\mathbf{EF}\varphi} \subseteq \tau^{\infty}{}_{\mathbf{EF}\varphi}(\varnothing)$. А именно, покажем, что если для некоторого $s_1 \in S$ верно, что $s_1 = Q_{\mathbf{EF}\varphi}$, то s_1 принадлежит также и $\tau^i{}_{\mathbf{EF}\varphi}(\varnothing)$. Если $s_1 = Q_{\mathbf{EF}\varphi}$, то на структуре Крипке с отношением переходов R существует такой путь $s_1, s_2, s_3, \ldots s_i$, что $s_1 = Q_\varphi$. Докажем по индукции, что при любой длине этого отрезка пути также справедливо и $s_1 \in \tau^i{}_{\mathbf{EF}\varphi}(\varnothing)$. Это очевидно для $i = 1$, поскольку $\tau^1{}_{\mathbf{EF}\varphi}(\varnothing) = Q_\varphi$. Для обоснования индуктивного шага предположим, что на структуре Крипке существует путь $s_1, s_2, s_3, \ldots s_n$ при некотором натуральном n, $s_n \in Q_\varphi$ и при этом $s_1 \in \tau^n{}_{\mathbf{EF}\varphi}(\varnothing)$.

Покажем, что если из s_1 существует путь $s_1, s_2, s_3, \ldots s_{n+1}$, $s_1 = Q_{\mathbf{EF}\varphi}$ и $s_{n+1} \in Q_\varphi$, то верно и $s_1 \in \tau^i{}_{\mathbf{EF}\varphi}(\varnothing)$. По индуктивному предположению, $s_2 \in \tau^n{}_{\mathbf{EF}\varphi}(\varnothing)$. Но поскольку $(s_1, s_2) \in R$, то и $s_1 \in \tau^{n+1}{}_{\mathbf{EF}\varphi}(\varnothing)$. Лемма доказана.

Таким образом, множество $Q_{\mathbf{EF}\varphi}$ является минимальной неподвижной точкой этого оператора, т. е.:

$$Q_{\mathbf{EF}\varphi} = \mu Z.\tau_{\mathbf{EF}\varphi}(Z) = \mu Z.\, Q_\varphi \cup \mathbf{EX}(Z)$$

Поскольку оператор $\tau_{\mathbf{EF}\varphi}$ монотонный, то в соответствии с теоремой Тарского его минимальная неподвижная точка может быть вычислена простой "аппроксимацией снизу". Приведем символьный алгоритм вычисления наименьшей неподвижной точки монотонного оператора $\tau_{\mathbf{EF}\varphi}(Z) = Q_\varphi \cup \mathbf{EX}(Z)$, который и является алгоритмом верификации для подформулы $\mathbf{EF}\varphi$. Этот алгоритм оперирует характеристическими булевыми функциями множеств, представленными в форме BDD. В алгоритме χ_φ —

это характеристическая функция множества состояний структуры Крипке, на которых выполняется темпоральная формула φ, $\chi_{\mathbf{EF}\varphi}$ — это характеристическая функция множества состояний, на которых выполняется формула $\mathbf{EF}\varphi$, χ^*_{φ} — функция, задающая вспомогательное множество состояний. Оператор `Tau(χ*)` вычисляет операцию "Обратный образ" ограничения отношения R на множестве состояний структуры Крипке, заданном характеристической функцией χ^*_{φ}.

```
χEFφ := False;  // Вначале определяем пустое множество состояний
do
 χ* := χEFφ;
 χEFφ := χφ ∨ ∃x'1,...,x'n(χR(x1,...,xn,x'1,..., x'n) ∧ χ*(x'1,...,x'n));
                                                          // Tau(χ*)
while (χEFφ ≠ χ*);
return χEFφ
```

Пример 10.10

Рассмотрим пример вычисления состояний структуры Крипке, удовлетворяющих темпоральной формуле $\mathbf{EF}f$, если уже построено множество состояний Q_f, удовлетворяющих формуле f (рис. 10.13). Это множество $Q_{\mathbf{EF}f}$ строится как наименьшая неподвижная точка оператора $\tau_{\mathbf{EF}f}(Z) = Q_f \cup \mathbf{EX}(Z)$ — предел последовательности приближений $\varnothing \subseteq \tau_{\mathbf{EF}f}(\varnothing) \subseteq \tau^2{}_{\mathbf{EF}f}(\varnothing) \subseteq \tau^3{}_{\mathbf{EF}f}(\varnothing) \subseteq \ldots$. Рис. 10.13 показывает, что на каждом шаге к множеству помеченных состояний добавляются состояния, в которых выполняется f и все те состояния, из которых за один шаг можно достичь какого-либо уже помеченного состояния. На первом шаге помечаются состояния Q_f. Вычисление $\tau_{\mathbf{EF}f}{}^2(\varnothing) = \tau_{\mathbf{EF}f}(Q_f)$ в нашем примере дает три дополнительных состояния — из каждого из них существует переход в уже помеченные состояния. Явные алгоритмы маркировки состояний будут добавлять их последовательно, символьные алгоритмы используют представление BDD характеристических функций, описывающих соответствующие множества. На третьем шаге вычисление $\tau^3{}_{\mathbf{EF}f}(\varnothing)$ добавляет еще одну вершину к ранее помеченным: из нее за один переход попадаем в множество помеченных ранее состояний. Последующее применение преобразования к множеству уже помеченных состояний нашего примера не добавит ни одного дополнительного состояния. Алгоритм завершается: мы пришли к неподвиж-

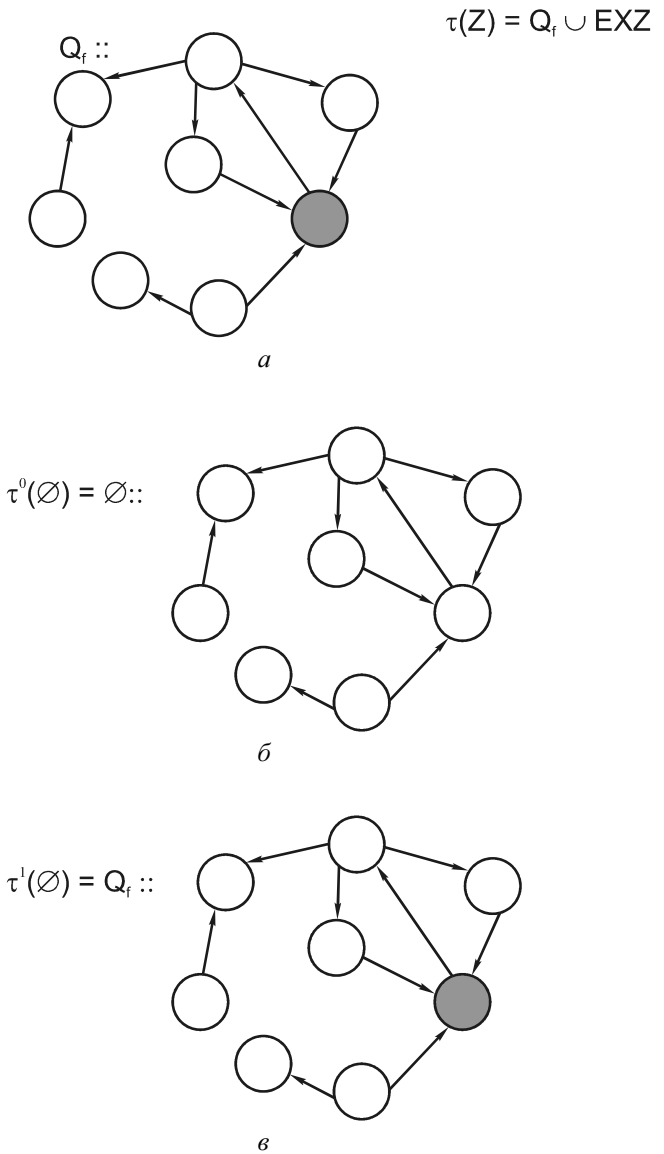
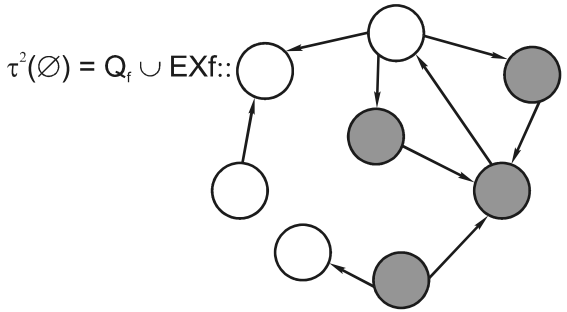
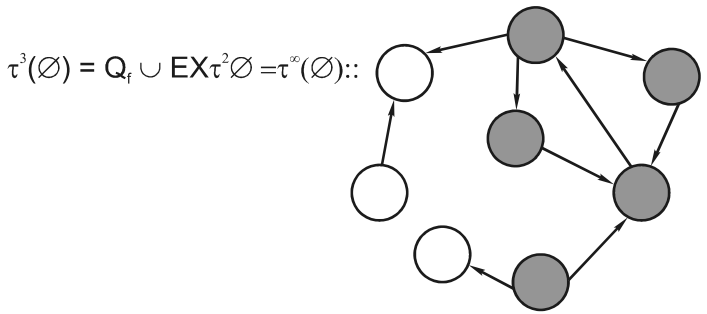


Рис. 10.13, *а—в*. Построение наименьшей неподвижной точки оператора $\tau_{\mathbf{EF}\varphi}(Z) = Q_\varphi \cup \mathbf{EX}(Z)$ (к примеру 10.10)



г



д

Рис. 10.13, *г—д*. Построение наименьшей неподвижной точки оператора $\tau_{\mathbf{EF}\varphi}(Z) = Q_\varphi \cup \mathbf{EX}(Z)$ (к примеру 10.10)

ной точке преобразования. Поскольку процедура была начата с пустого множества, полученная неподвижная точка — наименьшая.

На рис. 10.14 показана диаграмма Хассе всех неподвижных точек этого оператора (множества затемненных вершин). Для примера 10.10 таких различных множеств пять, все они образуют решетку. Одной из неподвижных точек является все множество S — оно включает в себя как множество Q_f, так и все состояния, из которых достижимы любые уже помеченные состояния. Это наибольшая из неподвижных точек оператора $\tau_{\mathbf{EF}f}(Z)$. Решением задачи примера 10.10 является наименьшая из этих неподвижных точек.

Алгоритмы построения наибольшей и наименьшей неподвижных точек монотонного преобразования τ на множестве S дает теорема Тарского — это последовательности приближений $\tau^0(S)$, $\tau^1(S)$, $\tau^2(S)$,... и

$\tau^0(\varnothing)$, $\tau^1(\varnothing)$, $\tau^2(\varnothing)$, ... Остальные неподвижные точки можно построить перебором. Здесь они построены по следующему правилу, диктуемому видом оператора $\tau_{\mathbf{EF}f}(Z)$: неподвижная точка оператора $\tau_{\mathbf{EF}f}(Z) \in Q_f \cup \mathbf{EX}(Z)$ — это любое множество состояний M, которое включает в себя множество Q_f, и если в M входит состояние s, то в M входят и все те состояния, из которых s достижимо.

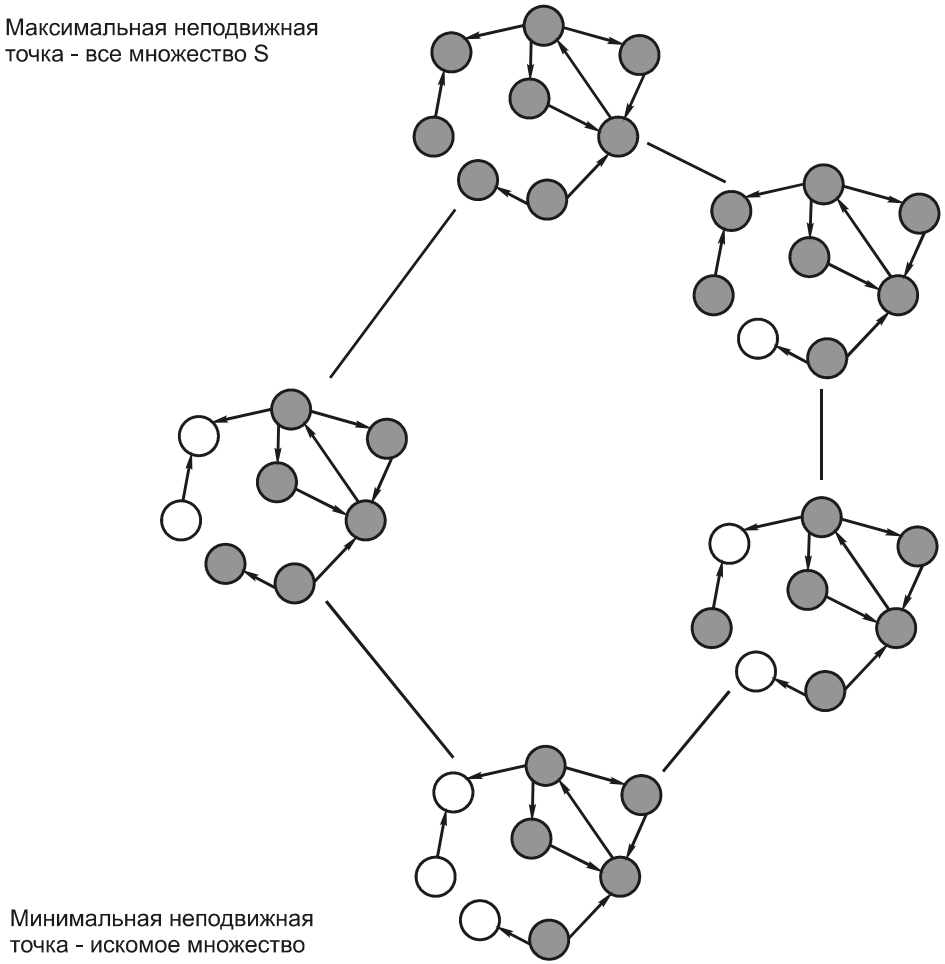


Рис. 10.14. Решетка неподвижных точек оператора $\tau_{\mathbf{EF}}$ примера 10.10

10.9. Определение темпоральных формул CTL через неподвижные точки операторов

Для других формул CTL также разработаны итеративные алгоритмы верификации, находящие нужную неподвижную точку соответствующего оператора на множестве состояний структуры Крипке. Все алгоритмы построения этих множеств манипулируют не отдельными состояниями множеств, а целыми множествами, заданными характеристическими булевыми функциями в форме BDD.

В действительности, необходимо построить лишь алгоритмы для тех формул, которые составляют один из возможных базисов CTL, например, $\{\mathbf{EX}\varphi, \mathbf{EG}\varphi, \mathbf{E}(\varphi_1 \mathbf{U} \varphi_2)\}$. Алгоритм для формулы $\mathbf{EX}\varphi$ построен выше, он не итеративный и находит множество $Q_{\mathbf{EX}\varphi}$ состояний по заданному множеству Q_φ состояний простыми операциями (в частности, операцией "Обратный образ") над BDD характеристических булевых функций, задающих множество Q_φ.

Все искомые неподвижные точки для темпоральных формул CTL — либо наибольшие, либо наименьшие. Построим, например, итеративный алгоритм для формулы $\mathbf{EG}\varphi$: этот алгоритм интересен тем, что он будет искать наибольшую неподвижную точку некоторого оператора на множестве состояний структуры Крипке. Покажем, что:

- множество $Q_{\mathbf{EG}\varphi}$ состояний структуры Крипке, удовлетворяющих формуле $\mathbf{EG}\varphi$, является неподвижной точкой некоторого оператора $\tau_{\mathbf{EG}\varphi}$ на множестве состояний структуры Крипке;
- оператор $\tau_{\mathbf{EG}\varphi}$ монотонный;
- множество $Q_{\mathbf{EG}\varphi}$ является наибольшей неподвижной точкой оператора $\tau_{\mathbf{EG}\varphi}$ и, следовательно, по теореме Тарского, оно может быть найдено простым алгоритмом — "аппроксимацией сверху": $\tau^0{}_{\mathbf{EG}\varphi}(S), \tau^1{}_{\mathbf{EG}\varphi}(S), \tau^2{}_{\mathbf{EG}\varphi}(S), \ldots$.

Рекурсивное определение темпоральной формулы $\mathbf{EG}\varphi$ следующее (*см. раздел 2.3 гл. 2*):

$\mathbf{EG}\varphi = \varphi \wedge \mathbf{EX}\left(\mathbf{EG}\varphi\right)$**:**

существует такой путь из текущего состояния, во всех состояниях которого темпоральная формула φ выполняется — это то же самое, что φ выполняется в текущем состоянии и существует такой путь, в следующем состоянии которого формула $\mathbf{EG}\varphi$ выполняется.

На основании этого определения можно записать:

$$Q_{\mathbf{EG}\varphi} = Q_{\varphi} \cap \mathbf{EX}\left(Q_{\mathbf{EG}\varphi}\right)$$

множество состояний, на которых формула $\mathbf{EG}\varphi$ истинна — это только такие состояния среди состояний Q_{φ}, что из них существует переход в состояния, в которых формула $\mathbf{EG}\varphi$ истинна.

Отсюда сразу следует, что можно построить оператор $\tau_{\mathbf{EG}\varphi}$ на множестве состояний структуры Крипке:

$$\tau_{\mathbf{EG}\varphi}\left(Z\right) = Q_{\varphi} \cap \mathbf{EX}\left(Z\right)$$

такой, что искомое множество $Q_{\mathbf{EG}\varphi}$ состояний структуры Крипке, на которых выполняется формула $\mathbf{EG}\varphi$, является неподвижной точкой этого оператора:

$$\tau_{\mathbf{EG}\varphi}\left(Q_{\mathbf{EG}\varphi}\right) = Q_{\varphi} \cap \mathbf{EX}\left(Q_{\mathbf{EG}\varphi}\right) = Q_{\mathbf{EG}\varphi}$$

Но какой из возможных неподвижных точек является искомое множество? Рассмотрим два определения.

(I) Неподвижная точка оператора $\tau_{\mathbf{EG}\varphi}\left(Z\right) = Q_{\varphi} \cap \mathbf{EX}\left(Z\right)$ определяется так: это любое множество состояний A, включающее только такие состояния из множества Q_{φ}, из которых *за один шаг* можно достичь состояний из A.

(II) Семантика формулы $\mathbf{EG}\varphi$ определяется так: это все такие состояния, на которых формула φ выполняется и из которых существует (бесконечный) путь, во всех состояниях которого формула φ выполняется.

Можно видеть, что определение неподвижной точки оператора $\tau_{\mathbf{EG}\varphi}\left(Z\right)$ более слабое, чем семантическое определение: в нем отсутствует информация о семантике формулы $\mathbf{EG}\varphi$ и следующем из нее алгоритме построения множества $Q_{\mathbf{EG}\varphi}$. Поэтому определению неподвижной точки оператора $\tau_{\mathbf{EG}\varphi}$ могут удовлетворять и те множества состояний, которые не являются решением

нашей проблемы. Например, пустое множество $\varnothing$ удовлетворяет этому определению: оно вовсе не включает состояний из Q_φ и поэтому включает (пустое) множество всех тех состояний, из которых за один шаг можно достичь состояний из $\varnothing$. Очевидно, что $\varnothing$ — это наименьшая неподвижная точка оператора $\tau_{\mathbf{EG}\varphi}(Z) = Q_\varphi \cap \mathbf{EX}(Z)$.

Интуитивно ясно, что искомое множество $Q_{\mathbf{EG}\varphi}$ является наибольшей неподвижной точкой оператора $\tau_{\mathbf{EG}\varphi}$ и, следовательно, по теореме Тарского, оно может быть найдено простым алгоритмом последовательных приближений — "аппроксимацией сверху": подсчетом $\tau^{\infty}_{\mathbf{EG}\varphi}(S)$. Доказательство этого факта аналогично доказательству Леммы 10.13.

Приведем символьный алгоритм вычисления наибольшей неподвижной точки монотонного оператора $\tau_{\mathbf{EG}\varphi}(Z) = Q_\varphi \cap \mathbf{EX}(Z)$, который и является алгоритмом верификации для подформулы $\mathbf{EG}\varphi$. Этот алгоритм манипулирует характеристическими булевыми функциями множеств, представленными в форме BDD и полностью аналогичен алгоритму вычисления наименьшей неподвижной точки оператора $\tau_{\mathbf{EG}\varphi}$, приведенному выше. Оператор `Tau(χ*)` вычисляет операцию "Прямой образ" ограничения отношения R структуры Крипке на множестве состояний этой структуры, заданном характеристической функцией χ^*.

```
χEGφ := True;                    // Вначале берем все множество состояний S
do
 χ* := χEGφ;
 χEGφ :=
  χφ ∧ ∃x'1,...,x'n(χR(x1,...,xn,x'1,..., x'n) ∧ χ*(x'1,x'2,...,x'n));
                                                          // Tau(χ*)
while (χEGφ ≠ χ*);
return χEGφ
```

Пример 10.11

Рассмотрим пример вычисления состояний структуры Крипке, удовлетворяющих темпоральной формуле $\mathbf{EG}f$. Пусть уже построено множество состояний Q_f, удовлетворяющих формуле f (рис. 10.15). Искомое множество $Q_{\mathbf{EG}f}$ строится как наибольшая неподвижная точка оператора $\tau_{\mathbf{EG}f}(Z) = Q_f \cap \mathbf{EX}(Z)$ — предел последовательности приближений $S \supseteq \tau_{\mathbf{EG}f}(S) \supseteq \tau_{\mathbf{EG}f}{}^2(S) \supseteq \tau_{\mathbf{EG}f}{}^3(S) \supseteq \ldots$.

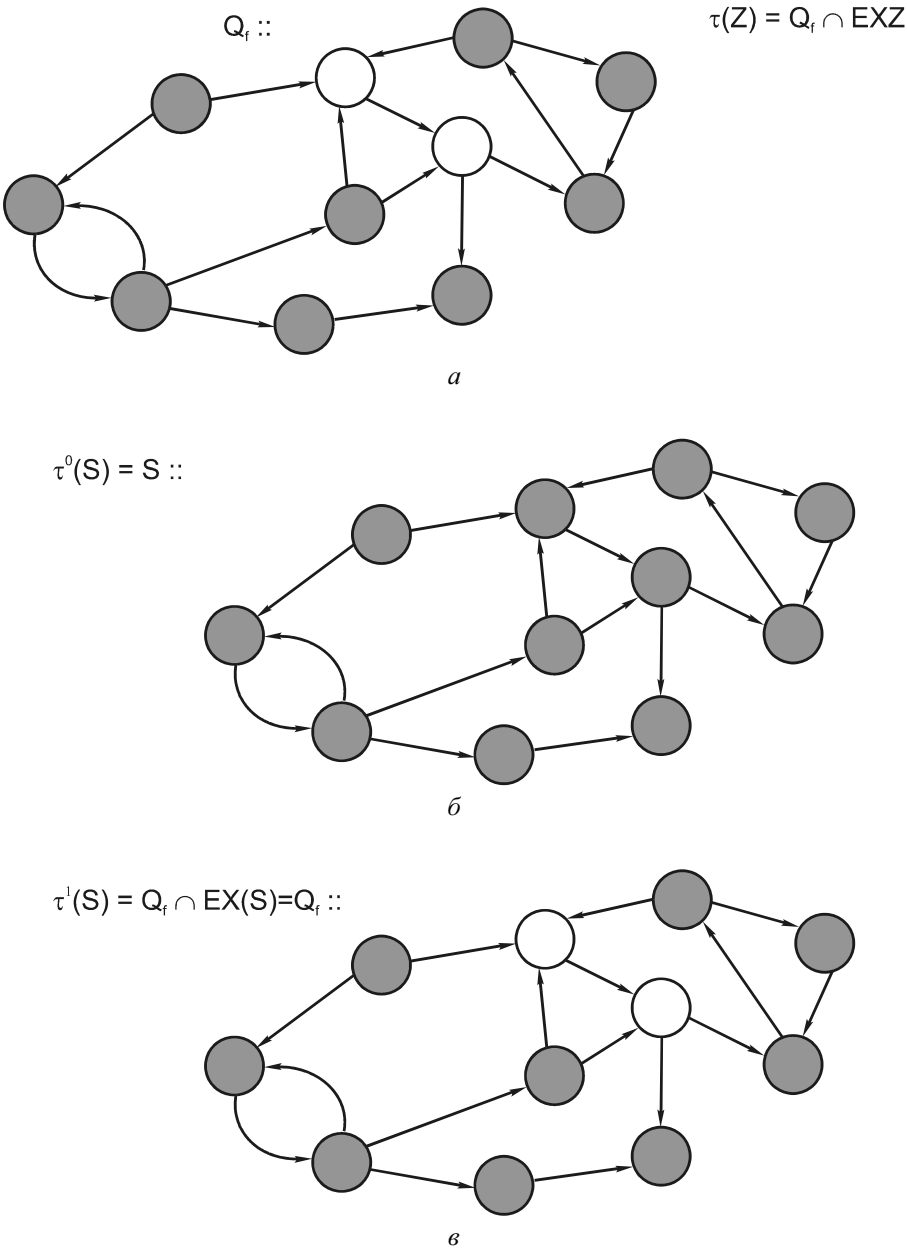
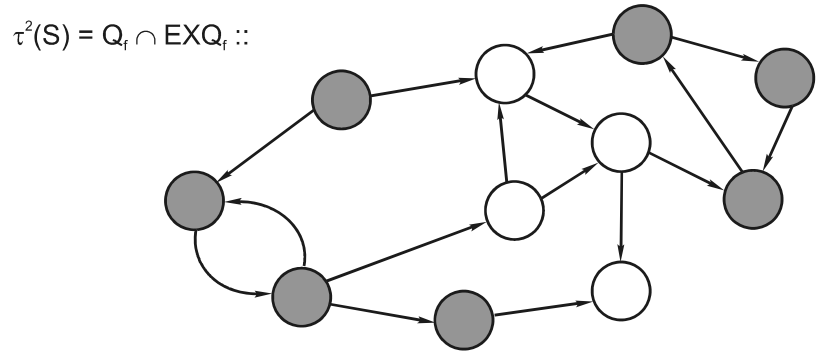
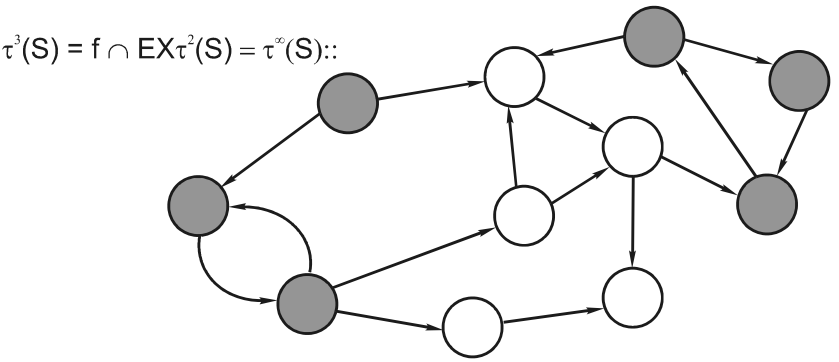


Рис. 10.15, *а—в*. Построение наибольшей неподвижной точки оператора $\tau(Z) = Q_f \cap \mathbf{EX}(Z)$ (к примеру 10.11)



г



д

Рис. 10.15, *г* и *д*. Построение наибольшей неподвижной точки оператора $\tau(Z) = Q_f \cap \mathbf{EX}(Z)$ (к примеру 10.11)

Рис. 10.15 показывает, что на каждом шаге из предварительно помеченных состояний остаются помеченными только те состояния, которые помечены f и из которых за один шаг можно достичь какого-либо помеченного состояния. Сначала помечено все множество S. На первом шаге помеченными остаются только состояния Q_f, потому что $Q_f \cap \mathbf{EX}(S) = Q_f \cap S = Q_f$. Вычисление $\tau_{\mathbf{EG}f}{}^2(S) = \tau_{\mathbf{EG}f}(Q_f)$ в нашем примере выбрасывает два помеченных состояния — из каждого из них не существует перехода в уже помеченные состояния. Явные алгоритмы маркировки состояний будут выбрасывать их последовательно, символьные алгоритмы используют представление BDD характеристических функций, описывающих соответствующие множества и операцию конъюнкции. На третьем шаге вычисление $\tau^3(\varnothing)$ выбрасывает еще одно состояние из помеченных: из него за один пе-

переход нельзя попасть в множество помеченных ранее состояний. Последующее применение преобразования к множеству уже помеченных состояний нашего примера не выбросит ни одного ранее помеченного состояния: из всех них существует переход в какое-нибудь помеченное состояние. Алгоритм завершен: мы пришли к неподвижной точке преобразования. Поскольку процедура была начата с полного множества, полученная неподвижная точка — наибольшая.

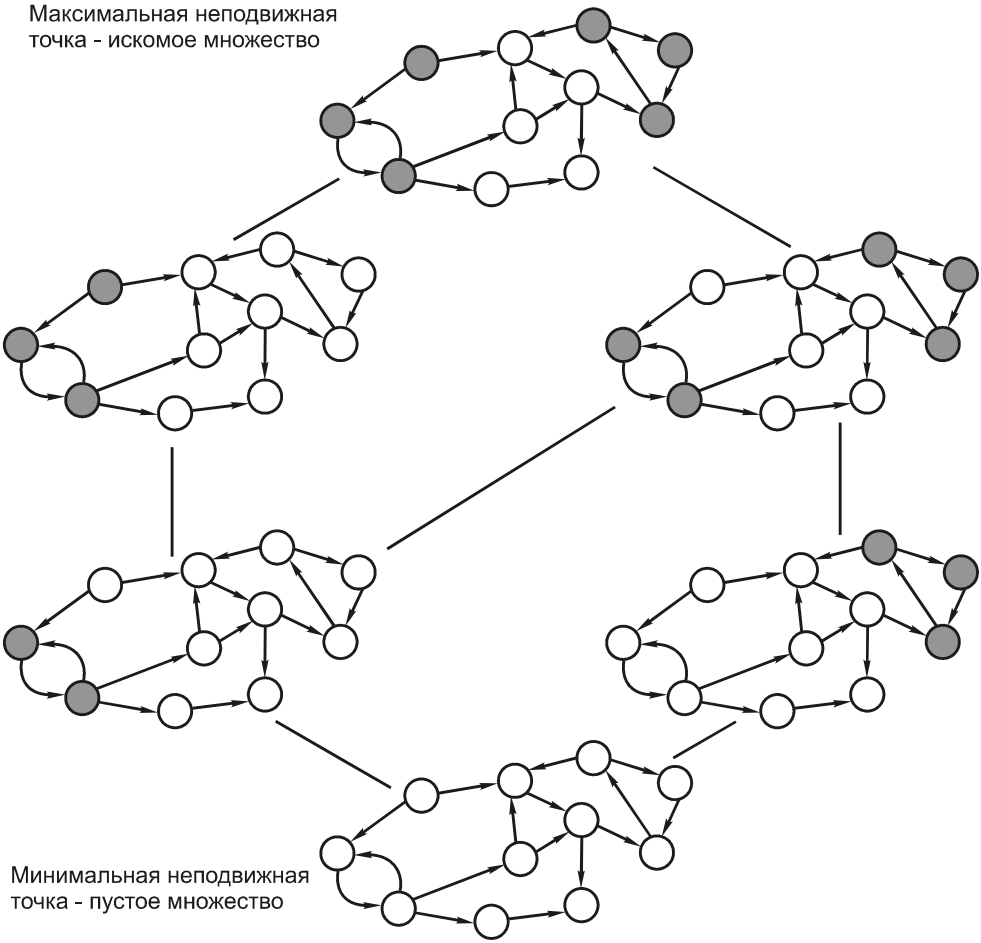


Рис. 10.16. Решетка неподвижных точек оператора $\tau_{\mathbf{EG}}$ примера 10.11

На рис. 10.16 показана диаграмма Хассе всех неподвижных точек этого оператора (множества затемненных вершин). В примере 10.11 таких различных множеств шесть, все они образуют решетку. Одной из неподвижных точек

является пустое множество $\varnothing$ — любое состояние этого множества (их нет) обладает следующим свойством: оно принадлежит подмножеству множества Q_f и из него достижимо состояние, обладающее этим же свойством. Это наименьшая из неподвижных точек оператора $\tau_{\mathbf{EG}f}$. Решением задачи примера 10.11 является наибольшая из этих неподвижных точек.

Алгоритмы построения наибольшей и наименьшей неподвижных точек монотонного оператора τ на множестве S дает теорема Тарского. Остальные неподвижные точки можно построить перебором. Здесь они построены по следующему правилу, диктуемому видом оператора $\tau_{\mathbf{EG}f}$: неподвижная точка оператора $\tau_{\mathbf{EG}f}(Z) = Q_f \cap \mathbf{EX}(Z)$ — это любое множество состояний M, которое является подмножеством множества Q_f, и если в M входит состояние s, то из s есть переход в состояние из M.

Рекурсивное определение всех остальных формул темпоральной логики CTL, введенное в *разделе 2.3 гл. 2*, дает возможность определить искомые множества состояний структуры Крипке как неподвижные точки соответствующих преобразований. Построим таблицу, в левой части которой запишем рекурсивное определение темпоральной формулы, а в правой части — определение этой формулы как наибольшей ($vZ.\tau(Z)$) или наименьшей ($\mu Z.\tau(Z)$) неподвижной точки оператора на множестве состояний структуры Крипке. Для удобства будем понимать каждую темпоральную формулу в правой части таблицы как множество таких состояний структуры Крипке, на которых истинна эта формула.

$\mathbf{EF}\varphi = \varphi \vee \mathbf{EX}\,\mathbf{EF}\varphi$	$\mathbf{EF}\varphi = \mu Z.\,\varphi \cup \mathbf{EX}(Z)$
$\mathbf{AF}\varphi = \varphi \vee \mathbf{AX}\,\mathbf{AF}\varphi$	$\mathbf{AF}\varphi = \mu Z.\,\varphi \cup \mathbf{AX}(Z)$
$\mathbf{EG}\varphi = \varphi \wedge \mathbf{EX}\,\mathbf{EG}\varphi$	$\mathbf{EG}\varphi = vZ.\,\varphi \cap \mathbf{EX}(Z)$
$\mathbf{AG}\varphi = \varphi \wedge \mathbf{AX}\,\mathbf{AG}\varphi$	$\mathbf{AG}\varphi = vZ.\,\varphi \cap \mathbf{AX}(Z)$
$\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2) = \varphi_2 \vee (\varphi_1 \wedge \mathbf{EX}\,\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2))$	$\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2) = \mu Z.\,\varphi_2 \cup (\varphi_1 \cap \mathbf{EX}(Z))$
$\mathbf{A}(\varphi_1 \mathbf{U} \varphi_2) = \varphi_2 \vee (\varphi_1 \wedge \mathbf{AX}\,\mathbf{A}(\varphi_1 \mathbf{U} \varphi_2))$	$\mathbf{A}(\varphi_1 \mathbf{U} \varphi_2) = \mu Z.\,\varphi_2 \cup (\varphi_1 \cap \mathbf{AX}(Z))$

10.10. Заключение

Алгоритмы проверки модели осуществляют вычисление множества состояний структуры Крипке, на котором выполняется заданная формула темпоральной логики. Увеличение числа компонентов верифицируемой системы или увеличение числа переменных экспоненциально увеличивает число состояний структуры Крипке, являющейся моделью системы. Этот эффект, названный "*взрывом числа состояний*", ограничивает применение обычных алгоритмов верификации model checking. Алгоритмы верификации, работающие с явным представлением состояний анализируемой системы и оперирующие состояниями и переходами последовательно, одно за другим, могут быть использованы только для проверки "игрушечных" систем.

Разработанные сравнительно недавно "*неявные*", или "*символьные*" алгоритмы, которые используют очень эффективный метод представления дискретных данных — множеств и отношений — в виде булевых функций в форме бинарных решающих диаграмм (BDD), позволяют увеличить число состояний верифицируемых систем до астрономических значений. Для логических схем и программ разработаны очень эффективные системы символьной верификации, что привело к возможности использования алгоритмов верификации model checking для реальных, разрабатываемых промышленностью систем и сделало верификацию этапом технологии разработки промышленных систем.

Обоснованием алгоритмов проверки свойств, выраженных формулами темпоральной логики CTL, является определение этих формул с помощью неподвижных точек (наибольших или наименьших) операторов на множестве состояний структуры Крипке. Теорема Тарского дает простые и ясные алгоритмы вычисления наибольших и наименьших неподвижных точек таких операторов. Эти вычисления могут быть реализованы либо явными алгоритмами, работающими с каждым элементом обрабатываемых множеств, либо неявными (символьными) алгоритмами, работающими с булевыми функциями в форме BDD, представляющими эти множества.

Метод проверки моделей, основанный на символьных алгоритмах, стал сегодня стандартным этапом технологии разработки микропроцессоров и встроенных систем во многих фирмах. Существуют также свободно распространяемые верификаторы, работающие на основе символьной верификации (например, SMV и NuSMV, разработанные в Университете Карнеги-Меллона). Символьные алгоритмы верификации являются, однако, только одним из подходов к решению проблемы "взрыва числа состояний". Другими подходами являются композициональная верификация, редукция частичных порядков, верификация "на лету" и др. Хотя ни одна из разработанных техник не

является универсально применимой, на основе каждой из них построены инструментальные системы верификации.

10.11. Замечания

Использование бинарных решающих диаграмм в верификации, в частности, для повышения эффективности алгоритмов model checking, было предложено в начале 90-х годов прошлого века в работе [14]. Этой проблеме посвящена докторская диссертация Мак-Миллана [112] и его монография [113]. В 1998 г. ассоциация ACM наградила специальной премией имени Каннелакиса за лучшую работу в области теории и практики (ACM Kanellakis Award for Theory and Practice) группу исследователей, в которую вошли Рэндел Брайант (Randal E.Bryant), Эдмунд Кларк (Edmund M.Clarke), Аллен Эмерсон (E. Allen Emerson) и Кеннет Мак-Миллан (Kenneth L. McMillan), "*за изобретение ими метода "символьной проверки модели" — метода формальной проверки проектов систем*".

10.12. Задачи к главе 10

10.1. Пусть $S = \{a, b, c, d\}$ — множество, и $f: 2^S \to 2^S$ такой оператор на S : $f(X) = X \cup \{a,c\}$.

а) Проверьте, является ли f монотонным.

б) Постройте наибольшую и наименьшую неподвижные точки оператора.

10.2. Пусть $S = \{a, b, c, d, e\}$ — множество, и $\tau: 2^S \to 2^S$ оператор на S: $\tau(Z) = Z \cap \{a,c,d\}$.

а) Проверьте, является ли τ монотонным.

б) Постройте наибольшую и наименьшую неподвижные точки τ.

в) Постройте решетку неподвижных точек оператора τ.

10.3. Пусть $\Gamma = (S,G)$ — орграф:

$S = \{p, q, r, s\}$

$G = \{(p,s),(r,q),(s,s),(q,s),(s,q),(r,p),(p,q)\}$

Пусть $\tau\colon 2^S \to 2^S$ такой оператор: $\tau(Q) = Q \cap Q'$, где Q' — состояния, достижимые из состояний Q за один шаг.

а) Проверьте, является ли τ монотонным.

б) Постройте наибольшую и наименьшую неподвижные точки τ.

в) Постройте решетку неподвижных точек оператора τ.

10.4. Пусть задано множество $S = \{a, b, c, d, e\}$ и переходы $\delta = \{(a,a), (a,b), (b,a), (c,a), (c,d), (d,c), (d,e), (e,e)\}$. Пусть оператор $\tau: 2^S \to 2^S$ на множестве всех подмножеств S определен так: $\tau(Q) = \{e\} \cup \mathbf{EX}(Q)$, где $\mathbf{EX}(Q)$ — множество таких элементов S, из которых за один шаг могут быть достигнуты элементы из Q. Постройте все неподвижные точки оператора τ и проверьте, образуют ли они решетку.

10.5. Пусть $S = \{a, b, c, d, e\}$ — множество состояний, на котором определена функция переходов $\delta: S \to S: \delta(a) = b; \delta(b) = c; \delta(c) = c; \delta(d) = a$. Пусть $f: 2^S \to 2^S$ такой оператор: $f(Q) = \{a, c, d\} \cap \delta(Q)$, где $\delta(Q) = \cup_{X \in Q} \delta(x)$.

а) Проверьте, является ли f монотонным.

б) Постройте наибольшую и наименьшую неподвижные точки f.

в) Постройте решетку неподвижных точек f.

10.6. Определите на множестве $S = \{a, b, c, d\}$ такое отношение переходов $\delta \subseteq s \times S$, чтобы монотонный оператор $\tau(Q) = \{b\} \cap \delta(Q)$, где $\delta(Q) = \cup_{X \in Q} \delta(x)$, имел бы четыре неподвижные точки: $\{b\}, \{a,b\}, \{b,d\}, \{a,b,d\}$. Заметьте, что наименьшая неподвижная точка этого оператора не $\varnothing$, а его наибольшая неподвижная точка — не S. Постройте решетку неподвижных точек этого оператора.

10.7. На произвольно выбранной структуре Крипке задайте множество состояний Ф. Найдите множество M таких состояний, из которых существует такой путь, на каждом расстоянии четной длины которого мы попадаем в состояния из Ф.

10.8. По аналогии с доказательством теоремы Тарского для наименьшей неподвижной точки, дайте доказательство алгоритма построения наибольшей неподвижной точки.

10.9. По аналогии с построением булевой формулы переходов для оператора ***wait*** (*см. раздел 10.4*) постройте булевы формулы для операторов ***lock*** и ***unlock*** .

Оператор ***lock*** имеет следующую семантику: при выполнении оператора $\textbf{\textit{lock}}(b)$ непрерывно проверяется, будет ли переменная b равна 0. Как только она стала равной 0, значение v устанавливается в 1 (это, фактически, операция *test-and-set* систем команд некоторых процессоров). Оператор $\textbf{\textit{unlock}}(b)$ устанавливает значение своего параметра в 0.

10.10. Пусть на множестве подмножеств множества $S = \{1, 2, 3, 4, 5, 6\}$ заданы операторы:

$f_1(Q) = Q \setminus \{1,3,5\}$;

$f_2(Q) = \{2,4,6\} \setminus Q$;

$f_3(Q) = \{1,2,3,5\} \cap (Q \cup \{2,4,5\})$;

$f_4(Q) = \{1,2,3,5\} \cup (Q \cap \{2,4,5\})$.

а) Какие из этих операторов монотонны?

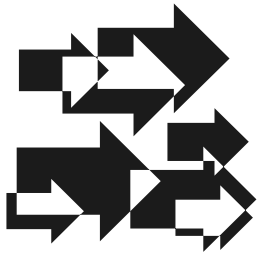
б) Постройте наибольшую и наименьшую неподвижную точки каждого монотонного оператора.

в) Постройте решетку неподвижных точек каждого монотонного оператора.

г) Имеет ли оператор f_2 неподвижные точки? Постройте все неподвижные точки оператора f_2.

д) Образуют ли они решетку?

ГЛАВА 11



Количественный анализ систем

Исследования и разработки в области верификации программ и аппаратуры в основном связаны с проверкой качественных свойств этих систем. При этом обычно игнорируются количественные аспекты поведения системы, связанные с оценками вероятности наступления некоторых событий или оценками, связанными с реальным временем. В то же время для многих систем вероятностный анализ весьма важен: для протоколов передачи данных, например, значительно более важна оценка вероятности доставки сообщения в ограниченном интервале времени, чем знание о том, что сообщение будет доставлено "когда-нибудь в будущем" (что может быть определено проверкой выполнения темпоральной формулы **F***receive*). Для оценки производительности таких систем необходим анализ вероятностных моделей. Кроме того, алгоритмы многих современных протоколов (например, Ethernet, схема отката в протоколах 802.11 и BlueTooth, динамическая маршрутизация и др.) основаны на процедурах рандомизации, позволяющих нарушить симметрию идентичных активных процессов, например, для того, чтобы избежать повторения патовой ситуации при захвате ресурса. Поэтому количественные вероятностные оценки часто важны при анализе систем — их функциональной корректности, качества сервиса, производительности.

В дополнение к анализу, который проводит обычная система верификации, основанная на методе *model checking*, не так давно был разработан метод *вероятностной проверки модели*, с помощью которого можно вычислить вероятность выполнения логического свойства системы, заданного темпоральной формулой, а также анализировать свойства, связанные с дискретным временем.

11.1. Вероятностный метод model checking

Простейшую модель системы для выполнения вероятностной верификации можно построить как расширение обычной дискретной Марковской цепи, т. е. системы переходов, каждый переход которой из любого состояния снабжен вероятностью активизации этого перехода. В марковских цепях вероятностный выбор перехода в следующее состояние полностью определяется тем, в каком текущем состоянии находится система, а не историей ее предыдущего поведения.

❖ ***Определение 11.1***

Помеченная вероятностная структура — это пятерка $M = (S, s_0, P, AP, L)$, где:

- S — конечное множество (состояний);
- $s_0 \in S$ — начальное состояние;
- $\mathbf{P} : S \times S \to [0,1]$ — вероятностная матрица, такая, что $(\forall s \in S) : \Sigma_{q \in S} \mathbf{P}(s,q) = 1$, т. е. сумма вероятностей переходов из любого состояния s в другие состояния равна 1;
- AP — конечное множество атомарных предикатов;
- $L : S \to 2^{AP}$ — функция пометок, ставящая в соответствие каждому состоянию множество атомарных предикатов (которые истинны в этом состоянии).

Элементы матрицы $\mathbf{P}$ определяют вероятности переходов из одного состояния в другое. Выбор возможного перехода здесь осуществляется исключительно на основе вероятностного распределения $\mathbf{P}$. Этот вероятностный выбор не отражает недетерминированности окружения данного процесса, он описывает вероятностный выбор, который делает сам процесс, находясь в каждом состоянии. В частности, вероятностный выбор отличен от недетерминизма, возникающего как следствие асинхронного функционирования параллельных процессов.

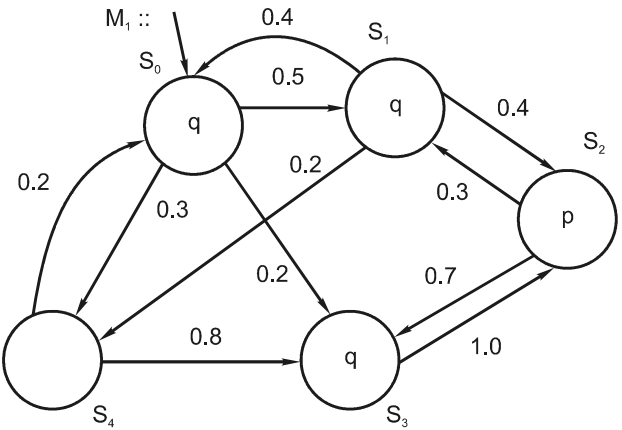
Помеченная вероятностная структура в литературе также называется помеченной Марковской цепью с дискретным временем (Labeled Discrete Time Markov Chain). Мы будем называть эту модель просто вероятностной структурой.

❖ ***Определение 11.2***

Вероятностный метод model checking — это совокупность методов и алгоритмов оценки вероятности того, что некоторая формула темпоральной логики выполняется на вычислениях вероятностной структуры.

Пример 11.1

На рис. 11.1, *а* представлен пример вероятностной структуры M_1. Структура M_1 имеет пять состояний, помеченных атомарными предикатами из множества $AP = \{p, q\}$. Начальное состояние этой структуры s_0. С каждым переходом из конкретного состояния связана вероятность активизации этого перехода. Вероятностная матрица **P** представлена на рис. 11.1, *б*. В матрице приведены только переходы с ненулевыми вероятностями.



а

P::	S_0	S_1	S_2	S_3	S_4
S_0	--	0.5	–	0.2	0.3
S_1	0.4	–	0.4	–	0.2
S_2	--	0.3	--	0.7	–
S_3	--	–	1.0	–	–
S_4	0.2	–	--	0.8	–

б

Рис. 11.1. *а*) Пример вероятностной структуры; *б*) ее вероятностная матрица

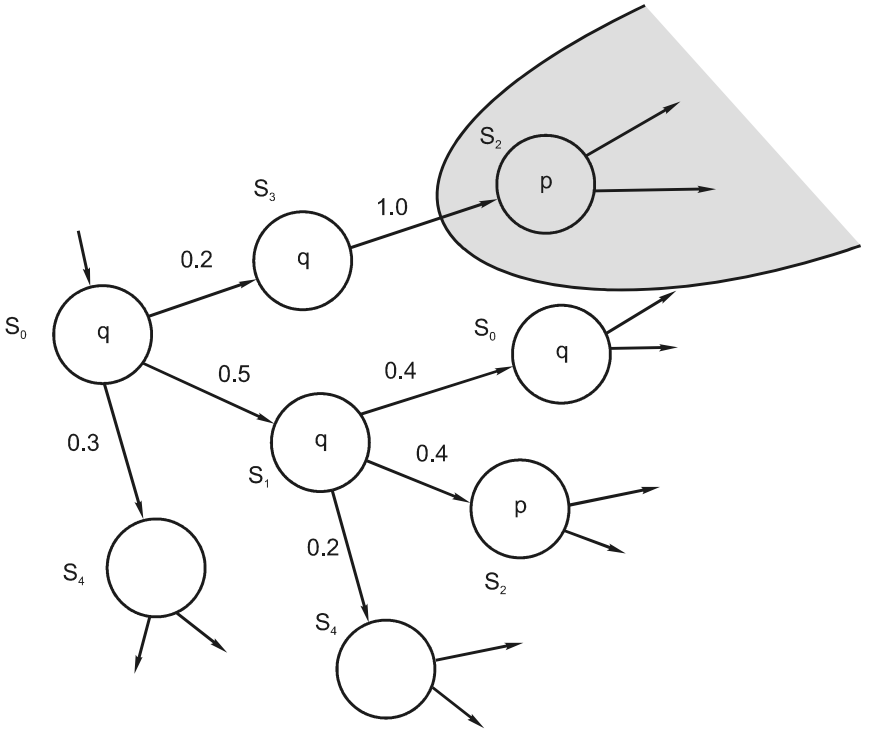


Рис. 11.1. *в*) Фрагмент развертки вычислений структуры M_1 из состояния s_0 и один из цилиндров

Анализ свойств вероятностных структур связан с рассмотрением их вычислений. Вычисление в вероятностной структуре M — это любая бесконечная последовательность состояний $\pi = s_0 s_1 s_2 \ldots$, таких, что для каждой соседней пары (s_i, s_{i+1}) состояний последовательности π вероятность соответствующего перехода ненулевая: $\mathbf{P}(s_i, s_{i+1}) > 0$. Множество всех непустых конечных префиксов вычисления π обозначим $pref(\pi)$.

Каждый шаг любого вычисления вероятностной структуры имеет некоторую вероятность его выполнения, задаваемую вероятностной матрицей этой структуры. Обозначим $Paths(M)$ множество всех вычислений структуры M, а множество всех конечных префиксов вычислений структуры M обозначим $Paths_{fin}(M)$.

При анализе вероятностных структур нужно рассматривать вероятности множеств бесконечных вычислений. Поэтому здесь следует тщательно определить, как подсчитать вероятностную меру множества вычислений.

Анализ вероятностных систем обычно основывается на аксиоматике сигма-алгебры Колмогорова, которая формально описывает понятия элементарного события, события и вероятности событий. Пусть Ω — множество элементов, которые называются "элементарными событиями". Каждое подмножество $A \subseteq \Omega$ множества Ω называется "случайным событием" или просто "событием". С событиями A и B можно выполнять любые теоретико-множественные операции: объединение $A \cup B$, пересечение $A \cap B$, дополнение $\backslash A$. Каждому событию $A \subseteq \Omega$ ставится в соответствие его вероятностная мера $Pr(A)$ — положительное вещественное число.

Тройка $\left(\Omega, 2^{\Omega}, Pr\right)$ называется *вероятностным пространством*, если:

- $0 \le Pr(A) \le 1$ для любого $A \in 2^{\Omega}$;
- $Pr(\varnothing) = 0$ и $Pr(\Omega) = 1$ (нормировка вероятности);
- $Pr\left(\cup_{k \ge 1} A_k\right) = \Sigma_{k \ge 1} Pr(A_k)$, если события A_k попарно не пересекаются (*аддитивность вероятности*).

Вероятностные структуры используются как конечные модели реагирующих систем, которые функционируют бесконечно. Для анализа таких структур следует определить вероятностное пространство, позволяющее выполнить анализ множеств бесконечных вычислений в вероятностных структурах. Пусть задана вероятностная структура M с вероятностной матрицей $\mathbf{P}$. Введем понятие *цилиндра* (или *конуса*).

> ❖ ***Определение 11.3***
>
> Цилиндром конечной цепочки $\sigma = s_0 s_1 s_2 \ldots s_n$ состояний вероятностной структуры M $\left(\sigma \in Path\, s_{fin}(M)\right)$ назовем все такие возможные бесконечные вычисления M, префикс которых совпадает с σ:
>
> $$Cyl(\sigma) = \left\{\pi \in Paths(M) \mid \sigma \in pref(\pi)\right\}$$

На рис. 11.1, *в* выделен цилиндр цепочки $s_0 s_3 s_2$:

$$Cyl(s_0 s_3 s_2) = \left\{s_0 s_3 s_2 s_3 s_2 (s_3 s_2)^{\omega} \ldots, s_0 s_3 s_2 s_1 (s_0 s_3 s_2 s_1)^{\omega}, \ldots\right\}$$

Каждый цилиндр вероятностной структуры M играет роль случайного события сигма-алгебры, ассоциированной с M. Два события этой сигма-алгебры независимы, если соответствующие цилиндры не пересекаются. Множество всех возможных событий будет здесь представлено всеми возможными конечными префиксами вычислений структуры M. Вероятностная матрица $\mathbf{P}$

структуры M для каждого конечного префикса $\in Paths_{fin}(M)$ вычислений этой структуры, $\sigma = s_0 s_1 s_2 \ldots s_n$, позволяет определить его вероятностную меру $Pr(\sigma)$ так:

$Pr(\sigma) = 1$, если префикс σ состоит из единственного состояния s_0,

$Pr(\sigma) = \mathbf{P}(s_0, s_1)\mathbf{P}(s_1, s_2)\ldots\mathbf{P}(s_{n-1}, s_n)$, если $n > 1$.

Пример 11.2

На рис. 11.1, *в* показан начальный фрагмент развертки вычислений структуры M_1 из начального состояния s_0. Вероятностная мера отрезка $s_0 s_3 s_2$ равна $\mathbf{P}(s_0, s_3) \times \mathbf{P}(s_3, s_2) = 0.2 \times 1.0 = 0.2$. Вероятностная мера отрезка $s_0 s_1 s_4$: $Pr(s_0 s_1 s_4) = 0.5 \times 0.2 = 0.1$.

Итак, помеченную вероятностную структуру можно рассматривать как расширение структуры Крипке, на переходах которой заданы вероятности этих переходов. Для состояний такой структуры можно ставить следующий вопрос: "с какой вероятностью в данном состоянии выполняется некоторая формула пути темпоральной логики"?

Обозначим $\mathcal{Pr}(s, \varphi)$ *вероятность* того, что в состоянии s заданной вероятностной структуры выполняется формула φ темпоральной логики. Рассмотрим сначала только формулы вида $\mathbf{X}p$ и $p\mathbf{U}q$, где p и q — атомарные предикаты.

Для состояния s_0, например, вероятность выполнения формулы пути $\mathbf{X}p$ равна 0, поскольку из s_0 нет переходов в состояние, в котором выполняется p. Вероятность выполнения формулы $\mathbf{X}q$ в состоянии s_0 равна сумме вероятностей переходов в состояния s_3 и s_1, т. е. 0.7 (см. рис. 11.1, *в*). Для того чтобы определить $\mathcal{Pr}(s, \mathbf{X}p)$ — вероятность того, что атомарный предикат p выполняется в состоянии, в которое есть переход из состояния s, нужно просто сложить вероятности переходов из состояния s во все такие состояния, в которых выполняется p. Из примера структуры M_1 (см. рис. 11.1, *в*) видно, что $\mathcal{Pr}(s_0, \mathbf{X}p) = 0$, $\mathcal{Pr}(s_0, \mathbf{X}q) = 0.7$, $\mathcal{Pr}(s_1, \mathbf{X}p) = 0.4$.

Пусть $Sat(s, p)$ — функция, равная 1, если p истинно в s, в противном случае $Sat(s, p) = 0$. Тогда вероятность $\mathcal{Pr}(s_0, \mathbf{X}q)$ определится так:

$\mathcal{Pr}(s, \mathbf{X}q) = \Sigma_{s' \in S} \mathbf{P}(s, s') \times Sat(s', p)$.

Более формально, пусть $\mathbf{Sat}(p)$ есть вектор, $\mathbf{Sat}(p): S \to \{0,1\}$ определяемая так: $\mathbf{Sat}_s(p) = 1$, если p истинно в s, в противном случае $\mathbf{Sat}_s(p) = 0$. Тогда вектор $\mathcal{Pr}(s_0, \mathbf{X}q)$ равен произведению матрицы $\mathbf{P}$ на вектор-столбец $\mathbf{Sat}(p)$:

$$\mathcal{Pr}(s_0, \mathbf{X}q) = \mathbf{P} \times \mathbf{Sat}(p)$$

Для примера вероятностной структуры M_1 (рис. 11.1) вектор $\mathcal{Pr}(\mathbf{X}q)$ строится так:

$$\begin{matrix} 0 & 0.5 & 0 & 0.2 & 0.3 \\ 0.4 & 0 & 0.4 & -- & 0.2 \\ 0 & 0.3 & 0 & 0.7 & 0 \\ 0 & 0 & 1.0 & 0 & 0 \\ 0.2 & 0 & 0 & 0.8 & 0 \end{matrix} \times \begin{matrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{matrix} = < 0.7;\ 0.4;\ 1.0;\ 0.0;\ 1.0>$$

Обратимся теперь к формуле $q\mathbf{U}p$. Рассмотрим ее анализ на примере.

Пример 11.3

Из рис. 11.1, *в* видно, что можно непосредственно подсчитать значения вероятностей выполнения формулы $q\mathbf{U}p$ в двух случаях:

- в тех состояниях, в которых выполняется p, вероятность выполнения формулы $q\mathbf{U}p$ равна 1;
- в тех состояниях, в которых не выполняются ни p, ни q, вероятность выполнения формулы $q\mathbf{U}p$ равна 0.

Поэтому $\mathcal{Pr}(s_2, q\mathbf{U}p) = 1$, $\mathcal{Pr}(s_4, q\mathbf{U}p) = 0$. Поскольку в состоянии s_3 выполняется q, и с вероятностью 1 из s_3 попадаем в состояние s_2, в котором эта вероятность равна 1, то $\mathcal{Pr}(s_3, q\mathbf{U}p) = 1$.

В состоянии s_0 вероятность выполнения формулы $q\mathbf{U}p$ зависит от подобной вероятности в состоянии s_1, и наоборот. Поэтому по рис. 11.1, *а* можем написать связывающие их уравнения:

$$\mathcal{Pr}(s_0, q\mathbf{U}p) = 0.2 \times \mathcal{Pr}(s_3, q\mathbf{U}p) + 0.5 \times \mathcal{Pr}(s_1, q\mathbf{U}p) + 0.3 \times \mathcal{Pr}(s_4, q\mathbf{U}p).$$

$$\mathcal{Pr}(s_1, q\mathbf{U}p) = 0.4 \times \mathcal{Pr}(s_0, q\mathbf{U}p) + 0.4 \times \mathcal{Pr}(s_2, q\mathbf{U}p) + 0.2 \times \mathcal{Pr}(s_4, q\mathbf{U}p).$$

Решая эту систему линейных уравнений, получим

$$\mathcal{Pr}(s_0, q\mathbf{U}p) = 0,5,\ \mathcal{Pr}(s_1, q\mathbf{U}p) = 0.6.$$

Более формально, определим для вероятностной структуры M три непересекающихся подмножества состояний: S^{no}, S^{yes} и $S^{?}$.

- S^{no} — это множество таких состояний, в которых ни формула p, ни формула q не выполняются, и таких состояний, из которых состояния, помеченные формулой p, вообще не достижимы. В этих состояниях вероятность выполнения формулы $q\mathbf{U}p$ равна 0.

- S^{yes} — это множество таких состояний, в которых вероятность выполнения формулы $q\mathbf{U}p$ равна 1. Это состояния, в которых формула p выполняется.

- $S^{?}$ — это множество таких состояний, в которых формула $q\mathbf{U}p$ может выполниться с некоторой вероятностью из интервала $[0, 1]$. Это те состояния, в которых выполняется только формула q и из которых достижимо хотя бы одно состояние, в котором выполняется формула p.

Вычисление вероятности выполнения утверждения $M, s \models \varphi$ для формулы пути $q\mathbf{U}p$ состоит в вычислении суммы вероятностей всех таких конечных путей вероятностной структуры M, которые начинаются в состоянии s, проходят через состояния, помеченные q и не помеченные p, и заканчиваются в каком-либо состоянии, помеченном p. Вычисление вероятностей выполнения этой формулы в состояниях вероятностной структуры связано с решением системы линейных уравнений.

Система линейных уравнений для подсчета вероятностей x_s выполнения формулы $\alpha\mathbf{U}\beta$ в состояниях s строится после определения множеств S^{no}, S^{yes} и $S^{?}$ так:

$x_s = 0$, если $s \in S^{no}$

$x_s = 1$, если $s \in S^{yes}$

$x_s = \Sigma_{s' \in S} P(s, s') \times x_{s'}$, если $s \in S^{?}$

Для решения системы линейных уравнений можно использовать любой численный метод.

Пример 11.4

Проанализируем одну из азартных игр — игру в кости (Крепс). В этой игре существует около 40 различных ставок. Найдем вероятность выигрыша для самой простой из них, ставки "*The Pass Bet*" — "*Проходит*".

Игрок делает свою ставку "*The Pass Bet*", помещая фишки на поле *Pass Line* (рис. 11.2). Игра состоит из двух этапов, называемых *Первый бросок* (*Come out roll*) и *Исходное очко* (*Point*). На первом этапе бросаются две игральные кости и подсчитывается сумма очков, выпавших на обеих гранях. Ставка "*The Pass Bet*" сразу выигрывает, если выпавшая сумма равна 7 или 11, и проигрывает, если в сумме выпадает 2, 3 или 12. Если выпала любая другая сумма очков (4, 5, 6, 8, 9 или 10), то она запоминается как *Исходное очко — Point* (на игровом столе набранное значение отмечается фишкой в соответствующей позиции поля *Point*), и игра переходит ко второму этапу. На втором этапе бросание двух костей продолжается многократно. Ставка "*The Pass Bet*" выигрывает, если значение *Point* выпадет раньше, чем выпадет 7. Как только выпадает сумма очков 7, ставка "*The Pass Bet*" проигрывает.

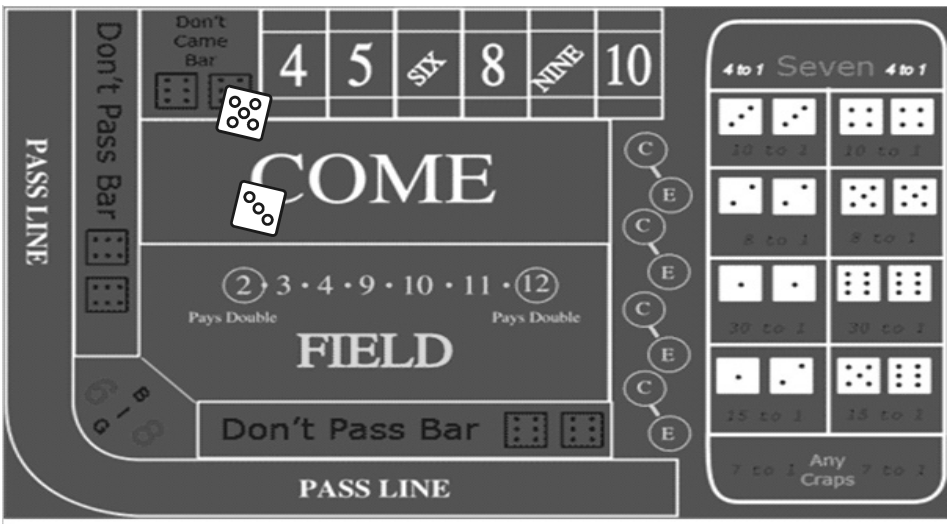


Рис. 11.2. Игровое поле для игры в кости

Для определения вероятности выигрыша по ставке "*The Pass Bet*" построим вероятностную структуру модели игры (рис. 11.3). Из состояния *start* этой структуры можно непосредственно попасть в состояние *won* (выигрыш) или *loose* (проигрыш), что соответствует выигрышу или проигрышу на первом этапе. Из состояния *start* также можно попасть в промежуточные состояния $4,5,6,8,9,10$ (они нумеруются суммой очков, не приводящих к выигрышу или

проигрышу на первом этапе). В этих промежуточных состояниях существуют переходы в состояние *won* (если в одном из повторных бросков выпадет то же очко), в состояние *loose* (если в одном из повторных бросков выпадет 7) и возврат в это же состояние при всех других исходах.

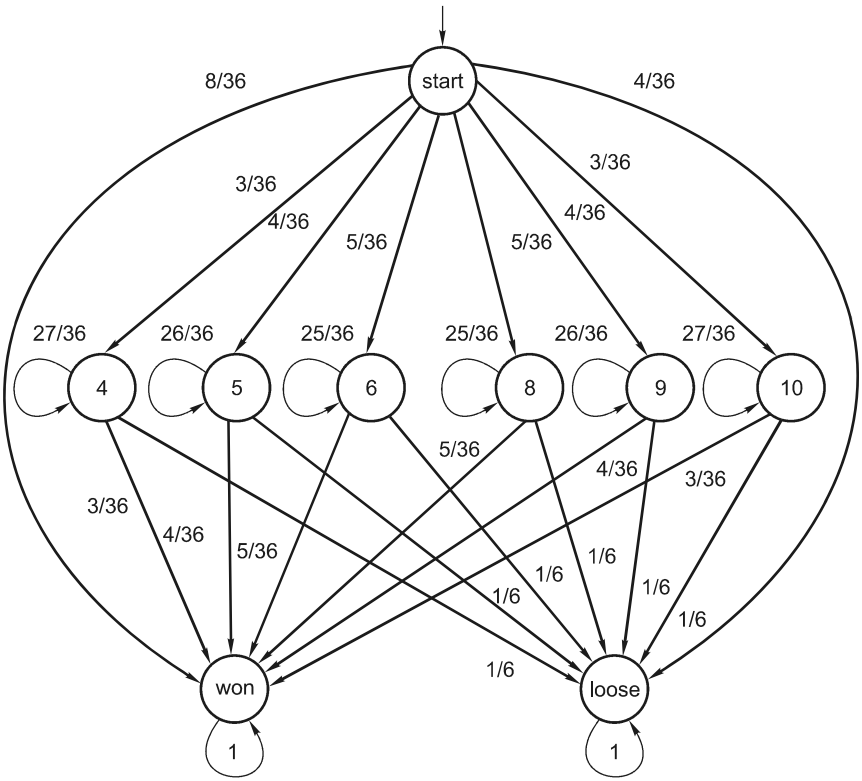


Рис. 11.3. Вероятностная структура для анализа игры в кости

Переходы структуры помечены вероятностями соответствующих событий. Вероятности определяются числом благоприятных исходов из множества всех возможных исходов на каждом шаге игры в каждом из состояний. Например, выпадение 2 при бросании двух костей возможно только при одном исходе — выпадении 1 у первой кости и 1 у второй. Сумма 3 может выпасть при двух исходах (1:2 и 2:1), число 5 выпадает при 4 исходах (1:4; 2:3; 3:2; 4:1) и т. д.

Благоприятные исходы из возможных 36 исходов:

2 ⇔1:1	число благоприятных	1
3 ⇔1:2; 2:1		2

4	⇔1:3; 2:2; 3:1	3
5	⇔1:4; 2:3; 3:2; 1:4	4
6	⇔1:5; 2:4; 3:3; 4:2; 5:1	5
7	⇔1:6; 2:5; 3:4; 4:3; 5:2; 6:1	6
8	⇔2:6; 3:5; 4:4; 5:3; 6:2	5
9	⇔3:6; 4:5; 5:4; 6:3	4
10	⇔4:6; 5:5; 6:4	3
11	⇔5:6; 6:5	2
12	⇔6:6	1

Переход из состояния *start* в состояние *won* помечен вероятностью 8/36, потому что на первом этапе выигрыш дают выпавшие суммы 7 или 11, что возможно при 8 благоприятных исходах из общего числа исходов 36. Переход из состояния *start* в состояние *loose* помечен вероятностью 4/36, потому что выпадение на первом этапе сумм 2, 3 или 12, приводящее к проигрышу, возможно при 4 благоприятных исходах из 36. Подобным образом подсчитываются вероятности всех переходов структуры.

Задачей нашего анализа является подсчет вероятности выигрыша, т. е. вероятности того, что из начального состояния существует путь в выигрышное состояние. Иными словами, для этой вероятностной структуры нужно определить вероятность темпоральной формулы пути $\mathbf{F}@won$, или, что то же самое, формулы пути $true\mathbf{U}@won$.

Определим множества S^{no}, S^{yes} и $S^{?}$:

$$S^{no}=\{loose\},\ S^{yes}=\{won\},\ S^{?}=S\setminus\left(S^{no}\cup S^{yes}\right)=\{start,\ 4,\ 5,6,\ 8,\ 9,\ 10\}$$

Здесь, несмотря на то, что формула *true* выполняется во всех состояниях структуры, состояние *loose* включено в множество S^{no}, поскольку из этого состояния недостижимо состояние, помеченное *won*.

Обозначим x_i вероятность выигрыша из состояния i. Очевидно, что x_i — это вероятность выполнения в состоянии i темпоральной формулы $\mathbf{F}@won=true\mathbf{U}@won$. Составим систему уравнений для вероятностей x_i, учитывая, что $x_{won}=1.0$, а $x_{loose}=0$:

$$x_{start}=8/36+3/36x_4+4/36x_5+5/36x_6+5/36x_8+4/36x_9+3/36x_{10}$$

$$x_4=27/36x_4+3/36$$

$$x_5 = 26/36x_5 + 4/36$$

$$x_6 = 25/36x_6 + 5/36$$

$$x_8 = 25/36x_8 + 5/36$$

$$x_9 = 26/36x_9 + 4/36$$

$$x_{10} = 27/36x_{10} + 3/36$$

Решение системы легко находится вручную. Результат $x_{start} = 244/495$ определяет вероятность перехода из начального состояния *start* в состояние *won*. Это значит, что шансы [выиграть : проиграть] на ставке "*The Pass Bet*" без мошенничества и при идеальных игральных костях составляют [244 : 251], т. е. выигрыш казино в среднем больше выигрыша игрока на этой ставке приблизительно на 3 %.

Очевидна ограниченность описанной здесь техники: в каждом состоянии вероятностной структуры мы можем подсчитать вероятность только простейших формул пути, подформулы которых — атомарные предикаты. В результате этих вычислений получаются значения вероятностей выполнения этих формул, а не логические значения "*истина*" или "*ложь*" выполнимости темпоральной формулы в конкретном состоянии, что является основой алгоритмов маркировки, которые используются в анализе выполнимости формул темпоральной логики CTL на обычной структуре Крипке. Поэтому использовать эти численные результаты далее при проверке выполнимости сложной темпоральной формулы CTL невозможно.

Очевидным решением здесь является требование включить в саму темпоральную формулу пороговое значение вероятности выполнения формулы в конкретном состоянии. При анализе можно сравнить численное значение подсчитанной вероятности выполнения формулы в данном состоянии с этим пороговым значением. В результате такого сравнения в каждом состоянии получатся истинностные значения "*истина*" или "*ложь*", показывающие, удовлетворяется или нет темпоральная вероятностная формула в этом состоянии. Полученные истинностные результаты могут быть использованы для дальнейшего анализа.

Пример 11.5

Запишем найденные в примере 11.5 вероятности $\mathcal{Pr}(s_i, q\mathbf{U}p)$ для всех состояний вероятностной структуры M_1 (см. рис. 11.1):

$\mathcal{Pr}(s_0, q\mathbf{U}p) = 0.5$, $\mathcal{Pr}(s_1, q\mathbf{U}p) = 0.6$, $\mathcal{Pr}(s_2, q\mathbf{U}p) = 1$, $\mathcal{Pr}(s_3, q\mathbf{U}p) = 1$, $\mathcal{Pr}(s_4, q\mathbf{U}p) = 0$.

Выберем в качестве порога выполнения формулы $q\mathbf{U}p$ в нашем примере значение ≥0.6. Пусть запись $\mathscr{P}_{\geq 0.6}(s, q\mathbf{U}p)$ обозначает следующее утверждение: "с вероятностью ≥0.6 формула пути $q\mathbf{U}p$ выполняется в состоянии s ". Такое утверждение может быть либо истинным, либо ложным. Для структуры M_1:

$\mathscr{P}_{\geq 0.6}(s_0, q\mathbf{U}p) = false$ — в s_0 формула $q\mathbf{U}p$ *не* выполняется с вероятностью ≥0.6;

$\mathscr{P}_{\geq 0.6}(s_1, q\mathbf{U}p) = true$ — в s_1 формула $q\mathbf{U}p$ выполняется с вероятностью ≥0.6.

В общем виде формула $\mathscr{P}_{\sim\rho}(\alpha)$, где

$\rho \in [0,1]$ — некоторая вероятность,

$'\sim' \in \{>, <, \geq, \leq\}$ — знак отношения,

α — темпоральная формула пути.

может рассматриваться как формула новой темпоральной логики для текущего состояния. Такая логика, которая носит название "*Вероятностной логикой ветвящегося времени*", или PCTL (Probabilistic CTL), является одним из возможных расширений обычной логики CTL. Определим эту логику формально.

11.2. Вероятностная логика ветвящегося времени (Probabilistic CTL, PCTL) и верификация вероятностных моделей

Формулы логики PCTL являются формулами состояний и имеют следующий вид:

$$\varphi ::= q \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \mathscr{P}_{\sim\rho}(\alpha),$$

где $\sim \in \{<, \leq, =, \geq, >\}$, а формула α — формула пути:

$$\alpha ::= \mathbf{X}\varphi \mid \varphi_1 \mathbf{U} \varphi_2 .$$

Как и в логике LTL, операторы $\mathbf{F}$ и $\mathbf{G}$ выражаются через Until:

$$\mathbf{F}\varphi \equiv True\mathbf{U}\varphi$$

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

Смысл этого определения такой: в состоянии s структуры M вероятностная мера $\sim\rho$ выполняется для формулы пути α тогда и только тогда, когда с этой мерой будет выбран такой путь из состояния s, на котором выполняется формула α. Подобно тому, как в логике CTL каждый темпоральный оператор **X**, **U**, **F** или **G** предварен квантором пути **A** (всеобщности) или **E** (существования), в логике PCTL каждый темпоральный оператор предварен вероятностным оператором $\mathcal{P}_{\sim\rho}$, который можно назвать вероятностным "*квантором*". В PCTL нет кванторов путей, которые используются в логике CTL: вместо формулы CTL $\mathbf{A}\alpha$ с квантором всеобщности (выполнение формулы α на всех путях) может быть использована формула PCTL $\mathcal{P}_{=1}(\alpha)$. Семантика формулы $\mathcal{P}_{=1}(\alpha)$ такова, что формула α выполняется "почти всегда". Формулу PCTL $\mathcal{P}_{>0}(\alpha)$ часто можно использовать вместо формулы $\mathbf{E}\alpha$ с квантором существования. Семантика формулы $\mathcal{P}_{>0}(\alpha)$ следующая: с некоторой, не равной 0 вероятностью из текущего состояния может быть выбран путь, на котором формула пути α выполняется.

Однако формулы $\mathbf{A}\alpha$ и $\mathcal{P}_{=1}(\alpha)$ не эквивалентны. Покажем это на простом примере.

Пример 11.6

Рассмотрим пример вероятностной структуры рис. 11.4. Подсчитаем вероятность выполнения формулы $\mathbf{F}a$ для состояний этой структуры. Множества S^{no}, S^{yes} и $S^{?}$ здесь таковы: $S^{no}=\varnothing$, $S^{yes}=\{t\}$, и $S^{?}=\{s\}$. Система уравнений:

$$x_s=(1-p)x_s+p+x_t$$

$$x_t=1$$

дает $x_s=1$. Таким образом, при любых значениях p, не равных 0, для состояния s этой структуры выполняется формула $\mathcal{P}_{=1}(\mathbf{F}a)$. В то же время существует путь s^{ω} — бесконечный цикл в состоянии s, на котором никогда

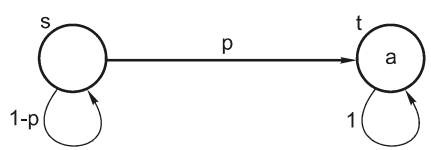


Рис. 11.4. Пример вероятностной структуры

не наступит a. Следовательно, формула $\mathbf{AF}a$ в состоянии s этой структуры не выполняется. Отсюда следует, что формулы $\mathbf{AF}\alpha$ и $\mathscr{P}_{=1}(\mathbf{F}a)$ не эквивалентны.

По свойству дуальности можно утверждать, что формулы $\mathbf{EG}\alpha$ и $\mathscr{P}_{>0}(\mathbf{G}a)$ также не эквивалентны. Действительно, на той же вероятностной структуре рис. 11.4 формула $\mathbf{EG}\neg a$ в состоянии s выполняется, а формула $\mathscr{P}_{>0}(\mathbf{G}\neg a)$ в этом состоянии не выполняется.

Построим алгоритмы вычисления истинности формул логики PCTL. Формулы PCTL интерпретируются над помеченными вероятностными структурами. Пусть $M=(S,s_0,\mathbf{P},AP,L)$ такая структура (см. определение 11.1). Семантика операторов PCTL такова:

$$M,s\models q \quad iff \quad q\in L(s)$$

$$M,s\models \neg\varphi \quad iff \quad M,s\not\models \varphi$$

$$M,s\models \varphi_1\vee\varphi_2 \quad iff \quad M,s\models \varphi_1 \text{ или } M,s\models \varphi_2.$$

$$M,s\models \mathscr{P}_{\sim\rho}(\alpha) \quad iff \quad Pr\left(\forall\pi\in Paths_{M,s}\,|\pi|=\alpha\right)\sim\rho$$

Справедливость трех первых соотношений очевидна. Смысл последнего соотношения следующий. В вероятностной структуре M, находящейся в состоянии s, будет выполняться формула $\mathscr{P}_{\sim\rho}(\alpha)$ тогда и только тогда, когда вероятность того, что на вычислениях, начинающихся в s, формула пути α будет удовлетворять требованию $\sim\rho$.

Например:

- $\mathscr{P}_{\leq 0.1}(\mathbf{X}\beta)$ — вероятность того, что из данного состояния существует переход в состояние, в котором выполняется β, не больше 0.1;
- $\mathscr{P}_{\geq 0.7}(q\mathbf{U}\neg r)$ — вероятность того, что на путях из данного состояния выполнится формула пути $(q\mathbf{U}\neg r)$, больше 0.7;
- $\mathscr{P}_{\leq 0.3}(\mathscr{P}_{\geq 0.2}\mathbf{X}q)\mathbf{U}\neg r$ — вероятность того, что на путях из данного состояния выполнится формула пути $(\mathscr{P}_{\geq 0.2}\mathbf{X}q)\mathbf{U}\neg r$, меньше, чем 0.3.

Алгоритм верификации для PCTL работает так же, как для CTL:

- с помощью синтаксического анализа определяются подформулы формулы φ;

- синтаксическое дерево формулы φ логики PCTL проходится снизу вверх;
- для каждой подформулы состояния ψ формулы φ определяется множество $Sat(\psi)$ тех состояний, которые удовлетворяют формуле состояния ψ;
- если $\psi = \mathcal{P}_{\sim\rho}(\alpha)$, то вероятности всех тех путей из состояния s, на которых формула пути α истинна, вычисляются и сравниваются с вероятностной границей $\sim\rho$, давая в результате для формулы ψ в конкретном состоянии s значение *true* или *false*.

Как и в логике CTL, будем считать, что темпоральная формула φ логики PCTL выполняется для всей структуры тогда и только тогда, когда φ выполняется в начальном состоянии структуры.

Обозначим $Sat(\varphi): S \to \{0,1\}$ — вектор, который для состояния $s \in S$ дает истинностное значение 1 (*true*), если $M, s \models \varphi$, и 0 (*false*) в противном случае. Тогда:

$$Sat\left(\mathcal{P}_{\sim\rho}(\mathbf{X}\varphi)\right) = \{\forall s \in S \mid x_s \sim \rho\}, \text{ где } x_s = \mathbf{P} \times Sat(\varphi).$$

Пример 11.7

Вычислим $\mathbf{Sat}\left(\mathcal{P}_{\geq 0.6}(\mathbf{X}q)\right)$ для вероятностной структуры M_1 примера 11.1, т. е. вектор истинностных значений для формулы $\mathcal{P}_{\geq 0.6}(\mathbf{X}q)$ на этой структуре. На рис. 11.5 показано, что для этого необходимо матрицу **P** вероятностей переходов структуры M_1 умножить на вектор $\mathbf{Sat}(q)$, т. е. на вектор, определяющий для каждого состояния, выполняется формула q в этом состоянии или нет. В результате получаем вектор

$$\mathcal{P}r(\mathbf{X}q) = \langle 0.7; 0.4; 1.0; 0.0; 1.0 \rangle$$

вероятностей выполнения формулы $\mathbf{X}q$ на состояниях структуры M_1. Формула состояний $\mathcal{P}_{\geq 0.6}(\mathbf{X}q)$ удовлетворяется в структуре M_1 в состояниях s_0, s_2 и s_4:

$$\mathcal{P}_{\geq 0.6}(\mathbf{X}q) = \langle 1; 0; 1; 0; 1 \rangle$$

Вычисление вектора **Sat** для формулы пути $\mathcal{P}_{\sim p}(q\mathbf{U}p)$ выполняется с помощью линейных уравнений, составленных на основе определенных выше множеств S^{no}, S^{yes} и $S^{?}$.

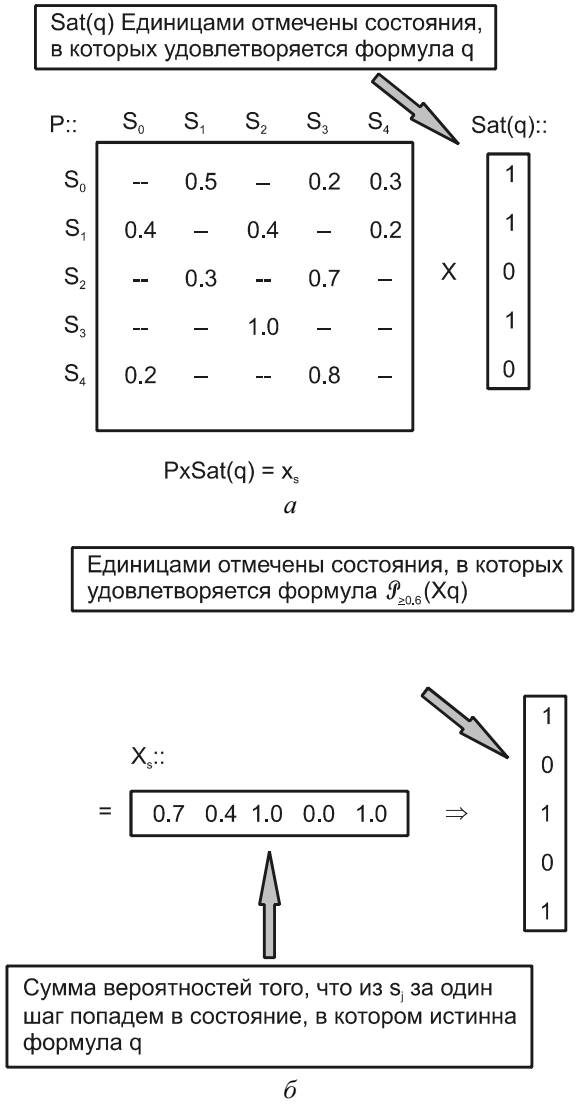


Рис. 11.5. Определение множества состояний, удовлетворяющих формуле $\mathscr{R}_{\geq 0.6}(\mathbf{X}q)$

Пример 11.8

Вычислим $\mathbf{Sat}\left(\mathscr{R}_{\geq 0.6}(q\mathbf{U}p)\right)$ для помеченной вероятностной структуры M_1 примера 11.2, т. е. построим для этой структуры вектор истинностных значений для формулы $\mathscr{R}_{\geq 0.6}(q\mathbf{U}p)$.

Множества S^{no}, S^{yes} и $S^{?}$ определяются по выполнению в них формул q и p: $S^{no}=\{s_4\}$, $S^{yes}=\{s_2\}$ и $S^{?}=S\setminus\left(S^{no}\cup S^{yes}\right)=\{s_0,s_1,s_3\}$.

Система уравнений:

$$x_0 = 0.5\,x_1 + 0.2\,x_3 + 0.3\,x_4$$

$$x_1 = 0.4\,x_2 + 0.2\,x_4 + 0.4\,x_0$$

$$x_2 = 1$$

$$x_3 = 1.0\,x_2$$

$$x_4 = 0$$

Решение системы — вектор вероятностей выполнения формулы $q\mathbf{U}p$ в этой структуре, имеет вид: $\langle x_0=0.5,\ x_1=0.6,\ x_2=1,\ x_3=1,\ x_4=0\rangle$. Вектор $\mathbf{Sat}\left(\mathscr{P}_{\geq 0.6}\left(q\mathbf{U}p\right)\right)=\langle 0,\,1,\,1,\,1,\,0\rangle$ показывает, что формула состояний $\mathscr{P}_{\geq 0.6}\left(q\mathbf{U}p\right)$ удовлетворяется в состояниях s_1, s_2 и s_3. Мы получили те же результаты, что и в примере 11.6, однако здесь эти результаты получены при выполнении описанного выше алгоритма model checking для вероятностных структур.

Сложность алгоритма верификации формул логики PCTL линейна по числу подформул проверяемой формулы и полиномиальна относительно числа состояний вероятностной структуры.

11.3. Проверка свойств, зависящих от дискретного времени и вероятностного выбора

Вероятностная структура, используемая в качестве модели для интерпретации формул вероятностной темпоральной логики PCTL, может быть использована для проведения совместного анализа не только поведенческих и вероятностных, но также и временных свойств систем, если каждый переход системы связывает состояния, отстоящие в дискретном времени на один шаг. Тогда временной анализ состоит в подсчете количества шагов между событиями и может быть использован для таких синхронных систем. Расширим возможные формулы пути вероятностной темпоральной логики PCTL еще одной конструкцией:

$$\varphi ::= q \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \mathscr{P}_{\sim\rho}\left(\alpha\right),$$

где $\sim \in \{<, \leq, =, \geq, >\}$, а формула α — формула пути:

$$\alpha ::= \mathbf{X}\varphi \mid \varphi_1 \mathbf{U} \varphi_2 \mid \varphi_1 \mathbf{U}^{\leq n} \varphi_2,$$

где n — натуральное число. Формула пути $\varphi_1 \mathbf{U}^{\leq n} \varphi_2$ с ограниченным оператором Until утверждает, что формула φ_2 будет выполнена на рассматриваемом пути не более чем через n шагов, а до этого во всех состояниях пути будет выполняться формула φ_1. В реальной системе каждый шаг выполнения модели может представлять любой временной интервал: час, секунду или миллисекунду.

Для ограниченного оператора Until имеем:

$$\mathbf{F}^{\leq n} \varphi = true \mathbf{U}^{\leq n} \varphi.$$

Путь удовлетворяет формуле $\mathbf{F}^{\leq n} \varphi$, если на нем состояние, удовлетворяющее φ, может быть достигнуто не более чем за n шагов.

Совместный анализ логических, вероятностных и временных свойств системы дает возможность строго доказывать корректность, например, таких утверждений о поведении систем:

- "После любого отказа, с вероятностью 0,9999, система восстановится не более чем через 5 временных шагов".
- "Вероятность того, что сигнал *ready* будет получен в течение следующих 6 единиц времени, больше 50 %."

Для проведения такого анализа достаточно определить вероятность выполнения темпоральной формулы $\alpha \mathbf{U} \beta$ на ограниченной по числу шагов развертке поведения структуры. Обозначим $\mathcal{Pr}(s, \alpha \mathbf{U} \beta, T)$ вероятность того, что в системе, находящейся в состоянии s, темпоральная формула пути $\alpha \mathbf{U} \beta$ будет выполнена в рамках временного интервала T, т. е. не более чем через T шагов алгоритма будет достигнуто состояние, в котором удовлетворяется формула β по пути, во всех состояниях которого удовлетворяется формула α.

Для подсчета $\mathcal{Pr}(s, \alpha \mathbf{U} \beta, T)$ по вероятностной структуре с вероятностной матрицей $\mathbf{P}$ построим дерево — развертку этой структуры с корнем в вершине s с длиной пути, не более предельного интервала T. Все листья дерева будут помечены 0 или 1 в соответствии с правилом:

```
Pr(s, αUβ, t) =
  if t<0 then 0                          // ограничение по длине траектории
    else if β∈L(s) then 1                // утверждение β выполняется в s
```

```
else if α∉L(s) then 0          // утверждение α не выполняется в s
else ∑ s'∈S P(s, s')× Pr(s', αUβ, t-1)
```

Подсчет $\mathcal{Pr}(s, \alpha \mathbf{U} \beta, t)$ по построенной развертке прост — это сумма вероятностей всех путей из начального состояния в терминальные состояния, помеченные 1.

Пример 11.9

Вычислим вероятность того, что не более чем за три шага вероятностная структура M_1 рис. 11.1, *a*, из начального состояния s_0 может достичь состояния, в котором выполняется p по пути, во всех состояниях которого выполняется q. Иными словами, подсчитаем $\mathcal{Pr}(s_0, q\mathbf{U}p, 3)$ для структуры M_1.

Для подсчета этой вероятности построим развертку структуры M_1 из начального состояния с длиной путей не более 3. Это дерево представлено на рис. 11.6. Каждый путь может оборваться раньше в состояниях, которым приписывается конкретное значение вероятности выполнения формулы в этом состоянии: либо 0, либо 1. Значение вероятности 1 приписывается состояниям множества S^{yes} (в нашем примере помеченным атомарным пре-

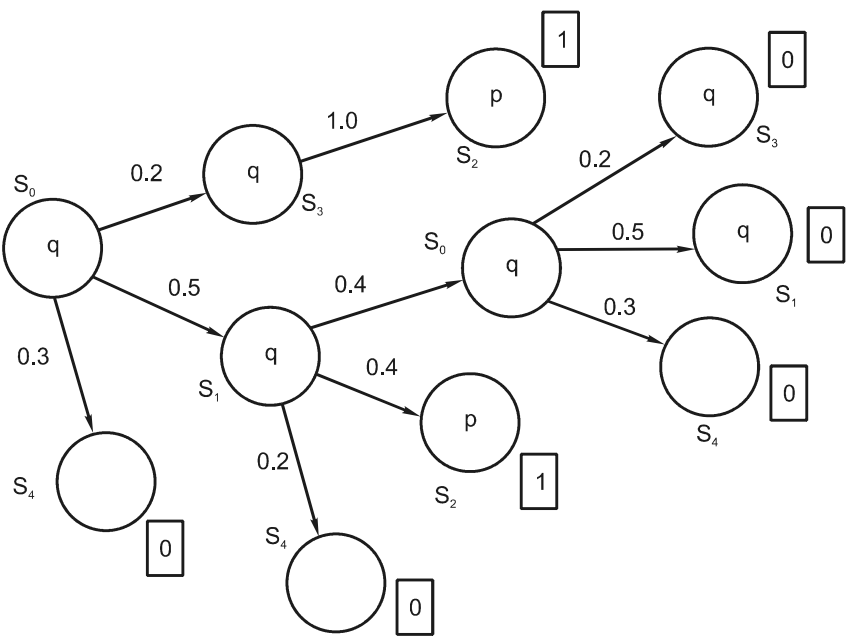


Рис. 11.6. Развертка вероятностной структуры по шагам времени

дикатом p). В примере это состояние s_2. Значение вероятности 0 приписывается состояниям множества S^{no} (т. е. таким, которые не помечены ни p, ни q, и таким, из которых недостижимы состояния, помеченные p). В примере это состояние s_4. Это же значение 0 приписывается тем состояниям множества $S^{?}$, которые отстоят ровно на t шагов от корня. В примере это листья $\{s_0, s_1, s_3\}$. Таким образом, все листья этого дерева помечены вероятностями либо 0, либо 1.

Вычисление искомой вероятности по построенному дереву состоит в подсчете суммы вероятностей тех путей, которые заканчиваются вершиной с приписанным ей значением вероятности 1. На рис. 11.6 таких путей два: $s_0 s_1 s_2$ и $s_0 s_3 s_2$. Сумма вероятностей этих путей 0.4. Следовательно, вероятность того, что в состоянии s_0 выполняется формула пути $q\mathbf{U}p$ не более чем за три временных шага, равна 0.4.

Вместо построения дерева удобно строить таблицу вероятностей выполнения формулы $\mathcal{P}_r(s, \alpha \mathbf{U} \beta, t)$ для всех s при различных значениях t, начиная с $t = 0$. Для этого используем алгоритм, вытекающий из вышеприведенного правила:

Алгоритм 11.1. Вычисление вероятностей $\mathcal{P}_r(s, \alpha \mathbf{U} \beta, T)$

```
00 begin
01        // вычисление вероятностей для t=0
02 for all s∈S do
03  if β∈L(s) then Pr(s, αUβ, 0) := 1        // β выполняется в s
04            else Pr(s, αUβ, 0) := 0;       // β не выполняется в s
05 od
06        // вычисление вероятностей для всех t>0
07  for t := 1 to T do
08   for all s∈S do
09    if β∈L(s) then P(s, αUβ, t) := 1       // β выполняется в s
10     else begin
11          Pr(s, αUβ, t) := 0;              // если α не выполняется в s,
                                                 то Pr =0
12          if α∈L(s) then
13           Pr(s, αUβ, t) = ∑s'∈S P(s, s')×Pr(s', αUβ, t-1);
14     end
```

```
15    od
16   od
17 end
```

При выполнении этого алгоритма удобно строить таблицу для всех состояний вероятностной структуры и для всех значений t от 0 до T.

Пример 11.10

На рис. 11.7 приведена вероятностная структура простого протокола передачи сообщений. Из начального состояния *init* протокол переходит в состояние *try*, находясь в котором он пытается передать сообщение. С вероятностью 0.2 канал оказывается занятым, с вероятностью 0.7 канал свободен и сообщение успешно передается, с вероятностью 0.1 переданное в канал сообщение искажается, и передача этого сообщения повторяется. Будем считать, что каждый шаг алгоритма требует 1 единицы времени.

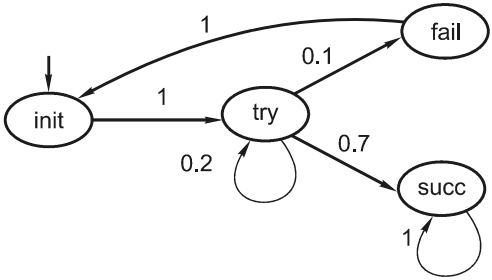


Рис. 11.7. Модель простого протокола

Проверим, выполняется ли для этого протокола следующее утверждение:

> "Сообщение будет успешно доставлено в течение не более чем 6 единиц времени с вероятностью, большей 0.95".

Иными словами, проверим, выполняется ли для этого протокола (т. е. в его начальном состоянии *init*) формула PCTL:

$$\mathcal{P}_{>0.95}\left(\mathbf{F}^{\le 6}\, @succ\right).$$

Для этого подсчитаем вероятность $\mathcal{P}r\left(init, true\mathbf{U}@succ, 6\right)$ по алгоритму 11.1. Построим таблицу подсчета $\mathcal{P}r\left(s, true\mathbf{U}@succ, t\right)$ для всех состояний структуры рис. 11.6 и всех t от $t=0$ до $t=6$.

При $t=0$ значение вероятности 1 стоит только в состоянии *succ*, в остальных состояниях вероятность того, что за 0 шагов из них можно достичь со-

стояния *succ*, равна 0. При всех значениях t искомая вероятность в состоянии *succ* также равна 1 (строка 09 алгоритма 11.1). Поскольку в нашем примере $\alpha = true$, то при всех значениях $t > 0$ искомая вероятность во всех других состояниях s определяется по формуле (строка 13 алгоритма 11.1):

$$\mathcal{Pr}(s, true\ \mathbf{U} @ succ, t) = \Sigma_{s' \in S} \mathbf{P}(s, s') \times \mathcal{Pr}(s', true\ \mathbf{U} @ succ, t-1)$$

(здесь $\mathbf{P}(s, s')$ — элемент вероятностной матрицы, т. е. вероятность перехода из состояния s в s'). Последовательный подсчет этих вероятностей дает табл. 11.1.

Таблица 11.1. Последовательный подсчет вероятностей перехода из s в s'

	t=0	t=1	t=2	t=3	t=4	t=5	t=6
init	0	0	0.7	0.84	0.868	0.944	0.971
try	0	0.7	0.84	0.868	0.944	0.971	0.981
succ	1	1	1	1	1	1	1
fail	0	0	0	0.7	0.84	0.868	0.944

Искомая вероятность при $t = 6$ оказалась равной 0.971. Следовательно, утверждение $init \mid P_{>0.95}\left(F^{\leq 6} @ succ\right)$ для этого протокола истинно.

11.4. Примеры свойств, выражаемых в PCTL

Как мы видели выше, формулы логики PCTL отличаются от формул CTL тем, что по каждой формуле пути φ в каждом состоянии вероятностной структуры можно вычислить вероятность того, что эта формула выполнится. Можно сказать, что логика PCTL — это логика CTL плюс вероятность: $PCTL = CTL + \mathrm{Pr}obability$

Эквивалентность формул PCTL устанавливается по отношению к их выполнению на вероятностных структурах. В частности, две формулы Φ и Ψ языка PCTL будут эквивалентны (записывается $\Phi \equiv \Psi$, если для любой вероятностной структуры M и любого ее состояния s справедливо:

$$M, s \models \Phi \equiv M, s \models \Psi.$$

Например, формула пути α выполняется на всех возможных вычислениях $Paths(M)$ с вероятностью самое меньшее p тогда и только тогда, когда

формула пути $\neg\alpha$ выполняется на этих вычислениях с вероятностью самое большее $1-p$. Отсюда можно определить:

$$\mathscr{P}_{>p}(\alpha) \equiv \mathscr{P}_{\leq 1-p}(\neg\alpha)$$

$$\mathscr{P}_{\geq p}(\alpha) \equiv \mathscr{P}_{<1-p}(\neg\alpha)$$

$$\mathscr{P}_{<p}(\alpha) \equiv \mathscr{P}_{\geq 1-p}(\neg\alpha)$$

$$\mathscr{P}_{\leq p}(\alpha) \equiv \mathscr{P}_{<1-p}(\neg\alpha)$$

Рассмотрим теперь формулу $\mathbf{G}\varphi$. Поскольку $\mathbf{G}\varphi = \neg\mathbf{F}\neg\varphi$, то

$$\mathscr{P}_{\geq p}(\mathbf{G}\varphi) \equiv \mathscr{P}_{\geq p}(\neg\mathbf{F}\neg\varphi) \equiv \mathscr{P}_{<1-p}(\mathbf{F}\neg\varphi).$$

Далее,

$$\mathscr{P}_{\leq p}(\mathbf{G}\varphi) \equiv \mathscr{P}_{>1-p}(\mathbf{F}\neg\varphi),$$

$$\mathscr{P}_{(p,q]}\left(\mathbf{G}^{\leq n}\varphi\right) \equiv \mathscr{P}_{(1-q,1-p]}\left(\mathbf{F}^{\leq n}\neg\varphi\right).$$

Пример 11.11

Проанализируем свойство "мягкого дедлайна" для упрощенной версии протокола альтернирующего бита. Свойство "мягкого дедлайна" устанавливает вероятность наступления события во временных рамках.

Архитектура протокола (рис. 11.8, *а*) содержит три блока: передатчик, приемник и ненадежный канал. В этой версии подтверждение *ack* посылается по надежному каналу и никогда не теряется и не искажается. Модель переходов протокола (рис. 11.8, *б*) является вероятностной структурой: обнаруживаемые потери в канале происходят с вероятностью 20 %. Проверим, будет ли для этого протокола выполняться свойство:

$$\mathbf{AG}\left(@s_0 \Rightarrow \mathscr{P}_{\geq 0.9}\mathbf{F}^{\leq 5}@s_4\right) = \neg\mathbf{EF}\neg\left(@s_0 \Rightarrow \mathscr{P}_{\geq 0.9}\mathbf{F}^{\leq 5}@s_4\right).$$

То есть что не более чем за 5 шагов по времени с вероятностью не ниже 0.9 переданное сообщение будет доставлено. Синтаксический анализ дает следующие подформулы этой формулы:

$$f_1 = @s_0$$

$$f_2 = @s_4$$

$$f_3 = \mathscr{P}_{\geq 0.9}\mathbf{F}^{\leq 5} f_2$$

$$f_4 = f_1 \Rightarrow f_3$$

$f_5 = \neg f_4$

$f_6 = \mathbf{EF} f_5$

$f_7 = \neg f_6$

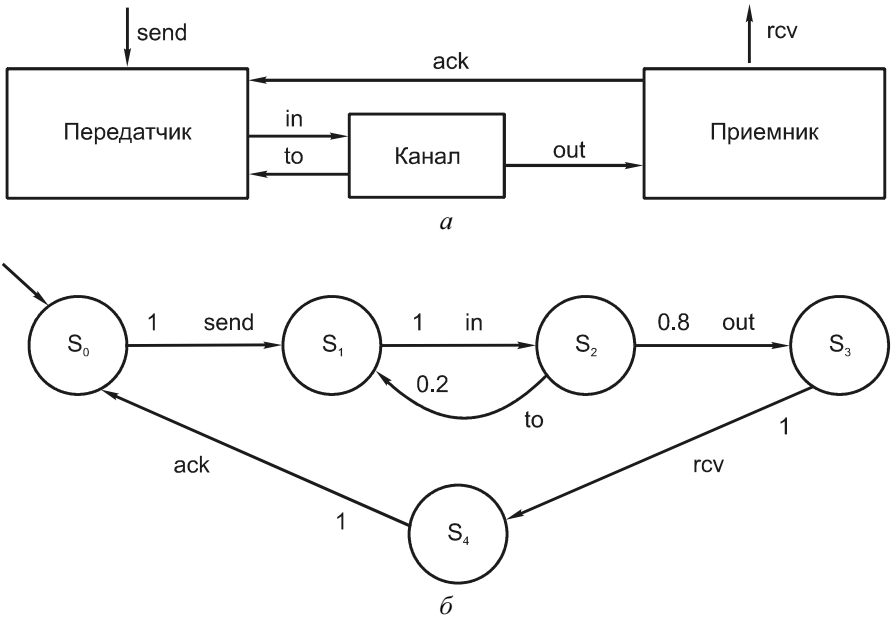


Рис. 11.8. Архитектура упрощенного протокола альтернирующего бита (*а*) и его модель (*б*)

Последовательно помечая состояния модели подформулами, получим, что:

$f_1 = @s_0$ выполняется на множестве состояний $\{s_0\}$,

$f_2 = @s_4$ выполняется на множестве состояний $\{s_4\}$.

Для нахождения множества состояний, на которых выполняется f_3, используем алгоритм 11.1. Построенная табл. 11.2 имеет вид:

Таблица 11.2. Множество состояний, на котором выполняется f_3

	t=0	**t=1**	**t=2**	**t=3**	**t=4**	**t=5**
s_0	0	0	0	0	0	0.8
s_1	0	0	0	0.8	0.8	0.96

Таблица 11.2 *(окончание)*

	t=0	t=1	t=2	t=3	t=4	t=5
s_2	0	0	0.8	0.8	0.96	0.96
s_3	0	1	1	1	1	1
s_4	1	1	1	1	1	1

Отсюда:

$f_3 = \mathscr{P}_{\geq 0,9}\mathbf{F}^{\leq 5} f_2$ выполняется на множестве состояний $\{s_1, s_2, s_3, s_4\}$

$f_4 = f_1 \Rightarrow f_3$ выполняется на множестве состояний $\{s_1, s_2, s_3, s_4\}$

$f_5 = \neg f_4$ выполняется на множестве состояний $\{s_0\}$

$f_6 = \mathbf{EF} f_5$ выполняется на всех состояниях $\{s_0, s_1, s_2, s_3, s_4\}$

$f_7 = \neg f_6$ не выполняется ни на одном состоянии.

Если в таблицу добавить еще один столбец, т. е. увеличить на единицу времени интервал, на котором производится оценка вероятности доставки сообщения, то эта формула окажется выполнимой во всех состояниях модели.

11.5. Заключение

Вероятностную структуру можно рассматривать с двух точек зрения. Во-первых, это расширение дискретной цепи Маркова введением пометок в состояниях — атомарных предикатов, которые истинны в этих состояниях. Но вероятностная структура является также и расширением структуры Крипке введением вероятностной меры на переходах структуры.

Марковские цепи широко используются во многих областях, не только в математике и информатике, но и в исследовании операций, биологии, демографии. Добавление возможности анализа логических утверждений и учета временных ограничений в Марковских цепях открывает новые горизонты для анализа во всех этих областях. С другой стороны, введение в структуру Крипке вероятностных оценок на переходы также позволяет расширить сферу применения классической верификации, выполнять более тонкий анализ свойств поведения дискретных систем. Методы традиционной верификации могут дать ответ, выполняется ли некоторое свойство в системе с вероятностью 0 или 1. Вероятностный и временной анализ позволяют ответить на вопрос, удовлетворяет ли вероятность выполнения некоторого свойства в

системе установленной (нижней или верхней) границе, и будет ли это свойство выполняться в заданном интервале дискретного времени. Таким образом, использование вероятностных структур дает новое качество в каждой из этих областей.

Для вероятностных структур может выполняться как количественный, так и качественный анализ свойств. Количественный анализ позволяет определить вероятности выполнения темпорального свойства или вероятность выполнения свойства в рамках заданного числа временных шагов. Качественный анализ вероятностной структуры позволяет проверить, что некоторое "хорошее" свойство будет выполнено в ней "почти обязательно" или что некоторое "плохое" свойство не случится "почти никогда".

Описанные в этой главе подходы и алгоритмы реализованы в нескольких системах верификации. В частности, можно упомянуть систему верификации PRISM (Probabilistic Symbolic Model Checker), разработанную в Университете г. Бирмингема, Англия, под руководством Марты Квятковски (Marta Kwiatkowska). Эта система верификации распространяется свободно для учебных и исследовательских целей. У этой системы несколько тысяч пользователей. Система PRISM использовалась для анализа надежности распределенных компьютерных систем, проверки криптографических протоколов, в биологии, анализе квантовых вычислений.

11.6. Замечания

Попытки расширить формальный анализ, основанный на темпоральной логике, в направлении анализа количественных параметров, были сделаны в начале 90-х гг. прошлого века [47]. Расширение темпоральной логики для работы с вероятностными моделями было предложено в работе Ханссона и Джонсона (Hans Hansson, Bengt Jonsson) [65]. В этой же работе переходы модели предложено использовать как шаги по времени. Материал данной главы основан на этой работе. Пример 11.4 с анализом игры в кости заимствован из лекций [86].

Эта область находится в стадии интенсивных исследований. Существуют опытные инструментальные системы для анализа количественных свойств поведения вероятностных систем в рамках различных расширений темпоральных логик. Кроме упомянутой выше системы PRISM ([182], [89]), следует упомянуть систему ETMCC, которая описана в [66]. В [64] изложены результаты исследования надежности системы управления космического телескопа Хаббл с помощью системы верификации ETMCC.

11.7. Задачи к главе 11

11.1. Для вероятностной структуры примера 11.1 подсчитайте, выполняется ли на ней формула $\mathscr{P}_{\le 0.3}\left(\mathscr{P}_{>0.2}\mathbf{X}q\right)\mathbf{U}\neg p$.

11.2. Ставка "*The Don't Pass Bet*" при игре в кости (Крепс) ставит на вариант развития событий, противоположный тому, что делается при ставке "*The Pass Bet*" (пример 11.5). А именно, деньги ставятся на то, что игрок в первом броске выбросит 2 или 3. При выпадении 12 засчитывается ничья, ставка возвращается игроку. В случае выпадения 7 или 11 ставка проигрывает. Если игра переходит во второй раунд, ставка выигрывает, если игрок выбрасывает 7 до того, как выпал *пункт*. Проверьте, какая из двух ставок, "*The Pass Bet*" или "*The Don't Pass Bet*", выгоднее игроку.

11.3. Другой вариант ставки "*The Don't Pass Bet*" при игре в кости (Крепс) отличается от того, который описан в задаче 11.2, следующим. Ничья при первом броске засчитывается при выпадении суммы очков 2, а не 12. Чем отличаются условия выигрыша в этих двух вариантах?

11.4. Для вероятностной структуры примера 11.1 подсчитайте, выполняется ли формула $\mathscr{P}_{\le 0.5}\, q\mathbf{U}^{\le 0,4} p$.

11.5. Для вероятностной структуры примера 11.1 подсчитайте, в каком временном интервале t выполняется формула $\mathscr{P}_{\le 0.5}\, q\mathbf{U}^{\le t} p$.

11.6. Для произвольной вероятностной структуры с заданной разметкой вершин атомарными предикатами проверьте, выполняются ли на ней следующие формулы:

а) $\mathscr{P}_{>0.7}\left(q\mathbf{U}^{\le 3}\neg r\right)$

б) $\mathscr{P}_{\le 0.1}\left(q\mathbf{U}\left(\mathscr{P}_{>0.2}\mathbf{X}\neg r\right)\right)$

11.7. Постройте программу, которая по заданным вероятностной структуре M и темпоральной формуле φ логики PCTL определяет, выполняется ли формула φ на структуре M.

11.8. Существует вариант вычисления вероятности $\mathscr{P}\left(s_i, \alpha\mathbf{U}\beta, t\right)$ с помощью умножения матриц. Этот способ вытекает из алгоритма 11.1. Определите

квадратную матрицу A с числом строк и столбцов, равным числу состояний структуры:

$$A\left[s_i, s_j\right] = \textit{if } s_i \in S^{?} \textit{ then } \mathbf{P}\left[i, j\right] \textit{ else if } s_i \in S^{yes} \textit{ and } j = i \textit{ then } 1 \textit{ else } 0$$

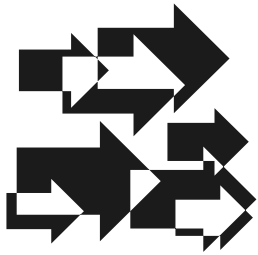
Определим также вектор-столбец C :

$$C\left[s_i\right] = \textit{if } s_i \in S^{yes} \textit{ then } 1 \textit{ else } 0$$

Величина вероятности $\mathcal{Pr}\left(s_i, \alpha\mathbf{U}\beta, t\right)$ определяется как i-й элемент вектора, являющегося результатом умножения t-й степени матрицы A на вектор C.

Постройте матрицу A и вектор C и подсчитайте с помощью этого метода величину $\mathcal{Pr}\left(s_i, \alpha\mathbf{U}\beta, 4\right)$ для задачи 11.1.

ГЛАВА 12



Системы реального времени

Системы реального времени — это такие системы, в которых требуется гарантия выполнения критических операций в ограниченном интервале времени. При прыжке с парашютом нас не очень успокоит, что для него гарантировано свойство: "Если дернуть за кольцо, то когда-нибудь в будущем парашют раскроется". Нам важна гарантия того, что парашют раскроется в заданном интервале времени. Гарантия выполнения временных ограничений в системах реального времени чрезвычайно важна. Превышение временной границы реакции на внешнее событие при работе программы в критических приложениях может привести к катастрофе. Например, во время "войны в заливе" в 1981 г. ошибка в системе управления противоракеты "Patriot" задержала ее запуск всего на полсекунды. Это привело тому, что ракета "Скад" была упущена, от ее взрыва погибли 29 и были ранены 79 американских солдат.

Требования недопустимости нарушения установленной временной границы называются "жесткими" (hard) требованиями реального времени. В отличие от "жестких" требований реального времени, в некоторых системах нарушения временной границы возможны, но они не должны быть слишком частыми (например, доставка почтового сообщения). Такие требования называют "мягкими" требованиями реального времени. Проверка свойств "мягкого" реального времени связана с оценкой вероятности нарушения временных границ, алгоритмы проверки таких свойств рассматривались в *гл. 11*. Свойства "жесткого" реального времени рассматриваются в данной главе.

Для распространения на такие системы методов верификации, объединенных под названием *model checking*, необходимо:

- разработать модель систем реального времени, операционная семантика которых адекватно отражает их поведение во времени;
- расширить темпоральную логику так, чтобы можно было бы формулировать свойства поведения, включающие временные ограничения;

- разработать алгоритмы model checking — алгоритмы проверки формул расширенной темпоральной логики на модели систем реального времени.

Именно этим вопросам посвящена данная глава.

12.1. Модель систем реального времени. Временнòй автомат и его поведение

❖ ***Определение 12.1 (системы реального времени)***

Системы реального времени — это такие системы, которые должны выполнять свои задачи в пределах строгих временных ограничений.

К классу систем реального времени относятся программные и аппаратные системы, применяющиеся в совершенно разных областях. Это протоколы связи, системы управления роботов, бортовые системы летательных аппаратов, встроенные системы медицинской техники и т. п. Как и для других систем, верификация систем реального времени требует формального задания их функционирования и формальной спецификации их свойств. И то и другое должно включать указание на интервалы реального времени между событиями.

Одна из наиболее распространенных формальных моделей для описания систем этого типа называется временным автоматом (Timed Automaton). В этой модели используется простое решение — в модель конечного автомата введены вещественные переменные специального вида — таймеры, или *локальные часы*.

Пример 12.1

Рассмотрим пример простой программы, различающей одинарный и двойной щелчок мыши. Очевидно, что для того, чтобы различить их, программа должна оценить *временной интервал* между последовательными нажатиями на кнопку мыши, и если этот интервал меньше некоторой границы, то считать два последовательных нажатия на клавишу одним двойным щелчком.

На рис. 12.1 представлена автоматная модель такой программы. В автомате A_1 переменная x — это переменная, которая по умолчанию непрерывно изменяется со временем, она моделирует поведение часов (например, аппаратного таймера) и называется локальными часами. С помощью таких переменных можно отсчитывать интервалы времени, прошедшие с момента прихода системы в некоторые состояния. В начальном состоянии s_0 автомат ожидает от окружения события "нажатие кнопки мыши" (действие a). Когда это событие наступает, автомат A_1 переходит в состояние s_1, и при переходе зна-

чение локальных часов x сбрасывается в 0. Когда автомат находится в состоянии s_1, переменная x непрерывно изменяется — увеличивается со временем в одном темпе с реальным временем. Условие $x<1$ на переходе из s_1 в s_2 говорит, что если менее чем через одну единицу времени после момента прихода в состояние s_1 произойдет следующее нажатие кнопки, то автомат перейдет в состояние s_2 и затем при переходе в s_0 выдаст сигнал b, информируя окружение (например, операционную систему) о событии "двойной щелчок". Если прошла в точности одна единица времени после первого нажатия кнопки, то автомат переходит из состояния s_1 в s_0 с выдачей сигнала c, информируя окружение о событии "одинарный щелчок". Неравенства $x>0$, $x<1$ и $x\geq 1$ на переходах называются "*защитами*" переходов.

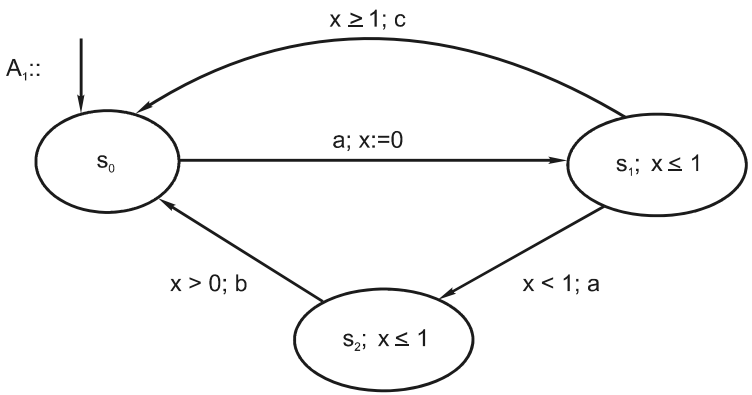


Рис. 12.1. Временной автомат,
различающий одинарный и двойной щелчок мыши

Для того чтобы указать, что автомат не может находиться в состоянии s_1 произвольно долго, это состояние помечено неравенством $x\leq 1$ — так называемым "*инвариантом состояния*". Инвариант определяет условия, при которых автомат может находиться в данном состоянии. Автомат, показанный на рис. 12.1, не может находиться в состоянии s_1 дольше, чем 1 единицу времени. В состоянии s_2 автомат выдаст сигнал b, информирующий операционную систему о том, что произошло событие "двойной щелчок кнопки", не позже, чем часы x примут значение 1. Таким образом, инвариант — это средство "заставить" временной автомат покинуть состояние.

Такое расширение модели конечного автомата удобно, но возникает одна трудность. Переменная x — это переменная специального типа, которая по

умолчанию непрерывно изменяется со временем, поэтому состояния такого автомата уже не являются состояниями в строгом смысле этого слова. Состояние системы — это набор значений всех ее переменных, однозначно определяющий дальнейшее поведение системы. Локальные часы временного автомата являются его переменными, от их значений зависит будущее поведение автомата. Но локальные часы x могут принимать бесконечное число значений (континуум), поэтому у любого временного автомата будет бесконечное число состояний.

Назовем s_0, s_1, и s_2 на рис. 12.1 локациями. Состоянием временного автомата будет являться имя локации плюс значения всех его переменных, в том числе его локальных часов. Например, состояниями автомата на рис. 12.1 могут быть: $\langle s_0;\ x=0.1\rangle$, $\langle s_0;\ x=201.1\rangle$, $\langle s_1;\ x=0.4\rangle$, $\langle s_2;\ x=0.345\rangle$, но не могут быть $\langle s_1;\ x=5.1\rangle$, поскольку здесь нарушен инвариант в локации s_1, или $\langle s_0;\ x=-2.66\rangle$ — часы не могут принимать отрицательные значения.

Определим формально модель временного автомата — конечного автомата, расширенного введением локальных часов, отсчитывающих интервалы реального времени.

❖ *Определение 12.2 (временной автомат)*

Временной автомат — это шестерка $A=(Loc,\ loc_0,\ \Sigma,\ X,\ Inv,\ E)$, где:

- Loc — конечное множество локаций, содержащее начальную локацию loc_0;
- X — конечное множество переменных специального типа, называемых локальными часами.

  Значения локальных часов непрерывно изменяются — они непрерывно возрастают синхронно с реальным временем. Часы могут принимать любые вещественные значения из $\mathbf{R}^{\geq 0}$, они могут только читаться (использоваться в ограничениях) и сбрасываться в 0 на переходах;
- Σ — конечное множество действий; будем считать, что пустое действие входит в Σ;
- $Inv: Loc \to \gamma$ — инвариант, определенный в некоторых локациях.

  Здесь γ — это предикат, ограничение на возможные значения часов, определяемое как логическая формула, построенная из элементарных ограничений $x \sim c$ или $x - y \sim c$, где c — рациональное число,

x, y — локальные часы, а отношение $\sim$ принадлежит множеству $\{\le, <\}$ (т. е. $\sim \in \{\le, <\}$). Будем обозначать далее $\Gamma(X)$ множество возможных ограничений на значения локальных часов из множества X. Если значение $Inv(loc)$ не определено, то инвариант в локации loc есть *true*.

Инварианты определяют прогресс системы: они ограничивают сверху те значения, которые часы могут принимать в локации. Если инвариант определен в локации loc, то из этой локации должен быть выполнен переход до того, как этот инвариант станет ложным. Ситуация нарушения инварианта в локации останавливает прогресс времени и является ошибочной;

- E — множество переходов автомата, $E \subseteq Loc \times \Gamma \times \Sigma \times 2^X \times Loc$;

  если $(loc, \gamma, a, \Lambda, loc') \in E$, то мы будем писать:

  $loc - (\gamma, a, \Lambda) \to loc'$.

Каждый переход во временно̀м автомате — это переход между двумя локациями, помеченный защитой перехода $\gamma \in \Gamma(X)$ (возможно, пустой), действием из множества Σ (возможно, пустым), а также множеством Λ часов, которые на данном переходе сбрасываются в 0 (рис. 12.2). Как показано на рис. 12.2, далее в этой главе операцию сброса локальных часов на переходе мы иногда будем обозначать явно, т. е. вместо $\Lambda = \{x, y\}$ будем писать $x := 0;\ y := 0$.

Например, переход автомата рис. 12.1 из локации s_1 в локацию s_2 можно записать так:

$$s_1 - (x < 1, a, \varnothing) \to s_2 .$$

Автомат переходит из локации s_1 в локацию s_2 с выполнением события a, при этом значение времени, накопленного часами x, должно быть меньше 1. Переход всегда считается мгновенным: в данном примере после выполнения перехода значение часов x сохраняется: множество сбрасываемых в 0 часов на этом переходе пусто.

Как и состояния структуры Крипке, локации временного автомата могут быть помечены атомарными предикатами из конечного множества AP. Пусть L_A — эта функция пометок: $L_A : Loc \to 2^{AP}$. В дальнейшем будем считать, что имена локаций также являются атомарными предикатами.

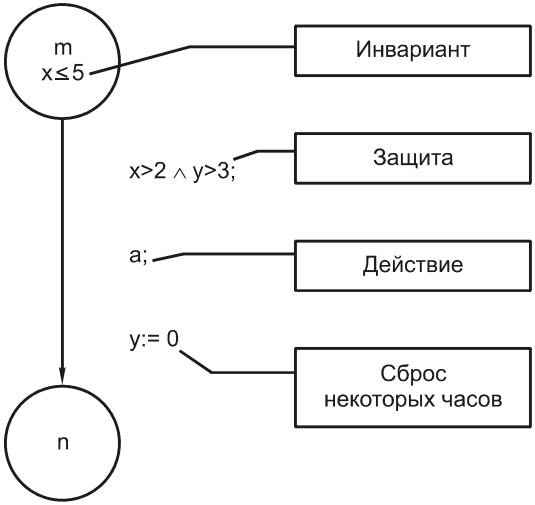


Рис. 12.2. Переход временного автомата

Таким образом, временной автомат — это такое расширение конечного автомата, у которого существует конечное множество локальных часов (непрерывных переменных специального типа). Локальные часы отличаются от переменных других типов: они принимают только неотрицательные вещественные значения, с часами нельзя выполнять операций, их можно только сбрасывать в ноль на переходах и их можно указывать в ограничениях. Часы используются для точной фиксации интервалов времени между событиями в системе и для ограничения времени нахождения системы в той или другой локации. Все локальные часы идут синхронно, с одной и той же скоростью 1 — синхронно с реальным временем. В каждый момент функционирования временного автомата каждые локальные часы показывают интервал реального времени, прошедший после их последнего сброса.

Такая модель позволяет выразить неопределенность временных интервалов между двумя событиями. Длительность интервалов может находиться от l (нижняя граница временного интервала, low bound) до u (верхняя граница, upper bound). Например, можно представить так называемые "неточные" временные переходы $s_1 - [l, u] \to s_2$, моделирующие протяженные во времени действия. Такие переходы можно выразить установлением инварианта в локации и установлением подходящей защиты перехода. Две возможности определить, что после события *start* событие *finish* наступит в произвольный момент во временном интервале [2, 8], показаны на рис. 12.3. Отличие этих двух вариантов заключается в том, что во втором случае, когда инвариант в локации n равен *true*, автомат может находиться в этой локации бесконечно долго и вовсе не выполнить переход в p.

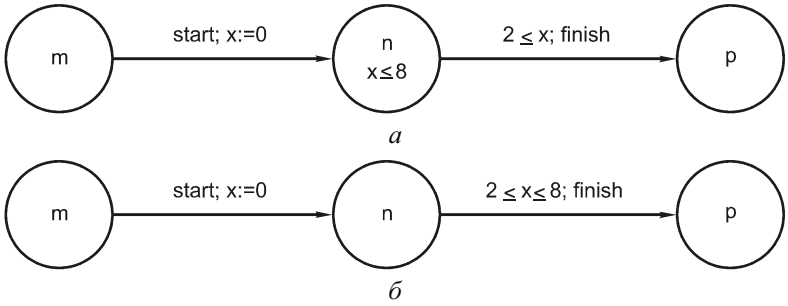


Рис. 12.3. Два варианта задания временного интервала между двумя мгновенными событиями *start* и *finish*

Временные автоматы являются одним из наиболее часто используемых формализмов для представления систем реального времени.

Не уменьшая точности задания временных ограничений, потребуем, чтобы константы в ограничениях реального времени γ были только рациональными числами. Например, мы можем их выразить с точностью до секунд, или до миллисекунд, или до микросекунд. Это требование имеет важное следствие: простым увеличением масштаба времени (выбрав наименьшее общее кратное m всех знаменателей ограничений и умножив на него все ограничения) мы можем свести все ограничения по времени к целым неотрицательным числам, не изменяя модель. Очевидно, что все асинхронные модели с непрерывным временем инвариантны относительно изменения масштаба времени.

В дальнейшем будем считать все константы в ограничениях реального времени γ натуральными числами. С учетом возможных логических операций над ограничениями значений часов можно получить любые их комбинации. Например, интервальное ограничение $[c_1, c_2]$ строится из элементарных ограничений таким образом:

$$x \in [c_1, c_2) \equiv (x \geq c_1) \wedge (x < c_2) \equiv \neg(x < c_1) \wedge (x < c_2)$$

Точное значение локальных часов $(x = c)$ может быть выражено конъюнкцией:

$$(x \leq c) \wedge \neg(x < c)$$

Модель временно̀го автомата достаточно удобна для формального описания функционирования систем реального времени.

Пример 12.2

Опишем стандартный протокол передачи данных FDDI. Временно̀й автомат, моделирующий процесс передачи, представлен на рис. 12.4.

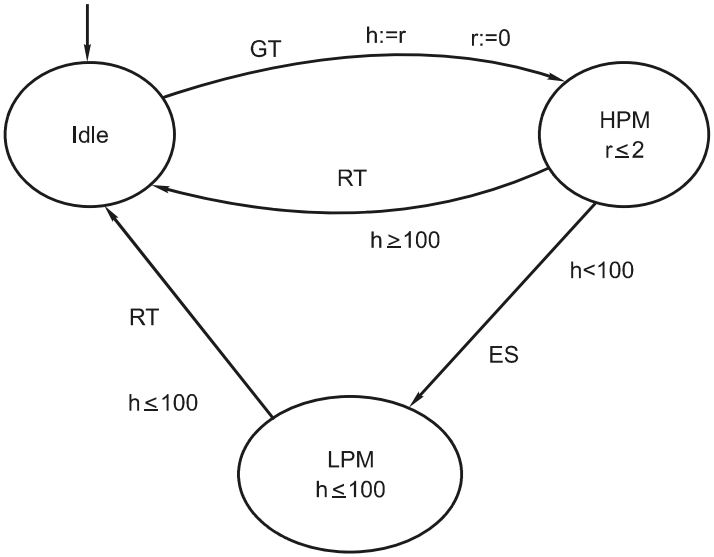


Рис. 12.4. Временной автомат, моделирующий работу передатчика протокола FDDI

- Вначале станция находится в режиме Idle. В этом режиме станция ожидает маркер. Станция имеет два таймера, r и h. Таймер r считает время, прошедшее после последнего прибытия маркера.
- По получении маркера (GT — действие "get token") таймеру h присваивается значение r, таймер r сбрасывается и станция переходит в режим HPM (High Priority Message).
- В режиме HPM станция посылает высокоприоритетные сообщения. Это может длиться не более 2 мс. Станция завершает передачу высокоприоритетных сообщений либо потому, что они кончились, либо потому, что истекли 2 мс.
- В режиме HPM, если с момента принятия маркера прошло более 100 мс, станция возвращается в режим Idle. В противном случае она переходит в режим LPM (Low Priority Message) — пересылки низкоприоритетных сообщений.
- В режиме LPM станция посылает низкоприоритетные сообщения до тех пор, пока они не кончатся, но по времени не более чем истекут 100 мс с момента начала посылки сообщений, после чего возвращается в режим Idle.

Модель временного автомата на рис. 12.4 достаточно адекватно представляет процесс передачи данных: здесь локации использованы как абстракция ре-

жимов передатчика, инварианты в состояниях — как средство описания прогресса системы, а временные защиты переходов используются для моделирования интервала нахождения передатчика в том или ином режиме.

Пример 12.3

Рассмотрим поведение простого временно̀го автомата A_2, изображенного на рис. 12.5, *а*. Автомат с одной локацией m и одними локальными часами x может выполнить действие a, только если при этом защита $x \geq 1$ выполняется. Инвариант — ограничение времени на нахождение системы в локации m, определен как $x < 2$. Все возможные значения часов — это кусочно-непрерывная функция от времени. График возможных показаний часов x автомата A_2 в реальном времени показан на рис. 12.5, *б*.

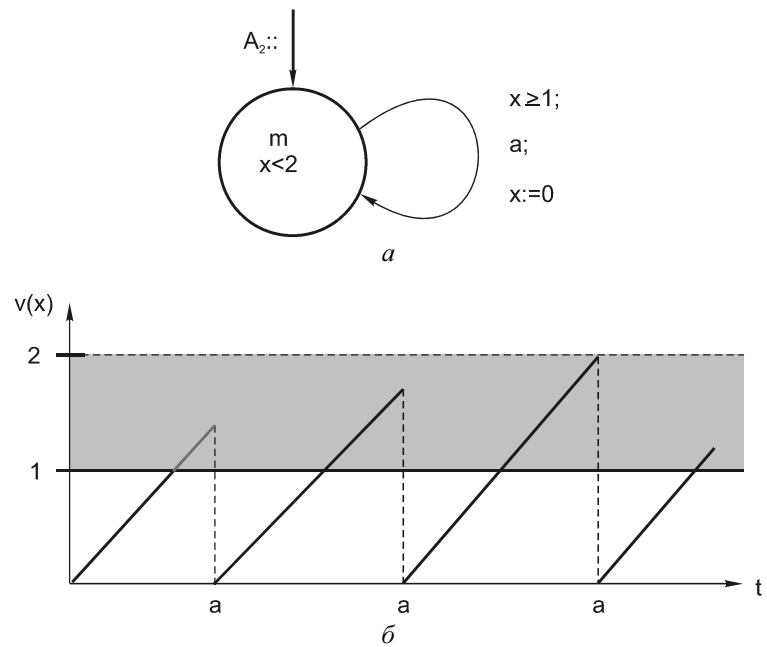


Рис. 12.5. Временно̀й автомат (*а*) и изменение значения его локальных часов во времени (*б*)

12.2. Сети временных автоматов: параллельная композиция

Как правило, системы реального времени состоят из нескольких компонент, взаимодействующих с помощью общих действий. Например, передатчик

протокола FDDI должен взаимодействовать с приемником, каналом, буфером, накапливающим сообщения для передачи. Свойства систем реального времени необходимо проверять для всей системы, потому что временные ограничения в одном компоненте системы существенно влияют на поведение остальных компонент. Поэтому необходимо уметь строить модели множества взаимодействующих компонент реального времени.

Несколько взаимодействующих временных автоматов, работающих параллельно, можно также описать моделью одного временнòго автомата, который строится из моделей компонент с помощью операции параллельной композиции. Пусть множества локальных часов в параллельно работающих автоматах не пересекаются, и автоматы взаимодействуют с помощью синхронной коммуникации — посылки/приема сообщений (рандеву). Обычно принимается, что действия посылки и приема одного типа сообщения в разных компонентах обозначаются одинаковыми именами.

Пусть два временных автомата A_1 и A_2 определены так:

$$A_1 = \left(Loc_1, loc_{01}, \Sigma_1, X_1, Inv_1, E_1\right)$$

$$A_2 = \left(Loc_2, loc_{02}, \Sigma_2, X_2, Inv_2, E_2\right)$$

Алфавиты автоматов имеют общие символы, характеризующие их синхронное взаимодействие, но каждый компонентный автомат имеет свое независимое множество локальных часов: $X_1 \cap X_2 = \varnothing$. Построим параллельную композицию автоматов A_1 и A_2 (обозначается $A_1 \| A_2$), которая описывает поведение двух автоматов, стоящих рядом и взаимодействующих с помощью рандеву полностью аналогично операции параллельной композиции процессов, рассмотренной в *гл. 5*. Параллельная композиция временных автоматов определяется так:

$$A_1 \| A_2 = \left(Loc_1 \times Loc_2, \left(loc_{01}, loc_{02}\right), \Sigma, X, Inv, E\right)$$

Локациями автомата $A_1 \| A_2$ являются пары локаций компонентов, начальной локацией этого автомата является пара начальных локаций компонентов. Инвариант Inv параллельной композиции ставит в соответствие паре состояний конъюнкцию соответствующих инвариантов: $Inv(s_1, s_2) = Inv_1(s_1) \wedge Inv_2(s_2)$. Алфавиты действий и множества локальных часов в композиции объединяются: $\Sigma = \Sigma_1 \cup \Sigma_2$, $X = X_1 \cup X_2$. Множество переходов автомата $A_1 \| A_2$:

$$E \subseteq \left(Loc_1 \times Loc_2\right) \times \Gamma\left(X\right) \times \Sigma \times 2^X \times \left(Loc_1 \times Loc_2\right)$$

определяется следующими правилами.

Пусть a — действие автомата A_1, $a \in \Sigma_1$, и существует переход в A_1:

$$(loc_1, \gamma_1, a, \lambda_1, loc_1') \in E_1.$$

Пусть b — действие автомата A_2, $b \in \Sigma_2$, и существует переход в A_2:

$$(loc_2, \gamma_2, b, \lambda_2, loc_2') \in E_2.$$

Тогда:

1. Если a — общее действие автоматов, $a = b$ (т. е. $a \in \Sigma_1 \cap \Sigma_2$), то E будет включать переход

   $$\big((loc_1, loc_2), \gamma_1 \wedge \gamma_2, a, \lambda_1 \cup \lambda_2, (loc_1', loc_2')\big)$$

   — одновременный переход обоих автоматов при их синхронном взаимодействии. В обоих автоматах этот переход должен удовлетворять их защитам γ_1 и γ_2, при этом локальные часы сбрасываются в 0 из множества $\lambda_1 \cup \lambda_2$ (из множества λ_1 в первом автомате и из множества λ_2 во втором; в каждом компонентном автомате сбрасываются свои часы).
2. Если a — действие только автомата A_1, $a \in \Sigma_1 \setminus \Sigma_2$, то для каждой локации второго автомата $loc_2 \in Loc_2$, E будет включать переход

   $$\big((loc_1, loc_2), \gamma_1, a, \lambda_1, (loc_1', loc_2)\big)$$

   — независимое действие (внутренний переход) первого автомата.
3. Если b — действие автомата A_2, $b \in \Sigma_2 \setminus \Sigma_1$, то для каждой локации первого автомата $loc_1 \in Loc_1$, E будет включать переход

   $$\big((loc_1, loc_2), \gamma_2, b, \lambda_2, (loc_1, loc_2')\big)$$

   — независимое действие (внутренний переход) второго автомата.

Эти определения показывают, что общие действия в параллельной композиции временных автоматов считаются взаимно дополнительными, они моделируют синхронное взаимодействие рандеву. Оба автомата выполняют их одновременно. Свои внутренние действия автоматы выполняют независимо (интерливинг).

Пример 12.4

Пример параллельной композиции двух временных автоматов приведен на рис. 12.6. Если защита отсутствует на переходе, то ее значение *true*. Например, переход $(s, true, a, x := 0, s')$ не включает ограничения по времени на пе-

реход автомата A_3, оно может быть выполнено при любых значениях локальных часов x. Этот переход из состояния с локацией s в состояние с локацией s' выполняется при действии a, при этом значение часов x устанавливается в 0, а показания остальных часов не меняются. Если множество сбрасываемых в 0 локальных часов пусто, оно также на переходе опускается. Из рис. 12.6 видно, что общее действие b оба автомата выполняют одновременно, и каждый из автоматов выполняет свои локальные действия a и c независимо от другого.

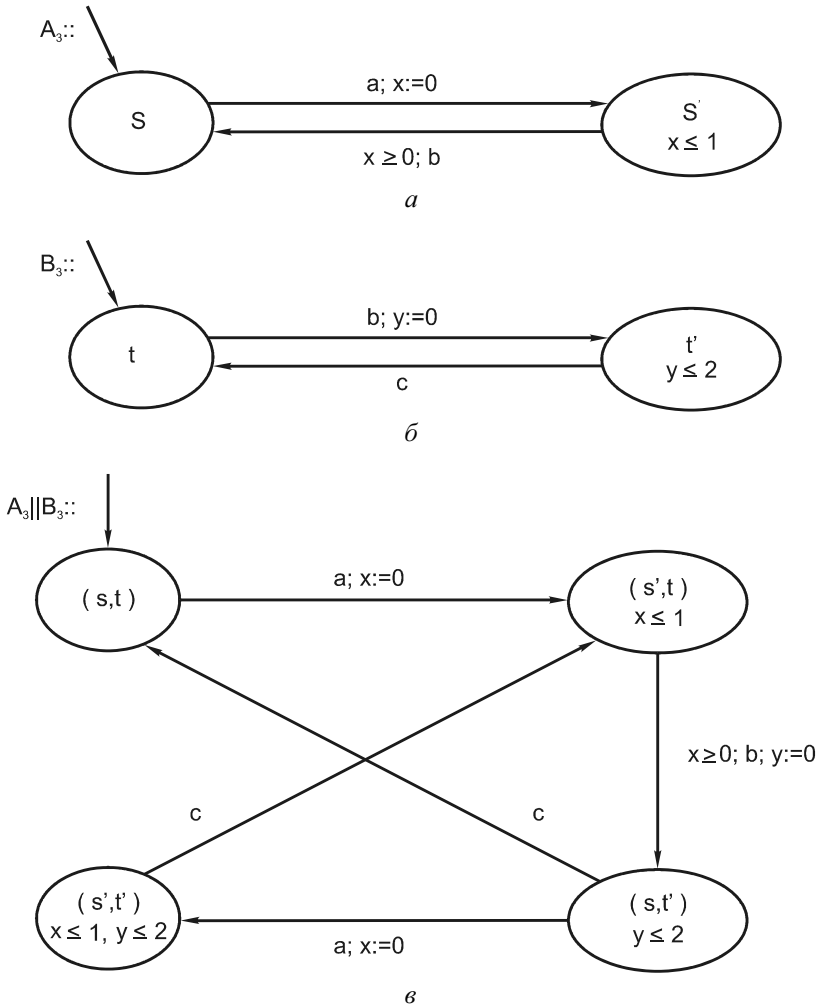


Рис. 12.6. Параллельная композиция двух временных автоматов

Таким образом, операция параллельной композиции временных автоматов с пересекающимися алфавитами действий позволяет построить формальную модель распределенной сети компонентов, взаимодействующих методом рандеву. Модель временно̀го автомата является очень удобной: операция параллельной композиции таких моделей в качестве результата дает временно̀й автомат, который описывает общее функционирование нескольких взаимодействующих компонентов. В инструментальных системах верификации параллельная композиция нескольких временных автоматов строится автоматически по заданным моделям компонентных подсистем.

12.3. Операционная семантика модели временно̀го автомата — система переходов

Анализ свойств поведения временного автомата невозможно выполнить непосредственно по его спецификации. Определение временного автомата — это просто синтаксические правила, конечное описание, определяющее возможные действия и ограничения на выполнение этих действий. В таком определении некоторые ограничения могут быть противоречивы, некоторые состояния и даже локации могут быть вообще недостижимыми. Интерпретацией этих синтаксических правил описания временного автомата является его поведение, его возможные вычисления. Каждое возможное вычисление временного автомата определяется последовательностью состояний, которые проходит автомат в вычислении, и переходами между этими состояниями.

Поведение временно̀го автомата A определяется его операционной семантикой, которую можно представить графом переходов $\mathsf{T}(A)$. Определим граф переходов временного автомата формально.

Введем понятие *интерпретации часов* v как присваивания каждым локальным часам временного автомата неотрицательного действительного значения, $v: X \to \mathbf{R}^{\geq 0}$. Обозначим $v + d$ увеличение показаний часов $x \in X$ с $v(x)$ до $v(x) + d$. Формально $(v + d)(x) = v(x) + d$.

❖ *Определение 12.3*

Графом переходов временного автомата $A = (Loc, loc_0, \Sigma, X, Inv, E)$ назовем четверку $\mathsf{T}(A) = (\mathrm{Q}, \mathrm{q}_0, \Sigma, R)$, где:

- $Q = Loc \times (X \to \mathbf{R}^{\geq 0})$ — множество состояний. Каждое состояние — это пара $\langle loc; v \rangle$, где $loc \in Loc$ — локация, а $v: X \to \mathbf{R}^{\geq 0}$ интерпре-

тации всех локальных часов данного автомата, удовлетворяющие инварианту локации loc:

$v \models Inv(loc)$;

- q_0 — начальное состояние, $q_0 = \langle loc_0; v_0 \rangle$, где v_0 — начальная интерпретация всех часов, такая, что для любых $x \in X$, $v_0(x) = 0$;
- Σ — множество действий (совпадает с множеством действий автомата A);
- R — множество переходов двух видов:
  - задержка в локации автомата A на временной интервал $d > 0$ (переход по времени):

    $\langle loc; v \rangle - d \to \langle loc; v + d \rangle$ — выполняется при условии удовлетворения инвариантов, т. е. в случае, если инвариант $Inv(loc)$ истинен для $v + t$ при любом t в интервале $0 \le t \le d$;
  - переход $a \in \Sigma$ с изменением локации автомата A (переход по действию):

    $\langle loc; v \rangle - a \to \langle loc'; v' \rangle$ — выполняется при условии истинности защиты на этом переходе, т. е. если $(loc, \gamma, a, \lambda, loc') \in E$, то v удовлетворяет защите $\gamma \in \Gamma(X): v \models \gamma$; v' — новые значения часов после перехода таковы: показания сбрасываемых часов из множества λ становятся равными 0, показания остальных часов не изменяются, и показания часов после выполнения перехода удовлетворяют инварианту новой локации loc': $v' \models Inv(loc')$.

Будем обозначать $s + d$ для любого состояния $s = \langle loc; v \rangle$ системы переходов временного автомата и неотрицательного d такое состояние $\langle loc; v + d \rangle$, в которое система переходов может перейти после задержки d.

Пример 12.5

На рис. 12.7 схематически представлен граф переходов $T(A_2)$ временного автомата A_2 (см. рис. 12.5). У автомата A_2 возможны следующие переходы, которые представлены в $T(A_2)$:

$\langle m;\ x=0.1\rangle - 0.3 \to \langle m;\ x=0.4\rangle$ (переход по задержке);

$\langle m;\ x=0.1\rangle - 0.2211 \to \langle m;\ x=0.3211\rangle$ (переход по задержке);

$\langle m;\ x=1.3\rangle - a \to \langle m;\ x=0\rangle$ (переход по действию).

Следующие переходы невозможны в автомате A:

$\langle m;\ x=0.1\rangle - 2.5 \to \langle m;\ x=2.6\rangle$ — нарушение инварианта в локации *m*;

$\langle m;\ x=1.1\rangle - a \to \langle m;\ x=1.6\rangle$ — часы x на переходе должны быть сброшены;

$\langle m;\ x=0.3\rangle - a \to \langle m;\ x=0\rangle$ — нарушение защиты перехода.

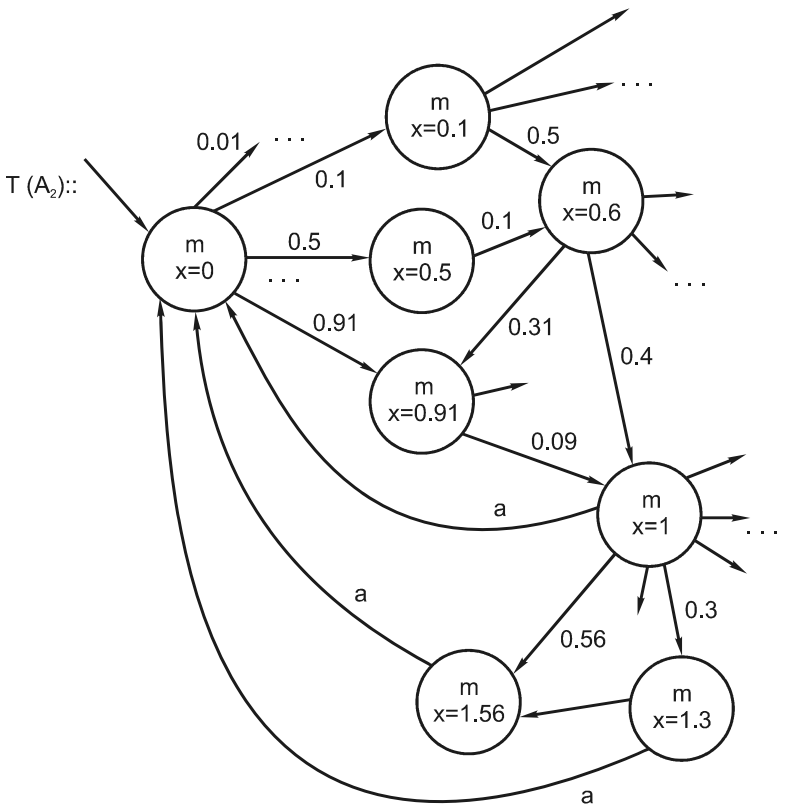


Рис. 12.7. Граф переходов временнòго автомата A_2

Граф переходов временного автомата представляет все возможные его поведения. Любое поведение временного автомата начинается с начального со-

стояния с начальной локацией loc_0 и всеми локальными часами, установленными в 0. Затем автомат альтернативно выбирает либо задержку по времени, синхронно увеличивая значения всех локальных часов, если удовлетворяется инвариант текущей локации, либо переход по действию, если значения локальных часов удовлетворяют защите этого перехода. Переход по действию мгновенный: значения локальных часов при его выполнении не меняются за исключением тех, которые сбрасываются на переходе в 0.

Все допустимые последовательности состояний графа переходов представляют возможные вычисления соответствующего временного автомата. Например, во временном автомате A_1 рис. 12.1 возможно следующее вычисление:

$$\sigma = \langle s_0;\, x=0\rangle - 0.5 \to \langle s_0;\, x=0.5\rangle - 6.1 \to \langle s_0;\, x=6.6\rangle - a \to \langle s_1;\, x=0\rangle - 0.5 \to$$
$$\langle s_1;\, x=0.5\rangle - 0.2 \to \langle s_1;\, x=0.7\rangle - a \to \langle s_2;\, x=0.7\rangle - 0.2 \to \langle s_2;\, x=0.9\rangle - b \to$$
$$\langle s_0;\, x=0.9\rangle - 0.3 \to \langle s_0;\, x=1.2\rangle \to \ldots$$

Для произвольного вычисления

$$\pi = q_0 - a_0 \to q_1 - a_1 \to q_2 - a_2 \to q_3 - a_3 \to \ldots$$

где a_i — либо действие, либо задержка, обозначим $\Delta(\pi, i)$ время, прошедшее с момента старта автомата из начального состояния q_0 до момента, когда автомат окажется в состоянии q_i. Формально:

$\Delta(\pi, 0) = 0$;

$\Delta(\pi, i+1) = \Delta(\pi, i) + d$, если переход a_i — это задержка d;

$= \Delta(\pi, i)$ — если переход a_i — это переход по действию.

Для приведенного выше вычисления σ автомата A_1:

$\Delta(\sigma, 2) = 6.6$

$\Delta(\sigma, 4) = 7.1$

$\Delta(\sigma, 5) = 7.3$

$\Delta(\sigma, 6) = 7.3$

Вычисления временного автомата можно рассматривать по-разному в зависимости от целей анализа. Например, абстрагируясь от действий, вычисление можно представить как последовательность состояний и переходов — временных задержек, например:

$$\sigma = \langle s_0; x = 0\rangle - 0.5 \to \langle s_0; x = 0.5\rangle - 6.1 \to \langle s_0; x = 6.6\rangle - 0 \to \langle s_1; x = 0\rangle - 0.5 \to$$
$$\langle s_1; x = 0.5\rangle - 0.2 \to \langle s_1; x = 0.7\rangle - 0 \to \langle s_2; x = 0.7\rangle - 0.2 \to \langle s_1; x = 0.9\rangle \ldots$$

Абстрагируясь от переходов (и задержек, и действий), вычисление можно представить просто как последовательность состояний. Приведенное выше вычисление с этой точки зрения можно представить так:

$$\sigma = q_0\, q_1\, q_2\, q_3\, q_4 \cdots,$$

где

$$q_0 = \langle s_0; x = 0\rangle$$

$$q_1 = \langle s_0; x = 0.5\rangle$$

$$q_2 = \langle s_0; x = 6.6\rangle \ldots$$

По вычислению временного автомата можно также построить допустимое "*временное слово*" автомата — последовательность выполненных автоматом действий и моменты реального времени, в которых эти действия выполнялись:

$$\langle t_0, a_0\rangle\langle t_1, a_1\rangle\langle t_2, a_2\rangle \ldots$$

Такая последовательность интерпретируется следующим образом: временной автомат A после запуска из его начального состояния в момент времени t_0 может выполнить действие a_0, потом в момент времени t_1 выполнить действие a_1, затем в момент времени t_2 выполнить действие a_2 и т. д. Например, по указанному выше вычислению σ автомата A_1 может быть построено "*временное слово*"

$$\langle 6.6,\ a\rangle\langle 7.3,\ a\rangle\langle 7.5,\ b\rangle \ldots .$$

Состояние $\langle loc; v\rangle$ достижимо во временном автомате A, если существует вычисление графа переходов A, начинающееся в состоянии $\langle loc_0; v_0\rangle$ и заканчивающееся в состоянии $\langle loc; v\rangle$. Например, в автомате A_1 достижимо состояние $\langle s_2; x = 0.7\rangle$, поскольку существует вычисление, начинающееся в начальном состоянии $\langle s_0; x = 0\rangle$ автомата A_1 и заканчивающееся в состоянии $\langle s_2; x = 0.7\rangle$. Непосредственно по временному автомату о достижимости состояний судить трудно. Таким образом, система переходов $\mathsf{T}(A)$ представляет реальное поведение временно̀го автомата A, т. е. все его возможные вычисления. Поскольку часы могут принимать бесконечно много (континуум)

значений, система переходов любого временного автомата бесконечна, а сам временной автомат — лишь удобная конечная модель для задания поведения систем реального времени.

Существует важный результат относительно семантики параллельной композиции временных автоматов:

ТЕОРЕМА 12.1

Графы переходов $\mathsf{T}(A) \| \mathsf{T}(B)$ и $\mathsf{T}(A \| B)$ изоморфны.

Смысл этой теоремы в том, что вычисления совместно функционирующих взаимодействующих систем реального времени можно получить как граф переходов параллельной композиции моделей этих систем — временных автоматов.

12.4. Нереалистичные поведения временнòго автомата

Для того чтобы вычисления графа переходов временного автомата отражали реальное поведение системы, они должны удовлетворять некоторым ограничениям, связанным со свойствами реального времени. В частности, для любых состояний s, s', s'' и неотрицательных чисел $d, d' \in \mathbf{R}^{\geq 0}$ вычисления графа переходов временного автомата должны удовлетворять следующим требованиям:

1. $s - d \to s' \wedge s - d \to s'' \Rightarrow s' = s''$ (детерминизм времени);
2. $s - 0 \to s$ (нулевая задержка);
3. $s - d \to s' \wedge s' - d' \to s'' \Rightarrow s - d + d' \to s''$ (аддитивность времени);
4. $s - d + d' \to s' \Rightarrow (\exists s''): s - d \to s'' \wedge s'' - d' \to s'$ (непрерывность времени).

Но даже при выполнении этих требований не все вычисления, допустимые в соответствии с описанием временного автомата, являются реалистичными. Рассмотрим их.

Конвергентные и дивергентные вычисления. Во-первых, рассмотрим следующую цепочку переходов по задержке времени в автомате A_1, представленном на рис. 12.1:

$$\langle s_1;\ x = 0.3 \rangle - 0{,}1 \to \langle s_1;\ x = 0.31 \rangle - 0.01 \to \langle s_1;\ x = 0.311 \rangle - 0.001 \to$$
$$\langle s_1;\ x = 0.3111 \rangle \to \ldots$$

Эту цепочку с уменьшением шага по времени можно продолжать бесконечно: она представляет вычисление, которое синтаксически возможно. Оно строится по правилам дробления задержки на произвольно малые части. Вычисление это проходит бесконечное число состояний и шагов по времени, но время при этом остается меньше некоторой границы. Нереалистичность таких вычислений состоит в том, что они не могут выполниться: здесь время делится бесконечно, фактически, останавливаясь на этих бесконечных вычислениях. Подобные бесконечные вычисления с остановкой времени называются "*конвергентными*". Конвергентные вычисления противоречат интуитивному понятию времени: при бесконечном числе временных шагов время в природе возрастает неограниченно (время не делится бесконечно!!). Бесконечные вычисления, в которых время растет бесконечно, называются "*дивергентными*". Формально, вычисление σ будет дивергентным, если

$$\lim \Delta_{i\to\infty}(\sigma, i) = \infty .$$

При анализе временных автоматов конвергентные вычисления не рассматриваются, хотя в модели формально их построить можно. Анализ временных автоматов ограничивается только дивергентными вычислениями.

Вычисления Зенона. Примерами другого типа конвергентных вычислений, которые формально могут существовать во временном автомате, хотя они и противоречат интуиции, являются так называемые "*вычисления Зенона*". Такие вычисления представляют выполнение бесконечного числа переходов по действиям между локациями автомата за конечное время. Например, у автомата A_4 (рис. 12.8, *а*) может быть такое вычисление:

$$\langle m;\ x=0\rangle - 1 \to \langle m;\ x=1\rangle - a \to \langle n;\ x=0\rangle - 1/2 \to \langle n;\ x=1/2\rangle - b \to$$
$$\langle m;\ x=0\rangle - 1/4 \to \langle m;\ x=1/4\rangle - a \to \langle n;\ x=0\rangle - 1/8 \to \langle n;\ x=1/8\rangle - b \to$$
$$\langle m;\ x=0\rangle - 1/16 \to \langle m;\ x=1/16\rangle - a \to \langle n;\ x=0\rangle - 1/32 \to \langle n;\ x=1/32\rangle - b \to ...$$

Это конвергентное вычисление, здесь на бесконечном вычислении время тоже растет ограниченно (рис. 12.8, *б*), но природа некорректности другая. Подобные вычисления являются моделью хорошо известного парадокса Зенона: *выполнение бесконечного количества действий в ограниченном интервале времени*. В этом и состоит нереалистичность таких вычислений, хотя они возможны в соответствии со спецификацией временного автомата A_4.

В модели временного автомата мы предполагаем, что переходы, выполняемые вследствие наступления событий (действий), мгновенны. Если последовательность таких мгновенных действий образует цикл, то это и приводит к вычислениям Зенона. Хотя временной задержкой каждого из действий в цикле мы можем пренебречь, бесконечное число таких действий не может быть

ни мгновенным, ни выполниться в ограниченный интервал времени, поскольку скорость процессоров не является бесконечной. Наличие вычислений Зенона свидетельствует о некорректности модели.

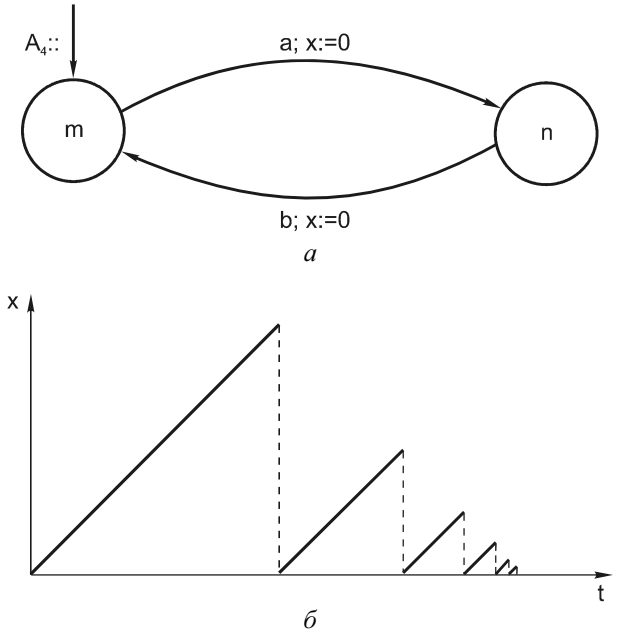


Рис. 12.8. *а*) Временнòй автомат A_4 ; *б*) изменение во времени показаний его локальных часов на вычислении Зенона

Задержку, соответствующую реальному действию в системе, моделью которой является мгновенный переход во временнòм автомате, можно учесть разными способами. Например, модель, у которой существует цикл с мгновенным действием a , можно легко изменить, введя дополнительно:

- новую локацию "*выполнение действия* a ";
- новые локальные часы x_a, учитывающие длительность действия a ;
- инвариант $\left(x_a \le \max\right)$ в локации "*выполнение действия* a ";
- мгновенные события *начало*$_a$ и *окончание*$_a$ выполнения действия a ;
- сброс часов x_a на переходах, помеченных событием *начало*$_a$;
- ограничение $\left(0 < \min \le x_a\right)$, которое установлено на переходе с мгновенным действием *окончание*$_a$ из локации "*выполнение действия* a ".

Возможны и другие варианты учета в модели задержки при выполнении действия в реальной системе. Два различных варианта введения подобной задержки показаны на рис. 12.3.

Блокировка времени в состоянии. Примером этой некорректности спецификации временного автомата является возможность попадания автомата в состояние, в котором текущие значения локальных часов не удовлетворяют инварианту соответствующей локации. Такая некорректность называется "*блокировкой времени*" (timelock). Если в автомате A_1 (см. рис. 12.1) в локации s_0 определить инвариант $x<1$, то переход из состояния $\langle s_2; x=1\rangle$ в состояние $\langle s_0; x=1\rangle$ при действии b запрещен: такого состояния с локацией s_0 быть не может. Но оставаться в локации s_2 автомат также не может: это запрещено инвариантом локации s_2.

Отсутствие прогресса. Еще одним примером некорректного поведения временного автомата является следующее вычисление автомата A_4 на рис. 12.8:

$$\langle m;\ x=0\rangle - 0.2 \to \langle m;\ x=0,2\rangle - 1.8 \to \langle m;\ x=2\rangle - 10.4 \to$$
$$\langle m;\ x=12.4\rangle - 6.5 \to \langle m;\ x=18.9\rangle \ldots$$

В этом вычислении время растет, но автомат не прогрессирует: он бесконечно остается в одной и той же локации, имея при этом в любой момент альтернативу — либо выполнить действие a, либо задержаться в текущей локации. Синтаксически определить, что автомат должен выполнить переход, можно с помощью инварианта, который ограничит время нахождения автомата в локации. В локации m автомата A_4 этот инвариант равен *true*, именно поэтому здесь автомат может оставаться бесконечно долго. Отсутствие инварианта в локации не всегда является ошибкой. Например, отсутствие инварианта в локации s_0 временного автомата A_1 (см. рис. 12.1) не противоречит требованиям к нему: этот автомат является моделью реагирующей системы, он реагирует на внешние события и может ожидать нажатия кнопки в начальной локации s_0 сколь угодно долго. Однако в локациях s_1 и s_2 этого автомата наличие инвариантов обязательно: внешнее окружение должно быть проинформировано о типе щелчка кнопки мыши в ограниченный интервал времени.

12.5. Проблема анализа системы переходов временного автомата

Одной из главных проблем, препятствующих анализу свойств поведения и верификации систем реального времени, является бесконечность числа со-

стояний системы переходов T. Поскольку состоянием графа переходов временнòго автомата является пара ⟨*локация; показания всех локальных часов*⟩, у семантической модели временнòго автомата всегда будет не просто бесконечное число состояний, число его состояний имеет мощность континуум. Например, временной автомат A_2 с единственной локацией, единственными часами и только одним переходом имеет систему переходов $\mathsf{T}(A_2)$ с бесконечным множеством возможных состояний и переходов (см. рис. 12.7).

Возможна ли в принципе автоматическая верификация (например, анализ достижимости, алгоритмы проверки модели и т. п.) таких систем? Алгоритмы верификации разработаны для систем переходов с конечным числом состояний, они не могут быть механически перенесены на системы с бесконечным числом состояний.

Один из методов упрощения системы переходов — объединить состояния системы в группы (классы) эквивалентных состояний и рассмотреть вместо всего множества состояний только группы таких состояний, которые в некотором смысле подобны (рис. 12.9). Группировать в классы эквивалентности можно по-разному. В нашем случае отношение эквивалентности на множестве состояний должно быть выбрано так, чтобы с точки зрения проверяемых свойств исходная система и редуцированная система меньшего размера были бы взаимозаменяемыми, неразличимыми.

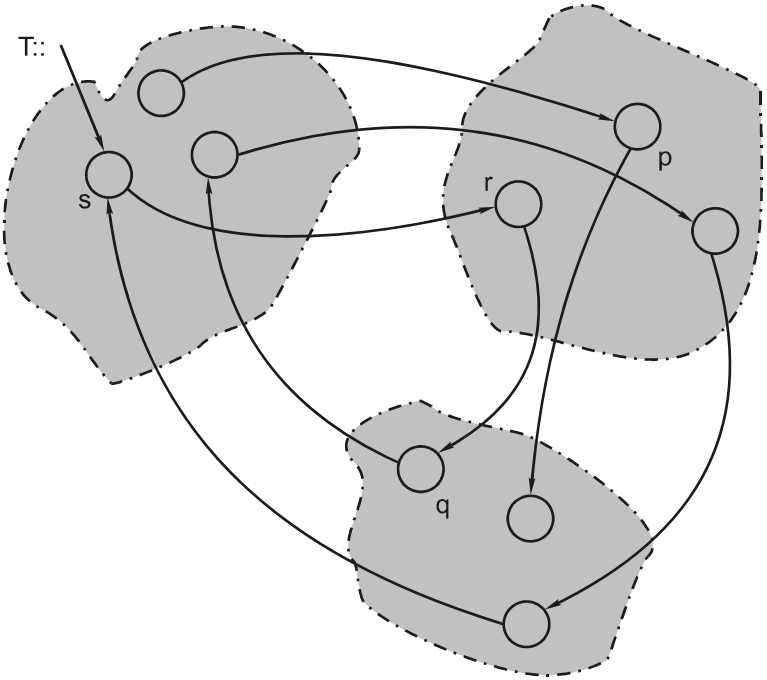
Пусть дана система переходов $\mathsf{T} = (S, s_0, \Sigma, R)$ и некоторое отношение эквивалентности π, объединяющее в один класс состояния системы T, которые согласуются с переходами T (рис. 12.9):

$$s \approx_\pi q \equiv (\forall(s, a, s') \in R)\,(\exists(q, a, q') \in R) : s' \approx_\pi q',$$

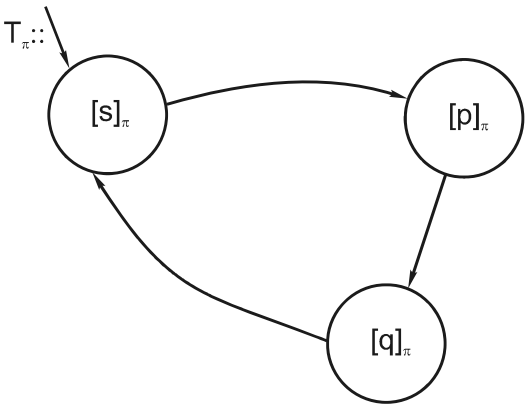
т. е. любые переходы из состояний, эквивалентных относительно эквивалентности π, осуществляются только в состояния, которые также эквивалентны по эквивалентности π.

Обозначим T_π новую систему переходов, которая называется "*фактор-системой переходов* T *по модулю отношения эквивалентности* π" и строится так:

- ❒ классы эквивалентности π являются состояниями T_π;
- ❒ класс эквивалентности π, включающий s_0, является начальным состоянием T_π;
- ❒ алфавит действий T_π совпадает с алфавитом действий T;
- ❒ множество переходов T_π определяется для пар классов эквивалентности π.



а



б

Рис. 12.9. Система переходов T и система переходов T_π по модулю отношения эквивалентности π

Частным случаем такой эквивалентности является эквивалентность, в которой каждое состояние системы переходов эквивалентно только себе. Фактор-система переходов по модулю такой эквивалентности совпадает с исходной

системой переходов. Нас интересуют эквивалентности, объединяющие в один класс множества состояний. Если даже число состояний системы переходов T бесконечно, но индекс (число классов) эквивалентности π будет конечным, то задача анализа бесконечной системы переходов после такой редукции может свестись к задаче анализа конечной системы переходов T_π.

Можно ли определить такое отношение эквивалентности π на множестве состояний системы переходов $\mathsf{T}(A)$ временнòго автомата A, чтобы редуцированная система переходов $\mathsf{T}_\pi(A)$, состояниями которой являются классы эквивалентности π, могла бы заменить $\mathsf{T}(A)$ при анализе, по крайней мере, некоторых интересных свойств?

Такое отношение эквивалентности существует, оно определяется на основе понятия временных регионов.

12.6. Временные регионы: один таймер

Когда можно объединять в один класс эквивалентности два состояния системы переходов $\mathsf{T}(A)$ временного автомата A, чтобы сохранились свойства системы переходов, важные для анализа ее свойств?

Пусть у временнòго автомата A только одни локальные часы. Рассмотрим два состояния: $\langle p; v\rangle$ и $\langle q; v\rangle$ системы переходов $\mathsf{T}(A)$. Если p и q — две разные локации, то объединять эти состояния в один класс эквивалентности, конечно, нельзя: состояния с разными локациями могут иметь различные значения атомарных предикатов в них, и свойства автомата A невозможно будет исследовать после такого объединения.

Рассмотрим теперь два состояния автомата A с одними и теми же именами локаций, но отличающиеся значениями локальных часов, например, $s_1 = \langle q;\ x = 1.5\rangle$ и $s_2 = \langle q;\ x = 1.6\rangle$. Можно ли утверждать, что s_1 и s_2 эквивалентны относительно каких-либо свойств автомата A?

Очевидно, что да, потому что в этих состояниях локация одна и та же, и между состояниями s_1 и s_2 прошло всего 0.1 единицы непрерывного времени. Такое изменение значения часов не повлияет ни на истинность условий перехода, ни на истинность инвариантов, определяющих нахождение автомата в локации. Действительно, все константы в неравенствах, используемых в инвариантах и защитах любого временнòго автомата, в соответствии с приня-

тым условием являются целыми числами. Поэтому все возможные ограничения на значения часов для обоих состояний $s_1 = \langle q;\ x = 1.5 \rangle$ и $s_2 = \langle q;\ x = 1.6 \rangle$ либо истинны, либо ложны одновременно. Следовательно, эти два состояния эквивалентны относительно всех свойств, которые можно сформулировать для временного автомата, у которого во временных ограничениях использованы только целые значения.

Пусть теперь $s_1 = \langle q; x = 1.9995 \rangle$ и $s_2 = \langle q; x = 2.0005 \rangle$. Хотя показания часов x в состоянии s_1 отличается от показания часов x в состоянии s_2 только на 0.001, нет гарантии, что между этими состояниями не произошло никаких изменений в значениях условий переходов для выполнения действий автомата. Например, условие $x < 2$ для первого состояния выполняется, для второго — нет.

Очевидно, что если показания (единственных) часов в одной и той же локации имеют ненулевую дробную часть и не отличаются целыми частями, то с точки зрения поведения ЛЮБОГО временного автомата с ЛЮБЫМИ временными ограничениями они эквивалентны.

Пусть C_x — максимальная целая константа в неравенствах для локальных часов x (эту константу будем называть "*потолком*" часов x). Обозначим $\lfloor V \rfloor$ — целую часть вещественного числа V, $fr(V)$ — дробную часть числа V.

❖ *Определение 12.4 (региональное отношение эквивалентности)*

Определим отношение эквивалентности $\cong$ на множестве вещественных значений единственных локальных часов x временного автомата следующим образом:

$v(x) \cong v'(x)$, если:

- $v(x) > C_x$ и $v'(x) > C_x$

  все показания локальных часов x, превышающие свой потолок, эквивалентны,

или

- $v(x) \le C_x \Rightarrow \lfloor v(x) \rfloor = \lfloor v'(x) \rfloor \wedge fr(v(x)) = 0 \wedge fr(v'(x)) = 0$

  целые показания локальных часов x, не превышающие потолок, составляют отдельные классы эквивалентности,

или

- $v(x) \le C_x \Rightarrow \lfloor v(x) \rfloor = \lfloor v'(x) \rfloor \wedge fr(v(x)) \neq 0 \wedge fr(v'(x)) \neq 0$

  любые показания часов x, не превышающие потолок C_x, с одинаковыми целыми частями и произвольными ненулевыми дробными частями, эквивалентны.

Классы эквивалентности определенного таким образом отношения $\cong$ называются *временными регионами* (timed regions), а само отношение $\cong$ — *региональной эквивалентностью.*

❖ ***Определение 12.5 (временной регион)***

Временной регион временного автомата — это класс эквивалентности $\cong$ интерпретаций локальных часов автомата.

Очевидно, таких классов (регионов) всегда конечное число, поскольку потолок часов ограничен. В случае единственных локальных часов регионы описываются как интервалы вещественной оси:

$$\{(x=0),(0<x<1),(x=1),(1<x<2),(x=2),...,(x=C_x),(x>C_x)\}.$$

Их количество равно $2*(C_x+1)$. Пример временных регионов для $C_x = 2$ представлен на рис. 12.10, *а*.

Заметим, что отношение региональной эквивалентности определено безотносительно какому-нибудь временному автомату или конкретному набору ограничений на эти часы. Оно зависит только от потолка часов и определено так, чтобы любые соотношения между значениями часов, имеющие в качестве констант только целые, принимали бы одно и то же значение истинности для всех интерпретаций часов x из одного и того же региона.

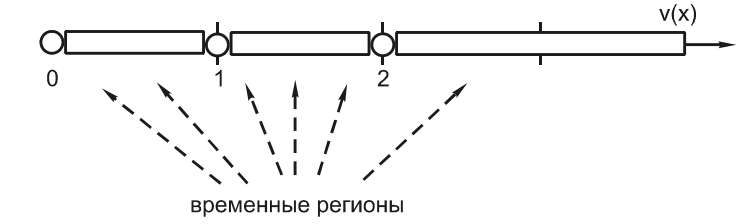
Обозначим $[v(x)]_{\cong}$ класс эквивалентности $\cong$ (регион), которому принадлежит интерпретация $v(x)$ часов x. Если отношение эквивалентности $\cong$ очевидно, класс эквивалентности $[v(x)]_{\cong}$ будем записывать $[v(x)]$. Например, пусть $v(x) = 0.3$; тогда $[v(x)] = (0<x<1)$. Другие примеры (при $C_x = 2$):

$$[x=0] = (x=0)$$

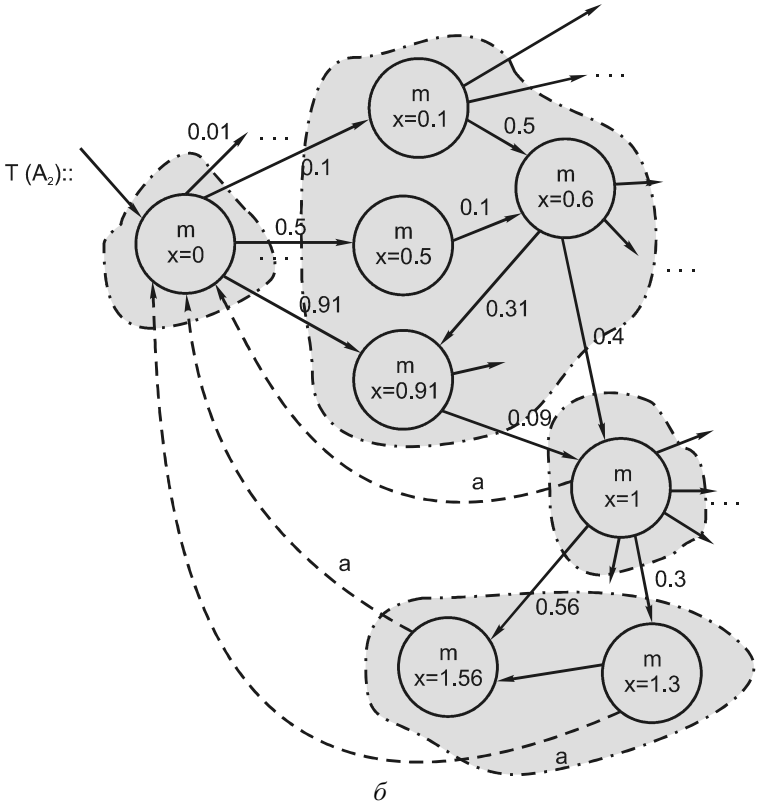
$$[x=1.56] = [x=1.3] = (1<x<2)$$

$$[x=1.4] = [x=1.3244] = [x=1.89] = (1<x<2)$$

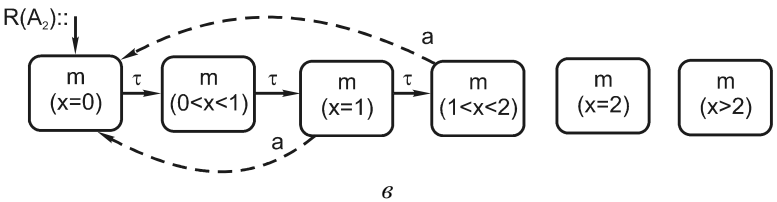
$$[x=2.3] = [x=2.993] = [x=3] = [x=20355.776] = (x>2)$$



а



б



в

Рис. 12.10. *а*) Временные регионы, представляющие классы эквивалентности $\cong$; *б*) классы эквивалентных состояний графа переходов автомата A_2; *в*) региональный автомат для временного автомата A_2

12.7. Граф регионов временного автомата

Расширим отношение эквивалентности $\cong$, определенное на бесконечном множестве интерпретаций локальных часов, на множество состояний системы переходов $\mathsf{T}(A)$ так: два состояния $\mathsf{T}(A)$ попадают в один класс эквивалентности $\cong$ тогда и только тогда, когда их локации совпадают, а интерпретации часов находятся в одном и том же регионе (принадлежат одному и тому же классу эквивалентности $\cong$).

Таким образом, граф регионов временного автомата A — это граф, полученный из системы переходов $\mathsf{T}(A)$ так: одно состояние графа регионов образуют все состояния $\mathsf{T}(A)$, принадлежащие одному и тому же классу эквивалентности $\cong$, т. е. все состояния $\mathsf{T}(A)$ с одной и той же локацией, интерпретации часов которых находятся в одном временном регионе. Формально:

$$\langle loc; v\rangle \cong \langle loc'; v'\rangle \;\Leftrightarrow\; (loc = loc') \wedge \left([v]_{\cong} = [v']_{\cong}\right)$$

Будем обозначать $[s]_{\cong}$ (или просто $[s]$, если отношение эквивалентности $\cong$ очевидно) класс эквивалентности $\cong$, которому принадлежит состояние s графа переходов $\mathsf{T}(A)$.

Пример 12.6

На рис. 12.10, *б* в графе переходов автомата A_2 (см. рис. 12.5) выделены классы эквивалентных состояний. В этом автомате только одни часы с потолком 2. Классы эквивалентности $\cong$ здесь образуются:

- отдельными состояниями, в которых значения локальных часов в точности равны целым 0, 1 и 2;
- теми состояниями, у которых значения часов v заключены между 0 и 1, между 1 и 2 и больше 2.

Классы эквивалентности $\cong$ на множестве состояний $\mathsf{T}(A)$ можно считать состояниями графа регионов $\mathsf{R}(A)$ (его также называют автоматом регионов). Он является фактор-графом переходов графа $\mathsf{T}(A)$ по модулю отношения региональной эквивалентности $\cong$. Поскольку число регионов всегда конечно, то и состояний у любого графа регионов тоже конечное число.

Дадим формальное определение графа регионов. Пусть $A = (Loc, loc_0, \Sigma, X, Inv, E)$ — временной автомат, $\mathsf{T}(A) = (Q, q_0, \Sigma, R)$ — его

система переходов и τ — новый символ, не входящий в алфавит действий автомата A.

❖ ***Определение 12.6 (граф регионов)***

Граф регионов временного автомата A — это граф переходов $\mathsf{R}(A) = \left(Q_{\cong}, \left[q_0\right]_{\cong}, \Sigma_\tau, R_{\cong}\right)$, где:

- $Q_{\cong}$ — множество состояний — это множество классов отношения эквивалентности $\cong$ на состояниях графа переходов $\mathsf{T}(A)$, $Q_{\cong} = \left\{ \left[q\right]_{\cong} \mid q \in Q \right\}$;
- $\left[q_0\right]_{\cong}$ — начальное состояние графа регионов, $\left[q_0\right]_{\cong} = \left\langle loc_0; \left[v_0\right] \right\rangle$, где $v_0(x) = 0$ для часов $x \in X$;
- $\Sigma_\tau = \Sigma \cup \{\tau\}$ — новый алфавит действий, включающий, кроме действий автомата A, новое действие τ. Символ τ будет помечать переход с течением времени, когда локация не меняется (временную задержку);
- $R_{\cong}$ — множество переходов, $R_{\cong} \subseteq Q_{\cong} \times \Sigma_\tau \times Q_{\cong}$ такое, что:
  - для любого действия a автомата A, и любых двух состояний $s, s' \in Q_{\cong}$, $s - a \to s' \in R_{\cong}$ тогда и только тогда, когда существуют два такие состояния q и q' системы переходов $\mathsf{T}(A)$, что $s \in \left[q\right]$, $s' \in \left[q'\right]$ и $q - a \to q' \in R$;
  - для всех $s, s' \in Q_{\cong}$ $s - \tau \to s' \in R_{\cong}$ тогда и только тогда, когда
    - состояния s и s' совпадают, и они представляют неограниченный регион, или
    - состояния s и s' не совпадают, $s \neq s'$, $s = \left\langle loc; \left[v\right] \right\rangle$, $s' = \left\langle loc; \left[v'\right] \right\rangle$, и существует такое $d \in \mathbf{R}^{\geq 0}$, что $\left\langle loc; v \right\rangle - d \to \left\langle loc; v' \right\rangle$ — переход в графе $\mathsf{T}(A)$, и для всех $d_1 \in \mathbf{R}^{\geq 0}$ таких, что $d_1 \leq d$, $\left[v + d_1\right] \in \left\{ \left[v\right] \cup \left[v + d_1\right] \right\}$ (т. е. $\left[v'\right]$ — это соседний с $\left[v\right]$ регион, достижимый из $\left[v\right]$ при удовлетворении ограничений на локальные часы автомата A).

Для каждого региона r следующий регион $succ(r)$, куда попадает граф регионов любого автомата по переходу τ (по задержке времени), всегда определен однозначно.

Граф регионов $\mathsf{R}(A_2)$ временного автомата A_2 представлен на рис. 12.10, *в*. В двух состояниях, $\langle m; (x=2)\rangle$ и $\langle m; (x>2)\rangle$, автомата $\mathsf{R}(A_2)$ не выполняется инвариант $(x<2)$ локации m, поэтому в них время не может прогрессировать. Это пример состояний с остановкой времени (timelock states). У автомата $\mathsf{R}(A_2)$ эти состояния недостижимы.

Итак, состояния графа регионов временного автомата A представляют *множества* состояний системы переходов $\mathsf{T}(A)$. А именно, каждое состояние $\langle loc; [v]\rangle$ графа регионов неявно задает (в общем случае бесконечное) множество таких состояний временного графа, интерпретации v таймеров которых эквивалентны относительно введенной региональной эквивалентности $\cong$. По аналогии с символьным представлением данных с помощью BDD, состояния графа регионов называют "*символьными*" состояниями.

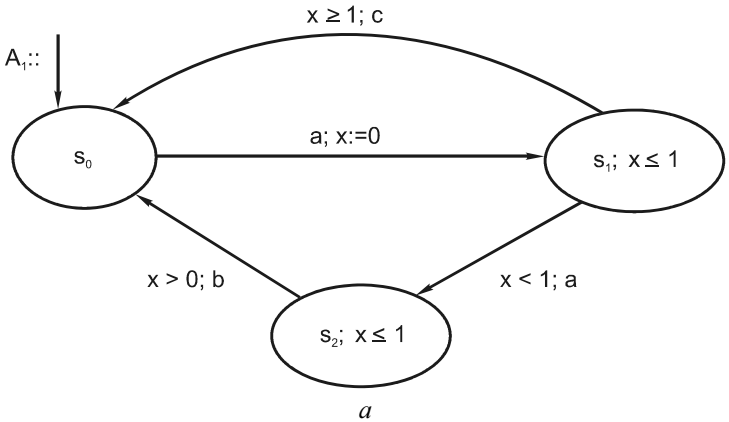
Легко видеть, что граф регионов $\mathsf{R}(A)$ моделирует поведение графа переходов $\mathsf{T}(A)$, поскольку любые два элемента одного и того же временного региона эквивалентны относительно любого свойства, выраженного с помощью условий, наложенных на локации, и ограничений на локальные часы (*clock constraints*). Например, цепочка переходов в $\mathsf{T}(A_2)$ на рис. 12.10, *б*:

$$\langle m; x=0\rangle - 0.1 \to \langle m; x=0.1\rangle - 0.5 \to \langle m; x=0.6\rangle - 0.4 \to \langle m; x=1\rangle - a \to \langle m; x=0\rangle$$

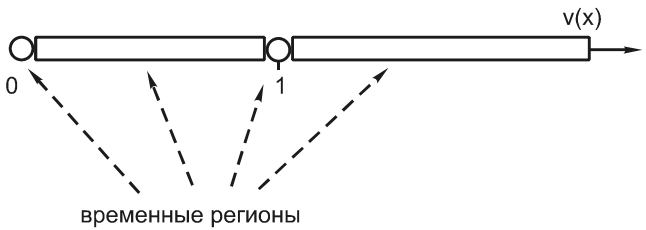
моделируется цепочкой переходов в $\mathsf{R}(A_2)$ (рис. 12.10, *в*):

$$\langle m; (x=0)\rangle - \tau \to \langle m; (0<x<1)\rangle - \tau \to \langle m; (x=1)\rangle - a \to \langle m; (x=0)\rangle$$

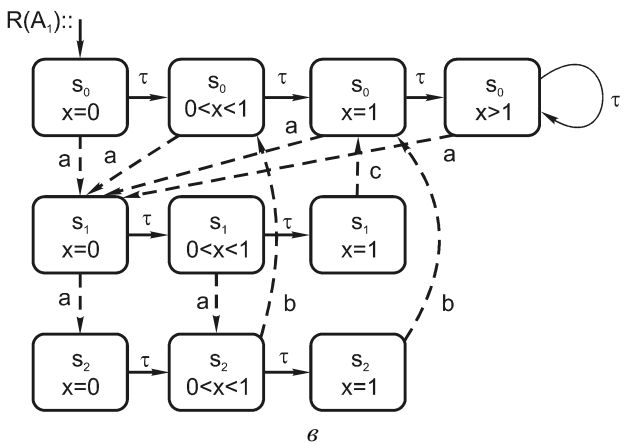
Каждое состояние графа регионов $\mathsf{R}(A)$ является парой ⟨*локация автомата A ; регион*⟩. Принадлежность состояний любого временного автомата p и q одному и тому же региону означает, что если из p достижимо какое-то состояние p', то из q достижимо состояние q', которое находится в одном регионе с p'. Некоторые состояния графа регионов могут быть недостижимы: например, в графе регионов $\mathsf{R}(A_2)$ состояния $\langle m; (x=2)\rangle$ и $\langle m; (x>2)\rangle$ недостижимы. Далее мы будем показывать на рисунках только достижимые состояния графа регионов. Переходы по действиям будем показывать пунктирными стрелками.



а



б



в

Рис. 12.11. *а*) Временной автомат A_1; *б*) временные регионы и *в*) граф регионов $\mathsf{R}(A_1)$

Из разных состояний одного и того же класса эквивалентности в другой класс эквивалентности переходы могут выполняться при различных значениях временных задержек. Точные значения интервалов времени между любыми событиями во временном автомате A можно подсчитать только по вычислениям системы переходов $\mathsf{T}(A)$, которая имеет бесконечное число состояний, и поэтому их невозможно анализировать. Во многих случаях для нас важны только переходы с учетом интервала времени, связанного с переходом, с точностью до целых, что достаточно и для проверки свойств реального времени, выраженных любым множеством ограничений, константы в которых — целые числа.

Если временной автомат с одними локальными часами имеет несколько локаций, то число состояний соответствующего графа регионов может иметь до $(2C_x + 2)^* |Loc|$ состояний. На рис. 12.11 еще раз приведен временной автомат A_1, различающий одинарный и двойной щелчок мыши, и его граф регионов $\mathsf{R}(A_1)$, имеющий 10 состояний.

Статическим анализом графа регионов можно проверить отсутствие во временном автомате вычислений Зенона. На графе регионов можно также проверить, достижимы ли состояния с остановкой времени.

12.8. Временные регионы для нескольких локальных часов

При нескольких локальных часах во временном автомате определение временных регионов несколько усложняется. Например, два состояния, $s_1 = \langle loc; x = 1.1, y = 0.4 \rangle$ и $s_2 = \langle loc; x = 1.4, y = 0.3 \rangle$, не будут эквивалентны относительно всех возможных ограничений на часы, поскольку при возрастании времени в состоянии s_1 всегда сохраняется соотношение $fr(x) < fr(y)$, а в состоянии s_2 это соотношение будет обратным. Отношение дробных частей пары локальных часов определяет порядок достижения целых значений (и, следовательно, следующего региона) различными часами, а от этого порядка может зависеть возможное дальнейшее поведение временного автомата.

Если система реального времени находится в состоянии $s_1 = \langle loc; x = 1.1, y = 0.4 \rangle$, то через 0.6 единиц времени она перейдет в состояние $s_1' = \langle loc; x = 1.7, y = 1 \rangle$, а из состояния $s_2 = \langle loc; x = 1.4, y = 0.3 \rangle$ через 0.6 единиц времени она перейдет в состояние $s_2' = \langle loc; x = 2, y = 0.9 \rangle$ (если инвариант локации loc остается истинным). Иными словами, в первом слу-

чае при естественном течении времени следующим регионом системы будет $r_1 = [x = 1.7,\ y = 1] = (1 < x < 2,\ y = 1)$, а во втором случае следующим регионом будет $r_2 = [x = 2,\ y = 0.9] = (x = 2,\ 1 < y < 2)$. Поэтому определение временных регионов при нескольких часах должно включать и сохранение отношения дробных частей значений всех пар часов.

Таким образом, определение отношения региональной эквивалентности для нескольких локальных часов должно быть уточнено по сравнению с подобным определением для одних часов.

> ❖ ***Определение 12.7 (региональная эквивалентность для нескольких локальных часов)***
>
> Две интерпретации v и v' конечного множества локальных часов X находятся в отношении региональной эквивалентности $\cong$ тогда и только тогда, когда:
>
> - для всех часов $x \in X$:
>
> $v(x) > C_x \wedge v'(x) > C_x$ в обеих интерпретациях значения всех часов превышают свой потолок, или
>
> - для всех пар часов $(\forall x, y \in X)$, если они не превышают свой потолок $(v(x), v'(x) \le C_x)$ и $(v(y), v'(y) \le C_y)$, то:
>
> ▫ $\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor \wedge (\ fr((v(x)) = 0 \equiv fr((v(x')) = 0\ )$ все они имеют совпадающие в обеих интерпретациях целые значения;
>
> и
>
> ▫ $(\ fr(v(x)) \le fr(v(y)) \equiv fr(v'(x)) \le fr(v'(y))\ )$ для каждой пары часов отношения между их дробными частями в обеих интерпретациях совпадают.

Векторы показаний часов, определяемые этими правилами, образуют классы эквивалентности (временные регионы) на множестве наборов неотрицательных вещественных чисел, характеризуемые следующими отношениями:

❒ для каждых локальных часов $x \in X$:

- всеми значениями целых 0, 1, ..., C_x;
- всеми открытыми интервалами между соседними целыми $(N,\ N+1)$ для $N < C_x$;
- интервалом (C_x, ∞);

- для каждой пары локальных часов $x, y \in X$, значения которых находятся в интервалах $(m < x < m+1)$ и $(n < y < n+1)$:
  - отношениями равенства дробных частей их значений $fr(x)$ и $fr(y)$ или их отношения $<$ или $>$.

Каждый регион определяется набором ограничений на часы (clock constraints). Число временных регионов определяется следующим соотношением:

$$|X|!\, 2^{|X|} \prod_{x \in X} (2C_x + 2).$$

Действительно, для часов x с потолком C_x существует $2C_x + 2$ возможных интервалов, и для каждых часов существует независимая возможность выбора интервала. Кроме того, для всех часов с ненулевыми дробными часами интервалы отличаются друг от друга также упорядоченностью этих дробных частей. Верхней границей числа таких перестановок является величина $|X|!$. Третий компонент оценки ограничивает сверху число тех часов, дробные части которых совпадают.

Пример 12.7

Структура регионов для пары часов x, y с максимальными значениями $C_x = 2$, $C_y = 1$ представлена на рис. 12.12. Эта структура содержит 28 регионов. Пусть граф регионов для временного автомата с двумя локальными часами x и y (рис. 12.12) находится в состоянии $<loc; [v_1]>$. На рисунке интерпретация часов v_1 находится в регионе $[v_1]$, который характеризуется следующими временными ограничениями:

$$(1 < x < 2) \wedge (0 < y < 1) \wedge (fr(x) < fr(y))$$

или, что то же:

$$(1 < x < 2) \wedge (0 < y < 1) \wedge (x - y < 1).$$

Этот регион объединяет интерпретации часов $(x = 1.25, y = 0.56)$, $(x = 1.1, y = 0.72)$, $(x = 1.1, y = 0.4)$ и т. п.

Рассмотрим примеры возможных переходов для этого графа регионов:

- переход по действию. Если выполнение перехода $\langle loc; [v_1] \rangle - a \rightarrow \langle loc'; [v_1'] \rangle$ сопровождается присваиванием $y := 0$ (штриховая стрелка вниз из v_1 на рис. 12.12), то каждая интерпретация часов из этого региона

попадает в один и тот же регион $[v_1']$, который определяется ограничениями: $(1 < x < 2) \wedge (y = 0)$. Присваивание $x := y$ значений локальных часов на переходе временного автомата из любой интерпретации часов в этом регионе (штриховая стрелка влево) приведет к переходу в состояние с той же локацией loc, но с интерпретацией часов в регионе $v'' = (0 < x = y < 1)$;

- при продолжении хода часов из интерпретации v_1 все интерпретации $v_1 + d$ для любого неотрицательного d располагаются на сплошной стрелке, идущей под 45°, τ.

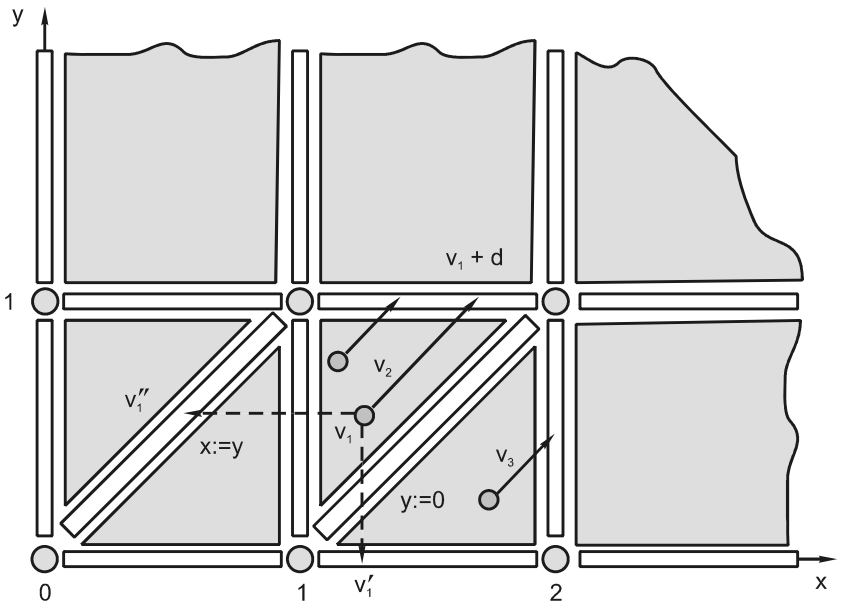


Рис. 12.12. Структура регионов для двух часов x, y с потолками 2 и 1

ЛЕММА 12.1 (свойства регионов)

Для любых двух интерпретаций часов v, u, находящихся в одном регионе $([v] = [u])$, выполняются следующие свойства:

- $\left(\forall d \in \mathbf{R}^{\geq 0}\right)\left(\exists d' \in \mathbf{R}^{\geq 0}\right)\left([v + d] = [u + d']\right)$

если добавлением любой неотрицательной величины d к показаниям v локальных часов мы попадаем в какой-нибудь регион r, то существует

такая величина d', что добавлением d' к показаниям локальных часов u мы попадем в тот же регион r;

- $\left[v\left(x:=0\right)\right]=\left[u\left(x:=0\right)\right]$

  при сбросе значений одних и тех же часов из интерпретаций v, u мы попадем в один и тот же регион;

- $\left(\forall n\in \mathsf{N}\right)\left(\left[v+n\right]=\left[u+n\right]\right)$

  добавление натурального n к показаниям всех часов переводит эти две интерпретации в один и тот же регион;

- $\left(\forall \gamma\in \Gamma\right)\left(\left[v\right]\models\gamma \equiv \left[u\right]\models\gamma\right)$

  интерпретации часов v, u эквивалентны относительно любого свойства, выраженного с помощью ограничений на локальные часы.

Рис. 12.12 иллюстрирует эту лемму. Действительно, пусть $v_1=\left(x=1.3,\, y=0.4\right)$, $v_2=\left(x=1.1,\, y=0.72\right)$. В соответствии с определением 12.7, эти две интерпретации часов эквивалентны. Переход автомата по временной задержке как из состояния $\left\langle loc; v_1\right\rangle$, так и из состояния $\left\langle loc; v_2\right\rangle$ приведет в один и тот же следующий регион $\left(1<x<2,\, y=1\right)$. Если интерпретации часов находятся в разных регионах (например, v_1 и v_3 на рис. 12.12), то следующие регионы, в которые будет выполнен переход по временной задержке из соответствующих состояний, будут разные.

Пример 12.8

Временной автомат A_5 с двумя таймерами (локальными часами) x, y и его граф регионов представлены на рис. 12.13. Любая последовательность состояний и переходов графа регионов $\mathsf{R}\left(A_5\right)$ представляет вычисления временного автомата A_5. Одна из возможных последовательностей изменения часов x и y в этом автомате при следующем вычислении графа регионов $\mathsf{R}\left(A_5\right)$:

$$\pi = q_1 - \tau \to q_2 - \tau \to q_3 - \tau \to q_4 - a \to q_8 - \tau \to q_{12} - \tau \to q_{14} - \tau \to$$
$$q_{10} - \tau \to q_{11} - c \to q_{13} - \tau \to q_{10} - \tau \to q_{11} - c \to q_{13} - \tau \to q_{10} - \tau \to ...$$

показана на рис. 12.14. Каждый переход τ (задержка по времени) переводит автомат $\mathsf{R}\left(A_5\right)$ в соседний регион по направлению изменения обоих локальных часов.

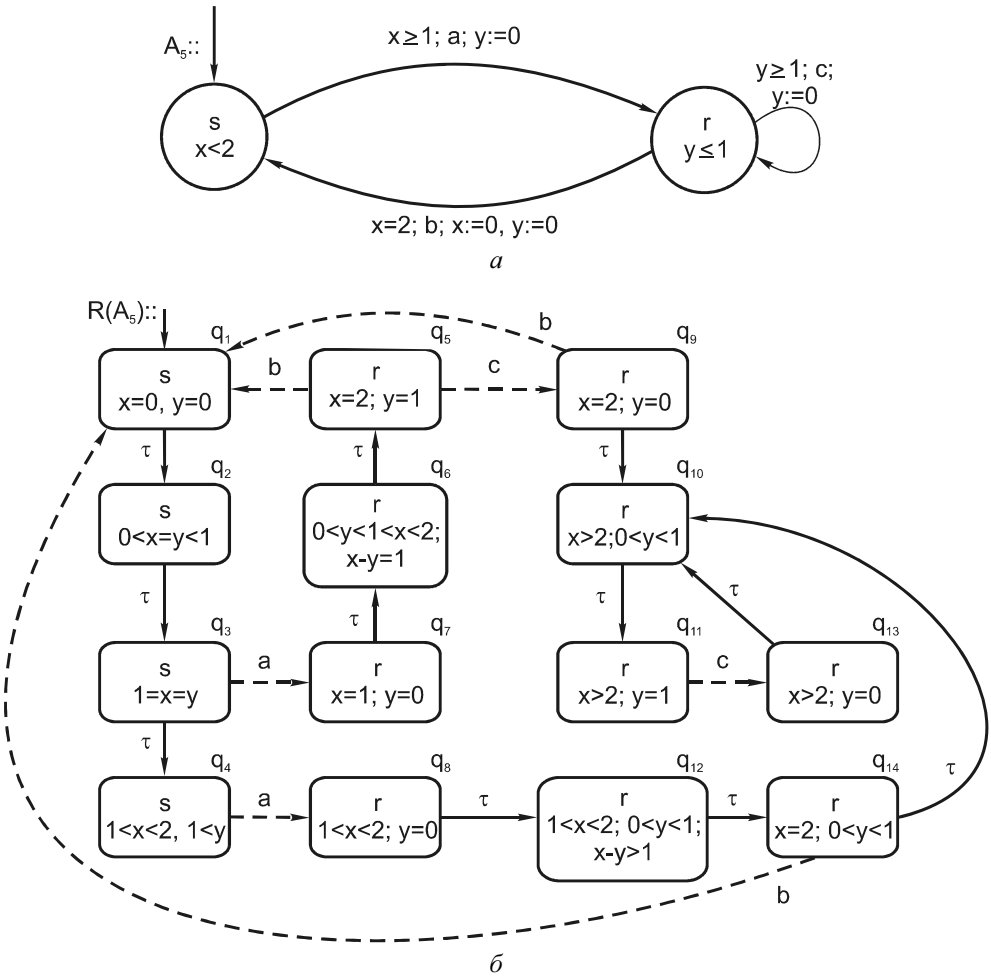


Рис. 12.13. *а*) Временной автомат A_5 с двумя таймерами;
б) его граф регионов

Многие свойства системы переходов временного автомата можно проверить по его графу регионов, который имеет конечное число состояний. Например, по графу регионов решается проблема достижимости состояний временного автомата: состояние $\langle loc; v\rangle$ достижимо во временном автомате A тогда и только тогда, когда регион $\left[\langle loc; v\rangle\right]$ достижим в графе регионов $\mathsf{R}(A)$ этого автомата. В частности, из рис. 12.13 можно заключить, что состояние $\langle r; x=1, y=1\rangle$ автомата A_5 недостижимо из его начального состояния $\langle s; x=0, y=0\rangle$.

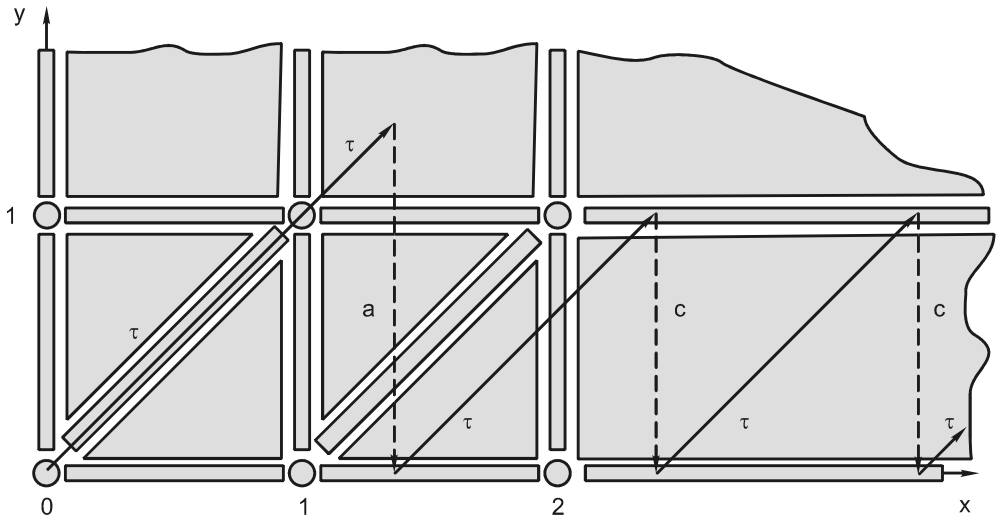


Рис. 12.14. Последовательность изменений показаний часов x и y на одном из вычислений автомата A_5

12.9. Анализ временных автоматов: логика CTL

Верификацию временного автомата A при спецификации его свойств формулами логики CTL можно свести к анализу выполнения этих формул для графа регионов $\mathsf{R}(A)$. Этот граф строится как раз таким образом, чтобы для любых двух наборов значений часов v и w, которые принадлежат одному временному региону, два состояния, $\langle loc;[v]\rangle$ и $\langle loc;[w]\rangle$, графа регионов удовлетворяли одним и тем же формулам темпоральной логики. Если Ф — формула CTL, то:

$$\left([v]=[w]\right)\Rightarrow\left(\langle loc;[v]\rangle\models\Phi\equiv\langle loc;[w]\rangle\models\Phi\right).$$

При верификации графа регионов используется обычный для CTL алгоритм разметки состояний структуры Крипке теми подформулами проверяемой формулы, которые в этих состояниях выполняются. В формулах логики CTL в этом случае не используется темпоральный оператор **X** (*NextTime*) как с квантором существования **EX**, так и с квантором всеобщности **AX**: в системах с непрерывным временем говорить о "*следующем моменте времени*" для систем реального времени бессмысленно.

В качестве атомарных предикатов формул CTL при анализе систем реального времени можно использовать не только атомарные предикаты (в том числе имена локаций), но также и соотношения между значениями локальных часов временного автомата, структура которых определена грамматикой:

$$\gamma ::= x \sim c \mid x - y \sim c \mid \neg\gamma \mid \gamma \wedge \gamma,$$

где c — натуральное число, x, y — локальные часы, а отношение $\sim$ принадлежит множеству $\{\leq, <\}$. Таким образом, формулами CTL при проверке свойств временных автоматов могут быть логические формулы, включающие как атомарные предикаты, так и ограничения на локальные часы. Например, формула CTL:

$$\mathbf{EF}\left(p \wedge (x < 3) \wedge (x - y > 0)\right)$$

говорит о том, что в автомате из начального состояния существует путь, приводящий в состояние, в котором выполняется атомарный предикат p, и при этом показание локальных часов x меньше 3 и больше, чем показание локальных часов y.

Рассмотрим несколько примеров.

Пример 12.9

Проверим, выполняется ли во временном автомате A_5 рис. 12.13 свойство $\mathbf{EFAG}\,r$ — существует такое вычисление, на котором найдется состояние, из которого все дальнейшие вычисления будут проходить через состояния, помеченные r.

Формулу $\mathbf{AG}\,r$ преобразуем так: $\mathbf{AG}\,r = \neg\mathbf{E}\neg\mathbf{G}\,r = \neg\mathbf{EF}\neg r$. По графу регионов $\mathsf{R}(A_5)$ (см. рис. 12.13) последовательно находим множество его состояний, удовлетворяющих подформулам формулы $\phi = \mathbf{EF}\neg\mathbf{EF}\neg r$:

$f_1 = \neg r;$ $\quad Sat(f_1) = \{q_1, q_2, q_3, q_4\}$

$f_2 = \mathbf{EF}\,f_1;$ $\quad Sat(f_2) = \{q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{12}, q_{14},\}$

$f_3 = \neg f_2;$ $\quad Sat(f_3) = \{q_{10}, q_{11}, q_{13}\}$

$\phi = f_4 = \mathbf{EF}\,f_3;$ $\quad Sat(\phi) = \{q_1, \ldots, q_{14}\}$

Поскольку начальное состояние q_1 графа регионов $\mathsf{R}(A_5)$ входит в множество $Sat(\phi)$, то формула $\mathbf{EFAG}r$ выполняется в $\mathsf{R}(A_5)$ и, следовательно, выполняется во временном автомате A_5.

С помощью графа регионов проблема достижимости во временном автомате решается достаточно просто. Рассмотрим решение этой проблемы на примере.

Пример 12.10

Один из вариантов протокола взаимного исключения Фишера для параллельной программы, состоящей из двух последовательных процессов P_1 и P_2, формулируется так. Процессы P_1 и P_2 взаимодействуют через разделяемую переменную f. Каждый из процессов независимо устанавливает значение разделяемой переменной соответственно в 1 или 2, а затем входит в свою критическую секцию при условии, что установленное значение разделяемой переменной сохранилось тем, которое было этим процессом установлено:

```
P1::                          P2::
  a1:  f:=1;                    a2:  f:=2;
  b1:  wait (f=1);              b2:  wait (f=2);
  cs1: Critical_1               cs2: Critical_2
```

Почти очевидно, что этот протокол некорректен. При анализе параллельных программ их обычно рассматривают как совокупности асинхронных процессов, абстрагируясь от времени выполнения операций. При такой абстракции этот протокол действительно некорректен. Из параллельной композиции соответствующих структур Крипке можно заключить, что в этом протоколе существуют вычисления, приводящие к состоянию, в котором оба процесса находятся в своих критических секциях. Одно из таких вычислений:

$$\langle a_1, a_2, f = \perp \rangle \to \langle a_1, b_2, f = 2 \rangle \to \langle a_1, cs_2, f = 2 \rangle \to \langle b_1, cs_2, f = 1 \rangle \to \langle cs_1, cs_2, f = 1 \rangle .$$

На этом вычислении свойство взаимного исключения нарушается: на нем достижимо некорректное глобальное состояние $cs_1 \wedge cs_2$: оба процесса вошли в свои критические секции. В терминах model checking для этой модели выполняется формула $\mathbf{EF}(cs_1 \wedge cs_2)$ логики CTL.

Проведем более тонкий анализ протокола Фишера. Проверим, достижимо ли некорректное состояние в этой параллельной программе, если временем выполнения операций не пренебрегать, как это делается в асинхронной модели. Временные автоматы, представляющие этот протокол, показаны на рис. 12.15, *а*. Для учета времени выполнения операций используются локальные часы x и y этих процессов. Отличие от асинхронной модели состоит в том, что временные автоматы, моделирующие взаимодействующие процессы, учитывают здесь время выполнения операции проверки значения разделяемой переменной f: каждый процесс должен установить общую переменную

f менее чем через 1 единицу времени после старта, а операцию проверки значения f не может выполнить менее, чем за 1 единицу времени после прихода вычислений на метку b_i.

Граф регионов $\mathsf{R}(P_1 \| P_2)$ содержит около 70 состояний. Фрагмент, показывающий два возможных вычисления этого графа, представлен на рис. 12.15, *в*. Анализ показывает, что после входа одного из процессов в свою

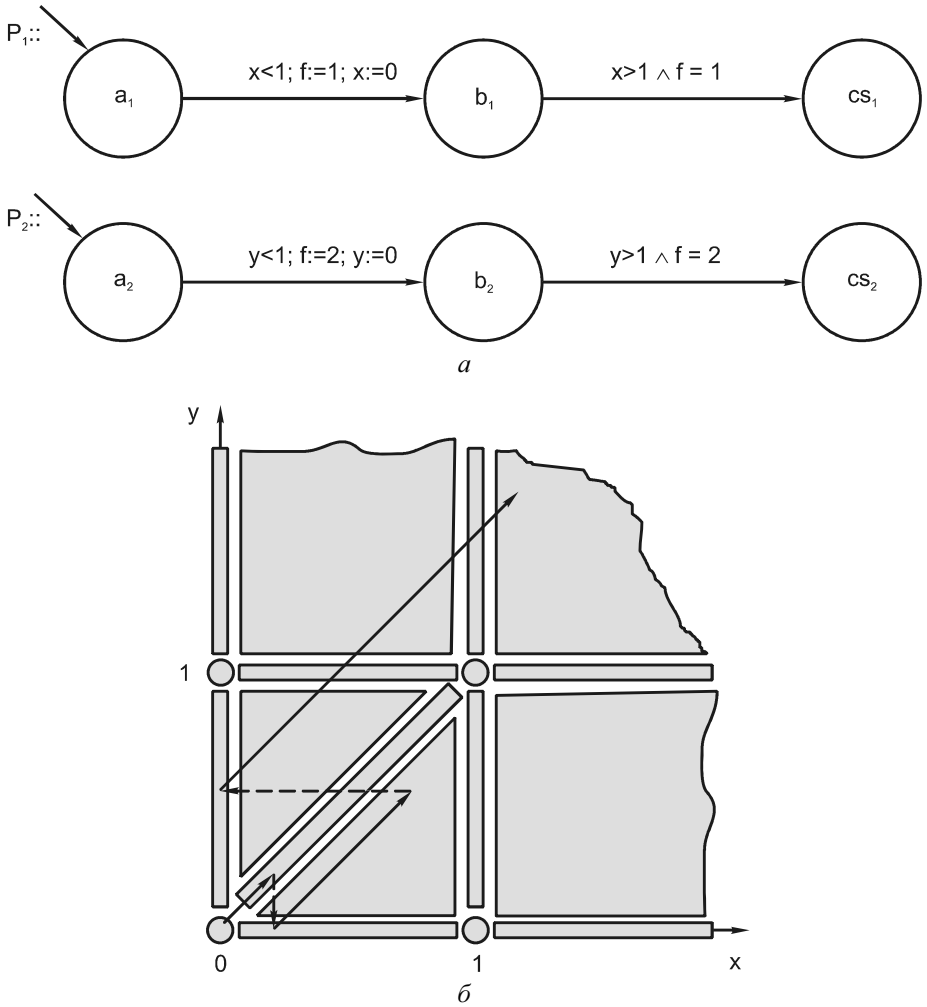


Рис. 12.15. *а*) Временные автоматы для протокола Фишера; *б*) регионы для двух локальных часов, $C_x = C_y = 1$

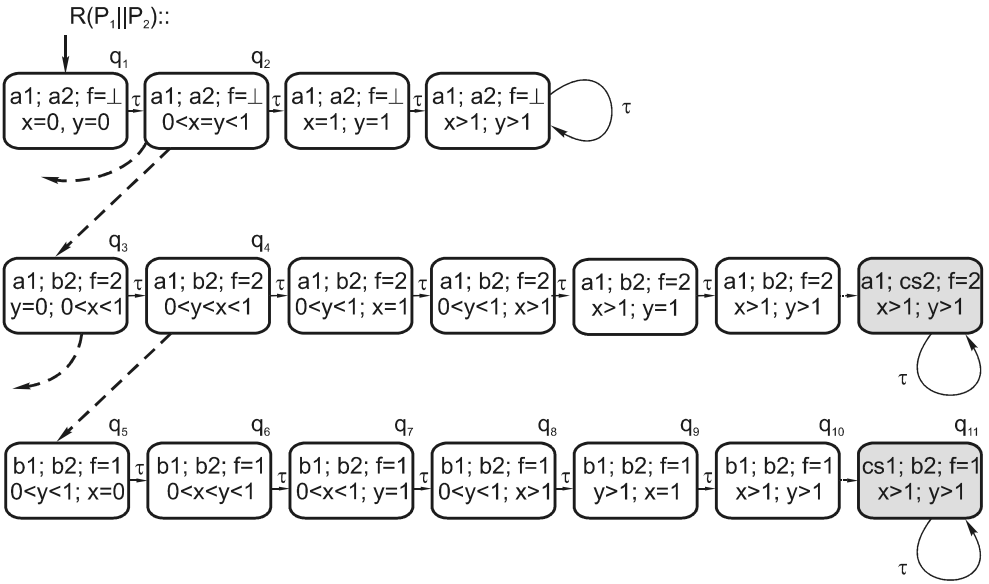


Рис. 12.15. *в*) Фрагмент графа регионов для протокола Фишера

критическую секцию, изменение значения общей переменной f другим процессом невозможно. При любом развитии вычислений два процесса не могут находиться в своих критических секциях одновременно. На структуре регионов рис. 12.15, *б* показаны изменения значений локальных часов этой параллельной программы для траектории вычисления графа регионов $q_1 q_2 \ldots q_{11}$. Можно сделать вывод, что протокол взаимного исключения Фишера корректен: глобальное состояние, удовлетворяющее формуле $cs_1 \wedge cs_2$ недостижимо: формула $\mathbf{EF}(cs_1 \wedge cs_2)$ на графе регионов $\mathsf{R}(P_1 \| P_2)$ не выполняется.

Пример 12.11

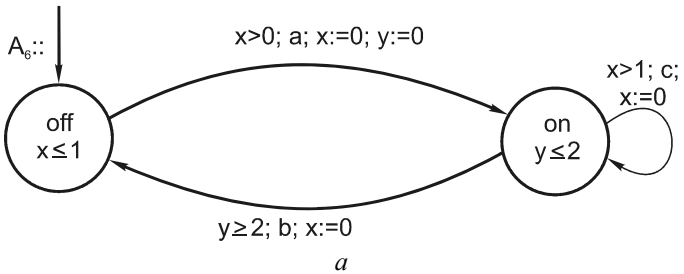
На рис. 12.16 приведен временной автомат A_6 и его граф регионов $\mathsf{R}(A_6)$. На структуре регионов рис. 12.16, *б* показаны изменения значений локальных часов автомата A_6 на вычислении $\pi = q_3 q_4 q_5 q_6 q_8 q_9 q_{10} q_{12} q_{13} q_3$.

Проверим, выполняется ли в автомате A_6 свойство:

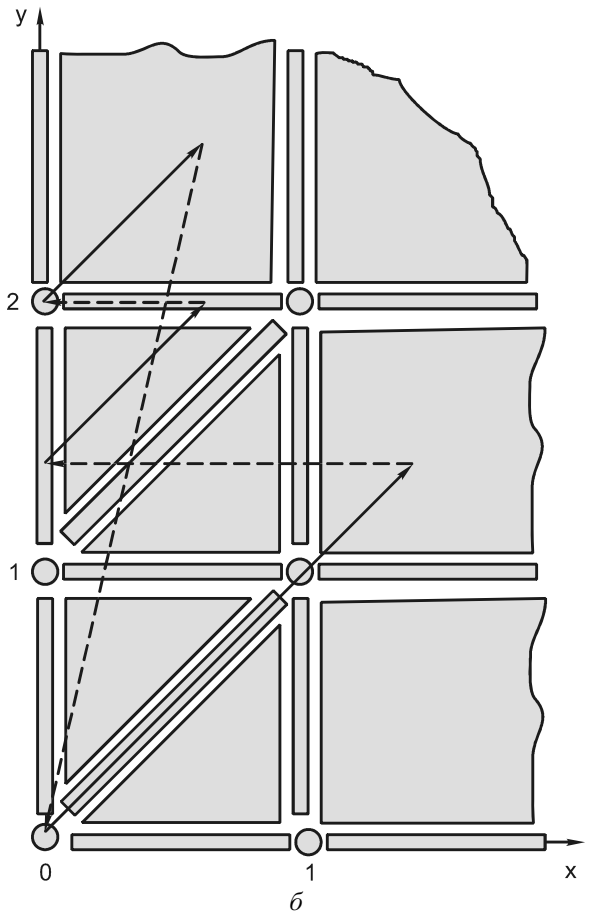
$$\varphi = \mathbf{E}\Big[@off \ \mathbf{U} \left(\mathbf{AF}\big((0 < x < 1) \wedge (y = 2)\big)\right)\Big],$$

а именно, в автомате A_6 существует такое вычисление, на котором автомат будет находиться в состоянии *off* до тех пор, пока не придет в состояние, из

которого на каждом вычислении достижимо состояние, в котором показания локальных часов x находятся между 0 и 1, а локальные часы y показывают 2.



а



б

Рис. 12.16, *а* и *б*. Временной автомат A_6 и изменение его часов на одном из вычислений

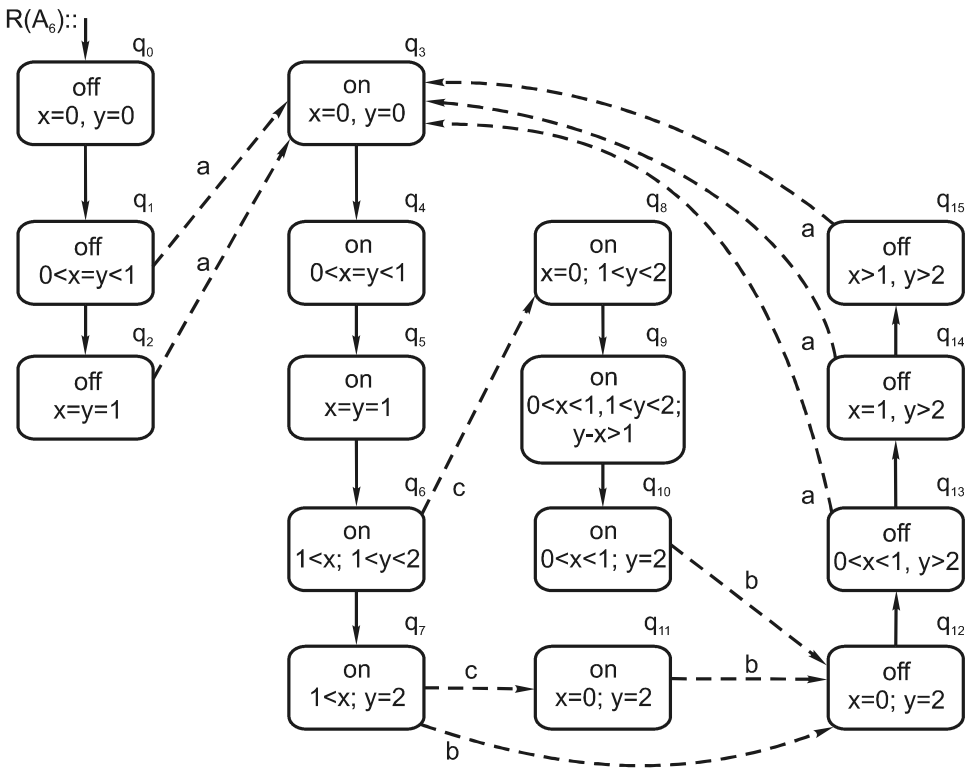


Рис. 12.16, *в*. Граф регионов автомата A_6

По графу регионов $\mathsf{R}(A_6)$ последовательно находим множества его состояний, удовлетворяющих подформулам формулы φ:

$f_1 = (0 < x < 1);$ $\quad Sat(f_1) = \{q_1, q_4, q_9, q_{10}, q_{13}\}$

$f_2 = (y = 2);$ $\quad Sat(f_2) = \{q_7, q_{10}, q_{11}, q_{12}\}$

$f_3 = f_1 \wedge f_2;$ $\quad Sat(f_3) = \{q_{10}\}$

$f_4 = \mathbf{AF} f_3;$ $\quad Sat(f_4) = \{q_8, q_9, q_{10}\}$

$f_5 = @off;$ $\quad Sat(f_5) = \{q_0, q_1, q_2, q_{12}, q_{13}, q_{14}, q_{15}\}$

$\varphi = \mathbf{E}(f_5 \mathbf{U} f_4);$ $\quad Sat(\varphi) = \{q_8, q_9, q_{10}\}$

Начальное состояние q_0 не входит в множество состояний, помеченных φ. Поэтому формула φ не выполняется в графе регионов $\mathsf{R}(A_6)$ и, следовательно, во временном автомате A_6.

Отметим, что при проверке свойств, выраженных в логике CTL, обычно принимаются во внимание все возможные траектории временного автомата, в то время как при проверке выполнимости формул логики TCTL, рассматриваемой в следующем разделе, анализируются только дивергентные вычисления.

12.10. Временная темпоральная логика ветвящегося времени (Timed CTL)

Для спецификации свойств систем реального времени важно иметь возможность выразить не только обычные темпоральные свойства, определяющие достижимость состояний, порядок событий в вычислениях или условия, выраженные через показания локальных часов временного автомата, но и количественную информацию, задающую временной диапазон действия темпоральных операторов. Примеры таких "жестких" требований реального времени:

- "*реакция на любой запрос наступит не позже, чем через 10 единиц времени*" (bounded response);
- "*событие c не может наступить раньше, чем через 7 единиц времени после последнего наступления события b*";
- "*временной интервал между событиями a и b составляет от l до u, не включая u*".

Классическая темпоральная логика не может выразить подобные ограничения на вычисления. Для выражения таких свойств было предложено несколько логик. Мы рассмотрим здесь Timed CTL (TCTL) — временную темпоральную логику ветвящегося времени. В TCTL действие темпорального оператора **U** и выводимых операторов **F** и **G** ограничено временными рамками, которые указываются непосредственно у темпорального оператора.

Например, требование того, что всегда после посылки запроса ответ придет не более чем через 10 единиц времени, можно выразить так:

$$\mathbf{AG}\left(request \Rightarrow \mathbf{AF}_{\leq 10} response\right)$$

Грамматика TCTL (Timed CTL). Грамматика формул логики TCTL определяет формулы состояний ϕ:

$$\phi ::= p \mid \gamma \mid \neg\phi \mid \phi \vee \phi \mid \mathbf{E}\left[\phi_1 \mathbf{U}_J \phi_2\right] \mid \mathbf{A}\left[\phi_1 \mathbf{U}_J \phi_2\right],$$

где p — атомарный предикат, γ — условия, наложенные на значения локальных часов, а J — временной интервал, $J \subseteq \mathbf{N}$. Атомарные утверждения в TCTL включают не только атомарные предикаты, но и предикаты, постро-

енные над значениями локальных часов — все условия, истинность которых можно вычислить в состояниях временного автомата. Логические связки трактуются как обычно.

Если интервал J у темпорального оператора $\phi_1 \mathbf{U}_J \phi_2$ равен $[0, \infty)$, то его можно опустить. Логику TCTL, в которой в качестве временных ограничений используется только интервал $[0, \infty)$, можно рассматривать как собственный подкласс логики CTL. В дальнейшем будем считать, что интервал J в формулах TCTL не равен $[0, \infty)$. Некоторые типы интервалов можно обозначать кратко:

$\leq n$ заменяет $[0, n]$

$< n$ заменяет $[0, n)$

$\geq n$ заменяет $[n, \infty)$

$> n$ заменяет (n, ∞)

$= n$ заменяет $[n, n]$

Для выводимых модальных операторов $\mathbf{F}$ и $\mathbf{G}$ синтаксис временных интервалов определяется аналогично:

$$\mathbf{F}_J \phi = [true \, \mathbf{U}_J \phi]$$

$$\mathbf{G}_J \phi = \neg \mathbf{F}_J \neg \phi$$

Неформальный смысл этих конструкций следующий.

Формула $\mathbf{F}_J \phi$ для пути π утверждает, что на этом пути из текущего состояния формула ϕ выполнится когда-нибудь в будущем во временном интервале J. Формула $\mathbf{G}_J \phi$ для пути π утверждает, что на этом пути из текущего состояния формула ϕ будет непрерывно выполняться во временном интервале J. Интерпретация формулы $\phi_1 \mathbf{U}_J \phi_2$ имеет свои особенности, которые мы рассмотрим ниже.

Пример 12.12

Рассмотрим примеры формул с явным указанием интервала их действия.

$\mathbf{F}_{\leq 10} \varphi$ для пути π: менее чем через 10 единиц времени на этом пути встретится состояние, удовлетворяющее φ. Если перед этой формулой пути стоит квантор существования $\mathbf{E}$, то формула $\mathbf{E}\mathbf{F}_{\leq c} \varphi$ определяет свойство *ограни-*

ченной достижимости (bounded reachability), формула $\mathbf{AF}_{\leq c}\varphi$ определяет свойство *ограниченной неизбежности* (bounded inevitability).

$\mathbf{AG}\left(problem \Rightarrow \mathbf{AF}_{\leq 5}\ alarm\right)$ — как только возникнет проблема, не более чем через 5 единиц времени зазвучит сигнал тревоги.

$\mathbf{EG}\left[send(m) \Rightarrow \mathbf{EF}_{>4} recover(r_m)\right]$ — существуют вычисления, на которых подтверждение на посланное сообщение может быть получено более чем за 4 единицы времени.

$\mathbf{AG}\left(on \Rightarrow \mathbf{AF}_{\leq 10}\ off\right)$ — свет не может непрерывно гореть больше 10 единиц времени.

$\mathbf{A}\left[off\ \mathbf{U}_{\leq 15} on\right]$ — свет будет включен не более чем через 15 единиц времени.

$\mathbf{AG}\left[send(m) \Rightarrow \mathbf{AF}_{(5,10]} recover(m)\right]$ — сообщение m обязательно будет получено не ранее, чем через 5 единиц времени, но не позже, чем через 10 единиц времени после посылки.

$\mathbf{EF}_{=M}\ @s$ — из текущего состояния найдется путь, который оканчивается в состоянии, помеченном s, и его длительность составляет точно M единиц времени.

Вследствие того, что в модели временного автомата используется понятие "плотного" (dense) реального времени, в языке формул TCTL отсутствуют пары **AX** и **EX**: в системах переходов временного автомата не существует понятия "*следующего*" момента времени.

Интерпретация формул TCTL

Формулы невременной темпоральной логики CTL интерпретируются на деревьях вычислений конечных структур Крипке, на которых семантика этих формул определяется достаточно естественно. Семантика CTL-формулы $\phi_1 \mathbf{U} \phi_2$ на вычислении $\pi = q_0 q_1 q_2 q_3 \ldots$ структуры Крипке определяется так: в каком-то из состояний вычисления π выполняется формула ϕ_2, а во всех предыдущих состояниях выполняется ϕ_1.

Формулы логики TCTL с временными ограничениями интерпретируются на деревьях вычислений временных автоматов. Состояниями вычисления временного автомата являются пары $\langle loc; v \rangle$, где loc — локация (возможно, набор атомарных предикатов, истинных в этом состоянии), а v — вектор значений локальных часов. Интерпретация формул TCTL должна учитывать

природу непрерывного времени, которая имеет следующие особенности. На любом вычислении временного автомата, между любыми двумя состояниями $\langle loc; v\rangle$ и $\langle loc; v+d\rangle$, связанными задержкой времени d:

$$\langle loc; v\rangle - d \to \langle loc; v+d\rangle$$

может быть произвольное число состояний:

$$\langle loc; v\rangle - d_0 \to \langle loc; v+d_0\rangle - d_1 \to \langle loc; v+d_0+d_1\rangle \to ... - d_{n-1} \to$$
$$\langle loc; v+d_0+d_1+...+d_{n-1}\rangle$$

таких, что:

$$d_0 + d_1 + ... + d_{n-1} = d$$

Очевидно, что два вычисления π и π', отличающиеся только разбиениями некоторых задержек на несколько частей, не должны рассматриваться как различные, и любая формула Φ логики TCTL должна быть одновременно истинна или одновременно ложна для этих двух вычислений:

$$\pi \models \Phi \equiv \pi' \models \Phi .$$

Отметим, что количество интервалов при таком разбиении не может быть бесконечным: при анализе свойств систем реального времени от конвергентных вычислений можно абстрагироваться как от нереалистичных подобно тому, как при анализе структуры Крипке абстрагируются от несправедливых (unfair) вычислений. Но и в дивергентных вычислениях возможность произвольного дробления задержек в вычислении приводит к необходимости несколько изменить семантику формулы $\phi_1 \mathbf{U}_J \phi_2$ логики TCTL.

Рассмотрим эту особенность подробнее. Пусть семантика формулы пути $\phi_1 \mathbf{U}_J \phi_2$ на вычислении π временного автомата определена так: в некотором состоянии вычисления π во временном интервале J от начального состояния π выполняется формула ϕ_2; а на всех предыдущих состояниях π выполняется формула $\phi_1 \wedge \neg\phi_2$.

Рассмотрим выполнимость формулы $\phi_1 \mathbf{U}_J \phi_2$ для автомата A_7, рис. 12.17, *а*, на вычислении σ:

$$\sigma = s_0 - 0.7 \to s_1 - a \to s_2 - 0.3 \to s_3 - 0.5 \to s_4 \to ...$$

где:

$$s_0 = \langle m; x=0\rangle; s_1 = \langle m; x=0.7\rangle; s_2 = \langle n; x=0\rangle; s_3 = \langle n; x=0.3\rangle; s_4 = \langle n; x=0.8\rangle .$$

На рис. 12.17, *б* приведен график зависимости от времени значений локальных часов x автомата A_7 на этом вычислении.

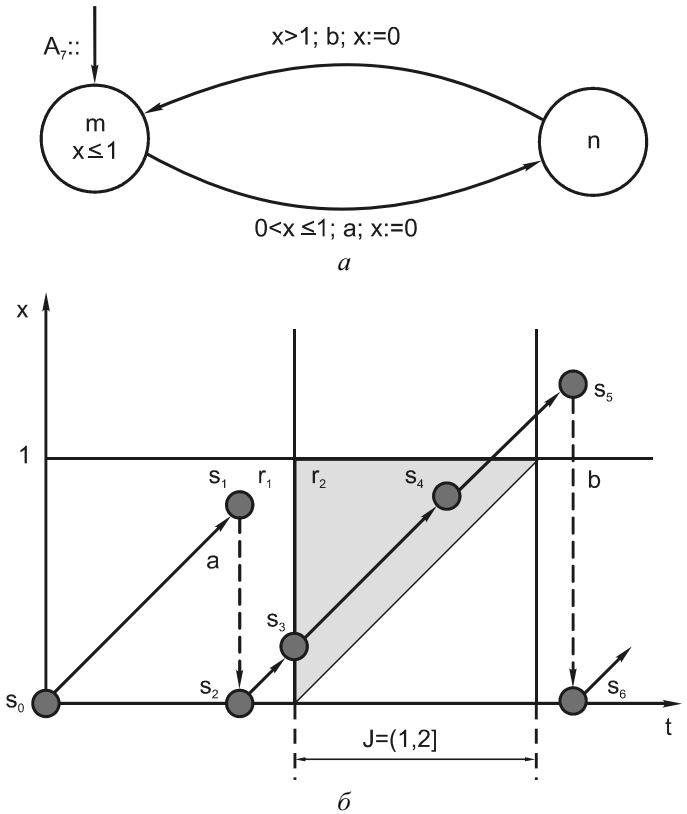


Рис. 12.17. К определению семантики формулы $\phi_1 \mathbf{U}_J \phi_2$ для верификации временных автоматов

Пусть на вычислении σ в состоянии s_4 выполняется формула ϕ_2, а на всех состояниях от s_0 до s_3 включительно выполняется формула $\phi_1 \wedge \neg\phi_2$. Но на вычислении σ не существует состояния, на котором выполняется ϕ_2, а на всех предыдущих его состояниях выполняется формула $\phi_1 \wedge \neg\phi_2$. Действительно, между s_3 и s_4 всегда найдется состояние в регионе r_2, в котором будет, как и в s_4, выполняться ϕ_2 (рис. 12.17, *б*).

Очевидно, что семантика формулы $\phi_1 \mathbf{U}_J \phi_2$ для временных автоматов должна быть изменена так. Формула $\phi_1 \mathbf{U}_J \phi_2$ выполняется на вычислении σ, если на σ во временном интервале J в какой-то момент времени встретится состояние, в котором выполняется атомарный предикат ϕ_2, а вплоть до этого момента (и включая его) в любом состоянии σ будет выполняться $\phi_1 \vee \phi_2$.

Заметим, что в логике CTL (так же, как и в логике LTL) формулы $\phi_1 \mathbf{U} \phi_2$ и $(\phi_1 \vee \phi_2) \mathbf{U} \phi_2$ эквивалентны.

Второй особенностью вычислений временных автоматов является то, что в интервале J может не находиться ни одного состояния вычисления.

Проверим выполнимость формулы $m \mathbf{U}_{(1,2]} n$ для вычисления π временного автомата A_7 (рис. 12.17):

$$\pi = s_0 - 0.7 \to s_1 - a \to s_2 - 1.4 \to s_5 \to \ldots \ .$$

Интуитивно ясно, что формула $m \mathbf{U}_{(1,2]} n$ должна выполняться на любом вычислении автомата A_7: стартуя из начального состояния $\langle m; x = 0 \rangle$, в какой-то момент времени $t \in J$ мы обязательно будем находиться в состоянии с локацией n, а до этого момента — либо в локации m, либо в локации n. Но на вычислении π в интервале $J = (1,2]$ вообще нет выделенных состояний. Как определить семантику формулы $\phi_1 \mathbf{U}_J \phi_2$ на таких вычислениях?

Проблемы, возникающие при анализе вычислений временных автоматов, будут решены, если семантику формулы $\phi_1 \mathbf{U}_J \phi_2$ на вычислении $\pi = q_0 q_1 q_2 q_3 \ \ldots$ определить так: существует такое i и такое разбиение интервала временной задержки d_i между парой последовательных состояний q_i и q_{i+1}, при котором формула ϕ_2 выполняется в интервале J, а на всех возможных предыдущих состояниях вычисления π выполняется формула $\phi_1 \vee \phi_2$.

Определим семантику формул логики TCTL формально.

Формальная семантика TCTL

Все формулы логики TCTL являются формулами состояний. Формальная семантика этих формул определяется для временного автомата, состояния которого имеют форму $\langle loc; v \rangle$, где loc — локация, а v — интерпретация значений локальных часов, причем для любого состояния $s = \langle loc; v \rangle$ состояние $\langle loc; v + d \rangle$ обозначается $s + d$. Обозначим также $Paths_{div}(s)$ множество всех дивергентных вычислений. Функция $\Delta(\sigma, i)$ обозначает $d_0 + d_1 + \ \ldots \ + d_i$ для дивергентного вычисления

$$\sigma = q_0 - d_0 \to q_1 - d_1 \to q_2 - d_2 \to q_3 - d_3 \to \ldots .$$

❖ *Определение 12.8 (формальная семантика TCTL)*

Пусть $A=\left(Loc, loc_0, \Sigma, X, Inv, E\right)$ — временной автомат, AP — конечное множество атомарных предикатов и L — функция пометок: $L: Loc \to 2^{AP}$ (будем считать, что имена локаций включены в множество атомарных предикатов). Пусть $Paths_{div}\left(s\right)$ — множество всех дивергентных путей автомата A, начинающихся в состоянии s.

Формулы состояний ϕ логики TCTL определяются следующей грамматикой:

$\phi ::= p \mid \gamma \mid \neg\phi \mid \phi \vee \phi \mid \mathbf{E}\varphi \mid \mathbf{A}\varphi$,
$\varphi ::= \phi \mathbf{U}_J \phi$

где p — атомарный предикат, γ — ограничения на локальные часы, φ — формула пути, а $J \subseteq \mathbf{N}$ — непрерывный временной интервал.

Для любого состояния $q=\langle loc, v\rangle$ автомата A выполнение формул состояния логики TCTL определяется так:

$q \models p$ ***iff*** $p \in L\left(loc\right)$

$q \models \gamma$ ***iff*** $v \models \gamma$

$q \models \neg\phi$ ***iff*** $q \not\models \phi$

$q \models \phi_1 \vee \phi_2$ ***iff*** $q \models \phi_1$ или $q \models \phi_2$

$q \models \mathbf{E}\varphi$ ***iff*** $\sigma \models \varphi$ для некоторого пути $\sigma \in Paths_{div}\left(q\right)$

$q \models \mathbf{A}\varphi$ ***iff*** $\sigma \models \varphi$ для всех путей $\sigma \in Paths_{div}\left(q\right)$

На любом дивергентном вычислении σ:

$\sigma = q_0 - d_0 \to q_1 - d_1 \to q_2 - d_2 \to q_3 - d_3 \to \ldots$

выполнимость темпоральной формулы $\phi_1 \mathbf{U}_J \phi_2$ определяется так:

$$\sigma \models \phi_1 \mathbf{U}_J \phi_2 \quad \textbf{\textit{iff}} \quad \left(\exists i \geq 0\right)\left(\exists d \in \left[0, d_i\right]: \left(\Delta\left(\sigma, i\right)+d \in J\right)\right)\ \left[q_i + d \models \phi_2\ \wedge\right.$$
$$\left.\left(\forall j \leq i\right)\left(\forall d' \in \left[0, d_j\right]: \left(\Delta\left(\sigma, j\right)+d' \leq \Delta\left(\sigma, i\right)+d\right)\right)\left[q_j + d' \models \phi_1 \vee \phi_2\ \right]\right]$$

Как обычно, формула состояний ϕ логики TCTL выполняется для временного автомата A, если эта формула выполняется в начальном состоянии A.

Семантика выводимых темпоральных операторов $\mathbf{F}_J\phi$ и $\mathbf{G}_J\phi$ определяется на основе семантики оператора Until.

Из соотношения $\mathbf{F}_J\phi = true\mathbf{U}_J\phi$ следует:

$$\sigma \models \mathbf{F}_J\phi \quad \textit{iff} \quad (\exists i \geq 0)\left(\exists d \in [0, d_i] : \left(\Delta(\sigma, i) + d \in J\right)\right) q_i + d \models \phi$$

Из того, что $\mathbf{G}_J\phi = \neg\mathbf{F}_J\neg\phi$, следует:

$$\sigma \models \mathbf{G}_J\phi \quad \textit{iff} \quad (\forall i \geq 0)\left(\forall d \in [0, d_i] : \left(\Delta(\sigma, i) + d \in J\right)\right) q_i + d \models \phi$$

Пример 12.13

Проверим, выполняется ли формула $m\mathbf{U}_{(1,2)}n$ на вычислении π временного автомата A_7 (см. рис. 12.17), если $\pi = q_0 - 0.7 \to q_1 - 0 \to q_2 - 1.4 \to q_3 \to \ldots$, где:

$$q_0 = \langle m; x = 0\rangle; q_1 = \langle m; x = 0.7\rangle; q_2 = \langle n; x = 0\rangle; q_3 = \langle n; x = 1.4\rangle.$$

В вычислении π существует натуральное $i = 2$ и состояние $q_2 = \langle n; x = 0\rangle$, такие, что $q_2 + d \models n$ для $d = 0.5$, причем $d \in [0, d_2] = [0, 1.4]$ и $\Delta(\pi, 2) + d = 0.7 + d \in J = (1, 2)$.

Для всех состояний q вычисления π, возможных до состояния $q_2 + d$, выполняется $q \models m \vee n$. Действительно:

- для всех состояний $q_0 + d'$, где $q_0 = \langle m; x = 0\rangle$ и $d' \in [0, 0.7]$, выполняется $q_0 + d' \models m$;
- для всех состояний $q_1 + d'$, где $q_1 = \langle m; x = 0.7\rangle$ и $d' \in [0, 0]$, выполняется $q_1 + d' \models m \vee n$;
- для всех состояний $q_2 + d'$, где $q_2 = \langle n; x = 0\rangle$ и $d' \in [0, d] = [0, 0.5]$, выполняется $q_2 + d' \models m \vee n$.

12.11. Анализ временных автоматов: сведение проверки TCTL-формул к проверке CTL-формул

Выполнимость формулы TCTL для временного автомата

Множество Sat_Φ состояний временного автомата A, в которых выполняется формула Φ логики TCTL, определяется следующим образом:

$$Sat_\Phi = \left\{q \in Loc \times \left(X \to \mathbf{R}^{\geq 0}\right) \mid q \models \Phi\right\}$$

Временной автомат A является моделью формулы состояний Φ логики TCTL тогда и только тогда, когда формула Φ выполняется в его начальном состоянии $\langle loc_0; v_0 \rangle$:

$$A \models \Phi \quad \textit{iff} \quad \langle loc_0; v_0 \rangle \models \Phi .$$

Введение формульных часов

На рис. 12.17, *б* видно, что для проверки выполнимости формулы $\Phi = \phi_1 \mathbf{U}_J \phi_2$ на вычислении $\sigma = s_0 - d_0 \to s_1 - d_1 \to s_2 \to \ldots$ временного автомата A_7, начинающегося с состояния s_0, необходимо вычисление σ представить в реальном времени t, которое протекло с момента прихода временного автомата в состояние s_0. Но это время могут отсчитывать дополнительные локальные часы, которые должны быть сброшены в 0 в начальном состоянии вычисления σ.

Для проверки выполнения формулы Φ на вычислении σ введем во временной автомат A новые локальные часы z с единственной целью — измерение временного интервала, прошедшего с начала вычисления. Эти новые часы называются *формульными* часами. Ограничение времени J, стоящее у темпорального оператора $\mathbf{U}$, может быть теперь представлено как ограничение $z \in J$, наложенное на эти локальные часы. Формулу пути $\phi_1 \mathbf{U}_J \phi_2$ логики TCTL можно представить как формулу $(\phi_1 \vee \phi_2)\mathbf{U}(\phi_2 \wedge z \in J)$ логики CTL при условии, что часы z сброшены в 0 в начальном состоянии этого пути.

Выполнимость формулы состояний $\Phi = \mathbf{E}(\phi_1 \mathbf{U}_J \phi_2)$ логики TCTL в состоянии s временного автомата при наличии "формульных" часов z определяется так:

$$s \models \mathbf{E}(\phi_1 \mathbf{U}_J \phi_2) \quad \textit{iff} \quad s\{z = 0\} \models \mathbf{E}\big((\phi_1 \vee \phi_2)\mathbf{U}(\phi_2 \wedge z \in J)\big)$$

— в состоянии s выполняется формула $\Phi = \mathbf{E}(\phi_1 \mathbf{U}_J \phi_2)$ логики TCTL тогда и только тогда, когда при сброшенных в этом состоянии формульных часах z выполняется формула $\Phi_{CTL} = E\big((\varphi_1 \vee \phi_2)\mathbf{U}(\phi_2 \wedge z \in J)\big)$ логики CTL.

Аналогично определяется выполнимость формулы $\mathbf{A}(\phi_1 \mathbf{U}_J \phi_2)$.

Для выводимых формул $\mathbf{F}$ и $\mathbf{G}$, при условии, что часы z сбрасываются в начальном состоянии того вычисления, на котором проверяется формула, имеем соотношения, позволяющие любую формулу логики TCTL свести

к формуле логики CTL при введенных специально для этой формулы дополнительных "формульных" часах:

$$\mathbf{F}_J\phi = true\mathbf{U}_J\phi = \mathbf{F}\left(\phi \wedge \left(z \in J\right)\right)$$

$$\mathbf{G}_J\phi = \neg\mathbf{F}_J\neg\phi = \neg\mathbf{F}\left(\neg\phi \wedge \left(z \in J\right)\right) = \neg\mathbf{F}\neg\left(\left(z \in J\right) \Rightarrow \phi\right) = \mathbf{G}\left(\left(z \in J\right) \Rightarrow \phi\right)$$

Ограничение $z \in J$ можно рассматривать как атомарный предикат графа регионов, построенного по автомату A с введенными дополнительными формульными часами z, сбрасываемыми в том состоянии, в котором проверяется выполнение соответствующих формул состояний. Такой временной автомат назовем $A \oplus z$.

ЛЕММА 12.2

Пусть два состояния s и q системы переходов $\mathsf{T}\left(A \oplus z\right)$ находятся в отношении $\cong$, т. е. $\left[s\right] = \left[q\right]$. Тогда для любой формулы Φ логики TCTL справедливо:

$$\mathsf{T}\left(A \oplus z\right), s \models \Phi \quad \textit{iff} \quad \mathsf{T}\left(A \oplus z\right), q \models \Phi_{CTL}.$$

Справедливость этой леммы очевидна: "формульные" часы z ничем не отличаются от других локальных часов временного автомата A.

Таким образом, проверка выполнимости формулы Φ логики TCTL для временного автомата A может быть сведена к проверке выполнимости преобразованной формулы Φ_{CTL} логики CTL на графе регионов $\mathsf{R}\left(A \oplus z\right)$ автомата $A \oplus z$ с добавленными "формульными" часами z.

При условии, что формулы ϕ_1 и ϕ_2 не содержат подформул, временные интервалы которых отличны от $\left[0, \infty\right)$, для проверки выполнимости формул $\mathbf{E}\left(\phi_1\mathbf{U}_J\phi_2\right)$, $\mathbf{A}\left(\phi_1\mathbf{U}_J\phi_2\right)$ (и выводимых из них формул $\mathbf{EF}_J\phi$, $\mathbf{AF}_J\phi$, $\mathbf{EG}_J\phi$, $\mathbf{AG}_J\phi$) во временном автомате A необходимо выполнить следующие шаги:

1. Для формулы Φ логики TCTL, содержащей временной интервал J, в автомат A добавляются формульные часы z. Потолок C_z часов z определяется максимальной константой в интервале J. Новый временной автомат назовем $A \oplus z$. Условие $z \in J$ является дополнительным ограничением в множестве Γ ограничений на локальные часы автомата $A \oplus z$.
2. Формула Φ логики TCTL, содержащая временной интервал $J \neq \left[0, \infty\right)$, заменяется на формулу Φ_{CTL} логики CTL по правилам:

$\mathbf{E}(\phi_1 \mathbf{U}_J \phi_2)$ заменяется на $\mathbf{E}\left[(\phi_1 \vee \phi_2)\mathbf{U}(\phi_2 \wedge z \in J)\right]$

$\mathbf{EF}_J \phi$ заменяется на $\mathbf{EF}(\phi \wedge (z \in J))$

$\mathbf{EG}_J \phi$ заменяется на $\mathbf{EG}((z \in J) \Rightarrow \phi)$

Формулы с квантором всеобщности преобразуются аналогично.

3. Обычный алгоритм проверки модели для формул логики CTL применяется к графу регионов $\mathsf{R}(A \oplus z, s_0\{z=0\})$. Состояние s_0 является начальным состоянием графа регионов $\mathsf{R}(A)$, в нем проверяется выполнением формулы Φ для временного автомата A.
4. Во временном автомате A формула Φ логики TCTL выполняется тогда и только тогда, когда формула Φ_{CTL} выполняется в состоянии $s_0\{z := 0\}$ графа регионов $\mathsf{R}(A \oplus z, s_0\{z=0\})$:

$A \models \Phi$ ***iff*** $s_0\{z := 0\} \models \Phi_{CTL}$

Пример 12.14

Проверим, выполняется ли формула $\Phi = \mathbf{AG}_{\le 1} n$ логики TCTL для временного автомата A_7, представленного на рис. 12.17, *а*. Формула Φ будет выполняться для временного автомата, если она будет выполняться в его начальном состоянии. Проверку будем выполнять по шагам:

1. В автомат A_7 вводим дополнительные к x формульные часы z, задача которых — измерять отрезок времени, прошедший после старта автомата из начального состояния. Потолок C_z этих часов равен 1 — это максимальная константа в ограничении J формулы Φ.
2. Формулу $\Phi = \mathbf{AG}_{\le 1} n$ логики TCTL заменяем формулой Φ_{CTL}:

$$\Phi_{CTL} = \mathbf{AG}((z \le 1) \Rightarrow n) = \neg\mathbf{EF}\neg((z \le 1) \Rightarrow n) = \neg\mathbf{EF}((z \le 1) \wedge \neg n).$$

Ограничения на формульные часы представлены здесь атомарным предикатом $(z \le 1)$.

3. Строим граф регионов $\mathsf{R} = \mathsf{R}(A_7 \oplus z, s_0\{z := 0\})$ (рис. 12.18). Этот граф строится для двух часов, x и z, с потолками $C_x = 1$ и $C_z = 1$. Поскольку формула Φ_{CTL} проверяется для начального состояния A_7, "формульные" часы z установлены в 0 в начальном состоянии графа регионов R. "Фор-

мульные" часы z в том графе не сбрасываются, они просто отсчитывают время, прошедшее в вычислениях после того, как были установлены в 0 в состоянии s_0.

4. Проверку выполнимости CTL-формулы $\Phi_{CTL} = \neg\mathbf{EF}\big((z \le 1) \wedge \neg n\big)$ для графа регионов $\mathsf{R}\big(A_7 \oplus z, s_0\{z := 0\}\big)$ проведем обычным образом: последовательно находим множества его состояний, удовлетворяющих подформулам формулы Φ_{CTL} (см. рис. 12.18):

$f_1 = (z \le 1);$ $\quad Sat(f_1) = \{s_0,\ s_1,\ s_2,\ s_3,\ s_8,\ s_9,\ s_{10}\}$

$f_2 = \neg n;$ $\quad Sat(f_2) = \{s_0,\ s_1,\ s_2,\ s_{11}, s_{12}, s_{13}\}$

$f_3 = f_1 \wedge f_2;$ $\quad Sat(f_3) = \{s_0,\ s_1,\ s_2\}$

$f_4 = \mathbf{EF} f_3;$ $\quad Sat(f_4) = \{s_0,\ s_1,\ s_2\}$

$\Phi_{CTL} = \neg f_4;$ $\quad Sat(\Phi_{CTL}) = \{s_3 - s_{13}\}$

Формула $\Phi_{CTL} = \neg\mathbf{EF}\big((z \le 1) \wedge \neg n\big)$ не выполняется в начальном состоянии графа регионов $\mathsf{R}\big(A_7 \oplus z, s_0\{z := 0\}\big)$, и, следовательно, формула $\Phi = \mathbf{AG}_{\le 1} n$ не выполняется во временном автомате A_7.

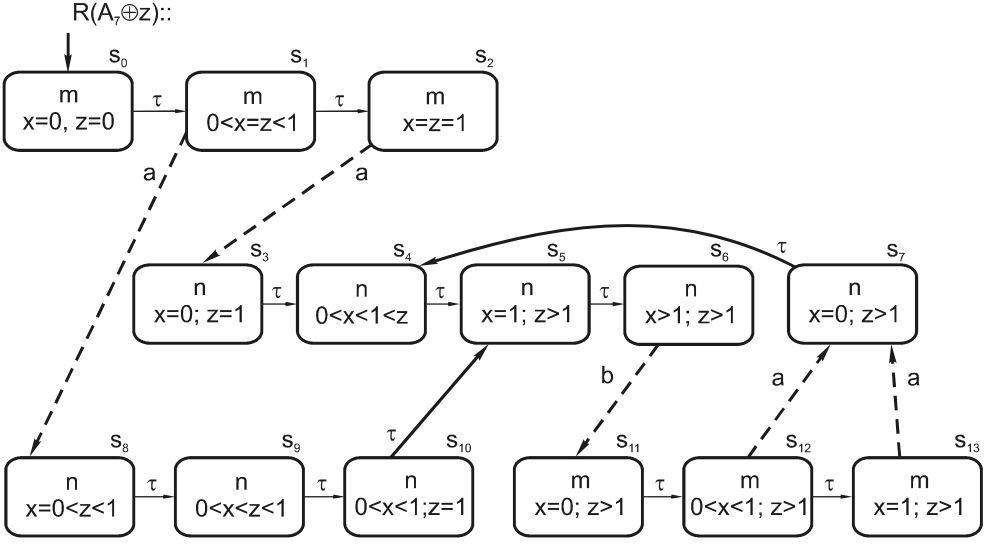


Рис. 12.18. Граф регионов $\mathsf{R}\big(A_7 \oplus z\big)$ временного автомата A_7 с добавленными формульными часами, сброшенными в начальном состоянии

Вложенные формулы TCTL

Пусть формула Φ логики TCTL содержит вложенные подформулы, включающие темпоральные операторы **U**, **F** или **G** с временными ограничениями J, отличными от $[0,\infty)$. Классический алгоритм, используемый в процедуре model checking для пометки состояний структуры Крипке подформулами, которые истинны в этом состоянии, могут быть использованы в этом случае так. При проверке выполнения формулы Φ в состояниях временного автомата A можно использовать граф регионов A, поскольку любая формула логики TCTL будет иметь одно и то же истинностное значение для всех состояний автомата A, принадлежащих одному и тому же состоянию графа регионов $\mathsf{R}(A)$ автомата A. Поэтому граф регионов временного автомата A в алгоритме model checking будет играть роль его структуры Крипке.

Будем считать, что формула Φ логики TCTL, проверяемая на выполнение в состояниях графа регионов $\mathsf{R}(A)$, не содержит подформул, которые имеют временные ограничения J. Для того чтобы проверить выполнение формулы Φ не только в начальном, но также и в любом другом состоянии графа регионов, необходимо для каждого состояния s графа регионов A строить граф регионов $\mathsf{R}(A \oplus z, s\{z := 0\})$, рассматривая состояние s графа регионов A с добавленным значением $z = 0$ как начальное состояние этого графа.

Пример 12.15

Проверим для временного автомата A_7 выполнение формулы $\mathbf{EF}_{<1}\mathbf{AG}_{\le 1} n$. Построим граф регионов автомата A_7 (рис. 12.19, *а*). Проверим выполнение подформулы $f_5 = \mathbf{AG}_{\le 1} n = \neg\mathbf{EF}\big((z \le 1) \wedge \neg n\big)$ во всех состояниях графа регионов $\mathsf{R}(A_7)$.

В состоянии $q_0 = \langle m; (x = 0)\rangle$ эта формула проверена в предыдущем примере, она не выполняется. Проверим ее выполнение в состоянии $q_3 = \langle n; (x = 0)\rangle$. Для этого построим граф регионов $\mathsf{R}(A_7 \oplus z, q_3\{z := 0\})$ (рис. 12.19, *б*), состояния которого отличаются от состояний графа регионов $\mathsf{R}(A_7)$ добавлением значений "формульных" часов z. Для подформул формулы $\neg\mathbf{EF}\big((z \le 1) \wedge \neg n\big)$ построим следующие множества состояний, в которых эти подформулы выполняются:

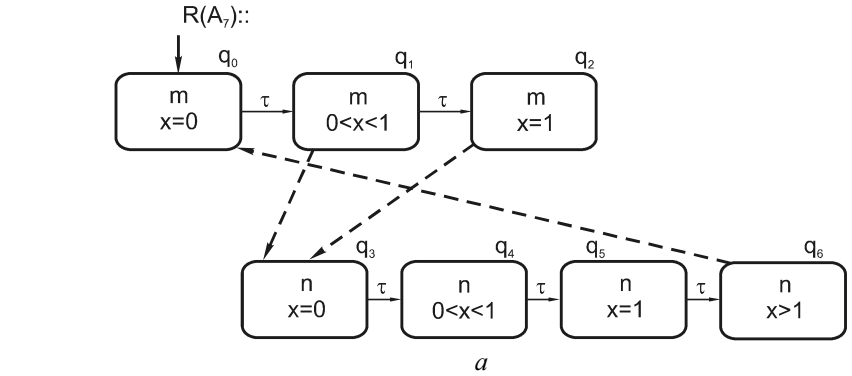
$f_1 = (z \le 1)$; $\quad Sat(f_1) = \{s_0,\ s_1,\ s_2\}$

$f_2 = \neg n$; $\quad Sat(f_2) = \{s_4,\ s_5,\ s_6\}$

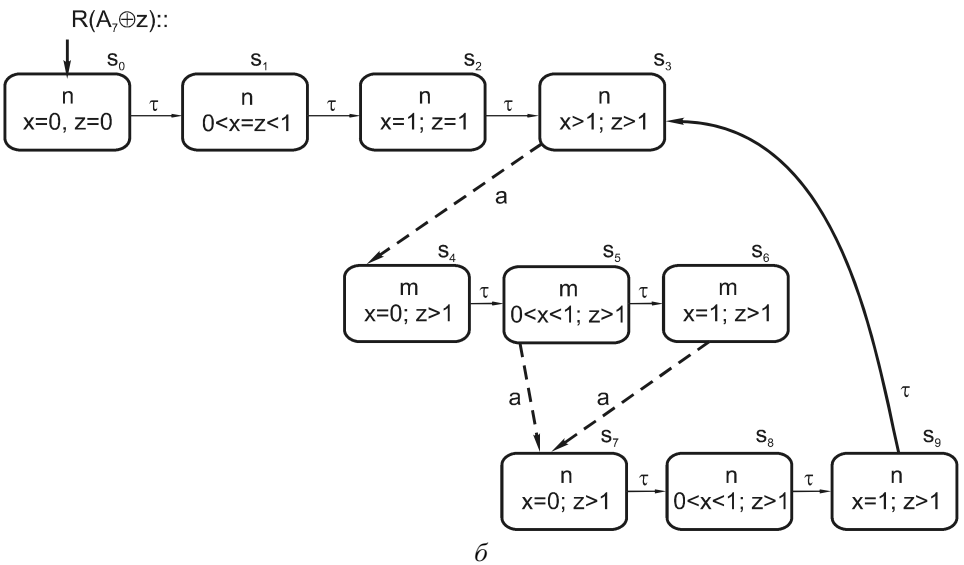
$f_3 = f_1 \wedge f_2; \quad Sat(f_3) = \{\ \}$

$f_4 = \mathbf{EF} f_3; \quad Sat(f_4) = \{\ \}$

$f_5 = \neg f_4; \quad Sat(\Phi_{CTL}) = \{s_0 - s_{13}\}$



а



б

Рис. 12.19. Граф регионов $\mathsf{R}(A_7)$ и граф регионов $\mathsf{R}(A_7 \oplus z, q_0\{z := 0\})$

Формула $f_5 = \mathbf{AG}_{\le 1} n$ выполняется в начальном состоянии графа регионов $\mathsf{R}(A_7 \oplus z, q_3\{x = 0\})$, построенного из начального состояния q_3 графа регионов $\mathsf{R}(A_7)$, и, следовательно, формула $\Phi = \mathbf{AG}_{\le 1} n$ выполняется в состоя-

нии q_3 графа регионов $\mathsf{R}(A_7)$ временного автомата A_7. На рис. 12.20 показан начальный фрагмент графа регионов, из которого видно, что формула $\mathbf{AG}_{\le 1} n$ не выполняется в состоянии q_6. Подобные построения показывают, что эта формула выполняется только в состоянии q_3 графа регионов временного автомата A_7.

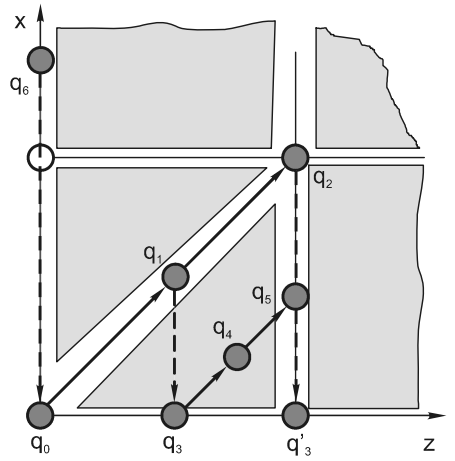


Рис. 12.20. Последовательность регионов при проверке выполнения формулы $\mathbf{AG}_{\le 1} n$ в графе регионов $\mathsf{R}\left(A_7 \oplus z, q_6\{z := 0\}\right)$

Для проверки выполнения формулы $\mathbf{EF}_{<1}\mathbf{AG}_{\le 1} n$ в автомате A_7 преобразуем ее так:

$$\mathbf{EF}_{<1}\mathbf{AG}_{\le 1} n = \mathbf{EF}\left((z<1) \wedge \mathbf{AG}_{\le 1} n\right) = \mathbf{EF}_{<1} f_5 = \mathbf{EF}\left((z<1) \wedge f_5\right).$$

Поскольку формула $f_5 = \mathbf{AG}_{\le 1} n$ выполняется в состоянии $q_3 = \langle n; (x=0)\rangle$ графа регионов $\mathsf{R}(A_7)$, то она выполняется и в состояниях s_3, s_7, s_8 графа регионов $\mathsf{R}\left(A_7 \oplus z, q_0\{z := 0\}\right)$ (рис. 12.18), отличающихся от q_3 только значением вспомогательных "формульных" часов. По графу регионов $\mathsf{R}\left(A_7 \oplus z, q_0\{z := 0\}\right)$ последовательно имеем:

$$f_5 = \mathbf{AG}_{\le 1} n; \qquad Sat(f_5) = \{s_3,\ s_7,\ s_8\};$$

$$f_6 = (z<1); \qquad Sat(f_6) = \{s_0,\ s_1,\ s_8, s_9\};$$

$f_7 = f_5 \wedge f_6;$ $\quad Sat(f_7) = \{s_8\};$

$f_8 = \mathbf{EF} f_7;$ $\quad Sat(f_8) = \{s_0, s_1, s_8\}.$

Поскольку формула $\mathbf{EF}_{<1}\mathbf{AG}_{\le 1} n$ выполняется в начальном состоянии графа регионов $\mathsf{R}(A_7 \oplus z, q_0\{z := 0\})$, построенного из начального состояния q_0 графа регионов $\mathsf{R}(A_7)$, то эта формула выполняется для автомата A_7.

Приведем алгоритм model checking проверки выполнимости формулы Φ логики TCTL для графа регионов $\mathsf{R}(A)$ временного автомата A с множеством состояний S, множеством атомарных предикатов AP, множеством локальных часов X и множеством ограничений на часы Γ. Он состоит в разметке состояний $\mathsf{R}(A)$ теми подформулами формулы Φ, которые выполняются в этих состояниях:

```
for all 0< i ≤ |Φ| do
  for all Ψ∈ Sub(Φ) with |Ψ| = i do
/* Все подформулы Ψ выделяются алгоритмом синтаксического анализа формулы
Φ */
    switch(Ψ):
       true     :     Satψ := S;
       p        :     Satψ := { s∈S| s= ⟨ loc;[v]⟩, p∈L(loc) };
       γ        :     Satψ := { s∈S| s= ⟨ loc;[v]⟩, [v]|= γ };
       ¬p       :     Satψ := { s∈S| s= ⟨ loc;[v]⟩, p∉L(l) };
       pvq      :     Satψ := { s∈S| p∈L(s) ∨ q∈L(s) };
       E(pUJq):       Satψ := SatCTL(EU(pvq,(z∈J) ∧ q ) );
       A(pUJq):       Satψ := SatCTL(AU(pvq,(z∈J) ∧ q ) );
    end switch
/* Вводим в AP новый атомарный предикат pψ и состояния графа регионов
из Satψ помечаем им */
  for all s∈ Satψ do L(s):=L(s)∪{pψ} od
  od
od
```

Отметим еще раз: алгоритм model checking может быть использован только в том случае, если во временном автомате нет вычислений Зенона. Эти вычисления включают бесконечное число действий в конечном интервале времени. Поскольку действий у временного автомата конечное число, то вычисление Зенона в графе регионов автомата включает цикл, приводящий к той же самой локации без приращения значения локальных часов на 1 или более. Существует простой способ выяснить наличие таких вычислений: для того

чтобы во временном автомате отсутствовали вычисления Зенона, достаточно, чтобы для всех состояний автомата выполнялась формула TCTL: $\mathbf{AF}_{\geq 1}\, true$.

Логика $\mathrm{TCTL_c}$

Существует вариант временной темпоральной логики ветвящегося времени, в которой, в отличие от TCTL, ограничения реального времени задаются явным введением "формульных" часов, "замороженных" в рамках формулы этой логики. Грамматика формул логики $\mathrm{TCTL_c}$ определяет формулы состояний ϕ, которые строятся по следующим правилам:

$$\phi ::= p \mid \gamma \mid \neg\phi \mid \phi \vee \phi \mid z\ \mathbf{in}\ \phi \mid \mathbf{E}\left[\phi \mathbf{U} \phi\right] \mid \mathbf{A}\left[\phi \mathbf{U} \phi\right],$$

где p — атомарный предикат, а γ — условия, наложенные на значения локальных часов. Оператор "z **in** " называется оператором "*замораживания*", он вводит новые "формульные" часы z и ограничивает область их действия формулой ϕ, к которой он применяется. Формула $z\ \mathbf{in}\ \phi$ истинна в состоянии s временного автомата A, если в состоянии $s\{z := 0\}$ истинна формула ϕ в автомате $A \oplus z$. Например, очевидно, что формула $z\ \mathbf{in}\ (z = 0)$ всегда истинна, а формула $z\ \mathbf{in}\ (z = 3)$ всегда ложна.

В отличие от логики TCTL, в которой явно указываются временные интервалы в темпоральном операторе *Until* , конструкция $z\ \mathbf{in}\ \phi$ логики $\mathrm{TCTL_c}$, фактически, отвязывает введенные "формульные" часы от этих темпоральных операторов, позволяя использовать условия, включающие единственные "формульные" часы в качестве атомарных предикатов любых формул ϕ, указанных в качестве аргумента оператора "замораживания". Например, формула:

$$\Phi_1 = z\ \mathbf{in}\left[\mathbf{EF}\left(\phi_1 \wedge (z < 10) \wedge \mathbf{EG}\left(z < 20 \Rightarrow \neg\phi_2\right)\right)\right]$$

выражает следующее свойство: из текущего состояния s за время t, меньшее 10 единиц времени, можно достичь такого состояния s', в котором выполняется формула ϕ_1 и из которого существует путь, на котором в течение по меньшей мере 20 единиц времени не будет выполняться свойство ϕ_2 . Ясно, что формула Φ_1 отлична от следующей формулы TCTL:

$$\Phi_2 = \mathbf{EF}_{<10}\left(\phi_1 \wedge \mathbf{EG}_{<20} \neg\phi_2\right),$$

в которой временные ограничения связаны с различными "формульными" часами.

Оказывается, формула Φ_1 не может быть выражена в логике TCTL. Следовательно, логика TCTL менее выразительна, чем логика $TCTL_c$. Однако алгоритм model checking для формул $TCTL_c$ не отличается от подобного алгоритма для формул TCTL.

12.12. Сложность алгоритмов model checking для логики TCTL

Алгоритмы проверки свойств, выраженных формулами TCTL, весьма сложны. Например, известно, что ниже сформулированная проблема является NP-трудной, но она формализуется как задача проверки свойства достижимости для простого временного автомата.

Пример 12.16

Проблема Subset-Sum состоит в нахождении такого подмножества заданного конечного множества S натуральных чисел, $S = \{m_1, m_2, \ldots m_k\}$, сумма элементов которого равна заданному числу M (эта проблема рассматривалась в примере 9.7). Для ее решения построим временной автомат с $k+1$ состоянием (рис. 12.21), переходы $(s_{i\text{-}1}, s_i)$ которого могут выполняться либо при точных значениях локальных часов $x = m_i$, либо мгновенно. Очевидно, что алгоритм проверки выполнимости формулы $\mathbf{EF}_{=M} s_k$ для этого временного автомата решает проблему Subset-Sum.

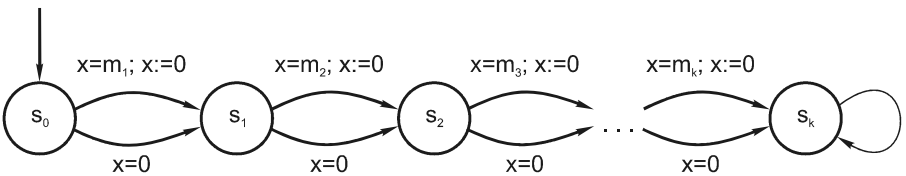


Рис. 12.21. Представление проблемы Subset-Sum временным автоматом

Сложность алгоритма model checking для TCTL-формулы Φ во временном автомате A линейна относительно числа подформул в формуле Φ, а также относительно числа состояний и переходов в графе регионов автомата A. Число состояний графа регионов линейно зависит от числа локаций автомата A и числа регионов. В свою очередь, число регионов временного автомата огромно: оно экспоненциально зависит от числа локальных часов и потолков этих часов.

На практике верификаторы систем реального времени могут проверять весьма ограниченный диапазон свойств. Например, система Uppaal анализирует сети взаимодействующих временных автоматов, но ограничивается проверкой для них свойств достижимости, а также проверкой простых одноуровневых формул логики TCTL.

12.13. Другие проблемы верификации временных автоматов

Рассмотрим, какие проблемы, встающие при верификации, могут быть решены для временных автоматов с использованием модели графа регионов.

Допускаемый язык

Одна из проблем верификации временного автомата $A = (L, loc_0, \Sigma, X, Inv, E)$ — это анализ его языка, т. е. возможных цепочек действий из алфавита Σ, которые может выполнить автомат. Временным действием автомата A была названа пара (t, a), где $a \in \Sigma$ — действие, которое выполняется автоматом после временного интервала $t \in \mathbf{R}^{\geq 0}$, прошедшим с момента старта автомата. Временная траектория — это (возможно, бесконечная) последовательность временных действий $(t_0, a_0)(t_1, a_1)(t_2, a_2)(t_3, a_3)\ldots(t_i, a_i)\ldots$, где $t_i \leq t_{i+1}$ для всех i.

❖ ***Определение 12.9 (траектория временного автомата)***

Траекторией временного автомата A над временной траекторией $\xi = (t_0, a_0)(t_1, a_1)(t_2, a_2)(t_3, a_3)\ldots$ назовем вычисление A:

$$\langle loc_0; v_0 \rangle - d_0 \to \langle loc_0; v'_1 \rangle - a_0 \to \langle loc_1; v_1 \rangle - d_1 \to \langle loc_1; v'_2 \rangle - a_1 \to$$
$$\langle loc_2; v_2 \rangle - d_2 \to \langle loc_2; v'_3 \rangle - a_2 \to \ldots$$

удовлетворяющей условию $t_i = t_{i-1} + d_i$. Значение интервала времени t в паре (t_i, a_i) назовем временным штемпелем действия a_i.

Временный язык временного автомата A (обозначим его $L(A)$) — это множество всех временных траекторий ξ, для которых существует вычисление A над ξ.

Пример 12.17

Для автомата A_1 рис. 12.1 временными траекториями будут:

$$(1,\ a)(1.2,\ a)(1.5,\ b)(4.2,\ a)(5.2,\ c)(16.2,\ a)\ ...,$$
$$(1.34,a)(1.42,a)(1.5,b)(4,a)(4.5,a)...$$

Оказывается, что проблема включения временных языков (следовательно, и проблема эквивалентности таких языков) для временных автоматов в общем случае алгоритмически неразрешима.

Если абстрагироваться от временных штемпелей в цепочках языка, то мы получим цепочки так называемого "невременно́го" языка. Невременно́й язык автомата A, обозначаемый $L_{untimed}(A)$, — это множество возможных цепочек $a_1a_2a_3\ ...$ действий A, для которых в A существует какая-либо временная траектория $\xi = (t_1,a_1)(t_2,a_2)(t_3,a_3)....$

ЛЕММА 12.3 (включение языков)

Проблема проверки включения $\left(L_{untimed}(A) \subseteq L_{untimed}(B)\right)$ невременных языков для двух временных автоматов A и B разрешима. Она может быть решена с помощью графов регионов этих автоматов.

Пример 12.18

По графу регионов автомата A_5 (см. рис. 12.13) видно, что его невременными траекториями будут: *ababcccc*, *acccc* и др. По этому графу можно заключить, что в автомате A_5 не могут выполниться цепочки действий *abcab*, *accba* и подобные им. Под воздействием входной цепочки *accb* временной автомат A_5 не может вернуться в свое начальное состояние, как это можно было бы предположить по его виду. Граф регионов $\mathsf{R}(A_5)$ показывает, что в этом автомате такая последовательность действий вообще невозможна.

Бисимуляция

Один из подходов к проверке свойств временных автоматов заключается в том, что для проверяемого автомата A строится контрольный временной автомат B, выражающий требуемое свойство, и проверяется эквивалентность поведений автоматов A и B.

Одной из форм эквивалентности поведений систем переходов является бисимуляция. Бисимуляция основана на идее моделирования одной системы переходов другой системой переходов. Более формально это определение зву-

чит так. Как обычно, значком $\to *$ будем обозначать транзитивное рефлексивное замыкание бинарного отношения $\to$, т. е.

$p \to *q$ обозначает $p = s_0 \to s_1 \to ... \to s_n = q$ при любом конечном $n = 0,1,2,....$

Пусть M и M' — две временных системы переходов с одним и тем же алфавитом действий Σ. Обозначим $= a \Rightarrow$ бинарное отношение на множестве состояний системы переходов так: $p = a \Rightarrow q$, если существует такое состояние s, что:

$p - \tau \to *s$ и $s - a \to q$,

где $a \in \Sigma$, а τ — внутренний переход системы по времени.

> ❖ ***Определение 12.10 (бисимуляция)***
>
> Бисимуляция — это отношение эквивалентности на множестве пар состояний $S \times S'$ систем M и M' и алфавите действий Σ, определяющееся как симметричное бинарное отношение, удовлетворяющее следующим условиям для всех пар (s,s') бисимуляционно эквивалентных состояний:
>
> - если в M существует переход $s = a \Rightarrow q$ для некоторых $a \in \Sigma$ и q, то в M' существует переход $s' = a \Rightarrow q'$ и состояния q, q' бисимуляционно эквивалентны;
>
> и
>
> - если в M' существует переход $s' - a \to q'$ для некоторых $\alpha \in \Sigma \cup R^+$ и q', то в M существует переход $s - a \to q$ и состояния q, q' бисимуляционно эквивалентны.

Две временных системы переходов M и M' бисимуляционно эквивалентны (обозначается $M \approx M'$), если начальные состояния этих систем бисимуляционно эквивалентны.

Таким образом, две системы переходов находятся в отношении "невременной бисимуляции" тогда и только тогда, когда они выполняют одни и те же "видимые" извне переходы, и при этом переходят в эквивалентные состояния.

ТЕОРЕМА 12.2

Система переходов $\mathsf{T}(A)$ и автомат регионов $\mathsf{R}(A)$ любого временного автомата A бисимуляционно эквивалентны при бисимуляции $\cong$.

Иными словами, каждое состояние s системы переходов $\mathsf{T}(A)$ бисимуляционно эквивалентно состоянию $[s]$ графа регионов $\mathsf{R}(A)$, т. е. если две интерпретации часов, v и v', находятся в одном временном регионе, $[v]=[v']$, и в графе переходов $\mathsf{T}(A)$ существует любая цепочка переходов:

$$\langle loc; v_0\rangle - d_1 \to \langle loc_1; v_1\rangle - d_2 \to \langle loc_2; v_2\rangle - d_3 \to \ldots - a \to \langle loc_n; v_n\rangle$$

то в графе регионов $\mathsf{R}(A)$ найдется соответствующая цепочка переходов:

$$\langle loc; [v]\rangle - \tau \to \langle loc_1; [v_1]\rangle - \tau \to \ldots - a \to \langle loc_n; [v_n]\rangle$$

и наоборот.

Как результат, любая цепочка действий допускается во временном графе переходов тогда и только тогда, когда она будет допустима и в графе регионов. Поэтому бисимуляционную эквивалентность двух временных автоматов A и B можно проверить, анализируя бисимуляционную эквивалентность их графов регионов, $\mathsf{R}(A)$ и $\mathsf{R}(B)$.

Достижимость

Проблема достижимости для временных автоматов является одной из важнейших проблем верификации. Неформально, эта проблема звучит так: достижимо ли некоторое (*некорректное*) состояние $\langle loc; v\rangle$ временного автомата из его начального (или некоторого промежуточного) состояния (т. е. справедливо ли $\langle loc; v\rangle \to {}^*\langle loc'; v'\rangle$?

Для состояний $\langle loc; v\rangle$ и $\langle loc'; v'\rangle$ временного автомата $A=(L, loc_0, \Sigma, X, Inv, E)$ будем писать: $\langle loc; v\rangle \to \langle loc'; v'\rangle$, если либо $\langle loc; v\rangle - a \to \langle loc'; v'\rangle$ для некоторого действия $a\in\Sigma$, либо $\langle loc; v\rangle - d \to \langle loc'; v'\rangle$ для некоторого $d\in \mathbf{R}^{\geq 0}$.

> ❖ ***Определение 12.11 (достижимость)***
>
> Пусть дан временной автомат $A=(L, loc_0, \Sigma, X, Inv, E)$ и некоторое его состояние $\langle loc; v\rangle$. Проблема достижимости состоит в проверке того, может ли это состояние $\langle loc; v\rangle$ быть достигнуто из начального состояния $\langle loc_0; v_0\rangle$, т. е. справедливо ли $\langle loc_0; v_0\rangle \to {}^*\langle loc; v\rangle$, (где $v_0(x)=0$ для всех локальных часов x из X).

Справедлива следующая теорема.

ТЕОРЕМА 12.3

Во временном автомате A $\langle loc_0; v_0\rangle \to^* \langle loc; v\rangle$ тогда и только тогда, когда в графе регионов $\mathsf{R}(A)$ этого автомата $\langle loc_0; [v_0]\rangle \to^* \langle loc; [v]\rangle$.

Таким образом, во временном автомате любое состояние достижимо тогда и только тогда, когда соответствующее состояние достижимо в графе регионов этого автомата. Следовательно, проблема достижимости для временных автоматов разрешима.

Пример 12.19

Из начального состояния $\langle s;\ x=0,\ y=0\rangle$ временного автомата A_5 (см. рис. 12.13) достижимо состояние $\langle r;\ x=2,\ y=0.3\rangle$. Это можно установить по графу регионов $\mathsf{R}(A_5)$, в котором достижимо состояние $q_{14} = \langle r; [x=2,\ y=0.3]\rangle = \langle r; ([x=2,\ 0<y<1])\rangle$. Одна из возможных цепочек в A_5:

$$\langle s;\ x=0, y=0\rangle - 1.7 \to \langle s;\ x=1.7,\ y=1.7\rangle - a \to \langle r;\ x=1.7, y=0\rangle - 0.3 \to$$
$$\langle r; x=2,\ y=0.3\rangle$$

Проблему достижимости можно выразить с помощью формулы CTL **EF***p* (или **AG***p*), где p — логическая комбинация атомарных предикатов и ограничений на показания локальных часов. Для ее решения можно строить значительно более грубое представление временного автомата, чем граф регионов.

12.14. Временные зоны

Временные регионы и временные зоны

Разбиение пространства значений локальных часов на регионы, на основе которого строится граф регионов, является фундаментальным приемом для анализа свойств временных автоматов: все доказательства разрешимости и оценки сложности алгоритмов верификации временных автоматов, рассматриваемые в литературе, основываются на конструировании такого разбиения. Однако на практике графы регионов фактически не применяются, поскольку они содержит огромное число состояний, экспоненциально растущее и с ростом числа локальных часов, и с ростом "потолков" локальных часов.

Причина такого огромного числа состояний понятна: идея регионов использует только одно свойство возможных ограничений локальных часов: то, что все константы в ограничениях целые. Разбиение на регионы вообще не учитывает реальных ограничений конкретного временного автомата. Следствием этого является то, что в графе регионов обычно присутствует огромное число состояний, эквивалентных относительно проверяемых свойств.

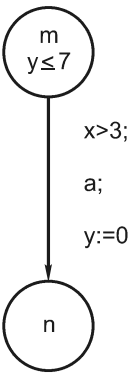
Значительно более эффективное представление поведения временного автомата дают так называемые "временные зоны", представляющие собой объединения таких состояний временного графа, которые эквивалентны относительно не всех возможных свойств поведения (как во временных регионах), а только тех временных свойств, которые выражены в ограничениях и инвариантах временного автомата.

Очевидно, что каждый временной регион и их объединения могут быть представлены как логические формулы над линейными ограничениями локальных часов временного автомата. Недавно была предложена идея анализировать временной автомат, не строя временные регионы, а символически манипулируя всеми ограничениями на показания локальных часов, которые встречаются в этом автомате. Одна из реализаций этой идеи привела к компактному представлению временного автомата с помощью временных зон. Зона — это некоторое подмножество регионов. С точки зрения проверки свойств временных автоматов, принципиальной разницы между графом регионов и графом зон нет, просто граф зон — это эффективное представление графа регионов, в котором одна зона является объединением многих эквивалентных регионов.

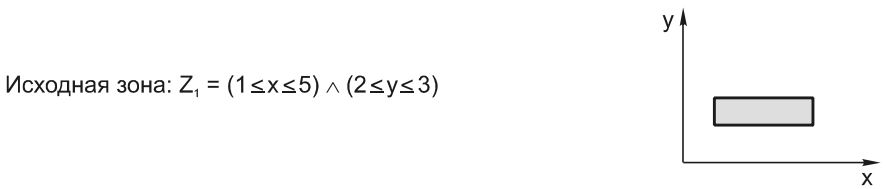
Временная зона задается ограничением на показания часов и является классом эквивалентности показаний часов относительно некоторого ограничения g на показания часов: $g \in \Gamma(X)$, $[g] = \{v \mid v \models g\}$. Иными словами, зона, определяемая ограничением g, это множество значений часов, которые удовлетворяют этому ограничению. Очевидно, что отдельный регион также является зоной, т. е. зона — это просто объединение нескольких регионов. Можно подойти к построению зон, рассматривая объединения эквивалентных регионов, однако зоны могут быть построены и без предварительного построения графа регионов с использованием операций над ограничениями локальных часов, встречающиеся во временном автомате. Точно так же, как граф регионов, граф зон представляет граф переходов временного автомата, но зоны (классы эквивалентности показаний локальных часов автомата), на которые разбивается все множество показаний часов, являются более грубыми, а построенный по этому разбиению граф зон — более компактным.

Пример 12.20

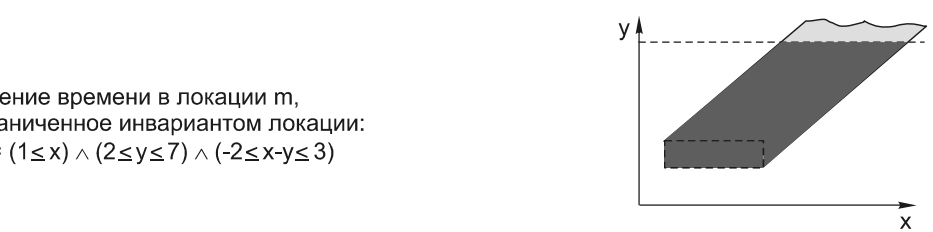
На рис. 12.22 приведен переход временного автомата с двумя локальными часами, x и y, из локации m с инвариантом $y \le 7$ в локацию n. Защита перехода — неравенство $x > 3$. Пусть исходная зона Z_1 (множество значений локальных часов, допустимых при переходе в эту локацию) определяется неравенствами $1 \le x \le 5,\ 2 \le y \le 3$. Если зона Z_1 удовлетворяет инварианту локации m, то какое множество значений локальных часов возможно при переходе в локацию n? Иными словами, если Z_1 — временная зона при входе в локацию m, то какова будет временная зона при входе в локацию n?



а



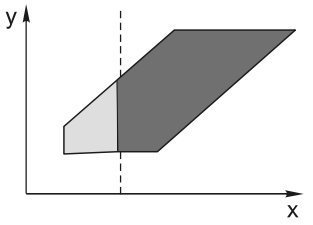
б



в

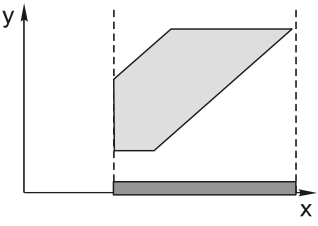
Рис. 12.22, *а—в*. Последовательное преобразование зон на одном переходе временного автомата $\langle m; Z_1 \rangle - a \to \langle n; Z_4 \rangle$

Ограничение (защита) перехода:
$Z_3 = (3<x) \wedge (2 \le y \le 7) \wedge (-2 \le x-y \le 3)$



г

Сброс локальных часов:
$Z_4 = (3<x \le 9) \wedge (y=0)$



д

Рис. 12.22, *г* и *д*. Последовательное преобразование зон на одном переходе временного автомата $\langle m; Z_1 \rangle - a \to \langle n; Z_4 \rangle$

Рис. 12.22 показывает последовательное преобразование зон в этом фрагменте временного автомата для нахождения искомой зоны. Во-первых, начальная зона $Z_1 = (1 \le x \le 5) \wedge (2 \le y \le 3)$ с течением времени изменяется: значения всех локальных часов одновременно растут с одним темпом. Инвариант $y \le 7$ локации m ограничивает рост не только часов y, но и часов x, поскольку между часами обязательно сохраняется соотношение $x - y = Const$. Поэтому все интерпретации часов x и y в локации m до возможного выхода из нее описываются зоной $Z_2 = (1 \le x) \wedge (2 \le y \le 7) \wedge (-2 \le x - y \le 3)$. Далее, все возможные показания пары локальных часов, при которых возможен дискретный переход по действию a, определяет зона $Z_3 = (3 < x) \wedge (2 \le y \le 7) \wedge (-2 \le x - y \le 3)$. Последняя зона $Z_4 = (3 < x \le 9) \wedge \wedge (y = 0)$ является проекцией зоны Z_3 на ось x, поскольку часы y сбрасываются на этом переходе. Таким образом, все возможные вычисления, связанные с этим переходом, представляются следующей цепочкой символьных состояний и переходов:

$$\langle m; Z_1 \rangle - \tau \to \langle m; Z_2 \rangle - a \to \langle n; Z_4 \rangle$$

Очевидно, что зоны дают значительно более компактное представление вычислений, чем регионы.

Операции над зонами

Часто встречающиеся задачи при верификации временных автоматов — это проверка достижимости некоторого состояния и построение графа зон (часто эта операция выполняется "на лету"). Обе задачи сводятся к последовательности построений очередного символьного состояния автомата по его текущему состоянию. Аналогично тому, что символьным состоянием графа регионов временного автомата является пара: $\langle loc; [v] \rangle$, где $[v]$ — часовой регион, состоянием графа зон является пара: $\langle loc; Z \rangle$, которая состоит из локации loc временного автомата и зоны. Так же, как и следующие регионы в графе регионов временного автомата, следующие зоны в его графе зон определяются однозначно. Как видно из примера 12.20, переход к очередной зоне требует выполнения следующих операций над зонами:

- пересечение текущей зоны с инвариантом локации и с защитой перехода;
- приращение времени;
- сброс некоторых локальных часов в зоне.

Рассмотрим эти операции последовательно.

Пересечение зоны с ограничениями на показания часов

Если Z — зона, а g — ограничение на показания часов, то операция пересечения выражается просто как конъюнкция: $Z \wedge g$, поскольку и Z, и g выражены логическими формулами над значениями локальных часов.

Приращение времени

Операция "приращение времени" для текущей зоны Z часто называется операцией построения *сильнейшего постусловия* (Strongest Post Condition). Она обозначается $SP(Z)$ или $Z^{\uparrow}$. Операция SP для зоны Z определяет все множество значений локальных часов при произвольном течении времени, если текущие значения часов принадлежат зоне Z.

$$Z^{\uparrow} = \left\{ v + d \mid d \in \mathbf{R}^{\geq 0}, v \in Z \right\}$$

В алгоритмах обратного поиска (backward reachability) по графу зон используется операция построения *слабейшего предусловия* (Weakest Precondition). Эта операция также дает единственную зону. Она обозначается $WP(Z)$ или $Z^{\downarrow}$. Операция WP для зоны Z определяет то множество значений ло-

кальных часов, которые при произвольном течении времени могут стать значениями, которые принадлежат зоне Z.

$$Z^{\downarrow} = \left\{ v \mid v + d \in Z \text{ для некоторого } d \in \mathbf{R}^{\geq 0} \right\}$$

Сброс локальных часов x определяется просто:

$$\{x := 0\} Z = \{ v[x := 0] \mid v \in Z \}$$

Оказывается, что множество зон замкнуто относительно всех операций над зонами, т. е. в результате любой указанной выше операции над зонами снова получим зону.

Достижимость за один шаг

Построение следующего состояния $succ\langle loc; Z \rangle$ графа зон для текущего состояния $\langle loc; Z \rangle$ определяется однозначно следующими операциями. Пусть Z удовлетворяет инварианту, $Z = Z \wedge Inv(loc)$.

Переход по задержке:

$$\langle loc; Z \rangle - \tau \rightarrow \langle loc; Z' \rangle, \text{ где } Z' = Z^{\uparrow} \wedge Inv(loc)$$

Переход по действию a на переходе $e = (loc, g, a, \{x\}, loc') \in E$, где g — защита перехода:

$$\langle loc; Z \rangle - a \rightarrow \langle loc'; Z' \rangle, \quad Z' = \{x := 0\}(Z \wedge g) \; if \; Z' \neq false \quad \text{и}$$
$$Z' \wedge Inv(loc) = Z'.$$

По построенному графу зон проблема достижимости решается очень просто: символьное состояние $\langle loc'; Z' \rangle$ достижимо во временном автомате, если в графе зон этого автомата найдется такое символьное состояние $\langle loc'; Z'' \rangle$, что $Z' \subseteq Z''$ и $\langle loc_0; Z_0 \rangle \rightarrow^* \langle loc'; Z' \rangle$.

Пример 12.21

На рис. 12.23 представлен временной автомат с тремя состояниями. Все возможные вычисления этого автомата могут быть описаны следующими переходами графа зон:

$$\langle m; Z_0 \rangle - \tau \rightarrow \langle m; Z_1 \rangle - a \rightarrow \langle n; Z_2 \rangle - \tau \rightarrow \langle n; Z_3 \rangle - b \rightarrow \langle r; Z_4 \rangle - \tau \rightarrow$$
$$\langle n; Z_5 \rangle - c \rightarrow \langle m; Z_0 \rangle$$

где:

$$Z_0 = \left(x = 0,\ y = 0\right)$$

$$Z_1 = \left(x - y = 0,\ 0 \le x \le 3\right)$$

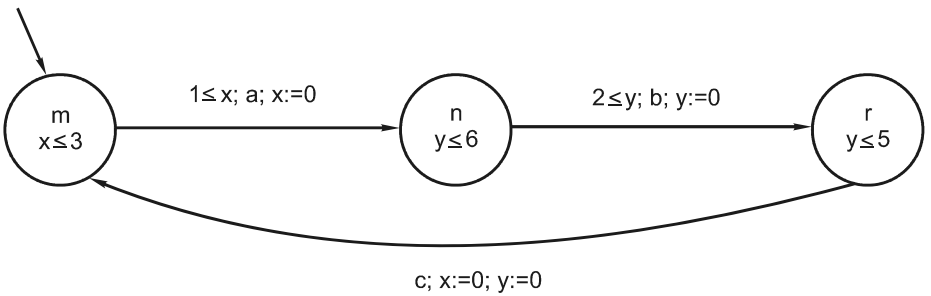
$$Z_2 = \left(x = 0,\ 1 \le y \le 3\right)$$

$$Z_3 = \left(1 \le y \le 6,\ 1 \le y - x \le 3,\ 0 \le x\right)$$

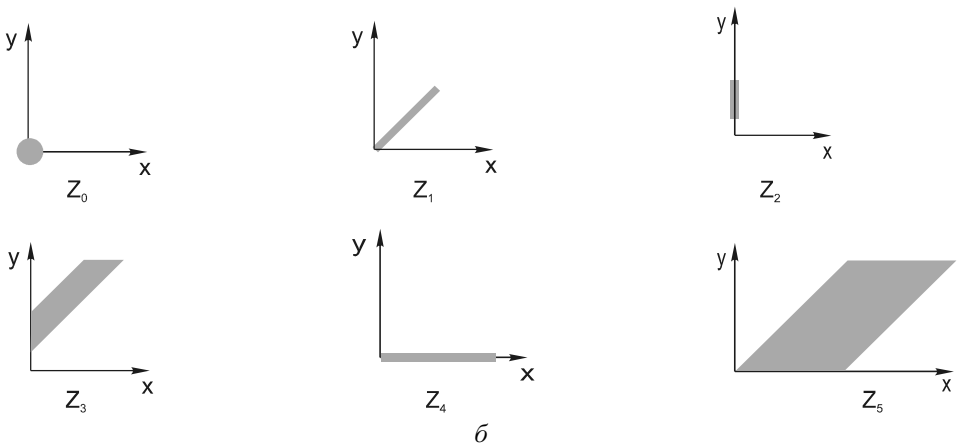
$$Z_4 = \left(y = 0,\ 0 \le x \le 5\right)$$

$$Z_5 = \left(0 \le x - y \le 5,\ 0 \le y \le 5\right)$$

Рис. 12.23 представляет эти зоны графически.



а



б

Рис. 12.23. Последовательные зоны, представляющие возможные значения локальных часов при вычислениях временного автомата

Пример 12.22

Рис. 12.24 представляет временной автомат, различающий одинарный и двойной щелчок мыши. Если второй щелчок мыши приходит до истечения N единиц времени (например, миллисекунд), автомат выдает операционной системе сигнал 'a'. Если между последовательными нажатиями на кнопку мыши прошло более N единиц времени, автомат выдает сигнал 'b'. При $N = 20$ в графе регионов этого автомата более сотни состояний. Граф зон этого автомата (рис. 12.24, *б*) при любом N содержит только 10 состояний

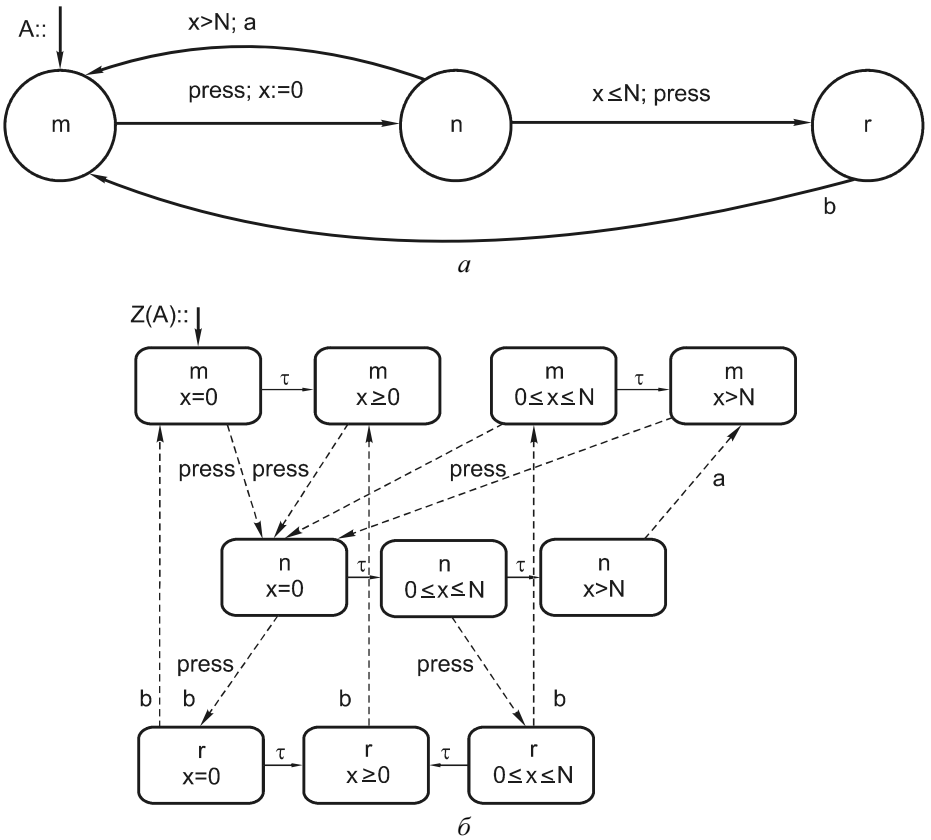


Рис. 12.24. Временной автомат, различающий одинарный и двойной щелчок мыши, и его граф зон

Пример 12.23

Вернемся к анализу варианта протокола Фишера (см. пример 12.17). На рис. 12.15, *в* представлен фрагмент графа регионов, показывающий, что про-

токол Фишера корректен: при любом развитии событий процессы не могут одновременно находиться в своих критических областях. Однако представленные на рис. 12.15, *а* параллельно протекающие процессы слишком примитивны: они используют в ограничениях локальных часов минимально возможное целое значение 1. Если бы в защитах переходов этих процессов стояло бы более реальное число, например 10, граф регионов, по которому можно верифицировать этот протокол, содержал бы много сотен состояний.

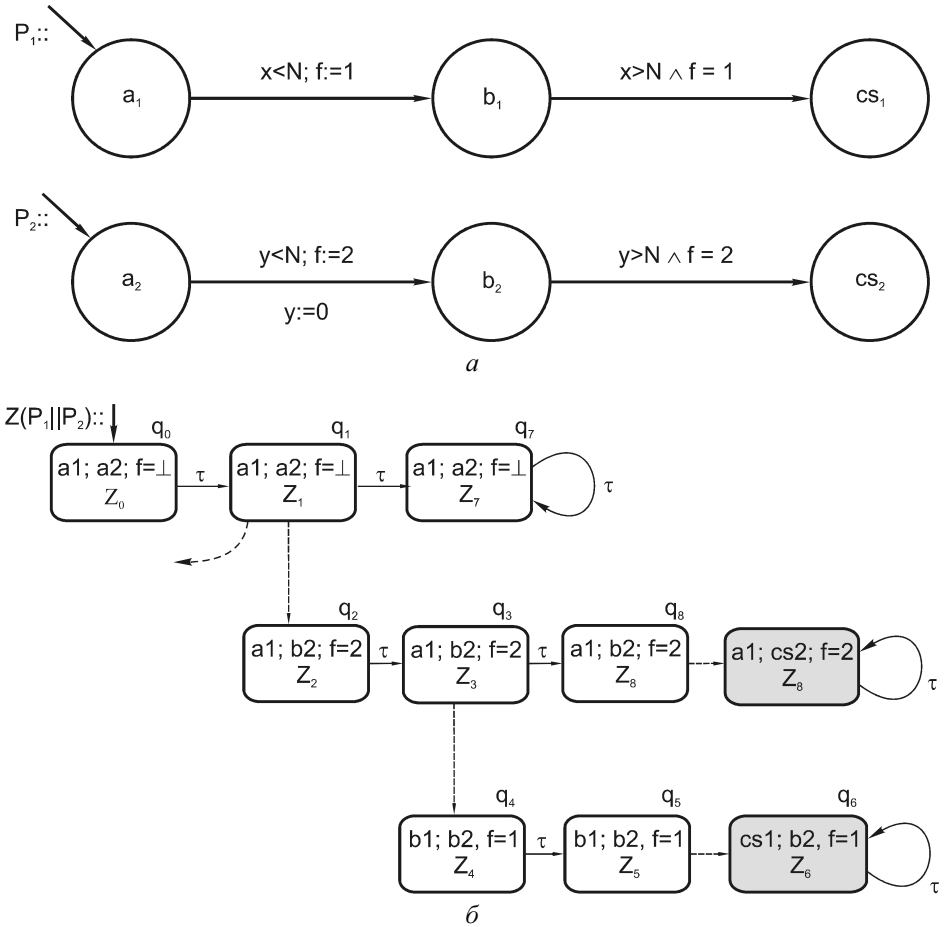


Рис. 12.25. Протокол Фишера и фрагмент его графа зон

С использованием временных зон такой анализ может быть выполнен очень легко. Рис. 12.25, *а* показывает эти параллельные процессы с ограничениями часов в N единиц времени (например, миллисекунд), рис. 12.25, *б* представляет начальный фрагмент графа зон для этого протокола. Рис. 12.26 показы-

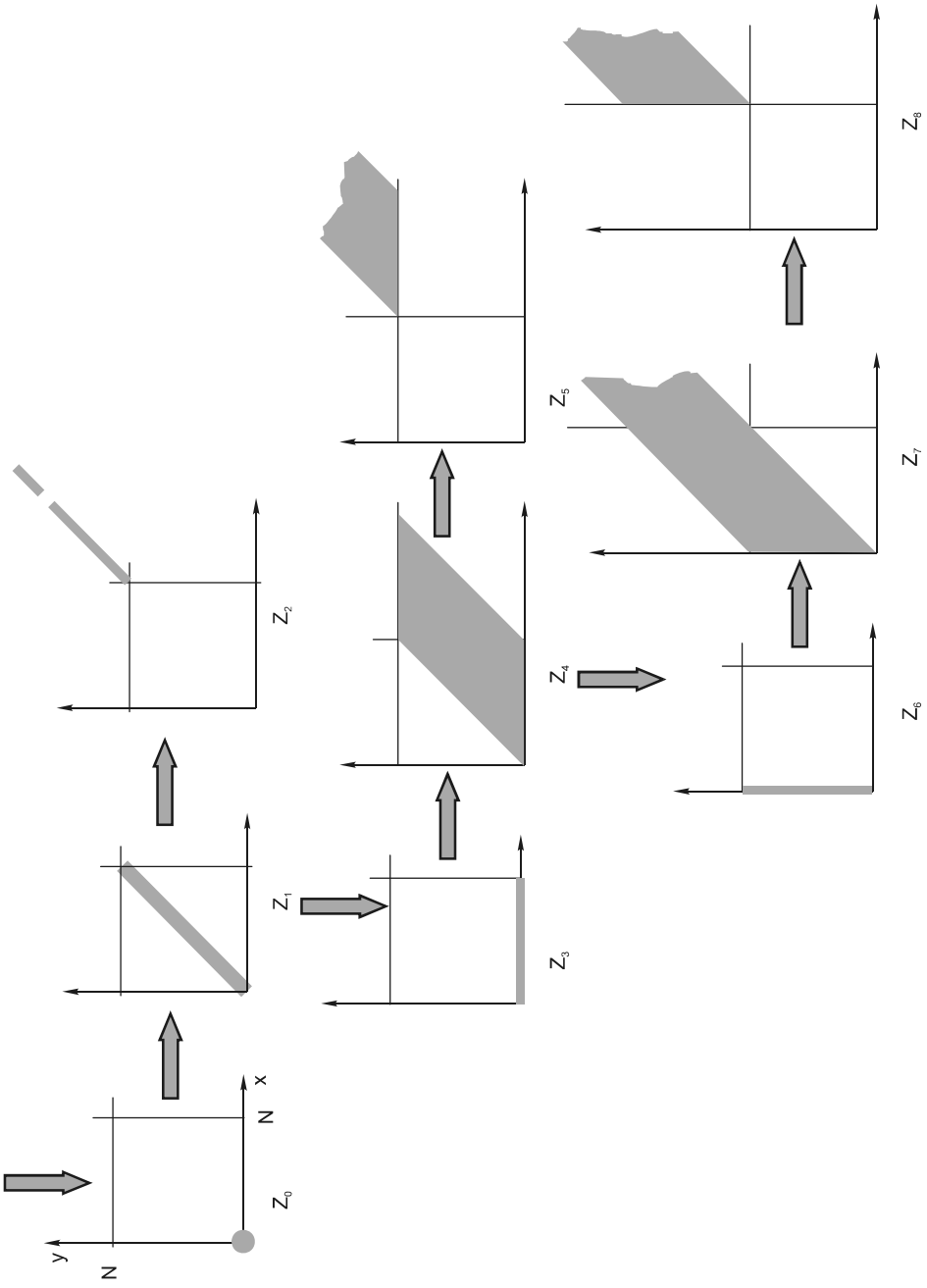


Рис. 12.26. Временные зоны для трех множеств вычислений протокола Фишера

вает последовательность зон, в которых могут находиться интерпретации локальных часов процессов P_1 и P_2 при указанных на рис. 12.25, *б* траекториях графа зон этого протокола. Фрагмент графа зон показывает, что одновременное нахождение в критической секции обоих процессов невозможен. При любом N граф зон для протокола Фишера содержит только 17 состояний. При $N=10$ граф зон имеет на три порядка меньше символьных состояний, чем граф регионов этого протокола. Например, только одна зона $Z_3=(0\le y-x\le N)\wedge(0\le y<N)$ при $N=10$ объединяет в себе 500 регионов.

Структуры данных для представления временных зон

Рассмотрим формальное определение зон и то, как зоны могут быть представлены в памяти компьютера.

Простейшие временные зоны определяются наборами ограничений, которые встречаются во временном автомате. При построении графа зон с зонами выполняются операции, отражающие изменения показаний часов при нахождении автомата в локации и при переходах между локациями. Оказывается, что любые построенные таким образом зоны могут быть представлены единым образом, одной структурой данных.

❖ ***Определение 12.12 (временная зона)***

Временная зона Z — это ограничение на показания часов в форме конъюнкции элементарных ограничений, задающаяся грамматикой:

$$Z ::= x_i \sim c \mid x_i - x_j \sim c \mid Z \wedge Z$$

где x_i, x_j — локальные часы временного автомата, c — целое число, $x_i, x_j \in X$, отношение, а '$\sim$' — все возможные неравенства: $\sim \in \{<, \le, \ge, >\}$.

Пример зоны для двух локальных часов, x_1 и x_2:

$$Z1=(x_2-x_1\ge 1)\wedge(0<x_1<1)\wedge(0\le x_2<3).$$

На рис. 12.27 зона $Z1$ выделена сплошными прямыми, соответствующими этим неравенствам.

Для удобного единообразного задания зон вводятся специальные часы x_0, значения которых всегда равны 0. Тогда, если временной автомат содержит

n локальных часов $X=\{x_1, x_2, \dots x_n\}$, то любая зона может быть задана конъюнкцией однотипных неравенств:

$$\Lambda_{0\le i\ne j\le n}\ x_i - x_j \sim c_{ij}$$

Такой набор неравенств можно свести только к отношениям $<$ или $\le$. Например, зона $Z1$ может быть представлена следующим набором однотипных неравенств:

$$Z1=(x_1 - x_2 \le -1)\wedge(x_0 - x_1 < 0)\wedge(x_1 - x_0 < 1)\wedge(x_0 - x_2 < 0)\wedge(x_2 - x_0 < 3)$$

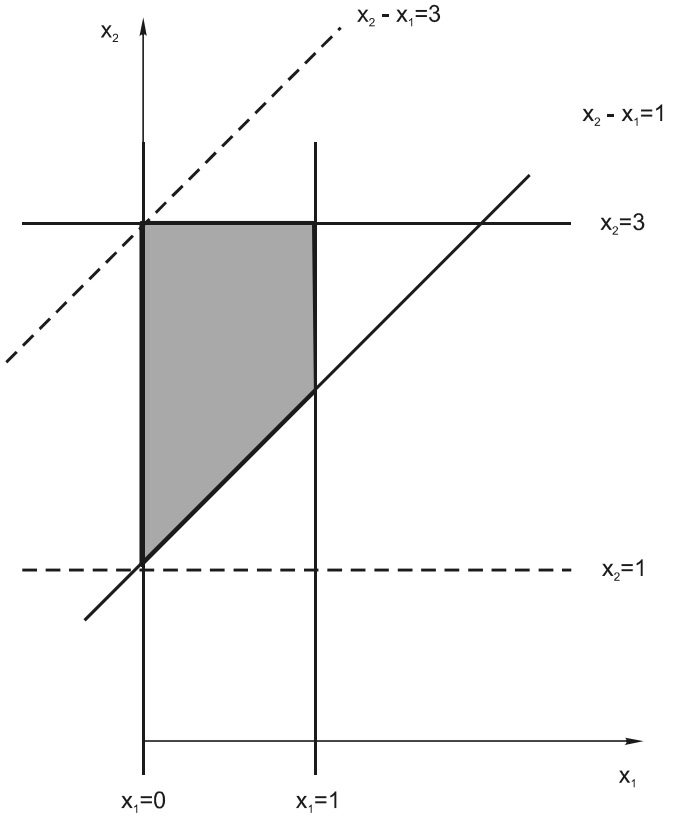


Рис. 12.27. Зона $Z1$, определяемая набором границ

Матрица разностей границ (Difference Bound Matrix, DBM)

Любую зону, заданную конъюнкцией однотипных неравенств, можно задать следующей матрицей $D(Z)=(d_{ij}, </\le)$, где $d_{ij}=x_i - x_j$. Такая матрица для зоны $Z1$ имеет вид:

	x_0	x_1	x_2
x_0	(0,≤)	(0,<)	(0,<)
x_1	(1,<)	(0,≤)	(-1,≤)
x_2	(3,<)	(∞,<)	(0,≤)

На диагонали матрицы $D(Z)$ всегда стоят пары $(0, \leq)$. Первая колонка матрицы представляет верхние границы часов в зоне. Например, $d_{2,0} = (3, <)$ в этой матрице, поскольку $x_2 - x_0 < 3$ в зоне $Z1$. Верхняя строка матрицы представляет нижние границы часов. Если какая-либо граница в задании зоны не определена (в нашей матрице это $d_{2,1}$), она считается бесконечной, и в соответствующем месте матрицы ставится пара $(\infty, <)$.

Подобная матрица называется *Матрицей разностей границ* (Difference Bound Matrix, DBM). Фактически, DBM — это структура данных, символьное представление множества значений локальных часов временного автомата, эквивалентных относительно свойств, проверяемых при верификации.

Оказывается, что одна и та же зона может быть задана разными наборами неравенств и, следовательно, разными матрицами. Например, зона $Z1$ не изменится, если к ней добавить ограничения $x_2 > 1$, и/или $x_2 - x_1 < 5$, или $x_2 - x_1 < 3$. Однако существует канонический вид таких матриц, однозначно определяющий зону. Канонической будет матрица при максимально узких границах. На рис. 12.27 видно, что для задания зоны $Z1$ неравенства $x_2 - x_1 < 3$ и $x_2 > 1$ действительно излишни: на этом рисунке они представлены штриховыми линиями. Добавление их, однако, делает задание таких зон однозначным. Ниже приведен канонический вид матрицы, задающей зону $Z1$:

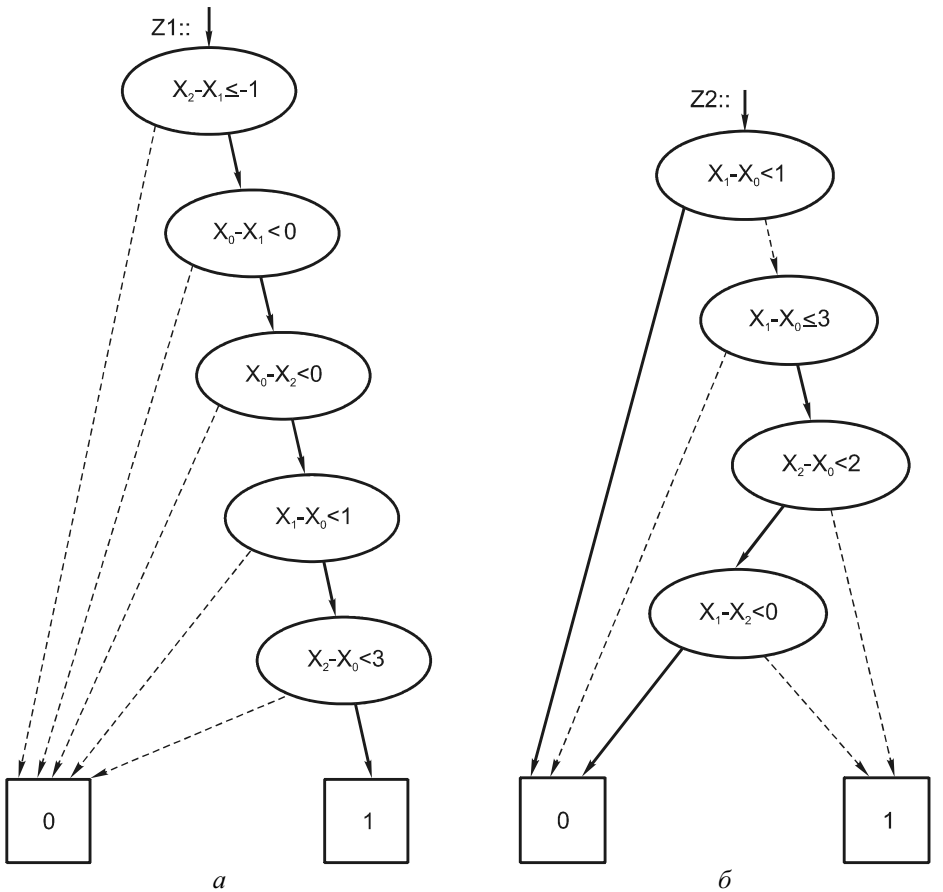
	x_0	x_1	x_2
x_0	(0,≤)	(0,<)	(-1,<)
x_1	(1,<)	(0,≤)	(-1,≤)
x_2	(3,<)	(3,<)	(0,≤)

Определенные ранее операции над зонами могут быть легко реализованы в случае, если зоны представлены матрицами разностей границ. После каждой операции необходимо выполнить приведение матрицы к каноническому виду. Существует простой алгоритм, выполняющий эту операцию.

Матрицы разностей границ могут эффективно представлять множества соседних временных регионов. Однако этой структурой данных оказалось невозможно эффективно описать множества регионов, которые не являются соседними. Поскольку такие множества могут быть описаны логическими формулами, связывающими все временные ограничения, их удобно представлять с помощью бинарных решающих диаграмм (BDD). На этой основе разработаны эффективные алгоритмы символьной верификации систем реального времени.

Разностная решающая диаграмма (Difference Decision Diagram, DDD)

Другой метод символьного представления множеств значений локальных часов во временном автомате основан на использовании так называемых *раз-*



а *б*

Рис. 12.28. Разностные решающие диаграммы для представления зон и их объединений

ностных решающих диаграмм (Difference Decision Diagrams, DDDs). Разностная решающая диаграмма строится полностью аналогично Бинарным решающим диаграммам (BDD), только роль двоичных переменных здесь играют неравенства вида $x_i - x_j < c_{ij}$ или $x_i - x_j \le c_{ij}$. Пример разностной решающей диаграммы, задающей зону $Z1$, представлен на рис. 12.28, *а*. DDD, однако, позволяют представить не только зоны, но и объединения зон, выражаемые произвольными логическими операциями над неравенствами. Например, разностная решающая диаграмма, рис. 12.28, *б*, представляет собой зону $Z2 = (1 \le x_1 \le 3) \wedge (2 \le x_2 \vee x_1 - x_2 < 0)$.

Целью использования разностных решающих диаграмм является упрощение представления объединений временных зон и ускорение проверки принадлежности конкретного набора показаний часов такому объединению. Разностные решающие диаграммы обладают меньшей гибкостью, чем BDD, для них вопросы упорядоченности, редукции, выполнения над ними логических операций решаются не так элегантно, как для BDD. Однако применение их в алгоритмах проверки некоторых свойств систем реального времени имеет определенные перспективы.

12.15. Инструменты верификации систем реального времени

Существует несколько программных сред для верификации систем реального времени. Кратко охарактеризуем две из них. Эти инструменты автоматизированной верификации временных автоматов сами выполняют построение параллельной композиции временных автоматов, построение графа регионов и/или графа зон, а также проверку выполнения свойств, выраженных формулами CTL и TCTL для не очень больших систем.

Uppaal

Uppaal — это программная среда для моделирования и верификации методом "проверки моделей" систем реального времени, разработанная объединенной командой двух университетов, Uppsala University, Швеция, и Aalborg University, Дания, под руководством W.Yi, Kim Larsen и P. Peterssen. Система состоит из двух подсистем, графического интерфейса пользователя и подсистемы верификации.

Графический редактор позволяет очень удобно описывать временные автоматы с целыми переменными (с ограниченным набором возможных значений), попарно взаимодействующие методом рандеву.

Подсистема верификации строит граф регионов для параллельной композиции временных автоматов. Система позволяет проверять, фактически, только свойства, выраженные с помощью одноуровневых (не вложенных) темпоральных формул логики CTL вида **AG**p или **EF**p, где p — это булева комбинация атомарных предикатов и ограничений на значения локальных часов. Возможна также проверка ограниченного во времени причинного следствия событий. Даже при этом ограничении, однако, можно выразить такие свойства систем, как безопасность, ограниченная живость и достижимость, например:

- **EF**p — достижимость (например, $\mathbf{EF}(x \geq y)$ — из начального состояния достижимо состояние, в котором показания часов x не меньше, чем показания часов y);
- **AG**p — безопасность $(\mathbf{AG}(A@s \Rightarrow x \geq 5))$ — при функционировании автомата A всегда, если автомат попадает в локацию s, то показания его часов x не меньше 5;
- $p \to q$ — причинно-следственная связь утверждений p и q, которая сокращенно обозначает формулу $\mathbf{AG}(p \Rightarrow \mathbf{AF}q)$ (это свойство под именем "leads-to" было введено в *гл. 6*).

Для проверки более сложных свойств в среде Uppaal требуется строить контрольный автомат (observer, или watchdog). Если свойство не выполняется, система выдает диагностическую траекторию.

Система Uppaal весьма удобна в использовании, она поставляется свободно для некоммерческих применений.

Kronos

Инструментальный пакет верификации сложных систем реального времени Kronos был разработан исследовательской группой Университета Гренобля и лаборатории VERIMAG, Франция, в 2002 г. (первая версия — в 1998 г.) В качестве модели используется сеть взаимодействующих временных автоматов, требования корректности выражаются формулами TCTL.

Для представления всех вычислений временного автомата Kronos использует граф зон. Для построенного графа зон Kronos может выполнить проверку свойств, выраженных в полной версии логики TCTL, для чего применяются прямой и обратный алгоритмы проверки достижимости (forward and backward reachability) с использованием матриц разностей границ (DBM) для хранения зон. Свойства, которые может проверять система Kronos, могут быть доволь-

но сложными. Например: "не более чем через 20 единиц времени с момента, когда приближающегося поезда нет, шлагбаум будет открыт", которое выражается в логике TCTL так:

$$init \Rightarrow \mathbf{AG}\left(\neg near \Rightarrow \neg \mathbf{E}\left(\left(\neg near \wedge \neg open\right)\mathbf{U}_{>20} true\right)\right).$$

Кроме того, система позволяет проверять эквивалентность двух систем, абстрагированную от времени (time-abstracting equivalence).

Система Kronos позиционируется как инструмент для продвинутых пользователей, имеющих основательную подготовку в теоретических вопросах верификации и построения формальных моделей.

12.16. Заключение

В настоящее время анализ свойств систем реального времени — это "горячая" область, в которой проводятся интенсивные исследования как в академических кругах, так и в индустрии. Поэтому многие термины здесь еще не устоялись, многие результаты являются только предварительными. Но даже эти предварительные результаты являются очень обнадеживающими, они показывают большие возможности подхода model checking для верификации сложных практических систем реального времени.

Одним из главных вопросов в этой области является следующий: можно ли системы реального времени описать формальной моделью, которая бы адекватно представляла поведение практических систем, разрабатываемых в различных областях? Существенным результатом здесь является модель временного автомата, которая была разработана сравнительно недавно ("Timed automata are recent models...", [13]). Существует много других формализмов в этой области, однако именно временной автомат оказался одной из самых мощных и удобных моделей для представления поведения систем реального времени. Этот формализм расширяет модель классического автомата введением локальных часов — вещественных переменных специального типа, значение которых возрастает синхронно со временем. Эти переменные играют роль таймеров, с помощью которых можно измерять временные интервалы между событиями. С помощью такого механизма оказалось возможным выразить задержки между переходами системы, длительность выполнения операций и т. д. Модель временного автомата в настоящее время можно считать стандартной для представления систем реального времени.

У временного автомата число состояний (в обычном понимании этого слова) — бесконечное число, каждый переход в таком автомате может быть представлен бесконечным числом переходов разбиением временной задержки на произвольно большое число интервалов. Поэтому система переходов,

представляющая вычисления временного автомата, не может быть проанализирована без ее факторизации. Понятие временного региона как бесконечного множества наборов показаний локальных часов, эквивалентных с точки зрения проверяемых свойств, является ключевым в анализе систем реального времени. Это понятие позволяет свести бесконечное (континуум) число состояний временного автомата к конечному числу классов (символьных состояний). С помощью регионов удалось показать разрешимость многих проблем верификации временных автоматов: любые верификационные свойства поведения временного автомата имеют один и тот же ответ для наборов значений локальных часов, находящихся в одном регионе.

Недостатком графа регионов является астрономическое число регионов даже для небольших временных автоматов: число регионов экспоненциально растет как от количества локальных часов, так и от значений максимальных констант в ограничениях на их значения. Поэтому непосредственное использование регионов для практической верификации невозможно. Существует несколько подходов, позволяющих манипулировать множествами регионов. Одним из самых очевидных является использование бинарных решающих диаграмм (BDD) для символьного представления множеств регионов. Такой подход был реализован, например, в системе COSPAN. Другие решения в этом направлении используют понятие временной зоны.

Для систем реального времени требуется анализ количественных характеристик поведения систем, чего не может дать обычная темпоральная логика. Для выражения свойств систем реального времени было предложено несколько логик, одной из них является довольно ясное расширение логики CTL введением временных границ на действие темпорального оператора Until и выводимых из него операторов. Формулы логики TCTL (Timed CTL) интерпретируются на деревьях вычислений временного автомата.

Верификация временного автомата относительно формулы логики TCTL сводится к верификации соответствующего графа регионов относительно некоторой CTL-формулы с введением дополнительных "формульных" часов, устанавливаемых в 0 в состоянии, в котором проверяется формула. Поэтому алгоритм model checking для проверки выполнения формулы Φ логики TCTL для графа регионов состоит в последовательном выделении подформул формулы Φ и нахождении такого множества состояний графа, на которых эти подформулы выполняются. При этом, фактически, используется та же идея пометок состояний автомата подформулами формулы Φ, что и при верификации структур Крипке.

Сложность таких алгоритмов при использовании графа регионов весьма велика. Временные зоны позволяют анализировать свойства временного автомата, манипулируя только неравенствами из описания временного автомата и

символически задавать огромные множества регионов. Для представления временных зон можно использовать матричное представление набора линейных неравенств в виде Difference Bound Matrix — форму, исследованную еще Ричардом Беллманом в его книге "Динамическое программирование" в 1957 г. Для этого представления символьных состояний временного автомата были разработаны и алгоритмы верификации, в частности, алгоритмы проверки выполнимости формул логики TCTL для временных автоматов. Несмотря на то, что эта область еще находится в стадии интенсивных исследований, уже разработаны опытные инструментальные системы (Kronos, Uppaal и др.) для анализа количественных свойств поведения систем в рамках расширений темпоральных логик. Существуют примеры промышленного применения изложенных здесь теоретических результатов.

Многие проблемы верификации для систем реального времени алгоритмически неразрешимы. Например, проблема model checking становится неразрешимой при следующих расширениях модели временного автомата:

- ограничения на переходах (защиты) представлены линейными неравенствами (например, $2x > 3y$);
- используются защиты переходов, в которых показания часов сравнивают с нерациональными числами;
- на переходах присутствуют изменения часов вида $x := x - 1$;
- присутствуют операторы остановки часов.

Проблема проверки модели для формул логики LTL, расширенной указанием временных интервалов действия темпоральных операторов, также алгоритмически неразрешима. Если во временной логике LTL запретить использование сингулярных интервалов ($\mathbf{U}_{=c}$), то в такой постановке проблема проверки модели является разрешимой.

Исследования в этой области продолжаются. В основном они направлены на поиск методов более компактного представления временных автоматов и разработку более эффективных алгоритмов проверки их свойств.

12.17. Замечания

Временные автоматы, как формальная модель для представления систем реального времени, были введены в [2, 4, 5] и исследовались в множестве других работ. В работе [71] была предложена более интуитивная форма временного автомата с инвариантами в состояниях, обеспечивающими прогресс. Именно эта модель рассмотрена нами в данной главе; она используется в нескольких инструментах верификации для анализа систем реального времени, например, Uppaal и Kronos.

Но временные автоматы — это только один из формализмов, разработанных для моделирования поведения и верификации систем реального времени. Разработано большое число формализмов для анализа подобных систем [70]. Другие формализмы из этого класса: временные сети Петри (Timed Petri Nets) [15], временная алгебра процессов (Timed Process Algebra) [131] и др. Операция параллельной композиции временных автоматов также может быть определена по-разному. Здесь используется, фактически, операция параллельной композиции из исчисления CCS Р. Милнера, которая принята в языках моделирования Uppaal и Kronos.

Техника, при которой пространство состояний графа регионов не строится для верификации, а вместо этого множество состояний этого графа представляется неявно, символически, с помощью манипуляций множествами неравенств, наложенных на значения локальных часов, была предложена в [71]. Временные зоны показаны в работе [8].

Логика TCTL (Timed Computational Tree Logic) предложена в 1991 г. Алуром как расширение CTL. В работе [70] был исследован вопрос использования логики TCTL для верификации систем реального времени. Разрешимость верификации временных автоматов относительно формул TCTL была показана в [2] и [3].

В работе [152] описывается опыт авторов по использованию символьного подхода к реализации системы анализа свойств реального времени в рамках подхода "проверка модели". Протокол FDDI, модель которого представлена в форме временного автомата, описан в [81]. Описания систем верификации Uppaal и Kronos можно найти в первом томе журнала Journal on Software Tools for Technology Transfer за 1997 г. ([102], [153]). Разностные решающие диаграммы, проблемы, возникающие при их построении, и их применение для проверки некоторых свойств систем реального времени описаны в [111]. Подробное и четкое изложение принципов верификации для систем реального времени дано в [19].

12.18. Задачи к главе 12

12.1. Постройте граф регионов для временного автомата с тремя локациями и одним таймером с потолком 3.

12.2. Постройте структуру регионов с двумя часами x и y, ограниченными константами $C_x = 2$ и $C_y = 3$.

12.3. Постройте траекторию изменения значений таймеров x и y при выполнении некоторой последовательности шагов для графа регионов A_5 (см. рис. 12.13).

12.4. Пусть во временном автомате разрешено действие $x := y$ — присваивание значений одного таймера другому таймеру. Покажите на структуре регионов для двух часов, как это присваивание изменяет регион, в котором находится граф регионов.

12.5. Для временного автомата, показанного на рис. 12.29, постройте граф регионов. Проверьте, выполняется ли в этом автомате формула $\mathbf{AF}(s_1 \wedge x = 1)$ логики CTL.

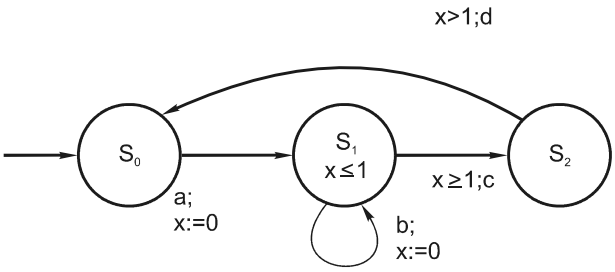


Рис. 12.29

12.6. Проверьте, выполняется ли формула $\Phi = \mathbf{EF}_{\leq 1} n$ для временного автомата A, представленного на рис. 12.30.

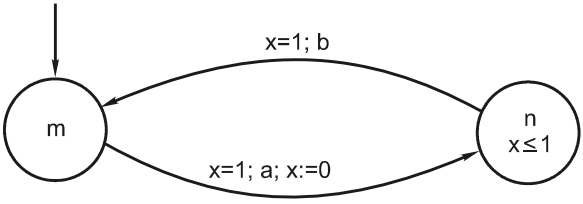


Рис. 12.30

❖ ***Указание***

Замените формулу Φ логики TCTL на формулу $\mathbf{EF}\big((z \leq 1) \wedge n\big)$ логики CTL и постройте граф регионов для автомата $A \oplus z$.

12.7. Объясните по рис. 12.23, почему протокол Фишера корректен.

12.8. Дайте формальное определение зон на рис. 12.23.

12.9. Достройте граф зон протокола Фишера (рис. 12.26).

12.10. Постройте канонические матрицы разностей границ для следующих зон:

$$Z_1 = 2 \le y < 5 \wedge y \le x \wedge x - 2 \le y$$

$$Z_2 = x - y < 2 \wedge y \le 2 \wedge x \ge 1$$

Заключение

Встроенные программно-аппаратные системы, на проверку которых в основном направлены описанные в монографии подходы, используются в областях применений, критических с точки зрения безопасности: медицинское оборудование, системы управления в автомобилях, бортовое оборудование спутников, авиационной, космической и военной техники, мобильные системы. Ошибки в таких системах принятия решений могут иметь катастрофические последствия. Многочисленные примеры последствий ошибок автоматических систем принятия решений, от "дружественного огня" в военных действиях до ошибок в системах управления космическими аппаратами, показывают необходимость повышения уровня надежности разрабатываемых программных и аппаратных систем. Обеспечение правильного функционирования таких систем является важнейшей проблемой при их разработке. Основным средством повышения надежности таких систем является их верификация.

Еще несколько лет назад верификация была искусством, логическим дедуктивным рассуждением, требовавшим значительного времени — многих месяцев работы квалифицированных специалистов для проверки правильности функционирования даже небольших программ. Представленный в монографии метод *model checking* превратил верификацию программных и аппаратных систем из искусства, которым были первые шаги дедуктивной верификации, в производственную технологию, которая достигла высокой степени автоматизации, фактически, уровня "нажатия кнопки". Описанные в этой книге методы верификации дискретных систем, объединенные под общим названием model checking, с успехом применяются как для верификации программных и аппаратных систем, так и во многих других областях. Многие основные компании по разработке ПО и аппаратуры в настоящее время интенсивно используют верификацию на основе model checking. Использование верификации для проектов в фирмах-разработчиках программных систем

привело к многократному уменьшению ошибок в их продуктах, сокращению времени выведения продуктов на рынок, снижению их стоимости, повышению производительности при их разработке. Изложенные в тексте алгоритмы позволяют полностью автоматизировать процесс верификации, хотя и требуют от разработчика достаточно высокой квалификации при построении модели проверяемой системы, при формулировке проверяемых свойств на языке темпоральной логики, а также при интерпретации результатов верификации.

Хотя представленный в монографии материал лишь в незначительной степени покрывает те элегантные теоретические результаты, а также эффективные технические приемы и алгоритмы, которые получены к настоящему времени в области верификации методом model checking, автор надеется, что ему удалось показать, как формальные методы можно продуктивно использовать в анализе программных систем аналогично тому, как формальные методы и математика в целом используются в инженерном деле. И здесь и там разрабатываются символические модели существенных свойств объектов и средства анализа этих моделей. В строительстве используется сопротивление материалов, в авиастроении — аэродинамика, в электроэнергетике строятся модели генераторов и линий передач. Эти разделы прикладной математики применяются инженерами-разработчиками для построения систем с гарантированным выполнением ими требуемых функций. Поскольку программные системы задают специфические динамические процессы, инженеры-разработчики программных средств должны уметь строить модели дискретных систем, использовать формальную логику для спецификации свойств поведения таких систем и разрабатывать алгоритмы для проверки выполнения этих свойств в своих разработках.

Список литературы

Иностранные издания

1. S.B. Ackers. Binary decision diagrams. IEEE Trans Comp. C-27, N6, 1978.
2. R. Alur, C. Courcoubetis, D. Dill. Model checking for real-time systems // Proc. of the 5th Annu. Symp. on Logic in Computer Sci. Notes in Computer Sci, 1990.
3. R. Alur, C. Courcoubetis, D. Dill. Model checking in dense real time. // INformation and Computation, v. 104, n. 1, 1993.
4. R. Alur, D. Dill. Automata for modeling real-time systems // LNCS, v. 443, Springer-Verlag, 1990.
5. R. Alur, D. Dill. A theory of timed automata // Theoretical Computer Sci, v. 126, n. 2, 1994.
6. R. Armoni, L. Fix, A., et al. The ForSpec Temporal Logic — A New Temporal Property-Specification Language // Proc. 8th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, 2002.
7. B. Anderson, J. Hansen, et al. The application of model checking for securing e-commerce transactions // Communications of the ACM, v. 49, N6, 2006.
8. R. Alur. Timed Automata // NATO ASI Summer School on Verification of Digital and Hybrid Systema, 1998.
9. Henrik Reif Andersen. An Introduction to Binary Decision Diagrams // Technical Uni of Denmark, 1998.
10. Mordechai Ben-Ari. Principles of the Spin Model Checker // Springer, 2008.
11. B. Berard, M. Bidoit, et al. Systems and Software Verification. Model-Checking Techniques and Tools // Springer-Verlag, 2001.

12. T. Ball, E. Bounimova, et al. Thorough Static Analysis of Device Drivers // EuroSys 2006.
13. P. Bouyer, F. Chevalier, N. Markey. On the expressiveness of TPTL and MTL // LNCS, vol. 3821, 2005.
14. J. R. Burch, J. M. Clarke, K. L. McMillan, et al. Symbolic model checking: 10^{20} states and beyond // In: IEEE Symp. on Logic in Comput. Sci., 1990, Information and Computation, v. 98, N2, 1992.
15. B. Berthomieu, M. Diaz. Modeling and verification of timed dependent systems using timed petri nets // IEEE Trans. on Software Engineering, v. 17, n. 3, 1991.
16. Evert Willem Beth. The Foundations of Mathematics // North Holland, 1959.
17. G. Bruns, P. Godefroid, Temporal Logic Query-Checking // Proc. 16th Ann. IEEE Symp. Logic in Computer Science, 2001.
18. F. Bacchus, F. Kabanza. Using Temporal Logics to express search control knowledge for planning. Artificial Intelligence, v. 116, 2000.
19. Christel Baier, Joost-Pieter Katoen. Principles of Model Checking // The MIT Press, 2008.
20. A. Biere, A. Kick. A case study on different modeling approaches based on model checking — verifying numerous versions of the alternating bit protocol with SM // Universitat Karlsruhe, 1995.
21. J. Bormann, J. Lohse, M. Payer, G. Venzl. Model checking in industrial hardware desigh // Proc. of Int. Design Automation Conference, 1995.
22. G. Bochmann. Finite state description of communication protocols // Comput. Networks, v. 2, 1978, pp. 361-371.
23. P. Bertoli, M. Pistore, M. Roveri. Planning as model checking // AIPS'02 Hands-On Tutorial, 2002.
24. Kevin Brewer. Binary decision diagrams and automated verification // Report 2004.
25. R. Briant. Graph Based Algorithms for boolean function manipulation // IEEE Transaction on Computers, v. C-35, no 8, 1986.
26. J. R. Buchi. On a decision method in restricted second order arithmetic // Proc. Internat. Congr. Logic, Method and Philos. Sci. Stanford, 1962.
27. A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A New symbolic model checker // International Journal on Software Tools for Technology Transfer, 2(4):410-425, 2000.
28. G. Cabodi, P. Camurati, S. Quer. Can BDDs compete with SAT solvers on bounded model checking? // Proc. of the Design Automation Conference, 2002.

29. Edmund M. Clarke, E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic // Logic of Programs, 1981.

30. E. M. Clarke, E. A. Emerson, A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications: A practical Approach // Proc. 10th ACM Symp. on Principles of Programming Languages, 1983.

31. E. M. Clarke, E. A. Emerson, A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications // ACM Trans. Program. Lang. Syst., 8(2), 1986.

32. O. Coudert, H. Fraisse, J. Madre. A breakthrough in two-level logic minimization // Int Workshop on Logic Synthesis, 1993.

33. E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. Counterexample-guided Abstraction Refinement // Proc. Int. Conf. Computer Aided Verification, 2000.

34. E. M. Clarke, O. Grumberg, D. Peled. Model checking // MIT Press, 1999. (Э. М. Кларк, О. Грамберг, Д. Пелед. Верификация моделей программ: Model Checking // М., 2002).

35. W. Chan. Temporal logic queries // 12th Conf. Computer Aided Verification (CAV00), 2000.

36. S. Chandra et al. Software model checking in practice: An Industrial Case Study, 2001.

37. A. Cimatti et al. Model Checking Safety Critical Software with SPIN: an Application to a Railway Interlocking System // LNCS 1516, Springer-Verlag, 1998.

38. J. C. Corbett et al. Extracting finite state models from Java source code // In Proc. 22rd Int Conf. on Software Engineering, 2000.

39. O. Coudert. Doing two-level logic minimization 100 times faster // Proc.Symp. on Discrete Algorithms, 1995.

40. H. Cao, S. Ying, D. Du. Towards model-based verification of BPEL with model checking // In: 6th IEEE Int. Conf. on Computer and Informat. Technology, 2006.

41. M. B. Dwyer, G. A. Avrunin, J. C. Corbett. Property specification patterns for finite-state verification // Proc. of the 2nd Workshop on Formal Models in Software Practice, 1998.

42. C. Dong, P. Molitor. What graphs can be efficiently represented by BDDs // Int Conf on Computing: Theory and Applications, 2007.

43. Rolf Drechsler, Detlef Sieling. Binary decision diagrams in theory and practice // International Journal on STTT, v 3, 2001.

44. E. W. Dijkstra, Derivation of a termination detection algorithm for distributed computations // Tech. Rep. EWD-840, Uni. of Texas at Austin.

45. E. A. Emerson, J. Y. Halpeirn. "Sometimes" and "not never" revisited: On branching versus linear time temporal logic // Proc. of the Annual ACM Symp. on Principles of Programming Languages, 1982.

46. E. Allen Emerson, Chin-Laung Lei. Modalities for model checking: branching time logic strikes back. Sci of Computer Programm, 8(3), 1987.

47. E. A. Emerson, A. K. Mok, A. P. Sistla, J. Srinivasan. Quantitative temporal reasoning // Computer Aided verification, 1990.

48. E. A. Emerson. Temporal and modal logic // In: J. van Leeuwen, ed. Handbook of Theoretical Copmputer Science, vol. B, 1990.

49. Rik Eshuis. Symbolic model checking of UML activity diagrams // ACM Transactions on Software Engineering and Methodology (TOSEM). v. 15, n. 1, 2006.

50. R. W. Floyd. Assigning meaning to programs. // Proc. Symposium on Applied Mathemtics, v. 9, 1967.

51. F. Laroussinie. About the expressive power of CTL combinators. // Information Processing Letters, (54) 6, 1995.

52. Federal Study, US Dept of Commerce, May, 2002.

53. D. Gabbay. The declarative past and imperative future: Executable Temporal logic for interactive systems // Proc. 1st Conf. on Temporal Logic in Specification (TLS'1987), 1987, LNCS, v 398, Springer-Verlag, 1987.

54. A. Gurfinkel, M. Chechik, B. Devereux. Temporal logic query checking: a tool for model exploration. IEEE Trans. on Software Engineering, v. 29, N10, 2003.

55. D. Giannakopoulou, F. Lerda. Efficient translation of LTL formulae into Buchi automata. // NASA Ames Research Center, TR, 2001.

56. Stephen J. Garland, Nancy Lynch. Using I/O automata for developing distributed systems // MIT Press, 1999.

57. R. Gentilini, C. Piazza, A. Policriti. Computing strongly connected components in a linear number of symbolic steps // Proc. of the 14th Annu ACM_SIAM Symp on Discrete Algorithms, 2003.

58. D. Gabbay, A. Pnueli, S. Shelan, J. Stavi. The temporal analysis of fairness // Proc.7th ACM Symp. on Principles of Programming Languades, 1980.

59. F. Giunchiglia, P. Traverso. Planning as Model Checking // In Proc. 5th European Conference on Planning (ECP-99), LNAI. Springer-Verlag, 1999.

60. A. Gurfinkel, O. Wei, M. Chechik. Yasm: A Software Model-Checker for Verification and Refutation // LNCS, v. 4144, Springer-Verlag, 2006.

61. D. Harel. Statecharts: A visual formalism for complex systems // Sci.Comput. Prog. v. 8, 1987.

62. Nicolas Halbwachs. Synchronous programming of reactive systems // Kluwer Academic Publishers, 1993.

63. J. Holzmann, D. Bosnacki. The Design of a Multi-Core Extension of the Spin Model Checker // IEEE Trans. Software Eng., v 33, N10, 2007.

64. H. Hermanns. Performance and reliability model checking and model construction // In: Proc. Int Conf. Formal Methods for Industrial Critical Systems, 2000.

65. Hans Hansson, Bengt Jonsson. A Logic for Reasoning about Time and Reliability // Formal Aspects of Computing, v. 6, no 5, 1994.

66. H. Hermanns, J.-P. Katoen, et al. A Markov chain model checker // LNCS, v.1785, Springer-Verlag, 2000.

67. K. Havelund, M. Lowry, J. Penix. Formal analysis of a space-craft controller using Spin // IEEE Trans. Software Eng., v. 27, 2001

68. C.A.R. Hoare. An axiomatic basis of computer programming. // Communications of the ACM, v. 12, N10, 1969.

69. K. Havelund, M. Lowry, J. Penix. Formal analysis of a space-craft controller using spin // IEEE Trans. Software Eng., v. 27, 2001.

70. C. Heitmeyer, D. Mandrioli, editors. Spesial Issue: Formal models for real-time computing // Trends in Software, n. 5 1996.

71. T. A. Henzinger, X. Nicollin, J. Sifakis, S. Yovine. Symbolic model checking for real-time systems. // Proc. 7th Symp. on Logic in Computer Science, 1992.

72. T. Henzinger, X. Nicollin, J. Sifakis, S. Yovine. Symbolic model checking for real-time systems // J. of Information and Computation, v. 111, n. 2, 1994.

73. M. Huth, M. Ryan. Logic in computer science: Modelling and Reasoning about Systems // Cambridge University Press, 1999.

74. S. Hornus, P. Schnobelen. On solving temporal logic queries // Proc. 9th Int. Conf. on Algebraic Methodology and Software Technology AMAST'02, 2002.

75. G. J. Holzmann, M. H. Smith. A practical method for verifying event-driven software // In Proc. 21st Int Conf. on Software Engineering, 1999.

76. Steve Hilditch, Tom Thomson. Distributed deadlock detection: algorithms and proofs // Dept. Comput. Sci., Uni. of Manchester, Technical Report, UMCS-89-6-1, 1989.

77. G. J. Holzmann. From code to models // Proc. 2nd Int. Conf. on Applications of Concurrency to System Design, 2001.

78. G. J. Holzmann. The Spin model checker — primer and reference manual // Addison-Wesley, 2003.

79. G. Holzmann. Design and validation of computer protocols // Prentice Hall, 1991.

80. Gerard J. Holzmann. The model checker SPIN // IEEE Trans. on Software Eng., 23(5), 1997.

81. R. Jain. FDDI handbook: high-speed networking using fiber and other media // Addison-Wesley, 1994.

82. C. Jard, T. Jeron. Verification and distributed observation of the alternating bit protocol // In IFIP TC6/WG6.1 Joint Int. Conf. on formal description techniques, special session on educational case studies in protocols, Paris, 1998.

83. W. Janssen, R. Mateescu, et al. Model checking for managers // LNCS, v.1680, Springer-Verlag, 2008.

84. A. Josang. Security Protocol Verification using Spin // Proc. SPIN Workshop, 1995.

85. R. P. Jacobi, A. M. Trullemans. A new logic minimization method for multiplexor-based FPGA synthesis // Design Automation Conf., 1993.

86. Joost-Pieter Katoen. Software Modeling and Verification // Graduate Course of Aachen University, 2005.

87. J. Kieffer, P. Flajolet, Yang En-Hui. Data compression via binary decision diagrams, 2000. Proceedings // IEEE International Symposium on Information Theory, 2000.

88. Donald Knuth. The art of computer programming, v.4. A draft of section 7.1.4. Binary Decision Diagrams // Addison Wesley, 2008.

89. M. Kwiatkowska, G. Norman, D. Parker. Probabilistic symbolic model checking with PRISM: a hybrid approach // LNCS v. 2280, Springer, 2002.

90. S. A. Kripke. Semantical consideration on modal logic // Acta Philosophica Fennica, v.16, 1963. Русский перевод [167].

91. M. Kwiatkowska. Model checking for probability and time // Proc. Conf. Logic in Computer Science, 2003, www.cs.bham.ac.uk/~dxp/prism/.

92. L. Lamport. Proving the correctness of multiprocess programs // IEEE Trans on Software Eng., v 3 N2, 1973.

93. L. Lamport. "Sometimes" sometimes is not "never" // Proc.7th ACM Symp.on Principles of Programming Languades, 1980.

94. L. Lamport. What good is temporal logic // Information Processing'83, IFIP Congress, 1983.

95. B. Langlois, J. Barata, D. Exertier. Improving MDD productivity with Software Factories. // First Int. Workshop on Software Factories, 2005.

96. Chung-Hung Lai; Tien-Fu Chen. Compressing inverted files in scalable information systems by binary decision diagram encoding // ACM/IEEE Conf. on Supercomputing, 2001.

97. C. T. Leiserson, T. H. Cormen, R. L. Rivest. Introduction to algorithms // MIT Press, 1997. (Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. Алгоритмы: построение и анализ. — М.: МЦНМО, 1999).

98. C. Y. Lee. Representation of switching circuits by binary decision diagrams // Bell Syst Tech Journal 38, 1959.

99. F. Laroussinie, N. Markey, and Ph. Schnoebelen. Temporal logic with forgettable past // In Proc. 17th IEEE Symp. Logic in Computer Science, 2002.

100. G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR // Tools and Algorithms for the Construction and Analysis of Systems: 2nd Int. Workshop, TACAS 96, 1996.

101. M. Langberg, A. Pnueli, Y. Rodeh. The ROBDD size of simple CNF formulas // CHARME, 2003.

102. Kim G. Larsen, J. Pearson, W. Yi. UPPAAL in a nutshell. // J. on Software Tools for Technology Transfer. v. 1, 1997.

103. D. Lehmann, A. Pnueli, J. Stavi. Impartiality, justice, and fairness: The ethics of concurrent termination // LNCS, v. 115, Springer-Verlag, 1981.

104. N. Lynch. Distributed Algorithms // Morgan Kaufman Publisher, 1966.

105. W. C. Lynch. Computer systems: reliable full-duplex file transmission over half-duplex telephone line // Commun. ACM, v. 11, no. 6, 1968.

106. Stephan Merz. Model Checking // A tutorial overview. LNCS 2067, 2001.

107. Alan Mishchenko. An Introduction to Zero-Suppressed Binary Decision Diagrams // Portland State Uni., 2001.

108. Steven P. Miller. Formal verification of synchronous models: An Industrial application of Formal Methods // Rockwell Collins, 2007.

109. Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems // Proc. of Design Automation Conference, 1993.

110. Shin-ichi Minato. Binary decision diagrams and applications for VLSI CAD, Springer; 1995.

111. J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference Decision Diagrams // Proc. Annual Conf. of the European Association for Computer Science Logic (CSL), 1999.

112. K. L. McMillan. Symbolic model checking — an approach to the state explosion problem // Ph.D. thesis, SCS, Carnegie Mellon University, 1992.

113. K. L. McMillan. Symbolic model checking // Kluwer Academic Publisher, 1993.

114. B. M. E. Moret. Decision trees and diagrams. // ACM Comput. Surv., v.14, 1982.

115. P. Maggi, R. Sisto. Using SPIN to verify security properties of cryptographic protocols // Model Checking Software: 9th International Spin Workshop, 2002.

116. N. Markey, P. Schnoebelen. TSMV — a symbolic model checker for quantitative analysis of systems // 1st Int Conf on Quantitative Evaluation of Systems (QUEST 2004).

117. M. Muller-Olm, D. Schmidt, B. Steffen. Model checking. A tutorial introduction // LNCS 1694, 1999.

118. C. Meinel, T. Theobald. Algorithms and data structures in VLSI design. OBDD Foundation and Applications // Springer, 1998.

119. C. Meinel, T. Theobald. Ordered binary decision diagrams and their significance in computer-aided design of VLSI circuits — a Survey // Electronic Colloquim on Computation Complexity, Report N39, 1998.

120. R. M. Needham, M.D. Schroeder. Using encryption for authentication in large networks of computers // Communication of the ACM, V. 21, N12, 1978.

121. S. Owicki, L. Lamport. Proving liveness properties of concurrent programs // ACM Trans. on Progr. Lang. and Systems, v 4, n3, 1982.

122. OMG UML Specification v. 1.5. // Доступно по адресу http://www.omg.org.

123. R. Puri, J. Gu. A BDD SAT solver for satisfiability testing: An industrial case study. // Annals of Mathematics and Artificial Intelligence, v.17, n2, 1996.

124. P. Pingree, E. Mikk, G. Holzmann, M. Smith, D. Dams. Validation of mission critical software design and implementation using model checking // Proc. of the 21st Digital Avioncs Systems Conf., v. 1, 2002.

125. A. Pnueli. The temporal logic of program // Proc. of the 18th Anny. Symp. on Foundation of Computer Science, 1977.

126. W. Pedrycz, Z. A. Sosnowski. Genetic BDD-oriented pattern classifiers // Springer, 2005.

127. M. Pradella, P. san Pietro, P. Spoletini, A. Morzenti. Practical model checking of LTL with Past // 1st Workshop on Automated Technology for Verification and Analysis, 2003.

128. A.N. Prior. Past, present and future // Oxford University Press, 1967.

129. J.–P. Queille, J. Sifakis. Specification and verification of concurrent programs in CESAR // 5th Int. Symp/ on Programming. LNCS, v. 137, Springer-Verlag, 1982.

130. H. Reichenbach. Elements of symbolic logic // New York: Macmillan, 1947.

131. G. M. Read, A. W. Roscoe. A timed model for communicating sequential processes // Theoretical Computer Science, v. 58, n.1-3, 1988.

132. J. Rushby. Formal Methods and Their Role in Digital System Validation for Airborne Systems // NASA contractor Report N 4673, 1995.

133. P. Schnobelen. The complexity of temporal logic model checking // Advances in Model Checking, v. 4, World Scientific Publishing Co., 2002.

134. P. Stuckey, P. Hawkins, V. Lagoon. Solving set constraint satisfaction problems using ROBDDs // J. of Artificial Intelligence Research, v. 24, 2005.

135. A. Sistla. Safety, Liveness and Fairness in Temporal Logic // Formal Aspects of Computing, 6:495—511, 1994.

136. H. Sutter, J. Larus. Software and the concurrency revolution // ACM Queue, v. 3, N7, 2005.

137. J. Slaney, S. Thiebaux. Blocks world revisited // Artificial Intelligence 125, 2001.

138. D. Sieling, I. Wegener. Reduction of OBDDs in linear time // Inf Process Lett v. 48, N3, 1993.

139. A. Tarski. A lattice-theoretical fixpoint theorem and its applications // Pacific J. of Math., v 5, 1955.

140. M. Y. Vardi, Branching vs. Linear Time: Final Showdown. Proc. 7th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01), 2001.

141. M. Y. Vardi, An Automata-Theoretic Approach to Linear Temporal Logic // LNCS v 1043, Springer-Verlag, 1996.

142. M. Y. Vardi, P. Wolper. An automata-heoretic approach to automatic program verification (preliminary report). // 1st Annu Symp. on Logic in Computer Science, 1986.

143. M. Y. Vardi, P. Wolper. Reasoning about infinite computations // Information and Computation, v. 115, n.1, 1994.

144. Katsumi Wasaki. A Formal Verification of case study for IEEE-P.896 Arbiter by Using a Model Checking Tool // Int Journal of Comput. Sci and Network Security, v. 7, N3, 2007.

145. Pierre Wolper. Constructing Automata from Temporal Logic Formulas: A Tutorial // LNCS v. 2090, Springer-Verlag, 2001.

146. Pierre Wolper. Temporal logic can be more expressive // Information and Control, v. 56(1-2), 1983.

147. P. Woelfel. Symbolic Topological Sorting with OBDDs // Proc of the 28th Int Symp on Math Foundation of Comput Sci. (LNCS, v. 2747), 2003.

148. Pierre Wolper. Expressing Interesting Properties of Programs in Propositional Temporal Logic // Proc. 13th Annu. Symp. on Principles of Programming Languages, 1986.

149. P. Wolper. An introduction to model checking. // Software Quality Workshop, 1995.

150. B. Yang, Y. Chen, R. Briant, D. O'Hallaron. Space and time efficient BDD construction via working set control // ASP Design Automation Conf., 1998.

151. Wing L. Yeung, K. Leung, et al. Model checking suspendible business processes via statechart diagrams and CSP // Proc. MSWEIS'06, 2006.

152. J. Yang, A. K. Mok. Symbolic model checking for event-driven real-time systems // ACM Trans. on Progr. Lang. and Systems, v. 19, N2, 1997.

153. S. Yovine. Kronos: a verification tool for real-time systems.// J. on Software Tools for Technology Transfer. v. 1, October, 1997.

154. S. Yovine. Model checking timed automata // LNCS, v. 1494, 1998.

Литература на русском языке

155. А. Аграновский, В. Зайцев, Б. Телеснин, Р. Хади. Верификация программ с помощью моделей. — Открытые системы, № 12, 2003.

156. Н. Ф. Богаченко, Р. Т. Файзуллин. Синтез дискретных автоматов. — Омск: ОмГУ, 2006.

157. М. А. Гаврилов. Современное состояние теории релейных устройств и конечных автоматов в кн. Теория конечных и вероятностных автоматов. — М.: Наука, 1965.

158. Ю. Г. Карпов. Теория автоматов. — СПб.: Питер, 2002.

159. Ю. Г. Карпов. Теория и технология программирования. Основы построения трансляторов. — СПб.: БХВ-Петербург, 2005.

160. Ю. Г. Карпов, А. В. Толстяков. Аналитический метод верификации протоколов. — Автоматика и вычислительная техника, № 1, 1985.

161. Ю. Г. Карпов. Спецификация и верификация протоколов на основе CCS. — Автоматика и вычислительная техника, № 6, 1986.

162. Ю. Г. Карпов. Простой протокол, который доказывается просто. — Автоматика и вычислительная техника, № 1, 1987.

163. Ю. Г. Карпов, А. В. Борщев, А. А. Мещерский. Проверка семантической корректности протоколов с использованием свойства когерентности. — Автоматика и вычислительная техника, № 5, 1989.

164. Ю. Г. Карпов. О корректности параллельных алгоритмов. — Программирование, № 4, 1996.

165. Ю. Г. Карпов. Анализ корректности параллельной программы разделения множеств. — Программирование, № 6, 1996.

166. С. А. Крипке. Семантический анализ модальной логики. I. Нормальные модальные исчисления высказываний в кн. Р. Фейса "Модальная логика". — М.: Наука, 1974.

167. А. С. Нариньяни, В. В. Телерман, Д. М. Ушаков, И. Е. Швецов. Программирование в ограничениях и недоопределенные модели. — Российский НИИ искусственного интеллекта, 2002.

168. Научно-технический вестник Санкт-Петербургского государственного университета информационных технологий, механики и оптики. — СПб.: Вып. 53, 2008.

169. С. Рассел, П. Норвиг. Искусственный интеллект. Современный подход. — М.: Вильямс, 2006.

170. Ч. Чень, Р. Ли. Математическая логика и автоматическое доказательство теорем. — М.: Наука, 1973.

Материалы из Интернета

171. US General Accounting Office Report 92026: **http://www.fas.org/spp/starwars/gao/im92026.htm**.

172. Анализ отказа системы управления Arian 5: **http://forums.airbase.ru/2007/03/12/topic-54337--Avariya-Arian-5-4-iyunya-1996.Podrobnosti-programmnoj-oshibki.html**.

173. CrocoPat: A Tool for Simple and Efficient Relational Computation: **http://mtc.epfl.ch/~beyer/CrocoPat/**.

174. ETMCC: **http://www7.informatik.uni-erlangen.de/etmcc/**.

175. **http://epizodsspace.testpilot.ru/bibl/ivanovskiy/rakety-i-kosmos/20.html**.
176. **http://citcity.ru/17571/**.
177. **http://www.globalsecurity.org/org/news/2003/030326-patriot01.htm**.
178. **http://www.esa.int/esaCP/SEM85Q71Y3E_index_0.html**.
179. Проблема фермер-волк-коза-капуста: **http://www.cut-the-knot.org/ctk/GoatCabbageWolf.shtml**.
180. Википедия. Проблема миссионеров и каннибалов: **http://en.wikipedia.org/wiki/Missionaries_and_cannibals_problem**.
181. **http://ru.wikipedia.org/wiki/Therac-25**.
182. PRISM: **http://www.cs.bham.ac.uk/~dxp/prism/**.
183. TwoTowers (Uni. Bologna): **http://www.sti.uniurb.it/bernardo/twotowers/**.
184. **http://en.wikipedia.org/wiki/Temporal_logic**.
185. Java PathFinder Homepage: **http://javapathfinder.sourceforge.net/**.
186. YASM: **http://www.cs.toronto.edu/~arie/yasm/**.
187. **http://en.wikipedia.org/wiki/SCADE**.
188. **http://research.microsoft.com/slam/**.
189. Spin Workshops: **http://spinroot.com/spin/Workshops/index.html**.
190. Basic Spin Manual: **http://www.spinroot.com/spin/Man/Manual.html**
191. **http://en.wikipedia.org/wiki/Chinese_rings**.
192. BDD-based Logic Synthesis system: **http://www.ecs.umass.edu/ece/labs/vlsicad/cadlab/cadlab.html**.
193. Б. Конев. Курс лекций "Введение в моделирование и верификацию систем": **http://logic.pdmi.ras.ru/~infclub/?q=courses/verification**.
194. Ю. Лифшиц: **http://yury.name/modern.html**.
195. К. Савенков. Курс лекций "Верификация программ на моделях": **http://savenkov.lvk.cs.msu.su/mc.html**.
196. **http://www.hellam/net/math2000/frogs.html**.

Предметный указатель

A

Amber 260
◊ графические элементы 260
Architectural Modelling Box for Enterprise design 260

B

BDD 295, 301, 304, 369
◊ библиотеки 327
◊ в информационном поиске 330
◊ и разложение Шеннона 306
◊ каноничность 308
◊ конечные структуры данных 345
◊ операции 312
◊ представление 309
◊ применение в решении задач 330
◊ с подавлением нулей 358
Binary Decision Diagram 295, 304
Binary Decision Tree 296

C

Clock constraints 478
Computational Tree Logic 75, 91
CTL 41, 91, 118, 403
◊ базисы 100
◊ спецификация справедливости 233
◊ справедливая 237
CTL* 74

D

DBM 523
DDD 525
Difference Bound Matrix 523
Difference Decision Diagram 525

E

Extended Computational Tree Logic 74

J

JavaPathFinder 205

K

Kronos 526

L

Linear Temporal Logic 62
Linear Time Logic 51
LTL 41, 51, 118, 147
◊ разметка состояний 149
LTL-формулы
◊ примеры 218

M

model checking 28, 83, 108, 120, 259, 369, 504
◊ анализ корректности криптографических протоколов 279

model checking *(прод.)*
◊ в системах планирования 267
◊ для CTL 380

P

PAR 177, 250
PCTL 427, 432
◊ эквивалентность 437
Promela 197, 199, 253, 260, 285
◊ управляющие структуры 200

R

Reactive systems 58

S

SLAM 206
Spin 197, 203, 260, 265

T

TCTL 489
◊ вложенные формулы 501
◊ формальная семантика 494
Tense Logic 46
Timed Automaton 446
Timed CTL 489
Timed region 470
Two Phase Commit Protocol 253

U

Uppaal 525

Y

YASM 205

A

Автомат Бюхи 131
◊ обобщенный 138, 158
◊ построение по формуле LTL 147, 157
◊ синхронная композиция 136
Адекватность 175
Алгоритм маркировки 104
Алгоритмы обработки данных 355
Анализ
◊ вероятностных систем 419
◊ схемы — построение BDD по логической схеме 343
Атака на протокол Нидхэма — Шредера 283
Атом формулы 150
Атомарность 173
Атомарный предикат 52, 60, 183, 187
Аутентификация 280

Б

Базис CTL 99, 100, 103
Безусловная справедливость 231
Библиотеки BDD 327
Бизнес-процесс
◊ формальное описание 259
Бинарная решающая диаграмма 295, 301, 524
Бинарное решающее дерево 296
◊ представление булевой функции 296
Бисимуляция 508, 509
Блокировка времени 465
Булева функция
◊ виды 299
◊ вычисление значения 312
◊ минимизация 319
◊ построение BDD 323
◊ представление 296, 299
◊ типичные задачи 300
Булевы операции над BDD 317, 323

B

Валидация 24
Верификация 24, 31
◊ вероятностная 416

◊ временного автомата 482
◊ графа регионов 482
◊ дедуктивная 27
◊ недостатки 31
◊ преимущества и недостатки 36
◊ символьная 367, 369
◊ схемы 344
◊ формальная 25
Верификация программ
◊ общая схема 26
Верификация систем
◊ примеры 33
Вероятностный метод model checking 417
Взаимное исключение 220
Взрыв числа состояний 30, 186
Вложенные формулы TCTL 501
Временной автомат 446, 448, 450, 529
◊ верификация 482
◊ вычисление 494
Временная зона 512
Временная темпоральная логика ветвящегося времени 489
Временное действие автомата 507
Временной регион 470, 477, 512
Вычисления Зенона 463

Г

Грамматика TCTL 489
Граф переходов временного автомата 457
Граф регионов 485
◊ верификация 482
◊ временного автомата 472, 473

Д

Дедуктивная верификация 27
Диаграмма Хассе 386, 408
Дивергентные вычисления 463
Дополнение ω-языка 136
Достижимость 510

З

Задача о расстановке ферзей 332
Законы Де Моргана 99

И

Игра 15 277
◊ в кости 423
Инвариант состояния 447
Инструменты верификации
◊ свободно распространяемые 35
Интерливинг 180, 204
Интерпретация логической формулы 28
Итеративный алгоритм верификации 403

К

Китайские кольца 268
Конвергентные вычисления 463
Конечный автомат 123
Контрольный автомат 120
Корреляция событий 220
Криптографический протокол 279
◊ атака 279

Л

Линейная темпоральная логика 52
Логика
◊ CTL 75
◊ PCTL 437
◊ TCTLc 505
◊ ветвящегося времени 75, 91
◊ предикатов 43
Логический синтез — реализация схемы по BDD 340
Локальные часы 446, 450
Локальный инвариант 220

М

Марковская цепь 416
Матрица разностей границ 523
Метод model checking 28
Минимизация булевых функций 319
◊ неполностью определенных 323
Мир блоков 265
Множество согласованное 150
Модальная логика 46
Модальность 45
Модальный оператор 45
Монотонный оператор 390
Мягкий дедлайн 438

Н

Невременно́й язык 508
Неподвижная точка оператора 387, 396
Неполностью определенная булева функция 323

О

Обобщенный автомат Бюхи 138, 158
Обход конем шахматной доски 275
Ограничения на часы 478
Ограничения отношения на подмножестве 352
Оператор
◊ модальный 45
◊ монотонный 390
◊ прошлого 223
Операции
◊ квантификации по переменной 349
◊ над BDD 312
◊ над множествами 346
Отношение общего вида 355
Отсутствие
◊ взаимной блокировки 220
◊ прогресса 465

П

Пакет верификации 197
Параллельная композиция временных автоматов 454
Параллельные программы 18, 189
Параллельные системы
◊ свойства 215
Пересечение языков 125, 126, 136
Переход 172
Помеченная вероятностная структура 416
Построение
◊ BDD булевой функции 323
◊ ДНФ и КНФ по BDD 315
Правила
◊ выполнимости формул 155
◊ реализации обязательств 155
Предикат 51
Проблема 269, 274
◊ Subset-Sum 506
◊ булевой выполнимости 328
◊ взрыва числа состояний 369
◊ планирования 262
Проверка модели
◊ вероятностная 415
Проверка эквивалентности 27
◊ логических схем 344
Программирование в ограничениях 334
Протокол
◊ PAR 250
◊ W.C.Lynch 21
◊ ABP 250
◊ аутентификации Нидхэма — Шредера 280
◊ двухфазной фиксации 253
◊ с положительными подтверждениями и повторением передачи 177
◊ Фишера 484
Прямой образ бинарного отношения 351

Р

Разностная решающая диаграмма 525
Расширенная темпоральная логика ветвящегося времени 73
Реагирующая система 58
◊ вычисление 60
Региональная эквивалентность
◊ для нескольких локальных часов 477
Региональное отношение эквивалентности 469
Редукция частичных порядков 204
Редуцированная упорядоченная бинарная решающая диаграмма 308
Решетка 392

С

Свойство
◊ безопасности 222, 227
◊ достижимости 221
◊ живости 228
◊ праведливости 229
Семантически эквивалентные формулы 99

Семантическое дерево 298
Семафор 195
Сети Петри 530
Сигма-алгебра Колмогорова 419
Сильная справедливость 231
Символьная верификация 367, 369
Системы
◊ верификации 35
◊ реального времени 446
Слабая справедливость 231
Смена состояний 172
Согласованное множество 150
Состояние 172
◊ программы 375
ССК
◊ справедливая 238
Структура Крипке 68, 70, 72, 93, 128, 142, 171, 176, 263, 382
◊ вычисление 70
◊ модель потокола PAR 180
◊ переход 173
◊ представление булевыми функциями 370
◊ состояние 172
◊ справедливая 237
◊ траектория 71

Т

Темпоральная логика 42, 46
◊ линейного времени 51
◊ формальное определение 62
Темпоральное следствие 217
Темпоральные операторы прошлого 82
Темпоральный базис LTL 63
Теорема Тарского 390
◊ для конечного множества 392
◊ применение 393
Теория планирования
◊ классическая 262
Теория Райхенбаха 44
Тестирование 30
◊ преимущества и недостатки 36
Траектория
◊ временного автомата 507
◊ структуры Крипке 71
Трансформационная программа 58

У

Управляющие структуры языка Promela 200

Ф

Фактор-система переходов 467
Формальная
◊ верификация программ 25
◊ модель 25
◊ семантика TCTL 495
Формальная спецификация
◊ причинно-следственная связь событий 216
Формальное описание бизнес-процессов 259
Формула линейной темпоральной логики 62
Формулы
◊ CTL 118
◊ CTL* 74
◊ пути 73
Формулы LTL 142
◊ семантика 63
Формулы TCTL
◊ интерпретация 491
Формульные часы 497

Х

Характеристическая функция множества 345, 346
Хэндшейк 260

Ц

Цилиндр цепочки состояний 419

Ч

Частичная корректность 220
Чередующееся исполнение 180

Я

Язык
◊ Promela 198
◊ ω 120, 131