Министерство образования и науки Российской Федерации

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

**Приоритетный национальный проект «Образование»**
**Национальный исследовательский университет**

***Ю. Г. КАРПОВ  И. В. ШОШМИНА***

# ВЕРИФИКАЦИЯ РАСПРЕДЕЛЕННЫХ СИСТЕМ

Санкт-Петербург
Издательство Политехнического университета
2011

УДК 004.4
ББК 32.973.26 – 018.2
К26

Рецензенты:
Доктор технических наук, заслуженный деятель науки РФ,
заместитель директора Санкт-Петербургского института информатики
и автоматизации РАН по научной работе *Б.В. Соколов*
Доктор технических наук, профессор
Санкт-Петербургского политехнического
университета *С.М. Устинов*

*Карпов Ю. Г.* **Верификация распределенных систем**: учеб. пособие / Ю. Г. Карпов, И. В. Шошмина. — СПб. : Изд-во Политехн. ун-та, 2011. — 212 с.

ISBN

В пособии представлены научные результаты в области верификации с помощью метода model checking. В теоретической части пособия рассматрены проблема верификации, темпоральные логики, алгоритм model checking для LTL, структура Крипке как модель реагирующей системы. В практической части приведен обзор основных конструкций языка Promela системы верификации Spin, на примерах демонстрируется разработка моделей на языке Promela. В последнем разделе изложена курсовая работа по верификации нетривиальной системы логического управления с несколькими вариантами заданий.

Учебное пособие предназначено для студентов вузов, обучающихся по магистерской программе «Компьютерное моделирование и распределенные вычисления» по направлениям подготовки магистров «Информатика и вычислительная техника». Оно может быть также использовано при обучении студентов по направлению подготовки «Фундаментальная информатика и информационные технологии», а также при обучении в системах повышения квалификации, в учреждениях дополнительного профессионального образования.

Работа выполнена в рамках реализации программы развития национального исследовательского университета «Модернизация и развитие политехнического университета как университета нового типа, интегрирующего мультидисциплинарные научные исследования и надотраслевые технологии мирового уровня с целью повышения конкурентоспособности национальной экономики»

Печатается по решению редакционно-издательского совета
Санкт-Петербургского государственного политехнического университета.



**ISBN**

## ОГЛАВЛЕНИЕ

# ВВЕДЕНИЕ

Появление на рынке многоядерных процессоров поставило широкие массы программистов перед проблемой корректного написания параллельных программ. Умение выражать свойства поведения параллельных и распределенных систем, проверять выполнение этих свойств в разработанных программах становится необходимым фактически каждому программисту.

В учебном пособии представлены результаты в области формальной верификации с помощью метода проверки модели (model checking), предназначенного в первую очередь для доказательства корректности параллельных и распределенных систем. В начальных разделах излагается описание темпоральных логик, на которых основана спецификация свойств дискретных систем, изменяющихся во времени, а также алгоритмы проверки выполнения этих свойств у моделей систем. Во второй части пособия рассматриваются основы практического использования метода проверки модели для верификации параллельных и распределенных алгоритмов и систем с помощью пакета Spin. Завершающий раздел содержит пример курсовой работы по материалам пособия.

Model checking — это метод формальной проверки 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 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.

В разделе 1 приведены примеры ошибок в программных системах и описаны их последствия. Показано, что именно параллельные и распределенные программы сложны для их понимания и анализа, что формальная верификация является важнейшим средством выявления тонких ошибок в параллельных системах. Приведена общая схема верификации, и рассмотрен метод model checking как один из подходов

к формальной верификации. Проанализированы сильные и слабые стороны верификации и тестирования — двух классических методов повышения уровня доверия к разработкам. В разделе 2 введены темпоральные логики LTL и CTL*, рассмотрено их использование для спецификации свойств поведения реагирующих систем. Темпоральные логики — это расширения классических логик для формализации утверждений, истинность которых может изменяться со временем. Формулы темпоральных логик интерпретируются на системах переходов — так называемых структурах Крипке, которые являются формальными моделями дискретных систем. В разделе 3 излагается алгоритм проверки модели для тех случаев, когда проверяемое свойство системы выражено формулой логики LTL.

Интенсивные исследования в области проверки моделей привели к тому, что разработанная к настоящему времени техника применения этого подхода для верификации намного проще, чем его теоретический базис. Современные пакеты верификации, например, Spin, UPPAAL, SMV, STeP, KRONOS, PRISM, автоматизируют основные этапы model checking. В данном пособии рассматривается проверка правильности программ с помощью пакета Spin.

Пакет Spin позволяет:

- строить модели параллельных программ (протоколов, драйверов, систем логического контроля и управления) и широкого класса дискретных систем;
- выразить требуемые свойства поведения параллельных программ (так называемые «темпоральные свойства»);
- автоматически (с помощью «push button technique» — «техники нажатия кнопки») проверить выполнение темпоральных свойств параллельных систем на их моделях на основе формального подхода.

Spin — свободно распространяемый пакет программ для формальной верификации систем. Он разработан в исследовательском центре Bell Labs Джерардом Холзманном (Jerard Holzmann, http://spinroot.com/). Spin широко применяется как в обучении студентов методам верификации, так и в промышленности для формального анализа свойств разрабатываемых протоколов и бортовых систем. Ежегодно проводятся конференции пользователей этого пакета, в литературе широко обсуждаются многочисленные примеры применения Spin для верификации сложных систем. Международ-

ная ассоциация вычислительных машин (Association for Computing Machinery — ACM) присудила программному средству Spin престижную премию ACM Software System Award за 2001 г.

Spin может использоваться в двух режимах — как симулятор и как верификатор. При симуляции Spin выводит информацию об одной конкретной траектории выполнения построенной модели — графическое представление поведения в виде диаграмм последовательностей сообщений (Message Sequence Diagrams), возникающих при функционировании параллельных процессов по данной траектории. Графический интерфейс пользователя XSpin визуализирует запуски симуляций и упрощает процесс отладки модели.

Выполняя симуляцию и отладку систем, надо понимать, что никакое количество симуляций не может *доказать* свойства модели. Для доказательства свойств модели используется верификация — другой режим работы Spin. Верификатор, анализируя все возможные поведения модели, пытается найти контрпример — неправильную, ошибочную траекторию поведения, опровергающую заданное пользователем свойство. Для этого он строит сложную конструкцию — синхронное произведение модели переходов анализируемой системы и автомата Бюхи, задающего все возможные некорректные поведения. В случае обнаружения контрпримера симулятор Spin демонстрирует его пользователю в управляемом режиме симуляции. Таким образом, симуляция и верификация в Spin тесно связаны.

С помощью системы Spin выполняется проверка не самих программ, а их *моделей*. Для получения модели по исходной параллельной программе или алгоритму пользователь (обычно вручную) строит представление этой программы на C—подобном входном языке пакета Spin, названном автором Promela (Protocol Meta Language). Конструкции языка Promela просты, они имеют ясную и четкую семантику, позволяющую транслировать любую программу на этом языке в систему переходов с конечным числом состояний, которая представляет собой модель переходов подлежащей верификации параллельной программы или алгоритма.

Разработанные на языке Promela модели отличаются от верифицируемых программ, обычно написанных на языках программирования высокого уровня, например, Java или C. Программы на Promela не имеют классов. Они представляют собой плоскую структуру взаимодействующих параллельных процессов, имеют минимум

управляющих конструкций. Все переменные имеют конечную область определения. Поэтому такие программы можно считать моделью анализируемой системы — абстракцией, в которой пользователь должен отразить те аспекты и характеристики реальной проверяемой системы, которые значимы для заданных для верификации свойств. Построенная модель может содержать траектории, которые не относятся к реализации проверяемой системы в исходном языке программирования. Например, такие траектории могут включать в себя явную спецификацию возможных ситуаций при наихудшем поведении среды. С другой стороны, качество результата проверки моделей программ, представленных на Promela, полностью зависит от степени адекватности построенной модели.

Верифицируемая модель может строиться на этапе начальной разработки структуры системы (при создании прототипа). Окончательная же система, как правило, отличается существенной детализацией: произвольными объемами буферов, наличием вещественных переменных, более богатым спектром управляющих конструкций, динамическим порождением произвольного числа процессов и т. п. Достаточно трудно предложить средство автоматического перехода от реализации системы к ее модели в виде программы на языке Promela. Построение модели сложной системы на языке Promela, сохраняющей нетривиальные свойства этой системы, является искусством, которому нужно учиться. Одна из целей пособия, особенно курсовой работы, — продемонстрировать наиболее важные этапы моделирования и верификации параллельных и распределенных систем.

Рекомендуется во время чтения дисциплины снабжать студента конспектами лекций и задачами соответствующей тематики, например, из данного пособия. Самостоятельная работа студента повысит качество освоения дисциплины. При изучении практической части пособия студенту следует, установив систему Spin, строить примеры, описанные в пособии. Все приведенные здесь программы — рабочие, их нужно ввести в поле редактора, запустить и при анализе пытаться получить те же результаты, которые приведены в тексте.

Для понимания материала пособия необходимы начальные знания в области дискретной математики: логики высказываний и теории автоматов в объеме вводных университетских курсов, а также первоначальный опыт программирования.

# 1. ПРОБЛЕМА ВЕРИФИКАЦИИ

В этом разделе обосновывается необходимость верификации программ. Приводятся примеры тяжелых последствий программных ошибок в системах для критических применений, от которых все больше зависят использующие технику люди. Проведен сравнительный анализ верификации и тестирования, представлена общая схема верификации, и кратко охарактеризован тот конкретный подход к верификации, которому посвящено учебное пособие — 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, кто разработал 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., содержала около 3 млн строк кода, Microsoft Windows 98 — 18 млн, RedHat Linux 6.2 (2000) около 20 млн, RedHat Linux 7.1 (2001) 30 млн, Microsoft Windows XP (2002) — уже 40 млн строк. Мысленно охватить функционирование таких систем ни один человек не в состоянии даже при использовании современных технологий и методов абстрагирования и управления сложностью. Сложность разрабатываемого программного обеспечения подошла к границе их понимания и, следовательно, к границе их управляемости. Число потенциальных ошибок в программных системах постоянно растет. Например, в ОС Windows 95, по оценкам Microsoft, до сих пор остается несколько тысяч ошибок. Но эта ОС используется в бытовых персональных компьютерах, от нее не зависит жизнь людей. Только сумасшедший поставит эту ОС в бортовую систему управления самолета или даже автомашины.

Особенно подвержены ошибкам параллельные, распределенные и многопоточные программы, которые характерны для бортовых систем управления. Даже в тех случаях, когда функционирование каждой из параллельных взаимодействующих компонент 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». 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 Овицки и Лэмпорт еще 25 лет тому назад. 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 с серьезными последствиями ошибочного поведения программ проводится глубокий 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», она разработана и производится Европейским космическим агентством. Бортовая система управления ракеты «Ариан-5» использовала ПО предыдущей версии, «Ариан-4», но архитектура аппаратной части была модифицирована. После тщательного расследования было установлено, что авария произошла

из-за ошибки в навигационной программе бортового компьютера. Потери от аварии были огромными.

25 декабря 2004 г. после семилетнего полета космический зонд «Гюйгенс» отделился от автоматической международной межпланетной станции «Кассини» и начал спуск в атмосферу Титана. 15 января 2005 г. зонд впервые в истории космонавтики 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. Агентство Associated Press сообщило: «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.

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, и станция «Фобос-1» потерялась в космосе. Это несомненная ошибка бортовой системы управления, которая обязана выявлять некорректные внешние команды и должным образом реагировать на них. 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».

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 Intel все они были заменены. Потери компании составили сотни миллионов долларов.

26 февраля 2009 г. агенство Bloomberg сообщило: «Японское подразделение крупнейшего швейцарского банка UBS из-за компьютерной ошибки чуть было не потратило 3 триллиона иен (31 миллиард долларов) на выкуп облигаций компании Capcom. 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. Необходимость более тщательной проверки программ и аппаратных систем приводит к задержкам в выполнении проектов, к увеличению time-to-market — времени вывода продукта на рынок, а это грозит потерями из-за конкуренции и санкций по причине срыва сроков поставки.

Иногда ошибки в автоматических системах приводят не только к материальным потерям, но и к смерти людей.

В 1982 г. канадская компания Atomic Energy of Canada Ltd. запустила 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 два 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 Boeing-757 (Колумбия, рейс из Майами в Кали) погибли 159 человек. Расследование показало, что причиной катастрофы стала 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 февраля 2003 г. системой Patriot уничтожен американский F-16, пилот

погиб. В обоих случаях причиной аварии стали ошибки в бортовой системе автоматического обнаружения цели и наведения. Приведенные примеры показывают, что ошибки в сложных программных и аппаратных системах не являются чем-то исключительным, они регулярно повторяются в разработках сложных систем. Их непосредственными причинами являются и некорректные спецификации, и неправильное понимание спецификаций разработчиками, и взаимное непонимание в коллективе разработчиков, и непредвиденные события, режимы и условия работы, и несогласованность параллельных ветвей программ, и многое другое.

В настоящее время, когда все чаще человеческие жизни зависят от функционирования автоматических систем, проблема гарантии правильности их программных и аппаратных компонент 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 получают все большее распространение. В современном автомобиле — 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. Происходящий в настоящее время поворот программирования в сторону параллелизма некоторые ученые называют 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 последовательность возможных событий, как в последовательном программировании, а возможные комбинации частично упорядоченных событий параллельных процессов, 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 трудно выявляемых ошибок в параллельных программах приведем несколько простых программ.

**Пример 1.1.** В период обучения в вузе студенты часто работают. Студент А поступил на работу в компанию SoftTech и разработал бухгалтерскую программу БухСофт, которую его компания использует сама и продает. Студент открыл депозитный банковский счет ХА с начальным капиталом 0, на который бухгалтерская программа БухСофт1 компании SoftTech должна перевести ему 3 тыс. 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:: XA := XA + 3000; БухСофт2::  XA := XA + 15.
```

К концу месяца студент рассчитывает, что после двух выплат у него на счете будет сумма в 3015 долларов, достаточная 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 `XA := XA + 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 БухСофт1 выполняет все три свои операции либо до, либо после того, как процесс БухСофт2 выполнит все свои операции, то значение `XA` в результате увеличится на `C1 + C2` т. е.

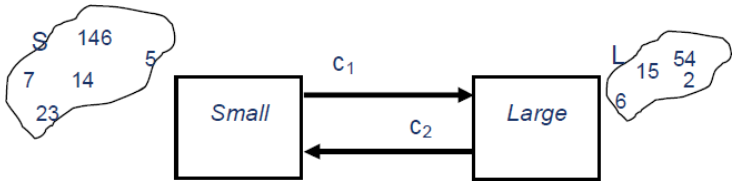


Рис. 1.1. Параллельная программа разделения множеств

программы работают 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 результате мы получим неожиданный результат: одно из изменений значения общей переменной, которое выполняет либо один, либо другой процесс, будет потеряно.

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

**Пример 1.2.** Программа разделения множеств. Эта простая программа состоит из двух параллельно функционирующих взаимодействующих процессов, `Small` и `Large`. Процесс `Small` оперирует на конечном множестве чисел `S`, процесс `Large` — на конечном множестве `L`. Процессы `Small` и `Large` взаимодействуют посылкой 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` — максимальные значения множеств (рис. 1.1).

Процесс `Small` на каждом шаге ищет максимальное значение во множестве `S` и посылает его процессу `Large` по каналу `c1`, затем принимает от `Large` найденное ею 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);
  while mx>x do {          /* 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);
  while mn<y do {     /* 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 для всех вариантов 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, однако, 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 для всех вариантов исходных данных она работает правильно. Только строгий формальный анализ, верификация программы может обнаружить ошибку.

**Пример 1.3.** Протокол W.C. Lynch. Это 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 — следующие данные.

Формализованное представление этих правил в стиле SDL представлено 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, то оно содержит `ack` в управляющей части. В 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.

Правила функционирования выглядят абсолютно корректно,

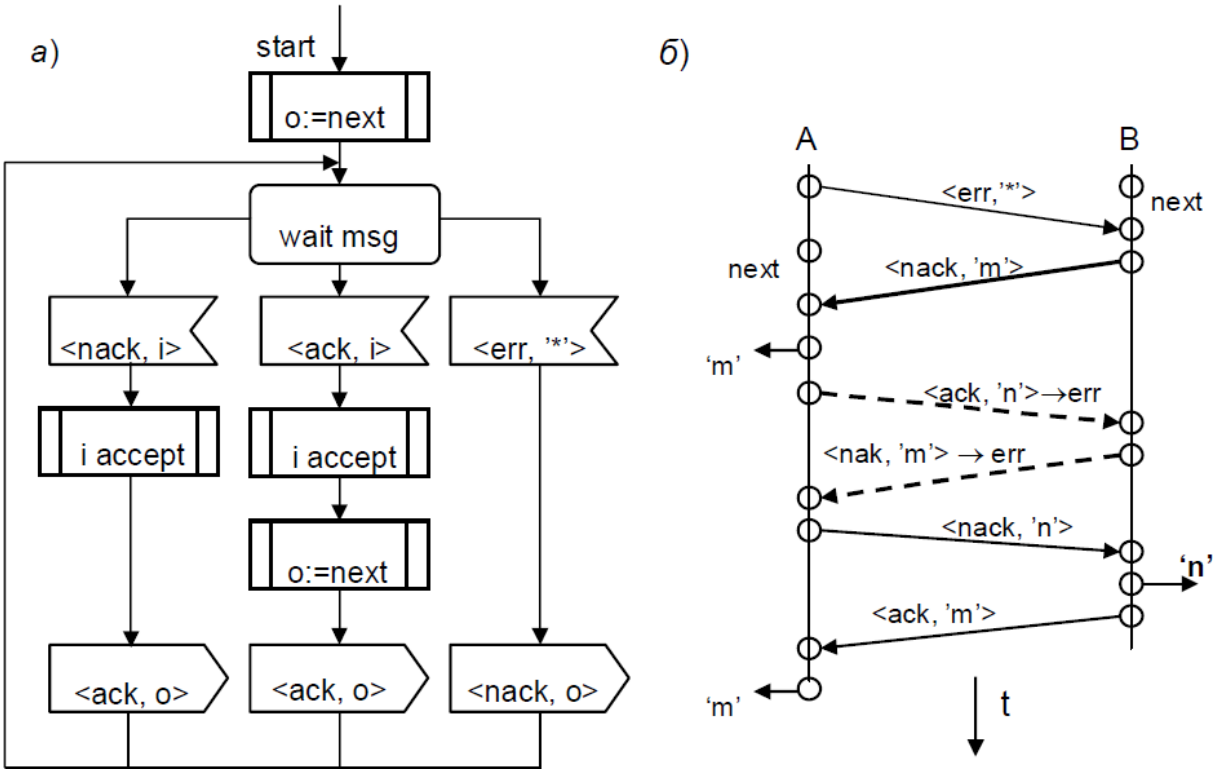


Рис. 1.2. Ошибка в протоколе

однако этот протокол содержит несколько ошибок. Во-первых, если нет данных для передачи в одном из направлений, то останавливается передача и в другом направлении. 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? Необходимы специальные правила инициализации и завершения передачи, которые сами могут оказаться непростыми и содержать ошибки.

В-третьих, протокол имеет серьезную принципиальную ошибку: при некоторых редко встречающихся сценариях он ошибочно принимает дубликат предыдущих данных как очередную информацию. Один из таких сценариев представлен 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 данном 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 квалифицированные разработчики следуют правилу: любая параллельная программа должна рассматриваться как некорректная до тех пор, пока не доказано обратное.

## 1.3. ОБЩАЯ СХЕМА ВЕРИФИКАЦИИ. ПРОВЕРКА МОДЕЛЕЙ

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

**Определение 1.1.** *Верификация* — подтверждение на основе представления объективных свидетельств того, что установленные требования были выполнены.

**Определение 1.2.** *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 рассматриваются как часть валидации.

В частности, если для анализа продукта используются 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 ее свойства, важные с точки зрения ее применения, также относится к процессу валидации. Сама же процедура проверки выполнения формальных требований на модели системы является процессом верификации.

В этом пособии рассматривается формальная верификация программных систем.

**Определение 1.3.** *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, для верификации анализируемая система (реализация) должна быть представлена 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 C. Формальная модель системы обычно строится вручную. Такая модель значительно проще самой проверяемой системы, это абстракция, в которой должны быть отражены наиболее существенные характеристики системы.

Спецификация системы также должна быть представлена фор-

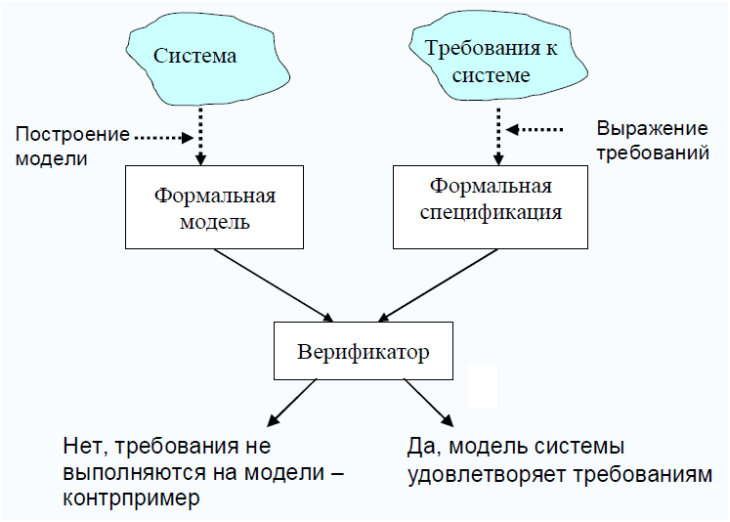


Рис. 1.3. Схема верификации

мально, набором формальных требований к ее функционированию. Но спецификации программы обычно не формальны. 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 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 бы достаточно выразительна, чтобы с ее помощью можно было представить поведение параллельных процессов и их 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 спектр важных свойств поведения системы.

Желательно, чтобы алгоритм верификации был эффективным, позволяющим автоматически выполнять проверку свойств систем, разрабатываемых для практического применения. Далее, если система удовлетворяет спецификации, верификатор должен выдать

подтверждение этому. Если система не удовлетворяет спецификации, то получение простого ответа «нет» малопродуктивно. Желательно, чтобы верификатор при этом выдавал 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. Верификации 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 и использующего интуицию при построении инвариантов и нетривиальном выборе альтернатив.

*Проверка эквивалентности* связана с разработкой формальных моделей взаимодействующих процессов, выражением в рамках таких формализмов как спецификации, так и реализации, и с проверкой эквивалентности поведений формально определенных моделей поведения. Это направление было начато разработкой Робином Милнером алгебры процессов Исчисление взаимодействующих систем 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 интересные 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 данном учебном пособии описывается метод 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* — 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, впервые этот метод был разработан для моделей систем переходов и так называемых «темпоральных» логик.

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. выполняться без участия человека, с использованием так называемой *технологии нажатия кнопки* (от *англ.* push-button technology).

Обсудим происхождение термина model checking.

В формальной логике *интерпретацией логической формулы* Ф называется объект, на котором формула принимает либо истинное, либо ложное значение. Те интерпретации формулы Ф, на которых Ф принимает истинное значение, называются *моделями логической формулы* Ф. Например, в логике высказываний множеством интерпретаций формулы $\Phi(p, q, r)$ являются наборы истинностных значений атомарных (элементарных) высказываний $p$, $q$, $r$, например $\langle p = true, q = false, r = true\rangle$. Те наборы истинностных значений $\langle p, q, r\rangle$, на которых Ф истинна, называются моделями формулы Ф.

Для темпоральной логики, используемой для спецификации

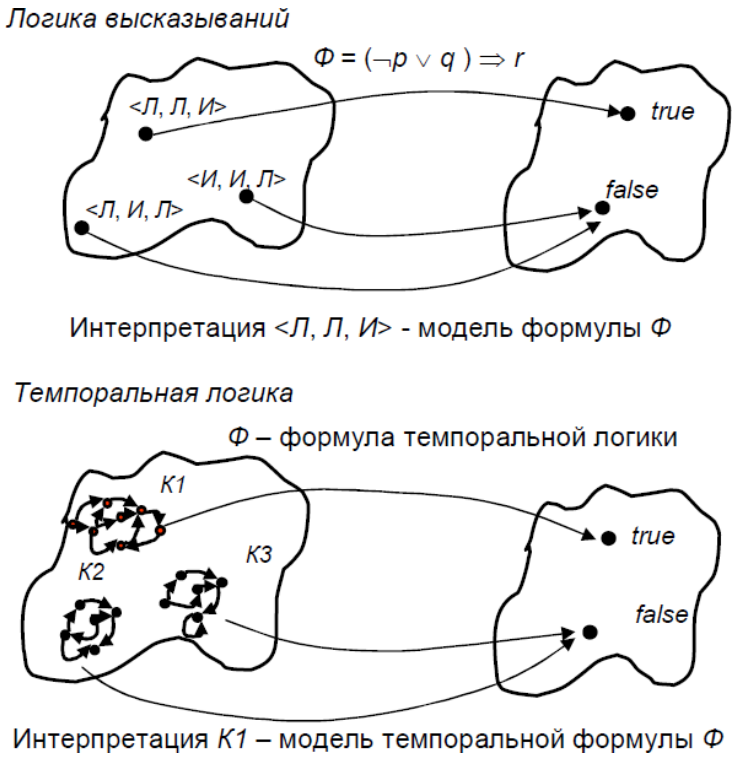


Рис. 1.4. Система переходов как модель темпоральной формулы

свойств поведения систем, множеством интерпретаций ее формул является множество систем переходов: на каждой системе переходов $M$ любая темпоральная формула Ф принимает либо истинное, либо ложное значение. Те системы переходов, для которых темпоральная формула Ф истинна, называются моделями формулы Ф.

**Определение 1.4.** *Model checking*. Это алгоритм, проверяющий, будет ли заданная логическая формула Ф истинна для данной системы переходов $M$, т. е. будет ли $M$ моделью Ф (см. рис. 1.4).

Поскольку система переходов является моделью верифицируемой системы (протокола, программы, схемы и т. п.), то часто термин model checking переводят 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 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 увеличивается экспоненциально с ростом числа компонент. Возникающая при этом проблема называется проблемой *взрыва числа состояний* (от *англ.* state explosion problem). То, что метод model checking в настоящее время успешно применяется при разработке реальных систем, является следствием создания эффективных методов представления данных (так называемых символьных методов) и соответствующих алгоритмов верификации, позволяющих в значительной степени решить проблему взрыва числа состояний и выполнять верификацию систем с громадным числом состояний ($10^{100}$ и более).

## 1.4. ТЕСТИРОВАНИЕ И ВЕРИФИКАЦИЯ

Широко распространенным методом проверки правильности систем является тестирование — проверка работы построенной системы в различных реальных ситуациях при разных 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 дорогостоящий процесс;
- 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 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» свидетельствует о том, что никаким количеством тестов нельзя установить правильность работы программной системы в любых условиях;
- тестированием невозможно выявить редко проявляющиеся ошибки, особенно в системах реального времени и 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 траекториях функционирования системы. Верификация проводится на самых ранних этапах проектирования по заранее разработанной модели, что существенно экономит время и затраты на исправление ошибок. Современные 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. Кроме того, модель может иметь свойства, которые в реальной системе не существуют. Таким образом, ценность результатов, полученных при верификации, напрямую зависит от адекватности модели и, следовательно, от уровня профессионализма человека, ее строящего;
- 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; автоматизированная система верификации может содержать ошибки;
- поскольку невозможно формально определить, что означает *полное отсутствие ошибок*, верификация так же, как и

тестирование, не может гарантировать абсолютной правильности системы;

- 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 их 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 [4], позволил не только обнаружить в этой программе ошибку, но и понять ее причину. Конечно, нельзя надеяться только на верификацию. Например, в работе [3] представлен анализ распределенного алгоритма обнаружения дедлоков в системе управления транзакциями распределенной базы данных, выполненный с помощью тестирования и имитационного моделирования. Этот алгоритм оказался ошибочным, хотя его корректность была доказана в работе [21]. Причина ошибок в доказательстве состояла в том, что и при разработке, и при доказательстве 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

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

В работе [11] 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.

С помощью системы верификации Spin были достигнуты значительные результаты в области верификации реальных систем. Среди них 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. В литературе приводятся примеры верификации систем с более чем 10100 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 описывается опыт использования корпоративного верификатора VeriSoft для систематической верификации библиотеки обработки вызовов протокола CDMA для беспроводных коммуникаций. В отчете делается вывод, что традиционное тестирование приносит мало пользы при разработке такого вида ПО, позволяя проверить только ничтожную долю поведений системы. В отличие от тестирования, с помощью верификации оказалосьо возможным систематически проанализировать полное пространство состояний системы, найти существенные ошибки разработки в нескольких версиях продукта. То, что ошибки были найдены на ранних стадиях разработки, позволило компании сэкономить значительные средства.

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 [20]. Был проведен формальный анализ с помощью системы верификации Spin многопроцессного ПО бортовой системы управления космическим аппаратом в проекте DeepSpace 1 для полета к Марсу. Обнаружено множество ошибок разработки, не выявленных методами, использующимися в стандартной технологии разработки ПО. По оценке разработчиков, четыре ошибки оказались критическими. Это были не примитивные описки или логические ошибки, характерные 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.

Множество компаний убедились в невозможности разработки качественных программ без использования верификации и включили этап верификации в технологический процесс разработки систем. Все основные производители программных и аппаратных систем: Intel, IBM, Nortel, Intel, Motorola, HP, Cadence, NEC, Bell Labs, Siemens, Lucent Technologies, Sun и многие другие имеют в своем составе группы, выполняющие верификацию проектов.

## 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).

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. 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.

## 1.7. БИБЛИОГРАФИЧЕСКИЙ КОММЕНТАРИЙ

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 [8]. Описание неполадок бортовой системы зонда «Гюйгенс» дано в [1]. Проблемы, возникшие на советских станциях «Фобос-1» и «Фобос-2», описаны в [2]. Информацию об ошибке в программе наведения ракеты Patriot можно найти в [7]. Некорректность параллельной программы разделения множеств доказана в [4], там же определены классы исходных множеств, на которых программа работает некорректно. Протокол W. C. Lynch, рассмотренный в этом разделе, приведен в [26], анализ его некорректностей дан в [23].

Впервые задача верификации программ была поставлена Р. Флойдом в [19]. Она формализована как проблема доказательства теорем на основании аксиом и правил вывода Хоаром в [22]. Метод model checking для верификации параллельных систем с конечным числом состояний был впервые предложен в 1981 г. Эдмундом Кларком, профессором Университета Карнеги-Меллона, и его аспирантом Алленом Эмерсоном. В работе [13] 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)». В этой же работе

был введен и термин model checking. Джозеф Сифакис, основатель и директор лаборатории VERIMAG, французского исследовательского центра по разработке встроенных систем, был руководителем научной группы, разработавшей первую систему автоматической верификации методом проверки модели [30]. Эти работы в немалой степени способствовали тому, что именно во Франции лучшая в мире авионика и технологии разработки бортовых информационных и управляющих систем.

На русском языке кроме перевода основополагающей монографии [16] нет книг и почти нет статей, дающих общее представление о предмете.

## ЗАДАЧИ

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

```
P1:: a1: A:=1;   P2:: a2: B:=A;
  b1: A:=2;           b2: C:=A+B;
  c1: A:=3;           c2: D:=A;
```

   Значение `A` вначале равно `0`. Рассмотрите все возможные варианты значений, которые могут иметь переменные `B`, `C` и `D` по завершении программы.
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`.
3. Протестируйте с помощью ручной прокрутки протокол W. C. Lynch при различных ситуациях в каналах передачи данных.

## ЗАКЛЮЧЕНИЕ К РАЗДЕЛУ 1

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

верификация анализирует *все варианты поведений абстрактной модели системы*.

Основным преимуществом тестирования является то, что тестируется реальная система в реальных (обычно критических) режимах ее работы. Основной недостаток тестирования — то, что могут быть проверены только некоторые из бесконечного числа режимов работы системы, и отсутствия ошибок в сложной системе никаким тестированием доказать нельзя. С помощью тестирования в принципе невозможно *доказать* наличие какого-либо свойства 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 и потому не могут быть проверены тестированием. Этот анализ проводится на ранних этапах разработки. Результатом верификации может стать 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 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 всей эффективности верификации она не заменяет тестирования готовой системы в типовых и критических режимах ее работы. 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.

# 2. ТЕМПОРАЛЬНЫЕ ЛОГИКИ

Для верификации автоматизированных систем свойства их поведения должны быть выражены формально логическими утверждениями, которые обеспечат их простую, лаконичную и недвусмысленную запись. В этом разделе показано, что обычная логика высказываний неадекватна для формулировки подобных утверждений о поведении технических систем, т. е. об их изменении 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. Формулы этих темпоральных логик являются удобным языком выражения свойств параллельных 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/неуверенности в истинности высказывания. Формулы логики могут принимать значения только *да* и *нет* на подходящей интерпретации, но не определяют возможные промежуточные значения. Формулы обычной логики истинны или ложны независимо от времени, в статическом мире.

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

Если утверждения естественного языка явно или неявно отражают зависимость высказываний от времени или от порядка событий

во времени, то их формализация в классической логике высказываний обычно является неадекватной.

Например, коммутативность операции конъюнкции ($A \wedge B \equiv B \wedge A$) не выполняется для следующих предложений:

- «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 обычной логике высказываний невозможен. Для адекватного формального выражения подобных свойств нужна логика, позволяющая отразить соотношения моментов времени наступления событий, в естественном языке определяемые такими словами, как *случилось после, случается иногда, случается всегда*. Это требует формализации высказываний, истинность которых меняется во времени.

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, например:

- «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 формальное доказательство истинности этого утверждения невозможно.

Большую долю утверждений, нуждающихся в формализации логической теорией с учетом времени, составляют свойства технических систем, обладающих динамикой, т. е. поведением во времени,

изменяющем некоторые параметры систем, например:

***C1:*** Посланный запрос когда-нибудь в будущем будет обработан;

***C2:*** 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. Мы не можем адекватно представить утверждение *C1* с помощью формулы логики высказываний:

$$\text{Послан}(R) \Rightarrow \text{Обработан}(R)$$(если запрос *R* послан, то он обработан).

Действительно, в классической логике утверждения обычно понимают как истинные либо ложные независимо от времени, а утверждение *C1* явно различает разные моменты времени. Оно утверждает, что если в некоторый момент времени $t$ запрос $r$ послан, то в какой-то будущий момент времени $(t' \geq t)$ 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 *C1* при этом будет выглядеть так:

$$(\forall t \geq 0)[\text{Послан}(R, t) \Rightarrow (\exists t' \geq t)\text{Обработан}(R, t')].$$

Утверждение *C2* в логике предикатов выглядит так:

$$(\forall t \geq 0)(\forall t' > t)[\text{Вызов}(n, t) \wedge \neg\text{Лифт}(n, t) \wedge (\exists t_1 : t' \geq t_1 \geq t)H(n, t_1) \Rightarrow (\exists t_2 : t' \geq t_2 \geq t)\text{Обслуживается}(n, t_2)]$$

(здесь $\text{Лифт}(n, t)$ — утверждение: *Лифт в момент t находится на этаже n*).

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 Г. Райхенбахом 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 два момента

времени: момент $S$, в который сделано высказывание (*Speech time*), и момент $E$ 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 $S < E$. Для того чтобы охватить большее число различных глагольных времен, Райхенбах ввел третий момент, а именно, момент точки референции $R$ (*Reference time*), т. е. момент, на который ссылается высказывание. Высказывание в Past Perfect, когда мы говорим, например, «I shall have seen John» (буквально: «Я буду иметь Джона увиденным»), отсылает нас не к тому моменту, когда я увидел Джона, а к моменту, по отношению к которому (*with reference to*) мое видение Джона уже произошло, т. е. соотношение между этими моментами времени есть $S < E < R$.

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* — $E < R = S$ (рис. 2.1).

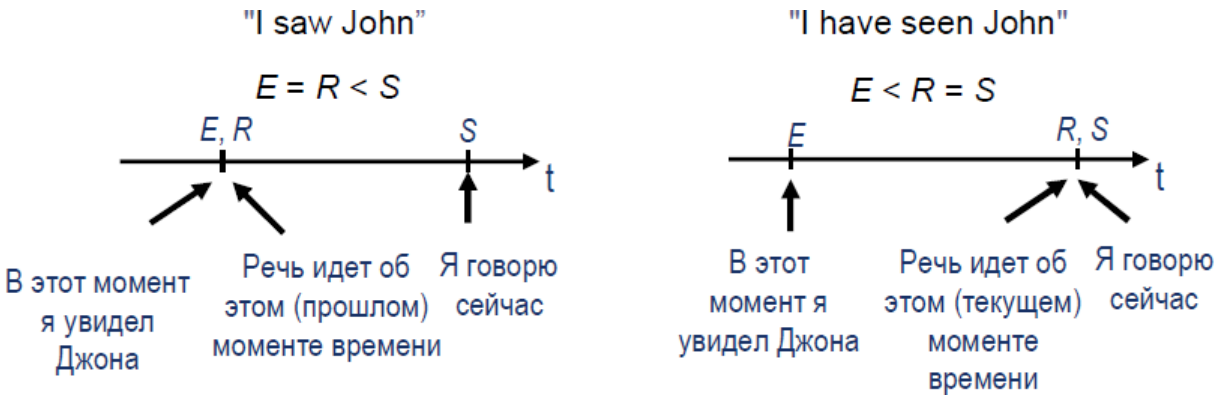


Рис. 2.1. Различия в формах глаголов Past Indefinite и Present Perfect

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

## 2.2. МОДАЛЬНЫЕ И ВРЕМЕННЫЕ ЛОГИКИ

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 знаменитый тезис Лэмпорта: «Время есть способ упорядочения событий»). Характеристики временных свойств систем используют категории вида *никогда не будет верно, что ...*, или *в будущем обязательно случится, что ...* и другие, характеризующие истинность суждения с учетом отношения между моментами наступления различных событий во времени. Можно расширить классическую логику, разрешив использовать такие категории перед утверждениями логики, например, выражение «$\mathcal{F}Получено(m)$» можно понимать так: «Когда-нибудь в будущем сообщение $m$ обязательно будет получено». Такие категории называются модальностями. Они расположены перед высказываниями и обозначают отношение содержания этих высказываний к действительности.

*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 необязательно связан со временем. Например, пусть $q$ — некоторое высказывание. Можно выразить неполную уверенность в истинности утверждения $q$, сказав: «Возможно, что $q$ наступит». Для формализации этого введем модальный оператор $\mathcal{M}$. Тогда $\mathcal{M}q$ имеет смысл «возможно, что $q$ наступит», или «может быть, наступит $q$».

Формальные теории устанавливают соотношения между формулами. Например, введем модальный оператор $\mathcal{L}$, имеющий смысл *необходимо, что ...* Тогда можно отношение между модальностями $\mathcal{M}$ и $\mathcal{L}$ определить следующим образом: $\mathcal{L}q \equiv \neg\mathcal{M}\neg q$. Пусть $q$ означает атомарное высказывание «писатель пишет». Тогда соотношение $\mathcal{L}q \equiv \neg\mathcal{M}\neg q$ согласуется с интуитивно истинным утверждением: «*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 по-разному, вводя разного типа модальности и даже разную их семантику (формальное определение смысла модальностей). Возможность использования различной семантики проистекает из того, что в естественном языке в разных его применениях трактовки одних и тех же модальных слов 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.

**Определение 2.1.** *Темпоральные логики*. Это логики, в которых истинностное значение логических формул зависит от момента времени, в котором вычисляются значения этих формул.

Темпоральные логики издавна 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.

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

На рис. 2.2 представлено, при каких значениях времени истинны утверждения $\mathcal{F}q$ и $\mathcal{P}q$, если заданы отрезки времени, в которых истинно утверждение $q$. Обозначим $t \models \Phi$ утверждение «В момент времени $t$ истинно утверждение Ф», тогда (см. рис. 2.2):

- утверждение $q$ истинно в момент времени $t$ ($t \models q$), если утверждение $q$ истинно в момент $t$ (что является, конечно, тавтологией);
- утверждение $\mathcal{F}q$ истинно в момент времени $t$ ($t \models \mathcal{F}q$), если для какого-то момента времени $t'$ в будущем (при некотором $t' \geq t$) $q$ станет истинным (заметьте, что будущее здесь включает настоящее);
- утверждение $\mathcal{P}q$ истинно в момент времени $t$ ($t \models \mathcal{P}q$), если для какого-то момента времени $t'$ в прошлом (при некотором $t' < t$) утверждение $q$ было истинным.

В Tense Logic введены и два дуальных темпоральных оператора: $\mathcal{G}$ (от англ. Globally) и $\mathcal{H}$ (от англ. History), с соотношениями: $\mathcal{G}q \equiv \neg\mathcal{F}\neg q$, $\mathcal{H}q \equiv \neg\mathcal{P}q$. Справедливость этих соотношений очевидна,

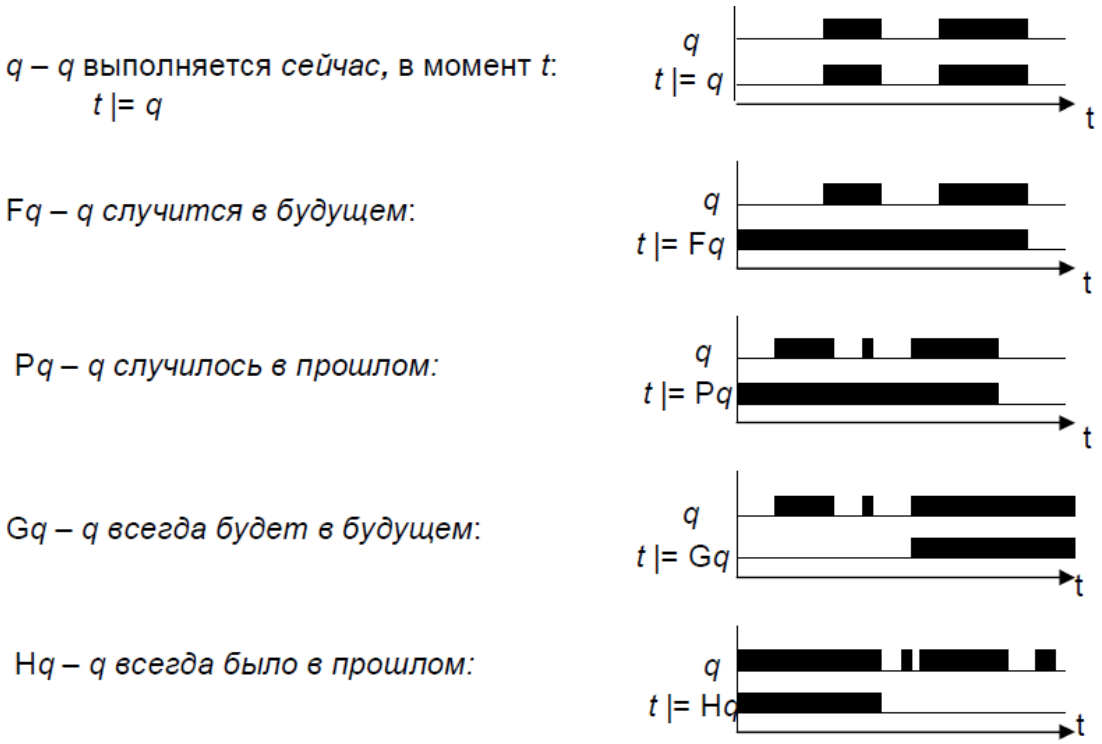


Рис. 2.2. Темпоральные операторы Tense Logic

например, первого: «Утверждать, что в будущем утверждение $q$ будет всегда истинно, это то же самое, что утверждать, что неверно, что когда-нибудь в будущем утверждение $q$ станет ложным». Эти операторы можно определить и независимо от $\mathcal{F}$ и $\mathcal{P}$:

- утверждение $\mathcal{G}q$ истинно в момент времени $t$ ($t \models \mathcal{G}q$), если для любого момента времени $t'$ в будущем (при всех $t' \geq t$) утверждение $q$ истинно (заметьте, что будущее здесь также включает в себя настоящее);
- утверждение $\mathcal{H}q$ истинно в момент времени $t$ ($t \models \mathcal{H}q$), если для любого момента времени $t'$ в прошлом утверждение $q$ истинно.

Уже с помощью двух темпоральных операторов $\mathcal{F}$ и $\mathcal{G}$ можно выразить сложные свойства, зависящие от времени. Утверждение $q$ будет истинным:

- всегда в будущем — $\mathcal{G}q$;
- хотя бы раз в будущем — $\mathcal{F}q$;
- никогда в будущем — $\mathcal{F}q$;
- бесконечно много раз в будущем — $\mathcal{GF}q$;
- с какого-то момента постоянно — $\mathcal{FG}q$.

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

*События* $e1$ *и* $e2$ *никогда не произойдут одновременно:*

$$\mathcal{G}[\neg(e1 \wedge e2)].$$

*Посланное сообщение когда-нибудь в будущем будет получено:*

$$\mathcal{G}(\text{Послано}(m) \Rightarrow \mathcal{F}\text{Получено}(m)).$$

*Джейн вышла замуж и родила ребенка:*

$$\mathcal{P}(\text{Джейн\_выходит\_замуж} \wedge \mathcal{F}\text{Джейн\_рожает\_ребенка}).$$

*Джейн родила ребенка и вышла замуж:*

$$\mathcal{P}(\text{Джейн\_рожает\_ребенка} \wedge \mathcal{F}\text{Джейн\_выходит\_замуж}).$$

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

$$\mathcal{G}(\neg\mathcal{P}\text{Зажигание} \Rightarrow \text{Старт}).$$

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

Оператор Next time: утверждение $\mathcal{X}q$ истинно в момент времени $t$, если $q$ истинно в следующий момент $t+1$:

*Джон убил, и ему стало страшно:*

$$\mathcal{P}(\text{Джон\_убивает} \wedge \mathcal{X}\mathcal{G}\text{Джону\_страшно}).$$

*Если я видел ее раньше, я ее узнаю при встрече:*

$$\mathcal{P}\text{Увидел} \Rightarrow \mathcal{G}(\text{Встретил} \Rightarrow \mathcal{X}\text{Узнал}).$$

*Джон умер, и его похоронили:*

$$\mathcal{P}(\text{Джон\_умирает} \wedge \mathcal{X}\mathcal{F}\text{Джона\_хоронят}).$$

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

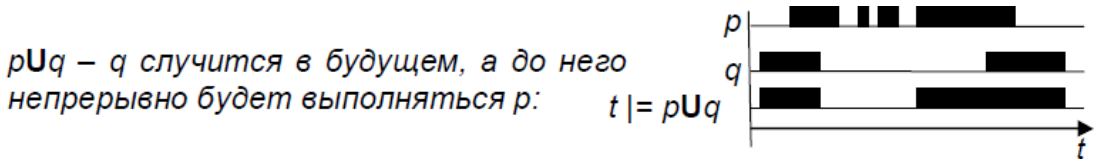


Рис. 2.3. Темпоральный оператор Until

С помощью оператора Until многие сложные утверждения легко перевести в формальную запись, например:

*Мы не друзья, пока ты не извинишься:*

$$\neg(\text{Мы\_друзья})\,\mathcal{U}\,\text{Ты\_извиняешься}.$$

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

$$\mathcal{G}[\text{Вызов}(n) \Rightarrow \neg(\text{Лифт}(n))\,\mathcal{U}\,\text{Обслуживается}(n)].$$

Через оператор $\mathcal{U}$ легко выражается оператор $\mathcal{F}$:

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

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

$$\mathcal{G} = \neg\mathcal{F}\neg q = \neg(true\,\mathcal{U}\,\neg q).$$

В этой логике для операторов $\mathcal{X}$ и $\mathcal{U}$ существуют их аналоги в прошлом: $\mathcal{X}^{-1}$ (*в предыдущий момент времени, вчера*) и $\mathcal{S}$ (от англ. 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 (тавтология): *Будет, что нечто было, если и только если оно или есть сейчас, или будет, или уже было* можно формализовать так:

$$\mathcal{FP}q \equiv q \vee \mathcal{F}q \vee \mathcal{P}q.$$

Философская мысль о времени: «Любое *вчера* было когда-то *завтра*, любое *завтра* когда-нибудь станет *вчера*» формально запишется так:

$$(\mathcal{P}\mathcal{X}^{-1}q \Rightarrow \mathcal{P}\mathcal{X}q) \wedge (\mathcal{F}\mathcal{X}q \Rightarrow \mathcal{F}\mathcal{X}^{-1}q).$$

Тождественно истинной формулой $q \equiv \mathcal{X}^{-1}\mathcal{X}q$ 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. $\text{Д}(NN) \wedge \mathcal{FX}\text{П}(NN)$ : *сейчас NN — мой друг, а в будущем он станет президентом*;
2. $\mathcal{FX}(\text{Д}(NN) \wedge \text{П}(NN))$ : *в будущем NN станет моим другом, и в это время он уже будет президентом*;
3. $\mathcal{F}(\text{Д}(NN) \wedge \mathcal{FX}\text{П}(NN))$ : *в будущем NN станет моим другом, а затем он станет президентом*.

## 2.3. ТЕМПОРАЛЬНАЯ ЛОГИКА ЛИНЕЙНОГО ВРЕМЕНИ

Современная *темпоральная логика линейного времени (Linear Time Logic, LTL)* является наследницей временной логики Tense Logic Артура Прайора. LTL разработана израильским ученым Амиром Пнуэли для спецификации свойств параллельных технических систем. В LTL максимально упрощены все включенные в нее концепции.

Во-первых, в LTL темпоральными операторами расширена простейшая логика высказываний, формулы которой строятся из конечного числа атомарных предикатов и булевых операций над ними. Предикат — это высказывание, принимающее значение либо «истина», либо «ложь» в зависимости от значений 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 высказываний, а не то, как они построены, или как определяется их истинностное значение.

**Определение 2.2.** *Атомарный предикат* (атомарное утверждение, от англ. atomic predicate). Это утверждение, принимающее истинное или ложное значение, 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, $\mathcal{U}$ и $\mathcal{X}$. Логика LTL не включает в себя темпоральных операторов прошлого. Основания для этого очевидны: прошлое для технических систем менее важно, чем будущее поведение, начинающееся с момента их включения, запуска.

В качестве интерпретации формул темпоральной логики рассматривается дискретная во времени, бесконечная, 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 считал, 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.

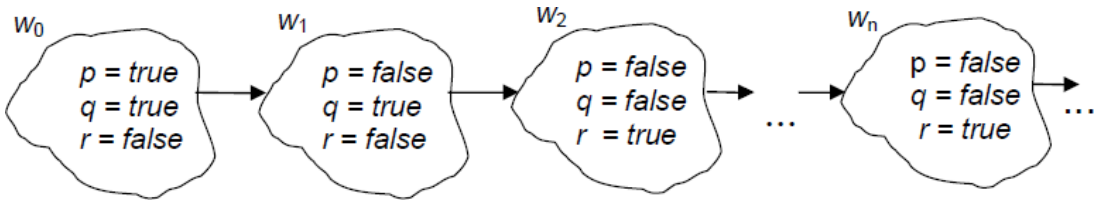


Рис. 2.4. Возможная интерпретация формул темпоральной логики LTL — бесконечная последовательность миров

Линейная темпоральная логика является расширением обычной логики высказываний для рассуждений о бесконечной последовательности миров (например, дней или лет нашей жизни, как у М. Ю. 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 родит ребенка. Направленность последовательности миров от прошлого к будущему позволяет прово-

дить рассуждения об относительном времени, в терминах *до* и *после*. На рис. 2.4 дан пример такой интерпретации. На последовательности миров $w_0, w_1, w_2, \ldots, w_n, \ldots$ задано множество атомарных предикатов $\{p, q, r\}$. В мире $w_0$ атомарные предикаты $p$ и $q$ истинны, а $r$ — ложен, в мире $w_1$ утверждения $q$ и $r$ истинны, а $p$ — ложно, и т. п. Иными словами, каждый мир не похож на другой.

Будем считать, что, начиная с $w_2$, все истинностные значения $p$, $q$ и $r$ во всех мирах бесконечной последовательности рис. 2.4 одинаковы и совпадают с их значениями в $w_2$. Как происходит переключение от одного мира к следующему, теория не рассматривает, от этого она абстрагируется.

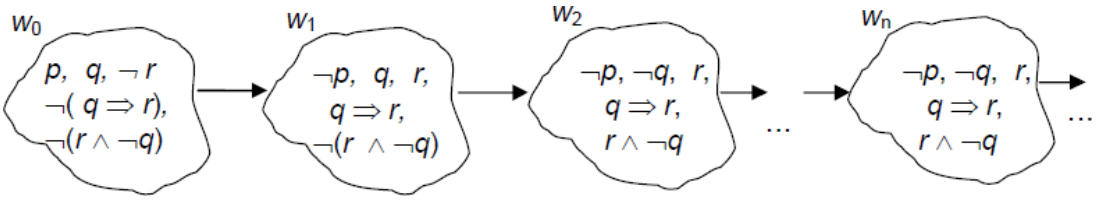


Рис. 2.5. Истинность логических формул в мирах

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 принимать различные истинностные значения, вычисленные по значениям своих аргументов. Например, в мире $w_0$ формула $q \Rightarrow r$ ложна, а во всех следующих мирах данной цепочки она истинна. Формула $r \wedge \neg q$ в мире $w_0$ ложна, а в $w_2$ — истинна. На рис. 2.5 представлено, в каких мирах упомянутые формулы выполняются, а в каких — нет (они записаны со знаком отрицания $\neg$).

На конкретной цепочке миров можно в принципе подсчитать и истинность формул с темпоральными операторами(см. рис. 2.6). Например, формула $r \wedge \neg q$ истинна, начиная с $w_2$, поэтому в $w_0$ истинна формула $q\,\mathcal{U}\,(r \wedge \neg q)$: действительно, существует такое $i \geq 0$ (равное 2), что до $w_i$ истинно $q$, а начиная с $w_i$, истинно $r \wedge \neg q$.

**Определение 2.3.** *Формула* $\Phi$ *линейной темпоральной логики* выполняется на последовательности миров $w_0, w_1, w_2, \ldots$, если $\Phi$ истинна в начальном мире $w_0$ этой последовательности.

Такая привязка к начальному моменту времени важна для технических систем: обычно важно знать, будет ли некоторое свойство

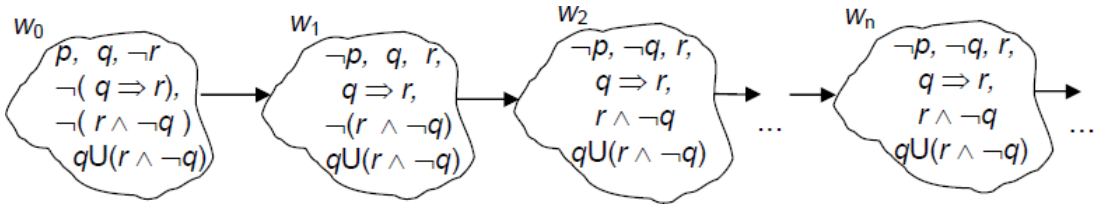


Рис. 2.6. Темпоральные формулы на цепочке миров

будущего поведения выполняться в системе, стартующей из *начального* состояния. Это означает, что для приведенной цепочки миров выполняются формулы $p, q, r, (q \Rightarrow r), (r \wedge \neg q), q\mathcal{U}(r \wedge \neg q)$ и т.д. — именно они истинны в $w_0$.

Формулы LTL построены из атомарных предикатов с помощью булевых операций и темпоральных операторов. Значения истинности обычных булевых формул в каждом конкретном мире определяется по значениям истинности, которые имеют атомарные предикаты в этом мире. Истинностные значения темпоральных операторов в каждом мире определяются по значениям истинности их аргументов в цепочке миров, начинающейся с данного конкретного мира.

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

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

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

**Пример 2.1.** Рассмотрим несколько примеров формализации высказываний естественного языка. Такую формализацию часто можно выполнить по-разному, поскольку высказывания естественного языка не всегда однозначны.

1. 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:

$$\mathcal{G}(\text{Я\_жив} \Rightarrow \text{Я\_надеюсь}).$$

   Сейчас и всегда в будущем, если я буду жив, я буду надеяться.
   Подобно этому, утверждение *Пока сердце бьется, я буду бороться!* формализуется так:

$$\mathcal{G}(\text{Сердце\_бьется} \Rightarrow \text{Я\_борюсь}).$$

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

$$\text{Мы\_боремся}\ \mathcal{U}\ \text{Мы\_победили}.$$

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

$$(\text{Мы\_боремся}\ \mathcal{U}\ \text{Мы\_победили}) \vee \mathcal{G}\text{Мы\_боремся}.$$

   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 $\mathcal{W}$, так называемый слабый Until, имеющий именно эту семантику:

$$p\mathcal{W}q \equiv (p\,\mathcal{U}\,q) \vee \mathcal{G}q.$$

С использованием слабого Until утверждение «Мы будем бороться, пока не победим» формализуется так:

$$\text{Мы\_боремся}\ \mathcal{W}\ \text{Мы\_победили}.$$

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.

3. Англоязычные документы часто снабжаются припиской: *Not valued until signed — (Не действителен до тех пор, пока не подписан)*. Эту фразу точно передает формула

$$\text{Документ\_не\_действителен}\ \mathcal{W}\ \text{Документ\_подписан}.$$

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), то эту фразу можно формально представить оператором Until:

$$\text{Документ\_не\_действителен}\ \mathcal{U}\ \text{Документ\_подписан}.$$

4. Если обозначить $p$ утверждение *Я твоя!*, то $\mathcal{G}p$ означает: *Я твоя навеки!*
5. Утверждение *Мы придем к победе коммунистического труда!* формально запишется так:

$$\mathcal{F}\text{Коммунистический\_труд\_победил!}$$

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

$$\text{Он\_играет\_джаз} \Rightarrow \mathcal{X}\text{Он\_продает\_Родину}.$$

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

$$\mathcal{G}(\text{Он\_играет\_джаз} \Rightarrow \mathcal{X}\text{Он\_продает\_Родину}).$$

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

Однако, возможно, поэт хотел данной 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):

$$\mathcal{G}(\text{Он\_играет\_джаз} \Rightarrow \mathcal{X}\mathcal{F}\text{Он\_продает\_Родину}).$$

7. Пусть $p$ означает *Я люблю Машу*, а $q$ — *Я люблю Дашу*. Тогда:
   $\mathcal{F}p$ — «Я когда-нибудь обязательно полюблю Машу»;
   $\mathcal{G}q$ — «Я люблю Дашу и буду любить ее всегда»;
   $\mathcal{G}(\neg p \vee \neg q)$ — «Я никогда не буду любить одновременно и Машу, и Дашу»;
   $q\,\mathcal{U}\,p$ — «В будущем я полюблю Машу, а до той поры я буду любить Дашу»;
   $\mathcal{F}\mathcal{G}p$ — «Когда-нибудь я полюблю Машу навечно»;
   $\mathcal{G}\mathcal{F}q$ — «Я буду бесконечно влюбляться в Дашу».
8. Рекламный слоган *Раз Персил — всегда Персил* не совсем ясен простому человеку. Что значит *Раз Персил*?
   По-видимому, рекламодатели хотели сказать, что если только кто-нибудь попробует «Персил», то с этого момента он уже не сможет от него отказаться и будет им пользоваться всегда. Но тогда абсолютно точно смысл данного слогана выражает следующая формула логики LTL:

$$\mathcal{G}(\text{Персил} \Rightarrow \mathcal{G}\text{Персил}).$$

   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 элементарных утверждений, принимающих только два значения *истина* или *ложь* в каждом из миров. Атомарные утверждения не включают в себя время, в каждом мире их истинностное значение статично, неизменно. Но в другом мире эти значения могут быть другими.

## 2.4. РЕАГИРУЮЩИЕ СИСТЕМЫ

**Определение 2.4.** *Реагирующие системы* (от англ. reactive systems). Это класс информационных систем, основной функцией которых является не преобразование информации, а поддержание взаимодействия с окружением.

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

Реагирующие системы отличаются от программ преобразования данных. Такие программы (их часто называют *трансформационными программами*) обычно принимают на вход исходные данные, производят вычисления и выдают результат на выход, после чего работа программы завершается. Программы обращения матрицы, поиска экстремума функции, решения дифференциального уравнения численным методом — все это трансформационные программы.

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

Реагирующие системы функционируют бесконечно, они контролируют окружение, реагируя на внешние события получение сообщения, щелчок клавишей 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 больших систем, с которыми они взаимодействуют. Операционные системы, протоколы коммуникации, планировщики, драйверы, контроллеры, параллельные взаимодействующие программы, системы логического управления — все это примеры реагирующих систем.

Введенные выше последовательности миров можно считать моделью 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.

**Определение 2.5.** *Вычисление реагирующей системы* — это бесконечная последовательность состояний, которые система проходит во времени. *Поведение реагирующей системы* — это все возможные варианты ее вычислений.

Таблица 2.1

**Сравнение трансформационных и реагирующих программных систем**

| **Характеристика** | **Трансформационные программные системы** | **Реагирующие программные системы** |
|---|---|---|
| **Примеры** | Обработка данных, численные методы, вычисление функции, распознавание образов | Протоколы, драйверы, системы логического управления, ОС, ... |
| **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)** | Например, в виде логических пред- и постусловий — $\{x \geq 0\}S\{y = x^3\}$: *если до запуска программы $S$ переменная $x$ не меньше 0, то после завершения программы $S$ переменная $y$ будет иметь значение $x^3$.* | Например, в виде формул темпоральной логики — $\mathcal{G}(p \Rightarrow \mathcal{F}q)$: *для всех состояний программы справедливо следующее: если выполнилось утверждение $p$, то потом, в каком-то следующем состоянии выполнится $q$* |

Обозначим вычисления греческими буквами $\pi$, $\sigma$ и т. п., например: $\sigma = s_0 s_1 s_2 s_3 s_4 s_5 \ldots$. В эти состояния система переходит в дискретные моменты времени $t_0, t_1, t_2, \ldots$. Начальное состояние вычисления часто помечается входящей стрелкой (рис. 2.7).

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 указывать не будем. По значениям этих элементарных неделимых утверждений можно строить как булевы формулы, характеризующие конкретное состояние, так и темпоральные формулы, характеризующие целое вычисление, начинающееся в данном состоянии.

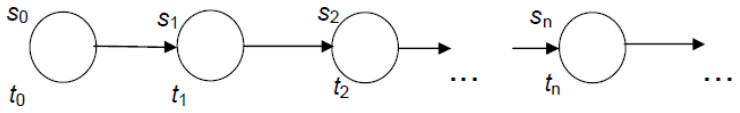


Рис. 2.7. Бесконечная цепочка миров как вычисление дискретной системы

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

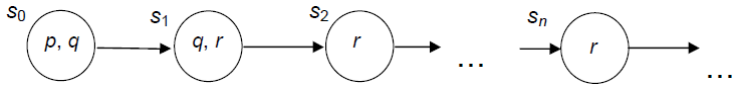


Рис. 2.8. Атомарные предикаты в состояниях вычисления

На рис. 2.9 формула $q \mathcal{U} p$ выполняется на всем представленном там вычислении, а формула $\mathcal{G} r$ на нем не выполняется. Формула $\mathcal{G} r$ 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.

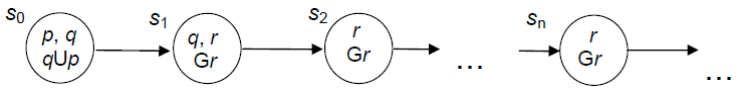


Рис. 2.9. Темпоральные формулы на последовательности состояний

В качестве атомарных предикатов в реагирующих системах используются любые утверждения, имеющие 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 находится дискретная система (например, метка оператора), также может быть атомарным предикатом. Атомарный предикат, принимающий истинное значение, когда модуль $M$ находится в состоянии $s$, запишем так: $M@s$, или, если имя модуля очевидно, то $@s$, или же просто $s$. Например: *принтер занят (находится в состоянии или режиме занятости)* $Pr@busy$, *процесс* $P$ *находится в своем критическом интервале* $P@crint$ и т. п.

Темпоральная логика оказалось очень эффективной для спецификации требований к поведению реагирующих систем, особенно для параллельных и распределенных систем, в которых именно упорядоченность событий выражает свойства их корректного поведения. Например, требование взаимного исключения параллельных процессов формулируется просто: $G\neg(@crint1@crint2)$ — *«два процесса никогда одновременно не будут находиться в своих критических интервалах»*.

Реагирующие системы обычно строятся из нескольких компонент, взаимодействующих друг с другом, и проверка свойств таких систем весьма важна: как было указано ранее, 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: можно ли гарантировать, что запрос сервиса никогда не останется без ответа, всегда ли

посланные сообщения будут доставляться коммуникационным протоколом в том же порядке, без пропусков и дублирования, может ли в распределенной системе комментарий на новость прийти раньше, чем сама новость, и т. п.

## 2.5. ФОРМАЛЬНОЕ ОПРЕДЕЛЕНИЕ ЛИНЕЙНОЙ ТЕМПОРАЛЬНОЙ ЛОГИКИ

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*. Это:

- атомарное утверждение (атомарный предикат) $p, q, \ldots$;
- формулы LTL, связанные логическими операторами $\neg, \vee$;
- формулы LTL, связанные темпоральными операторами $\mathcal{X}$, $\mathcal{U}$.

Прошлое в логике LTL не рассматривается.

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

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

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 $\mathcal{X}$ (neXt time) и бинарный $\mathcal{U}$ (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 другие логические операторы используются для сокращения записи формул:

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

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

- $\mathcal{F}\varphi$ — сокращение для $true \mathcal{U} \varphi$;
- $\mathcal{G}\varphi$ — сокращение для $\neg\mathcal{F}\neg\varphi$

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 $\varphi$ логики LTL на конкретном вычислении $\sigma = s_0 s_1 s_2 s_3 \ldots$ может быть либо истинной, либо ложной.

Обозначим $\sigma(i)$ состояние $s_i$ вычисления $\sigma$, а $\sigma^i$ — $i$-й суффикс цепочки $\sigma$, т. е. $\sigma^i = s_i s_{i+1} s_{i+2} s_{i+3} \ldots$. Тогда $s_i \models \varphi$ обозначает утверждение *В состоянии $s_i$ выполняется (истинна) формула $\varphi$*, а $\sigma \models \varphi$ — утверждение *На вычислении $\sigma$ выполняется формула $\varphi$*. Таким образом, 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 $\varphi$ логики LTL на вычислении $\sigma$ (обозначается $\sigma \models \varphi$) определяется индуктивно по структуре этой формулы:

- $\sigma \models p$, если атомарный предикат $p$ истинен в начальном состоянии вычисления $\sigma$;
- $\sigma \models \neg\varphi \equiv \sigma \not\models \varphi$, если формула $\varphi$ не выполняется на $\sigma$;
- $\sigma \models \varphi_1 \vee \varphi_2 \equiv \sigma \models \varphi_1 \vee \sigma \models \varphi_2$, если на $\sigma$ выполняется или формула $\varphi_1$, или формула $\varphi_2$;
- $\sigma \models \mathcal{X}\varphi \equiv \sigma^1 \models \varphi$, если $\varphi$ выполняется на вычислении $\sigma^1 = s_1 s_2 s_3 s_4 \ldots$;
- $\sigma \models \varphi_1 \,\mathcal{U}\, \varphi_2 \equiv (\exists k \geq 0) : (\sigma^k \models \varphi_2 \wedge (\forall j : 0 \leq j < k) : \sigma^j \models \varphi_1)$, если когда-то в будущем состоянии вычисления $\sigma$ (включая настоящее) станет справедливой формула $\varphi_2$, а до этого во всех состояниях вычисления $\sigma$ будет выполняться $\varphi_1$.

Еще раз отметим, что определение оператора Until рефлексивно в том смысле, что настоящее является частью будущего. То же относится и к выводимым темпоральным операторам $\mathcal{F}$ и $\mathcal{G}$:

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

## 2.6. ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ ФОРМУЛ ЛОГИКИ 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

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

должна пониматься так:

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

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

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

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

может выглядеть так:

$$[]p \Rightarrow <> rOq.$$

**Пример 2.2.** Рассмотрим примеры записи некоторых свойств бесконечных вычислений с помощью формул LTL.

1. $\mathcal{G}(q \Rightarrow \mathcal{X}\mathcal{G}\neg q)$ — $q$ встретится в будущем не более одного раза,
2. $\mathcal{F}q \wedge \mathcal{G}(q \Rightarrow \mathcal{X}\mathcal{G}\neg q)$ — $q$ встретится в будущем точно один раз,
3. $p \Rightarrow \mathcal{F}q$ — на $p$, наступившем в начальном состоянии, когда-нибудь в будущем будет реакция $q$,
4. $\mathcal{G}(p \Rightarrow \mathcal{F}q)$ — на любое встретившееся в вычислении $p$ всегда в будущем будет реакция $q$,
5. $\mathcal{G}(p \Rightarrow p\,\mathcal{U}\,q)$ — всегда если запрос $p$ будет подан, реакция $q$ на него обязательно будет получена, а до ее получения запрос $p$ не сбросится,
6. $\mathcal{F}q \Rightarrow (\neg p)\,\mathcal{U}\,q$ — событие $p$ не выполнится до наступления события $q$.

**Пример 2.3.** Свойства поведения дискретных логических схем тоже могут быть описаны с помощью формул логики LTL. На рис. 2.10 представлена временная диаграмма — график значений двух сигналов на выходе электронной схемы. Состояния схемы фиксируются в каждом такте.

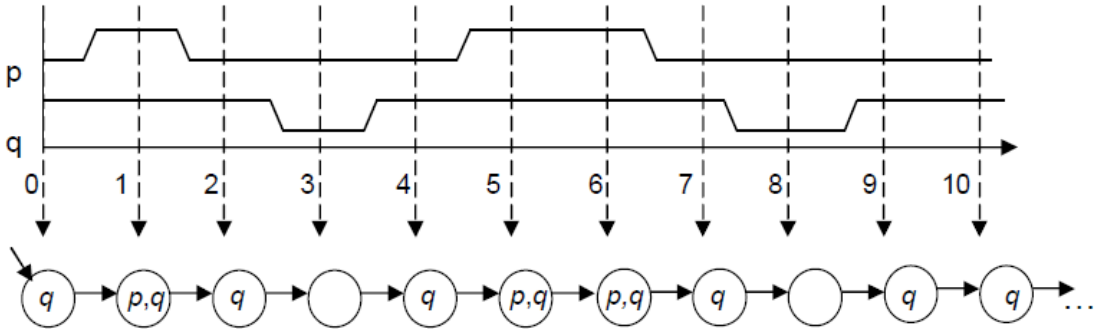


Рис. 2.10. Пример временной диаграммы и ее модели — вычисления

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 $q$.

На этом вычислении (назовем его $\sigma$) можно интерпретировать (определить истинностное значение) любую формулу LTL с атомарными предикатами $p$ и $q$. В частности, на вычислении $\sigma$ выполняются следующие свойства:

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

Последнее свойство истинно на вычислении $\sigma$, 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 $\omega$-языков). Например, формула

$$(a \vee b)\,\mathcal{U}\,\mathcal{G}b$$

задает все бесконечные цепочки из $a$ и $b$, содержащие только конечное число вхождений символа $a$ (подробнее этот вопрос рассматривается в разделе 3).

## 2.7. СООТНОШЕНИЯ МЕЖДУ ОПЕРАТОРАМИ ЛОГИКИ LTL

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

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

$$p\,\mathcal{U}\,q \equiv q \vee p \wedge \mathcal{X}q \vee p \wedge \mathcal{X}p \wedge \mathcal{X}\mathcal{X}q \vee \ldots;$$
$$\mathcal{F}q \equiv q \vee \mathcal{X}q \vee \mathcal{X}\mathcal{X}q \vee \mathcal{X}\mathcal{X}\mathcal{X}q \vee \ldots;$$
$$\mathcal{G}q \equiv q \wedge \mathcal{X}q \wedge \mathcal{X}\mathcal{X}q \wedge \mathcal{X}\mathcal{X}\mathcal{X}q \wedge \ldots.$$

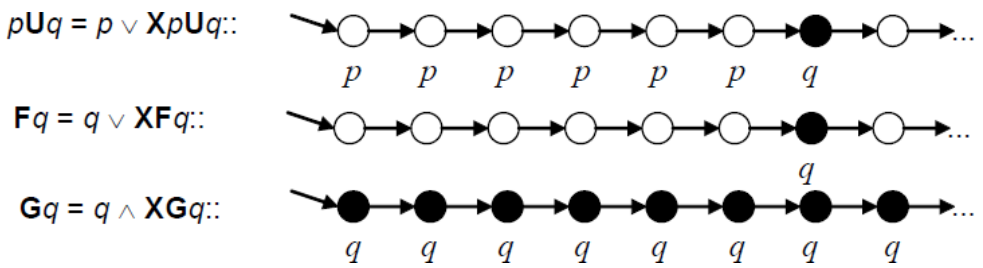


Рис. 2.11. Рекурсивное определение темпоральных операторов

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

$$p\,\mathcal{U}\,q \equiv q \vee p \wedge \mathcal{X}(p\,\mathcal{U}\,q);$$
$$\mathcal{F}q \equiv q \vee \mathcal{X}\mathcal{F}q;$$
$$\mathcal{G}q \equiv q \vee \mathcal{X}\mathcal{G}q.$$

Эти определения ясны и без формального доказательства. Приведем также несколько соотношений, которые легко могут быть доказаны на основании формального определения семантики формул LTL.

Комбинации темпоральных операторов и булевых операций:

$$\mathcal{X}(p \vee q) \equiv \mathcal{X}p \vee \mathcal{X}q;$$
$$\mathcal{X}(p \wedge q) \equiv \mathcal{X}p \wedge \mathcal{X}q;$$
$$\neg\mathcal{X}p \equiv \mathcal{X}(\neg p);$$
$$\mathcal{F}(p \vee q) \equiv \mathcal{F}p \vee \mathcal{F}q;$$
$$\mathcal{G}(p \wedge q) \equiv \mathcal{G}p \wedge \mathcal{G}q;$$
$$(p \wedge q)\,\mathcal{U}\,r \equiv p\,\mathcal{U}\,r \wedge q\,\mathcal{U}\,r;$$
$$p\,\mathcal{U}\,(q \vee r) \equiv p\,\mathcal{U}\,q \vee p\,\mathcal{U}\,r.$$

Законы поглощения (идемпотентность):

$$\mathcal{F}\mathcal{F}p \equiv \mathcal{F}p;$$
$$\mathcal{G}\mathcal{G}p \equiv \mathcal{G}p;$$
$$p\,\mathcal{U}\,q \equiv p\,\mathcal{U}\,(p\,\mathcal{U}\,q);$$
$$\mathcal{F}\mathcal{G}\mathcal{F}p \equiv \mathcal{G}\mathcal{F}p;$$
$$\mathcal{G}\mathcal{F}\mathcal{G}p \equiv \mathcal{F}\mathcal{G}p.$$

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

- $\mathcal{G}p \Rightarrow \mathcal{F}p$ — то, что всегда будет, то когда-нибудь наступит;
- $p \Rightarrow \mathcal{F}p$ — то, что есть, то когда-нибудь наступит;
- $(\mathcal{G}p \vee \mathcal{G}q) \Rightarrow \mathcal{G}(p \vee q)$ — если всегда будет истинно $p$ или всегда будет истинно $q$, то всегда будет истинно $p \vee q$.

## 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 можно 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.

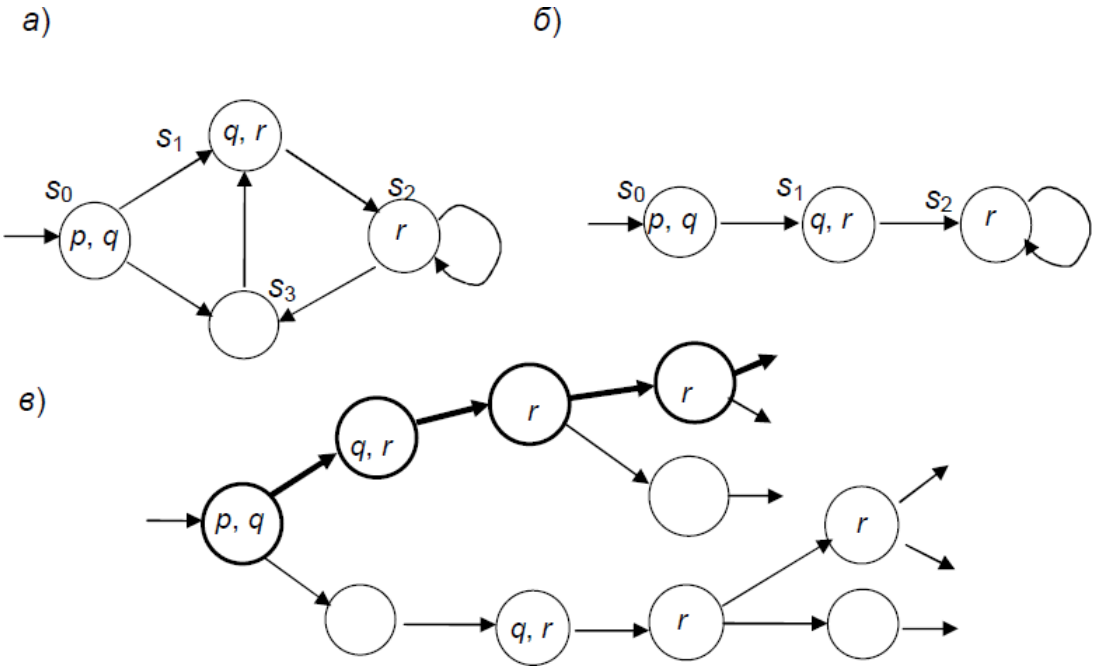


Рис. 2.12. Структура Крипке (а), одно из ее вычислений (б) и ее дерево вычислений (в)

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). Но поведения реальных информационных систем 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 возможных. 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* представлено дерево всех возможных вычислений модели — 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.* Это пятерка $M = (S, S_0, R, AP, L)$, где

- $S$ — конечное непустое множество состояний;
- $S_0$ — непустое множество начальных состояний;
- $R \subseteq S \times S$ — тотальное отношение на $S$, т. е. множество переходов, удовлетворяющее требованию $(\forall s)(\exists s') : (s, s') \in R$ (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 $\{s_0, s_1, s_2, s_3\}$, одним начальным состоянием $s_0$, множеством атомарных предикатов $\{p, q, r\}$ и функцией пометок $L : L(s_0) = \{p, q\}, L(s_1) = \{q, r\}, L(s_2) = \{r\}, L(s_3) = \{\}$.

**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. $\sigma(i)$ — некоторое состояние структуры Крипке.

Одно из вычислений структуры Крипке (см. рис. 2.12, *а*) — $s_0 s_3 s_1 s_2 s_2 s_3 \ldots$

**Определение 2.10.** *Траектория структуры Крипке.* Траекторией структуры Крипке $M$ при вычислении $\sigma = q_0 q_1 q_2 q_3 \ldots$ называется бесконечная цепочка $L(\sigma) = L(q_0)L(q_1)L(q_2)L(q_3)\ldots$, т. е. цепочка подмножеств атомарных предикатов, истинных в соответствующих состояниях вычисления $\sigma$.

Траекторией структуры Крипке при вычислении $s_0 s_3 s_1 s_2 s_2 s_3 \ldots$ является следующая цепочка: $\{p, q\}\{\}\{q, r\}\{r\}\{r\}\{\}\ldots$.

Траектории структуры Крипке иногда называют *трассами*. Траектории фокусируются на последовательностях 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 $\omega$-словами. Каждая траектория структуры Крипке бесконечна. Поэтому траектории структуры Крипке называются также ее $\omega$-словами.

**Пример 2.4.** Вычисления, представленные на рис. 2.12, *в*, определяют следующие траектории ($\omega$-слова) — бесконечные цепочки подмножеств множества атомарных предикатов, которые выполняются в последовательно проходимых состояниях:

$$\begin{aligned}
&\pi_1 = \{p,q\}\{q,r\}\{r\}\{\}\{q,r\} && \ldots;\\
&\pi_2 = \{p,q\}\{q,r\}\{r\}\{r\} && \ldots;\\
&\pi_3 = \{p,q\}\{\}\{q,r\}\{r\}\{\}\{q,r\} && \ldots;\\
&\pi_4 = \{p,q\}\{\}\{q,r\}\{r\}\{r\} && \ldots.
\end{aligned}$$

Множество всех $\omega$-слов структуры Крипке $M$ называется $\omega$-языком, допускаемым $M$.

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. Тотальность множества переходов означает, что из каждого состояния существует хотя бы один переход.

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. [6]).

Поскольку при анализе реагирующих систем нас интересуют бесконечные вычисления и соответствующие им траектории ($\omega$-слова), то все состояния структуры Крипке являются принимающими состояниями — любое $\omega$-слово, которое соответствует какому-либо вычислению структуры Крипке $M$, допускается $M$. Если моделируемая система останавливается в каком-то 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.

Главным преимуществом структуры Крипке является возможность конечным образом задавать поведения дискретных систем — бесконечное число бесконечных вычислений и траекторий.

## 2.9. РАСШИРЕННАЯ ТЕМПОРАЛЬНАЯ ЛОГИКА ВЕТВЯЩЕГОСЯ ВРЕМЕНИ

С помощью формул линейной темпоральной логики можно описать свойства конкретного вычисления — одного из путей, начинающегося из любого состояния системы переходов. 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 временные свойства. Например, формула $\mathcal{X}\mathcal{G}r$ будет истинна не для всех вычислений структуры Крипке $M$ (рис. 2.12, *б*), а только для одной из них, выделенной на рис. 2.12, *в*). Поэтому для каждой формулы пути на структуре Крипке следует указать, для какого именно вычисления (для какого пути) проверяется истинность этой формулы.

Каждое состояние структуры Крипке является началом бесконечного множества вычислений (корнем дерева вычислений), поэтому точно указать 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 *англ.* Exists) перед темпоральной формулой пути $\varphi$ утверждает, что *существует вычисление, на котором формула $\varphi$ истинна*. Если формула $\mathrm{E}\varphi$ истинна в начальном состоянии структуры Крипке $M$, то будем говорить, что она выполняется для $M$ и писать $M \models \mathrm{E}\varphi$ со

следующей интерпретацией: *на структуре Крипке $M$ существует вычисление, на котором выполняется формула $\varphi$.*

**Пример 2.5.** (см. рис. 2.12, *в*). Формула $\mathtt{E}\mathcal{X}\mathcal{G}r$ истинна для структуры Крипке $M$, т. е. $M \models \mathtt{E}\mathcal{X}\mathcal{G}r$. Действительно, из начального состояния структуры Крипке $M$ существует ($\mathtt{E}$) такой путь (а именно, $s_0s_1s_2s_2s_2\ldots$), что со следующего состояния этого пути ($\mathcal{X}$) во всех его состояниях ($\mathcal{G}$) выполняется атомарный предикат $r$.

Темпоральная формула $\mathtt{E}\varphi$ с квантором пути $\mathtt{E}$ характеризует уже не вычисление, а состояние структуры Крипке, из которого существует вычисление (начинается ветвь вычисления), удовлетворяющее темпоральной формуле пути $\varphi$. Формально возможность спецификации всех таких свойств дает расширенная темпоральная логика ветвящегося времени CTL* (Extended Computational Tree Logic). В этой логике, в дополнение к темпоральным операторам Until и NextTime, введен квантор пути $\mathtt{E}$. В CTL* могут присутствовать как формулы пути (характеризующие некоторое вычисление), так и формулы состояния.

**Определение 2.11.** *Логика CTL*.* Формулы CTL* — это формулы состояний, они задаются следующей грамматикой:

$\Phi$ ::= // формулами CTL* являются такие формулы состояний:
- $p \mid q \ldots \mid$ // атомарные предикаты;
- $\neg\Phi \mid \Phi \vee \Phi \mid$ // формулы состояний, связанные булевыми операциями;
- $\mathtt{E}\varphi$ // формулы пути, связанные квантором пути.

Формулы пути логики CTL* задаются следующей грамматикой:

$\varphi$ ::= // формулами пути CTL* являются такие формулы:
- $\Phi \mid \ldots \mid$ // формулы состояний этой логики (характеризуют пути, начинающиеся в этом состоянии);
- $\neg\varphi \mid \varphi \vee \varphi \mid$ // формулы пути, связанные булевыми операциями;
- $\mathcal{X}\varphi \mid \varphi\mathcal{U}\varphi$ // формулы пути, связанные темпоральными операторами.

Дополнительно к определению формул логики LTL в логике CTL* добавляется только один квантор $\mathtt{E}$ — *существует такой путь, что* .... Для удобства вводится квантор пути $\mathtt{A}$ (от *англ.* All) как сокращенное обозначение: $\mathtt{A}\varphi = \neg\mathtt{E}\neg\varphi$. Формула $\mathtt{A}\varphi$ понимается так:

*на всех путях из данного состояния формула пути $\varphi$ истинна*. Как и в LTL, в качестве формул пути также используются формулы $\mathcal{F}\varphi$ и $\mathcal{G}\varphi$ как сокращения: $\mathcal{F} = true\,\mathcal{U}\,\varphi$ и $\mathcal{G} = \neg\mathcal{F}\neg\varphi$.

Квантор пути `E` понимается как *возможное (possibly)* выполнение следующей за ним темпоральной формулы пути: она 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 считают формулой пути в следующем смысле: формула состояния выполняется на некотором пути, если она истинна в начальном состоянии этого пути. Для каждой формулы пути в состоянии необходимо указывать, какому вычислению, начинающемуся в данном состоянии, соответствует формула пути. Если явного указания нет, то предполагается, что формула пути относится ко всем возможным вычислениям, начинающимся из данного состояния.

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

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, интерпретирующиеся на вычислениях, имеющих единственное будущее. Если на структуре Крипке $M$ проверяется выполнение формулы LTL $\varphi$, то это значит, что проверяется утверждение: *На всех вычислениях $M$ выполняется формула $\varphi$*. Поэтому любую формулу пути $\varphi$ логики LTL можно представить в логике CTL* формулой состояния `A`$\varphi$ явным добавлением перед $\varphi$ 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 различных областях.

Другим подмножеством логики CTL* является логика CTL (Computational Tree Logic) — *логика ветвящегося времени*. В этой логике допустимы только формулы, в которых каждый темпоральный оператор $\mathcal{X}, \mathcal{U}, \mathcal{F}$ и $\mathcal{G}$ (характеризующий некоторое вычисление) предваряется квантором пути — A или E, что превращает любую темпоральную формулу 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 его альтернатив. Следовательно, время в этих логиках представляется ветвящейся структурой в каждом текущем состоянии.

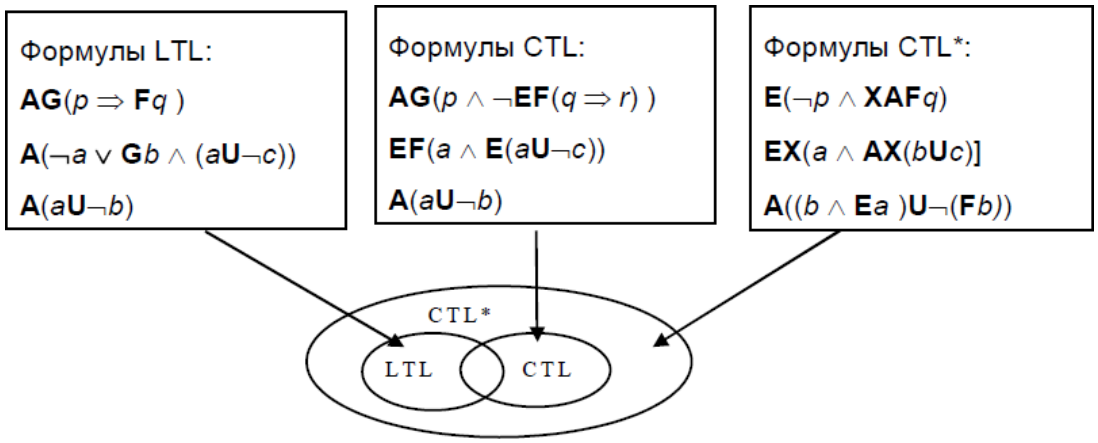


Рис. 2.13. Темпоральные логики

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 — формулы путей, в то время как все формулы CTL — формулы 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 A$\varphi$, где $\varphi$ — формула, не содержащая кванторов пути, но она может содержать произвольную комбинацию темпоральных операторов, а логика CTL включает только формулы, в которых каждому темпоральному оператору непосредственно предшествует квантор пути. Приведем синтаксис (правила построения формул) всех трех логик.

| | | | | |
|---|---|---|---|---|
| LTL | Формулы состояний: | $\Phi$ | ::= | $\mathtt{A}\varphi$ |
| | Формулы путей: | $\varphi$ | ::= | $p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathcal{X}\varphi \mid \varphi\,\mathcal{U}\,\varphi$ |
| CTL | Формулы состояний: | $\Phi$ | ::= | $p \mid \neg\Phi \mid \Phi \vee \Phi \mid \mathtt{E}\varphi \mid \mathtt{A}\varphi$ |
| | Формулы путей: | $\varphi$ | ::= | $\mathcal{X}\Phi \mid \Phi\,\mathcal{U}\,\Phi \mid \mathcal{F}\Phi \mid \mathcal{G}\Phi$ |
| CTL* | Формулы состояний: | $\Phi$ | ::= | $p \mid \neg\Phi \mid \Phi \vee \Phi \mid \mathtt{E}\varphi \mid \mathtt{A}\varphi$ |
| | Формулы путей: | $\varphi$ | ::= | $\Phi \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathcal{X}\varphi \mid \varphi\,\mathcal{U}\,\varphi$ |

Некоторые формулы относятся к обоим классам. Например, формула $\mathtt{A}(a\,\mathcal{U}\,\neg b)$ является как формулой LTL, так и формулой CTL. Формула $\mathtt{A}[\mathcal{G}(p \Rightarrow \mathcal{F}q)]$ — это формула LTL, но не формула CTL. Формула $\mathtt{A}\mathcal{G}(p \Rightarrow \mathtt{E}\mathcal{F}q)$ — это формула CTL, но не формула LTL. Все эти формулы являются формулами логики CTL*.

Формулы логики CTL* интерпретируются на вычислениях структуры Крипке, которые можно представить бесконечным деревом (см. 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.** 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:

$$\varphi_{br} = \mathtt{A}[(\mathcal{G}life) \Rightarrow (\mathcal{G}\mathtt{E}\mathcal{X}death)].$$

Эта формула логики CTL* утверждает, что вдоль всех вычислений ($\mathtt{A}$) с вечной жизнью ($\mathcal{G}life$) всегда ($\mathcal{G}$) возможно ($\mathtt{E}$) встретить смерть в следующий момент ($\mathcal{X}death$). Строфа Пушкина как раз и свидетельствует о возможности в любой момент переключения нашего бытия на другую ветвь, на которой нас ожидает смерть.

Структура Крипке, которая соответствует строкам Пушкина, представлена на рис. 2.14, *а*: в каждый момент времени (в каждом состоянии) перед нами открывается альтернатива (которую обычно выбираем не мы!): или жить дальше, или умереть. Особенно четко это показывает развертка структуры Крипке (рис. 2.14, *б*). Формула $\varphi_{br}$ на ней выполняется.

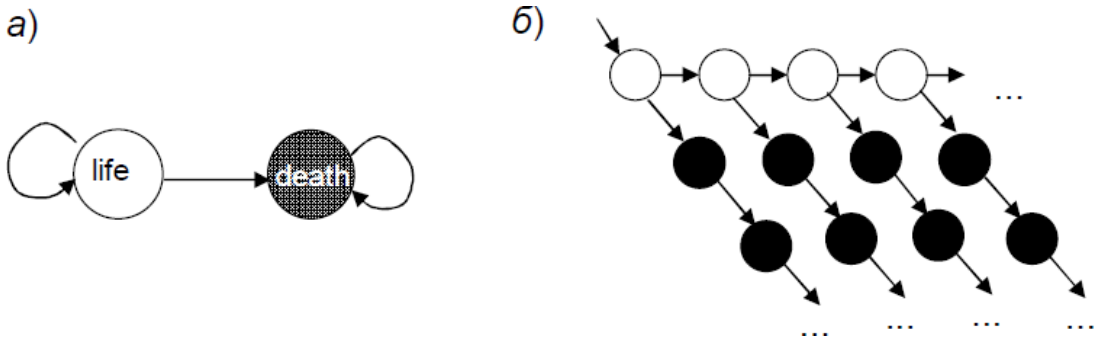


Рис. 2.14. Структура Крипке и ее развертка

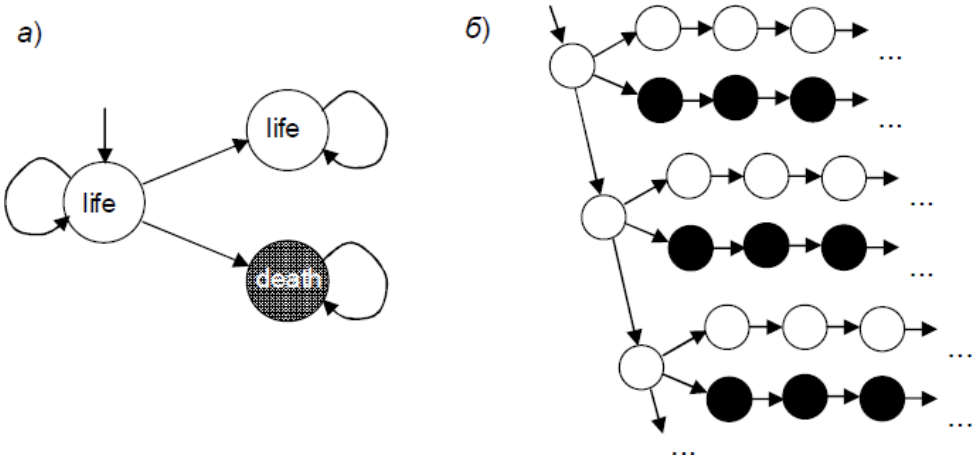


Рис. 2.15. Структура Крипке (а) и ее дерево вычислений (б)

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, представленные на рис. 2.14 и 2.15, имеют одно и то же множество вычислений (рис. 2.16), которое может быть выражено $\omega$-регулярным выражением $\{life^{\omega} \cup life^{+}death^{\omega}\}$: одна бесконечная цепочка $life$ и множество цепочек с любым (ненулевым) конечным числом $life$, за которыми следует бесконечное число $death$.

Поскольку обе структуры Крипке имеют одно и то же множество линейных вычислений, никакая формула LTL не может их различить. Этот пример, в частности, показывает, что логика CTL* более мощная, чем логика LTL: формулы CTL* различают рассмотренные структуры Крипке.

Комбинации кванторов путей A и E с темпоральными операторами $\mathcal{G}$ и $\mathcal{F}$ имеют достаточно очевидный смысл:

- $\mathtt{A}\mathcal{G}\varphi$ — $\varphi$ — инвариант ($\varphi$ выполняется всегда, в любом состоянии системы);

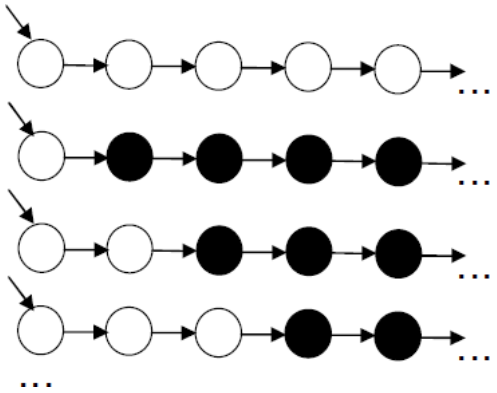

Рис. 2.16. Вычисления двух структур Крипке (см. рис. 2.14 и 2.15)

- $\mathtt{E}\mathcal{G}\varphi$ — потенциально всегда $\varphi$ (возможно, что $\varphi$ будет выполняться все время);
- $\mathtt{A}\mathcal{F}\varphi$ — когда-нибудь обязательно выполнится $\varphi$;
- $\mathtt{E}\mathcal{F}\varphi$ — когда-нибудь, возможно, выполнится $\varphi$.

**Пример 2.7.** Пусть $p$ означает *Я люблю Машу*, $q$ — *Я люблю Дашу*. Тогда:

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

## 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 применяются в корпоративных технологических циклах разработки систем.

Существует множество черт, которые отличают логики LTL и

CTL друг от друга, но главные различия необходимо выделить 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 свидетельствует о том, что эта формула должна выполняться на *всех возможных* путях данной структуры Крипке. Логика LTL более проста и интуитивно более понятна, чем CTL;
- формулы CTL являются формулами состояний; в логике CTL любой темпоральный оператор предварен квантором пути, который определяет, на всех или только на некоторых путях из текущего состояния структуры Крипке этот темпоральный оператор действует.

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 $\varphi$ LTL выполняется на структуре Крипке $M$, если $\varphi$ выполняется на *всех* траекториях вычислений структуры Крипке. Эти вычисления бесконечны, и их бесконечное число. Поэтому возникают большие трудности при проверке выполнения формулы LTL на структуре Крипке, чем при проверке формул CTL. Формула $\varphi$ LTL выполняется на структуре Крипке, если $\varphi$ истинна *в начальном состоянии* каждого бесконечного вычисления данной структуры;
- формулы логики CTL интерпретируются на деревьях вычислений структуры Крипке. Дерево вычислений — это развертка структуры Крипке с корнем в данном состоянии. Формулы CTL принимают истинное или ложное значение в каждом состоянии структуры Крипке, а таких состояний конечное число. Это является причиной эффективности алгоритма model checking для формул CTL. Формула $\varphi$ CTL выполняется на структуре Крипке $M$, если $\varphi$ истинна в *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 ал-

горитмы model checking для LTL, например, Spin. Сложность алгоритма верификации линейна относительно числа состояний структуры Крипке и экспоненциальна относительно сложности (числа подформул) формулы LTL;

- существуют другие инструменты, реализующие алгоритмы верификации для формул CTL, например, SMV. Сложность алгоритма верификации 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»:

$$CTL: \mathtt{A}\mathcal{G}(send \Rightarrow \mathtt{A}\mathcal{F}req);$$
$$LTL: \mathcal{G}(send \Rightarrow \mathcal{F}req).$$

Однако неверно утверждение, что если в формуле LTL каждый темпоральный оператор будет предварен квантором пути A, то построенная таким образом формула CTL будет эквивалентна исходной формуле.

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 $\mathtt{A}\mathcal{G}\mathtt{E}\mathcal{F}\varphi$ логики CTL, выражающая *достижимость свойства* $\varphi$, невыразима в LTL;
- формула $\mathcal{F}\mathcal{G}\varphi$ логики LTL, выражающая *стабилизацию свойства* $\varphi$ *когда-нибудь в будущем*, невыразима в CTL.

## 2.11. МЕТОД ПРОВЕРКИ МОДЕЛЕЙ 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 (на структуре Крипке).

За последние 20 лет структуры Крипке стали общепризнанной моделью реагирующих систем, с помощью которой можно адекватно представить поведение реагирующих систем: дискретные системы управления, параллельные и распределенные алгоритмы, протоколы и

т. п., а темпоральные логики признаны очень эффективным формализмом для выражения свойств таких систем. Этапы доказательства того, что поведение реагирующей системы обладает некоторым свойством:

1. для верифицируемой системы строится адекватная модель Крипке $M$, т. е. система переходов с конечным числом состояний. Поведения реальной системы представляются разверткой (деревом вычислений) структуры Крипке;
2. из переменных и параметров системы строятся интересующие разработчика атомарные предикаты структуры Крипке — логические выражения, которые могут принимать значения *истина* или *ложь* в каждом состоянии системы,
3. проверяемое свойство выражается формулой $\varphi$ 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 $\varphi$) с помощью полностью автоматизированной процедуры.

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

## 2.12. БИБЛИОГРАФИЧЕСКИЙ КОММЕНТАРИЙ

Первые попытки учесть роль временного фактора в логических утверждениях относятся еще к античности. Идея введения временных понятий в логику восходит к греческой философии. 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 гг. прошлого века английский логик Артур Прайор впервые выполнил формализацию темпоральной логики, которая была им названа Tense Logic [29]. Во временной логике Прайора были впервые введены темпоральные операторы $\mathcal{F}$ и $\mathcal{P}$. Эти операторы (и выводимые из них операторы $\mathcal{G}$ и $\mathcal{H}$) вошли в так называемую Minimal Tense Logic, которая впоследствии была расширена во многих направлениях. Различные расширения Tense Logic изучаются и в настоящее время.

В лингвистике одну из первых попыток формального анализа предложений, ссылающихся на время, сделал Ганс Райхенбах. Его идея о возможности формализации времен английских глаголов с

помощью соотношения моментов наступления трех событий $S, R$ и $E$ была изложена в книге [31].

Использование темпоральной логики для спецификации свойств и верификации технических систем связано с именем Амира Пнуэли. 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, [28]) как развитие логики 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 особые методы анализа.

Формальная модель *структура Крипке* введена в [24] Саулом Крипке. Темпоральная логика ветвящегося времени была введена и исследована Эдмундом Кларком и Алленом Эмерсоном в работах [14], [15] и других. Сравнение различных темпоральных логик можно найти в [18].

## ЗАДАЧИ

1. Постройте в модели Райхенбаха соотношение времен наступления событий $E, R$ и $S$ для следующих предложений английского языка:

   а) 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.

   Расположите эти события на временной оси.
2. Пусть $p$ означает *Я люблю Машу*, а $q$ — *Я люблю Дашу*. Каким высказываниям соответствуют следующие формулы LTL:
   (a) $\mathcal{F}(p \Rightarrow \neg q)$;
   (b) $p \mathcal{U} \mathcal{G} q$.
3. Представьте в виде формулы LTL высказывание: *Канал может терять сообщение только конечное число раз*.
4. Постройте LTL формулу, которая будет задавать вычисления со

следующими свойствами:

(a) если произойдет $p$, то $q$ никогда не случится;

(b) если произойдет $p$, то когда-нибудь в будущем будет выполнено $q$, а сразу после этого произойдет $r$;

(c) если произойдет $p$, то в будущем произойдет $q$, но между ними не произойдет $r$.

5. Можно ли выразить любую формулу CTL* с помощью операторов $\neg, \vee, \mathcal{X}, \mathcal{U}$, E? Если да, то выразите с их помощью формулу $\mathtt{A}\mathcal{F}[p \Rightarrow \mathcal{X}(q\,\mathcal{U}\,r)]$. Постройте структуру Крипке, на которой эта формула выполняется.
6. Выразите формулу $\mathtt{A}[p\,\mathcal{U}\,q]$ логики CTL с помощью других формул этой логики.
7. Пусть $p$ означает *Я люблю Машу*, а $q$ — *Я люблю Дашу*. Каким высказываниям соответствуют следующие формулы CTL:

(a) $\mathtt{A}\mathcal{F}\mathtt{E}\mathcal{G}p$;

(b) $\mathtt{E}\mathcal{F}\mathtt{A}\mathcal{G}p$;

(c) $\mathtt{A}(p\,\mathcal{U}\,q)$.

8. Докажите, что формула $\mathcal{G}(p \vee q) \Rightarrow (\mathcal{F}\mathcal{G}p \vee \mathcal{G}\mathcal{F}q)$ общезначима (т. е. истинна на всех ее интерпретациях).
9. Формализуйте утверждения в логике LTL:

(a) «я выйду замуж не менее двух раз»;

(b) «я выйду замуж не более двух раз»;

(c) «я выйду замуж точно два раза»;

(d) «я выйду замуж не менее одного раза».

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

10. Постройте структуру Крипке, на которой выполняется формула $\mathtt{E}\mathcal{G}\mathtt{E}\mathcal{F}p$, и не выполняется формула $\mathtt{E}\mathcal{G}\mathcal{F}p$.
11. Можно ли выразить любую формулу CTL* с помощью операторов $\neg, \vee, \mathcal{X}, \mathcal{U}$, E? Если да, то выразите с их помощью формулу $\mathtt{A}\mathcal{F}(p \Rightarrow \mathtt{E}\mathcal{X}(q\,\mathcal{U}\,r))$. Постройте структуру Крипке, на которой эта формула выполняется.
12. Выразите в LTL свойство: между каждой парой состояний, удовлетворяющих свойству $p$, обязательно встретится состояние, удовлетворяющее свойству $q$.

## ЗАКЛЮЧЕНИЕ К РАЗДЕЛУ 2

Задачей временной логики является построение формализованных языков, способных сделать рассуждения о предметах и явлениях, развивающихся во времени, т. е. обладающих поведением, более точными и поэтому 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 важен именно порядок событий, а не явное время 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».

Существует несколько вариантов темпоральных логик. В области верификации технических систем используются в основном две такие логики — логика 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. В логике линейного времени все вычисления структуры Крипке рассматриваются как множество бесконечных траекторий поведения системы (трасс). Формулы этой логики построены из атомарных утверждений, связанных логическими операциями и темпоральными операторами. В логике ветвящегося времени рассматриваются все варианты развития вычислений в каждом состоянии бесконечных траекторий.

До настоящего времени остается открытым вопрос о том, какая из логик, CTL или LTL, более удобна и выразительна для спецификации свойств реагирующих систем. Достаточно широко распространено мнение, что свойства систем легче выражать в логике LTL, в то время как верификация проще для формул, выраженных в CTL [33]. Обе логики могут быть использованы для формулировки утверждений о поведении технических систем. Существуют свойства систем, которые можно формализовать в одной логике и нельзя в другой.

# 3. АЛГОРИТМ MODEL CHECKING ДЛЯ LTL

В этом разделе рассмотрен алгоритм проверки моделей для подкласса темпоральных логик — логики 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 LTL задаются важные свойства поведения реагирующих систем, причем некоторые из этих свойств выразить формулами CTL нельзя (впрочем, как было указано ранее, справедливо и обратное: некоторые свойства, выражаемые в логике CTL, нельзя выразить в логике LTL). Разработано несколько подходов к верификации LTL формул, один из них основывается на теории автоматов.

Алгоритм проверки выполнимости формулы LTL на структуре Крипке, построенный на автоматном подходе, не так уж 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 — небольшие, 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 нее верифицированы многие практические разработки.

## 3.1. ПРОВЕРКА ВЫПОЛНИМОСТИ ФОРМУЛ LTL

При использовании алгоритма проверки модели для формулы $\varphi$ логики LTL необходимо убедиться в выполнении $\varphi$ на *каждом* вычислении, начинающемся в начальном состоянии структуры Крипке. Например, для проверки того, выполняется ли на структуре Крипке $M$ (рис. 3.1, *а*) формула $\mathcal{GF}(r \Rightarrow q)$, нужно проверить, что на всех возможных вычислениях $M$ (а их бесконечное число!) свойство $r \Rightarrow q$ будет выполняться бесконечное число раз!

Иначе говоря, 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. 3.1, *б* вычислению $s_0 s_1 s_2 s_2 \ldots$ соответствует траектория $\{p, q\}\{q, r\}\{r\}\{r\} \ldots$

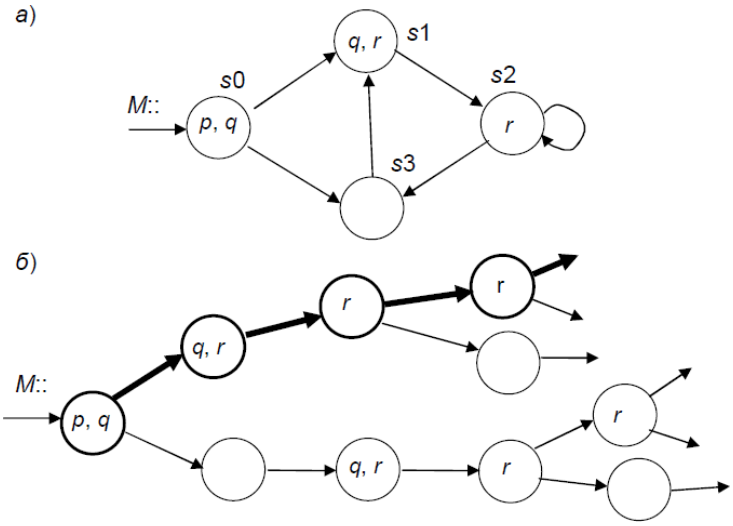


Рис. 3.1. Структура Крипке (а) и ее дерево вычислений (б)

Отметим, что для произвольной формулы LTL $\varphi$ и произвольной структуры Крипке $M$ может оказаться, что на $M$ не выполняются ни $\varphi$, ни $\neg\varphi$.

Если на структуре Крипке $M$ *не* выполняется формула $\varphi$, то $M$ имеет хотя бы одно вычисление, удовлетворяющее $\neg\varphi$. Такое вычисление назовем *контрпримером*. Даже одного контрпримера достаточно, чтобы опровергнуть утверждение $M \models \varphi$. Например, мы видим (см. рис. 3.1, *б*), что некоторые траектории $M$ удовлетворяют формуле $\varphi = \mathcal{GF}(r \Rightarrow q)$ (например, $\{p, q\}\{q, r\}\{r\}\{\}\{q, r\}\{r\}\{\} \ldots$), *а* другие не удовлетворяют ей (например, $\{p, q\}\{q, r\}\{r\}\{r\}\{r\}\{r\} \ldots$). Именно метод поиска контрпримеров лежит в основе одного из наиболее удобных и понятных подходов к разработке алгоритма проверки моделей для формул LTL. Если контрпример найти не удается, то формула на данной структуре Крипке выполняется. Бессмысленно искать контрпримеры перебором всех траекторий структуры Крипке $M$. Мало того, что траекторий бесконечное число, еще и каждая такая траектория бесконечна. Поэтому решение этой проблемы нетривиально.

**Определение 3.1.** *$\omega$-Слова и $\omega$-языки.* Любое конечное множество $\Sigma$ будем называть алфавитом. Бесконечные цепочки, состав-

ленные из элементов (символов) алфавита $\Sigma$, называются $\omega$-словами над $\Sigma$. Любое множество $\omega$-слов над $\Sigma$ называется $\omega$-языком над алфавитом $\Sigma$.

Множество всех бесконечных цепочек из символов алфавита $\Sigma$ обозначается $\Sigma^\omega$. Например, если $\Sigma = \{a, b, c\}$, то $\Sigma^\omega$ составят все бесконечные цепочки из этих символов, т. е. $\Sigma^\omega = \{aaaaa\ldots, ababab\ldots, baccbaccbacc\ldots, \ldots\}$. Примером $\omega$-языка над алфавитом $\Sigma = \{a, b\}$ является язык $b^*a^\omega$, состоящий из всех цепочек, у которых конечный начальный фрагмент состоит из символов $b$, а после него идет бесконечная последовательность символов $a$. Этот язык описывается LTL формулой $\varphi = (b \wedge \neg a)\,\mathcal{U}\,\mathcal{G}(a \wedge \neg b)$. Таким образом, темпоральные формулы LTL могут задавать $\omega$-языки — множества тех бесконечных цепочек, которые удовлетворяют этой формуле. Оказывается, что $\omega$-языки задаются и автоматами.

Идея, лежащая в основе алгоритма model checking для заданной формулы LTL, близка идее использования контрольного автомата (watchdog, буквально — *сторожевого пса*), проверяющего правильность функционирования основного устройства. Контрольный автомат (см. рис. 3.2, *а*) строится согласно спецификациям 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 соответствуют требованиям, то контрольный автомат переходит в свое состояние *ошибки*. Например, на рис. 3.2, *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 секунд основным устройством не будет выдан ответ ($response$). Достаточно найти только одну неправильную траекторию поведения системы, чтобы доказать некорректность этой системы. Очень часто такую проверку осуществить более просто, чем проверять, что *все* вычисления из бесконечного числа вычислений системы правильны.

Проблему построения контрольного автомата в общем случае для любой LTL-формулы $\varphi$ и для любой структуры Крипке $M$ можно формализовать в терминах формальных языков. Если мы сможем задать конечной моделью два $\omega$-языка: $\omega$-язык $L_M$, допускаемый $M$,

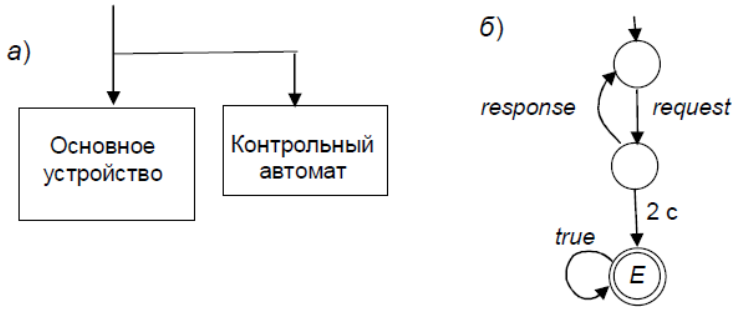


Рис. 3.2. Контроль правильности функционирования с помощью контрольного автомата *(а)* и вариант контрольного автомата *(б)*

и $\omega$-язык $L_{\neg\varphi}$, все цепочки которого неправильны, потому что удовлетворяют LTL-формуле $\neg\varphi$, и затем найти пересечение $L_M \cap L_{\neg\varphi}$ этих $\omega$-языков, то наша проблема будет решена. Действительно, если пересечение языков окажется пустым, то на структуре Крипке $M$ формула $\varphi$ выполняется, поскольку у $M$ нет ни одного вычисления, на котором выполняется формула $\neg\varphi$. Если пересечение $L_M \cap L_{\neg\varphi}$ этих $\omega$-языков окажется непустым, то $M$ допускает вычисление, удовлетворяющее $\neg\varphi$, следовательно, неверно, что *все* вычисления $M$ удовлетворяют формуле $\varphi$ (рис. 3.3).

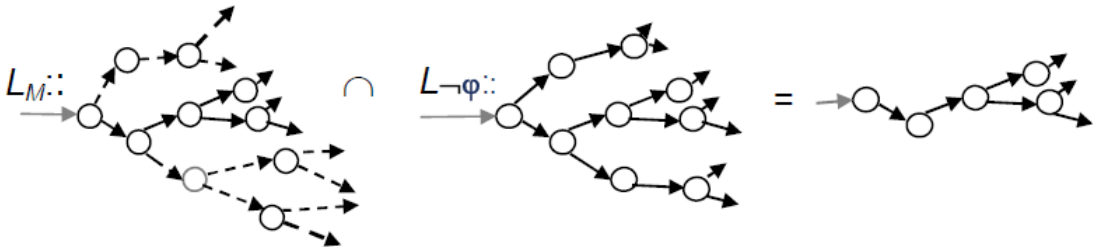


Рис. 3.3. Основная идея алгоритма проверки выполнимости LTL-формулы

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

## 3.2. ПЕРЕСЕЧЕНИЕ ЯЗЫКОВ В ТЕОРИИ КОНЕЧНЫХ АВТОМАТОВ

В теории конечных автоматов и автоматных языков идея использования контрольного автомата может быть формализована следующим образом. Языком над конечным алфавитом 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 два автомата, стоящие рядом и синхронно обрабатывающие одну и ту же входную цепочку.

**Определение 3.2.** *Конечный автомат.* Конечным автоматом называется пятерка $(S, \Sigma, s_0, \delta, F)$, где

- $s$ — конечное множество состояний;
- $\Sigma$ — конечное множество символов (входной алфавит);
- $s_0 \in S$ — начальное состояние (для недетерминированного автомата 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 алфавита $\Sigma$, называется *словом над* $\Sigma$. Множество всех слов над $\Sigma$ обозначается $\Sigma^*$. Например, если $\Sigma = \{a, b, c\}$, то множество $\Sigma^*$ составят все цепочки из этих символов, т. е. $\Sigma^* = \{\varepsilon, a, b, c, ab, aa, abbc, \ldots\}$. Символом $\varepsilon$ обозначается пустая цепочка, вовсе не содержащая символов.

Динамика (поведение) конечного автомата определяется его переходами из состояния в состояние под воздействием очередного входного символа. На вход автомата поступают слова — элементы множества $\Sigma^*$. Конечный 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 $L \subseteq \Sigma^*$ называется *автоматным языком*, если существует конечный автомат, допускающий $L$.

**Пример 3.1.** На рис. 3.4 приведены графы переходов двух

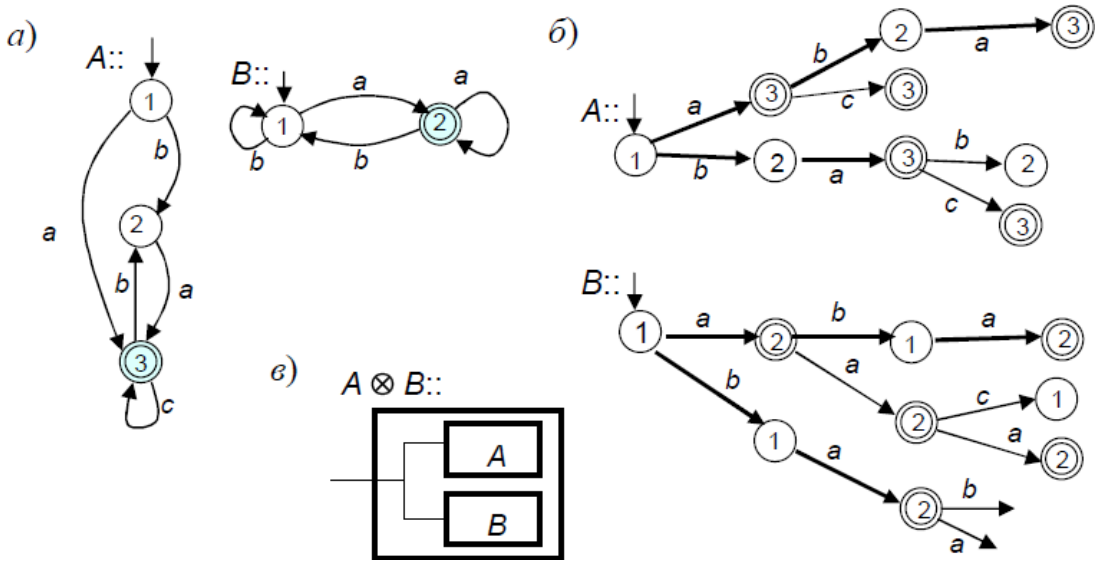


Рис. 3.4. Два конечных автомата *(а)*, развертки их поведений *(б)* и синхронная композиция автоматов *(в)*

конечных автоматов, $a$ и $b$. Начальные состояния автоматов показаны стрелками, финальные — двойными окружностями. Обозначим $L_A$ язык, допускаемый автоматом $a$. Слова языка, допускаемого конечным автоматом — это конечные цепочки из символов алфавита $\{a, b, c\}$, переводящие автомат из начального состояния в любое из допускающих состояний. Оба автомата допускают языки, содержащие бесконечное число 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 $a \in \Sigma$, можно считать, что этот переход ведет в новое состояние, не представленное в графе переходов, из которого любое допускающее состояние недостижимо.

Таким образом, $L_A = \{ba, acc, baccba, \ldots\}$, $L_B = \{a, ba, babbba, \ldots\}$. Конечные автоматы являются удобным конечным формализмом, позволяющим задавать бесконечные множества слов (языки) над конечным словарем.

Языки $L_A$ и $L_B$ пересекаются (см. рис. 3.4): 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. 3.4, *в*).

Множеством состояний автомата $A \otimes B$ являются все пары состояний $a$ и $b$. Начальные состояния $A \otimes B$ — пары, оба элемента которых — начальные состояния соответствующих автоматов. Допускающие состояния композиции автоматов $a$ и $b$ — это такие пары $\langle s_A, s_B \rangle$ состояний, в которых $s_A$ — допускающее состояние автомата $a$, а $s_B$ — допускающее состояние автомата $b$.

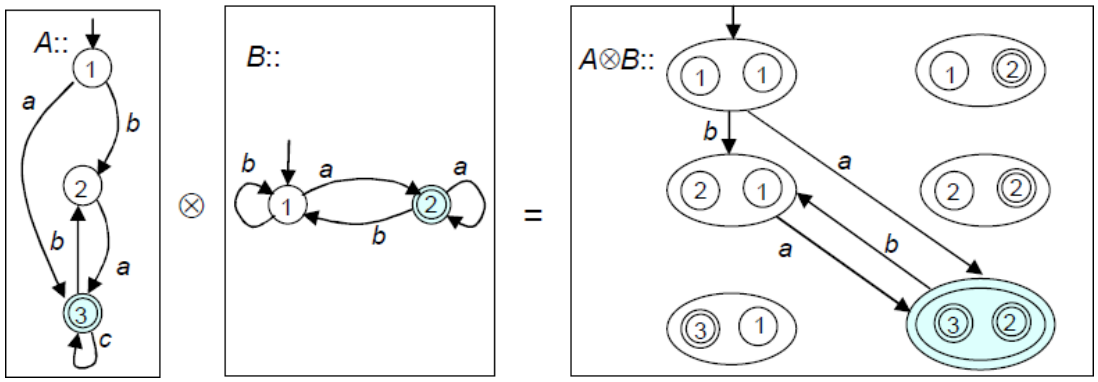


Рис. 3.5. Синхронная композиция автоматов $a$ и $b$

**Пример 3.2.** На рис. 3.5 построен конечный автомат $A \otimes B$ по автоматам, представленным на рис. 3.3. 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: $\langle 3, 2 \rangle$. Состояние 3 является допускающим в автомате $a$, а состояние 2 — допускающее 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 этот автомат $A \otimes B$ в его допускающее состояние, следовательно, она допускается этим автоматом. Легко заметить, что такая цепочка допускается как автоматом $a$, так и автоматом $b$ — слово $ba$ переводит каждый автомат из его начального в его допускающее состояние.

**Теорема 3.1.** *(о пересечении автоматных языков). Пересечение языков, допускаемых конечными автоматами $a$ и $b$, совпадает с языком, допускаемым синхронной композицией этих автоматов, т. е.* $L_A \cap L_B = L_{A \otimes B}$.

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.

Рассмотрим синхронную композицию автоматов 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 множества цепочек, допускаемых автоматом $a$ (потому что оно допускается и контрольным автоматом $b$ тоже). Например, любая цепочка, которая принимается синхронной композицией основного устройства и контрольного автомата (см. рис. 3.2), является ошибочной.

Определим как *ошибочную* любую цепочку, которая состоит из $a$ и $b$, но заканчивается $a$. Автомат $b$ (см. рис. 3.5) как раз и определяет 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 граф переходов $a$ и все возможные пути, ведущие в его принимающие состояния. 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 $a$ 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 (язык $L_A$) какие-нибудь цепочки из другого *бесконечного* множества (язык $L_B$) ошибочных цепочек.

Алгоритм проверки моделей для LTL формул работает полностью аналогично, хотя оперирует более сложными формальными моделями.

## 3.3. ТЕОРЕТИКО-АВТОМАТНЫЙ МЕТОД

Для реализации теоретико-автоматного подхода в верификации реагирующих систем, моделью которых является структура Крипке, требуется уметь конечным образом задавать язык $L_M$ — все цепочки, возможные в произвольной структуре Крипке $M$, задавать язык $L_{\neg\varphi}$ — все цепочки, не удовлетворяющие заданной формуле $\varphi$ логики LTL, а также 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 $\neg\varphi$.

Цепочки, характеризующие вычисления реагирующих систем, бесконечны, они называются $\omega$-словами. Множества $\omega$-слов называются $\omega$-языками, они задаются конечной моделью, которая называется автоматом Бюхи.

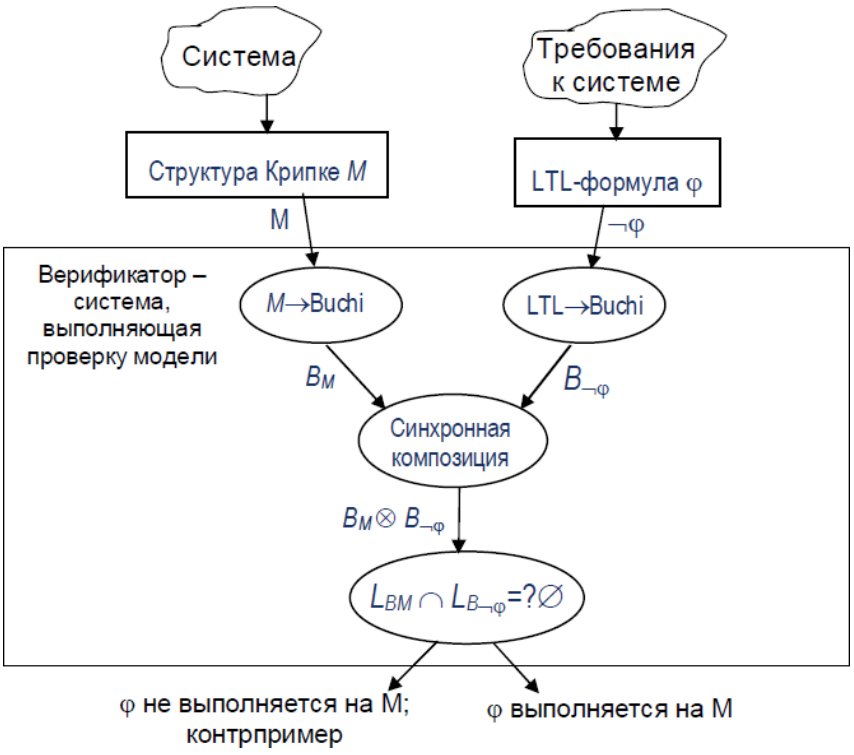


Рис. 3.6. Шаги алгоритма model checking для LTL формулы

Алгоритм проверки выполнимости формулы $\varphi$ LTL на структуре Крипке $M$ состоит из следующих шагов (рис. 3.6):

1. По структуре $M$ строится автомат Бюхи $B_M$. Этот автомат допускает все возможные траектории ($\omega$-слова) структуры Крипке $M$.
2. По формуле $\varphi$ строится ее отрицание и затем автомат Бюхи $B_{\neg\varphi}$, допускающий множество $\omega$-слов, которые удовлетворяют $\neg\varphi$. Этот автомат будет контрольным автоматом, принимающим 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 принимает пересечение языков, допускаемых обоими автоматами.
4. Проверяется пустота языка $L_{B_M \otimes B_{\neg\varphi}}$. Формула $\varphi$ выполняется для $M$, если и только если этот язык пуст. Если формула $\varphi$ не выполняется для $M$, то можно построить контрпример, путь в $M$, нарушающий свойство $\varphi$.

В следующих разделах мы обоснуем необходимость введения нового типа формальных моделей — автомата Бюхи, рассмотрим свойства таких автоматов и правила построения их синхронной композиции. Далее обсудим связь формул LTL и автоматов Бюхи как моделей для задания свойств бесконечных вычислений. Затем рассмотрим алгоритм построения синхронной композиции двух автоматов Бюхи и алгоритм определения того, что автомат Бюхи допускает пустой язык. В заключение рассмотрим правила построения автомата Бюхи, допускающего все цепочки, на которых заданная LTL формула выполняется.

## 3.4. АВТОМАТЫ БЮХИ

Классическая теория формальных языков 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, любой программист может написать только конечную программу, все сообщения имеют только конечную длину и т. п.). Теория формальных языков применяется в трансляции языков программирования и анализе текстов. Все эти приложения имеют дело только с конечными цепочками символов.

Цепочки, порождаемые вычислениями реагирующих систем, бесконечны, поскольку такие системы после запуска должны функци-

онировать неопределенно долго. Формальные модели, используемые при верификации данных систем, должны иметь дело с бесконечными цепочками — $\omega$-словами, и их множествами — $\omega$-языками. Поэтому первой нашей задачей является следующая: как с помощью автомата описать $\omega$-язык?

Конечные автоматы — удобный формализм, который конечным образом задает обычные формальные языки — бесконечные множества цепочек конечной длины. Нам нужен новый тип автомата, задающего бесконечные цепочки, $\omega$-слова. Существует несколько способов обобщения модели конечного автомата таким образом, чтобы он мог допускать бесконечные цепочки. Рассмотрим два 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 ($\omega$-слова) как допустимые.

**Определение 3.3.** *Автомат Бюхи.* Автомат Бюхи — это пятерка $(S, \Sigma, S_0, \delta, F)$, где

- $s$ — конечное множество состояний;
- $\Sigma$ — конечное множество символов (входной алфавит);
- $S_0 \in S$ — множество начальных состояний (содержащее только один элемент для детерминированного случая);
- $\delta : S \times \Sigma \to S$ — отношение перехода (для недетерминированного автомата $\delta : S \times \Sigma \to 2^S$);
- $F \subseteq S$ — множество допускающих (финальных, заключительных) состояний.

Любое $\omega$-слово $w$ над алфавитом $\Sigma$ можно считать функцией $w : N \to \Sigma$, понимая под $w(i)$ $i$-ю букву слова $w$. Аналогично, если $\pi : N \to S$ — бесконечная последовательность состояний автомата $a$, то $\pi(i)$ — $i$-е состояние в цепочке $\pi$. Обозначим $inf(\pi)$ множество состояний, которые встречаются в бесконечной цепочке $\pi$ бесконечно много раз.

**Определение 3.4.** *$\omega$-слово $w$ над алфавитом $\Sigma$ допускается автоматом Бюхи $A = (S, \Sigma, S_0, \delta, F)$*, если существует такое $\omega$-слово $\pi$ над множеством состояний $s$ (бесконечная цепочка состояний автомата $a$), что:

1. $\pi(0) \in S_0$ — цепочка состояний начинается одним из начальных состояний;
2. $\forall i \geq 0 : \pi(i+1) \in \delta(\pi(i), w(i))$ — переход в очередное состояние вызван очередной буквой $\omega$-слова $w$;
3. $inf(\pi) \cap F \neq \emptyset$ — среди бесконечно повторяющихся состояний цепочки $\pi$ существует хотя бы одно принимающее.

Таким образом, автомат Бюхи *допускает (распознает, принимает)* $\omega$-слово, если при чтении этого слова автомат бесконечно много раз проходит хотя бы одно принимающее состояние. Множество $\omega$-слов, допускаемых автоматом Бюхи $a$, называется *$\omega$-языком, допускаемым $A$*. $\omega$-язык $L \subseteq \Sigma^\omega$ называется *Бюхи-допускаемым*, если существует автомат Бюхи, допускающий $L$. Автоматы Бюхи эквивалентны, если допускаемые ими языки совпадают. Поскольку автоматы Бюхи используются для задания $\omega$-языков, их также называют $\omega$-автоматами.

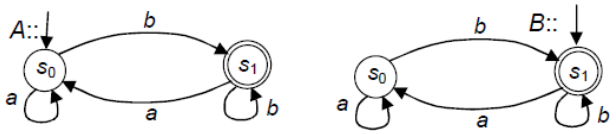


Рис. 3.7. Два эквивалентных автомата Бюхи

**Пример 3.3.** Два автомата, $a$ и $b$, представленные на рис. 3.7, 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 такое множество описывается $\omega$-регулярным выражением $(a^*b)^\omega$. Точно те же $\omega$-слова допускает и автомат $b$, если его рассматривать как автомат Бюхи. Как конечный автомат, $a$ допускает любую конечную цепочку из символов $a$ и $b$, оканчивающуюся на $b$. Все эти цепочки 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, как автоматы Бюхи, $a$ и $b$ эквивалентны, как конечные автоматы на конечных словах они различаются. Заметим, что автоматы Бюхи $a$ и $b$ на рис. 3.7 допускают $\omega$-слова, которые могут включать как конечное, так и бесконечное число символов $a$ наряду с бесконечным числом символов $b$.

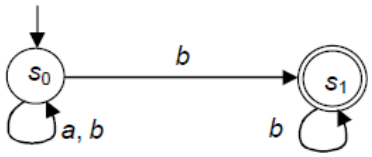


Рис. 3.8. Автомат Бюхи, допускающий цепочки с конечным числом вхождений $a$ и бесконечным числом вхождений $b$

**Пример 3.4.** На рис. 3.8 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 $L = \{(a + b)^* b^\omega\}$. Этот автомат недетерминированный. Можно показать, что не существует детерминированного автомата Бюхи, допускающего этот язык. Таким образом, распознающая мощность недетерминированных автоматов Бюхи строго больше, чем детерминированных.

Пример 3.4 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.

Итак, автомат Бюхи допускает те бесконечные цепочки из символов алфавита $\Sigma$, которые заставляют его пройти последовательность состояний, в которой бесконечно часто встречается хотя бы одно состояние из множества допускающих состояний. Это значит, что автомат Бюхи допускает хотя бы одно $\omega$-слово, если и только если существует достижимый из начального состояния цикл, проходящий через какое-либо его финальное состояние (рис. 3.9).

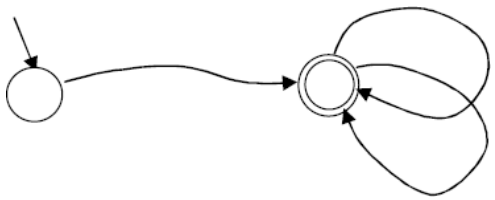

Рис. 3.9. Цикл, проходящий через финальное состояние автомата Бюхи на допустимом вычислении

**Пример 3.5.** 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 пересекающемся с ним западном направлении. Пусть LTL-формула $\varphi$ выражает требование запрещения одновременного движения по двум пересекающимся направлениям: $\varphi = \neg\mathcal{F}(N\_go \wedge W\_go)$.

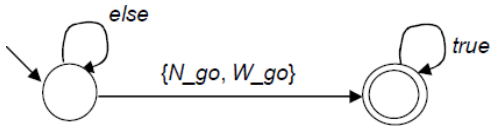


Рис. 3.10. Автомат Бюхи, допускающий все вычисления, на которых выполняется формула $\mathcal{F}(N\_go \wedge W\_go)$

Автомат Бюхи $B_{\neg\varphi}$, допускающий все ошибочные цепочки, удовлетворяющие формуле $\neg\varphi = \mathcal{F}(N\_go \wedge W\_go)$, представлен на рис. 3.10. Поставим этот автомат рядом с автоматом Бюхи $B_M$, построенным по структуре Крипке $M$, так, чтобы он перехватывал все входы и выходы $B_M$. Тогда $B_{\neg\varphi}$ будет выступать как контрольный автомат, который будет работать совместно и синхронно с $B_M$. Когда в $B_M$ встретится переход, разрешающий одновременное движение по пересекающимся направлениям, $B_{\neg\varphi}$ перейдет в ошибочное принимающее состояние и останется в нем навсегда.

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.

## 3.5. ОПЕРАЦИИ НАД АВТОМАТАМИ БЮХИ

Бюхи-распознаваемые $\omega$-языки замкнуты относительно обычных теоретико-множественных операций.

*Объединение двух $\omega$-языков* $L_A$ и $L_B$, распознаваемых соответственно автоматами Бюхи $a$ и $b$, распознает автомат Бюхи, 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 не связанных между собой автомата Бюхи, стоящие рядом, и входная

цепочка переводит из одного состояния в другое либо только один, либо только другой автомат.

**Пример 3.6.** На рис. 3.11 построены автоматы Бюхи $a$ и $b$, распознающие языки, цепочки которого содержат конечное число вхождений $b$ и бесконечное число вхождений $a$ ($L_A$), либо наоборот ($L_B$). Эти два автомата можно считать одним *недетерминированным* автоматом, допускающим объединение языков $L_A$и $L_B$. Два начальных состояния этих автоматов можно объединить в одно.

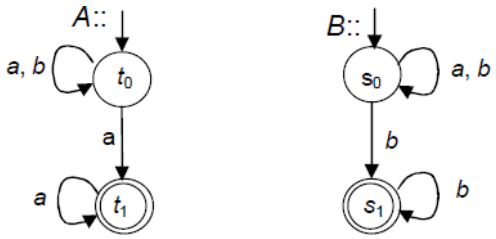


Рис. 3.11. Автоматы Бюхи, допускающие цепочки с конечным числом вхождений одной буквы и бесконечным числом вхождений другой

*Дополнение языка* $L_A$, допускаемого автоматом Бюхи $a$, это язык $\Sigma^\omega \backslash L_A$. Оказывается, что существует автомат Бюхи, допускающий язык $\Sigma^\omega \backslash L_A$. Он строится по исходному автомату $a$ с помощью весьма непростого алгоритма. Результирующий автомат может быть экпоненциально сложнее исходного.

*Пересечение двух* $\omega$*-языков* $L_A$ и $L_B$, допускаемых автоматами Бюхи $a$ и $b$, также допускается автоматом Бюхи. Для построения такого автомата строится синхронная композиция $A \otimes B$ автоматов $a$ и $b$, т. е. автоматы $a$ и $b$, которые работают синхронно, оба одновременно принимают очередной входной сигнал, и каждый из них выполняет свой переход под воздействием этого сигнала.

Для конечных автоматов принимающим состоянием их синхронной композиции является такая пара состояний компонентных автоматов, в которой обе компоненты являются допускающими состояниями соответствующих автоматов (см. пример 3.2 и рис. 3.5). Для автоматов Бюхи ситуация сложнее. $\omega$-Слово допускается автоматом Бюхи, если бесконечная последовательность состояний, которые автомат проходит при приеме этого слова, содержит бесконечное число допускающих состояний. Цепочка должна приниматься автоматом $A\otimes$

$B$, если она принимается как автоматом $a$, так и автоматом $b$. Но нет гарантий, что эта цепочка будет проходить допускающие состояния и в $a$, и в $b$ *одновременно*.

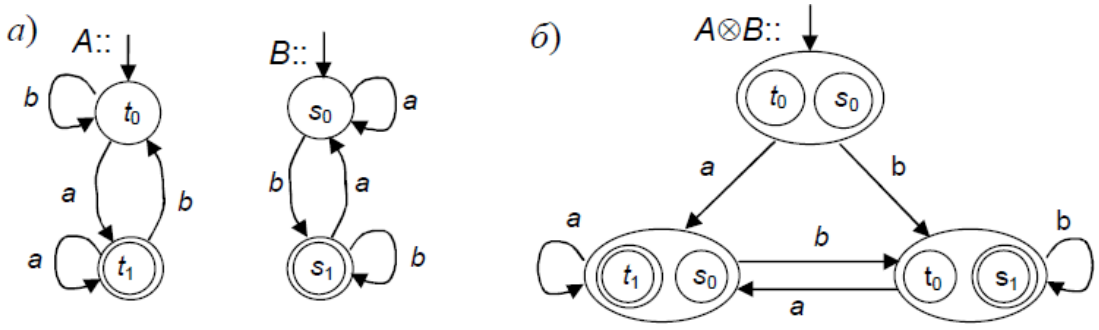


Рис. 3.12. Автоматы Бюхи (а) и их синхронная композиция (б)

**Пример 3.7.** На рис. 3.12 приведен такой пример. Два автомата, $a$ и $b$, допускают $\omega$-языки $L_A$ и $L_B$ над словарем $\{a, b\}$. Язык $L_A$ состоит из цепочек с бесконечным числом вхождений $a$ и произвольным числом вхождений $b$, язык $L_B$ состоит из цепочек с бесконечным числом вхождений $b$ и произвольным числом вхождений $a$. Синхронная композиция этих автоматов (см. рис. 3.12, *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, они недостижимы. Но в то же время, например, цепочка $(ab)^\omega$, которая включает бесконечное число вхождений и $a$, и $b$, проходит допускающие состояния $a$ и $b$ попеременно, одно за другим, и каждое из них проходит бесконечное число раз. Как определить допустимость цепочки в синхронной композиции автоматов Бюхи?

Введем понятие обобщенного автомата Бюхи.

**Определение 3.5.** *Обобщенный автомат Бюхи.* Этот автомат отличается от обычного автомата Бюхи тем, как в нем определяется множество допускающих состояний. Множество $F$ обобщенного автомата Бюхи состоит из конечного числа подмножеств состояний, $F = \{F_1, F_2, \ldots, F_k\}$. $\omega$-Слово допускается обобщенным автоматом Бюхи, если цепочка состояний $\sigma$, которые автомат проходит при приеме этого слова, содержит бесконечное число состояний из каждого множества $F_i \in F$, т. е. $(\forall F_i \in F) inf(\sigma) \cap F_i \neq \emptyset$.

Для примера 3.7 (см. рис. 3.12, *б*) множество $F$ обобщенного

автомата Бюхи состоит из двух одноэлементных подмножеств: $F_1 = \{\langle t_1, s_0\rangle\}, F_2 = \{\langle t_0, s_1\rangle\}$.

Существует простой метод построения обычного автомата Бюхи, эквивалентного обобщенному автомату Бюхи. Идея построения такого автомата состоит в том, что строится $k$ копий обобщенного автомата Бюхи. В $i$-й копии состояния маркируются целым числом $i$. Если состояние $s$ не является допускающим в $i$-й копии, то переходы из него не изменяются. Все переходы из каждого допускающего состояния $i$-й копии направляются в соответствующие состояния копии $i \mod k + 1$.

**Теорема 3.2.** *Для обобщенного автомата Бюхи существует эквивалентный ему обычный автомат Бюхи.*

□. Пусть $A = (S, \Sigma, S_0, \delta, F)$, где $F = \{F_1, F_2, \ldots, F_k\}$ — обобщенный автомат Бюхи. Обычный автомат Бюхи $A' = (S', \Sigma', S'_0, \delta', F')$, эквивалентный автомату $a$, строится так:

1. строится столько копий обобщенного автомата $a$, сколько подмножеств допускающих состояний содержит $a$. Состояния каждой копии снабжаются индексами: $S' = S \times \{1, \ldots, k\}$;
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 принимающих состояний, относившихся к принимающему множеству $F_i$, направляются в состояния следующей по номеру копии, т. е. $\delta'$ определяется так:

$$\begin{array}{rl} \text{если } q \in \delta(p, a), \text{ то } \langle q', j\rangle \in \delta'(\langle p', i\rangle, a), & \text{при чем} \\ j = i, & \text{если } p \notin F_i, \\ j = i \mod k + 1, & \text{если } p \in F_i \end{array}$$

   Таким образом, в переходах автомата $A'$ получается цикл, проходящий через все копии $a$, и каждый цикл содержит допускающие состояния из каждого допускающего множества $F_i$;
4. $F' = F_1 \times \{1\}$ — допускающими состояниями нового автомата являются только допускающие состояния копии 1 из множества $F_1$.

Если под воздействием входной цепочки $w$ автомат $A'$ проходит допускающие состояния нулевой копии бесконечно часто, то он проходит и все состояния из каждого $F_i$ бесконечно часто в обобщенном автомате Бюхи $a$, что и требуется для того, чтобы цепочка $w$ являлась допустимой. ■

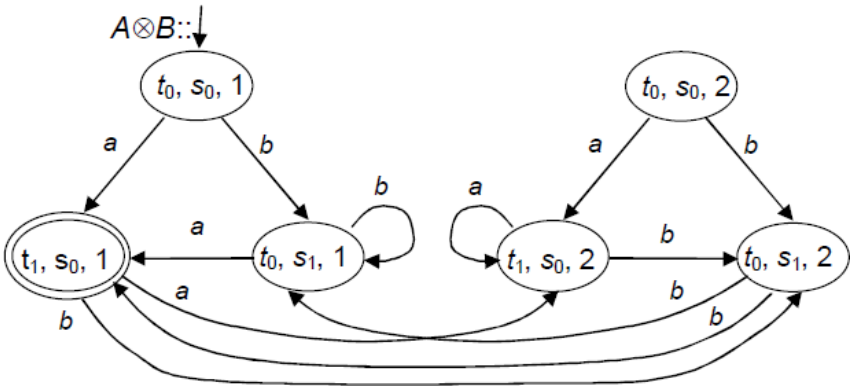


Рис. 3.13. От обобщенного автомата Бюхи к автомату Бюхи

**Пример 3.8.** По обобщенному автомат Бюхи $A \otimes B$ (см. рис. 3.12, *б*) по вышеприведенному алгоритму построен эквивалентный обычный автомат Бюхи (рис. 3.13). Этот автомат Бюхи допускает все возможные цепочки с бесконечным числом вхождений как $a$, так и $b$.

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 в себя 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* $\omega$*-слово*.

Обобщенный автомат Бюхи допускает непустой язык, если существует достижимая из начального состояния непустая сильно связная компонента, имеющая пересечение с каждым множеством $F_i$ допускающих состояний.

**Пример 3.9.** В автомате Бюхи на рис. 3.13 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 $\omega$-слова (т. е. непустой $\omega$-язык).

**Пример 3.10.** 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. 3.11. На рис. 3.14, *а* построена их синхронная композиция — обобщенный автомат Бюхи с двумя множествами допускающих состояний $F = \{F_1, F_2\}$, где $F_1 = \{\langle t_1, s_0\rangle\}, F_2 = \{\langle t_0, s_1\rangle\}$. По этому

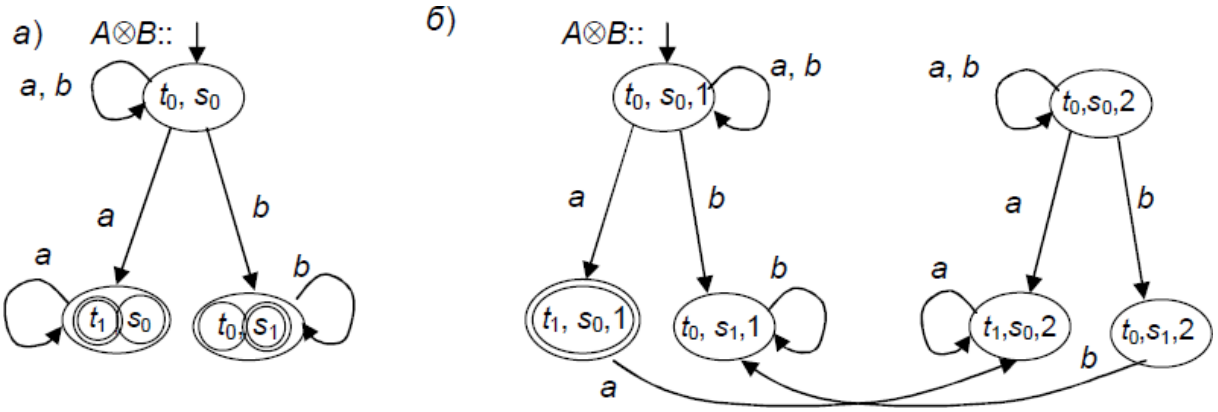


Рис. 3.14. Обобщенный автомат Бюхи синхронная композиция автоматов $a$ и $b$ (см. рис. 3.11) *(а)* и эквивалентный автомат Бюхи *(б)*

обобщенному автомату Бюхи легко строится обычный автомат Бюхи (рис. 3.14, *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 такие цепочки конечные, они никаким автоматом Бюхи не допускаются.

## 3.6. АВТОМАТЫ БЮХИ И LTL-ФОРМУЛЫ

Конечные автоматы можно считать формальными системами, задающими языки, т. е. некоторые подмножества (обычно бесконечные) слов в его входном алфавите. Например, автомат $a$, представленный на рис. 3.7 допускает (и, следовательно, задает) все цепочки из $a$ и $b$, оканчивающиеся на $b$, если его рассматривать как конечный автомат. Все такие цепочки определяются регулярным выражением $(a + b)^*b$.

Каждый автомат Бюхи также допускает (и, следовательно, задает) некоторое множество бесконечных цепочек ($\omega$-слов), составленных из символов его входного алфавита $\Sigma$. Все такие цепочки определяют его циклическое прохождение через одно из допускающих состояний (или через несколько допускающих состояний для обобщенного автомата Бюхи). Иными словами, автомат Бюхи задает $\omega$-язык.

Формулы LTL интерпретируются на вычислениях структуры Крипке. Каждое вычисление структуры Крипке — это бесконечная последовательность состояний. Ясно, однако, что вычисление характеризуют не названия состояний. Вычисление характеризует

цепочка подмножеств тех атомарных предикатов структуры Крипке, которые истинны в проходимых состояниях, т. е. траекторией. Каждая траектория структуры Крипке — это бесконечное слово в алфавите $2^{AP}$. Например, если $AP = \{q, r\}$, то $2^{AP} = \{\emptyset, \{p\}, \{q\}, \{q, r\}\}$.

Если $\sigma = s_0 s_1 s_2 s_3 \ldots$ — вычисление структуры Крипке, то $L(\sigma)$ будем обозначать $\omega$-слово $L(s_0)L(s_1)L(s_2)L(s_3)\ldots$ Например, траектория, соответствующая вычислению на рис. 2.8 — это бесконечная цепочка $\{p, q\}\{q, r\}\{r\}\{r\}\{r\}\ldots$ Все траектории структуры Крипке рис. 2.12, *б*) — это бесконечное множество $\omega$-слов (т. е. $\omega$-язык), примерами которых являются:

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

$\omega$-Слово $\pi$ называется моделью LTL формулы $\varphi$, если $\pi \models \varphi$. Например, $\omega$-слово $\pi = \{p\}\{\}\{p\}\{q\}\{\}^{\omega}$ является моделью формулы $p \wedge (\neg q \mathcal{U}\, q)$.

Любую формулу $\varphi$ линейной темпоральной логики можно считать описанием (спецификацией) $\omega$-языка, все цепочки которого удовлетворяют этой формуле.

Зададимся вопросом: можно ли задать автоматом Бюхи те траектории вычисления, которые описываются заданной LTL формулой $\varphi$?

Во-первых, рассмотрим, какой алфавит должен быть у такого автомата. Пусть, например, формула $\varphi$ есть $\mathcal{F}p$, т. е. эта формула логики LTL описывает такие бесконечные траектории, в которых когда-нибудь в будущем встретится $p$. Автомат Бюхи, допускающий 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 $\Sigma$ автомата Бюхи должно быть еще что-то (например, символ, показывающий отсутствие символа $p$). Входным алфавитом $\Sigma$ автомата Бюхи, допускающего $\omega$-слова, удовлетворяющие формуле $\varphi$ LTL с множеством атомарных предикатов $AP$, является $2^{AP}$. Для формулы $\mathcal{F}p$ такой автомат будет иметь входной алфавит $\{\emptyset, \{p\}\}$.

На рис. 3.10 показан контрольный автомат Бюхи — watchdog, отслеживающий корректность системы управления перекрестком, в качестве одного из символов входного языка имеет подмножество $\{N\_go, W\_go\}$ двух атомарных предикатов, которые не должны быть

истинны одновременно ни в каком состоянии системы управления перекрестком.

Автоматы Бюхи, допускающие $\omega$-языки, заданные некоторыми LTL-формулами, показаны на рис. 3.15.

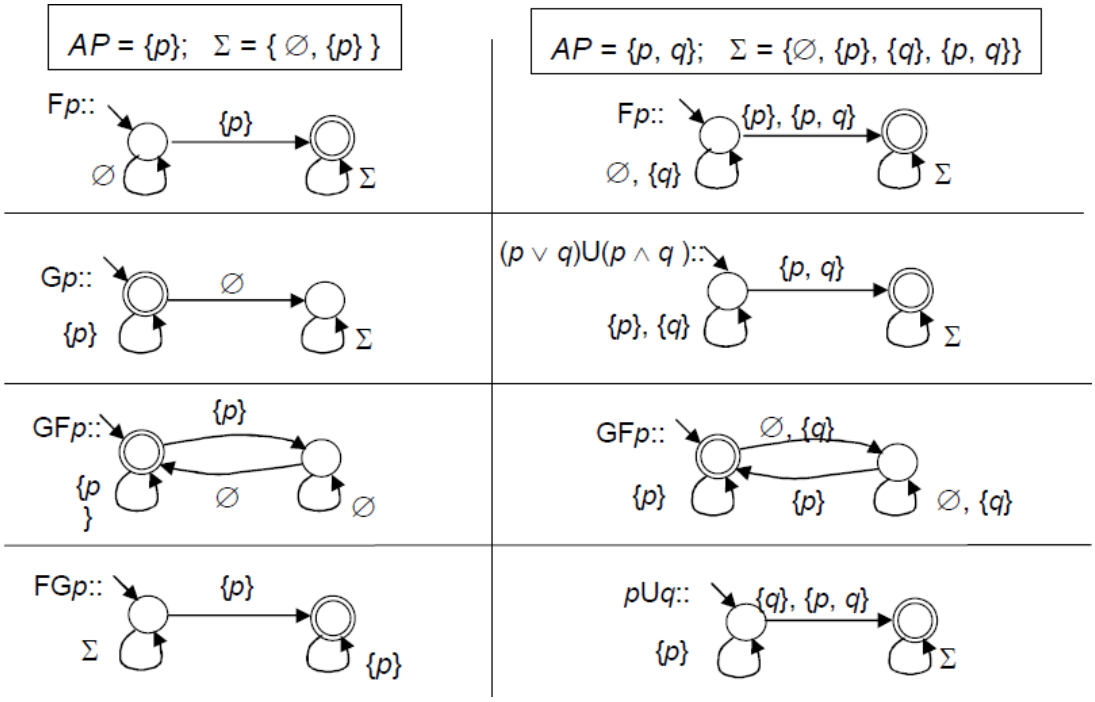


Рис. 3.15. Автоматы Бюхи для некоторых формул LTL ($\Sigma$ обозначает любой символ из $\Sigma$)

Существует важный результат, связывающий формулы линейной темпоральной логики и автоматы Бюхи:

**Теорема 3.3.** *Для любой LTL формулы $\varphi$ существует автомат Бюхи, допускающий все $\omega$-слова, на которых выполняется $\varphi$, и только их.*

Для некоторых автоматов Бюхи не существует LTL-формулы, которая задает все $\omega$-слова, распознаваемые этим автоматом. Иными словами, автоматы Бюхи более выразительны, чем LTL-формулы.

**Теорема 3.4.** *Существуют автоматы Бюхи, $\omega$-языки которых не могут быть заданы никакой LTL формулой.*

В качестве примера приведем вычисления, удовлетворяющие требованию: *$p$ истинно на каждом четном шаге*. Они описываются $\omega$-регулярным выражением $((\{p\} + \emptyset)\{p\})^\omega$ и определяются автоматом Бюхи (см. рис. 3.16). Оказывается, что LTL-формулой это свойство выразить нельзя.

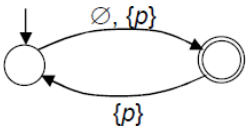


Рис. 3.16. Автомат Бюхи, допускающий вычисления, в которых на каждом четном шаге выполняется $p$

## 3.7. СТРУКТУРЫ КРИПКЕ И АВТОМАТЫ БЮХИ

Структура Крипке задает траектории — множество бесконечных цепочек над алфавитом $2^{AP}$, т. е. символами цепочек являются множества атомарных предикатов. По заданной структуре Крипке $M$ автомат Бюхи $A_M$, допускающий все траектории $M$, строится по следующему алгоритму:

- каждый переход в $A_M$, ведущий из состояния $s$, помечается подмножеством атомарных предикатов, истинных в $s$, и
- каждое состояние $A_M$ является допускающим — все траектории в этом автомате Бюхи допускаемые $\omega$-слова.

Пусть структура Крипке $M$ задана так: $M = (S_M, S_{0M}, R, AP, L)$. Соответствующий автомат Бюхи $A_M = (S_A, \Sigma, S_{0A}, \delta, F)$, который допускает любые траектории структуры Крипке $M$, определим следующим образом:

- $S_A = S$ — состояния $M$ не изменяются;
- $S_{0A} = S_0$ — множество начальных состояний $M$ не изменяется;
- $\Sigma = 2^{AP}$ — алфавит автомата $A_M$ составляют все подмножества атомарных $S_{0A}$;
- $F = S_A$ — все состояния автомата $A_M$ являются допускающими;
- $(s, a, s') \in \delta \quad iff \quad a \in L(s)$ и $(s, s') \in R$ — все переходы, ведущие из состояния $s$, помечаются множеством $L(s)$ атомарных предикатов, истинных в этом состоянии.

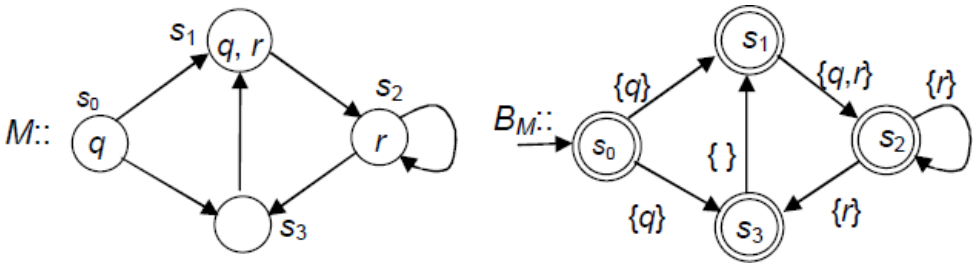


Рис. 3.17. Автомат Бюхи $B_M$, построенный по структуре Крипке $M$

На рис. 3.17 по структуре Крипке (см. рис. 3.1, $a$) построен автомат Бюхи, допускающий все те траектории, которые задаются данной структурой Крипке.

## 3.8. ПРОВЕРКА МОДЕЛИ

Все шаги алгоритма проверки модели для формул LTL (см. рис. 3.6) теперь понятны: по заданной структуре Крипке $M$ строится автомат Бюхи $B_M$, допускающий все траектории, которые возможны в $M$, по LTL-формуле $\varphi$ строится контрольный автомат Бюхи $B_{\neg\varphi}$, допускающий все $\omega$-слова, на которых выполняется $\neg\varphi$, 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 $\varphi$.

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 допускающей является вторая компонента.

**Пример 3.11.** Проверим, выполняется ли свойство $\varphi = \mathcal{GF}(r \Rightarrow q)$ на структуре Крипке $M$ (рис. 3.17, $a$). Очевидно, $\neg\varphi = \neg(\mathcal{GF}(r \Rightarrow q)) = \mathcal{FG}(\neg(r \Rightarrow q)) = \mathcal{FG}(r \wedge \neg q)$.

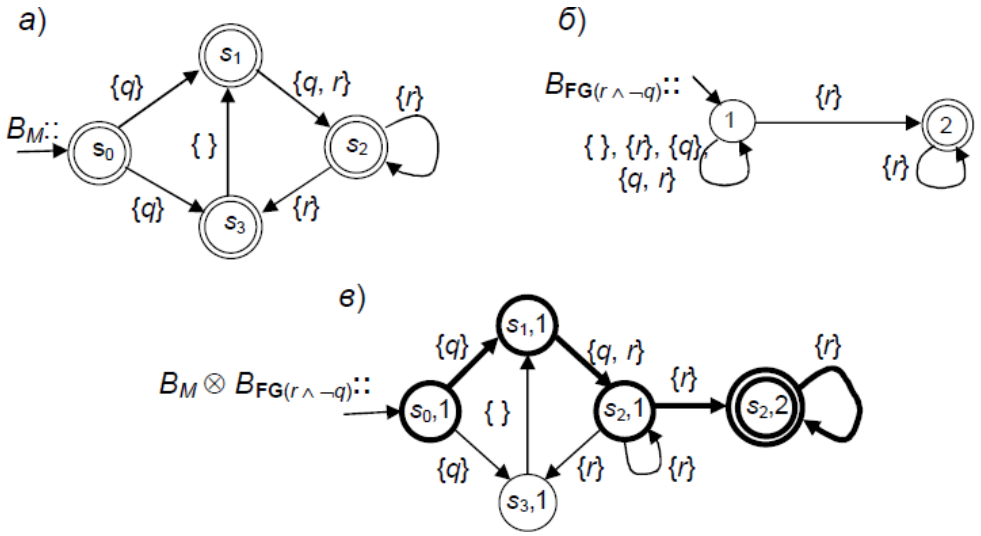


Рис. 3.18. Контрпример в структуре Крипке

Автомат Бюхи $B_M$, построенный по структуре Крипке $M$, представлен на рис. 3.17, *б*. На рис. 3.18, *а* этот автомат для удобства показан повторно, а на рис. 3.18, *б*) представлен автомат Бюхи $B_{\neg\varphi}$ допускающий все вычисления, удовлетворяющие формуле $\neg\varphi = \mathcal{FG}(r \wedge \neg q)$. Синхронная композиция этих двух автоматов представлена на рис. 3.18, *в*. Найденный контрпример выделен на рис. 3.18, *в*. Он показывает одно из вычислений, на которых формула $\varphi = \mathcal{GF}(r \Rightarrow q)$ для исходной структуры Крипке не выполняется. Это вычисление $s_0 s_1 s_2 s_2 s_2 \ldots$ Действительно, на таком вычислении не будет бесконечно повторяться формула $r \Rightarrow q$, что определено формулой $\varphi$.

## 3.9. ПОСТРОЕНИЕ АВТОМАТА БЮХИ ПО ФОРМУЛЕ LTL

Существует несколько алгоритмов построения автомата Бюхи, допускающего $\omega$-слова, на которых выполняется формула $\varphi$ логики LTL, и почти все они весьма сложные. Рассмотрим один из алгоритмов, который предложен в [34] и усовершенствован в [10]. Этот алгоритм концептуально прост, однако он дает в результате автомат Бюхи с экспоненциальным числом состояний относительно длины формулы $\varphi$.

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 $\wedge$, а также темпоральные операторы $\mathcal{X}, \mathcal{U}$ . Пусть такие формулы задаются грамматикой:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathcal{X}\varphi \mid \varphi \mathcal{U} \varphi.$$

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 входные цепочки, удовлетворяющие формуле $\varphi$.

**Пример 3.12.** Рассмотрим формулу $\varphi = (p \vee q)\mathcal{U}(p \wedge q)$. Эта формула LTL построена над алфавитом $AP = \{p, q\}$ и определяет все те вычисления, на которых когда-то в будущем встретится и $p$, и $q$, а до этого в состояниях вычисления будут истинны либо $p$, либо $q$. Формула $\varphi$ определена над последовательностями, построенными из символов алфавита $2^{AP}$, т. е. алфавит искомого автомата Бюхи есть $\Sigma = \{\emptyset, \{p\}, \{q\}, \{p, q\}\}$.

Формулы LTL интерпретируются на вычислениях — бесконечных цепочках состояний структуры Крипке (например, на цепочке $\sigma = s_0 s_1 s_2 \ldots$ (см. рис. 3.19, *а*), в каждом состоянии которых определены множества истинных в этом состоянии атомарных предикатов из множества $AP$. Последовательность подмножеств атомарных предикатов, соответствующая вычислению $\sigma$, определяется функцией пометок $L$. Вычислению $\sigma$ соответствует цепочка множеств атомарных предикатов:

$$L(\sigma) = \{p\}\{q\}\{p, q\}\{q\}\{\}\{p\}\ldots.$$

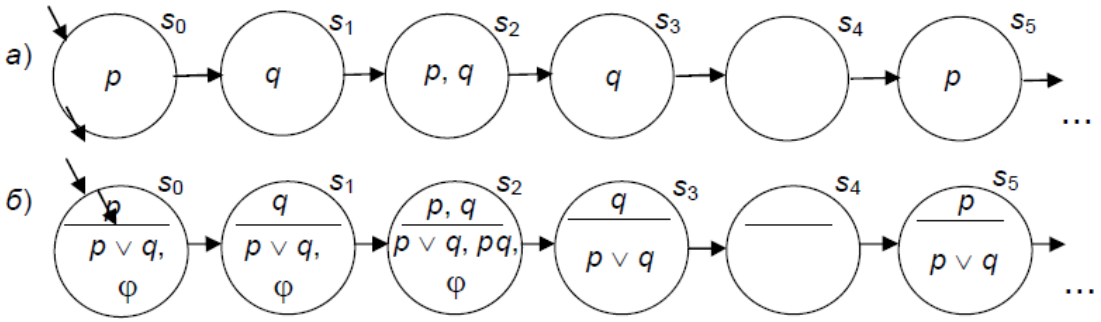


Рис. 3.19. Вычисление и множества подформул формулы $\varphi = (p \vee q)\,\mathcal{U}\,(p \wedge q)$, выполняющихся в его состояниях

### 3.9.1. Разметка состояний вычисления формулами LTL

Рассматриваемый метод построения автомата Бюхи по заданной LTL формуле основан на простой идее, близкой идее разметки состояний структуры Крипке при проверке выполнения CTL формул. Следуя данной идее, рассмотрим сначала наивный алгоритм определения того, выполняется ли формула LTL на заданном бесконечном вычислении. Например, проверим, будет ли формула $\varphi = (p \vee q)\,\mathcal{U}\,(p \wedge q)$ выполняться на вычислении, представленном на рис. 3.19, *а*.

Будем помечать состояния вычисления теми подформулами формулы $\varphi$, которые выполняются в этом состоянии. Начиная с атомарных предикатов, истинных в данном состоянии, добавим в состояние все более и более сложные подформулы формулы $\varphi$, которые истинны в данном состоянии.

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 требует анализа подформул в последующих состояниях конкретного вычисления. Например, на вычислении $\sigma = s_0 s_1 s_2 \ldots$ (см. рис. 3.19, *а*) в состоянии $s_1$ формула $\varphi = (p \vee q)\,\mathcal{U}\,(p \wedge q)$ выполняется, а в состоянии $s_3$, также помеченном только атомарным предикатом $q$, эта формула не выполняется. Но обе такие разметки допустимы. Для всех состояний вычисления $\sigma$ такая разметка представлена на рис. 3.19, *б*. Видно, что на вычислении $\sigma$ формула $\varphi = (p \vee q)\,\mathcal{U}\,(p \wedge q)$ выполняется.

Проверить выполнимость произвольных формул LTL этим способом невозможно, поскольку вычисления бесконечны. Однако 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 $\varphi$, 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 $\varphi$ определяет так называемое *замыкание формулы* $\varphi$ (обозначается $cl(\varphi)$, от *англ.* closure). Фактически $cl(\varphi)$ — это множество всех подформул формулы $\varphi$, которые могут появиться в синтаксическом дереве при анализе формулы $\varphi$. Для формулы $\varphi = (p \vee q)\,\mathcal{U}\,(p \wedge q)$ таких подформул пять:

$$cl((p \vee q)\,\mathcal{U}\,(p \wedge q)) = \{p, q, p \vee q, p \wedge q, \varphi\}.$$

Конечно, не все комбинации подформул анализируемой формулы $\varphi$ могут помечать одно и то же состояние любого вычисления. Подформулы, которые маркируют состояние, должны быть непротиворечивыми, согласованными. Например, рассмотрим рис. 3.19, *б*. Состояние $s_5$ помечено формулами $\{p, p \vee q\}$ и очевидно, что оно не может быть помечено 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 $p \vee q$, ни $p \wedge q$, ни сама анализируемая формула $\varphi$: в этом состоянии ни одна из подформул формулы $\varphi$ не выполняется. Состояние $s_2$ помечено формулами $\{p, p \vee q, (p \vee q)\,\mathcal{U}\,(p \wedge q)\}$ — все эти подформулы

выполняются в $s_2$ (см. рис. 3.19, *б*).

Легко определить формальные правила построения всех максимальных согласованных множеств подформул анализируемой формулы $\varphi$, которыми могут быть помечены состояния вычислений, *допускающих формулу* $\varphi$. Назовем такие множества атомами.

**Определение 3.6.** *Атомы темпоральной формулы*. Множество подформул произвольной темпоральной формулы $\varphi$ назовем согласованным, если существует вычисление, на котором все они могут выполниться. Любое максимальное множество согласованных подформул формулы $\varphi$ будем называть атомом $\varphi$.

Атомы $A_\varphi$ формулы $\varphi$ можно считать максимальными множествами LTL-формул из $cl(\varphi)$, которые могут помечать состояния вычислений.

Атомы $A_\varphi$ формулы $\varphi$ строятся по правилам, следующим из семантики логики высказываний и семантики LTL. Для любых $\psi, \varphi_1, \varphi_2, \varphi_1 \vee \varphi_2, \varphi_1 \mathcal{U} \varphi_2 \in cl(\varphi)$ должны выполняться правила:

$$
\begin{array}{lllll}
R1. & \psi \in A_\varphi & \Leftrightarrow & \neg\psi \notin A_\varphi; \\
R2. & \varphi_1 \vee \varphi_2 \in A_\varphi & \Leftrightarrow & \varphi_1 \in A_\varphi \text{ или } \varphi_2 \in A_\varphi; \\
R3. & \varphi_1 \mathcal{U} \varphi_2 \in A_\varphi \text{ и } \varphi_2 \notin A_\varphi & \Rightarrow & \varphi_1 \in A_\varphi; \\
R4. & \varphi_2 \in A_\varphi & \Rightarrow & \varphi_1 \mathcal{U} \varphi_2 \in A_\varphi.
\end{array}
$$

Эти правила легко можно дополнить для выводимых логических операций и темпоральных операторов. Например, для любых $\psi, \varphi_1, \varphi_2, \varphi_1 \wedge \varphi_2, \varphi_1 \Rightarrow \varphi_2, \mathcal{F}\psi, \mathcal{G}\psi \in cl(\varphi)$ должны выполняться правила:

$$
\begin{array}{lllll}
R5. & \varphi_1 \wedge \varphi_2 \in A_\varphi & \Leftrightarrow & \varphi_1 \in A_\varphi \text{ и } \varphi_2 \in A_\varphi; \\
R6. & \varphi_1 \Rightarrow \varphi_2 \in A_\varphi & \Leftrightarrow & \neg\varphi_1 \in A_\varphi \text{ или } \varphi_2 \in A_\varphi; \\
R7. & \psi \in A_\varphi & \Rightarrow & \mathcal{F}\psi \in A_\varphi; \\
R8. & \mathcal{G}\psi \in A_\varphi & \Rightarrow & \psi \in A_\varphi.
\end{array}
$$

Правила $R5$ и $R6$ очевидны. Справедливость $R7$ вытекает из $R4$, поскольку $\mathcal{F}\psi = true\,\mathcal{U}\,\psi$. Правило $R8$, в свою очередь, вытекает из $R1$ и $R7$. Действительно, из правила $\psi \in A_\varphi \Rightarrow \mathcal{F}\psi \in A_\varphi$ имеем $\neg(\mathcal{F}\psi \in A_\varphi) \Rightarrow \neg(\psi \in A_\varphi)$, откуда по правилу $R1 : \neg\mathcal{F}\psi \in A_\varphi \Rightarrow \neg\psi \in A_\varphi$, или, что то же $\neg\mathcal{F}\neg\psi \in A_\varphi \Rightarrow \neg\neg\psi \in A_\varphi$. Теперь правило $R8$ вытекает из определения $\mathcal{G}\psi = \neg\mathcal{F}\neg\psi$.

Если $\neg\psi \in A_\varphi$, то эту формулу $\neg\psi$ мы указывать в атоме не будем.

Правила $R1 - R8$ определяются очевидными правилами согласования двух формул $\psi_1, \psi_2 \in cl(\varphi)$ для одного и того же произвольного состояния $s_i$ любого вычисления: если формула $\psi_1$ выполняется на вычислении, начинающемся с $s_i$, то должна ли на этом же вычислении выполняться формула $\psi_3$? Их можно назвать *локальными ограничениями на выполнимость формул LTL*.

**Пример 3.13.** Построим атомы для нескольких простых темпоральных формул.

1. Формула $\mathcal{F}p$ имеет три атома:

$$A_1 = \emptyset,$$
$$A_2 = \{\mathcal{F}p\},$$
$$A_3 = \{p, \mathcal{F}p\}.$$

   Среди атомов здесь отсутствует подмножество множества $cl(\mathcal{F}p) = \{p, \mathcal{F}p\}$, состоящее из одной формулы $p$. Множество формул $\{p\}$ является несогласованным: оно нарушает правило $R7$.

2. У формулы $\mathcal{GF}p$ пять атомов:

$$A_1 = \emptyset,$$
$$A_2 = \{\mathcal{F}p\},$$
$$A_3 = \{p, \mathcal{F}p\},$$
$$A_4 = \{\mathcal{F}p, \mathcal{GF}p\},$$
$$A_5 = \{p, \mathcal{F}p, \mathcal{GF}p\}.$$

   Эти атомы построены по правилам $R7$ и $R8$: $p \in A_\varphi \Rightarrow \mathcal{F}p \in A_\varphi$ и $\mathcal{GF}p \in A_\varphi \Rightarrow \mathcal{F}p \in A_\varphi$.

3. Для формулы $\varphi = (p \vee q)\,\mathcal{U}\,(p \wedge q)$ все множество ее атомов встречается в состояниях вычисления (см. рис. 3.19, *б*):

$$A_1 = \emptyset,$$
$$A_2 = \{p, p \vee q\},$$
$$A_3 = \{p, p \vee q, (p \vee q)\,\mathcal{U}\,(p \wedge q)\},$$
$$A_4 = \{q, p \vee q\},$$
$$A_5 = \{q, p \vee q, (p \vee q)\,\mathcal{U}\,(p \wedge q)\},$$
$$A_6 = \{p, q, p \vee q, p \wedge q, (p \vee q)\,\mathcal{U}\,(p \wedge q)\}.$$

### 3.9.2. Реализация обязательств

Правила, определяющие ограничения на переходах между атомами, следуют из семантики LTL. Это правила выполнимости формул

в *соседних* состояниях любого вычисления $\sigma = s_0 s_1 s_2 s_3 \ldots$ Такие правила можно назвать правилами *реализации обязательств*. Они фактически реализуют обязательства, которые выражены текущим набором подформул в атоме. Например, если текущее состояние вычисления помечено формулой $\mathcal{X}\psi$, то это можно считать обязательством того, что в следующем состоянии такого вычисления обязательно должна выполниться формула $\psi$, или, что то же, что любое следующее состояние должно быть помечено атомом, включающим $\psi$.

Формализуем указанные правила. Пусть $A_\varphi$ и $A'_\varphi$ — атомы, являющиеся соседними состояниями какого-нибудь вычисления, так что из состояния $A_\varphi$ ведет переход в состояние $A'_\varphi$. Тогда правила реализации обязательств имеют вид:

$$
\begin{array}{lll}
C1. & \mathcal{X}\psi \in A_\varphi & \Leftrightarrow \quad \psi \in A'_\varphi; \\
C2. & \varphi_1 \mathcal{U} \varphi_2 \in A_\varphi \text{ и } \varphi_2 \notin A_\varphi & \Rightarrow \quad \varphi_1 \mathcal{U} \varphi_2 \in A'_\varphi; \\
C3. & \varphi_1 \mathcal{U} \varphi_2 \in A'_\varphi \text{ и } \varphi_1 \in A_\varphi & \Rightarrow \quad \varphi_1 \mathcal{U} \varphi_2 \in A_\varphi.
\end{array}
$$

Правило $C1$ следует из семантики формулы $\mathcal{X}\psi$ : $\sigma^i \models \mathcal{X}\psi \Rightarrow \sigma^{i+1} \models \psi$.

Правила $C2$ и $C3$ вытекают из семантики формулы $\varphi_1 \mathcal{U} \varphi_2$, которая выражается рекуррентным определением оператора Until: $\varphi_1 \mathcal{U} \varphi_2 \equiv \varphi_2 \vee X(\varphi_1 \mathcal{U} \varphi_2)$.

Для темпоральных операторов $\mathcal{F}$ и $\mathcal{G}$ из $C1 - C3$ легко выводятся три дополнительных правила:

$$
\begin{array}{lll}
C4. & \mathcal{F}\psi \in A_\varphi \text{ и } \psi \notin A_\varphi & \Rightarrow \quad \mathcal{F}\psi \in A'_\varphi; \\
C5. & \mathcal{F}\psi \in A'_\varphi & \Rightarrow \quad \mathcal{F}\psi \in A_\varphi; \\
C6. & \mathcal{G}\psi \in A_\varphi & \Leftrightarrow \quad \mathcal{G}\psi \in A'_\varphi \text{и } \psi \in A_\varphi.
\end{array}
$$

Эти правила интерпретируются следующим образом. Если в текущем состоянии $s_i$ есть обязательство выполнения формулы $\mathcal{F}\psi$, то это обязательство должно сохраняться во всей цепочке состояний $s_i s_{i+1} s_{i+2} s_{i+3} \ldots$ до тех пор, пока не встретится состояние, в котором выполнится $\psi$. В состояние, помеченное $\mathcal{F}\psi$, не может быть перехода из состояния, не помеченного $\mathcal{F}\psi$. Далее, состояние, в котором есть обязательство выполнения $\mathcal{G}\psi$, должно также иметь обязательство выполнения $\psi$, и любое предыдущее и последующее состояние также должны иметь обязательства выполнения $\mathcal{G}\psi$.

Из правил $C1 - C6$, следует, например, что в любом вычислении за состоянием, помеченным множеством подформул $\{p, p\mathcal{U}q\}$, не может идти состояние, помеченное множеством $\{p, r\}$, как и наоборот.

### 3.9.3. Построение автомата Бюхи по LTL формуле

Автомат Бюхи, допускающий все $\omega$-слова, на которых выполняется формула $\varphi$, может иметь только состояния, помеченнные атомами формулы $\varphi$. Поэтому все атомы формулы $\varphi$, определяемые локальными правилами $R1 - R8$, принимаются за состояния искомого автомата. Переходы этого автомата определяются правилами реализации обязательств $C1 - C6$.

**Определение 3.7.** *Автомат Бюхи* $B_\varphi$. Обобщенный автомат Бюхи $B_\varphi = (Q, 2^{AP}, Q_0, \delta, F)$, допускающий все вычисления, на которых выполняется LTL формула $\varphi$, определяется следующим образом:

- множество $q$ состояний $B_\varphi$ составляют все атомы формулы $\varphi$: $A_i \subseteq cl(\varphi)$,
- алфавит $2^{AP}$ автомата $B_\varphi$ состоит из подмножеств атомарных предикатов, над которыми определена формула $\varphi$,
- множество $Q_0$ начальных состояний $B_\varphi$ составляют те атомы формулы $\varphi$, в которые входит сама формула $\varphi$,
- функция переходов $\delta : Q \times 2^{AP} \to 2^Q$ согласована с семантикой формул LTL и определяется так:
  - для каждого состояния $a$ автомата $B_\varphi$ и множества $A \cap 2^{AP}$ атомарных предикатов из $AP$, включенных в атом $a$, переход $\delta(A, A \cap 2^{AP})$ — это максимальное множество таких атомов $A' \in Q$, которые удовлетворяют правилам реализации обязательств $C1 - C6$:
    - для каждой формулы $\mathcal{X}\psi \in cl(\varphi)$,
    $$\mathcal{X}\psi \in A_\varphi \Leftrightarrow \psi \in A'_\varphi;$$
    - для каждой формулы $\varphi_1 \mathcal{U} \varphi_2 \in cl(\varphi)$,
    $$\varphi_1 \mathcal{U} \varphi_2 \in A_\varphi \Leftrightarrow \big(\varphi_2 \in A_\varphi \text{ или } (\varphi_1 \in A_\varphi \text{ и } \varphi_1 \mathcal{U} \varphi_2 \in A'_\varphi)\big)$$
  - $F = \{\{A \in Q \mid \varphi_i \mathcal{U} \psi_i \notin A \text{ или } \psi_i \in A\} \mid \varphi_i \mathcal{U} \psi_i \in cl(\varphi)\}$: множество $F = \{F_1, \ldots, Fn\}$ подмножеств допускающих состояний обобщенного автомата Бюхи $B_\varphi$ строится так. Каждое подмножество $F_i \in F$ определяется для каждой формулы $\alpha_i = \varphi_i \mathcal{U} \psi_i$, которая встречается в замыкании формулы $\varphi$.
    Множество $F_i$ составляют такие атомы формулы $\varphi$, в которые или не входит сама формула $\alpha_i$, или входит $\psi_i$.

Доказательство того, что построенный автомат допускает в точности все те $\omega$-слова, на которых LTL формула $\varphi$ выполняется, приведено в [10].

Поясним построение обобщенного автомата Бюхи на примерах.

**Пример 3.14.** Для формулы $\varphi = \mathcal{F}p$ недетерминированный обобщенный автомат Бюхи $B_\varphi$, построенный по приведенным выше правилам, представлен на рис. 3.20. Множество его состояний включает в себя три атома: $A_1 = \emptyset, A_2 = \{\mathcal{F}p\}$ и $A_3 = \{p, \mathcal{F}p\}$. Множеством начальных состояний является $\{A_2, A_3\}$ — именно эти состояния помечены формулой $\mathcal{F}p$.

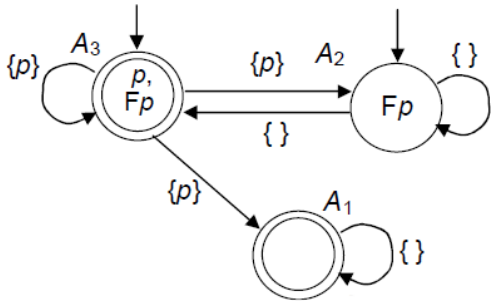


Рис. 3.20. Автомат Бюхи для формулы $\mathcal{F}p$

Множество допускающих состояний здесь одно, поскольку в формуле $\mathcal{F}p$ только одна формула вида $\mathcal{F}p = true\mathcal{U}p$ : $F = \{F_1\}$. Состояние $A_1$ вообще не включает формулу $\mathcal{F}p$, а состояние $A_3$ включает в себя и $\mathcal{F}p$, и $p$, поэтому $F_1 = \{A_1, A_3\}$. Поскольку множество начальных 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 перехода из состояния $A_2$ в состояние $A_1$ (это противоречит правилу $C4$ реализации обязательств), нет и переходов из $A_1$ в $A_2$ и $A_3$ (поскольку это противоречит правилу $C5$).

Легко заметить, что любая цепочка, на которой формула $\mathcal{F}p$ выполняется, этим автоматом допускается. т. е. она проходит бесконечное число раз одно из допускающих состояний. Цепочки, на которых формула $\mathcal{F}p$ не выполняется, этим автоматом не допускаются.

Размер автомата Бюхи, построенного приведенным выше алгоритмом, в некоторых случаях не очень высок. Так, на рис. 3.21 приведен минимальный автомат Бюхи для формулы $\mathcal{GF}p$, построенный

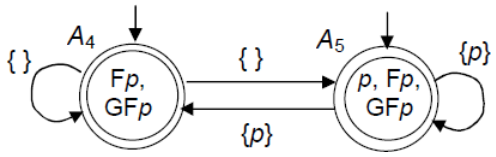


Рис. 3.21. Автомат Бюхи, допускающий все возможные траектории, на которых выполняется формула $\mathcal{GF}p$

с помощью этого алгоритма. Однако, почти всегда построенный автомат Бюхи экспоненциален относительно размера формулы $\varphi$. Верхняя граница сложности такого автомата $2^{O(|\varphi|)}$. Минимальный автомат Бюхи, допускающий тот же язык, может быть значительно меньше. Например, автомат Бюхи, допускающий $\omega$-язык, включающий все цепочки с бесконечным числом вхождений $a$ и конечным числом вхождений $b$, показан на рис. 3.11. Он содержит только два состояния. На рис. 3.15 представлен автомат Бюхи с двумя состояниями, допускающий все цепочки, удовлетворяющие формуле $\varphi = (p \vee q)\,\mathcal{U}\,(p \wedge q)$, для которой приведенный алгоритм дает автомат с шестью состояниями. Существуют автоматы Бюхи, для которых граница сложности $2^{O(|\varphi|)}$ не может быть улучшена.

## 3.10. БИБЛИОГРАФИЧЕСКИЙ КОММЕНТАРИЙ

Теоретико-автоматный подход к проверке выполнения формул LTL предложен в [32]. Как отмечено ранее, существуют разные пути обобщения модели конечного автомата, чтобы он мог допускать бесконечные цепочки. Формальная модель рассмотренного нами автомата Бюхи введена в [12].

Проблема построения автомата Бюхи по заданной формуле LTL вызвала большой интерес, который не пропал и в настоящее время. Существует несколько подходов к ее решению, однако все они в худшем случае дают автомат Бюхи, экспоненциально сложный относительно длины LTL формулы. Приведенный в этом разделе алгоритм построения автомата Бюхи основан на результатах, изложенных в [32] и [10]. О конечных автоматах и распознавании ими языков можно узнать в [5]. Наибольшее распространение среди систем верификации, основанных на описанной здесь технике, получила система Spin ([23]).

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 [9]), разработанный компанией Интел, использует логические и арифметические операции на битовых векторах для описания свойств состояний компьютерного оборудования, регулярные выражения для событий и отношений между этими событиями, хотя основой его остается LTL.

## ЗАДАЧИ

1. Какое свойство структуры Крипке с $n$ состояниями $\{s_1, \ldots, s_n\}$, выражается формулой:

$$\mathtt{E}[\mathcal{F}p1 \wedge \mathcal{G}(p_1 \Rightarrow \mathcal{X}\mathcal{G}\neg p_1) \wedge \ldots \mathcal{F}p_n \wedge \mathcal{G}(p_n \Rightarrow \mathcal{X}\mathcal{G}\neg p_n)]$$

   где $p_1, \ldots, p_n$ — атомарные предикаты, причем предикат $p_k$ истинен только в состоянии $s_k$. Постройте структуру Крипке, удовлетворяющую этому свойству.
2. Какое свойство структуры Крипке выражается формулой:

$$\mathtt{E}[p_0 \wedge \mathcal{X}p_1 \wedge \mathcal{X}\mathcal{X}p_2 \ldots \mathcal{X}^n p_n],$$

   где $p_1, \ldots, p_n$ — атомарные предикаты, причем предикат $p_k$ истинен только в состоянии $s_k$ структуры Крипке.
   Постройте структуру Крипке, удовлетворяющую этому свойству.
3. 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. 3.8). 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 принимающее состояние после конечного числа символов $b$. Покажите, что эту процедуру можно повторить.
4. Постройте формулу LTL, выражающую свойство *состояние, удовлетворяющее $p$, встречается самое большее один раз на каждом вычислении*. Постройте структуру Крипке, на которой эта формула выполняется.

5. Постройте минимальный конечный автомат, распознающий язык $b^*a^*$. Какой язык допускает этот автомат, если его рассматривать как автомат Бюхи?
6. По произвольной заданной структуре Крипке $K$ и формуле $\Phi$ логики LTL проверьте вручную, удовлетворяется ли формула $\Phi$ на данной структуре. Для этого постройте автоматы Бюхи $A_K$ и $B_{\neg\Phi}$ и их синхронную композицию и проверьте, пусто ли пересечение языков $L_K$ и $L_{\neg\Phi}$.
7. Постройте автомат Бюхи, распознающий все цепочки, задаваемые LTL-формулой $\mathcal{G}p$ для входных алфавитов $\Sigma = \{p\}$ и $\Sigma = \{p, q\}$.

## ЗАКЛЮЧЕНИЕ К РАЗДЕЛУ 3

Алгоритм *model checking* для формул LTL принимает на вход структуру Крипке $M$ и формулу $\varphi$ логики LTL. В том случае, когда $M \not\models \varphi$, алгоритм выдает контрпример — одно из вычислений $M$, на котором формула $\varphi$ не выполняется. Такой контрпример является ценной информацией, которая позволяет найти ошибки в разработке, представленной моделью $M$.

Основа этого подхода — формальная модель, а именно конечный автомат Бюхи, позволяющий конечным образом задавать бесконечные языки. Автомат Бюхи отличается от обычного конечного автомата тем, что его семантика определена на $\omega$-словах — бесконечных цепочках символов. $\omega$-Слово допускается автоматом Бюхи, если существует допускающее состояние автомата, которое проходится бесконечное число раз при приеме этого слова.

Следует понимать, что автомат Бюхи — не операционное устройство, которое *действует*, *функционирует* под воздействием входных сигналов, 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 просто абстракция, формальная модель, удобная для задания и анализа $\omega$-языков. С помощью такой модели можно выполнить абстрактные операции, ее анализ позволяет решать важные практические проблемы. Многие проблемы анализа для автоматов Бюхи неразрешимы (например, проверка эквивалентности двух автоматов Бюхи). Но некоторые проблемы разрешимы, и для их решения можно построить алгоритмы.

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 поведения, не удовлетворяющие желаемому свойству (т. е. допускает все неправильные траектории), то, выполнив операцию синхронной композиции над этими моделями, можно построить новый автомат Бюхи $B \otimes K$. По этому новому автомату Бюхи легко установить, возможны ли «некорректные» траектории в будущем поведении нашей системы, выполнив проверку непустоты языка, допускаемого автоматом $B \otimes K$. Именно эта техника и описана в данном разделе.

# 4. ОБЗОР ЯЗЫКА СПЕЦИФИКАЦИИ МОДЕЛЕЙ PROMELA

Язык Promela — абстрактный язык спецификации алгоритмов. Абстрагирование направлено на то, чтобы с помощью минимальных выразительных средств (параллельные процессы и коммуникация с помощью каналов, простейшие типы данных с конечным числом возможных значений и простые управляющие структуры) строить такие абстрактные модели реальных параллельных и распределенных систем, которые легко представляются формальной моделью с конечным числом состояний (структурой Крипке и автоматами Бюхи). Таким образом, Promela является не языком реализации систем (языком программирования), а *языком описания моделей распределенных систем*.

Язык Promela включает в себя примитивы для создания процессов и богатый набор примитивов для межпроцессного взаимодействия. Базовые блоки моделей в Promela составляют асинхронные процессы, каналы сообщений, синхронизирующие утверждения, структурированные данные. В языке также поддерживается спецификация структур недетерминированного управления.

Таким образом, Promela содержит некоторые средства, которые отсутствуют в обычных, широко распространенных языках программирования. Эти средства способствуют конструированию моделей распределенных систем с высоким уровнем абстракции. В то же время в языке отсутствует множество возможностей, которые есть в большинстве языков программирования (например, функции, возвращающие значения, указатели на данные и функции, сложные структуры данных и т. п.). В язык намеренно не включено понятие времени или часов (свойства реального времени не проверяются в системе Spin); отсутствуют операции с плавающей точкой (чтобы ограничиться моделями с конечным числом состояний); ограничено количество вычислительных функций. Подобные ограничения делают относительно сложной реализацию в языке Promela, например, вычисления квадратного корня, но относительно простой реализацию, например, клиент-серверного взаимодействия в сети. Перекос в сторону средств взаимодействия процессов в языке объясняется

направленностью языка Promela на верификацию именно параллельных и распределенных систем, в первую очередь синхронизацию и координацию параллельно протекающих процессов.

На практике именно разработка корректной структуры синхронизации и координации для программного обеспечения параллельных и распределенных систем оказывается гораздо более сложной и чреватой ошибками, чем разработка последовательных вычислений. Логическая верификация взаимодействия в параллельных системах, несмотря на значительную вычислительную сложность, может быть сегодня выполнена на основе новой техники верификации model checking более надежно и тщательно по сравнению с верификацией последовательных вычислительных процедур. Именно эту технику Spin использует при верификации. Отметим, например, что большинство ошибок, выявленных с помощью Spin при верификации бортовой системы управления космического аппарата Deep Space 1, были ошибками, допущенными разработчиками этой системы в силу непредвиденных взаимодействий и перекрытий выполнений параллельных процессов.

## 4.1. ГРАФИЧЕСКАЯ ОБОЛОЧКА XSPIN И МОДЕЛЬ HELLO WORLD

Программное средство Spin можно использовать в командной строке, указывая различные ключи. Однако мы рекомендуем воспользоваться графической оболочкой XSpin. Запустите XSpin.

Основное окно XSpin является простым текстовым редактором (см. рис. 4.1). В меню *Edit* доступны обычные опции для работы с текстом: *Copy*, *Cut*, *Paste*. С помощью меню *View* можно изменять размеры шрифта в основном окне. Под основным окном находится окно с историей команд (`command-log`). В дальнейшем все выполняемые команды будут отображаться в окне команд.

Рассмотрим пример простейшей программы на языке Promela. Наиболее часто встречающимся первым примером в руководствах по языкам программирования является программа, которая печатает строку hello world на терминал пользователя. Традиция началась с руководства для языка программирования C (1978 г.), авторами которого были Керниган и Ритчи. С тех пор этот пример кочует по руководствам языков программирования. Специфицируем эту маленькую программу как модель на языке Promela:

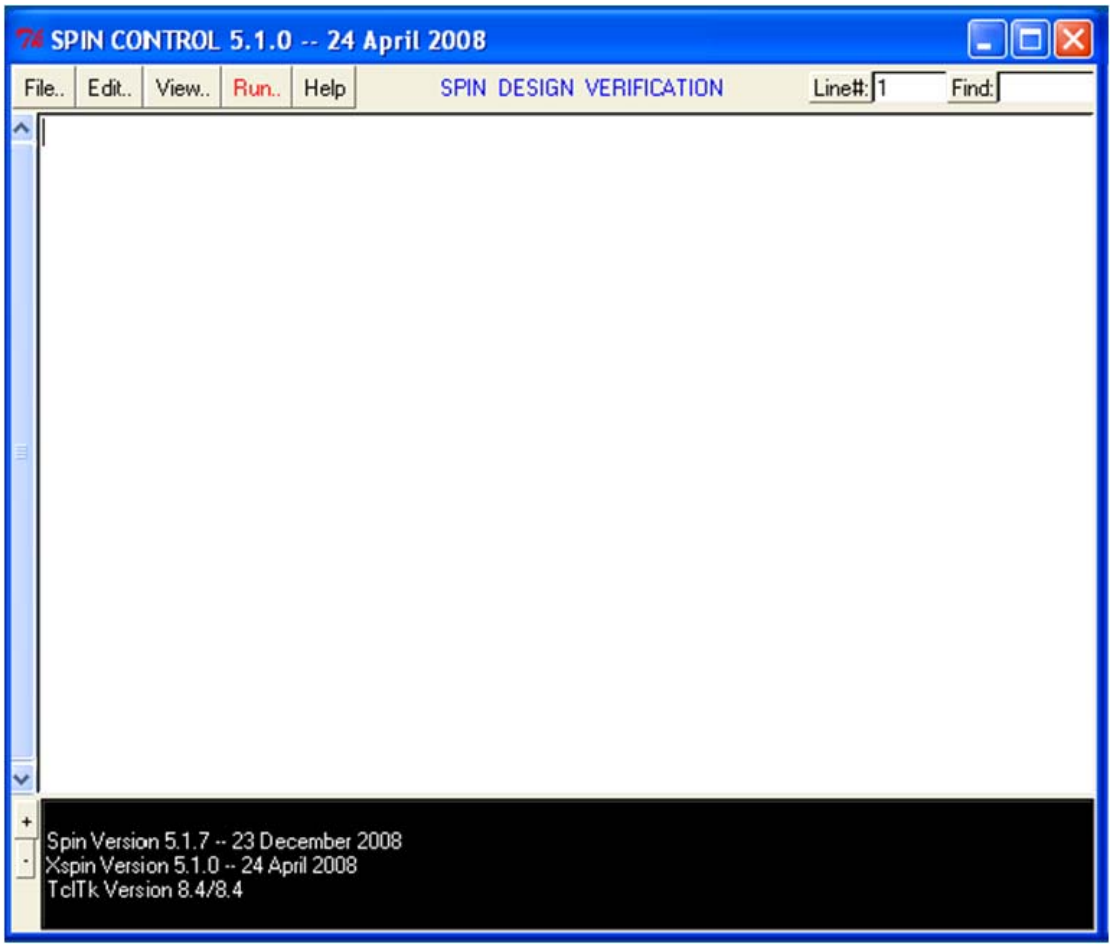


Рис. 4.1. Рабочая область программы XSpin

```
active proctype example(){
    printf("MSC: hello world\n")}
```

Модели на Promela используются для описания поведения систем потенциально взаимодействующих процессов, т. е. нескольких асинхронно выполняемых потоков. Поэтому первичная единица выполнения в Spin — это процесс. В отличие от C, здесь нет основной процедуры `main`. Служебное слово `proctype` описывает в этом примере новый тип процессов с названием `example`. Префикс `active` говорит о том, что один экземпляр процесса типа `example` должен быть создан сразу в начале работы модели. Если префикс опущен, то экземпляр процесса не будет создан при таком объявлении. Создание экземпляров процессов, объявленных с помощью служебного слова `proctype`, может быть выполнено другими средствами.

Введем текст модели в основное окно XSpin. Сохраним модель в файле `hello` при помощи меню *File*. Можно воспользоваться и другим

текстовым редактором, обновляя файл с помощью опции *ReOpen* в меню *File* XSpin.

Для того чтобы проверить синтаксические ошибки в файле с моделью на Promela, выберите в меню *Run* пункт *Run Syntax Check*. В нашем файле `hello` синтаксических ошибок нет, и в командном окне выведется сообщение `no syntax errors`.

Spin может продемонстрировать один из возможных вариантов поведения модели — такой режим работы Spin называется *симуляцией* (пункт *(Re)Run Simulation* меню *Run*).

До запуска симуляции необходимо проверить параметры этого режима (пункт *Set Simulation Parameters* меню *Run*).

В окне параметров симуляции (рис. 4.2) на панели режима отображения (*Display Mode*) задаются окна, которые необходимо выводить при симуляции — панель диаграммы взаимодействия *MSC Panel*, режим вывода событий отдельных процессов *Time Sequence Panel*, панель для просмотра значений данных *Data Values Panel* и панель выполненных шагов модели *Execution Bar Panel* (о параметрах симуляции см. раздел 6.1.3).

В нашей первой модели присутствует вывод на панель диаграммы взаимодействия, о чем свидетельствует синтаксис оператора `printf`:

```
printf("MSC: ")
```

поэтому необходимо установить флаг *MSC Panel*.

На панели *Simulation Style* устанавливается тип симуляции. Возможны три вида симуляции: случайная *Random* — все недетерминированные решения определяются случайным образом, управляемая *Guided*, интерактивная *Interactive* — все недетерминированные решения запрашиваются у пользователя в интерактивном режиме.

Установим случайный тип симуляции *Random simulation*. Запустим модель на симуляцию, нажав кнопку *Start*.

При симуляции Spin найдет некоторое возможное вычисление модели. В окне вывода симуляции (*Simulation Output*) отображаются все события модели, произошедшие во время этого конкретного вычисления. Нажмите *Run*, запустив тем самым режим симуляции. В данном конкретном примере на шаге 1 был выполнен оператор `printf`.

Информация в окне вывода симуляции является исчерпывающей (рис. 4.3): по ней можно точно установить всю последовательность

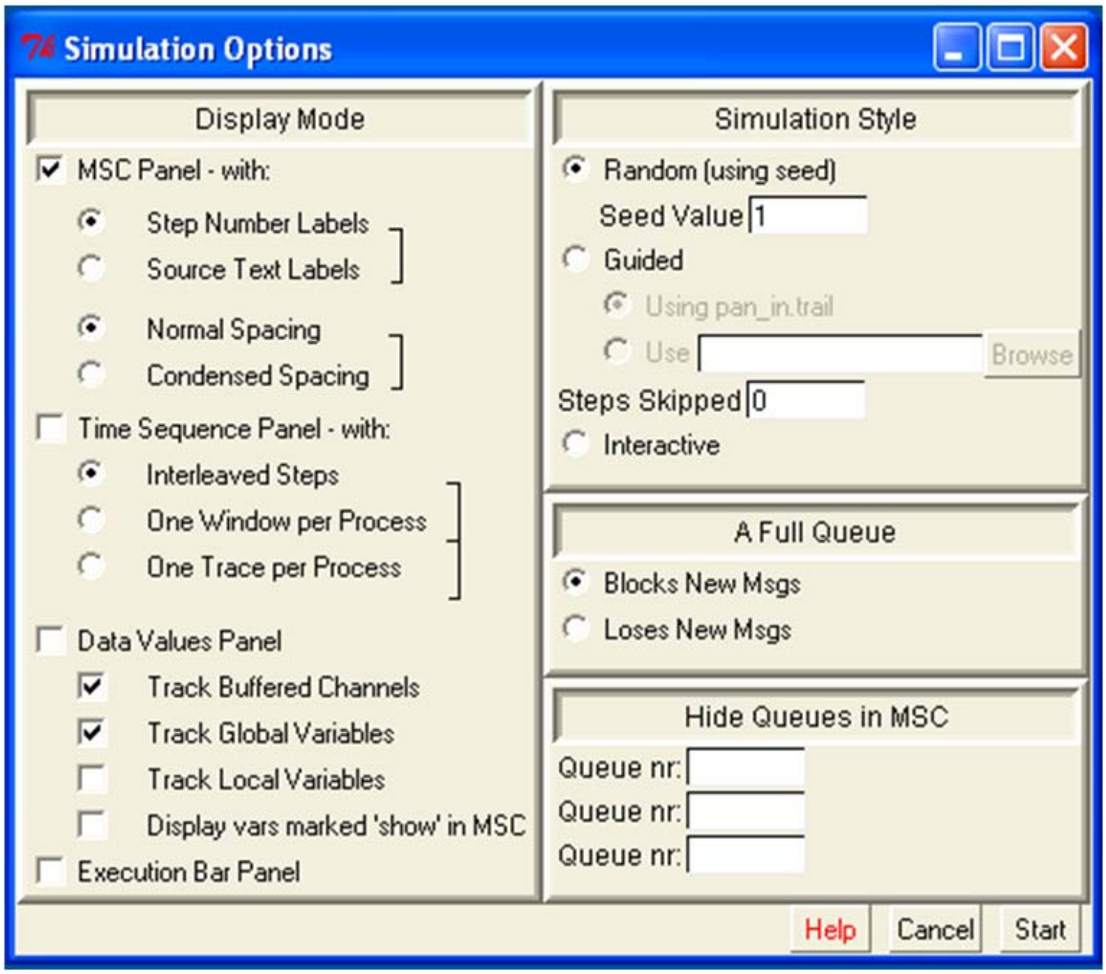


Рис. 4.2. Окно параметров симуляции XSpin

событий конкретного вычисления. Однако формат представления не удобен для пользователя.

Второе окно, открывшееся при запуске, — панель диаграммы взаимодействия, *Sequence Chart*, (рис. 4.4) — наглядно демонстрирует результат симуляции нашего первого примера. Единственный процесс типа `example` с идентификатором `0` вывел сообщение `hello world`.

## 4.2. ТИПЫ ОБЪЕКТОВ ЯЗЫКА PROMELA

Программа, написанная на Promela, состоит из объектов трех основных типов: *процессы (processes)*, *каналы сообщений (message channels)* и *переменные (data objects)*. Процессы задают поведение, каналы и глобальные переменные определяют окружение, в котором выполняются процессы.

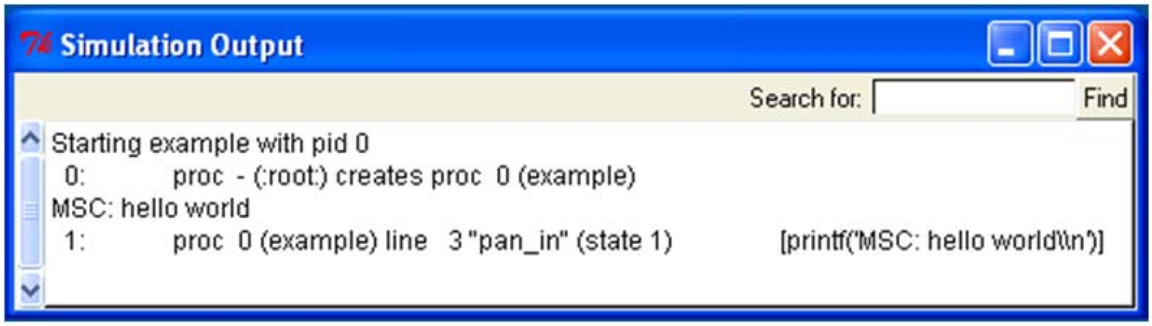


Рис. 4.3. Окно вывода симуляции для модели Hello world


Рис. 4.4. Панель диаграммы взаимодействия для модели Hello world

### 4.2.1. Типы процессов

Типы процессов описываются при помощи объявления `proctype`. Для разумного использования модели на Promela должнs быть хотя бы одно объявление типа процесса и хотя бы один экземпляр процесса какого-нибудь типа. Процессы всегда объявляются глобально.

Существует несколько способов создания экземпляров типа процесса. В приведенном примере создается 2 экземпляра типа процесса `you_run` с помощью префикса `active`:

```
active [2] proctype you_run(){
    printf("MSC: my pid is: %d\n", _pid)}
```

Каждый запущенный процесс имеет уникальный идентификатор, значение которого неотрицательно. Идентификатор процесса хранится в зарезервированной локальной переменной `_pid`.

Тело процесса может состоять из *объявлений данных* и операторов, а может быть и пустым. Состояние переменной или канала сообщений изменяется или проверяется только в процессах.

Точка с запятой является *разделителем операторов* в теле процесса (но не указателем конца оператора, поэтому после последнего оператора точка с запятой может отсутствовать). Лишние точки

с запятой не являются ошибкой, поскольку в Promela допускается пустой оператор. В программах на Promela существуют два различных разделителя операторов: стрелка -> и точка с запятой ;. Эти два разделителя эквивалентны. Стрелка иногда используется как неформальный способ указания на причинно-следственное отношение между двумя операторами: если предыдущий оператор выполняется, то управление передается второму оператору (что и соответствует семантике точки с запятой).

В окне вывода симуляции отобразится информация о том, что было создано два процесса типа `you_run` (рис. 4.5).

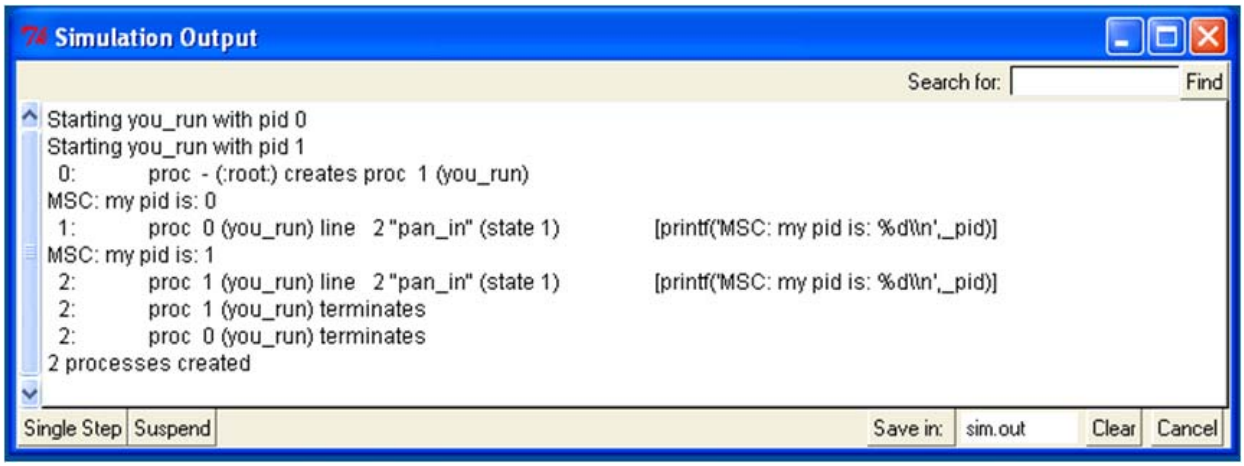


Рис. 4.5. Окно вывода симуляции `you_run`

На шаге 1 процесс с идентификатором 0 выполнил оператор `printf`, на шаге два процесс с идентификатором 1 выполнил оператор `printf`, после чего оба процесса завершили работу. Поскольку процессы асинхронны, то очевидно, что последовательность выполнения операторов `printf` могла бы быть противоположной. На панели диаграммы взаимодействия (рис. 4.6) каждый столбец отображает один запущенный процесс.

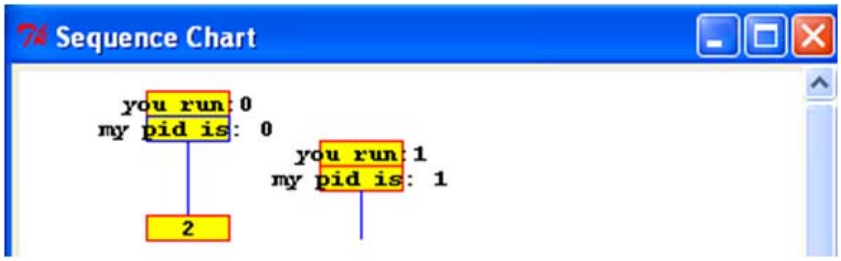


Рис. 4.6. Панель диаграммы взаимодействия модели `you_run`

Процесс также можно запустить при помощи оператора `run`:

```
proctype you_run(byte x){
    printf("MSC: x is  %d\n",  x);
    printf("MSC: my pid is = %d\n", _pid)}
init {
    run you_run(0);
    run you_run(1) }
```

В этом примере процесс `init` запускает два процесса типа `you_run` и передает каждому из них входной параметр. Все запущенные процессы работают параллельно. Процесс `init` — базовый процесс в Promela, который всегда активируется в начальном состоянии модели. Процессу `init` нельзя передать параметры или создать его копию. Если он есть в модели, то его идентификатор всегда равен `0` (рис. 4.7).

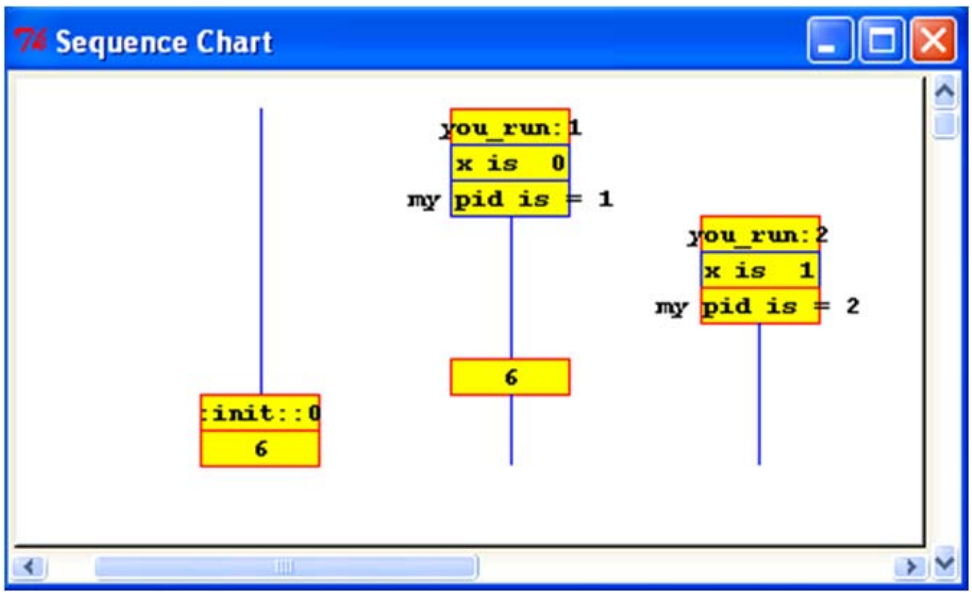


Рис. 4.7. Панель диаграммы взаимодействия модифицированной версии `you_run`

Данное решение имеет недостаток, связанный с созданием лишнего процесса (процесса `init`), что увеличивает размер модели. Размер модели не влияет на режим симуляции, но отражается на времени верификации .

Оператор `run` используется в любых процессах для того, чтобы породить новый экземпляр процесса заданного типа, т. е. `run` может встретиться не только в процессе `init`. Выполняющийся процесс завершается, когда достигает конца тела описания своего типа процесса, но не позже своего процесса—родителя. Количество процессов, которое может быть создано в Spin, ограничено числом 255.

### 4.2.2. Типы данных

Типы данных Promela близки к данным языка C (табл. 4.1).

Таблица 4.1

**Типы данных и диапазоны значений для компьютера с длиной слова 32 бит**

| Тип данных | Диапазон | Пояснение |
|---|---|---|
| `bit` | 0, 1 | |
| `bool` | false, true | |
| `byte` | 0 .. 255 | |
| `chan` | 1 .. 255 | Число каналов |
| `mtype` | 1 .. 255 | Число значений перечислимого типа |
| `pid` | 0 .. 255 | Число возможных процессов |
| `short` | $-2^{15}..2^{15}-1$ | |
| `int` | $-2^{31}..2^{31}-1$ | |
| `unsigned` | $0..2^{32}-1$ | |

В Promela существуют два уровня доступа к объектам: локальный и глобальный. Объект, объявленный глобально, будет доступен всем процессам.

Используются ли *переменные* для хранения глобальной информации о системе в целом или информации, касающейся конкретного процесса, зависит от того, где помещается *объявление переменной*. Переменная является *глобальной*, если она объявлена вне описания процесса, и *локальной*, если она объявлена в описании процесса. Не существует возможности ограничить доступ к локальной переменной лишь для части модуля, описывающего процесс (т. е. не существует аналога блока или области видимости). Все переменные инициализируются нулями (для булевой переменной это `false`):

```
bool flag;  /* примеры объявлений переменных */
int state;
byte msg;
```

В языке допускаются комментарии только такого вида: `/* комментарий */`.

В Promela поддерживаются только одномерные массивы. Например, `byte state[N]` объявляет массив с именем `state` из `N` байт, к которому можно обращаться следующим образом:

```
state[0] = state[3] + 5*state[3*2/n];
```

где `n` является константой или переменной.

Индексом массива может являться любое выражение, которое вычисляет целочисленное значение. При значении индекса вне диапазона `0..N-1` результат не определен; наиболее вероятно, что это вызовет ошибку при выполнении программы (`runtime error`).

Многомерные массивы могут быть заданы неявно с помощью конструкции `typedef` (см. пример алгоритма Нидхама — Шредера в разделе 6.3).

Перечислимый тип `mtype` (mnemonic type, мнемонический тип) может содержать символьные значения переменных, которые задаются при помощи одного или нескольких объявлений. Фактически это аналог перечислимого типа в других языках программирования. Все переменные, объявленные с помощью `mtype`, представляют собой одно множество перечислимого типа, независимо от числа объявлений (рис. 4.8). При помощи всех объявлений типа `mtype` можно задать не более 255 значений.

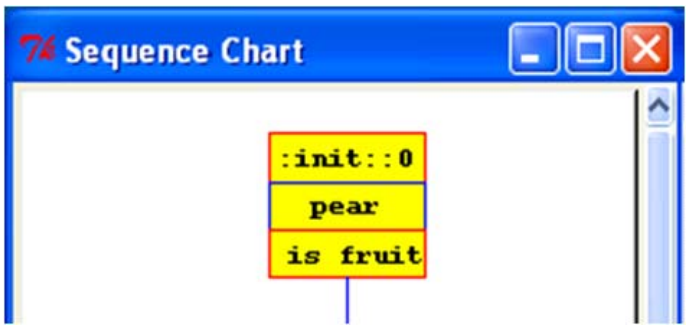


Рис. 4.8. Панель диаграммы взаимодействия модели с `mtype`

```
mtype = { apple, pear, orange, banana };
mtype = { fruit, vegetables, cardboard };
init {
  mtype n = pear;  /* инициализация переменной n значением
                      pear */
```

```
  printf(«MSC: %e », n);
  n = fruit;  /* присвоение переменной n значения fruit
              */
  printf(«MSC: %e », n);
}
```

Синтаксис оператора `printf` подобен аналогичному оператору в C. У этого оператора два аргумента: строка и список аргументов. В строке задаются форматы выводимых переменных: `d` — целое число в десятичном формате, `i` — беззнаковое целое число, `c` — один символ, `e` — константа типа `mtype`. Как уже отмечалось ранее, для вывода сообщения на диаграмме взаимодействия XSpin строка в `printf` должна начинаться с пяти символов `"MSC: "`.

### 4.2.3. Каналы

*Каналы сообщений* используются для моделирования передачи данных от одного процесса к другому. Они объявляются локально или глобально при помощи служебного слова `chan`:

```
chan qname = [16] of { short };
```

Здесь объявлен канал `qname` с буфером 16, т. е. этот канал может хранить до 16 сообщений типа `short` (в канале может находиться не более 16 непрочитанных сообщений).

Для работы с каналами существуют операторы посылки и получения сообщений. Оператор `!`:

```
qname ! expr;
```

посылает сообщение со значением переменной `expr` в только что объявленный канал `qname`, добавляя это значение к концу очереди в канале. По умолчанию оператор выполняется, если канал назначения не переполнен, в противном случае он блокируется. Каналы передают сообщения в порядке FIFO: первым вошел — первым вышел.

Оператор приема сообщения `?`:

```
qname ? msg;
```

принимает сообщение из начала буфера канала и сохраняет его в переменной `msg`. Выполнение оператора приема сообщения возможно только в случае, если канал не пуст.

Для передачи составных сообщений, т. е. сообщений, содержащих конечное число полей, используются каналы, объявленные подобно следующему:

```
chan pname = [16] of { byte, int, chan };
```

В примере объявляется канал `pname` для сообщений, каждое из которых содержит три поля — одно восьмибитное значение (типа `byte`), стоящее в начале сообщения, одно 32–битное значение (типа `int`) и имя канала. Передача идентификатора (имени) канала от одного процесса другому возможна посредством сообщения (как в данном примере) или в качестве параметра экземпляра процесса (см. пример алгоритма выбора лидера в разделе 6.1). Заметим, что использование массивов как поля сообщения в явном виде не допускается.

При посылке нескольких значений в одном сообщении задается список переменных через запятую:

```
pname ! expr1,expr2,expr3;
```

В другом процессе можно принять это сообщение, присвоив три переданные значения трем различным переменным:

```
pname ? var1, var2, var3;
```

Часто первое поле сообщения используется для задания типа сообщения (фактически для связывания с сообщением некоторой характеристики). Эти данные объявляются при помощи типа `mtype`:

```
/* объявляется тип сообщения: */
mtype = { ack, nak, err, next, accept }
/* объявляются две переменные этого типа: */
mtype msgtype1, msgtype2;
```

При использовании типа передаваемых данных `mtype` в объявлении канала соответствующие поля сообщений будут всегда интерпретированы символически, а не численно. Например:

```
chan tname = [4] of { mtype, int, bit };
```

Здесь по каналу `tname` будут пересылаться сообщения, состоящие из трех полей: первое — типа `mtype`, второе — целое значение, третье — битовое значение. Для задания операций посылки и получения сообщения определенного типа используется следующее обозначение:

```
tname ! msgtype (data, b);
```

Эта запись эквивалентна следующей альтернативной записи:

```
tname ! msgtype, data, b;
```

Некоторые аргументы операторов как посылки, так и приема сообщений могут быть константами:

```
/* По каналу tname передается сообщение из двух констант
(ack типа mtype и 0) и значения целой переменной var */
tname ! ack, var, 0
/* Из канала tname читается сообщение, в котором первым
элементом должна быть константа ack типа mtype, вторым
— произвольное значение типа int, которое запишется в
переменную data, третьей частью принятого сообщения должна
быть битовая константа 1 */
tname ? ack (data,1)
```

Оператор получения сообщения при наличии констант будет выполним, только если сообщение в начале буфера канала в соответствующих полях имеет значения заданных констант. В противном случае он будет блокирован. Например, приведенный выше оператор получения сообщения будет невыполним, если в начале буфера находится сообщение `<ack,15,0>`.

Если оператор окажется невыполнимым, то процесс, пытающийся его выполнить, будет приостановлен, пока оператор не станет выполнимым. Таким образом, можно моделировать коммуникацию «точка — точка» нескольких процессов, связанных одним каналом.

Рассмотрим упрощенную реализацию протокола альтернирующего бита (Bartlett, Scantlebury и Wilkinson):

```
/* объявляем два возможных типа сообщения */
mtype = { msg, ack };
/* объявляем канал отправителя */
chan to_sndr = [2] of { mtype, bit };
/* объявляем канал получателя */
chan to_rcvr = [2] of { mtype, bit };
/* процесс отправителя */
```

```
active proctype Sender(){
again:
  to_rcvr!msg,1;  /* отправляем сообщение, помеченное битом
                     1*/
  to_sndr?ack,1;  /* получаем подтверждение, помеченное би-
                     том 1*/
  to_rcvr!msg,0;  /* отправляем сообщение, помеченное битом
                     0*/
  to_sndr?ack,0;  /* получаем подтверждение, помеченное би-
                     том 0*/
  goto again }
/* процесс получателя */
active proctype Receiver(){
again:
  to_rcvr?msg,1;  /* получаем сообщение, помеченное битом 1
                     */
  to_sndr!ack,1;  /* отправляем подтверждение, помеченное
                     битом 1 */
  to_rcvr?msg,0;  /* получаем сообщение, помеченное битом 0
                     */
  to_sndr!ack,0;  /* отправляем подтверждение, помеченное
                     битом 0*/
  goto again }
```

В алгоритме альтернирующего бита процесс-отправитель попеременно отправляет сообщения, помеченные то битом 1, то битом 0, и ожидает соответствующие подтверждения. Процесс-получатель получает сообщения, помеченные то битом 1, то битом 0, и отсылает процессу-отправителю на них подтверждения. В полной версии протокола сообщения могут быть потеряны, но здесь потери канала не моделируются. Бесконечный цикл реализован в этой модели при помощи оператора перехода `goto` и локальной метки `again` в каждом процессе (рис. 4.9).

В языке Promela для каналов определено несколько функций (`len`, `empty`, `nempty`, `full`, `nfull`). Например, предопределенная функция `len(qname)` возвращает количество сообщений, хранящихся в канале `qname` в текущий момент времени. Если `len` используется как оператор по правую сторону присваивания, то он будет невыполимым,

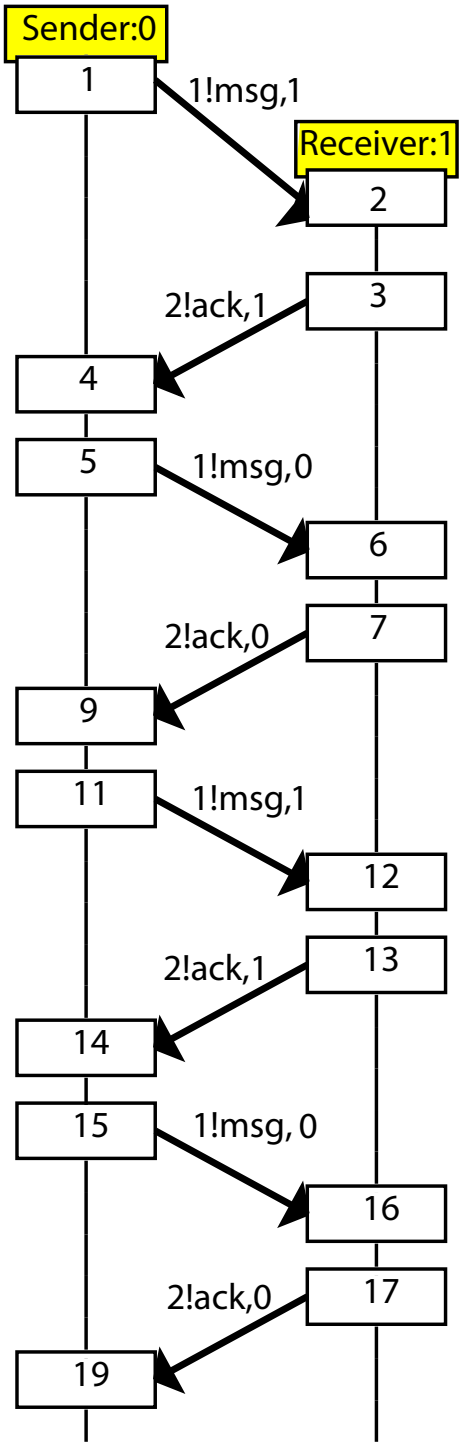


Рис. 4.9. Диаграмма взаимодействия модели протокола альтернирующего бита

если канал пуст, так как он возвращает нулевой результат, который по определению означает, что оператор временно невыполним. Покажем, как отправить сообщение msgtype, если канал qname не переполнен:

```
(len(qname) < MAX) -> qname ! msgtype
```

Здесь если доступ к каналу qname разделяется несколькими процессами, то выполнение второго оператора необязательно будет происходить сразу после выполнения первого оператора проверки. Например, это случится, если другой процесс пошлет сообщение в канал qname (и число сообщений в нем станет равно MAX) сразу после того, как данный процесс определит, что канал не полон.

### 4.2.4. Взаимодействие рандеву

Ранее мы рассматривали только асинхронные взаимодействия между процессами через канал сообщений, декларируемый как:

```
chan qname = [N] of { byte }
```

где N является положительной константой, которая определяет размер буфера.

Установив размер буфера равным нулю,

```
chan port = [0] of { byte }
```

определим *рандеву–канал* (который в данном примере может передавать однобайтные сообщения). Поскольку буфер рандеву-канала равен 0, то такой канал может передавать, но не может хранить сообщения. Взаимодействия процессов по рандеву-каналам по определению синхронны. Рассмотрим следующий пример:

```
#define msgtype 1
chan name = [0] of { byte, byte };
proctype A(){
    name ! msgtype(4);
    name ! msgtype(1) }
proctype B(){
    byte state;
    name ? msgtype(state) }
init{
    atomic { run A(); run B() }
```

Канал `name` объявлен здесь как глобальный рандеву-канал. Два процесса синхронно выполнят свои первые операторы: подтверждение связи («рукопожатие») по сообщению `msgtype` и передачу значения 4 в локальную переменную `state`. Второй оператор посылки в процессе `A` невыполним, потому что отсутствует соответствующая операция приема сообщения в процессе `B`.

Если бы канал `name` был определен с ненулевым размером буфера, поведение было бы отличным. Допустим, что размер буфера равен 2, тогда процесс типа `A` может закончить свое выполнение даже до того, как процесс `B` начнет работу. Если назначить размер буфера равным 1, то последовательность событий будет следующей. Процесс типа `A` может закончить свое первое действие посылки, но он блокируется на втором действии, потому что теперь канал полон. Процесс типа `B` читает первое сообщение и завершается. В этой точке `A` становится опять выполнимым и завершается, оставляя свое последнее сообщение в канале.

Взаимодействия рандеву бинарные: только два процесса, отправитель и получатель, могут быть синхронизированы при рандеву рукопожатии (пример использования рандеву—канала для построения семафора см. в разделе 5.2).

### 4.2.5. Правило выполнимости

Promela основывается на семантике *выполнимости*, которая обеспечивает базовые средства в языке для моделирования синхронизации процессов. В зависимости от состояния системы любой оператор в модели Spin или *выполним*, или *блокирован*. Определены четыре основных *типа операторов* в Promela: оператор вывода переменных `printf`, оператор присваивания, операторы ввода/вывода (при передаче данных по каналам) и операторы-выражения. Если процесс достиг точки кода, где содержится невыполнимый оператор, то процесс *блокируется*. Он может стать снова выполнимым, если *другой* активный процесс выполнит действия, позволяющие оператору, блокированному в данном процессе, выполняться далее.

Благодаря правилу выполнимости запись многих операций в Promela упрощается. Например, вместо того чтобы писать цикл активного ожидания

```
while (a != b) ->skip   /* ожидать до тех пор, пока не выполнится условие a==b */
```

где skip является пустым (нулевым) оператором, в Promela достаточно написать выражение

```
(a == b)
```

Это пример «пассивного» ожидания, когда процесс приостанавливается до выполнения условия (в данном случае равенства a и b).

Рассмотрим следующую модель, в которой два процесса разделяют доступ к глобальной переменной state:

```
byte state = 1;
active proctype A(){
    byte tmp;
    (state==1) -> tmp = state; tmp = tmp+1; state = tmp }
active proctype B(){
    byte tmp;
    (state==1) -> tmp = state; tmp = tmp-1; state = tmp }
```

Процессы ожидают выполнения условия (state == 1). Если каждый из процессов пройдет это условие до того, как другой изменит значение state, выполнив оператор state = tmp, то оба процесса завершатся, хотя итоговое значение глобальной переменной state будет непредсказуемо. В результате работы этой программы после ее завершения переменная state может иметь любое значение: 0, 1 или 2. Если же один из процессов успеет изменить значение переменной state до проверки условия другим процессом, то другой процесс заблокируется.

### 4.2.6. Выражения

Для построения выражений в языке Promela используются операторы, схожие с операторами языка C. В порядке приоритетов они приведены в табл. 4.2. Как обычно, при сомнениях о порядке выполнения операторов и для большей ясности рекомендуется использовать круглые скобки.

Выражения в Promela рассматриваются как утверждения, т. е. проверяются на истинность — ложность в любом контексте. Выражение выполнимо тогда и только тогда, когда его булево значение

Таблица 4.2

**Допустимые операторы предшествования в выражениях на языке Promela (по старшинству — от старших к младшим)**

| Операторы | Направление действия | Комментарий |
|---|---|---|
| () [] | Слева направо | Скобки, скобки массивов |
| ! ++ − | Справа налево | Отрицание, увеличение на 1, уменьшение на 1 |
| * / % | Слева направо | Умножение, деление, модуль |
| + - | Слева направо | Сложение, вычитание |
| << >> | Слева направо | Сдвиг влево, сдвиг вправо |
| < <= > >= | Слева направо | Операторы отношений |
| == != | Слева направо | Равенство, неравенство |
| & | Слева направо | Побитовое И |
| ^ | Слева направо | Побитовое исключающее ИЛИ |
| \| | Слева направо | Побитовое ИЛИ |
| && | Слева направо | Логическое И |
| \|\| | Слева направо | Логическое ИЛИ |
| -> : | Справа налево | Операторы условий |
| = | Справа налево | Присваивание |

истинно (`true`), что эквивалентно любому ненулевому значению целочисленной переменной.

Оператор *присваивания*

```
variable = expression
```

не рассматривается как выражение так же, как и оператор печати. Эти операторы всегда выполнимы. При выполнении оператора присваивания в Promela сначала оценивается выражение, стоящее справа (сначала инициализируются значения переменных выражения, если это не было сделано ранее, потом к ним применяются соответствующие операторы, см. табл. 4.2). После оценки полученный результат выражения при необходимости обрезается до значения того типа, которому принадлежит переменная `variable`, и переменная `variable` получает это значение.

Использование инкремента и декремента в Promela отличается от их использования в C: они могут быть только постфиксными (т. е.

можно записать a++, но нельзя ++a) и могут использоваться только в выражении, но не в операторе присваивания.

*Входные*/выходные *операторы* выполняются только при наличии возможности отправки и получения сообщения, иначе процесс, выполняющий соответствующий оператор, блокируется (см. разд. 4.2.3). В некоторых случаях требуется протестировать возможность операции отправки или получения, не выполняя ее. Для этого входные/выходные операторы трансформируются в обычные выражения взятием аргумента оператора в квадратные скобки. Таким образом, при использовании квадратных скобок во входных/выходных операторах сам оператор не выполняется, а лишь вычисляется его значение как выражения, например:

```
qname ? [ack, var, 0]
```

Смысл этого логического выражения состоит в проверке того, что очередным сообщением в канале с именем `qname` является структура, состоящая из мнемонического значения `ack`, некоторого значения переменной `var` и значения 0. Если утверждение выполнимо, то возвращается 1, иначе возвращается 0.

Похожие на это условные выражения С—подобного вида

```
(qname ? var == 0)    /* синтаксическая ошибка */
```

или

```
(a > b && qname ! 123)    /* синтаксическая ошибка */
```

в Promela недопустимы, поскольку они не могут быть вычислены без побочных эффектов (попытки выполнить операции ввода/вывода вместо того, чтобы просто проверить, что очередное сообщение содержит 0 в первом случае или может быть послано — во втором). Кроме того, сами операторы отправки и получения данных не являются выражениями.

Для *вывода значений переменных или текста* используется оператор `printf`. По своему действию на состояние системы `printf` подобен пустому оператору `skip`. Оба этих оператора используются в моделях, когда необходимо осуществить всегда выполнимый шаг. Вывод текста `printf` производится только в режиме симуляции Spin, поскольку является побочным действием оператора.

## 4.3. СОСТАВНЫЕ ОПЕРАТОРЫ

В разд. 4.2 были рассмотрены базовые операторы, существующие в Promela. Кроме них в языке определены составные операторы: последовательности вида `atomic`, детерминированные шаги, структура выбора и структура повторения.

### 4.3.1. Блок `atomic`

Последовательность операторов, заключенных в фигурные скобки, перед которой стоит ключевое слово `atomic`, представляет блок кода, который должен выполняться как один неделимый модуль, не чередующийся с другими процессами. Действие `atomic` подобно использованию семафора. Выполнение последовательности операторов внутри `atomic` является неделимым «с точки зрения» других процессов: другие процессы «видят» разделяемые глобальные переменные и каналы, используемые в этом блоке, либо до, либо после выполнения всей последовательности операторов внутри блока `atomic`. Если какой-либо оператор внутри `atomic` не является выполнимым, вся последовательность операторов не выполняется.

Модифицируем пример с разделяемым использованием общей для двух процессов переменной `state`:

```
byte state = 1;
active proctype A(){
    atomic {
        (state==1) -> state = state+1 } }
active proctype B(){
    atomic {
        (state==1) -> state = state-1 } }
```

Использование блока `atomic` позволяет здесь предотвратить доступ конкурирующего процесса к глобальной переменной. При запуске этой модели может быть недетерминированно выбран либо процесс `A`, либо процесс `B` для того, чтобы выполнить все операторы блока `atomic`. Если начал выполняться процесс `A`, то он завершится, после чего запустится процесс `B`, который, однако, будет приостановлен на проверке условия `(state == 1)`. Аналогично, если начал выполняться процесс `B`, то он завершится, после чего процесс `A` будет приостановлен на проверке такого же условия. Итоговое значение переменной `state` будет либо 0, либо 2, в зависимости от того, какой

из двух процессов выполнится.

Блоки `atomic` являются важным инструментом понижения сложности моделей верификации — уменьшения числа глобальных состояний строящейся Spin формальной модели переходов: блоки `atomic` ограничивают количество чередований (интерливинга).

Однако надо понимать, что интерливинг часто возникает в распределенных системах и порождает неопределенность в выполнении параллельных процессов, избавление от которой может являться нежелательным эффектом при верификации систем: если *в реализации* операторы в блоке `atomic` выполняются не неделимо, то их чередование может иметь разные последствия.

### 4.3.2. Выбор по условию

Оператор `if` отличается от обычных условных операторов современных языков программирования. В простом примере воспользуемся относительными значениями двух переменных `a` и `b` для выбора между двумя опциями:

```
if
:: (a != b) -> option1
:: (a == b) -> option2
fi
```

*Структура выбора* `if` содержит последовательности операторов, каждая предваряется двойным двоеточием. Выполняется только одна последовательность из списка возможных (выполнимых). Последовательность выполнима, если ее первый оператор выполним. Первый оператор называется *защитой (guard)* или *условием*.

Если выполнимыми оказываются несколько операторов, недетерминированно выбирается какой-либо один. При этом порядок перечисления альтернатив выбора несущественен, он ни на что не влияет. Если все условия невыполнимы, то процесс будет заблокирован, пока хотя бы одно из них не сможет быть выбранным. Ограничения на типы выражений, которые могут быть использованы в качестве защиты, отсутствуют. Рассмотрим пример с использованием входных операторов.

```
#define a 1
#define b 2
```

```
chan ch = [1] of { byte };
proctype A(){ ch ! a }
proctype B(){ ch ! b }
proctype C(){
    if :: ch ? a -> option1
       :: ch ? b -> option2
    fi }
init{
    atomic { run A(); run B(); run C() } }
```

Здесь определены три процесса и один канал `ch`. Первая опция в структуре выбора процесса типа `C` является выполнимой, если канал содержит сообщение-константу `a` (`a` является константой со значением 1). Вторая опция будет выполнимой, если канал содержит сообщение `b` (`b` также является константой). Какой из операторов будет выполнен, зависит от относительных скоростей процессов, которые неизвестны.

Рассмотрим процесс, который либо увеличивает, либо уменьшает значение переменной `count`:

```
byte count;
proctype counter(){
    if :: count = count + 1
       :: count = count - 1
    fi }
```

Поскольку оба выражения в примере всегда выполнимы, то выбор между ними полностью недетерминирован, он не зависит от начального значения переменной `count`.

Процессы могут быть смоделированы как *рекурсивные*. Возвращаемое значение передается обратно в вызывающий процесс посредством глобальной переменной или через сообщение. Процесс `fact(n,p)` рекурсивно вычисляет факториал `n`, передавая результат с помощью сообщения своему родительскому процессу `p`:

```
proctype fact(int n; chan p) {
    chan child = [1] of { int };
    int result;
    if :: (n <= 1) -> p ! 1
       :: (n >= 2) ->run fact(n-1, child);
```

```
            child ? result;
            p ! n*result
        fi }
    init{
        chan child = [1] of { int };
        int result;

        run fact(7, child);
        child ? result;
        printf("MSC: result: %d\n", result) }
```

### 4.3.3. Цикл

Логическим расширением структуры выбора является *структура повторения (цикл)*. Модифицируем представленный выше пример таким образом, чтобы получить циклическую программу, которая произвольно меняет значение переменной, увеличивая или уменьшая его:

```
byte count;
active proctype counter(){
    do :: count = count + 1
       :: count = count - 1
       :: (count == 0) -> break
    od }
```

Аналогично предыдущему оператору `if` в операторе `do` для текущего выполнения может быть выбрана только одна опция. После того как выполнение выбранной опции завершится, управление передается на начало оператора цикла: выполнение структуры повторяется.

Обычный способ выхода из цикла — с помощью оператора `break`. Можно считать `break` не оператором, а указанием на то, что цикл завершается, и управление должно быть передано оператору, следующему за ключевым словом `od`. В примере цикл будет прерван, когда `count` будет 0. В данном примере цикла два других оператора всегда будут выполнимыми, поэтому выбор условия для выхода будет недетерминированным даже при значении `count`, равном 0.

Для того чтобы гарантировать завершение цикла, когда счетчик станет равным 0, необходимо изменить модель следующим образом.

```
active proctype counter(){
    do :: (count != 0) ->
            if  :: count = count + 1
                :: count = count - 1
            fi
       :: (count == 0) -> break
    od }
```

В структурах выбора и повторений бывает полезно специальное условие `else`. Условие `else` становится выполнимым в операторе выбора, *только если* ни одно другое условие в процессе в той же самой точке контроля потоком невыполнимо. Предыдущий пример преобразуем в эквивалентный следующим образом:

```
active proctype counter(){
    do :: (count != 0) ->
          if :: count = count + 1
             :: count = count - 1
          fi
       :: else -> break
    od }
```

Условие `else` становится выполнимым именно тогда, когда `!(count != 0)` или, что то же, `(count == 0)`, и потому сохраняет возможность выхода из цикла.

Другим способом завершения цикла является использование оператора безусловного перехода `goto`. Оператор `goto` всегда является выполнимым, если существует метка, на которую выполняется переход. Это проиллюстрировано в алгоритме Эвклида нахождения наибольшего общего делителя двух положительных чисел:

```
proctype Euclid(int x, y){
    do :: (x >  y) -> x = x - y
       :: (x <  y) -> y = y - x
       :: (x == y) -> goto done
    od;
    done: skip }
```

где выполнение `goto` вызовет передачу управления на метку `done`.

*Метка* может быть поставлена только перед оператором. Здесь метка стоит перед оператором завершения программы. В этом случае пустой оператор `skip` является полезным: он выступает как оператор-заполнитель, который всегда является выполнимым, но не производит никакого эффекта.

Приведем пример с процессом–фильтром, получающим сообщения из канала `ch` и делящим их на два канала `large` и `small` в зависимости от значения данных:

```
#define N    128
#define size  16
chan ch    = [size] of { short };
chan large = [size] of { short };
chan small = [size] of { short };
proctype split(){
    short data;
    do :: ch ? data ->
           if :: (data >= N) ->large ! data
              :: (data <  N) ->small ! data
           fi
    od }
init{
    run split() }
```

Очевидно, что в данной программе процессы заблокируются в начале работы, так как `split` будет ждать сообщений из канал `ch`, но этот канал пуст! Ни один из процессов не отправляет в него сообщения. Введем процесс, который объединяет два потока входных данных в один выходной поток, направляемый в канал `ch` в произвольном порядке:

```
proctype merge(){
    short data;
    do :: if :: large ? data
             :: small ? data
          fi;
          ch ! data
    od }
```

Если модифицировать процесс `init` следующим образом:

```
init{
    ch ! 345; ch ! 12;
    ch ! 6777; ch!32; ch ! 0;
    run split();
    run merge() }
```

то разделяющий и объединяющий процессы будут выполняться бесконечно. Действительно, изначально `init` отправляет сообщения в канал `ch`. Далее процессы `merge` и `split` постоянно поставляют сообщения во входные каналы друг друга. Ни один, ни другой процесс не имеет условий, по которым работа может считаться завершенной.

# 5. ОПИСАНИЕ ВЕРИФИЦИРУЕМЫХ СВОЙСТВ СРЕДСТВАМИ PROMELA И SPIN

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 «правильное поведение» включает некоторое множество свойств, которые делают систему полезной для использования, и все эти свойства должны быть проверены последовательно, одно за другим. Свойства распределенных систем традиционно разбиваются на следующие классы (подробнее в [6]):

- 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 произойдет при любом развитии процесса;
- свойства справедливости (*fairness*), устанавливающие, что нечто будет выполняться неопределенно часто.

*Свойства достижимости* являются одними из наиболее часто проверяемых классов свойств параллельных систем: *некоторая конкретная ситуация в процессе функционирования системы случится*. Свойство достижимости естественным образом выражается LTL формулой $EF\varphi$.

*Свойство безопасности* гарантирует, что при некоторых условиях некоторая ситуация никогда не может быть достигнута (гарантия того, что *нечто плохое никогда не произойдет*). Свойство безопасности выражается формулой $G\neg\varphi$. 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 другим процессом.

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. Прогресс в «правильном» направлении обеспечивается свойствами живости. *Свойство живости* утверждает, что *нечто хорошее в будущем обязательно произойдет*, что выражается формулой CTL $AF\varphi$, или *нечто хорошее обязательно в будущем будет происходить неопределенно часто*, что выражается формулой LTL $GF\varphi$.

Во многих случаях выполнение свойств, определяющих корректное функционирование модели, должно проводиться только на тех вычислениях, на которых выполняется некоторое дополнительное условие, которое называется требованием *справедливости*:

$$fairness \Longrightarrow \text{свойство}$$

Существуют две причины возникновения требования справедливости. Первая причина состоит в том, что анализируемая система будет работать в некотором окружении, к функционированию которого мы формулируем эти требования, поскольку только с этими дополнительными требованиями к окружению наша система может выполнять полезную функциональность. Например, справедливый планировщик должен удовлетворять требования предоставления ресурса от каждого процесса. 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 траектории (вычисления), возникшие из-за ограниченных выразительных средств модели. Назовем нереалистичные траектории поведения модели несправедливыми. Справедливые траектории поведения системы в этом случае не нужно специально обеспечивать при реализации: например, реальный канал сам обеспечивает ненулевую вероятность доставки сообщения, но проверку свойств системы нужно выполнять при соблюдении условий справедливости — отбрасывая нереалистичные траектории поведения модели.

При разработке параллельных систем кроме специфичных требований, которым должно удовлетворять поведение данной конкретной системы, для системы следует проверять еще и общие свойства, гарантирующие отсутствие некорректностей, типичных для параллельных систем. Spin поддерживает верификацию следующего набора общих свойств, называемых в Spin базовыми:

- из класса свойств безопасности:
  - проверка сохранения локальных инвариантов, описанных при помощи оператора `assert`;
  - проверка наличия некорректных конечных состояний, т. е. обнаружение взаимных блокировок процессов;
- из класса свойств живости:
  - проверка отсутствия бесконечных циклов, не содержащих операторов с меткой `progress`;
  - проверка отсутствия циклов с бесконечно частым выполением операторов с меткой `accept`.

## 5.1. ОПЕРАТОР ASSERT

Оператор `assert` позволяет задавать локальные инварианты, т. е. такие свойства, которые должны выполняться в определенных точках программы:

```
assert(любое выражение Promela)
```

Если выражение, стоящее под оператором `assert`, истинно, то `assert` не производит никакого эффекта. Если выражение будет ложно при симуляции, то Spin выдаст сообщение `Error: assertion violated`. При выполнении базовой верификации свойств безопасности нарушение операторов `assert` проверяется на всех конечных вычислениях. В Spin в режиме симуляции могут быть проверены только свойства системы, описанные оператором `assert`. Все другие механизмы описания свойств, поддерживаемые Spin, интерпретируются только в режиме верификации.

## 5.2. МЕТКА ЗАКЛЮЧИТЕЛЬНОГО СОСТОЯНИЯ END

Когда Promela используется для спецификации модели, которая будет верифицироваться в Spin, пользователь может делать индивидуальные утверждения о поведении, которое моделируется. Например, если код проверяется на наличие взаимных блокировок, верификатор

должен уметь отличать нормальные, с точки зрения пользователя, завершающие (заключительные) состояния от ненормальных.

Нормальное заключительное состояние может быть состоянием, когда каждый процесс правильно достиг конца тела описания программы, и все каналы сообщений пусты. Но это не означает, что все процессы кода *должны* достигнуть конца своего тела программы.

Для того чтобы сделать понятным для верификатора, что такие альтернативные заключительные состояния допускаются и не являются дедлоком, можно использовать метки заключительного состояния. В модели может быть несколько таких меток.

В примере, реализующем взаимоисключающий доступ к ресурсу при помощи семафора Дейкстры, добавляя метку, начинающуюся со слова `end`, к оператору цикла,

```
#define p 0
#define v 1
chan sema = [0] of { bit };
proctype dijkstra(){
          byte count = 1;
endpoint: do :: (count == 1) ->
                  sema ! p; count = 0
             :: (count == 0) ->
                  sema ? v; count = 1
          od }
active[3] proctype user(){
         if :: sema ? p;
               /* критическая секция */
               sema ! v;
               /* некритическая секция */
         fi }
```

мы сообщаем системе Spin, что не будет ошибкой (блокировкой процесса), если в конце выполняемой последовательности процесс `dijkstra` не достигнет своей закрывающей фигурной скобки, а будет ожидать запроса от каких-либо процессов.

Процесс `dijkstra()` выполняет здесь роль семафора. Семафор гарантирует, что не более одного пользовательского процесса может войти в критическую секцию. Поскольку процесс `dijkstra` является обслуживающим, естественно его определить так, чтобы он не

знал числа обращающихся к нему клиентских процессов. Поэтому следует рассматривать остановку процесса `dijkstra` в состоянии, в котором его может запросить клиентский процесс о выполнении последовательности операций `p` и `v` над семафором, как корректное состояние завершения процесса.

## 5.3. МЕТКИ АКТИВНОГО СОСТОЯНИЯ PROGRESS

Также как и метки состояния, в процессе могут быть использованы метки активного состояния `progress`. Этими метками помечают операторы, выполнение которых желательно, тем самым усиливая необходимость прогресса в поведении модели. При базовой верификации модели в Spin проверяется, что любой потенциально бесконечный цикл проходит хотя бы через одну метку `progress`, и таким образом, удовлетворяется свойство живости. Если же находится контрпример, нарушающий данное свойство, то Spin сообщает о существовании непрогрессивного цикла, а следовательно, о возможном голодании процессов. Добавим метку активного состояния к процессу `dijkstra`, моделирующему работу семафора посредством канала `sema`. Операции P и V над семафором должны выполняться здесь внешними процессами как `sema ? p` и `sema ! v`:

```
proctype dijkstra(){
          byte count = 1;
endpoint: do :: (count == 1) ->
progress:       sema ! p; count = 0
             :: (count == 0) ->
                sema ? v; count = 1
          od }
```

Считая прогрессом постоянную восстребованность семафора внешними процессами, пометим удачную передачу токена семафором меткой `progress`, тогда при базовой верификации Spin проверит, что в любом бесконечном вычислении семафор будет заблокирован бесконечное число раз.

В процессе допускается использование нескольких меток активного состояния.

## 5.4. МЕТКА ПРИНИМАЮЩЕГО СОСТОЯНИЯ ACCEPT

Метки принимающих состояний используются обычно в особом процессе never (см. разд. 5.5), хотя Promela не ограничивает их использования. Меткой принимающего состояния считается любая метка, начинающаяся префиксом accept:

```
accept[a-zA-Z0-9_]*: операторы
```

Каждая метка принимающего состояния в теле одного процесса должна имеет свое собственное имя, например, accept, acceptance, accepting. Эти метки выделяют состояния, которые соответствуют принимающим (допускающим) состояниям автомата Бюхи, описывающим все бесконечные «ошибочные», «нежелательные» траектории проверяемой программы. Ошибочные траектории характеризуются именно тем, что вычисление проходит такое принимающее состояние бесконечное число раз. Поэтому при верификации проверяется, что в системе не существует вычислений, которые проходят через операторы, помеченные accept, бесконечно часто.

## 5.5. ОСОБЫЙ ПРОЦЕСС NEVER

Многие из свойств могут быть проверены введением операторов assert, меток конечного и активного состояния в тело типа процесса proctype. Однако достаточно трудно определить в модели свойства, подобные данному: *любое состояние системы, в котором* p *истинно, приведет к такому состоянию системы, что истинно* q. Проблема состоит в том, что рассматриваемые ранее способы описания свойств плохо предназначены для проверки во всех состояниях системы. Тем более что мы не можем делать никаких предположений об относительных скоростях процессов, т. е. между любыми двумя операторами процесса могут выполниться несколько (неопределенное число) шагов других процессов.

Особый процесс never дает возможность задания глобальных инвариантов. Процесс never используется для того, чтобы описать такое поведение, которого *не должно* быть в системе. Процесс never предназначен лишь для слежения за поведением системы и не оказывает никакого влияния на нее. В этом процессе нельзя объявить переменные или манипулировать каналами сообщений. В нем запрещены всякие действия, способные изменить состояние системы. В модели может быть только один процесс never.

Процесс never учитывается только при верификации. Он позволяет проверить свойство системы в точности в начальном состоянии и после каждого шага вычисления (т. е. после выполнения каждого оператора, независимо от того к какому процессу этот оператор относится).

Простейший вариант использования процесса never — проверка выполнения условия p на каждом шаге системы:

```
never {
    do :: !p -> break
       :: else
    od }
```

Процесс never с заданным правилом выполняется на каждом шаге системы. Как только условие p станет ложным, процесс never выходит из цикла и прерывается, переходя в завершающее состояние. Завершение процесса never интерпретируется как ошибочное поведение анализируемой системы. Если p остается истинным, то процесс never остается в цикле, он не завершается, и ошибки в анализируемой системе нет.

Для приведенного свойства можно сформулировать альтернативное описание без процесса never. Добавим процесс monitor:

```
active proctype monitor(){
    atomic { !p -> assert(false) } }
```

Здесь процесс с именем monitor может инициировать выполнение последовательности операторов, определенных внутри группы atomic, в любой точке вычисления системы. Поэтому в любом достижимом состоянии системы, в котором инвариант (в данном случае истинность утверждения p) нарушается, процесс monitor выполняет проверку и сообщает об ошибке с помощью оператора assert, т. е. в этом случае процесс monitor решает задачи процесса never. Однако для более сложных темпоральных свойств без процесса never не обойтись.

Рассмотрим свойство: *всегда если p стало истинным, то когда-нибудь в будущем станет истинным q, а p будет оставаться истинным до тех пор, пока q не станет инстинным.*

Это свойство достаточно просто описывается формулой линейной темпоральной логики (LTL): $G(p \rightarrow (p\, U q))$ (учтите, что будущее включает в себя настоящее).

При проверке модели нас не интересуют все те вычисления, в которых свойство удовлетворяется: наоборот, для модели проверяется наличие в ней вычислений, в которых свойство нарушается. Все такие «ошибочные» вычисления для нашего случая удовлетворяют формуле $!G(p \rightarrow (p\ U q))$. При верификации нам нужно выявить среди всех возможных вычислений проверяемой системы хотя бы одно ошибочное вычисление, т. е. такое, в котором $p$ стало истинным, а $q$ осталось ложным на всем вычислении или $p$ стало ложным до того, как $q$ стало истинным (рис. 5.1).

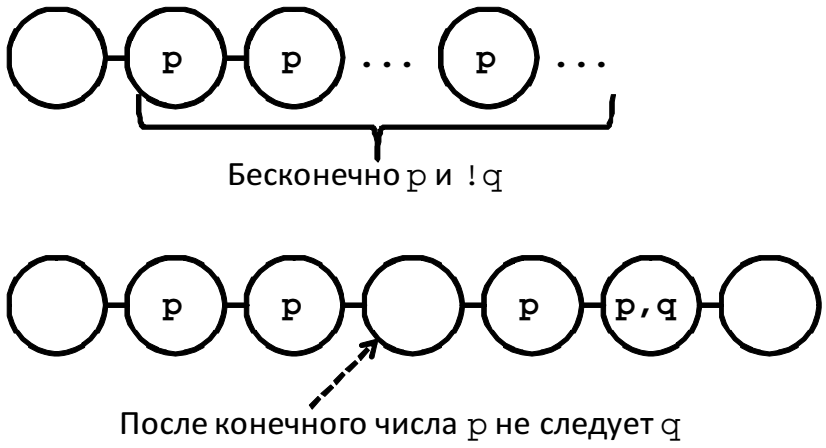


Рис. 5.1. Примеры ошибочных вычислений для LTL формулы $G(p \rightarrow (p\,U q))$. Именно такие трассы и будет искать процесс `never`

Очевидно, что нарушение свойства, где $q$ остается ложным всегда, может случиться только на бесконечных вычислениях. Promela имеет дело с моделями с конечным числом состояний, поэтому бесконечное вычисление появляется только внутри циклов. Мы не можем использовать обычные процессы с `assert` для обнаружения таких ошибок живости (напомним, `assert` проверяются только на конечных вычислениях). Воспользуемся процессом `never` для правильной проверки заданного свойства:

```
never {  /*  !G (p -> (p U q))   */
S0:  do :: p && !q -> break
        :: true
     od;
accept:
     do :: p && !q
```

```
    :: !(p || q) -> break
od }
```

В режиме верификации сначала (в начальном состоянии системы) проверяется возможность выполнения первого оператора процесса `never`. В данном примере выполнение процесса `never` начинается с метки `S0`, которая помечает цикл с недетерминированным выбором. Второе условие состоит из одного оператора с выражением `true`. Очевидно, что эта опция всегда выполнима и не оказывает никакого эффекта на вычисления. Она возвращает процесс `never` в его начальное состояние. Процесс не может заблокироваться в состоянии `S0`, так как выражение `true` всегда выполнимо. Наличие `true` позволяет системе не выполнять проверку свойства, отложив ее на потом.

Первое условие первого цикла `do` выполнимо, только когда выражение `p` истинно, а выражение `q` ложно. Этот случай обнаруживает следующее поведение модели: стало истинно `p`, но `q` еще не истинно. Именно с этого состояния может начаться некорректная траектория выполнения анализируемой программы. Она будет некорректной в двух случаях: 1) когда во всех последующих состояниях будет истинно `p`, а `q` будет ложно; 2) если встретится состояние, в котором не будут истинны ни `p`, ни `q`.

Если первый оператор цикла выполнится (т. е. найдется состояние, в котором `p` истинно, а `q` ложно), то вычисление будет продолжено с метки `accept`.

В состоянии, помеченном этой меткой, цикл также имеет два условия выбора. Один из путей — вечное пребывание в цикле по условию «`p` истинно, а `q` — ложно». Если такое вычисление существует, то это соответствует нарушению свойства $G(p \to (p\,U q))$. Бесконечное повторение этого состояния соответствует ошибочному вычислению. Поэтому это состояние помечено меткой `accept`. Нарушение будет найдено верификатором как цикл с бесконечным выполнением оператора с меткой `accept`.

Другое возможное нарушение возникнет, когда `p` становится ложным до того, как `q` стало истинным. Этот тип нарушения приведет к завершению процесса `never`. Завершение процесса `never` интерпретируется как найденное «ошибочное» поведение, совпадающее с одним из с поведений, которого не должно было быть при выполнении свойства.

Если оба выражения, p и q, окажутся истинными при выполнении второго оператора (цикла), то ни одно из условий выбора невыполнимо, следовательно, процесс never заблокируется. Блокировка процесса never — не ошибка, а желаемое поведение! Блокировка процесса never показывает, что он не завершился и не проходил бесконечный цикл с меткой accept, т. е. не нашел ошибочных поведений модели.

Важно заметить, что в состоянии S0 возможна ситуация, когда оба условия (guard) исполнимы. Здесь недетерминированный выбор имеет критическое значение. Если в приведенном выше примере процесса never заменить условие с true на оператор else, то процесс будет реагировать только на первое выполнение выражения p, которое сразу должно приводить к выполнению выражения q. Наличие недетерминированного условия позволяет сформулировать более сложное условие $G(p \to (p\,U q))$.

На рис. 5.2 приведен недетерминированный автомат Бюхи для формулы $!G(p \to (p\,U q))$, который переходит в заключительное (принимающее) состояние accept, если, начиная с любого состояния (переход true в состоянии s0) встретится бесконечная цепочка состояний, помеченная p, или же после конечного числа состояний, помеченных p, встретится состояние, в котором не истинны ни p, ни q. Этот недетерминированный автомат иллюстрирует поведение, описываемое процессом never.

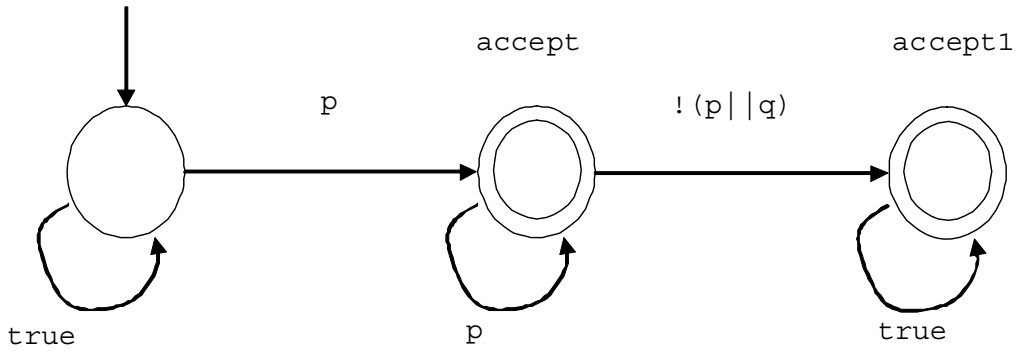


Рис. 5.2. Недетерминированный автомат Бюхи для формулы $!G(p \to (p\,U q))$

Напомним, что процесс never не должен оказывать влияния на поведение системы, поэтому в нем, как видно из примеров, используются лишь операторы условия и accept.

Любая метка, начинающаяся словом `accept`, например, `accept1`, `accept_init`, `accepting`, играет ту же роль в процессе `never`, что и метка `accept`.

Построение процесса `never` для обнаружения ошибочных поведений модели не всегда является простым. Spin позволяет описать проверяемые свойства модели формулами линейной темпоральной логики, используя темпоральные операторы $F$, $G$ и $U$, определенные пользователем атомарные условия (атомарные предикаты), построенные как булевы выражения языка Promela, и обычные булевы операторы &&, || и ! (и, или, нет). Темпоральный оператор $X$ не может быть использован при построении LTL формул в системе Spin (поскольку понятия «следующее состояние» в системе параллельных процессов нет: в общем случае каждый из параллельных процессов может выполнить независимый шаг). По историческим причинам в системе Spin темпоральный оператор $G$ представляется парой квадратных скобок `[]`, а темпоральный оператор $F$ — парой угловых скобок `<>`. Импликация представляется парой символов `->`. Формула LTL $G(p \to (p\,U q))$ будет записана в Spin как `[](p -> (p U q))`.

По введенной темпоральной формуле Ф, выражающей требуемое свойство поведения модели, Spin строит ее отрицание !Ф (формулу, описывающую все «ошибочные» поведения) и по формуле !Ф строит автомат Бюхи $B_{!\Phi}$, конечным образом описывающий все такие поведения. Далее по автомату Бюхи $B_{!\Phi}$ Spin строит процесс `never`, который в режиме верификации будет пытаться обнаружить хотя бы одно «ошибочное» поведение модели аналогично тому, как это описано в приведенном выше примере.

# 6. ПРИМЕРЫ ЗАДАЧ ДЛЯ ВЕРИФИКАЦИИ В SPIN

В разделе подробно описаны несколько задач, модели которых составлены на языке Promela и верифицированы средствами Spin. Кроме того, изложены основы работы с графической оболочкой XSpin. Ознакомление с этим разделом позволит читателям не только самостоятельно моделировать интересующие его задачи, но и проверять корректность разработанных ими моделей при помощи Spin.

## 6.1. ПРОТОКОЛ ВЫБОРА ЛИДЕРА В ОДНОНАПРАВЛЕННОМ КОЛЬЦЕ

Рассмотрим протокол выбора лидера в однонаправленном кольце (алгоритм предложен Долевым, Клаве, Роде (Dolev, Klawe, Rodeh) и независимо Петерсоном (Peterson) в 1982 году) [27, 17]. До создания этого алгоритма считалось, что при выборе лидера в однонаправленном кольце количество сообщений не может быть меньше, чем $O(N^2)$, где $N$ — число узлов в кольце. В данном алгоритме достигается количество сообщений $2N \log 2N + O(N)$. Рассмотрим этот алгоритм.

Дано $N$ распределенных узлов с некоторыми весами. Узлы асинхронно взаимодействуют только с соседями, образуя однонаправленное кольцо. Количество узлов фиксировано и не меняется. Порядок сообщений в каналах не меняется. Требуется построить протокол — набор локальных правил для каждого узла, которые позволят получить глобальный результат — каждому узлу определить единственного лидера (например, узел с наибольшим весом).

Изначально алгоритм был разработан для двунаправленных колец, и при обходе круга активный процесс сравнивал себя с двумя соседними активными процессами по часовой стрелке и против нее. Таким образом, если его номер являлся локальным максимумом, он оставался в круге, иначе он становился пассивным. Активным узлом считается узел, который создает сообщения с новой информацией. Пассивный узел лишь передает полученные сообщения соседу, никак их не изменяя.

В ориентированных кольцах сообщения можно посылать только по часовой стрелке, что затрудняет получение номера соседнего актив-

ного процесса, находящегося в этом направлении. Будем считать, что направление обхода в кольце по часовой стрелке, и только по этому направлению посылаются все сообщения (рис. 6.1).

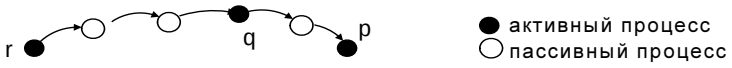


Рис. 6.1. Однонаправленное кольцо с пассивными и активными узлами

В алгоритме Долева, Клаве, Роде на каждом активном узле хранятся два параметра: локальный максимальный вес `maximum` и параметр предыдущего активного соседа, находящегося против часовой стрелки `neighbourR`. Локальный максимальный вес вычисляется по данным от двух предшествующих против часовой стрелки активных соседей.

Алгоритм состоит из повторяющихся раундов, в которых обновляются значения параметров на активных узлах. За один раунд не менее половины активных узлов переходит в пассивное состояние. Именно за счет этого правила достигнуто сокращение общего количества сообщений, необходимых для выбора лидера. Каждый из раундов состоит из двух шагов: на первом шаге `A1` активные узлы получают сообщения типа `one`, а на втором шаге `A2` — типа `two`.

Пассивные узлы только передают по кольцу полученные сообщения.

Опишем последовательность действий активного узла. В начале работы алгоритма все узлы являются активными:

`A0.` `maximum :=` собственный вес. Послать сообщение `one(maximum)` ближайшему соседу.

`A1.` Если пришло сообщение `one(q)`, то:

1. если `q != maximum`, то присвоить `neighbourR` значение `q` и послать сообщение `two(neighbourR)`;
2. иначе `maximum` и есть глобальный максимум. Лидер найден.

`A2.` Если пришло сообщение `two(q)`, то:

1. если `neighbourR` больше и `q`, и `maximum`, то присвоить `maximum` значение `neighbourR` и послать сообщение `one(maximum)`;
2. иначе узел становится пассивным.

На каждом узле чередуются раунды `A1`, `A2`, `A1`, `A2` и т.д., пока не будет найден лидер.

Этот протокол непрозрачен. Его непросто понять и тем более невозможно гарантировать без глубокого анализа, что он выполняет задачу нахождения единственного лидера множества процессов при любом числе процессов и любой расстановке их весов. Такая ситуация характерна для параллельных взаимодействующих процессов: мы имеем точное описание того, что делает каждый процесс, но из этого описания невозможно заключить, выполняется ли некоторое глобальное свойство, которое должно быть обеспечено при работе всех процессов. Именно поэтому этот пример используется здесь для демонстрации анализа (симуляции и верификации), который может быть выполнен с помощью пакета Spin.

### 6.1.1. Описание модели протокола на языке Promela

Каждому узлу ставится в соответствие процесс типа `p`. Запускается $N$ таких процессов. У каждого процесса есть один входящий канал `in`, через него приходят сообщения от соседа против часовой стрелки, и один исходящий канал `out`, в который процесс посылает сообщения ближайшему соседу по часовой стрелке.

Процесс `p` является активным, если у него флаг `Active` установлен в `true`. Иначе `p` является пассивным и просто пропускает через себя все получаемые сообщения. Активный процесс посылает свой текущий локальный максимум следующему активному процессу и получает текущий локальный максимум предыдущего активного процесса, используя сообщения типа `one`. Полученный номер сохраняется в переменной `neighbourR`, и если процесс не выбывает из числа активных, он будет текущим локальным максимумом `p` в следующем круге. Чтобы определить, остается ли номер `neighbourR` в кольце, его сравнивают с максимальным номером `maximum`, пришедшим в этот процесс, и активным номером, полученным в сообщении типа `two`. Процесс `p` посылает сообщение `two(neighbourR)`, чтобы следующий активный процесс мог провести такое же сравнение. Исключение возникает, когда на каком-то узле `neighbourR` оказывается равным `maximum`: в этом случае остался один активный процесс, и об этом все процессы оповещаются сообщением `winer(nr)`.

Для перевода модели на язык описания Promela сделаны допущения:

- каналы между процессами будут конечной глубины `L`, например, 10, т. е. как только в канале накопится 10 сообщений, произойдет

переполнение канала, и следующие сообщения не смогут в него поступать до того, как не освободится место;

- количество процессов ограничено `N` 5, так как в языке Promela следует задавать какое-либо произвольное, но конечное число процессов;
- все процессы подключаются одновременно, т. е. никакой процесс не может включиться в выбор лидера на более поздней стадии.

В данной модели вес процесса не может быть больше количества процессов. Номер процесса, которому присвоен наименьший вес, обозначен `I`.

### 6.1.2. Модель алгоритма выбора лидера на языке Promela

```
1  #define N 5
2  #define L 10

3  #define I 3

4  /* три возможных типа сообщений */
5  mtype = one, two, winner;

   /* массив асинхронных каналов с буфером размера L */
6  chan q[N] = [L] of mtype, byte;

7  /* переменная, равная количеству процессов, считающих себя
   лидерами кольца */
8  byte nr_leaders = 0;

9  /* определение процесса */
10 proctype node(chan in, out; byte mynumber) {
     /* при значении Active, равном 1, процесс активен */
11   bit Active = 1,
     /* флаг know_winner — известен ли лидер */
     know_winner = 0;

     /* объявляется три переменные типа byte, одной присваивается значение mynumber */
```

```
12  byte nr, neighbourR, maximum = mynumber;

13  /* В Promela если по алгоритму процесс будет единственным
    осуществляющим запись в канал (или чтение из канала),
    для уменьшения числа состояний рекомендуется запросить
    эксклюзивный доступ на запись (или на чтение), что здесь и
    сделано. */
    /* запрашивается эксклюзивный доступ к получению сообще-
    ний из канала in */
14  xr in;
    /* запрашивается эксклюзивный доступ к посылке сообщений
    по каналу out */
15  xs out;
16
17  printf("MSC: %d \n," mynumber);  /* отладочная печать */

    /* посылаем сообщение типа one с параметром mynumber —
    номером данного процесса */
18  out ! one(mynumber);

    /* метка end показывает, что не будет ошибкой, если процесс в
    итоге остается в этом цикле вечно */
19  end: do

    /* если получаем сообщение типа one с номером, то записываем
    номер в переменную nr и смотрим: */
20  :: in?one(nr)->
21    if
22    :: Active ->                    /* если процесс активен,
                                       то */
23      if
        /* если максимум еще не найден*/
24      :: nr != maximum ->
```

```
          /* то передаем сообщение типа two с полученным
          номером соседа */
25        out ! two(nr);
26        neighbourR = nr
        /* если максимум найден (т. е. возникла ситуация, когда
        остался один активный процесс)*/
27      :: else ->
28        /* убедимся, что этот максимум равен количеству
          процессов */
29        assert(nr == N);
30        know_winner = 1;  /* и отметим, что теперь
                               известен лидер */
          /* передадим сообщение типа winner с весом узла,
          который признан лидером */
31        out ! winner,nr;
32      fi
33    :: else ->            /* если же процесс не
                               активен*/
34      out ! one(nr)       /* то просто пропускаем
                               сообщения дальше*/
35    fi

36  /* если получаем сообщение типа two, то записываем получен-
    ный номер в переменную nr */
37  :: in?two(nr) ->
38    if
    /* если процесс активен, то сравниваем его номер с
    соседним активным слева и локальным максимумом */
39    :: Active ->
40      if
        /* если вес соседа оказывается новым локальным
        максимумом */
```

```
41      :: neighbourR > nr && neighbourR > maximum ->
42        maximum = neighbourR;    /* сохраняем значение
                                    нового максимума */
43        out ! one(neighbourR)    /* передаем вес дальше
                                    */
        /* иначе процесс становится пассивным */
44      :: else ->
45        Active = 0
46      fi
47    :: else ->                   /* если же процесс не
                                    активен */
48      out ! two(nr)              /* то пропускаем сооб-
                                    щения дальше*/
49    fi

    /* если получили сообщение о том, что лидер выбран, то: */
50  :: in ? winner,nr ->
51    if
52    :: nr != mynumber ->         /* если текущий процесс
                                    — не лидер */
53      printf("MSC: LOST\n");
54    :: else ->                   /* если текущий процесс
                                    является лидером */
55      printf("MSC: LEADER\n");
56      nr_leaders++;              /* количество лидеров
                                    увеличилось*/
57      assert(nr_leaders == 1)    /* проверим, что лиде-
                                    ров в системе только 1
                                    */
58    fi;
59    if
      /* если процесс уже знал о том, что лидер выбран, то
      этот процесс его и назначил. Значит, мы уже обошли все
      кольцо, и нужно заканчивать работу*/
```

```
60      :: know_winner
        /* иначе передадим сообщение о лидере дальше*/
61      :: else -> out!winner,nr
62      fi;
63      break
64    od
65  }

66  /* инициализирующий процесс */
67  init
68  { byte proc;
      /* одной атомарной операцией запускаем N копий
      процесса node*/
69    atomic
70    { proc = 1;
71      do
72      :: proc <= N ->
          /* каждые 2 соседних процесса связывают 2
          уникальных канала in и out */
73        run node(q[proc-1], q[proc%N], (N+I-proc)%N+1);
74        proc++
75      :: proc > N ->
76        break
77      od
78    }
79  }
```

### 6.1.3. Параметры симуляции

Для того чтобы ознакомиться с параметрами симуляции, надо выбрать в меню *Run* пункт *Set Simulation Parameters*. Откроется диалоговое окно *Simulation Options* (см. рис. 6.2).

На панели режима отображения (*Dispaly Mode*)задается, какие окна необходимо выводить при симуляции.

- *MSC Panel* — панель диаграммы взаимодействия. Для данного окна можно задать, как будут отображаться надписи на диаграмме.
  - *Step Number Labels* — на диаграмме указываются номера шагов симуляции.

  - *Source Text Labels* — на диаграмме указываются строчки исходного кода.
- *Time Sequence Panel* — панель временной последовательности — показывает, какие действия выполняет каждый процесс в определенный момент времени.
  - *Interleaved Steps* — в каждый момент времени на диаграмме выводится номер процесса, который выполняется, и строка кода, которая выполняется в этом процессе.
  - *One Window per Process* — для каждого процесса открывается окно, содержащее описание процесса на языке Promela. Во время симуляции подсвечивается строка, которая в данный момент времени выполняется.
  - *One Trace per Process* — для каждого процесса открывается окно. Во время симуляции выводится номер строки, которая в данный момент времени выполняется.
- *Data Values Panel* — панель вывода значений данных.
  - *Track Buffered Channels* — вывод значений каналов с буфером.
  - *Track Global Variables* — вывод значений глобальных переменных.
  - *Track Local Variables* — вывод значений локальных переменных.
  - *Display vars marked* `show` *in MSC* — вывод изменения значения переменной на MSC диаграмме. Для отображения переменной необходимо перед объявлением этой переменной поставить префикс `show` (например, написать `show byte cnt` вместо `byte cnt`).
- *Execution Bar Panel*— панель просмотра процента выполненных шагов каждым процессом в зависимости от общего числа выполненных шагов.

На панели *Simulation Style* устанавливается тип симуляции. Возможны три вида симуляции:

- *Random* — все недетерминированные решения определяются случайным образом.
- *Guided* — симуляция для сгенерированного верификатором ошибочного пути. Данная симуляция используется для отладки ошибки, найденной верификатором.
- *Steps Skipped* — число первых шагов, которые пропускаются.

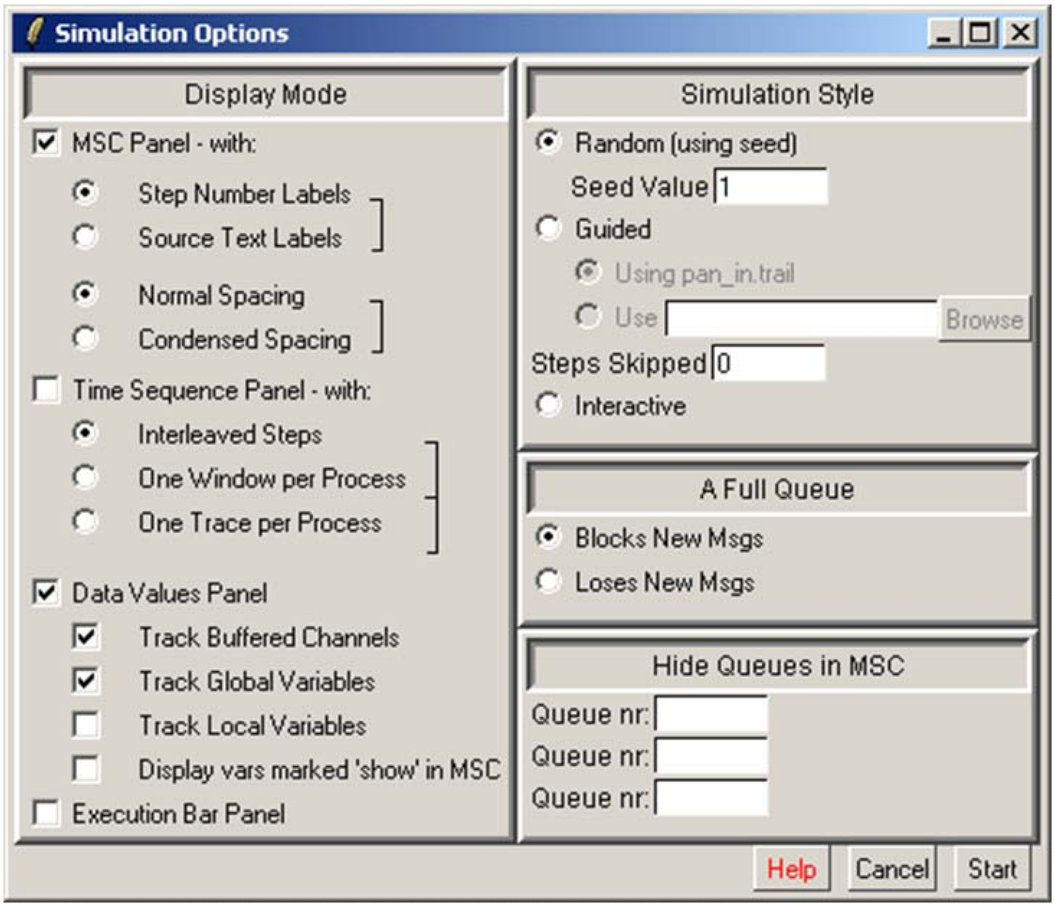


Рис. 6.2. Окно установки параметров симуляции

- *Interactive* — все недетерминированные решения запрашиваются у пользователя.

### 6.1.4. Пример симуляции модели выбора лидера

На рис. 6.3 и 6.4 изображено взаимодействие процессов по модели на Promela, описанной ранее. Каждому из процессов Spin присваивает свой номер, начиная с 0, например, `node: 3`. Согласно модели каждый процесс имеет уникальный вес от 1 до 5 (он изображен ниже номера процесса, определенного Promela).

Процесс с весом 5 первым посылает свой номер по каналу соседнему процессу (шаг 22). Прежде чем начать принимать соседние значения, все остальные процессы делают то же самое (24, 29, 34, 36).

Процесс с весом 4 отправляет свой номер следующему с весом 3 (шаг 24), потом получает номер от процесса с весом 5 (для него предыдущего) (шаг 28). Так как полученное сообщение типа `one`, и этот номер не равен сохраненному максимуму (четырем), передаем этот номер в сообщении типа `two` следующему процессу (шаг 41). То же

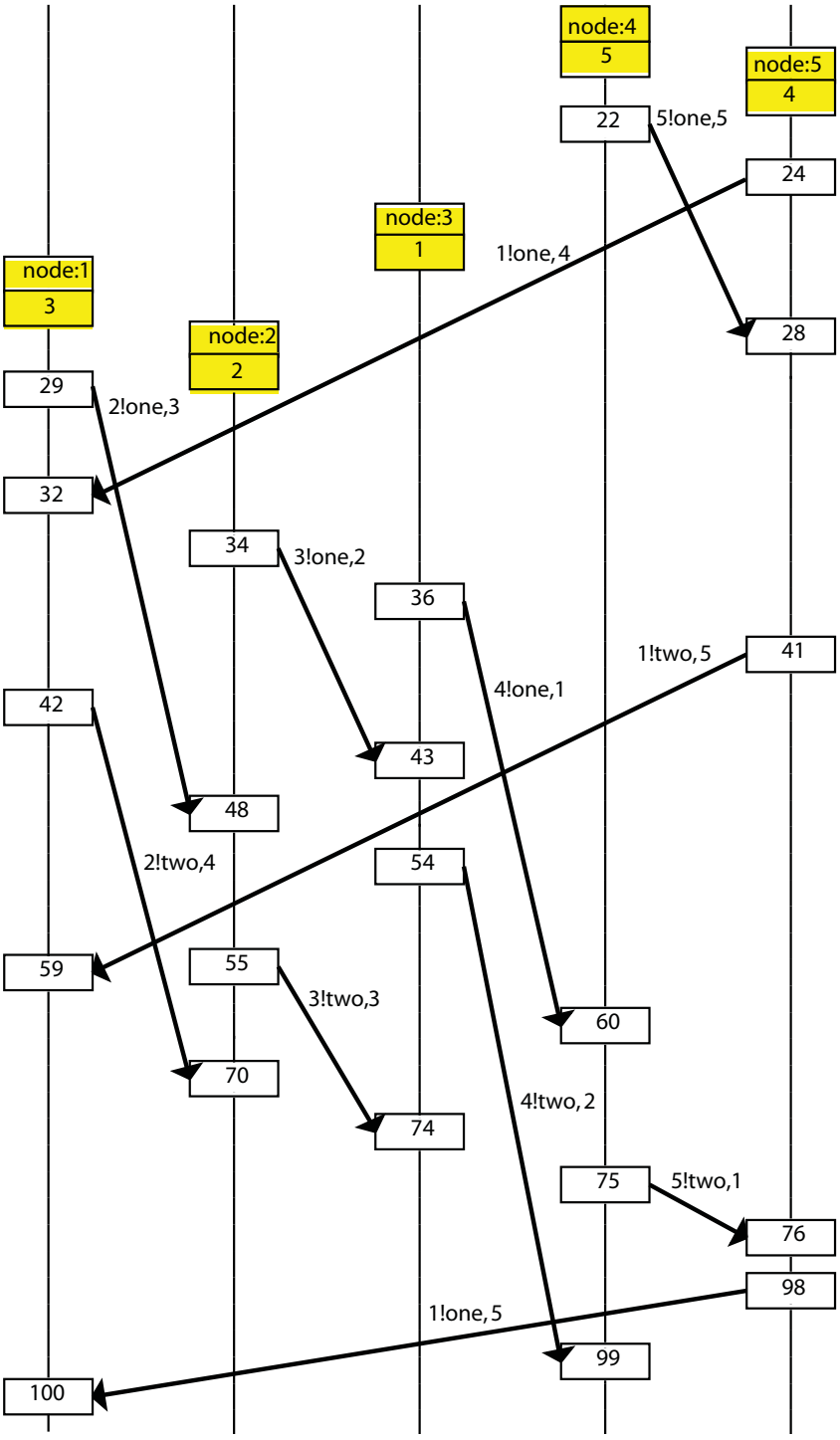


Рис. 6.3. Симуляция модели выбора лидера

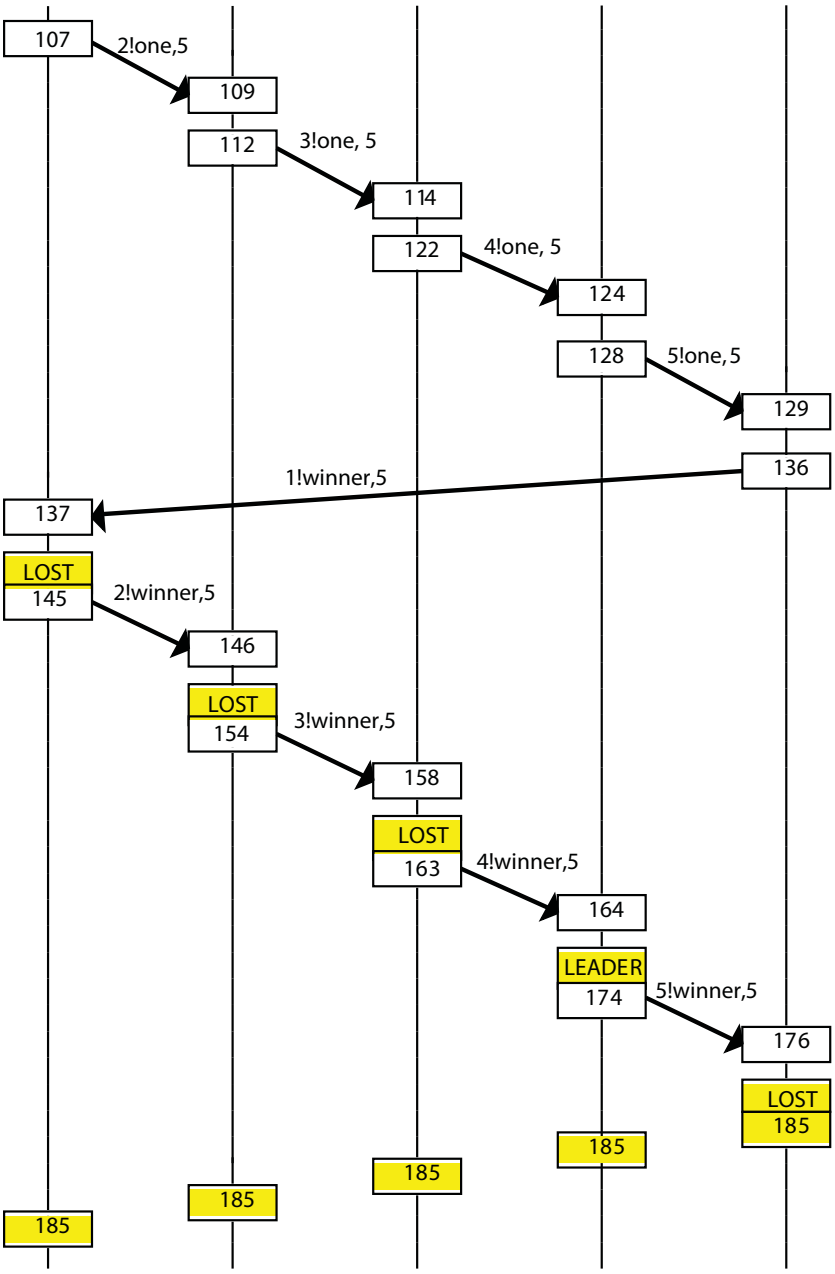


Рис. 6.4. Продолжение симуляции модели выбора лидера

самое делают и все остальные процессы (шаги 42, 55, 54, 75), так как максимум пока не может быть достигнут.

Когда процесс с весом 4 получает сообщение типа `two` со значением 1 (шаг 76), он сравнивает сохраненный номер активного соседа (5, получен на шаге 28) и свой максимум (4, т.к. он еще не менялся) с полученным значением. Получается, что номер активного соседа больше в обоих случаях, следовательно, это локальный максимум, который сохраняется и отправляется в следующем сообщении (шаг 98).

Процесс с весом 3, получив сообщение типа `two` со значением 5 (шаг 59), сравнивает его с сохраненным номером активного соседа (4) и своим максимумом (3). Получается, что номер активного соседа меньше полученного, поэтому процесс 3 становится пассивным и дальше только пропускает через себя сообщения (шаги 107 — 129, 136 — 176). То же самое происходит и с процессом с весом 2 после шага 70, и с процессом с весом 1 после шага 74, и, наконец, с процессом, имеющим вес 5 после шага 99.

Таким образом, сообщение типа `one` со значением 5 возвращается обратно (шаги 107 — 129) к процессу с весом 4. Процесс проверяет, равно ли полученное значение локальному максимуму, и сообщает остальным, что лидер выбран (шаги 136 — 176).

После окончания симуляции в окне *Simulation Output* выведены все сообщения от Spin. В данных сообщениях для каждого шага симуляции показано, какой процесс выполнялся и какая строчка модели протокола на Promela была выполнена.

В окне *log*, ниже основного окна, выводятся строчки:

```
.starting simulation.
spin -X -p -v -g -l -s -r -n1 -j0 pan_in
.at end of run.
```

Они отображают команды, синтезированные XSpin для выполнения симуляции с указанными параметрами.

Для закрытия всех окон симуляции нажмите кнопку *Cancel* на панели *Simulation Output*.

### 6.1.5. Параметры верификации базовых свойств

Установить параметры базовой верификации можно в диалоговом окне *Basic Verification Options* (см. рис. 6.5), выбрав пункт *Set Verification Parameters* меню *Run*.

На панели *Correctness Properties* указываются базовые свойства, которые можно верифицировать в Spin.


Рис. 6.5. Окно параметров верификации базовых свойств Spin

- *Safety* — свойства безопасности:
  - *Assertions* — проверка сохранения локальных инвариантов, описанных при помощи оператора `assert` (см. разд. 5.1);
  - *Invalid Endstates* — проверка на наличие некорректных конечных состояний, т. е. обнаружение взаимных блокировок процессов (см. разд. 5.2).
- *Liveness* — свойства живости:
  - *Non-Progress Cycles* — проверка отсутствия бесконечных циклов, не содержащих операторов, помеченных меткой `progress` (см. разд. 5.3);
  - *Acceptance Cycles* — проверка отсутствия циклов с бесконечно частым выполнением операторов с меткой `accept` (см. разд. 5.4).
- *Apply Never Claim* — учитывать процесс `never`, если таковой имеется в тексте модели (см. разд. 5.5).

- *Report Unreachable Code* — выводить сообщения о недостижимом коде.
- *Check xr/xs Assertions (channel assertions)* — проверять, что только процесс, в котором описан канал с использованием оператора `xr`, имеет право на чтение данных из канала (см. стр. 153). Для задания эксклюзивного права на запись данных в канал используется оператор `xs`.

### 6.1.6. Верификация базовых свойств

Запустите верификацию базовых свойств, нажав кнопку *Run* на панели *Set Verification Parameters*. Верификация модели `leader` должна пройти без ошибок, и в открывшемся окне *Verification Output* выводится сообщение `errors: 0`. Закройте окно верификации.

Введем искусственную ошибку в программу, добавив в строку 22 оператор `assert(false)`. В результате строка будет выглядеть:
```
:: Active -> assert(false);
```

Снова запустим верификацию. Теперь Spin обнаружит ошибку и предложит посмотреть контрпример (рис. 6.6), ее демонстрирующий: *Run Guided Simulation* (запустить управляемую симуляцию).

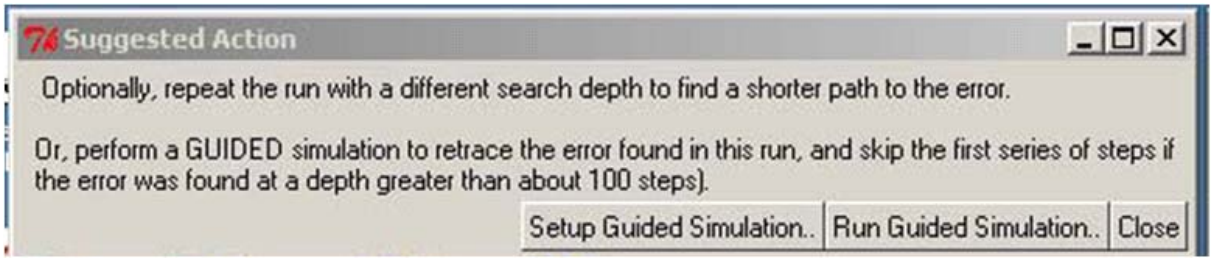


Рис. 6.6. Сообщение о просмотре контрпримера

Если выбрать изменение установок управляемой симуляции *Setup Guided Simulation*, то XSpin снова выдаст панель задания параметров симуляции, все опции в которой, кроме одной, остались без изменений. XSpin изменил лишь тип симуляции (*Simulation Style*) со случайной (*Random Simulation*) на управляемую (*Guided Simulation*).

Если выбрать *Run Guided Simulation*, затем *Run* в окне *Simulation Output*, то XSpin покажет путь, который привел к ошибке (см. рис. 6.7). В этот раз симуляция завершается в момент, когда нарушен `assert`, а не по достижению всеми процессами своих конечных состояний.

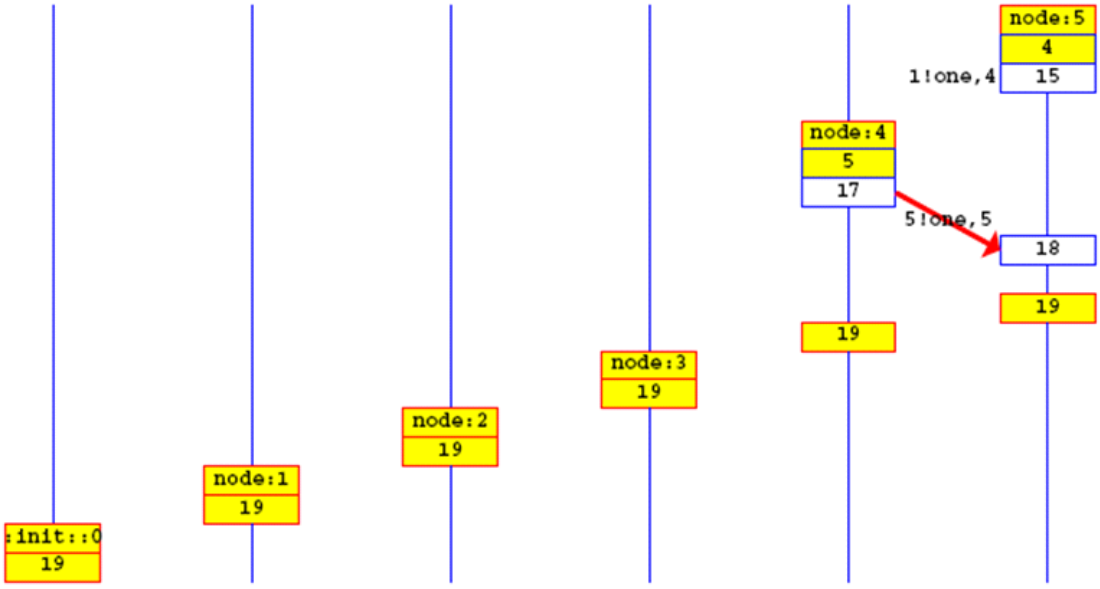


Рис. 6.7. Вывод с нарушением `assert`

После завершения симуляции удалим добавленный оператор `assert`, т. е. вернем строку 22 к прежнему виду `:: Active ->`.

### 6.1.7. Проверка LTL формул

Пользователь может определить свои специфичные требования к модели в Spin, выразив их в виде LTL формул. Обозначение темпоральных операторов в редакторе LTL формул (*LTL Property Manager* меню *Run*), где `q` — атомарный предикат:

| | | |
|---|---|---|
| `<>q` | `Fq` | Хотя бы раз в будущем `q` |
| `[]q` | `Gq` | Всегда в будущем `q` |
| `w U q` | `w U q` | `w` до тех пор, пока `q` |

При открытии окна редактора формул по умолчанию XSpin пытается считать файл с расширением `.ltl` и именем файла на Promela, загруженного в основном окне (см. рис. 6.8). В нашем случае будут загружены данные из `leader.ltl`.

Проверим требование, что **лидер должен быть только один**, т. е. что в нашей модели не может быть выбрано два и более лидера.

Введем новую формулу в окно *Formula* редактора:

```
[] noMore
```

Данное свойство означает, что выражение, заданное с помощью атомарного предиката `noMore`, должно быть всегда истинно. Можно задать, является ли выполнение данной формулы желаемым пове-

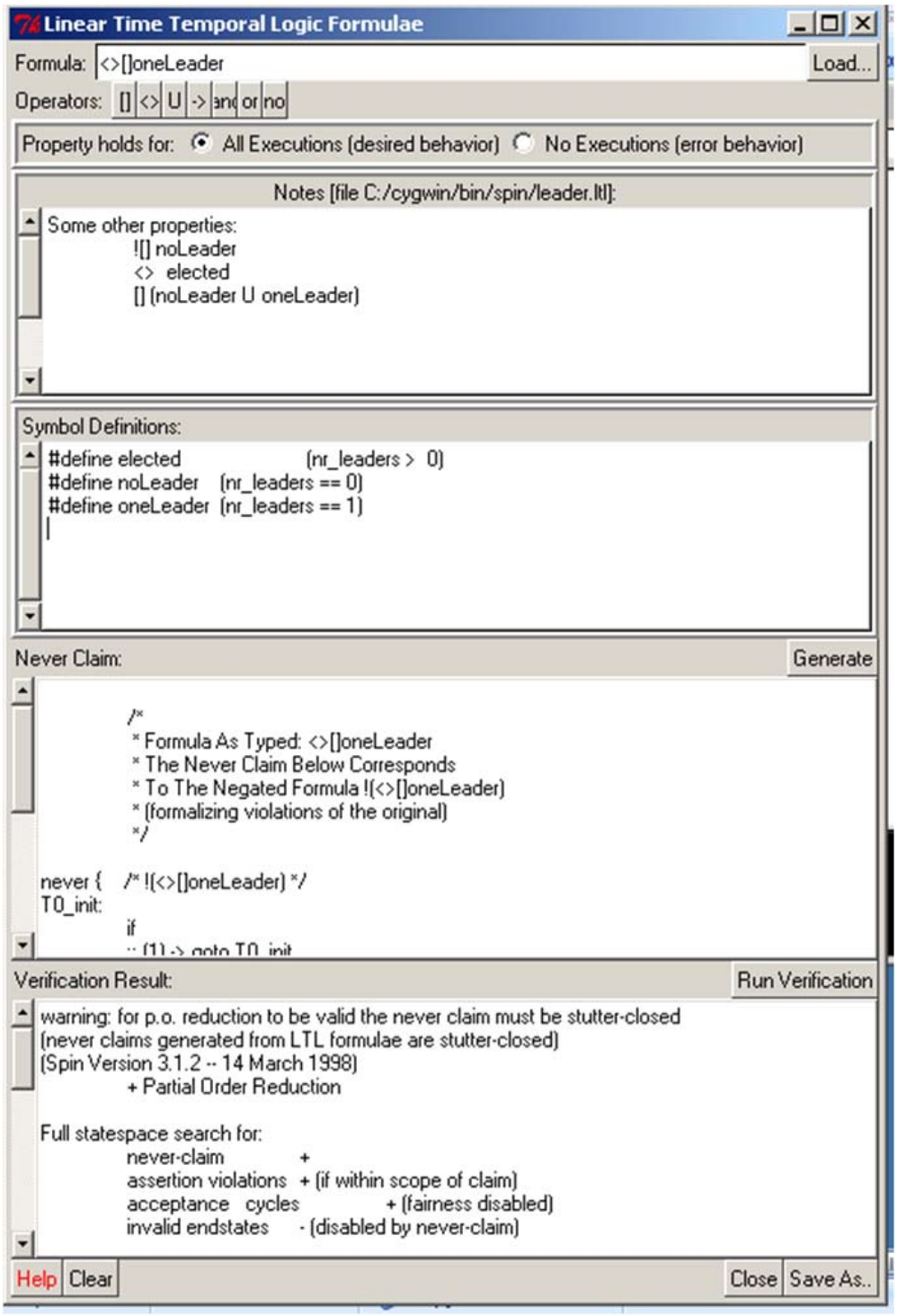


Рис. 6.8. Рабочее окно верификации LTL формул

дением системы или ошибочным: в первом случае в *Property holds for* выбирается пункт *All Executions*, иначе — *No Executions*. Так как выполнение свойства является желаемым, выберем пункт *All Executions*.

Атомные предикаты, подобные `noMore`, задаются в окне *Symbol Definition Box*:

```
#define noMore (nr_leaders <= 1)
```

По нажатию кнопки *Generate* в окне *Never Claim* для заданной LTL формулы генерируется особый процесс `never`, с включенным в него **отрицанием** формулы:

```
/*
* Formula As Typed: []  notMore
* The Never Claim Below Corresponds
* To The Negated Formula !([]  notMore)
* (formalizing violations of the original)
*/
never {    /* !([]  notMore) */
T0_init:
    if :: (! ((notMore))) -> goto accept_all
       :: (1) -> goto T0_init
    fi;
accept_all: skip }
```

Результаты верификации будут положительные — в том смысле, что данная формула выполняется на данной модели:

```
Full statespace search for:
  never claim          +
  assertion violations + (if within scope of claim)
  acceptance   cycles  + (fairness disabled)
  invalid end states - (disabled by never claim)
State-vector 200 byte, depth reached 205, errors: 0
  97 states, stored
   1 states, matched
  98 transitions (= stored+matched)
  12 atomic steps
hash conflicts: 0 (resolved)
unreached in proctype node
```

```
    line 53, "pan.___", state 28, "out!two,nr"
    (1 of 49 states)
unreached in proctype :init:
    (0 of 11 states)
unreached in proctype :never:
    line 104, "pan.___", state 8, "-end-"
    (1 of 8 states)
```

Spin выводит достаточно много дополнительной информации (число состояний, вектор глобального состояния, недостижимые участки кода и т. п.), которая необходима для оптимизации модели.

Таким образом, мы убедились, что лидер выбирается всегда только 1. Аналогичную проверку можно сделать и с помощью оператора `assert` и базовой верификации (см. модель алгоритма в разделе 6.1.2): `57 assert(nrleaders == 1)`. Строчка идет в коде сразу после добавления нового лидера, чтобы Spin проверил, что добавленный лидер оказывается единственным в модели.

Проверим, что **лидер в конце концов будет выбран** при помощи LTL формулы (см. рис. 6.8): `<>[]oneLeader`, где в качестве атомного предиката задано:

```
#define oneLeader (nr_leaders == 1)
```

Эта формула означает, что когда-нибудь наступит момент, после которого количество лидеров всегда будет один, т. е. лидер будет выбран. Свойство на этой модели выполняется.

Проверим, что **номер выбранного лидера всегда будет максимальным**, т. е. в данной модели равным количеству процессов.

Это свойство легче всего проверить с помощью добавления оператора `assert(nr == N)` в том месте кода, где считается, что лидер найден (см. строчку 29 в коде на стр. 154), `nr` — номер процесса, который признан лидером в результате работы алгоритма, а `N` — число процессов в модели. В рассматриваемой модели свойство выполняется.

Немного модифицируем модель так, чтобы не все процессы участвовали в выборе. Введем глобальные переменные: `election_started=0` и `first_message=1`. Если раньше каждый процесс в начале работы был активен и отправлял свой идентификатор в канал, то теперь все процессы в начале неактивны (`Active=0`) и ничего не отправляют в канал (уберем 18-ю строчку). Лишь

один, недетерминированно выбранный процесс, инициирует протокол, установив `election_started=1` (между 19-й и 20-й строками добавим):

```
:: atomic {
  (!election_started && !Active) ->
  /* Один процесс может начать выборы, послав первое сообще-
  ние msg и став активным */
  election_started = 1;
  out ! one(mynumber);
  first_message = 0;
  Active = 1}
```

Далее один из неактивных процессов, первым обнаруживший, что `first_message=1`, может вступить в выборы, а потом установить `first_message=0` (добавим следующий код между 20-й и 21-й строчками и 37-й и 38-й строчками):

```
if
:: (first_message && !Active) ->
  if
  :: Active = 1;  /* присоединиться к выборам */
    out ! one(mynumber)
  :: skip         /* не присоединяться к выборам */
  fi
:: else
fi;
first_message = 0;
```

Проверим корректность измененной модели относительно свойства, что номер лидера всегда будет максимальным:

```
pan: assertion violated (nr==5) (at depth 45)
pan: wrote pan_in.trail
(Spin Version 4.2.7 -- 23 June 2006)
Warning: Search not completed
  + Partial Order Reduction
Full statespace search for:
  never claim          - (not selected)
  assertion violations +
```

```
  cycle checks        - (disabled by -DSAFETY)
  invalid end states - (disabled by -E flag)
State-vector 196 byte, depth reached 45, errors: 1
  33 states, stored
   0 states, matched
  33 transitions (= stored+matched)
  13 atomic steps
hash conflicts: 0 (resolved)
```

Из результатов видно, что в поведении модели существуют такие пути, что оператор `assert(nr == N)` не выполнился (`assertion violated`). Причина нарушения свойства кроется в изменении модели: теперь не все процессы участвуют в голосовании, т. е. будет выбираться не абсолютный максимальный вес, а максимальный вес из числа участвующих процессов.

Остальные же свойства, проверявшиеся ранее, и на измененной модели останутся истинными.

## 6.2. ЗАДАЧА О ФЕРМЕРЕ, ВОЛКЕ, КОЗЕ И КАПУСТЕ

В качестве второго примера рассмотрим задачу, хорошо известную всем со школьных лет: фермер, волк, коза и капуста находятся на левом берегу реки. Фермер хочет переправить волка, козу и капусту на правый берег. В лодку вместе с собой он может взять лишь одного из них (либо волка, либо козу, либо капусту). Он может плыть в лодке и один. Однако если волк останется наедине с козой на одном берегу, то он может ее съесть. Коза, оставшись вместе с капустой, поедает капусту. Существует ли такая последовательность переправ фермера, чтобы доставить всех на правый берег целыми и невредимыми?

Покажем, что с помощью системы Spin можно получить решение этой задачи, сформулировав в качестве проверяемого свойства модели то, что **не существует решения с желаемыми свойствами**. Как результат верификации Spin выдаст котрпример — траекторию поведения системы <фермер—волк—коза—капуста>, которая нарушает проверяемое свойство. Этот контрпример и будет являться решением задачи.

### 6.2.1. Модель задачи о фермере, волке, козе и капусте на языке Promela

В данной модели достаточно одного единственного процесса `river`. За местоположение каждого из участников задачи пусть отвечает своя битовая переменная: всего будет четыре такие переменные (`f`, `w`, `g`, `c`). Будем считать, что значение переменной равно 0, если соответствующий участник находится на левом берегу, и равно 1, если участник находится на правом берегу. Применим недетерминированный подход к моделированию задачи о волке, козе и капусте. Недетерминированно выбираются партнеры фермера по переправе из числа тех участников задачи, которые находятся с ним на одном берегу. Например, изначально партнерами фермера могут быть или волк, или коза, или капуста, или же он может поехать один. Повторяя такой выбор в цикле, фермер в принципе бесконечно будет перевозить своих подопечных с левого берега на правый и с правого берега на левый. Поскольку по условию задачи все должны быть переправлены на правый берег, то добавим условие выхода из цикла:

```
(f==1) && (f == w) && (f ==g) && (f == c)
```

Это условие будет истинным, когда все битовые переменные равны 1 (т. е. волк, коза, капуста, фермер оказались на правом берегу). С его помощью можно остановить выполнение цикла всех возможных переправ.

В результате получится следующая модель задачи на языке Promela:

```
/* булевы переменные, используемые в LTL формулах */
bool all_right_side, g_and_w, g_and_c;
active proctype river(){
  /* вначале волк, коза, капуста и фермер находятся на левом
  берегу реки */
  bit f = 0,  /* положение фермера */
    w = 0,    /* положение волка*/
    g = 0,    /* положение козы*/
    c = 0;    /* положение капусты*/
  all_right_side = false;
```

```
g_and_w = false;
g_and_c = false;
printf(«MSC: f %c w %c g %c c %c \n», f, w, g, c);
do :: (f==1) && (f == w) && (f ==g) && (f == c) ->
  all_right_side = true;     /* все на правом берегу */
  break;                     /* завершаем цикл*/
:: else ->
  if                         /* недетерминированный выбор */
  :: (f == w) ->             /* фермер и волк на одном берегу
                                реки */
    f = 1 - f;               /* фермер переправляет волка */
    w = 1 - w;
  :: (f == c) ->             /* фермер и капуста на одном
                                берегу реки */
    f = 1 - f;               /* фермер переправляет капусту */
    c = 1 - c;
  :: (f == g) ->             /* фермер и коза на одном берегу
                                реки */
    f = 1 - f;               /* фермер переправляет козу */
    g = 1 - g;
  :: (true) ->               /* в любой ситуации */
    f = 1 - f;               /* фермер может пересечь реку
                                один */
  fi;
  printf(«MSC: f %c w %c g %c c %c \n», f, w, g, c);
  if                         /* проверяем выполнение ограни-
                                чений */
  ::(f != g && g == c ) ->   /* съела ли коза капусту? */
    g_and_c = true;
  ::(f != g && g == w ) ->   /* съел ли волк козу? */
    g_and_w = true;
  ::else -> skip
  fi
od; printf(«MSC: OK! \n»)
```

### 6.2.2. Поиск решения задачи средствами Spin

При запуске симуляции обычно получаются реализации, где волк может съесть козу, а коза — капусту; и более того, такие случаи могут встретиться несколько раз. Эти результаты связаны с тем, что на поведение участников не накладывались никакие ограничения. Приведем пример одного из прогонов симуляции. При выводе используются соглашения, принятые ранее: значение переменной, отражающей местоположение участника, равно 0, если участник находится на левом берегу, и 1, если на правом. Например, на рис. 6.9 сначала волк и коза остаются на левом берегу без фермера (`f = 1 w = 0 g = 0 c = 1`), а через несколько шагов коза и капуста — на правом (`f = 0 w = 0 g = 1 c = 1`).

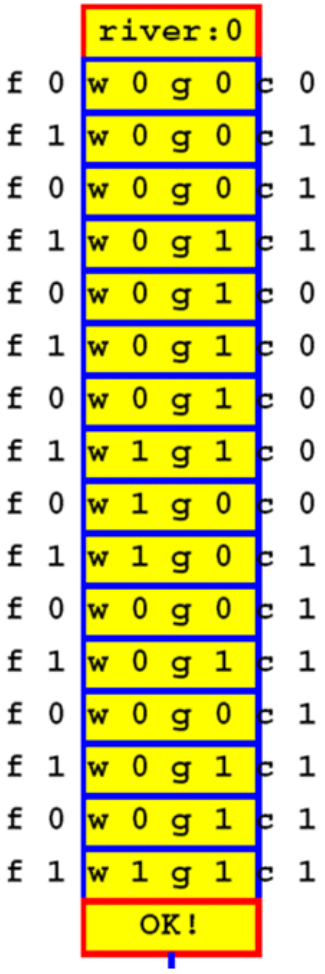


Рис. 6.9. Возможное вычисление задачи о фермере, волке, козе и капусте

Обычно Spin используется как средство верификации модели. Однако в данном случае представляет интерес задача обнаружения во всем разнообразии переправ, при которой фермер перевезет все свое имущество (волка, козу и капусту) на правый берег реки в целости. Зададим следующую LTL формулу:

```
<>  fin && [] ( wg  && gc )
```

где значения предикатов таковы (`all_right_side`, `g_and_w`, `g_and_c` — булевы переменные, определенные в тексте программы):

```
#define fin (all_right_side == true)
#define wg (g_and_w       == false)
#define gc (g_and_c        == false)
```

Предположим, что не существует ни одной реализации, в которой выполняется заданное правило: «Когда-нибудь в будущем все участники задачи окажутся на правом берегу, и при этом всегда будет выполняться условие, что волк не съест козу, и коза не съест капусту». Установим флаг *No Executions* в пункте *Property holds for* окна проверки LTL формул.

При верификации Spin выдаст ошибку, сообщая тем самым, что он обнаружил ветку поведения процесса `river`, в которой заданное условие выполняется:

```
State-vector 20 byte, depth reached 73, errors: 1
  56 states, stored (57 visited)
  8 states, matched
  65 transitions (= visited+matched)
  0 atomic steps
hash conflicts: 0 (resolved)
```

Spin предложит перейти к управляемой симуляции (*Run Guided Simulation*). При прогоне управляемой симуляции отображено одно из возможных решений задачи (см. рис. 6.10):

1. сначала все вчетвером находятся на левом берегу реки:
   `f = 0 w = 0 g = 0 c = 0;`
2. фермер отвозит козу на правый берег, волк и капуста остались на левом берегу:
   `f = 1 w = 0 g = 1 c = 0;`

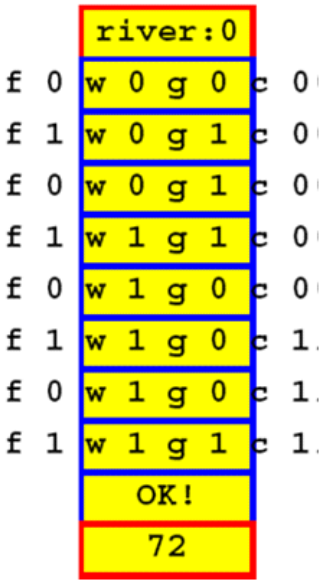


Рис. 6.10. Решение задачи о фермере, волке, козе и капусте

3. фермер один возвращается на левый берег:
   `f = 0 w = 0 g = 1 c = 0;`
4. фермер забирает волка и переплывает с ним на правый берег:
   `f = 1 w = 1 g = 1 c = 0;`
5. фермер с козой переправляется на левый берег:
   `f = 0 w = 1 g = 0 c = 0;`
6. взяв капусту, фермер плывет на правый берег:
   `f = 1 w = 1 g = 0 c = 1;`
7. за оставшиеся шаги фермер переправляет на правый берег козу.

Таким образом, система верификации Spin **сгенерировала** решение задачи о фермере, волке, козе и капусте, как контрпример свойства «интересующего нас решения задачи **не существует**».

## 6.3. КРИПТОГРАФИЧЕСКИЙ АЛГОРИТМ НИДХАМА — ШРЕДЕРА

Рассматрим алгоритм аутентификации — построения доверительных отношений между двумя партнерами A и B, широко применяемый в разных пакетах, например в Kerberos. Алгоритм, предложенный в 1978 г. Нидхамом (Needham) и Шредером (Schroeder), использует криптографию публичных ключей. Каждый участник имеет пару ключей. Один ключ, открытый или публичный, известен всем другим участникам, другой ключ, закрытый (хранящийся скрытно),

известен только самому владельцу. Связь между двумя ключами такова, что информация, зашифрованная одним ключом, может быть расшифрована только вторым ключом в паре. Таким образом, отправитель может быть уверен в том, что сообщение, зашифрованное им публичным ключом A, сможет прочесть лишь сам A.

В алгоритме аутентификации Нидхама — Шредера стороны обмениваются сообщениями, состоящими из открытой части и зашифрованной части. В открытой части сообщения содержатся сведения об отправителе, получателе сообщения и номер сообщения согласно алгоритму. Вторая часть сообщения шифруется публичным ключом предполагаемого получателя сообщения. В зашифрованной части содержится так называемый nonce — секрет — случайное число, генерируемое каждой стороной для очередного сеанса доверительного обмена. Сторона A генерирует свое случайное число, оно является ее частью их общего секрета. Сторона B генерирует в ответ свое число. В результате работы протокола каждая сторона должна быть уверена в своем партнере и должна узнать его часть секрета.

Полный вариант алгоритма построения доверительных отношений предполагает, что при построении отношений публичные ключи партнеров не хранятся на каждой стороне, за ними надо обращаться к серверу. Без потери общности будем считать, что каждый участник знает публичные ключи всех возможных партнеров.

Сокращенный алгоритм Нидхама — Шредера:

**Сообщение 1.** Сторона A отправляет стороне B сообщение 1, в зашифрованной части которого передает свой идентификатор и nonceA. Сообщение шифруется открытым ключом B.

**Сообщение 2.** Сторона B, получив сообщение 1, расшифровывает его, проверяет соответствует ли идентификатор отправителя в зашифрованной части идентификатору открытой части письма. Если соответствие есть, то сторона B уверена в своем партнере и отправляет сообщение 2, зашифровав публичным ключом A два секрета: nonceA и nonceB.

**Сообщение 3.** Сторона A, получив сообщение 2, расшифровывает его, находит в сообщении свой nonceA, что убеждает ее, что отправителем является B, поскольку только B мог прочитать зашифрованное сообщение 1. Теперь сторона A знает полный секрет, она подтверждает этот факт, посылая стороне B ее часть секрета nonceB в зашифрованной части сообщения 3. Сторона B,

получив сообщение 3, уверена, что, во-первых, ее партнер — A; во-вторых, A знает полный секрет.

Спустя почти 20 лет использования алгоритма, в 1995 г. Лоуве (Lowe) описал модель алгоритма на CSP и верифицировал его при помощи верификатора FDR [25]. Лоуве ввел в процесс обмена сообщениями третью сторону — злоумышленника (intruder), поведение которого не подчинялось определенному алгоритму. Злоумышленник мог выполнять несколько действий с сообщениями:

1. прослушивать сообщения, идущие по каналу;
2. сохранять захваченные сообщения; если сообщения предназначались ему, то он расшифровывает скрытую часть, в противном случае — хранит в том виде, в котором ее получил;
3. генерировать сообщения на основе имеющейся информации;
4. отправлять свои сообщения в сеть.

Введение злоумышленника с описанным поведением оказалось достаточным для того, чтобы обнаружить криптографическую атаку в алгоритме Нидхама — Шредера.

### 6.3.1. Описание модели на языке Promela

Смоделируем протокол Нидхама — Шредера на языке Promela и попытаемся обнаружить атаку, найденную Лоуве, средствами Spin. В модели присутствуют три участника: `Alice` (сторона A), `Bob` (сторона B) и `Intruder` (злоумышленник), которым поставим в соответствие три одноименных процесса. Будем моделировать некорректное поведение алгоритма, поэтому считаем, что все сообщения в сети проходят через злоумышленника. Для обмена сообщениями между процессами введем три рандеву-канала: `fakedA` и `fakedB` (испорченные), `intercepted` (прослушиваемый). Записывая сообщение в `intercepted` канал, `Alice` или `Bob` не знают, что сообщение будет перехвачено. Прочитывая сообщение из своего испорченного канала `fakedA`, `Alice` не предполагает, что сообщение было испорчено (аналогичная ситуация для `Bob`). `Intruder` читает сообщение из канала `intercepted`, записывает сообщение в канал `fakedA` или в канал `fakedB` в зависимости от того, кому предназначается сообщение. Используется лишь один тип сообщения `msg`.

```
/* читающий из канала faked не подозревает, что сообщение
 было испорчено */
```

```
chan fakedA = [0] of {mtype, mesCrypt};
chan fakedB = [0] of {mtype, mesCrypt};
/* пишущий в канал intercepted не знает, что сообщение будет
прочитано */
chan intercepted = [0] of {mtype, mesCrypt};
```

Структура сообщения сложная и состоит из нескольких частей: чтобы ее описать, введем данные такого типа:

```
typedef mesCrypt {
  /* открытая часть сообщения */
  mtype s,  /* отправитель */
  r,        /* получатель */
  nummsg,   /* номер сообщения */
  key,      /* публичный ключ, которым зашифровано сообщение */
  /* зашифрованная часть сообщения */
  d1,       /* здесь может быть идентификатор или nonce */
  d2 };
```

Кроме того, в перечислимых типах еще введены идентификаторы процесса, публичные ключи, случайные числа каждой стороны и состояния участников:

```
mtype = {msg,                /* тип сообщения */
  alice, bob, intruder,      /* идентификаторы участников */
  pkA, pkB, pkI,             /* публичные ключи участников */
  nonceA, nonceB, nonceI,    /* случайные числа участников */
  ok, err};
```

Введем некоторые ограничения в модели для уменьшения числа возможных состояний системы:

- инициирует начало обмена сообщениями `Alice`;
- `Intruder` прослушивает сообщения от `Alice` и `Bob`, поэтому он знает их идентификаторы изначально;
- `Intruder` хранит лишь последнее прослушиваемое сообщение;
- получив сообщение, `Intruder` обязательно отправляет сообщение (прослушанное или сохраненное) в сеть;
- если `Alice` или `Bob` получают сообщение, не соответствующее ожидаемому по любому из признаков (согласно действующему алгоритму), то процесс прекращает работу.

Получаемые из своих каналов данные Alice и Bob хранят в переменной data описанного выше типа mesCrypt. Опишем лишь одну из проверок, которую проходит получаемое сообщение на честной стороне. Например, первое сообщение получает Bob:

```
if :: (data.r == bob) &&    /* Bob ли получатель сообщения ? */
   (data.key == pkB) &&     /* можно ли расшифровать сообщение */
   (data.s == data.d1) &&   /* можно ли доверять отправителю ? */
   (data.nummsg == 1)->     /* правильный ли номер сообщения ? */
   partnerB = data.s;
   pnonce = data.d2;
  :: else -> goto stopB;
fi;
```

Bob проверяет, является ли он получателем сообщения; проверяет номер сообщения (должно прийти первое сообщение); проверяет, что информация зашифрована его ключом. Если все проверки проходят положительно, Bob может расшифровать сообщение. Затем он проверяет, что идентификатор отправителя в открытой части сообщения соответствует идентификатору в зашифрованной части сообщения. Bob запоминает партнера и его часть секрета, только если все эти проверки пройдены, иначе он останавливает работу. Совершенно очевидно, что при таком количестве условий большинство попыток злоумышленника вторгнуться во взаимодействие Alice и Bob закончатся безуспешно. Нас, однако, интересует, существуют ли успешные попытки вторжения. Оказывается, что такие возможности существуют, и они могут быть **сгенерированы** системой верификации точно так же, как было сгенерировано решение в задаче о фермере, волке, козе и капусте.

Intruder использует две переменные типа mesCrypt — переменную data для получения данных из канала, переменную fake для генерации своего сообщения на основе имеющейся информации. Злоумышленник не подвергает приходящие сообщения никаким проверкам, кроме одной: если зашифрованные данные зашифрованы его ключом (data.key == pkI), то он может расшифровать закрытую часть сообщения и узнать ее содержимое (значимые величины nonceA и nonceB). Остальные действия Intruder связаны с составлением своего сообщения, которое он отправит в сеть:

```
active proctype Intruder() {
  /* ... */
  /* бесконечный цикл, в котором читает сообщения из канала
  intercepted */
end: do
  :: intercepted ? msg(data) ->
    /* осуществляет проверку данных data  */
    if  :: (data.key == pkI) ->  /* если может расшифровать
                                    секретную часть, то извлекает
                                    оттуда информацию в откры-
                                    том виде */
          if  :: (data.d1 == nonceA || data.d2 == nonceA) ->
                /* возможно, попался nonceA, зафиксирует,
                что теперь его знает knowNA=1 */
                knowNA = true;
              /* аналогично для nonceB */
          fi;
        :: else -> skip;             /* иначе — ничего не делает */
    fi;
    /* построение испорченного сообщения fake */
    /* недетерминированно выбирает отправителя */
    if  :: fake.s = alice ->         /* если отправитель Alice */
          fake.r = bob;              /* то получатель Bob */
          fake.key = pkB;
        /* аналогично для Bob и самого злоумышленника*/
    fi;
    /* поскольку злоумышленник не знает правила построения со-
    общения, он помещает в часть сообщения d1 любую известную
    ему открытую информацию */
    if  :: fake.d1 = alice;
        :: fake.d1 = bob;
        :: fake.d1 = intruder;
        :: fake.d1 = nonceI;
        :: (knowNA) -> fake.d1 = nonceA;
```

```
        :: (knowNB) -> fake.d1 = nonceB;
        fi;
    /* аналогично он поступает и с частью d2 ... */
    /* злоумышленник может вставить неизвестную зашифрован-
    ную часть из только что полученного сообщения */
    if  :: (data.key != pkI) ->
          fake.key = data.key;
          fake.d1 = data.d1;
          fake.d2 = data.d2;
        :: else -> skip;
    fi;
    /* выбирает номер сообщения ... */
    /* отправляет сообщение или полученное, или свое созданное в
    канал, предназначая его либо Alice, либо Bob */
    if  :: (data.r == alice) ->
          fakedA ! msg(data);  /* повторяет полученное сооб-
                                  щение Alice */
        :: (data.r == bob) ->
          fakedB ! msg(data);  /* повторяем полученное со-
                                  общение Bob */
        /* аналогично для испорченного сообщения ... */
    fi;
  od; }
```

### 6.3.2. Поиск криптографической атаки

В описанном алгоритме атакой можно считать ситуацию, когда сторона B будет считать своим партнером сторону A, сторона A будет доверять своему партнеру, а злоумышленник будет знать обе части их секрета. Сформулируем эту ситуацию в виде LTL формулы: не существует такого пути в системе, что когда-нибудь в будущем сторона B, успешно завершив работу, будет полагать, что ее партнер A, и подслушивающий при этом знает nonceA и nonceB:

```
<> (agentB_finished && bobtrustsA && intrknowNA && intrknowNB)
```

где предикаты заданы следующим образом:

```
#define   agentB_finished   (statusB == ok)
#define   bobtrustsA        (partnerB == alice)
#define   intrknowNA        (knowNA == true)
#define   intrknowNB        (knowNB == true)
```

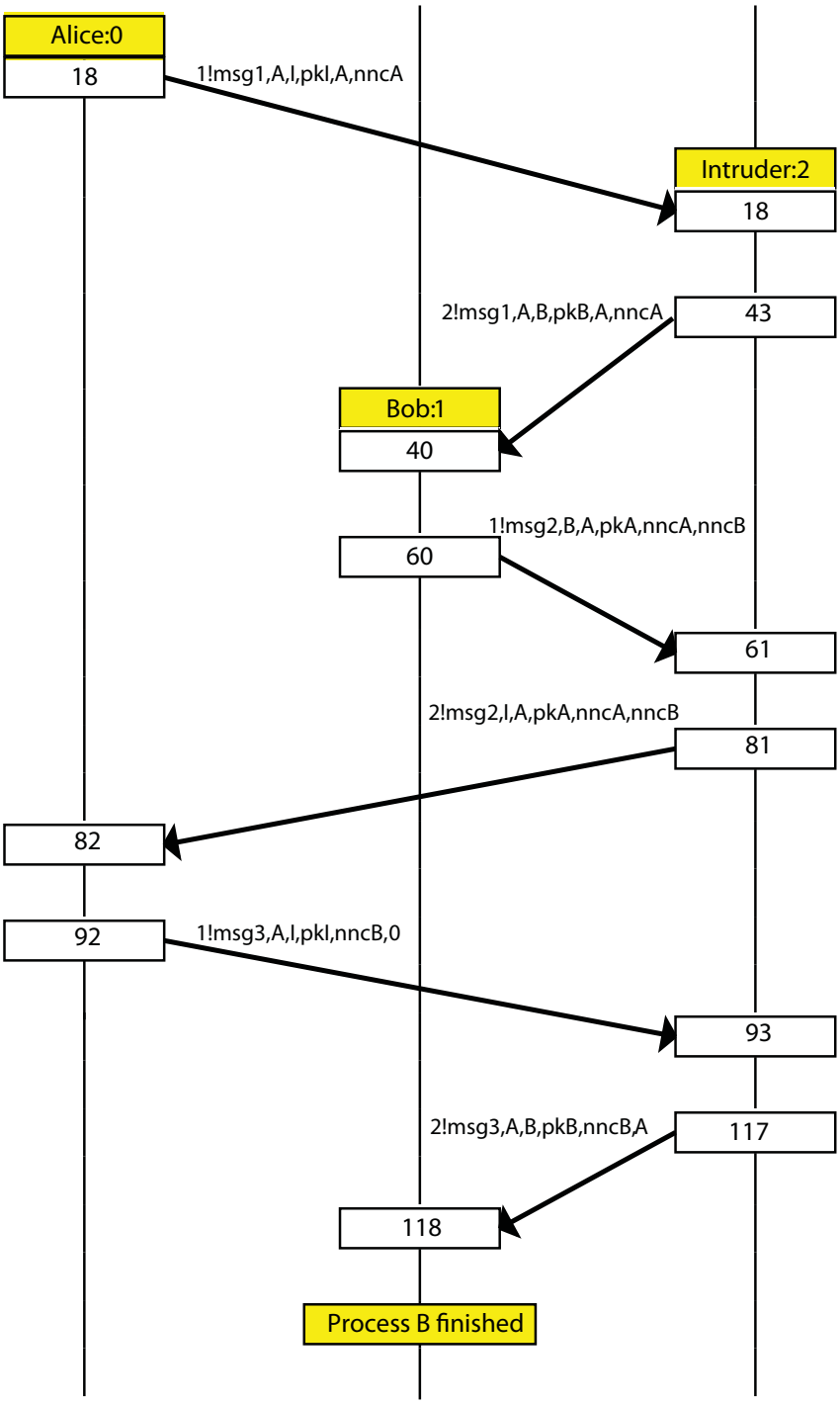


Рис. 6.11. Атака в алгоритме Нидхама — Шредера

переменные statusB, partnerB, knowNA, knowNB определены в теле программы.

Полный код модели алгоритма Нидхама — Шредера можно найти в разделе 6.3.3.

Поставив флаг *No Executions* для заданной формулы, запустим верификатор Spin. Spin обнаружил атаку, описанную Лоуве в 1995 г. (см. рис. 6.11).

1. Alice, считая Intruder честной стороной, посылает ему сообщение № 1 со своим идентификатором и своей частью секретного ключа, зашифровав все публичным ключом Intruder (шаги 18 — 19). Intruder уже известен nonceA.
2. Intruder посылает сообщение № 1 Bob, поставив в открытую часть сообщения в качестве отправителя Alice, а далее повторив ее письмо себе, зашифровав секретную часть публичным ключом Bob (шаги 43 — 44).
3. Bob отвечает Intruder сообщением № 2, полагая, что тот является Alice. В письме содержится часть, непонятная Intruder, поскольку она зашифрована публичным ключом Alice (шаги 60 — 61). В зашифрованной части письма идет nonceB — часть секрета Bob, но она не доступна злоумышленнику.
4. Злоумышленник отправляет сообщение № 2 Alice (шаги 81 — 82). В качестве основы он берет сообщение Bob, зашифрованную часть он вставляет целиком, а в открытой части меняет отправителя на себя.
5. Alice, доверяя Intruder, расшифровывает сообщение № 2, извлекает nonceB, отвечает Intruder сообщением № 3, содержащим nonceB, причем секретная часть зашифрована публичным ключом Intruder (шаги 92 — 93).
6. Intruder расшифровывает сообщение № 3, узнает nonceB, посылает сообщение № 3 Bob. Полученное сообщение проходит все проверки Bob, и он успешно завершает работу (117 — 126).

### 6.3.3. Код алгоритма Нидхама — Шредера на языке Promela

```
/* Протокол Нидхама — Шредера */
/* три участника: Alice, Bob и Intruder, которые обмениваются
сообщениями друг с другом по сети */
```

```
mtype = {msg, alice, bob, intruder, pkA, pkB, pkI,
  nonceA, nonceB, nonceI, ok, err};
/* по сети передается сообщение (отправитель, получатель, номер,
зашифрованные данные (случайные числа или ID))*/
typedef mesCrypt {      /* сообщение */
  mtype s, r, nummsg,   /* открытая часть */
  key, d1, d2;          /* зашифрованная часть */
};
/* читающий из канала faked не подозревает, что сообщение было
испорчено */
chan fakedA = [0] of {mtype, mesCrypt};
chan fakedB = [0] of {mtype, mesCrypt};
/* пишущий в канал intercepted не знает, что сообщение будет
прочитано */
chan intercepted = [0] of {mtype, mesCrypt};
/* переменные, используемые в LTL формулах */
mtype partnerA, partnerB;
mtype statusA, statusB;
/* переменные отражают знание посторонним секретных частей
общего ключа */
bool knowNA, knowNB;
/* честный инициатор процесса обмена */
active proctype Alice() {
  mtype pkey, pnonce;
  mesCrypt data;
  statusA = err;
  if                    /* недетерминированно выбирает партне-
                           ра */
  :: partnerA = bob; pkey = pkB;
  :: partnerA = intruder; pkey = pkI;
  fi;
  /* конструирует сообщение № 1 и отправляет */
  d_step {
    data.s = alice;
    data.r = partnerA;
    data.nummsg = 1;
```

```
    data.key = pkey;
    data.d1 = alice;
    data.d2 = nonceA; }
  intercepted ! msg(data);
  /* ожидает сообщения № 2 и расшифровывает его */
  fakedA ? msg(data);
end_errA:
  /* проверяет, что сообщение предназначалось именно ему и
  что отправитель — его партнер, иначе останавливает процедуру
  обмена */
  if
  :: (data.key == pkA) && (data.d1 == nonceA) &&
    (data.s == partnerA) && (data.r == alice) &&
    (data.nummsg == 2)->
    pnonce = data.d2;
  :: else -> goto stopA
  fi;
  /* отвечает сообщением № 3 и успешно завершается*/
  d_step {
    data.s = alice;
    data.r = partnerA;
    data.nummsg = 3;
    data.key = pkey;
    data.d1 = pnonce;
    data.d2 = 0; }
  intercepted ! msg(data);
  statusA = ok;
stopA:
  printf(«MSC: Process A finished \n»); }
/* честный партнер по обмену */
active proctype Bob() {
  mtype pkey, pnonce, receiver, partner;
  mesCrypt data;
  statusB = err;
```

```
  /* ожидает сообщения № 1, идентифицирует партнера */
  fakedB ? msg(data);
end_errB1:
  /* проверяет сообщение на правильность построения, если что-
  то не соответствует ожиданию, останавливается */
  if
  :: (data.r == bob) && (data.key == pkB) && (data.s ==
  data.d1) && (data.nummsg == 1)->
    partnerB = data.s;
    pnonce = data.d2;
  :: else -> goto stopB
  fi;
  /* устанавливает публичный ключ партнера */
  if
  :: (partnerB == alice) -> pkey = pkA;
  :: (partnerB == intruder) -> pkey = pkI;
  :: else -> goto stopB
  fi;
  /* отвечает сообщением № 2 */
  d_step {
    data.s = bob;
    data.r = partnerB;
    data.nummsg = 2;
    data.key = pkey;
    data.d1 = pnonce;
    data.d2 = nonceB; }
  intercepted ! msg(data);
  /* ожидает сообщения № 3, проверяет сообщение на корректность, и в случае успеха завершается в состоянии ok */
  fakedB ? msg(data);
end_errB2:
  if
  :: (data.r == bob) && (data.s == partnerB) && (data.key
  == pkB) && (data.d1 == nonceB) && (data.nummsg == 3)->
```

```
      statusB = ok;;
  :: else -> goto stopB
  fi;
stopB:
  printf(«MSC: Process B finished \n»); }
/* злоумышленник не следует детерминированному алгоритму*/
active proctype Intruder() {
  mesCrypt data, fake;
  knowNA = false;
  knowNB = false;
end:  do
  :: intercepted ? msg(data) ->
      if
      :: (data.key == pkI) ->
        if
        :: (data.d1 == nonceA || data.d2 == nonceA) ->
          knowNA = true;
        :: (data.d1 == nonceB || data.d2 == nonceB) ->
          knowNB = true;
        fi;
      :: else -> skip;
      fi;
      /* построение испорченного сообщения */
      if
      :: fake.s = alice ->
        fake.r = bob; fake.key = pkB;
      :: fake.s = bob ->
        fake.r = alice; fake.key = pkA;
      :: fake.s = intruder;
        if
        :: fake.r = bob; fake.key = pkB;
        :: fake.r = alice; fake.key = pkA;
        fi
      fi;
```

```
/* злоумышленник не знает правила построения сообщения */
if
:: fake.d1 = alice;
:: fake.d1 = intruder;
:: fake.d1 = bob;
:: fake.d1 = nonceI;
:: (knowNA) -> fake.d1 = nonceA;
:: (knowNB) -> fake.d1 = nonceB;
fi;
if
:: fake.d2 = alice;
:: fake.d2 = intruder;
:: fake.d2 = bob;
:: fake.d2 = nonceI;
:: (knowNA) -> fake.d2 = nonceA;
:: (knowNB) -> fake.d2 = nonceB;
fi;
/* вставляет в сообщение неизвестную зашифрованную
часть из полученного сообщения */
if
:: (data.key != pkI) ->
  fake.key = data.key;
  fake.d2 = data.d2;
  fake.key = data.key;
:: else -> skip;
fi;
/* выбираем номер сообщения */
if
:: fake.nummsg = 1;
:: fake.nummsg = 2;
:: fake.nummsg = 3;
fi;
if
:: (data.r == alice) ->
```

```
      fakedA ! msg(data);  /* повторяем полученное сообщение*/
    :: (data.r == bob) ->
      fakedB ! msg(data);  /* повторяем полученное сообщение*/
    :: (data.r == alice) ->
      fakedA ! msg(fake);  /* отправляем испорченное сообщение */
    :: (fake.r == bob)
      fakedB ! msg(fake);  /* отправляем испорченное сообщение */
    fi
  od }
```

## 6.4. ЗАДАЧА ОБ ОБЕДАЮЩИХ ФИЛОСОФАХ

### 6.4.1. Описание алгоритма

Знаменитая задача об обедающих философах, предложенная Эдсгером Дейкстрой, является образным описанием, метафорой, позволяющей ясно представить проблемы, возникающие при совместном использовании ресурсов несколькими параллельными процессами.

За круглым столом сидят несколько философов, каждый из которых проводит время в размышлениях, изредка прерываясь на еду, когда он проголодается. На столе стоит блюдо со спагетти, и лежат вилки, по одной между каждыми двумя философами. Для того чтобы поесть, каждый философ должен использовать две вилки, которые лежат непосредственно слева и справа от него. Проголодавшись, он по очереди берет вилки, если они не заняты, ест спагетти, после чего возвращает вилки на место и продолжает думать до тех пор, пока снова не проголодается, затем опять ест и т.д. Если правая либо левая вилка занята соседом, философ ждет ее освобождения.

Сформулированная таким образом ситуация кажется совершенно безобидной, поведение каждого философа очевидно, и трудно предположить наличие каких-либо проблемы.

В то же время формализация задачи в виде параллельных процессов, когда философы представляются процессами, а вилки представляются общими ресурсами, демонстрирует возможность возникновения блокировки процессов, конкурирующих за разделяемые ресурсы. Такая блокировка возникает очень редко, но она приводит к остановке всех процессов. На языке метафоры это означает, что может сложиться условие, при котором все философы, упрямо следующие своим правилам поведения (фактически просто выполняющие свой

алгоритм поведения), умрут голодной смертью даже при наличии полного блюда спагетти.

Построим формальное описание ситуации. Опишем возможные состояния философа:

**S0.** Философ размышляет. В этом состоянии он может проголодаться, что представляется спонтанным переходом в состояние S1.

**S1.** В этом состоянии философ хочет есть, поэтому ему требуются вилки. Философ тянется за вилкой, лежащей слева от него. Если эта вилка отсутствует, то он в состоянии S1 ждет ее освобождения. Если левая вилка свободна, то он ее берет и переходит в состояние S2.

**S2.** В этом состоянии философ ожидает правую вилку. Если правая вилка отсутствует, то он будет находиться в этом состоянии до ее освобождения. Если правая вилка свободна, то он ее берет и переходит в состояние S3: имея обе вилки, он в этом состоянии ест спагетти.

**S3.** Философ ест. После того как он поел, он кладет на стол левую вилку, переходя в состояние S4.

**S4.** В этом состоянии философ кладет на стол правую вилку и возвращается к размышлениям (в состояние S0).

Возможные состояния каждой вилки:

**F0.** Вилка свободна. Она лежит на столе и готова к тому, что ее возьмет один из философов (слева или справа). Если ее берет левый философ, она переходит в состояние F1, если ее берет правый философ, она переходит в состояние F2.

**F1.** Вилка находится у философа слева и ожидает возвращения на стол (состояние F0).

**F2.** Вилка находится у философа справа и ожидает возвращения на стол (состояние F0).

Возможна ли ситуация общего голодания, т. е. когда все философы не смогут поесть из-за отсутствия у них необходимых вилок? Мы обнаружим ситуацию общего голодания при помощи пакета верификации параллельных процессов Spin.

### 6.4.2. Описание модели на языке Promela

Решаем задачу для `N` философов. Введем в модель `2N` процессов: `N` процессов потребуется для описания поведения философов и `N` процессов — для описания поведения вилок.

Состояние философа фиксируем в переменной cur_state. Взятию вилки философом и ее возвращению (освобождению) поставим в соответствие передачу сообщения по рандеву-каналу от философа к вилке. Всего вводится 2N каналов:

```
chan l_fork[N] = [0] of {bit}; /* для левой к философу
                                  вилки */
chan r_fork[N] = [0] of {bit}; /* для правой к философу
                                  вилки */
```

Каждому философу необходимы два канала: один — для организации обмена с левой вилкой, другой — для организации обмена с правой вилкой. Использование рандеву-каналов гарантирует синхронизацию взаимодействия философа и вилки: философ получит вилку только тогда, когда вилка будет свободна; лежащая на столе вилка поступит в распоряжение философа только тогда, когда философ ее запросит.

Для передачи запроса о доступе к вилке и ее освобождении будет достаточно сообщений одного типа:

```
#define msgtype 1  /* тип сообщения */
```

которые фактически будут играть роль сигнала для вилки и философа для изменения состояния:

```
proctype phil (chan left, right; byte mn) {
  byte cur_state = 0;
  printf(«MSC: phil # %d \n», mn);
  do
  :: (cur_state == 1) ->
    left ! msgtype ->     /* философ запросил левую вилку */
    cur_state = 2;
  :: (cur_state == 2) ->
    right ! msgtype;      /* философ запросил правую вилку */
    cur_state = 3;        /* философ ест */
  :: (cur_state == 3) ->  /* философ наелся */
    right ! msgtype;      /* философ вернул правую вилку */
    cur_state = 4;
```

```
  :: (cur_state == 4) ->
    right ! msgtype;  /* философ вернул левую вилку */
    cur_state = 0;    /* философ размышляет */
  ::(cur_state == 0) ->
    cur_state = 1;    /* философ решил поесть */
  od }
```

Для хранения состояния процесса-вилки используется переменная cur_state_fork. Взаимодействие с философами осуществляется по тем же рандеву-каналам — по одному каналу на каждого философа (один канал для левого философа, один — для правого). Когда вилка лежит на столе (она свободна), то ожидается запрос либо от левого, либо от правого философа на ее использование. Когда один из философов взял вилку, то по соответствующему каналу ожидается сигнал о возвращении вилки:

```
proctype fork(chan left_phil, right_phil)
{ byte cur_state_fork = 0;     /* вилка свободна */
end: do
  ::( cur_state_fork == 0) ->  /* если вилка свободна */
    if
    ::right_phil ? msgtype ->  /* правый философ запросил вилку */
      cur_state_fork = 2;      /* вилка у правого философа */
    ::left_phil ? msgtype ->   /* левый философ запросил вилку */
      cur_state_fork = 1;      /* вилка у левого философа */
    fi
  ::( cur_state_fork == 1) ->  /* если вилка занята */
    left_phil ? msgtype ->     /* левый философ возвращает вилку */
    cur_state_fork = 0;        /* вилка свободна */
  ::( cur_state_fork == 2) ->  /* если вилка занята */
    right_phil ? msgtype ->    /* правый философ возвращает вилку */
    cur_state_fork = 0         /* вилка свободна */
  od }
```

Запуск процессов-вилок и процессов-философов происходит в процессе init. Рассмотрим нумерацию процессов и каналов. Даны N процессов-философов с номерами от 1 до N (аналогично для вилок), N каналов для левых (аналогично для правых) по отношению к философам вилок с номерами от 0 до N-1.

- Припишем процессу-философу номер proc.
- l_fork — канал для взаимодействия философа № proc с левой вилкой получит № proc-1.
- r_fork — каналу для взаимодействия философа № proc с правой вилкой назначим № proc.
- Процессу, соответствующему правой вилке этого философа, назначим № proc.

Согласно предложенной нумерации, получим номера и названия каналов для взаимодействия вилки с левым и правым философом: r_fork с № proc для левого по отношению к вилке философа и l_fork с № proc для правого по отношению к вилке философа. Запуск начинается с процессов с номером 1:

```
init {
  byte proc;
  proc = 1;
  get_in_stuck = 0;
  do
  :: proc <= N ->
  /* запуск процесса-философа №proc с левым каналом
  №(proc-1) и правым каналом №proc */
    run phil (l_fork[proc-1], r_fork[proc%N], proc);
  /* запуск процесса-вилки №proc с каналом для правого
  философа №proc и каналом для левого философа №proc */
    run fork (r_fork[proc%N], l_fork[proc%N], proc);
    proc++;
  :: proc > N -> break
  od }
```

### 6.4.3. Обнаружение состояния общего голодания

Ситуация общего голодания, которую необходимо обнаружить, проявляется во взаимной блокировке процессов — ни один из

процессов-философов не может продолжить работу, ожидая поступления вилки. Свойство взаимной или системной блокировки (system deadlock) относится к базовым верифицируемым Spin и проверяется автоматически при поиске некорректных конечных состояний. Для этого необходимо убедиться, что в окне *Basic Verification Options* установлен флаг *Invalid Endstates*. В выводе результатов верификации появится сообщение, свидетельствующее об обнаружении пакетом системной блокировки:

```
Full statespace search for:
  never claim          - (not selected)
  assertion violations - (disabled by -A flag)
  cycle checks        - (disabled by -DSAFETY)
  invalid end states +
```

Контрпример, обнаруженный верификатором, демонстрирует, как один за другим философы тянутся за левыми вилками до тех пор, пока не приходят к ситуации, когда каждый философ ждет правую вилку (рис. 6.12).

## 6.5. МИР БЛОКОВ

Мир блоков — это искусственная среда планирования действий, которая вследствие своей ясности и простоты была одним из наиболее часто приводимых примеров в литературе по планированию в области искусственного интеллекта в 1960-х гг.

Среда состоит из кубиков (блоков), стоящих на столе по отдельности или в нескольких вертикальных башнях. Взаимное расположение кубиков можно считать состоянием среды. Целью является построение заданной конфигурации — одной или нескольких башен. За один шаг может быть перемещен только один кубик, на котором ничего не стояло: он может быть либо поставлен на стол, либо помещен сверху на какую-нибудь башню.

### 6.5.1. Описание модели на языке Promela

Построим модель на языке Promela, описывающую все возможные конфигурации и допустимые их изменения для любого заданного числа блоков и любого их расположения. Число блоков обозначим `N`. Блок идентифицируется уникальным номером от 0 до `N-1`. Конфигурацию будем задавать двумя массивами длины `N`: `up`,

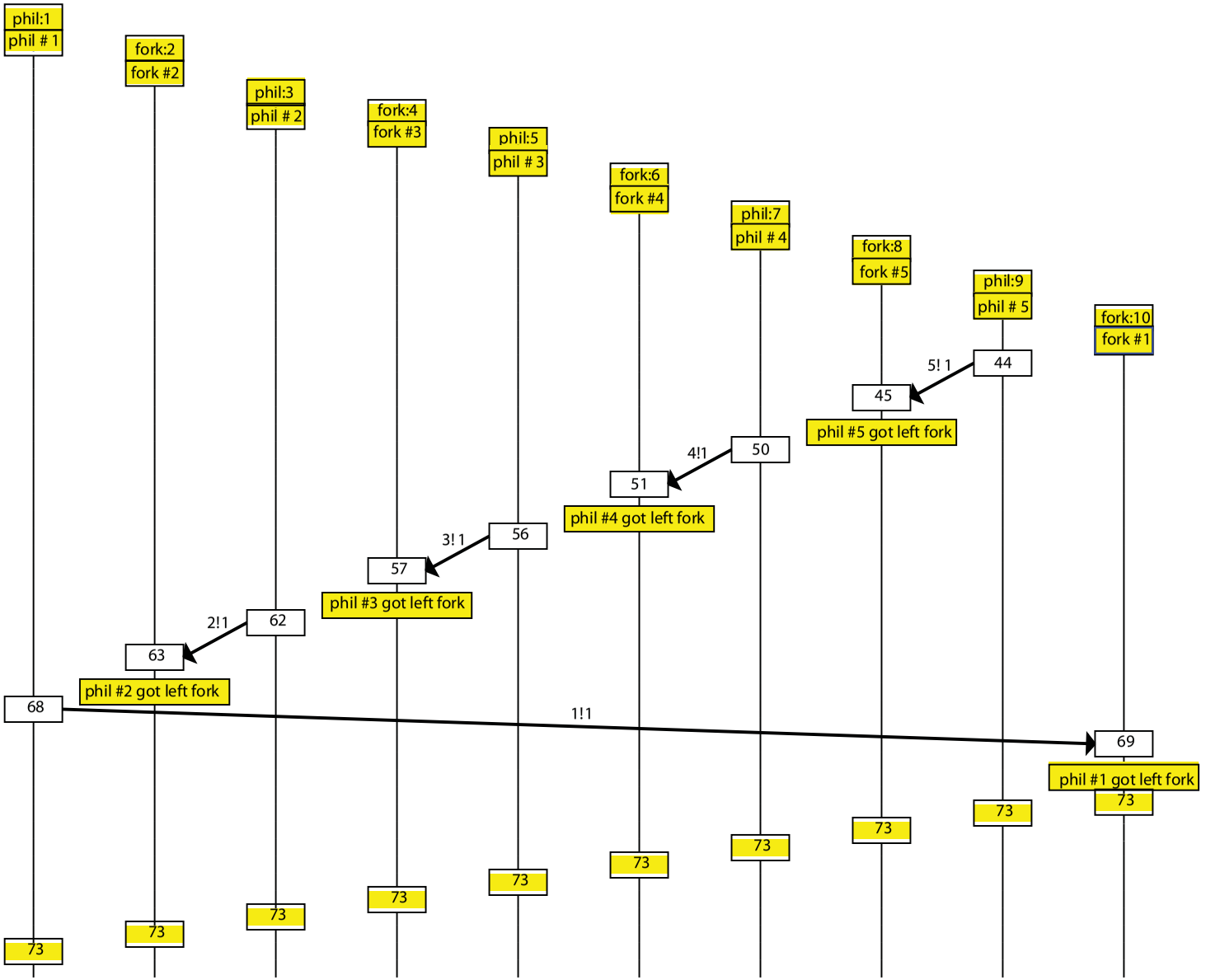


Рис. 6.12. Последовательность шагов алгоритма об обедающих философах, приводящая к ситуация общего голодания

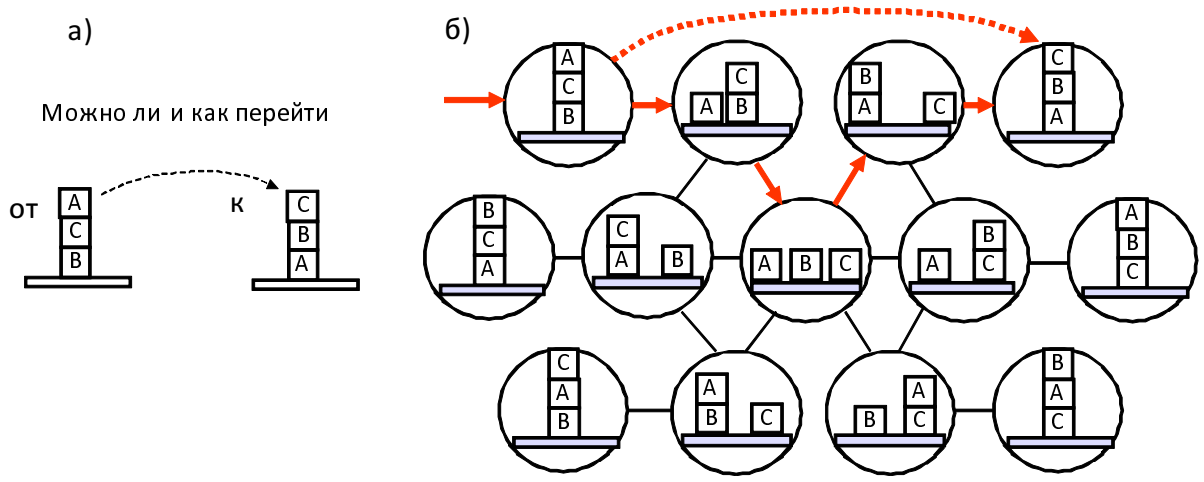


Рис. 6.13. Мир блоков: а) задача, б) пространство состояний

где `up[i]` определяет номер блока, стоящего над i-м кубиком; `down`, где `down[i]` — номер блока, стоящего под i-м кубиком. Специальная константа `NOTHING` определяет, что над или под блоком ничего не стоит. Целевая конфигурация определяется массивами `Tup` и `Tdown`, подобными массивам `up` и `down`.

Исходная конфигурация на рис. 6.13 описывается: `up[0]=NOTHING`, `up[1]=2`, `up[2]=0`, `down[0]=2`, `down[1]=NOTHING`, `down[2]=2`. Возможные переходы между состояниями программируются так:

1. выбирается случайный блок `i`, стоящий сверху (над ним ничего нет, если `up[i]==NOTHING`), и этот блок снимается (т. е. `up[down[i]] = NOTHING`);
2. выбранный блок либо кладется на стол (`down[i]=NOTHING`), либо выбирается другой блок `j`, который стоит наверху (`up[j]==NOTHING`), и на него помещается блок `i` (`up[j]=i; down[i] = j`); блок `j` отличен от `i` (`j` $\neq$ `i`).

Проверяемое условие задается как совпадение текущей и целевой конфигурации, т. е. совпадением массивов `up` и `Tup`, а также `down` и `Tdown`. Программа в цикле сначала проверяет, совпали ли текущая и целевая конфигурации; если нет — то случайно делает один из возможных переходов — изменяет текущее состояние (конфигурацию) на соседнее:

```
#define N 3  /* число кубиков */
#define NOTHING 255
```

```
#define N-1 2
/* массивы текущего состояния */
byte up[N];                          /* номер над i-м кубиком */
byte down[N];                        /* номер под i-м кубиком
                                     */
/* массивы целевого состояния */
byte Tup[N];                         /* номер над i-м кубиком */
byte Tdown[N];                       /* номер под i-м кубиком
                                     */
byte i, j, k;                        /* номера кубиков */
bool equal = false,
  findi = true,                      /* определяем снимаемый
                                     кубик i */
  findj = false,                     /* определяем кубик j, на
                                     который ставим i */
  checkconf = false;                 /* проверяем полученную
                                     конфигурацию */
active proctype cube_world(){
  /* начальное состояние*/
  up[0] = NOTHING; up[1] = 2; up[2] = 0;
  down[0] = 2; down[1] = NOTHING; down[2] = 1;
  /* целевое состояние */
  Tup[0] = 1; Tup[1] = 2; Tup[2] = NOTHING;
  Tdown[0] = NOTHING; Tdown[1] = 0; Tdown[2] = 1;
  i = 0; j = 0;
  do                                 /* основной цикл */
  :: findi ->                        /* случайный выбор i */
    if
    :: (up[i] == NOTHING) -> findj = true; findi = false;
      if
      :: ( down[i] != NOTHING ) ->
        up[ down[i] ] = NOTHING;  /* i-й сняли */
      :: else -> skip;
      fi
```

```
  :: (i < N-1) -> i ++
  :: (i > 0) -> i -
  fi
:: findj && ( down[i] != NOTHING) ->
  down[i] = NOTHING;          /* i-й положили на стол */
  printf(«MSC: [%d] to table \n», i);
  findj = false; checkconf = true;
/* случайно выберем j-й и i-й поставим на j-й */
:: findj ->
  if
  :: ( i != j ) && ( up[j] == NOTHING ) ->
    up[j] = i; down[i] = j;  /* i-й кубик ставим на j-й
                              */
    printf(«MSC: [%d] to [%d] \n», i, j);
    findj = false; checkconf = true
  :: (j < N-1) -> j ++
  :: (j > 0) -> j -
  fi
/* проверка, достигли ли целевого состояния */
:: checkconf ->
  equal = true; k = 0;
  do
  :: ( k < N ) ->
    equal=equal && (up[k]==Tup[k]) && (down[k]==Tdown[k]);
    k++
  :: ( !equal ) -> break
  :: ( k == N ) -> break
  od;
  if
  :: (equal) -> break
  :: else -> checkconf = false; findi = true
  fi
od;
printf(«MSC: movement finished \n») }
```

### 6.5.2. Поиск решения задачи средствами Spin

Задав начальное и целевое состояния, попробуем найти решение (последовательность допустимых переходов) средствами верификации. Для этого поставим задачу перед Spin: не существует ни одного пути, такого, что выполняется LTL формула: `FG (equal == true)`, т. е. не существует ни одного пути, на котором возможно прийти к целевому состоянию из данного начального. Spin находит контрпример, предлагая решение данной задачи (рис. 6.14).

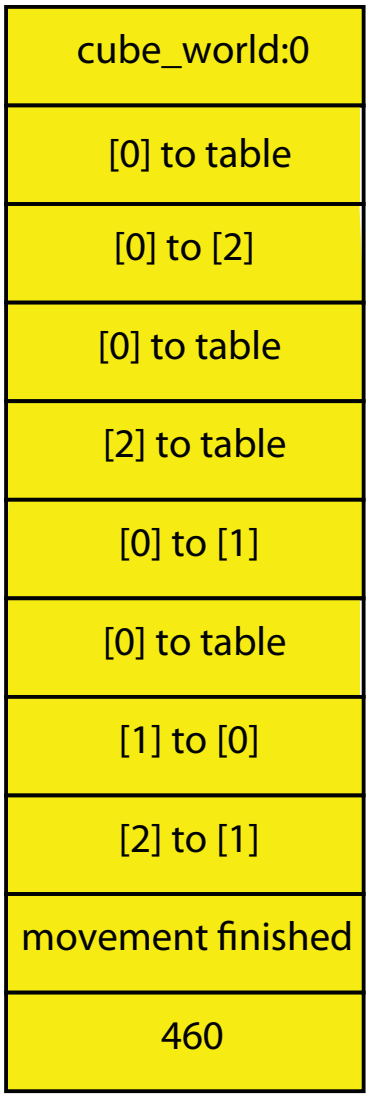


Рис. 6.14. Решение задачи мира кубиков, найденное Spin

Поскольку модель допускает несколько решений, то Spin находит одну из возможных последовательностей переходов — совсем неоптимальную. Изобразим для наглядности решение, предложенное верификатором, как последовательность перемещения кубиков (рис. 6.15).

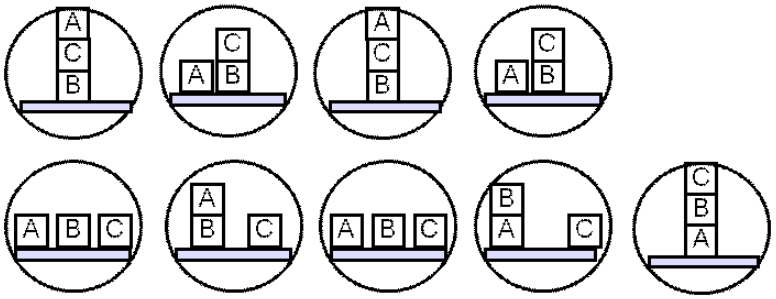


Рис. 6.15. Перестановка в мире кубиков, приводящая к решению

# 7. КУРСОВАЯ РАБОТА «РАЗРАБОТКА КОНТРОЛЛЕРА СВЕТОФОРОВ И ЕГО ВЕРИФИКАЦИЯ»

Цель курсовой работы — сформировать у студента следующие знания и умения:

- владение методами обеспечения качества программных и аппаратных систем, понимание общей схемы, этапов выполнения тестирования и верификации;
- понимание теоретических основ и методов верификации параллельных и распределенных систем на основе темпоральной логики и алгоритмов проверки моделей;
- моделирование программной системы для верификации одним из подходящих методов;
- составление желаемых свойств программной или аппаратной системы на языке темпоральной логики;
- использование для выполнения верификации одного из доступных инструментальных пакетов (на примере Spin).

Данные знания и умения развивают компетенции студента:

- способность совершенствовать и развивать свой интеллектуальный и общекультурный уровень (ОК-1);
- использование на практике умения и навыки в организации исследовательских и проектных работ, в управлении коллективом (ОК-4);
- способность самостоятельно приобретать с помощью информационных технологий и использовать в практической деятельности новые знания и умения, в том числе в новых областях знаний, непосредственно не связанных со сферой деятельности (ОК-6);
- способность к профессиональной эксплуатации современного оборудования и приборов;
- навык применять перспективные методы исследования и решения профессиональных задач на основе знания мировых тенденций развития вычислительной техники и информационных технологий (ПК-1);
- навык формировать технические задания и участвовать в разработке аппаратных и/или программных средств вычислительной техники (ПК-4).

Курсовая работа основывается на теоретических методах, изложенных в разделах 1–3. Конструкции языка Promela и основы работы с пакетом верификации Spin рассмотрены в разделах 4–6.

Курсовая работа предполагает творческий характер выполнения. При решении задачи из курсовой работы студент должен самостоятельно уточнить модель системы, построить модель на входном языке средства верификации, сформулировать формулы линейной темпоральной логики, описывающие свойства системы, верифицировать систему относительно заданных свойств, объяснить и исправить ошибки, повторно верифицировать систему. Отчет по курсовой работе должен содержать описание индивидуального задания и основные этапы выполнения.

Курсовая работа оценивается по следующим критериям:

- студент понимает и может объяснить конструкции входного языка средства верификации, а также режимы работы верификатора;
- модель, представленная в курсовой, соответствует заданию и проходит верификацию заданных в курсовой свойств;
- студент может составить свойства поведения, сформулированные преподавателем, верифицировать их, объяснить полученный результат;
- дополнительно поощряется оригинальность, ясность решения.

## 7.1. СОДЕРЖАНИЕ КУРСОВОЙ РАБОТЫ

В курсовой работе ставится задача разработки *контроллера светофоров* для управления проездом через перекресток. Каждое направление движения транспорта на перекрестке регулируется своим *светофором*. *Датчики*, установленные на соответствующих полосах, фиксируют наличие автомобилей в каждом направлении. В отсутствие автомобилей светофор соответствующего направления горит красным. При появлении автомобилей на полосе процесс управления светофором этого направления начинает борьбу за ресурсы. *Ресурсами* в задаче являются области перекрестка, в которых возможны пересечения разных направлений движения. Если процесс получает нужные ресурсы, то он переключает свой светофор на зеленый, разрешая движение транспорта в этом направлении. Ядром работы является построение корректных алгоритмов захвата и освобождения ресурсов в процессах управления светофорами.

*Контроллер светофоров* состоит из *контроллеров направ-*

*лений*, моделируемых параллельными процессами (по одному для каждого направления), и глобальных разделяемых переменных, которые могут модифицироваться (устанавливаться) и читаться этими процессами. Каждый контроллер направления управляет светофором, разрешающим или запрещающим движение по перекрестку в этом направлении. В число локальных переменных процесса входит переменная, отвечающая за разрешающий (зеленый) или запрещающий (красный) сигнал связанного с процессом светофора. Глобальные переменные используются для реализации алгоритма синхронизации процессов, гарантирующего корректное использование процессами общих разделяемых ресурсов.

Для выполнения курсовой работы необходимо, отталкиваясь от простейшего алгоритма (раздел 7.2), построить модель работы контроллера светофоров на языке Promela для индивидуального плана проезда на рис. 7.2 и верифицировать полученную модель средствами Spin относительно свойств (раздел 7.3). В этих процессах могут быть ошибки синхронизации. Все ошибки студент должен выявить и проанализировать с помощью системы верификации Spin на основе выдаваемых системой контрпримеров, демонстрирующих те сценарии движения на перекрестке, в которых процессы управляют своими светофорами неверно. Впоследствие студент должен модифицировать алгоритм работы контроллера светофоров так, чтобы удовлетворить свойствам в разделе 7.3.

## 7.2. ПРИМЕР РЕШЕНИЯ ТИПОВОЙ ЗАДАЧИ ДЛЯ ПРОСТОЙ СХЕМЫ ПЕРЕКРЕСТКА

Построим контроллер, управляющий светофорами перекрестка (рис. 7.1) в направлениях N (на север), S (на юг) и E (на восток). Внешние события `N_SENSE`, `S_SENSE` и `E_SENSE` указывают, что сенсор данного направления обнаружил автомобиль, желающий получить право проезда в соответствующем направлении (на север, юг или восток). Движение разрешается при зеленом сигнале светофора в соответствующем направлении. Для простоты в этом примере будем считать, что:

- машины пересекают перекресток, не поворачивая;
- цикл переключения светофоров: красный — зеленый — красный — зеленый ...

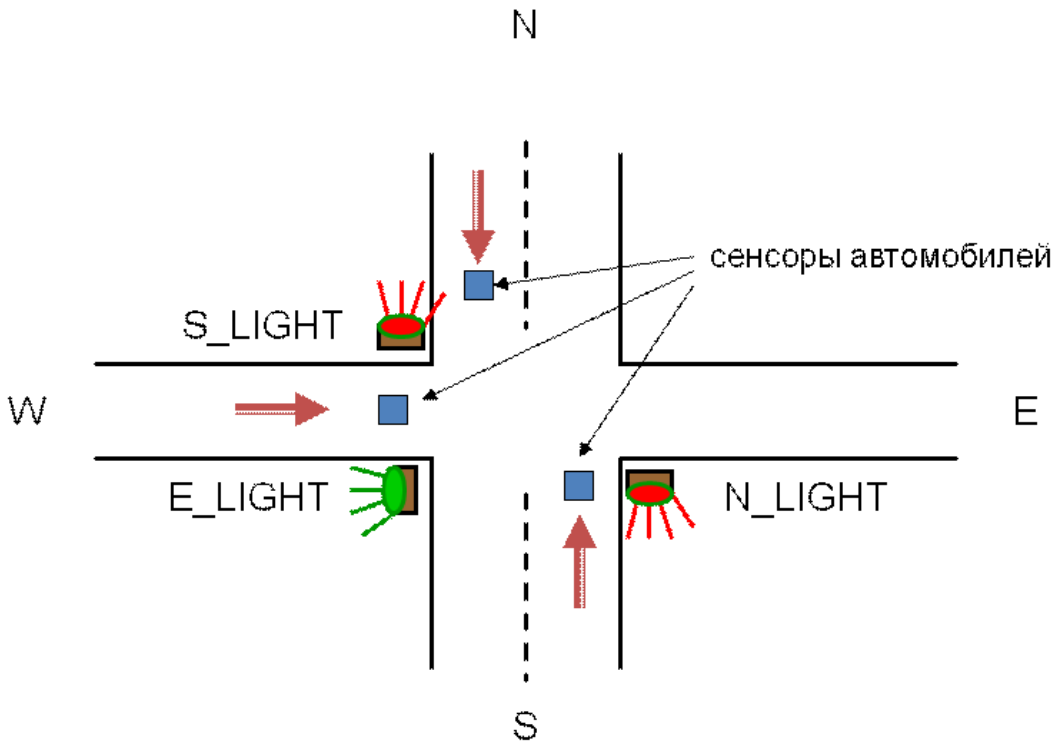


Рис. 7.1. Схема проезда простого перекрестка

Контроллер будем строить из трех параллельно функционирующих процессов: `N`, `S` и `E`. Каждый процесс управляет своим светофором, разрешающим движение в своем направлении. Выход контроллера — установка конкретного значения цвета светофоров: локальные переменные процессов `N_LIGHT`, `S_LIGHT` и `E_LIGHT` принимают значения `RED` и `GREEN`. Процесс `N` читает и может сбросить значение своей булевой переменной `N_REQ`, которая указывает, что поступил запрос на проезд в северном направлении. Эта переменная устанавливается событием `N_SENSE` в процессе, моделирующем внешнюю среду, который работает параллельно с процессом `N`:

```
while true do if (!N_REQ && N_SENSE)
  then N_REQ := true fi od
```

То же справедливо для процессов `S` и `E`.

Для синхронизации процессов следует ввести переменные, с которыми могут работать все процессы. Введем глобальные булевы переменные (флаги направлений движения), которые назовем `NS_LOCK` и `WE_ LOCK`. Процессы `N`, `S` и `E` будут взаимодействовать через эти общие булевы переменные.

Все представленные процессы записаны на языке, подобном Паскалю. Алгоритм процесса N, регулирующий движение на север, может быть представлен так:

```
process N (){
  while true do          /* в бесконечном цикле */
    if(N_REQ)            /* если есть запрос на проезд на
                            север */
      wait(!WE_LOCK);    /* то ждем освобождения поперечного
                            направления */
      NS_LOCK:=true;     /* устанавливаем замок на направле-
                            ние С-Ю */
      N_LIGHT:=GREEN;    /* и разрешаем проезд в северном
                            направлении; */
      wait(!N_SENSE);    /* ждем, когда на север проедут все
                            машины */
      if(S_LIGHT=RED)    /* если светофор на встречном направ-
                            лении красный, снимаем замок */
        NS_LOCK:=false fi;
      N_LIGHT:=RED;      /* устанавливаем на своем светофоре
                            красный */
      N_REQ:=false;      /* сбрасываем запрос на проезд на
                            север */
    fi
  od }
```

Процесс S строится аналогично. Алгоритм процесса E, регулирующего движение на восток:

```
process E (){
  while true do          /* в бесконечном цикле */
    if(E_REQ)            /* если есть запрос на проезд на восток
                            */
      WE_LOCK:=true;     /* то устанавливаем свой замок */
      E_LIGHT:=GREEN;    /* и разрешаем проезд на восток */
      wait(!NS_LOCK);    /* ждем, когда процессы С и Ю снимут
                            замок */
```

```
      wait(!E_SENSE);   /* ждем, когда на восток все проедут */
      WE_LOCK:=false;   /* снимаем свой замок */
      E_LIGHT:=RED;     /* устанавливаем на своем светофоре
                           красный */
      E_REQ:=false;     /* сбрасываем запрос на проезд на
                           восток */
    fi
  od }
```

Алгоритмы поведения внешней среды не описаны. Их требуется определить самостоятельно.

## 7.3. СПЕЦИФИКАЦИЯ СВОЙСТВ КОНТРОЛЛЕРА

Алгоритмы этих процессов выглядят разумно. Однако корректность их не доказана. Как и в любых параллельных системах, в этой системе параллельных процессов ошибки очень вероятны. Если при проверке корректности этой или подобной системы процессов выявятся ошибки, их нужно исправить.

Корректность построенного контроллера светофоров следует проверить относительно следующих свойств:

*Безопасность*:

```
G !(( N_LIGHT=GREEN || S_LIGHT=GREEN) and (E_LIGHT = GREEN))
```

— никогда не будет разрешен проезд в пересекающихся направлениях.

*Живость*:

```
G ( N_SENSE and ( N_LIGHT=RED ) ==> F ( N_LIGHT=GREEN ) );
G ( S_SENSE and ( S_LIGHT=RED ) ==> F ( S_LIGHT=GREEN ) );
G ( E_SENSE and ( E_LIGHT=RED ) ==> F ( E_LIGHT=GREEN ) )
```

— при появлении машины ей всегда предоставится возможность проезда в нужном направлении (возможно, не сразу).

Эти свойства должны быть выполнены при ограничениях *справедливости*:

```
GF !( (N_LIGHT=GREEN) & N_SENSE );
GF !( (S_LIGHT=GREEN) & S_SENSE );
GF !( (E_LIGHT=GREEN) & E_SENSE );
```

— предполагаем, что каждый сенсор устанавливается в `false` неопределенно часто, т. е. в каждом направлении не движется непрерывный поток машин.

## 7.4. ИНДИВИДУАЛЬНОЕ ЗАДАНИЕ ДЛЯ КУРСОВОЙ РАБОТЫ

Для выполнения курсовой работы выбран более сложный перекресток с четырьмя двусторонними направлениями движения (рис. 7.2). В каждом направлении имеются три полосы движения для трех траекторий движения: для проезда прямо по перекрестку, для поворота направо, для поворота налево. Движение по каждой траектории регулируется своим светофором: один светофор — для проезда прямо через перекресток, один (стрелка) — для поворота направо, один (стрелка) — для поворота налево.

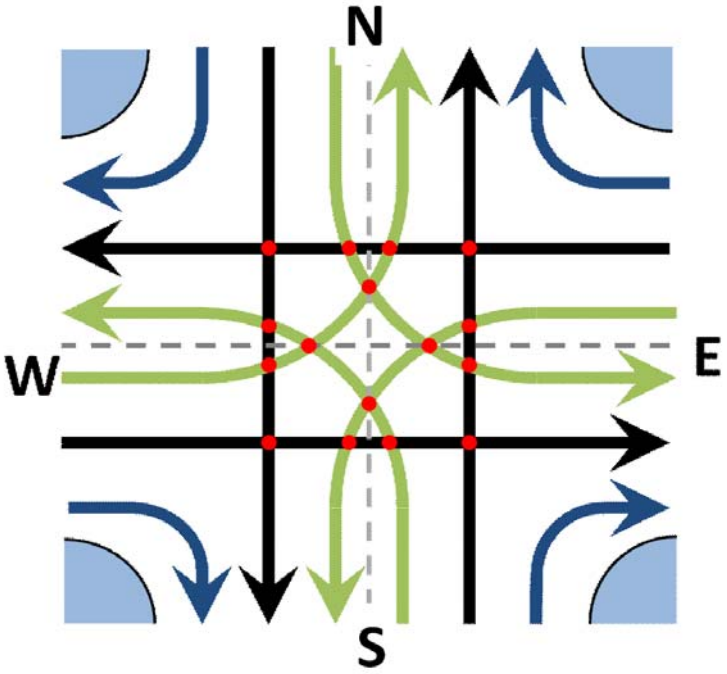


Рис. 7.2. Схема сложного перекрестка

Поворот направо разрешен всегда. Однако светофор, регулирующий движение по правой полосе, горит красным, если машин на этой полосе нет. При появлении машин он загорается зеленым и горит до тех пор, пока поток машин не прекратится, тогда светофор загорается красным.

Светофор, регулирующий движение по любой траектории, загорается зеленым, если, во-первых, есть запрос от машин в соответ-

ствующей полосе и, во-вторых, по всем тем направлениям, с которыми существует пересечение, движение запрещено — на них горит красный сигнал.

Таблица 7.1

**Варианты пересечений направлений для индивидуальных заданий**

| Вариант | Пересечение | Вариант | Пересечение |
|---|---|---|---|
| 1 | WN,NS | 9 | SN,WE |
| 2 | WN,NE | 10 | SN,EW |
| 3 | WN,SW | 11 | SN,ES |
| 4 | WN,EW | 12 | NE,EW |
| 5 | NS,SW | 13 | NE,ES |
| 6 | NS,WE | 14 | SW,WE |
| 7 | NS,EW | 15 | SW,ES |
| 8 | SN,NE | 16 | WE,ES |

Индивидуальное задание строится по рис. 7.2 и табл. 7.1. Преподаватель выбирает два или три уникальных пересечения из табл. 7.1. Другие направления движения по перекрестку на рис. 7.2 игнорируются.

**Пример индивидуального задания.** Допустим, что заданы два пересечения, например, 1, 10. Отсюда следует, что по направлениям WN, NS, SN, EW проезд есть. По всем другим направлениям в конкретной конфигурации перекрестка (для данного варианта задания) полос для проезда нет.

Разнообразие моделей, допускаемых данной постановкой задачи, достаточно велико. Для того, чтобы избежать упрощения задачи, следует придерживаться рекомендаций при разработке модели контроллера на языке Promela:

1. При разработке алгоритма контроллера светофоров необходимо сохранить наличие процессов, управляющих внешней средой, регистрирующей автомобили, и процессов — контроллеров направлений, регулирующих движение по перекрестку для каждого направления.
2. При написании кода контроллера на Promela следует аккуратно пользоваться конструкций `atomic`. Использование `atomic` возможно для решения технических задач (например, для принуди-

тельного вывода значения переменной сразу после ее изменения). При решении проблем синхронизации процессов конструкция `atomic` требует полного понимания: чтобы не уничтожить явление чередования параллельных процессов и корректно смоделировать известные примитивы синхронизации.

# БИБЛИОГРАФИЧЕСКИЙ СПИСОК


1. http://www.esa.int/esaCP/SEM85Q71Y3E_index_0.html.

2. http://epizodsspace.testpilot.ru/bibl/ivanovskiy/rakety-i-kosmos/20.html.

3. *Карпов Ю. Г.* О корректности параллельных алгоритмов / Ю. Г. Карпов. — *Программирование*. — 1996. — № 4.

4. *Карпов Ю. Г.* Анализ корректности параллельной программы разделения множеств / Ю. Г. Карпов. — *Программирование*. — 1996. — № 6.

5. *Карпов Ю. Г.* Теория автоматов / Ю. Г. Карпов. — СПб.: Питер, 2002.

6. *Карпов Ю. Г.* Model Checking. Верификация параллельных и распределенных программных систем / Ю. Г. Карпов. — СПб:БХВ-Петербург, 2009. — 560 с.

7. Tech. Rep. US General Accounting Office Report 92026: 1992.

8. Анализ отказа системы "Arian-5". http://forums.airbase.ru.

9. *Armoni R. et al.* The forspec temporal logic — a new temporal property-specification language // Proc. $8^{th}$ Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems. — 2002.

10. *Baier C., Katoen J.-P.* Principles of Model Checking. — Boston: The MIT Press, 2008. — P. 560.

11. *Bormann J. et al.* Model checking in industrial hardware design // Proc. of $32^{nd}$ Design Automation Conference. — San Jose, 1995.

12. *Buchi J. R.* On a decision method in restricted second order arithmetic // Proc. Int. Congr. Logic, Methods and Philos. Sci. — Stanford, 1962.

13. *Clarke E. M., Emerson E. A.* Design and synthesis of synchronization skeletons using branching-time temporal logic // Logic of Programs Workshop. — Yorktown Heights, 1981.

14. *Clarke E. M., Emerson E. A., Sistla A. P.* Automatic verification of finite-state concurrent systems using temporal logic specifications: A practical approach // $10^{th}$ ACM Symp. on Principles of Programming Languages. — New York, 1983.

15. *Clarke E. M., Emerson E. A., Sistla A. P.* Automatic verification of finite-state concurrent systems using temporal logic specifications // *ACM Trans. Program. Lang. Syst.* — 1986. — no. 8(2).

16. *Clarke E. M., Grumberg O., Peled D.* Model checking. — Boston: The MIT Press, 1999.

17. *Dolev D., Klawe M., Rodeh M.* An o(n log n) unidirectional distributed algorithm for extrema finding in a circle // *Journal of Algorithms*. — 1982. — Vol. 3. — Pp. 245 — 260.

18. *Emerson E. M.* Temporal and modal logic // *Handbook of Theoretical Computer Science / ed. by J. van Leeuwen*. — 1990. — Vol. B.

19. *Floyd R.* Assigning meaning to programs // Proc. Symposium on Applied Mathematics. — Vol. 19. — Providence, 1967.

20. *Havelund K., Lowry M., Penix J.* Formal analysis of a space-craft controller using spin // *IEEE Trans. Software Eng.* — 2001. — Vol. 27.

21. *Hilditch S., Thomson T.* Distributed deadlock detection: algorithms and proofs: Tech. Rep. Dept. Comput. Sci., Uni. of Manchester, Technical Report UMCS-89-6-1: 1989.

22. *Hoare C. A. R.* An axiomatic basis of computer programming // *Communications of the ACM*. — 1969. — Vol. 12, no. 10.

23. *Holzmann G.* Design and validation of computer protocols. — Prentice Hall, 1991.

24. *Kripke S. A.* Semantical consideration on modal logic // *Acta Philosophica Fennica*. — 1963. — Vol. 16.

25. *Lowe G.* Breaking and fixing the needham — schroeder public-key protocol using fdr // Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems table of contents. — Vol. 1055 of *LNCS*. — 1996. — Pp. 147 — 166.

26. *Lynch W. C.* Computer systems: reliable full-duplex file transmission over half-duplex telephone line // *Commun. ACM*. — 1968. — Vol. 11, no. 6.

27. *Peterson G.* An o(n log n) unidirectional algorithm for the circular extrema problem // *ACM Transactions on Programming Languages and Systems (TOPLAS)*. — 1982. — Vol. 4. — Pp. 758 — 762.

28. *Pnueli A.* The temporal logic of program // $18^{th}$ Anny. Symp. on Foundation of Computer Science. — Providence, 1977.

29. *Prior A. N.* Past, present and future. — Oxford: Oxford University Press, 1967.

30. *Queille J.-P., Sifakis J.* Specification and verification of concurrent programs in cesar // *Lecture Notes in Comput. Sci.* — 1982. — Vol. 137.

31. *Reichenbach H.* Elements of symbolic logic. — New York : Macmillan, 1947.

32. *Vardi M.* An automata-theoretic approach to linear temporal logic // *Lecture Notes in Comput. Sci.* — 1996. — Vol. 1043.

33. *Vardi M. Y.* Branching vs. linear time: Final showdown // $7^{th}$ Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01). — Genova, 2001.

34. *Wolper P.* Constructing automata from temporal logic formulas: A tutorial // *Lecture Notes in Comput. Sci.* — 2001. — Vol. 2090.

# ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Карпов Юрий Глебович,
Шошмина Ирина Владимировна

# ВЕРИФИКАЦИЯ РАСПРЕДЕЛЕННЫХ СИСТЕМ

Учебное пособие

Лицензия ЛР № 020593 от 07.08.97

Налоговая льгота — Общероссийский классификатор продукции
ОК 005-93, т.2; 95 3005 — учебная литература

---

Подписано в печать 2011. Формат 60x84/16. Печать цифровая
Усл. печ. л. . Уч.-изд. л. . Тираж 40. Заказ .

---

Отпечатано с готового оригинал-макета, предоставленного авторами,
в Цифровом типографском центре
Издательства Политехнического университета,
195251, Санкт-Петербург, Политехническая ул., 29.
Тел.: (812) 550-40-14.
Тел./факс: (812) 297-57-76.