Министерство Общего и Профессионального Образования
Российской Федерации
Ростовский Государственный Университет
Механико-математический Факультет
кафедра Информатики и Вычислительного Эксперимента

Дипломная работа на тему:

Создание и проверка моделей распределённых систем

студент 5 курса Зайцев В. В.

Научный руководитель: к.т.н., доцент Литвиненко А. Н.

Ростов-на-Дону
2003

Оглавление

Глава 1

Введение

Данная работа находится на стыке двух больших областей и представляет собой приложение, проекцию одной на другую. Первой областью является ***проверка программного обеспечения на правильность***. Необходимость быть уверенным в тех или иных свойствах программы (например, в свободе системы от тупиков или доступности в любой момент) встала достаточно давно, с появлением первых крупных программных комплексов. Первые шаги в доказательстве правильности компьютерных программ были сделаны в семидесятых годах (в основном имеются в виду взаимодействующие последовательные процессы Хоара [42] и системы их доказательства [8, 31]). В последние годы второе дыхание этой области подарили модельные подходы [19, 43, 44, и стр. 9–16 данной работы], в которых вместо человека [56] основную работу выполняет ЭВМ [45]. Они многие годы до того применялись в верификации аппаратного обеспечения, но только недавно вычислительные мощности сделали возможным переход на программное.

Второй областью, к которой можно отнести работу, является ***разработка распределённых приложений***. С развитием Интернета те программы, которые работают только для одного пользователя только на одном компьютере, постепенно уходят в прошлое. Современные тенденции отражают другой подход: рассматривать большие комплексы разнородных программ, совместно предоставляющих некий сервис пользователю. Одна из глав работы полностью посвящена изучению того, как специфика области влияет на верификацию программ и какие именно части распределённых систем могут быть верифицированы наиболее успешно.

Картина была бы неполной, если бы в работе освещалась только теоретическая сторона вопроса. Поэтому в последней главе будут проведены практический анализ, всестороннее рассмотрение, моделирование,

верификация и (в некоторых случаях) исправление и уточнение модели с повторной верификацией. Будут рассмотрены три большие системы: бесследовый протокол [20, 22, 23, 34], система управления доступом [24, 43] и система обработки транзакций [47]. В каждой из них наряду с классической схемой работы системы автором предложены различные усложнения.

В главе 2 даётся краткий обзор существующих методов верификации программного обеспечения, излагается теория, имевшаяся уже на момент начала работы (см. список литературы). Глава 3 подробно рассматривает распределённые системы и приложение к их верификации описанных методов. Это теоретическая основа проделанной работы.

На страницах 45–87 (глава 4) проводится моделирование трёх крупных распределённых систем; их детальное рассмотрение; верификация и её анализ. Всё это составляет практическую основу данной работы и показывает, как именно должна применяться выбранная методология проверки моделей в приложении к распределённым системам.

Глава 2

Описание базовой методологии

В этом разделе будет произведён обзор средств, предназначенных для отлова и уменьшения количества ошибок в программах, описана роль моделирования в этом процессе и приведён список проблем, возникающих при проверке моделей.

2.1 Верификация

Область информационных и коммуникационных технологий (ICT) развивается в последнее время всё быстрее. И так же быстро возрастает зависимость людей от работы подобных систем, требовательность к их верному функционированию и чувствительность к различного рода ошибкам. Распределённые системы ICT[1] получают всё более широкое распространение как через Интернет, так и через всевозможные типы встроенных систем (банкоматы, смарт-карты, ноутбуки, мобильные телефоны, переносные компьютеры, карманные компьютеры, телевизоры и многое другое).

Дефекты программного и аппаратного обеспечении вызывают большие материальные потери. Например, небольшая ошибка в блоке деления с плавающей точкой в процессоре Pentium-II привела к потере фирмой Intel 475 миллионов долларов, потраченных на замену уже поставленных процессоров, и серьёзно подмочила репутацию фирмы. Дефекты программного и аппаратного обеспечении вызывают катастрофы, уносящие жизни людей. 4 июня 1996 года космический аппарат Ариан-5 взорвался через 36 секунд после старта из-за ошибки в одной из подпрограмм, производящей округление 64-битового числа с плавающей точкой

[1] Ветвь информатики, изучающую распределённые системы ICT иногда называют телематикой — сокращение от слов телекоммуникации и информатика.

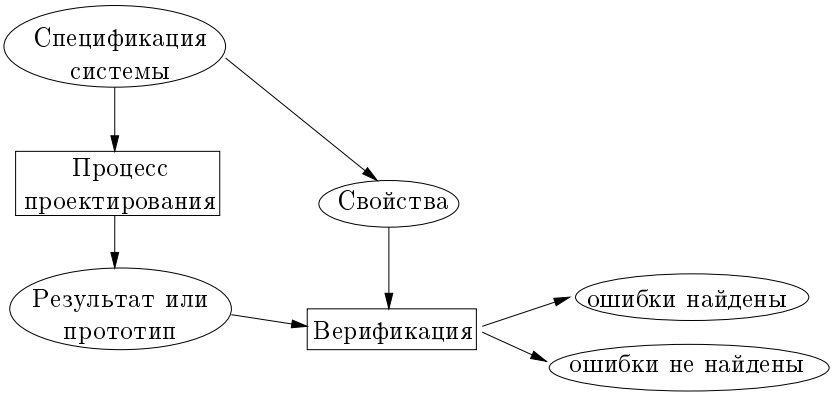


Рис. 2.1: Общая схема процесса верификации

в 16-битовое целое. Многие строки кода пишутся для управления самолётами и космическими аппаратами, химическими заводами, атомными станциями, системами светофоров и метеорологическими станциями, заблаговременно предупреждающих о природных катаклизмах.

Распределённые системы представляют собой ещё более сложную область, потому как количество дефектов в программах растёт экспоненциально с увеличением числа взаимодействующих компонент [49]. Тем не менее, согласно списку основных факторов, влияющих на уменьшение дефектов программного обеспечения [12], от 40 до 50% программ содержат нетривиальные ошибки, которые не отлавливаются ни бета-тестированием, ни простым анализом (peer review, который отлавливает, тем не менее, согласно данным того же списка, до 60% дефектов).

В последнее время вопрос решается либо формальным тестированием (на научной базе), либо верификацией. Первый подход состоит в создании определённого списка входных и выходных данных, которые программа должна производить согласно своей спецификации. Тестирование считается успешным, если ошибка найдена. В этом случае данный кортеж входных и выходных данных служит контр-примером. Это также большая и быстро растущая область науки, содержащая множество различных подходов (тестирование по принципу «чёрного ящика», построение графа связей, покрытие ветвлений, запусков, условий, операторов и пр.), на которых мы не будем подробно останавливаться (см. тж. стр. 37). Второй подход будет рассмотрен ниже.

Верификация — это процесс, удостоверяющий, что данная система удовлетворяет данным требованиям или имеет данные свойства.

Валидация — это разновидность *верификации*, при котором система проверяется на соответствие своей спецификации, то есть это ве-

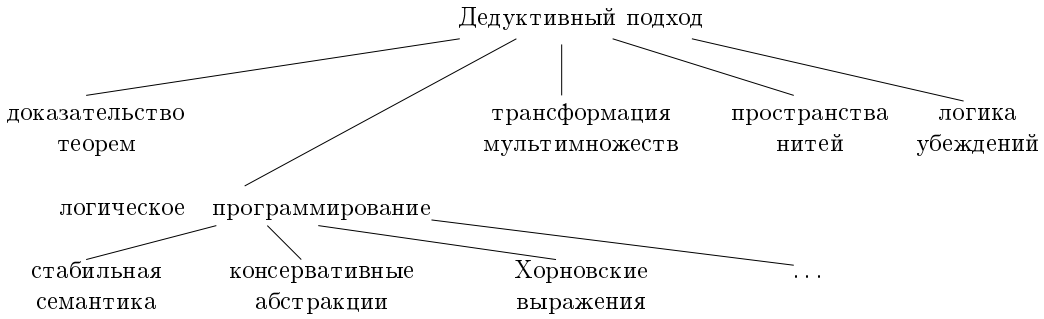


Рис. 2.2: Дедуктивные методы верификации

рификация свойств, которыми должна обладать система.

Как показано на рис. 2.1, процесс извлечения критически важных свойств из спецификации и их формулирования в рамках выбранной логики может идти параллельно с собственно процессом проектирования. Тем не менее, иногда проектировщики идут на дополнительные траты и сначала верифицируют саму спецификацию (как это сделано в [19] с использованием модельного подхода).

2.2 Существующие подходы к верификации

Широкое множество различных техник верификации используются при создании распределённых систем. Часть из них восходит корнями к верификации аппаратного обеспечения (области куда более старой и исследованной), другая — к сравнительно новым математическим формализмам. Все эти техники, во-первых, весьма многочисленны и не будут здесь ни описываться, ни даже перечисляться, а, во-вторых, трудно классифицируемы, потому что зачастую заимствуют удачно работающие приёмы друг у друга. Тем не менее, была предпринята попытка внести в один список и объяснить основные используемые этими техниками концепции, коих сравнительно немного.

Несмотря на всю нечёткость классификации, граница между *дедуктивной методологией* и *модельным подходом* проводится всегда. Методы первого типа исследуют систему в том виде, в каком она есть, тогда как методы, относящиеся ко второму типу, строят модель системы и работают только с нею.

Дедуктивные методы (рис. 2.2) используют чистую логику для проверки корректности рассматриваемой системы. Это требует серьёзных знаний в области математики и накладывает существенные ограничения на спектр задач. В зависимости от конкретного метода некоторые

задачи могут оказаться нерешаемыми в принципе, тогда как другие потребуют видоизменения системы. Тем не менее, дедуктивная методология — это единственный способ **доказательства** корректности системы, другие методы не могут гарантировать на 100% того, что система действительно удовлетворяет определённым требованиям.

Вероятно, самым древним представителем такого рода методов является *автоматическое доказательство теорем*. В двух словах, доказательство теорем — это логически непротиворечивое выведение определения теоремы из заданного множества аксиом с помощью правил математической логики. Для автоматизации этого процесс были написаны специализированные пакеты (Isabelle/HOL [52], Maude [30], daTac [48] и другие), каждый из которых применяет свой набор приёмов для достижения желаемой цели. Это НЕ означает того, что можно подать на вход такого рода программе формулировку любой теоремы (например, теоремы Ферма), нажать на кнопку, подождать немного и получить готовое доказательство — на данном этапе развития науки подобное неосуществимо. Нужна не только программа, но и путь, которым она должна следовать для гарантированного достижения цели. Ещё далеко не все стратегии мышления, используемые человеком, нашли свою формализацию [49]. Как правило, каждая система автоматического доказательства теорем имеет свою (путь даже узкую) область применимости, в пределах которой работает более-менее успешно. Многие концепции этой области логики перешли в другие дедуктивные методы и даже в методы, использующие модели (например, символьная проверка моделей в [13]).

Другим широко используемым методом является *переписывание мультимножеств*. Мультимножеством (или комплектом) M называется неупорядоченная коллекция объектов или элементов, возможно, с повторениями (этим оно и отличается от обычного множества). Пустое мультимножество не содержит ни одного элемента. Объединение мультимножеств обозначается M, N и содержит все элементы мультимножества M, равно как и все элементы мультимножества N. В качестве элементов могут быть рассмотрены, например, предикаты первого порядка над множеством переменных. Правило r переписывания мультимножества представимо парой мультимножеств F и G (называемых соответственно антецедентом и консеквентом):

$$r : \; F(\vec{x}) \mapsto \exists \vec{n}\, G(\vec{x}, \vec{n}) \tag{2.1}$$

где r — собственно правило (точнее, метка правила, его идентифицирующая), $\vec{x}$ — множество переменных исходного мультимножества, а $\vec{n}$ — новые (*свежие*) переменные, внесённые правилом. Система пере-

писывания мультимножеств $\mathcal{R}$ — это множество (обычное) такого рода правил. Правило (2.1) называется применимым к мультимножеству фактов M тогда и только тогда, когда $M = F\left(\vec{t}\right), M'$ (и применение его даёт мультимножество $N = G(\vec{t}, \vec{c})$, где $\vec{c}$ — новые переменные или константы, не встречавшиеся в M). Таким образом, правила переписывания позволяют делать из одного мультимножества другое с помощью локальных изменений. Вектор использованных правил отвечает за собственно последовательность вычислений [17].

К вопросу об использовании приёмов родственных методов, пакет автоматического доказательства теорем daTac [48], написанный М.Русиновичем и другими, использует формализм переписывания мультимножеств.

В другом подходе, называемом *пространствами нитей* (strand spaces), событие представляет собой кортеж из собственно сообщения и направления его движения (посылаемое: $+m$, получаемое: $-m$). Множество всех событий логично обозначается $\pm\mathcal{M}$. Нить (strand) — это конечная последовательность событий (элемент рефлексивно-транзитивного замыкания[2] множества всех событий: $s \in (\pm\mathcal{M})^*$). Нить может быть представлена как цепной[3] помеченный граф с метками из $\pm\mathcal{M}$, соответствующими элементам s в порядке их прохождения по дугам (то есть граф имеет вид $s_1 \to s_2 \to s_3 \to \cdots \to s_{|s|}$). Пространство нитей — это множество нитей с дополнительной операцией $\to$ (практически превращающей множество в граф), долженствующей обозначать передачу сообщения от создателя к адресату. Единственное условие, накладываемое на эту операцию, очевидно:

$$\nu \to \mu \quad \Longleftrightarrow \quad \exists m \exists i, j : \ \nu_i = +m, \ \mu_j = -m \tag{2.2}$$

Или же, пространство нитей может рассматриваться как помеченный биграф с дугами двух типов, один из которых означает последовательность выполнения событий, а другой — направление передачи сообщения [17, 26]. Пространства нитей очень удобны в использовании, если верифицируемая система во многом зависит от передачи сообщений (например, при верификации протоколов, рассмотренной в разделе 4.1).

В восьмидесятых годах появилась первая *логика убеждений* под названием BAN (по первым буквам фамилий авторов: Барроуза, Абади и Нидхэма). Основная идея её была проста: предоставить разработчи-

[2]Ещё называемого замыканием Клини.

[3]Ориентированный граф, в котором каждая вершина, кроме двух: корневой и листовой — имеет по одной входящей и одной исходящей дуге. Корневая вершина имеет только одну исходящую, листовая — только одну входящую.

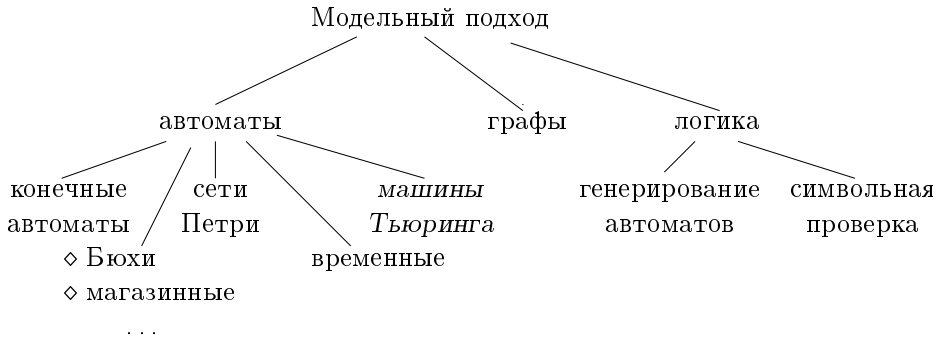


Рис. 2.3: Методы верификации, основанные на моделях

кам систем механизм определения убеждений сторон и их изменения во время работы системы [7], причём чтобы в результате анализа полученных данных можно было бы определить, на чём те или иные убеждения базируются. Кроме различных обозначений для разделённого владения ключами или другой информацией, в логике убеждений используются операции **believes** (такой-то участник верит в то, что какой-то факт имеет место), **said** (такой-то участник послал такое-то сообщение), **sees** (такой-то участник получил такое-то сообщение и может прочесть его и/или повторить) и другие. Это позволяет собирать фразы вроде P **believes** Q **said** X (P считает, что Q послал сообщение X), незначительно отличающиеся от предложений естественного языка. Кроме логики BAN [16], часто используются её модификации или подмножества: GNY [38] (в честь авторов Гонга, Нидхэма, Яхалома), AT [5] (Абади и Таттла) и SVO [55] (Сайверсона и ван Оорсхота).

Существует множество менее известных и менее успешных методов, в большинстве своём использующих менее глобальные механизмы, например, просто программирование на Прологе или даже определение нового языка логического программирования (например, реализованный на Standard ML язык $\mathrm{LO}_\forall$ в [13]). Язык программирования, использованный в [6], базируется на понятии т. н. *стабильной семантики*, [11] использует Пролог для реализации *консервативных абстракций* (Хорновские выражения служат там конструкторами и деструкторами передаваемых сообщений), многие переводят систему в область *программирования, ориентированного на ограничения* (constraint programming), и т. д. Одна из наиболее успешных систем — CoProVe [28], доступная в Интернете по адресу `http://130.89.144.15/cgi-bin/show.cgi`, базируется на системе Джонатана Миллена и Виталия Шматикова, гарантирует успешное завершение процесса верификации для любого конечного числа сессий.

Совершенно другой подход базируется на **моделях** (рис. 2.3). Главная идея его заключается в том, что по системе (то есть, по её спецификации) строится модель в виде автомата, который затем проверяется на удовлетворение некоторому свойству в каждом состоянии, которого он может достичь. Очевидно, что методы такого рода эффективны в том и только в том случае, когда система генерирует конечную поведенческую модель. Парадоксально, что модельные подходы особенно мощны именно потому, что заведомо ограничивают себя, работая не с системой, а с моделью (упрощённым вариантом системы). Это позволяет проверить желаемое свойство не только в наиболее вероятных ситуациях (как это делает тестирование), но вообще во всех возможных. Именно поэтому в критичных приложениях применяются именно модельные технологии (например, в [40] сделан формальный анализ программы управления полётом космического аппарата в том же верификаторе, который будет использоваться в данной работе). Тем не менее, результаты верификации в таком случае следует относить не к системе, а к её модели. Перефразируя и обобщив [10], можно сказать, что если модель $\mathcal{M}$ программы $\mathcal{P}$, построенная по спецификации $\mathcal{S}$, не обладает неким свойством, которое должно быть, судя по $\mathcal{S}$, присуще $\mathcal{P}$, ошибка может быть:

- **В спецификации** $\mathcal{S}$: возможно, ошибка была сделана на этапе формулировки требований и позже была (вольно или невольно) исправлена в программе. Или была исправлена не в программе, а в модели. Или не была исправлена вовсе — пока спецификация не будет приведена в порядок, мы не сможем ничего утверждать относительно программы и модели. (На ложных предпосылках возможны любые выводы).

- **В модели** $\mathcal{M}$: модель также является программой на языке, понятном верификатору, поэтому может содержать ошибки. Следует заметить, что, как правило, верификаторы не позволяют писать столь долгие и пространные программы, какие обычно скармливаются компиляторам: программа-модель обычно проста и понятна, что облегчает поиск ошибок в ней.

- **В программе** $\mathcal{P}$: это самый желаемый результат для заказчика и для того, кто занимается верификацией, и самый нежелательный для программиста. Он означает, что программа неверна и её надо изменять.

Если свойство не выполняется, программа-верификатор предоставляет контр-пример, состоящий из последовательности событий, ведущих к опасному состоянию системы.

Модельные подходы бывают двух типов: основанные на **автоматах** и на **логике**. В первом случае в качестве модели используется автомат, создаваемый по спецификации. Наиболее часто используемые случаи перечислены ниже.

1. *Конечные автоматы.* Конечные автоматы обычно представляются в виде графов (меченые состояниями узлы и меченые событиями дуги) или таблиц (события по строкам и состояния по столбцам). Первое представление как правило используется при небольших размерах графов, второе часто применяется в теории лексического разбора (ведь в ячейке таблицы на пересечении события и состояния можно записать не только следующее состояние, но и какие-то действия, например, печать символа или перекладывание его в стек). Несмотря на то, что на этот вариант работает большая теория, конечные автоматы применяются очень мало ввиду своей малой выразительности. Даже для очень простой системы конечный автомат (если он возможен) имеет огромные размеры. Бесконечные автоматы не применяются по очевидным причинам.

2. *Улучшенные конечные автоматы.* Тем не менее, некоторые попытки употребления конечных автоматов присутствуют, но при этом несколько изменяют оригинальную концепцию. Например, автоматы Бюхи, подробнее рассмотренные в разделе 2.4, представляют из себя конечные автоматы, где, во-первых, в метку каждой вершины внесены те предикаты, которые выполняются в соответствующем состоянии, и во-вторых, среди вершин выделяются вершины концевого типа. Вычисление в данном автомате считается возможным, если оно посещает хотя бы одну концевую вершину бесконечное число раз. Широко применяются также временны́е автоматы, где либо (*системы с дискретным временем*) введено понятие тика, либо (*системы с непрерывным временем*) каждая вершина снабжена инвариантом, указывающим на то, сколько времени можно в ней оставаться, а каждый переход — разрешающим условием по времени (условием, выполнение которого делает его разрешённым). Временные автоматы (особенно с непрерывным временем, которые при наивной интерпретации дают бесконечное (не счётное!) число состояний) лежат вне области интересов данной работы, хотя необходимость в них слегка рассмотрена в разделе 3.7. Также широкое применение получили магазинные автоматы всевозможных конфигураций.

3. *Сети Петри.* Конечные автоматы — это всего лишь урезанная вер-

сия сетей Петри. Каждый конечный автомат может быть выражен сетью Петри, и по каждой сети Петри можно построить эквивалентный автомат (вообще говоря, бесконечный). Сети Петри — это очень простой формализм, насколько вообще формализм для параллельных вычислений может быть таковым. Граф состоит из дуг, позиций и переходов. Дуга может вести из перехода в позицию или наоборот, но не может соединять узлы одного типа. Переходы обычно изображаются планками или прямоугольниками, позиции — кружками, а дуги — стрелками. Позиция может иметь одну или более (но обычно конечное число) фишек. Переход считается разрешённым, если в каждой позиции, соединённой с ним, есть хоть одна фишка (строго говоря, одна фишка должна приходиться на каждую дугу). Выполнение перехода означает поглощение каждой входящей дугой фишки из инцидентной позиции и отправление по одной фишке по каждой исходящей дуге. Обычно переходы модели отвечают за события системы, а позиции (и фишки в них) — за условия, разрешающие или запрещающие выполнение данного события.

Сети Петри сложны в использовании, но применяются весьма успешно. Существует автоматический верификатор (*PEP*) и регулярно проводятся конференции по проектированию и использованию сетей Петри.

4. *Машины Тьюринга.* В свою очередь, сети Петри представляют собой ограниченный вариант машины Тьюринга. Описание машины Тьюринга просто: бесконечная лента, читающая/пишущая головка и программа. Последняя выглядит как таблица, похожая на табличную форму записи конечного автомата, содержащая правила движения головки и изменения содержимого ленты в зависимости от текущего положения головки и текущего содержимого ленты (точнее, только видимой ячейки). Однако машина Тьюринга обладает памятью (бесконечной), чего нельзя сказать о конечном автомате. Из одного и того же состояния машина Тьюринга потенциально может перейти в любое другое состояние (число которых конечно или счётно), а конечный автомат — только в одно. Машина Тьюринга была создана как строго математически определённый эквивалент *алгоритма*, то есть всё, что может быть записано в алгоритмической форме, имеет также эквивалентную запись в виде машины Тьюринга.

   Описание машины Тьюринга включено здесь только для полноты

обзора, на практике в проверке моделей они не применяются по следующим причинам. Реализация произвольной машины Тьюринга невозможна. Проверка машин Тьюринга в качестве моделей алгоритмически неразрешима, и это можно строго доказать. Возможно, со временем появится возможность с помощью некоторых уловок и теорем ограничить её слишком широкие полномочия и таким образом сделать возможным проверку ограниченного подмножества машин Тьюринга, но пока что и это не проделано.

5. *Графы.* Теория графов как таковая используется очень редко, потому как чрезвычайно сложна в эксплуатации, не была совершена ни одна попытка приблизить её результаты к данной области, а также ввиду совершенно нединамической природы графов. Тем не менее, например, в [20] теория графов используется для доказательства пары теорем при верификации протокола. При этом каждому участнику коммуникации соответствует вершина, а используемому криптографическому ключу — дуга (то есть инцидентные вершины используют между собой один ключ). Мы воспользуемся нотацией теории графов в разделе 4.1 (см. теорему на стр. 55).

На практике обычно используют модельные подходы, основанные на **логике**. При этом верификатор берёт на себя самую чёрную работу: построение нужного автомата. Получившиеся автоматы существенно менее эффективны, зато могут достигать грандиозных размеров (страшно даже предположить, сколько времени может отнять рисование конечного автомата с миллионом-другим вершин!). Ниже мы дадим описание самого распространённого верификатора Спин (Spin), который позволяет описывать модель на сиподобном языке Промела и вводить свойства из спецификации в форме логики PLTL, после чего автоматически строит автоматы по обеим и пробует произвести их параллельный запуск.

Возвращаясь к небольшим дедуктивным методам (стр. 9), можно подчеркнуть такой их аспект: будучи не столь хорошо структурированы и интуитивны, как методы, описанные в самом начале раздела, они ищут контр-примеры для данного им свойства в достаточной мере хаотично и неоптимально, что приводит к неоправданно большому времени на поиск (в некоторых случаях было доказано, что задача поиска является NP-полной даже для конечного ограниченного числа параллельных сессий). Поэтому все эти методы ([6, 11, 13, 28, и др.]) нуждаются в уменьшении пространства для поиска, то есть требуют того же, чего и модельный подход. Борьба со слишком большим пространством для поиска ведётся с переменным успехом и очень разными методами: некоторые подходы

требуют хотя бы одного решения априори, другие работают интерактивно, пользуясь помощью человека, третьи требуют ограничить количество сессий или время работы системы, четвёртые разрешают себе наряду с верными решениями докладывать и неправильные, и т. д.

2.3 Проблемы модельных подходов и пути их решения

Модельные подходы, при всех своих достоинствах, встречаются с некоторыми проблемами, основные из которых перечислены ниже [13, 26, 28, 39, 49, и др.]:

Проблема взрыва числа состояний: экспоненциальное увеличение размера конечной модели при увеличении числа компонент системы (процессов, каналов, счётчиков, переменных, ветвлений, etc). Разнообразные методы применяются при решении этой проблемы, и в данный момент можно с уверенностью сказать, что их вполне достаточно для того, чтобы сделать возможной верификацию аппаратного обеспечения любой степени сложности. К сожалению, системы программного обеспечения имеют тенденцию быть куда более сложными, так что здесь не всегда известных методов может быть достаточно. Автор модели должен хорошо разбираться как в моделируемой области (для того, чтобы суметь отбросить несущественные детали, сохранив ценность модели), так и в процессе моделирования. Последний аспект может подчас стать критическим, если с помощью тонкостей работы верификатора уменьшить число состояний до приемлемого без изменения смысла модели.

Проблема конструирования модели: преодоление суровой семантической разницы между тем, что выдают разработчики программного обеспечения, и тем, что принимают существующие инструменты верификации. Основная разработка идёт на так называемых языках программирования общего назначения[4] (Си++, Ява, Ада, ОКамл, Кобол) и даёт огромные листинги, достигающие в размерах многих сотен модулей и миллионов строк кода, тогда как большинство инструментов верификации воспринимают небольшие тексты на очень высокоуровневых языках, специально разработанных для восприятия очень специфичных вещей (алгебры процессов, состояний порождённых машин, временно́й логики и т. п).

[4]Большая часть которых не являются на деле таковыми. Так, Ява для подавляющего большинства задач слишком медленна, а Кобол — не более как язык узкого назначения (domain specific language).

Одно из наиболее известных приложений для автоматической генерации моделей по исходному коду на ЯВУ — Java PathFinder [14]. Оно может создать модель по любой программе на языке Ява, но это ещё не означает, что полученная модель может быть верифицирована. Для оптимизации и урезания таких моделей используется, например, Bandera [25, 39], простая программа, отсекающая от модели всё лишнее в соответствии с множеством свойств, которые будут у этой модели проверяться. Строго говоря, Java PathFinder является просто компилятором из байт-кода виртуальной ява-машины в Промелу, а Bandera — сложным препроцессором, использующим для внутреннего представления семантической структуры язык Jimple. Подробнее этот метод и обратный ему рассмотрены на стр. 36.

Более серьёзный подход представляет собой [46], где с нуля создаётся верификатор моделей, изначально ориентированный на определённый язык (Эрланг). Успех многоязыковой платформы .NET позволяет ожидать в самом ближайшем будущем подобный инструментарий и для неё, в том числе и не только ориентированных на язык C#.

Проблема требований спецификации: спецификации пишутся в достаточно вольной форме, что, с одной стороны, не исключает внутренних противоречий, а с другой, поражает количеством несвоевременно употреблённых терминов (*локальные переменные, контрольные точки программы,* etc), принадлежащих совсем другому уровню абстракции. Верификаторы понимают формулы в рамках строго определённой (временной) линейной логики. Другими словами, иногда возникают трудности с выражением требований спецификации на языке, понятном верификатору, что приводит к двусмысленным толкованиям спецификации и, как следствие, двоякой верификации модели.

Эта проблема на даёт такого простора для решения, как предыдущая. Единственный путь заключается в стандартизации описания требований: либо с использованием специализированного языка, либо, например, UML или подмножества XML. Успехи модельной верификации во многом объясняются её строгой математической базой, и любые покушения на неё пагубны.

Проблема интерпретации результатов: как уже было сказано выше, результат верификации не всегда легок в интерпретации. В [53] приводится пример из проверки модели планировщика операционной системы реального времени DEOS, когда контр-пример, выданный Спином, имел длину 2700 шагов. При такой длине уже очень нелегко определить, что происходит при его выполнении и к чему относится ошибка: к модели, к спецификации или всё-таки к программе. Во-первых, сама длина контр-примера требует часов для трассировки, во-вторых, сам

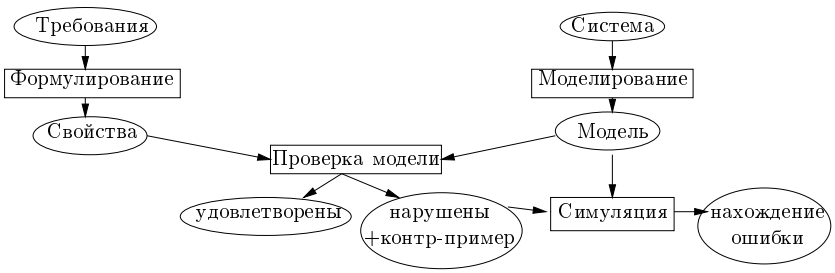


Рис. 2.4: Схема процесса верификации при использовании моделей

контр-пример выражен в терминах низкоуровневой оптимизации описания модели, а не в терминах самой модели.

Во многих современных верификаторах (Спин, безусловно, в их числе) присутствует опция нахождения минимального эквивалентного контр-примера. Такой процесс требует некоторого дополнительного времени, но иногда позволяет сократить контр-пример вдвое или даже втрое, во столько же раз уменьшив работу автора модели. Контр-пример со стр. 71 данной работы был уменьшен таким образом с 3677 до 30 шагов!

Схему, представленную ранее на рис. 2.1, можно изменить и расширить для представления модельного подхода. Как видно из рис. 2.4, модельная верификация может дать один из двух результатов: либо сформулированные свойства удовлетворены, то есть система работает так, как хотелось бы, либо свойства нарушены. В последнем случае верификатор предоставляет контр-пример, с помощью которого можно, симулируя работы модели, добраться до того состояния, которое нарушило свойство. Анализ этого состояния и пути к нему позволяет локализовать ошибку либо в модели, либо в спецификации, либо в программе.

2.4 Инструментарий

Спин[5] позволяет производить симуляцию спецификации, описанной на языке Промела (PROtocol/PROcess MEta LAnguage), а также верификацию нескольких типов свойств. Ниже следует очень краткая выжимка из [43, 44, 45], описывающая сам язык и используемую верификатором логику. Дальше будет рассмотрено использование препроцессоров для этого языка и компиляторов в него.

[5] http://netlib.bell-labs.com/netlib/spin/whatispin.html

Промела — метаязык протоколов

Программа на языке Промела (обычно называемая спецификацией[6]) состоит из определений последовательных процессов, переменных (локальных и глобальных) и каналов общения процессов. *Процесс* `P` определяется следующим образом:

```
proctype P(параметры)
{
 локальные определения переменных;
 операторы
}
```

Эта конструкция описывает процесс, его внутреннюю структуру и поведение, но не запускает его на выполнение. В каждой спецификации присутствует секция инициализации, на которой лежит ответственность за собственно запуск процессов. Параметры — это формальная часть. Их можно использовать, к примеру, для того, чтобы различать экземпляры одного и того же процесса. При запуске процесса необходимо указывать значения этих параметров, например:

```
proctype P(a,b) {...}

init
 {
  run P(5,10);
  run P(8,12);
 }
```

Процесс без параметров может быть запущен неявно, если перед его названием написать `active proctype`... . Всего в каждый момент работы модели может быть не более 255 активных (незавершившихся) процессов.

Каждый оператор Промелы может быть либо *разрешён*, либо *заблокирован*. В первом случае он может быть выполнен, во втором — нет, что-то этому мешает. Если данная строка процесса заблокирована, выполнение этого процесса не может быть продолжено. Если все процессы остановлены, программа зашла в тупик.

Правила разрешения операторов таковы:

[6]Этот термин не очень удачен, как и, например, название схем UML *диаграммами*. Схемы на UML могут и не иметь форму диаграм, а на нашем довольно подробном уровне абстракции разница между моделью и спецификацией важна.

- *Пустой оператор:* `skip` — ничего не делает, всегда разрешён.
- *Оператор присваивания:* `x = 7` — всегда разрешён, работает так же, как в большинстве императивных языков: присваивает данной переменной данное значение. Следует отметить, что Промела — язык со слабой статической типизацией.
- *Выражение:* `x==5`, `1` или `(x+y)&&(!u)` — разрешено, если его вычисление даёт не ноль (и не `false`, который в Промеле, как и в Си, равен нулю). То есть первый пример выражения эквивалентен следующей псевдоконструкции: `while (x!=5) skip;`

  Истоки подобной философии лежат в работе Эдсгера Дейкстры об охраняемых командах и недетерминизме [32].
- *Оператор выбора:*

```
if
:: условие1 -> операторы1;
:: условие2 -> операторы2;
:: ...
:: условиеN -> операторыN;
fi;
```

  — где разрешённость условия$_j$ позволяет операторам$_j$ быть запущенными. Недетерминизм возможен (и даже очень часто используется), то есть если несколько условий разрешены, Спин случайным образом выбирает, какой блок операторов запускать. Если выполнение операторов блокируется где-то на полпути, верификатор не возвращается к другим ветвям, в терпеливо ждёт разрешения продолжить. Операторы выбора могут быть вложенными. Последнее условие может быть сформулировано в виде `else` — оно разрешается в том и только в том случае, когда все остальные условия заблокированы, то есть является краткой нотацией для `(!условие1)&&(!условие2)&&` $\cdots$. Если такого условия нет и все имеющиеся заблокированы, весь оператор выбора блокирован. Истинные условия («`(1)->`», «`true->`») могут быть опущены, если следующий за ними оператор разрешен. Например,

```
if
:: skip -> x=7;
fi;
```

эквивалентно

```
if
:: 1 -> x=7;
fi;
```

или

```
if
:: x=7;
fi;
```

- *Оператор цикла:*

```
do
:: условие1 -> операторы1;
:: условие2 -> операторы2;
:: ...
:: условиеN -> операторыN;
od;
```

— представляет собой тот же оператор выбора, повторяемый бесконечно. Для выхода из цикла может использоваться оператор `break` или переход по метке.

- *Оператор безусловного перехода* `goto x` — осуществляет переход по метке `x`. Всегда разрешён, хоть и может вести в наглухо заблокированное состояние. Ясно, что его комбинация с условным оператором может заменить оператор цикла.

- *Оператор посылания:* `c!v` — посылает сообщение `v` по каналу `c`. Разрешён, если канал может принять сообщение (то есть либо буфер не заполнен, либо, в случае буфера нулевой длины, другой процесс ожидает получения). О каналах будет рассказано на стр. 21.

- *Оператор получения:* `c?v` — получает сообщение `v` из канала `c`. Если `v` — переменная, содержимое сообщения кладётся в неё, если константа, осуществляется проверка. Разрешён, если в канале есть сообщение (либо буфер не пуст, либо, в случае буфера нулевой длины, другой процесс в данный момент посылает) и оно соответствует `v` (для переменной — по типу данных, для константы — по значению).

Другие часто используемые конструкции выражаются через этот базис следующим образом:

- *Цикл с параметром:*

```
counter=lower
do
:: counter<=upper -> операторы; counter++;
:: else -> break;
od;
```

- *Вечный цикл с параметром:*

```
counter=lower
do
:: counter<=upper -> операторы; counter++;
:: else -> counter=lower;
od;
```

- *Выбор случайного числа:*

```
if
:: random=lower;
:: random=lower+1;
:: ...
:: random=upper;
fi;
```

- *Перемешивание массива:*

```
do
:: !done[0] ->
    list[counter]=values[0]; counter++; done[0]=true;
:: !done[1] ->
    list[counter]=values[1]; counter++; done[1]=true;
:: ...
:: !done[N] ->
    list[counter]=values[N]; counter++; done[N]=true;
:: else -> break;
od;
```

- *Необязательный блок:*

```
preblock;
if
:: (1) -> block;
:: skip;
fi;
postblock;
```

Переменные объявляются в Промеле так же, как и во многих других императивных языках программирования. Они могут быть глобальными (объявляться в начале программы) или локальными (в начале процесса). Каждый экземпляр процесса имеет свой набор локальных переменных. Доступ к чужим пространствам имён невозможен.

`bit c;`	$\{0, 1\}$
`bool b;`	$\{false, true\}$
`byte i;`	$[0, 255]$
`short j;`	$[0, 65'535]$
`int k;`	$[0, 4'294'967'296]$

Переменные могут быть инициализированы любым подходящим значением в момент объявления, по умолчанию это 0 или `false`. Спин не занимается разрешением проблем несовместимости типов, компилируя все ошибки в Си, где они и обнаруживаются на этапе выполнения (верификации). Из сложных типов данных Промела содержит одномерные массивы (`byte array[10]`), записи (`{byte i; int n;}`) и тип *«сообщение»* (будет рассмотрен ниже). Строк и символов в Промеле нет.

Особое понятие, встречающееся далеко не во всяком языке программирования (даже высокого уровня), но имеющееся в Промеле — *канал*. Канал представляет собой средство общения процессов друг с другом. Другого средства обмена информацией у них нет (за исключением, конечно, глобальных переменных). Канал объявляется следующим образом:

```
chan name = [N] of {type};
```

где `name` — имя переменной канала, N — длина буфера, а `type` — тип передаваемых по каналу сообщений. Буфер пассивный, то есть манипулировать им нельзя. Например:

```
chan socket = [5] of {int,int};
```

объявляет канал `socket`, который может передавать два длинных целых за раз и поддерживать очередь из не более чем пяти сообщений. Один раз на спецификацию можно объявлять перечислимый тип *сообщение* и затем его использовать (не обязательно в объявлении канала):

```
mtype = {data,ack,negack,token};
chan newsocket = [2] of {mtype,int};
```

Канал можно объявить и с буфером нулевой длины, что чаще всего и делается. Такой канал называется асинхронным или рандеву-каналом. Так как он не имеет места для хранения даже одного сообщения, то соответствующие операторы посылания и получения должны выполняться одновременно. Если это по каким-то причинам не может произойти, система блокируется.

Имея возможность описывать процессы для параллельного выполнения, хотелось бы также получить аналогичную возможность для описания последовательного выполнения. Такая возможность имеется и называется *подключёнными функциями*. На самом деле они являются макросами, но могут применяться как функции. Тем не менее, для упрощения системы следует помнить, что содержимое подключённой функции вставляется вместо каждого её вызова, что оказывает своё пагубное влияние на размер модели. Но отказываться от них не стоит, ибо подключённые функции могут существенно улучшить читаемость спецификации. Функции определяются и используются следующим образом:

```
inline check(n)
{
 if
 :: n>MAXINT  -> printf("ВНИМАНИЕ: %i - слишком много!\n",n);
 :: n<-MAXINT -> printf("ВНИМАНИЕ: %i - слишком мало!\n",n);
 :: else;
 fi;
}

...

do
:: ...
   check(a);
   ...
   check(b);
```

```
   ...
:: ...
   check(x);
   ...
   check(y);
   ...
od;
```

Кроме того, можно пользоваться препроцессором Промелы, синтаксис которого схож с сишным препроцессором, но существенно более беден:

```
#define MAXINT 100
#define check(n) assert((n>MAXINT)&&(n<-MAXINT))
```

Но при этом в режиме симуляции Спин будет пользоваться не функциями, а тем, что он подставил на их место (в том и состоит отличие препроцессора от макросов).

Кроме определения, препроцессор поддерживает и подключение файлов (`#include "file.pr"`), что часто используется для повторной верификации той же формулы.

С технической стороны может быть полезно знать, что Спин не реализует функции препроцессора, но пользуется оригинальным сишным. Однако этот факт редко используется, и практически во всех моделях можно встретить только слова `#define` и `#include`.

Иногда возникает потребность в атомарном выполнении нескольких операторов, то есть таком выполнении, которое происходит за один шаг. Мы уже видели такого рода связь между различными процессами (в рандеву-каналах), теперь же обратимся к тому, как это реализовать внутри одного процесса. Объединение нескольких последовательных операторов в атомарную группу делается так:

```
atomic{ оператор1;
        оператор2;
        ...
        операторN; }
```

или так:

```
d_step{ оператор1;
        оператор2;
        ...
        операторN; }
```

Разница между этими двумя методами состоит в том, что если один из операторов (скажем, `оператор2` окажется невыполнимым), то в первом случае атомарность будет сломана, а во втором — система зайдёт в тупик. То есть, второй способ сильнее, но может приводить к неожидаемым тупикам.

Автоматы Бюхи

Формализм, который используется при работе верификатора — автоматы Бюхи. По каждому процессу и каждой формуле строится автомат Бюхи, и все они после этого запускаются параллельно. Если такая система работает, значит, работает и спецификация на Промеле.

Автомат Бюхи — это конечный автомат, имеющий вершины двух типов: обычные и *концевые*. Выполнение автомата считается *допустимым* тогда и только тогда, когда хотя бы одна концевая вершина посещается бесконечное число раз. Спин использует слегка оптимизированную версию автоматов Бюхи, в которых конечные запуски также допускаются, если обрываются на концевой вершине. Очевидно, что такой автомат эквивалентен автомату Бюхи с добавленными дугами из всех концевых вершин в самим себя.

Каждая вершина автомата соответствует состоянию системы, тогда как каждая дуга соответствует переходу системы из состояния в состояние. Каждая вершина помечена информацией о динамике выполнения всех процессов и о содержимом всех переменных (а также буферов всех каналов, если такие есть). Эта информация носит название *вектора состояния* и однозначно отделяет одно состояние от другого. Таким образом, в сгенерированном Спином автомате не может быть дублирующих вершин (то есть получаемый автомат минимален!), потому что вершина идентифицируется своим вектором состояния. Ясно, что если векторы состояния двух вершин отличаются хотя бы в одном бите, эти состояния не эквивалентны, потому что в них либо хотя бы одна переменная имеет разные значения, либо выполнение хотя бы одного процесса находится на разных стадиях.

Общее максимально возможное количество состояний определяется через длину вектора состояний как 2^{8n}, где n — длина вектора состояния в байтах. Но на практике такой максимум никогда не достигается, потому что редко в каких программах используются все до одного значения каждой переменной.

Несмотря на это, длина вектора состояния может характеризовать сложность решаемой задачи: количество требуемого времени, занимаемой памяти и т. п. (хотя бы потому, что памяти для хранения состояний

требуется $m \cdot n$, где n — длина вектора состояния и m — количество этих состояний). Длина до десяти байт обычно встречается в очень простых программах, либо учебных примерах, либо реальных моделей, написанных на очень высоком уровне абстракции. Длина вектора состояний до полусотни встречается чаще всего и означает сравнительно лёгкую (нересурсоёмкую и быструю) верификацию. При длине до сотни верификация становится затруднительной, может возникнуть необходимость использования некоторых специализированных опций верификатора (сжатие данных в памяти и т. п.) и — такого исхода тоже не следует исключать — она может оказаться неосуществимой.

Умелая компоновка модели, отбрасывание несущественных аспектов спецификации, правильный выбор уровня абстракции, довольствование небольшим количеством переменных и предпочтение буферизованным каналам рандеву-каналов, равно как и конгруэнтное внутреннее строение верификатора и верный выбор его опций — всё это может существенно сократить длину вектора состояний и перевести вашу спецификацию в разряд легко верифицируемых.

PLTL — линейная временна́я алгебра высказываний

Требования, которые можно предъявлять к модели, представляют собой описание либо желательного поведения системы (тогда они должны выполняться), либо нежелательного поведения (тогда они не должны выполняться). Для описания поведения системы, реагирующей на внешние раздражители, и имеющей вообще говоря параллельную структуру, обычной алгебры высказываний недостаточно. Нужен формализм для выражения того, что одно высказывание имеет место *до* другого, и т. п. Таким формализмом является PLTL, propositional linear temporal logic, линейная временна́я логика высказываний.

Для последовательных программ поведение может описываться с помощью предусловий и постусловий. $\{\Phi\}S\{\Psi\}$ называется тройкой Хоара [41], где Φ — предусловие (формула алгебры предикатов над константами, переменными и функциями, возможно, содержащая кванторы), Ψ — аналогичного вида постусловие, а S — строка программы (оператор и/или вызов функции). Например:

$$\{\exists i : a_i = 0\}\texttt{removeAllZeros(a)}\{\forall i : a_i \neq 0\}$$

Этот метод напрямую неприменим к параллельным программам (доказательство можно увидеть в [49, глава 3]). Похожий метод работает и для параллельных программ.

Имеется некоторое множество $\mathcal{A}$ элементарных высказываний. В любой момент выполнения программы каждое элементарное высказывание может быть истинным или ложным. Для каждого состояния модели можно определить множество тех элементарных высказываний, которые в этом состоянии являются истинными. Пусть множество состояний обозначается как $\mathcal{S}$, тогда мы считаем функцию интерпретации $f_i : \mathcal{S} \to 2^{\mathcal{A}}$ полностью определённой. (Аналогично, можно определить функцию ценности $f_v : \mathcal{A} \to 2^{\mathcal{S}}$, которая каждому элементарному высказыванию ставит в соответствие множество состояний, в котором оно истинно). Алгебра высказываний классически определяется следующим образом [3]:

- Любое элементарное высказывание $p \in \mathcal{A}$ и любой символ логической переменной — формулы алгебры высказываний.
- Если φ — формула алгебры высказываний, то $(\neg\varphi)$ — также формула алгебры высказываний.
- Если φ и ψ — формулы алгебры высказываний, то $(\varphi \vee \psi)$ — также формула алгебры высказываний.
- Всё остальное — не формулы алгебры высказываний.

Скобки могут быть опущены в соответствии с приоритетами операций. Коньюнкция, импликация, эквивалентность и пр. определяются через отрицание и дизъюнкцию:

$$
\begin{aligned}
(\varphi \wedge \psi) &\equiv (\neg((\neg\varphi) \vee (\neg\psi))) && (2.3)\\
(\varphi \Rightarrow \psi) &\equiv ((\neg\varphi) \vee \psi) && (2.4)\\
(\varphi \Leftrightarrow \psi) &\equiv ((\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)) && (2.5)\\
1 &\equiv (\varphi \vee (\neg\varphi)) && (2.6)\\
0 &\equiv \neg 1 && (2.7)
\end{aligned}
$$

Множество всех формул алгебры высказываний обозначим через Φ. Функция интерпретации f_i легко обобщается до оператора приемлемости $\models : \mathcal{S} \times \Phi$.

$$
\begin{aligned}
s \in \mathcal{S}, p \in \mathcal{A}, \quad s \models p &\iff p \in f_i(s)\\
s \in \mathcal{S}, \varphi \in \Phi, \quad s \models \neg\varphi &\iff s \not\models \varphi\\
s \in \mathcal{S}; \varphi, \psi \in \Phi, \quad s \models (\varphi \vee \psi) &\iff s \models \varphi \text{ или } s \models \psi
\end{aligned}
$$

Временна́я логика является подклассом модальной логики. Она допускает выражения вроде «*когда-нибудь* φ». Временна́я логика построена на основе алгебры высказываний с добавлением понятия времени. В PLTL время считается линейным и дискретным (существуют другие временны́е логики, где, например время является непрерывным или имеет древовидную структуру: CTL, TCTL, CTL*). Определяется она следующим образом:

- Любое $p \in \mathcal{A}$ — формула PLTL.
- Если φ — формула PLTL, то $(\neg\varphi)$ — также формула PLTL.
- Если φ и ψ — формулы PLTL, то $(\varphi \vee \psi)$ — также формула PLTL.
- Если φ — формула PLTL, то $\mathsf{X}\varphi$ — также формула PLTL.
- Если φ и ψ — формулы PLTL, то $\varphi\mathsf{U}\psi$ — также формула PLTL.
- Всё остальное — не формулы PLTL.

$\mathsf{X}\varphi$ означает, что в следующем состоянии должно выполняться φ. $\varphi\mathsf{U}\psi$ означает, что должно выполняться φ, пока ψ не станет истинным.

Вторичные операторы PLTL определяются так:

$$\mathsf{F}\varphi \equiv 1\mathsf{U}\varphi \tag{2.8}$$
$$\mathsf{G}\varphi \equiv \neg\mathsf{F}\neg\varphi \tag{2.9}$$

$\mathsf{F}\varphi$ означает, что φ должно быть выполнено когда-то в будущем. $\mathsf{G}\varphi$ означает, что φ должно быть истинным всегда (в каждом состоянии). Альтернативной нотацией [51] является $\Diamond$ вместо F, $\Box$ вместо G и $\circ$ вместо X. Подводя итоги, можно сказать, что обычная формула (без временны́х операторов) относится к текущему состоянию модели, X переносит действие в следующее состояние, G относится ко всем состояниям, включая текущее, а F — к какому-то одному в будущем или к текущему.

Задача удовлетворения данной формулы данной моделью является NP-полной, то есть даже лучшие из известных алгоритмов экспоненциальны по отношению к длине проверяемой формулы. Из соображений оптимизации оператор X в Спине не реализован, все остальные имеют следующий вид:

- $\neg p \Longleftrightarrow$ `!p`
- $p \vee q \Longleftrightarrow$ `p || q`

- $p \wedge q \Longleftrightarrow$ `p && q`
- $p \Rightarrow q \Longleftrightarrow$ `p -> q`
- $p \mathsf{U} q \Longleftrightarrow$ `p U q`
- $\mathsf{F} p \Longleftrightarrow$ `<> p`
- $\mathsf{G} p \Longleftrightarrow$ `[] p`

В виде p и q можно использовать выражения с константами и переменными либо предикаты типа `P@L`, которые имеют значение истина только тогда, когда процесс `P` находится в состоянии, помеченном меткой `L`.

По отрицанию формулы Спин создаёт автомат Бюхи, который запускается на выполнение совместно (параллельно) с моделью. Если такая комбинированная система верифицируется нормально (произведение автоматов пусто), то формула выполняется. Если же Спин находит контр-пример, следовательно, отрицание формулы в некотором состоянии выполняется, значит, сама формула нарушается. Сгенерированный автомат Бюхи также имеет форму текста на Промеле, поэтому его можно сохранить в файл и подключать с помощью команды препроцессора `#include "file.ltl"`.

Также верификацию можно проводить без формул. По умолчанию система проверяется на конечные состояния, то есть на отсутствие тупиков. Некоторые проверки (см. стр. 72) гораздо удобнее осуществлять внутри программы с помощью механизма *утверждений*. Выражение `assert(p);` всегда является выполнимым, при этом если `p` выполняется, то оно эквивалентно оператору `skip` (то есть не производит никаких действий), если же `p` не выполняется, верификация (или симуляция) модели прерывается с сообщением об ошибке.

Компиляторы переднего плана (front-end)

Тот факт, что Промела является языком очень высокого уровня, даже вкупе с тем фактом, что она компилируется в Си, не означает, что не существует компиляторов в Промелу. Они существуют и процветают, используясь очень широко.

Одной из самых известных систем является Java PathFinder [14], компилирующий Яву в Промелу, то есть автоматически строящий модель по данному исходному тексту на Яве. Этот подход революционен ввиду критической разницы в уровнях языков, имеющей к тому же

нетрадиционную направленность (представьте себе компилятор из Ассемблера в Перл!), и ввиду этой же разницы не всегда успешен (в некоторых источниках он именуется «производством свиней из колбасы»). Модели, построенные автоматически, чересчур громоздки и результаты верификации (даже если она возможна) чересчур трудны в интерпретации. Также возможен и обратный подход, когда сначала пишется модель, а по ней автоматически генерируется код (см. стр. 36).

Другой пример представляет собой Bandera [25, 39], извлекающая из программы на Яве только те участки кода, которые могут оказать влияние на проверяемые формулы. Таким образом, для каждой интересующей нас формулы можно подобрать такой автомат, который при небольшом размере мог бы способствовать правильному моделированию. В разделе 3.7 рассказано о построениии регионных автоматов с использованием того же принципа: усечения модели согласно условию задачи (формулам).

2.5 Подведение итогов

В данном разделе был произведён обзор области моделирования и проверки моделей, коснувшийся основных методов, которые будут использованы далее. Верификация — это процесс, удостоверяющий, что данная система удовлетворяет данным требованиям или имеет данные свойства. Существует много методов верификации, принадлежащих к двум большим категориям. Дедуктивная методология пытается формализовать систему целиком и далее пользоваться всеми достоинствами формального описания. Модельная методология строит модель системы, которая, с одной стороны, в рамках рассматриваемой задачи неотличима от оригинала, а с другой стороны — её проще создать в рамках какого-то формализма.

Модельные подходы при всех своих достоинствах (простоте, дешевизне, легкости интерпретации результатов) обладают рядом проблем: слишком большие (подробные) модели не могут быть проверены, автоматическое построение моделей по коду программ чересчур сложно, спецификация также должна быть формализована. Наконец, результат проверки модели может выявить ошибку не в системе, а всего лишь в модели или в самой спецификации.

Наиболее широко используемым верификатором является Спин, позволяющий описывать модели на сиподобном языке Промела. Страницы 16–23 посвящены подробному объяснению этого языка. Для проверки пунктов спецификации Спин предоставляет возможность формально

описывать спецификацию в виде формул временно́й алгебры высказываний.

Всё содержимое этого раздела будет использовано дальше. Следующая глава содержит обзор области распределённых систем с оглядкой на наши возможности в верификации, глава за ней применяет эти возможности на практике.

Глава 3

Моделирование распределённых систем

3.1 Особенности распределённых систем

Определение. *Распределённая система есть множество взаимозависимых или взаимосвязанных частей, работающих как единое целое* [47].

Таким образом, в определении можно выделить две смысловые концепции. Во-первых, распределённая система может быть разобрана на части, которые каким-то образом связаны или зависимы между собой. Во-вторых, когда эти части работают вместе, они формируют поведение системы в целом. Как велосипед состоит из многих деталей разной формы и назначения, но позволяет, собрав их вместе (в нужном порядке), передвигаться с большой скоростью, так и распределённая система, собранная из множества более мелких программ, вместе предоставляет качественно новые возможности.

Когда используется термин *распределённая система*, может иметься в виду широкий спектр значений. Обычно под распределённой системой понимается компьютерная программа или больший проект, система из нескольких программ, выполнение которой распределено в пространстве и/или во времени. Распределение в пространстве появилось сравнительно недавно (как и употребление термина в данном контексте), с появлением сети Интернет с одной стороны и многочисленных устойств, которые могут быть к ней подключены, с другой. В наши дни не только рабочие станции и домашние компьютеры имеют доступ к сети, но и мобильные телефоны, переносные и карманные компьютеры, ноутбуки, банковские и телефонные автоматы, телевизоры, равно как и

всевозможные компьютеризированные системы от автомобилей до холодильников. Встроенные системы в последнее время оказывают всё большее влияние на структуру распределённых систем и их разработку.

Распределение во времени имеет более долгую историю, но от этого не становится менее значимым. Например, банки имеют небольшой строго отмеренный отрезок времени, в течение которого могут совершаться денежные переводы между счетами в разных банках. Но требовать соблюдать это правило для обычных пользователей (например, при использовании оплаты по кредитной карте через Интернет), не очень дружелюбно и совершенно для них не удобно. Некоторые сети передачи данных имеют строго фиксированное время для передачи привилегированного трафика (например, час ZMH в сети Fidonet для передачи нетмейла, личной почты). Аналогичным образом, только крупные узлы сети должны соблюдать это правило, рядовые пользователи могут о нём и вовсе не знать, пользуясь своими станциями круглосуточно или по любой другой индивидуальной схеме работы узла (для Fidonet определяется флагами узла в нодлисте). Пожалуй, система передачи сообщений является наиболее старой системой, распределённой во времени.

Наиболее древней топологией распределённой системы является система клиент-сервер (рис. 3.1, слева вверху). Сервер в данном случае представляет собой пассивную сторону, предоставляющую некоторые возможности и ждущий обращения к нему. Клиент при этом — активная сторона, запрашивающая сервер о выполнении того или иного действия. Сейчас, несмотря на появление альтернативных топологий, система клиент-сервер остаётся наиболее распространённой из используемых в небольших системах. Типичный пример системы клиент-сервер — веб-сервер, обслуживающий WWW-запросы, идущие от браузеров пользователей сети. Браузер посылает на сервер запрос с именем желаемого файла, который обрабатывается сервером и клиенту возвращается либо запрошенный файл, либо сообщение об ошибке. В качестве другого примера можно привести банкомат, который может показать клиенту состояние его счета или снять с него деньги, но делает это только по соответствующему запросу (защищённому паролем и опознавательным знаком пользователя (смарт-картой или картой магнитной памяти)).

Двухъярусная топология (two-tier) представляет собой тот случай, когда сервер не является последней инстанцией (рис. 3.1, слева внизу). Другими словами, сервер действительно предоставляет некоторые возможности своим клиентам, но не сам занимается выполнением связанных действий, а делегирует их другому серверу. Например, сервер может по получении запроса сначала связываться с отдельным сервером данных, получать от него всю информацию об аутентифицированном

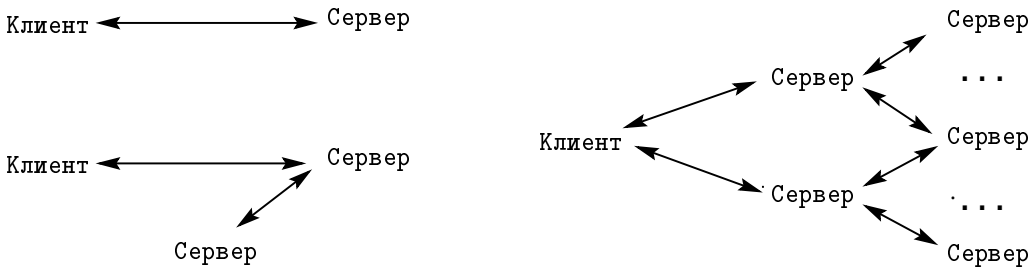


Рис. 3.1: Топологии распределённых систем

пользователе и только после этого продолжать работу. На определённом уровне абстракции двухъярусная топология *может* считаться топологией клиент-сервер, потому что является лишь более подробной и эффективной вариацией последней.

Логичным продолжением является многоярусная топология (multi-tier), в которой сервер, предоставляющий некие возможности клиенту, может обращаться к нескольким разным серверам (рис. 3.1, справа). Например, система веб-почты может иметь отдельный сервер для хранения файлов данных о пользователе (логин, пароль, имя, адрес, настройки), другой сервер для хранения собственно почты, третий сервер, работающий собственно как стандартный агент передачи сообщений (MTA, mail transfer agent), и четвёртый — сайт. Запрос пользователя к сайту заставляет его сначала обратиться к первому для проверки пароля пользователя и возможной загрузки его конфигурации, затем ко второму для выдачи на экран имеющихся сообщений. При этом тот же сервис предоставляется и агентом передачи сообщений (третьим вышеописанным сервером), когда его используют, например, через протокол POP3. Однако, при этом этот агент передачи сообщений обращается к тому же серверу хранения сообщений, что и сайт веб-почты, что безусловно экономит ресурсы и позволяет, усложнив архитектуру, упростить её реализацию.

Итак, распределённые системы получают всё большую популярность в самых разных кругах. При этом, безусловно, новые возможности требуют новых архитектур, а новые архитектуры ведут к новым технологическим решениям. Далее в этой главе будут рассмотрены приложения верификационных техник к этой области. Очевидно, что другие решения требуют других методов проверки их правильности, но в равной мере очевидно и то, что большинство новых проблем могут быть сведены к старым и таким образом оказаться решёнными.

Главная проблема распределённых систем состоит в том, что раз-

ные их части работают по-разному, с разной скоростью, встречают разные проблемы и т. п., но при этом работают в большинстве случаев одновременно и взаимозависимо.

3.2 Части систем, подверженные эффективной верификации

Любая система программного обеспечения вне зависимости от своей распределённости или нераспределённости может быть поделена на части. Таким образом, здесь усложнения не предвидится. Скорее даже наоборот: в случае распределённой системы (см. определение на стр. 31) границы между частями системы выражены куда более чётко. Но некоторое усложнение всё равно присутствует, как будет видно ниже.

Любая задача, кроме лежащей в классе элементарных, может быть поделена на *подзадачи*. Умножение сводится к сложению; передача файлов по сети сводится к установлению соединения и передаче пакетов по нему; задача создания и поддержания музыкального сервера в Интернете сводится к большому множеству подзадач, начиная от хранения файлов и перевода данных в альтернативные форматы и кончая способами расширения рядов пользователей и удовлетворения исков компаний, занимающихся выпуском музыкальных дисков. Иногда также для решения специфичных проблем, размазанных по сотням тысяч строк кода, можно составить небольшой *алгоритм*, который будет иметь смысл и чисто теоретически, в отрыве от контекста.

В распределённых системах также имеет смысл говорить о соединениях. Более современный термин: *ассоциации*, то есть соединения не только одной стороны с другой, но и множественные связи. *Протоколы*, или интерфейсы, также занимают здесь свою нишу. На разных уровнях абстракции и в различных контекстах можно говорить об эквивалентности протокола с сервисом, который предоставляется с его помощью. Также можно выделить особняком подкласс протоколов, который занимается некоторым большим действием, которое должно свершиться распределённо (то есть одинаково в разных местах). Подобные действия называются, как правило, *транзакциями*. В четырёх следующих разделах все перечисленные здесь части распределённых систем будут описаны на должном уровне детализации.

3.3 Верификация программ по раздельности

Как известно, путь разработки программы начинается с предъявления требований, которым она должна удовлетворять, затем следует анализ этих требований с желанием определить оптимальные пути решения проблемы, за ним идёт этап проектирования, когда создаётся каркас программы, затем следует фаза реализации, и весь путь завершается поддержкой работоспособной системы.

Так выглядит классическая модель нисходящей разработки, известная также как модель водопада [54]. С тех пор, как она была впервые использована, прошло много времени, с тех пор было изобретено множество других моделей, так или иначе её изменяющих, но смысл остался прежним. Теперь между этими фазами совсем не обязательно стоит чёткая граница (работа по ним даже может идти параллельно), тестирование программ перенесено из последнего этапа на все предыдущие, и так далее. Помня об этом, мы всё же будем следовать оригинальным названиям этапов, благо их можно встретить и в позднейших источниках.

Итак, сначала формулируются требования к будущей программе. Так как это самое начало работы, требования обычно описаны неформальным, человеческим языком, допускающим толкования и неточности. Главная цель следующего этапа состоит в том, чтобы эти требования формализовать и создать *спецификацию* системы. Например, если мы хотим создать систему, дублирующую возможности телефонной службы через Интернет, для неформального описания этого достаточно. «*Система должна предоставлять возможность телефонных разговоров через Интернет*». Формальная же спецификация должна быть более точна. Например, там должны содержаться недвусмысленные ответы на следующие вопросы:

- Должен ли каждый пользователь сервиса иметь отдельный номер, аналогичный номеру телефона, или для коммуникации достаточно знать его IP-номер?
- Возможны ли звонки с обычного телефона на узел новой службы?
- Возможны ли звонки на обычный телефон с узла новой службы?
- Каким образом осуществляется кодирование передаваемого сигнала?
- Каким образом реализована безопасность соединения?

- На каком ярусе стека протоколов находится тот, которым осуществляется передача информации?
- В каком объёме допустимы потери передаваемой информации?
- Предоставляются ли вторичные услуги телефонной сети (автоответчик, автодозвон, месячный отчёт и пр.)?
- ...

Таких вопросов можно придумать сотни, тысячи. Поэтому спецификация обычно пишется в таком стиле: система делает то-то, тогда пользователь может сделать это или то. Всё, что не описано в спецификации, не будет являться частью проекта системы. С другой стороны, лишь немногое отделяет подробную спецификацию от этого проекта. На этапе анализа может быть осуществлена и формальная верификация. Спецификация также является системой на очень высоком уровне абстракции, поэтому может быть смоделирована и проверена [19]. Способность формальной верификации находить ошибки в требованиях, предъявляемых к системе, была одним из первых замеченных достоинств, обеспечившим рост популярности этой области.

Наиболее эффективно работает верификация на этапе разработки. На этом этапе уже создаются UML-диаграммы или подобные им (по уровню, но не обязательно по типу) аппараты, которые могут быть использованы далее, на этапе реализации. На сегодняшний день используется три пути. Первый путь состоит в том, что над верификацией работает отдельный коллектив специалистов. Он избран, например, в [40], где проделан формальный анализ программы управления полётом космического аппарата. Следуя этому пути, проектировщики отдают свои труды (или их куски) верификаторам, которые делают свою работу и сообщают назад о найденных ошибках. Второй путь состоит в том, что сама разработка заключается в разработке модели, по которой потом *автоматически создаётся* текст программы. Он избран в [15], статье, выпущенной в прошлом году, и пока что такой подход не близок к реальности в большинстве приложений ввиду сложности автоматического создания *эффективной* реализации абстрактной модели. Тем не менее, если эта область будет развиваться теми же темпами, это станет возможным уже ближайшие годы. В частности, уже сейчас кажется возможным создавать UML-диаграммы по моделям [36]. Третий путь состоит в том, что специалисты по верификации пропускают этот этап и ждут, когда будет закончен текст программы, и после этого по коду *автоматически*

создаётся текст модели. Этот подход успешно развивается в [14] и серии статей Джеймса Корбета и Джона Этклиффа [25, 39, и т. д.].

Раз уж зашла речь об этапе реализации, очень близко к последнему подходу лежит область формального тестирования [29]. При этом из спецификации (при тестировании системы типа чёрный ящик) или кода (при тестировании системы типа прозрачный ящик) создаются последовательности вводов в программу и последовательности соответствующих им выводов, позволяющие в совокупности достичь высокого шанса обнаружения ошибок.

Несколько более подробное отступление в сторону тестирования. Классическое тестирование заключается в том, что система сдаётся в условную эксплуатацию (как правило, в узких кругах доверенных пользователей), что приводит к наблюдению её поведения в наиболее вероятных опасных ситуациях. Общее количество ситуаций, которые могут возникнуть при работе системы, конечно, либо бесконечно, либо невероятно велико. Однако, классическое тестирование строится на том, что поведение доверенных пользователей (α— и β—тестеров) не будет слишком уж отличаться от поведения настоящих. Формальное же тестирование использует математический аппарат для создания тестовых последовательностей (в классическом тестировании генератором подобных последовательностей служат люди). Это, с одной стороны, требует больших усилий (необходимо привести спецификацию в соответствие с аппаратом и выразить её языком математики), но с другой позволяет точно определить вероятность нахождения ошибки и, конечно, максимизировать её (вероятность, а не ошибку). Также формальное тестирование сдвинуто во времени на более ранние стадии разработки программы, что позволяет проводить его с большими временны́ми затратами и сильнее влиять на дальнейшую работу. Согласно многочисленным источникам [10, 12, 15, 29, 49, 54, и мн.др.], сейчас в различных крупных проектах на тестирование тратится от 30% до 75% общих ресурсов проекта. Критическое же отличие верификации от тестирования состоит в том, что последняя имеет дело с реальным объектом (программой), а модельная верификация — с моделью программы.

Также верификацией на этапе реализации можно назвать отладку. Отладка, как правило, выполняется самим программистом, и заключается в тестировании небольших (или даже сверхмалых) кусков кода отдельно от остальных. Существует деление на отладку отдельных методов, затем отладку объектов, затем отладку сторон ассоциации, и лишь затем отладку системы в целом, но на практике отладка чего-то достаточно громоздкого невозможна или близка к тому. Также отладка может использоваться в процессе, обратном проектированию (reverse

engineering) для восстановления алгоритма или даже проекта программы по имеющемуся коду (употребляется широко для перепроектирования программного обеспечения (software re-engineering) и менее широко для взлома защищённых программ и воровства алгоритмов).

Также возможно использовать проверку моделей для автоматического создания тестов по спецификациям [37]. Более подробно тестирование и отладка в данной работе не освещаются.

3.4 Верификация алгоритмов

Алгоритмами, как правило, чаще занимаются в академических кругах, чем в программистских. Однако, такие книги, как *«Искусство программирования»* Дональда Е. Кнута [4, 50], *«Жемчужины программирования»* Джона Бентли [9] и *«Алгоритмы и структуры данных»* Никлауса Вирта [1], равно как и различные книги и статьи Эдсгера В. Дейкстры [2, 27, 31, 32, 33], пользуются широкой и заслуженной популярностью и среди кодописателей.

Вычленение алгоритмов из программ для дальнейшего их анализа, упрощения и оптимизации, является отдельной областью информатики. В данный момент существует множество самых различных алгоритмов от сортировки массива до передачи информации по каналу с потерями. Например, широко известная задача об обедающих философах [33] была сформулирована ещё в 1971 году и тогда же разрешена и доказана[1]. Система реализует в упрощённой форме все параметры так называемого кругового тупика, когда один процесс ожидает действия другого, который ожидает действия третьего, и так далее, а последний ожидает действия первого, и так все и находятся в ожидании. Тупики такого рода невероятно сложны в обнаружении, потому что не гарантируют гибель всей системы, но только отмирание кольца. После опубликования условия задачи было предложено несколько решений. В одном из них, например, философам разрешалось брать только две вилки одновременно. Это решение не очень естественно, и возможно именно поэтому плохо реализуется на практике. В другом решении один философ был левшой: он сначала брал левую вилку, а потом правую, тогда как остальные всё делали наоборот. В третьем создавался новый процесс слуги, который

[1]Пять философов сидят за круглым столом, в центре которого находится неиссякаемый источник спагетти. Между философами лежат пять вилок. Каждый философ может либо думать, положив вилки, либо взять две и начать есть. Система имеет неочевидный тупик, когда каждый философ берёт одну вилку (скажем, правую) и ждёт вторую, которая не может освободиться.

координировал действия философов. В чётвертом от философов требовалась синхронизация с особым сервером, выдающим вилки. Как видно, чем дальше приходится отходить от естественности оригинальной формулировки при изобретении решения, тем более громоздким и неоптимальным является реализация. Вероятно, самое красивое решение добавляет на стол философам энциклопедию, которая в каждый момент времени должна находится в руках у одного из них. Понятное дело, что даже философ не станет есть с книгой в руках. Этот метод реализован в ряде токен-методов (включая метод «горячей картофелины») во многих сетях передачи данных.

Понятно, что, чем выше уровень абстракции, на котором описывается алгоритм, тем более широкую область применения он заслуживает. Также абстрактные модели содержат минимум деталей и потому легко верифицируются. Однако, верификация абстрактной модели даёт и минимум результатов. Вполне возможно, где-то среди отброшенных деталей есть та, которая критическим образом влияет на работу системы.

Например, в третьем томе «Искусства программирования» [4] приводятся алгоритмы сортировки с поиска в массиве отдельно от таковых в файле или в списке. Рассмотрение деталей внутренней реализации позволяет провести более точный анализ оптимальности методов, что и сделано в этой книге. Если же рассматривать задачу в общем виде, то есть сортировку как упорядочивание чего-либо протяженного, как это делается в математике, то достаточно будет ввести отношения частичного порядка. Ясно, что такой подход полезен только при доказательстве общих теорем, но не при программировании[2]. Чем выше уровень детализации алгоритма, тем труднее его верифицировать, тем меньшую академическую ценность он имеет и тем большее влияние оказывает на конкретную задачу.

Также играет важную роль возможность инкапсуляции алгоритмов, то есть их связь с окружающей системой. В случае положительного исхода верификации всё прекрасно: это служит своего рода лишним подтверждением правильности его использования. Но чаще всё-таки верификация находит ошибки, если не критического, то по крайней мере уточняющего плана. И тогда чрезвычайно важна связь необходимых исправлений с изменениями в системе, которые за ними последуют. На-

[2] *Позволим себе заметить, что легко можно представить себе язык программирования сверхвысокого уровня, который позволяет программисту конструкции подобного рода. В частности, язык описания множеств SETL это позволяет и мощные функциональные языки (например, Хаскелл) при некоторых усилиях также могут позволить. Однако в данном случае проблема оптимальности реализации всего лишь перекладывается на транслятор языка.*

пример, в последнее время пытаются развиваться устройства биометрической аутентификации, такие как сканирование отпечатков пальцев, фотографирование радужки глаза, тепловые фотографии лица, определение формы и размера руки и тому подобные. И техника позволяет, и результаты существенно лучше обычной проверки пароля, но — по сравнению с другими близкими областями биометрика развивается очень медленно. Главная причина этого торможения — в том, что, предоставляя те же возможности (пусть и на лучшем уровне), биометрические механизмы чересчур сложны для включения в существующие системы. Таким образом, эта область пока что представлена либо традиционными методами типа дактилоскопии, либо в любом случае уникальными аппаратами сканирования радужки, заменяющими в некоторых аэропортах паспортный контроль.

3.5 Верификация протоколов (интерфейсов)

Под протоколом понимается соглашение между сторонами коммуникации о том, данными какого рода они обмениваются, в каком виде и в какой последовательности. Термин *интерфейс* употребляется примерно в таком же контексте, но редко когда — в отношении общения более чем двух сторон. Мы будем далее употреблять термин *протокол* как более общий.

Чёткая граница проводится между вертикальными и горизонтальными протоколами. Первые используются для связи разных уровней абстракции, когда программа с более высокого уровня использует более низкий уровень для описания выполняемых действий. Типичными примерами вертикальных протоколов могут служить API (интерфейс программирования приложений) операционной системы, библиотеки функций (особенно в словарных или функциональных языках программирования, в противном случае библиотеки должны быть хорошо продуманными и содержать только небольшой набор ортогональных операций) и механизмы наследования в некоторых объектно-ориентированных языках. Горизонтальные протоколы обеспечивают общение сущностей, находящихся на одном уровне абстракции. Например, интернетовский протокол IP является горизонтальным, равно как и протокол веб-сервиса HTTP. Обратим внимание, что с точки зрения протокола TCP протокол IP лежит на более низком уровне абстракции, следовательно, может рассматриваться как вертикальный. В случае стека протоколов, как правило, только верхний используемый рассматривается в качестве горизонтального, хотя это зависит от точки зрения (от постановки задачи).

Вертикальные протоколы моделируются в Промеле, как правило, каналами. По каналу команда передаётся процессу, моделирующему более низкий уровень абстракции, после чего тот может либо выполнить свою работу, либо передать команды ещё ниже. Мощи верификатора хватает для верификации всего стека протоколов ISO/OSI разом.

Методов моделирования горизонтальных протоколов, конечно, гораздо больше, потому что собственно понятие протокола такого рода ничего нового в моделирование не привносит. Протокол неявно оседает в поведении тех процессов, которые его используют. Раздел 4.1 содержит пример верификации протокола безопасности.

3.6 Верификация транзакций

Транзакция — это множество связанных операций, выполняемых над определёнными данными (в рассматриваемом случае — выполняемых в распределённом окружении). Главная идея теории транзакций состоит в том, что все операции, составляющие одну транзакцию, либо успешно завершаются, либо терпят неудачу — все вместе, не по раздельности. Теория транзакций появилась на свет много лет назад и широко употребляется в проектировании систем управления базами данных и сфере обработки он-лайновых запросов.

Одна из основных проблем в распределённой системе состоит в том, что серверы обеспечивают доступ к разделяемым ресурсам, которыми *одновременно* пользуется большое число клиентов. Целый спектр вопросов возникает при изменении какого-либо ресурса, устаревшее содержимое которого могли успеть прочесть некоторые из клиентов, и, более того, предпринять на его основе какие-то шаги, теряющие смысл из-за изменения антецедента. Некоторые из этих вопросов **неразрешимы**, например, связанные с попыткой одновременной записи одного и того же сегмента данных разными пользователями. Базовая идея, на которой построены серверы обработки транзакций, заключается в *последовательной эквивалентности*. Этот термин, изначально введённый в теории параллельных процессов, применим и к распределённым системам, в большинстве своём имеющим параллельную архитектуру. Он означает, что если некоторое число транзакций имеет место параллельно (одновременно), суммарный эффект должен быть таким же, как если они происходили последовательно одна за другой. Это требование не позволяет серверу транзакций допускать такие нежелательные эффекты, как, например, несвоевременное обновление данных.

Рассуждая глобально, к системам обработки транзакций обычно

предъявляют следующие четыре требования:

1. **Атомарность** — операции, составляющие транзакцию, при выполнении свободны от влияния извне (например, другой операции) и могут считаться одной большой операцией. Либо все эти операции совершаются, либо ни одна из них.

2. **Целостность** — транзакция не должна оставлять системы в неопределённом состоянии (например, банковская транзакция при переводе денег со счёта на счёт должна как снять сумму с одного счёта, так и добавить её к другому, или не делать ничего). Обычно это означает, что операции, составляющие транзакцию, должны быть выполнены в соответствии со своей спецификацией[3]. Комбинация требований атомарности и целостности даёт ещё более строгую формулировку: транзакция должна перевести систему из одного дозволенного состояния в другое.

3. **Изоляция** — транзакция не оказывает внешнего влияния и не испытывает его. Промежуточные результаты операций недоступны (точнее, не покидают границ данной транзакции). Говоря нестрого, эти три требования и означают последовательную эквивалентность, описанную выше.

4. **Долговечность** (прочность) — результаты завершенной транзакции не должны кануть в Лету из-за последующего краха системы или по другой безрадостной причине, если транзакция имела место, её результат должен быть так или иначе сохранён. Это может быть достигнуто, например, записью результатов на диск, но такие детали реализации не включаются в требования (как минимум из-за того, что запись на диск — тоже своего рода транзакция (возможно, также распределённая!), к которой все требования должны применяться рекурсивно).

3.7 Верификация свойств, зависящих от времени

Иногда в спецификации встречаются такие свойства, которые нелегко выразить с помощью обычной логики. Например, «ответ должен поступить не позже, чем через 10 секунд после запроса». С помощью вре-

[3]Спецификации также были рассмотрены выше, в разделе 3.3.

менно́й алгебры высказываний, описанной на стр. 25–28, некоторые проблемы такого рода разрешаются, но не все. В частности, при реализации нашего примера с 10 секундами пришлось бы явным образом вставлять в модель счётчик времени. Однако мы ничего не можем указывать верификатору относительно приоритетов выполнения тех или иных строк кода, поэтому возможным выходом будет полная переработка всей системы в пошаговую. Однако дискретное время может быть полезно только в узком классе задач моделирования некоторых электрических цепей. Распределённые системы пошаговыми не являются, и для их верификации нужно нечто более удобное и мощное.

Конечно, верификация свойств, зависящих от времени, также производится на специальных автоматах. Сложность заключается в том, что непрерывное время (в отличие от дискретного) даёт бесконечное, несчётное множество состояний (точнее, мощность уже имеющегося множества состояний умножается на $2^{\aleph_0}$). Для преодоления этого препятствия, которое могло бы стать фатальным для желания верифицировать свойства спецификации, зависящие от времени, используется следующие два шага упрощения.

1. Строятся классы эквивалентности состояний автомата, исходя из *данной модели*. Например, если определённое событие одного процесса системы может произойти между двумя событиями другого процесса, все состояния автомата, в котором оно происходит между ними, попадают в один класс эквивалентности. При этом информация о том, произошло ли оно точно посередине между ними, или почти сразу после первого, или непосредственно перед вторым, теряется.

   Данный шаг уменьшает множество состояний до счётного.

2. Рассматриваются только те классы эквивалентности, которые попадают в сферу интереса *данных формул*, которые должны будут проверяться на данной модели. Например, если в формулах используется максимум число 10, то состояния с бо́льшим временем не имеют для нас ценности.

   Данный шаг уменьшает множество состояний до конечного.

Подобные автоматы, дерево выполнения которых равносильно усечению множества классов эквивалентности состояний исходных автоматов, называются регионными автоматами. Их поддерживает своя, достаточно обширная теория, у них есть свой верификатор UppAal

(http://www.uppaal.com). В практической части этой работы верификация свойств, зависящих от времени, не используется. Данный раздел только отмечает возможность возникновения необходимости их верификации.

3.8 Подведение итогов

Распределённая система — это множество взаимозависимых или взаимосвязанных частей, работающих как единое целое. Наиболее распространённые топологии распределённых систем включают: клиент-сервер, двухъярусную и многоярусную топологии (рис. 3.1).

Распределённые системы могут быть верифицированы с различных точек зрения. Можно моделировать и затем проверять части систем по раздельности (сначала методы, потом модули, потом целые программы) — как это делается при отладке или тестировании. Полезно также вычленить из системы алгоритмы преодоления общих проблем и верифицировать и исследовать их отдельно. Часто верификация касается протоколов, используемых при общении сторон системы между собой. Модели следующей главы также можно считать моделями протоколов. Особняком стоят протоколы, предназначены для выполнения транзакций. Они требуют верификации дополнительных свойств (атомарности, целостности, изоляции, долговечности).

Для любого метода необходима хорошо (в идеальном случае — формально) написанная недвусмысленная спецификация системы.

Верификация свойств, зависящих от времени, вообще говоря, невозможна в системах верификации, которые на неё не рассчитаны. Однако, существуют специализированные системы (такие, как верификатор UppAal), её позволяющие.

Глава 4

Практическое приложение верификации

4.1 Верификация протоколов: задача об обедающих криптографах

Бесследовые протоколы

Современные криптографические технологии достигли впечатляющих высот в достижении одной из поставленных перед ними задач, а именно: сокрытии *содержимого* передаваемых сообщений (чаще всего путём его шифрования). Тем не менее, даже те алгоритмы, которые обеспечивают должную стойкость и не позволяют получить ни единого бита не прошедшему аутентификацию лицу, не скрывают самого факта отправления данного сообщения. Иными словами, анализ потока данных позволяет с лёгкостью установить как отправителя, так и получателя любого перехваченного сообщения. Адреса отправителя и получателя никак не могут быть зашифрованы вместе с текстом сообщения, потому как требуются каждому узлу сети по пути следования, включая маршрутизаторы, и необходимы для того, чтобы сообщение было доставлено. Таким образом, если из некоторого места (от определённого узла сети) идёт большой поток сообщений, и атакующий имеет возможность этот поток засечь, это означает, что данное место достаточно активно, чтобы стать вероятной мишенью [34].

Протоколы, с помощью которых участники коммуникаций могут отправлять и получать сообщения без привлечения внимания, называются *бесследовыми*. Впервые такие протоколы были рассмотрены в 1981 году Давидом Чаумом [22]. Можно выделить два существенных резуль-

тата, полученных с тех пор. Во-первых, было опубликовано [34] решение для сети в форме произвольного графа, в котором после подготовительной фазы можно посылать сообщения практически произвольной длины, используя только $\mathcal{O}(1)$ бит на каждый канал на бит данных. Подготовительная фаза заключается в построении покрывающего дерева, требующем рассылки $\mathcal{O}(g \cdot (kn)^2)$ бит по каждому каналу, где g — размер числа, инициирующего датчик псевдослучайных чисел, n — количество узлов в сети, $k < \lfloor \frac{n}{2} - 1 \rfloor$ — число тех их них, доступ ко внутренним данным которых предположительно имеет атакующий. Во-вторых, была создана [23] стойкая структура, обеспечивающая *безусловную секретность* и *терпимость к ошибкам*. Первое означает, что никакое подмножество участников коммуникации, насчитывающее меньше трети общего их количества, не может никакими средствами ничего раскрыть за пределами этого подмножества. Встроенная терпимость к ошибкам означает, что никакое подмножество участников коммуникации, меньшее трети всех участников, никакими действиями не может помешать остальным честно пользоваться протоколом.

Ситуация слегка усугубляется тем, что в подавляющем большинстве исследований касательно определения свойств протоколов безопасности и их верификации используется так называемая модель Долева—Яо [35]. В рамках этой модели термины «сеть» и «атакующий» могут употребляться как синонимы, потому что одним из основных предположений является следующее: *атакующий имеет полный контроль над сетью, может перехватывать чужие и вырабатывать свои сообщения.* Следует отметить, что это верно только в том случае, когда все каналы связи являются одинаково легко доступными. Но более близкое к жизни (и менее требовательное с математической точки зрения) предположение допускает существование аутентифицированных секретных каналов, то есть такого способа обмена сообщениями, при котором подлинность каждого участника не подлежит сомнению (аутентификация) и при этом передаваемые данные доступны только тому, для кого они предназначались (секретность). Такие каналы очень практичны и легко воплотимы широким спектром средств различной степени устойчивости: от физической передачи сообщения из рук в руки до использования методов асимметричной криптографии.

Таким образом, новая задача обеспечения бесследовости сводится к старой и известной задаче сокрытия содержимого сообщений. Потери в ширине канала при этом, конечно, неизбежны. В зависимости от конкретного протокола они варьируются от половины [34] до трёх четвертей [20].

Задача об обедающих криптографах

Три криптографа обедают в ресторане, и их официант сообщает им, что, согласно договорённости с метрдотелем, счёт был оплачен анонимно. Это могло быть сделано либо одним из криптографов, либо NSA[1]. Криптографы уважают право друг друга на анонимность, но им интересно, платит ли NSA. Они разрешают этот вопрос, следуя простому протоколу:

- каждый криптограф кидает монетку и показывает её соседу так, чтобы третий не мог видеть результат (например, под столом);
- затем каждый криптограф произносит вслух, одинаково ли выпали увиденные им монеты (одна, которую он кинул сам, и другая, показанная ему соседом слева);
- если кто-то из криптографов оплатил счёт, он лжёт и сообщает всем не то, что видел на самом деле;
- если чётное число криптографов утверждает, что монеты упали поразному, это означает, что все ответили честно, следовательно, счёт был оплачен их работодателем;
- если нечётное число криптографов сообщает о разных монетах, значит, один из них солгал (так как такая ситуация невозможна), следовательно, NSA не имеет никакого отношения к оплате счёта.

Ни один криптограф, действуя в одиночку, не может определить, кто из двух других заплатил, не зная монеты, которую один из них показал другому [20, 21].

Задача допускает обобщение как на случай большего числа участников, так и на бóльшую, чем один бит (одна монета), длину ключа. Три — это минимальное число криптографов, позволяющее протоколу сработать, и было выбрано автором [20, 21] исключительно для упрощения доказательства. Для моделирования будет использован случай пяти криптографов (рис. 4.1 — сплошные линии соединяют каждого криптографа с кинутой им монетой, прерывистые — с монетой, показанной ему соседом). В обобщённом случае сеть имеет вид неориентированного графа, в котором каждая вершина отвечает узлу (криптографу, хосту, передающей станции, и так далее), а каждая дуга — ключу, который

[1]NSA (National Security Agency) — Национальное агенство безопасности в США, крупнейший в мире работодатель математиков и криптографов и крупнейший пользователь вычислительной мощи (супер)компьютеров. В [20] предполагается, что все обсуждаемые криптографы работают на NSA, поэтому речь и идёт об оплате их счёта.

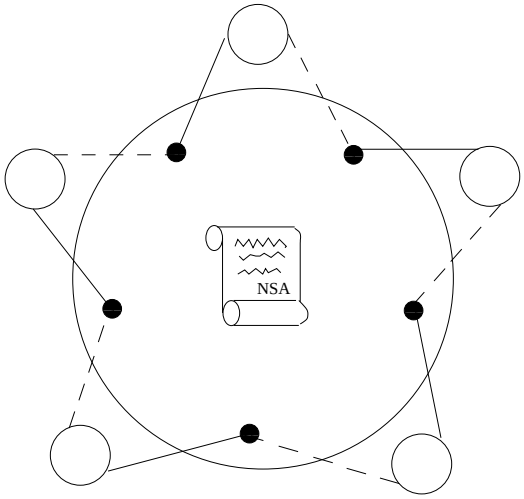


Рис. 4.1: Задача о пяти обедающих криптографах

известен двум инцидентым вершинам. Таким образом, исходная задача может быть представлена в виде треугольника, а задача с пятью криптографами — в виде кольца с пятью узлами.

Этот алгоритм обеспечивает БЕЗУСЛОВНУЮ секретность и бесследовость, то есть даже теоретически невозможно определить заплатившего вне зависимости от потраченного на размышления времени и хитрости используемых алгоритмов (если объём доступной информации остаётся прежним).

Для его моделирования нужно описать систему в терминах языка Промела[2]. В рамках этого языка мы можем оперировать такими терминами, как *параллельные процессы, каналы связи* и пр. Пользуясь нашим описанием, Спин может построить автомат Бюхи и затем произвести его проверку.

В данном случае нам совсем необязательно реализовывать каждого криптографа отдельным процессом. Такой метод моделирования лишь неоправданно увеличит количество состояний системы, ведь на самом деле никакой параллельности не требуется. Более того, в параллельный случай пришлось бы включать части, обеспечивающие синхронизацию.

Действия, производимые системой, должны быть таковы:

1. Решить, кто платит по счёту: NSA или один из пяти криптографов
2. Кинуть пять монет (по одной каждому криптографу)

[2]PROMELA = PROtocol MEta LAnguage — метаязык для протоколов.

3. Показать монеты левым соседям

4. Дать каждому криптографу высказаться относительно равенства увиденных монет (и соврать, если он платит)

5. Определить, заплатило ли NSA

Третий пункт на самом деле не является критичным, потому что, раз мы не проводим чёткой границы между криптографами (не моделируем их как отдельные процессы), то и чёткой границы между локальными переменными (монетами) нет.

Итак, полный текст модели с комментариями выглядит так:

```
bit  coin [5]; /* массив битов-монет */
bool diff [5]; /*массив реплик криптографов о равенстве монет*/
int pay;       /* номер заплатившего или -1 для NSA */
byte i, diffs;

/* основная часть модели */
init
 {
  /* Решить, кто платит - недетерминизм модели */
  if
  :: pay =-1;  /* NSA */
  :: pay = 0;  /* "первый" криптограф    */
  :: pay = 1;  /* "второй" криптограф    */
  :: pay = 2;  /* "третий" криптограф    */
  :: pay = 3;  /* "четвертый" криптограф */
  :: pay = 4;  /* "пятый" криптограф     */
  fi;
  /* Кинуть монеты */
  i=0;
  do  /* недетерминированный цикл с параметром */
  :: i<5  -> coin[i] = 0; i++;
  :: i<5  -> coin[i] = 1; i++;
  :: i==5 -> break;
  od;
  /* Реплика каждого криптографа */
  diff[0] = ((coin[0]==coin[4] && !pay) ||
             (coin[0]!=coin[4] &&  pay));
  if
  :: diff[0] -> printf("C0 говорит: упали по-разному!\n");
```

```
  :: else    -> printf("C0 говорит: упали одинаково!\n");
  fi;
  i=1;
  do
  :: i<5  -> diff[i] = ((coin[i]==coin[i-1] && pay==i)
                      || (coin[i]!=coin[i-1] && pay!=i));
             if
             :: diff[i] ->
                printf("C%d говорит: упали по-разному!\n",i);
             :: else    ->
                printf("C%d говорит: упали одинаково!\n",i);
             fi;
             i++;
  :: i==5 -> break;
  od;
  i=0;
  /* XOR все реплики вместе */
  do
  :: i<5 -> if
            :: diff[i] -> diffs=1-diffs;
            :: else;
            fi;
            i++;
  :: else -> break;
  od;
  /* К этому момент чётность числа реплик о разных монетах */
  /*       должна быть эквивалентна заплатившему NSA       */
  assert((diffs==0)==(pay<0));
}
```

Данная модель содержит 10621 состояний (для сравнения: та же модель в параллельной версии содержит около миллиона!)[3]. Верификатору требуется ровно одна секунда для того, чтобы вынести свой вердикт: система действительно работает. Утверждение в последней строчке модели прервало бы верификацию в противном случае, поэтому мы можем верифицировать модель и без формулирования свойств в PLTL. Свойство, которое верифицируется по умолчанию (и используется в небольших системах чаще прочих) — отсутствие тупиков. (Из каждо-

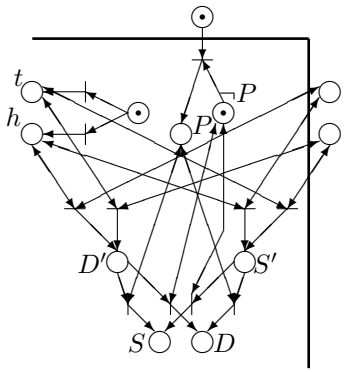
[3]Для ещё более грустного сравнения: вспоминая сказанное на стр. 11, мы попробовали создать модель обедающих криптографов с помощью сетей Петри. Каждый криптограф в ней выглядел так:

го неконцевого состояния автомата Бюхи есть хотя бы один переход в другое допустимое состояние. Состояние с нарушенным утверждением `assert` не является допустимым).

Итак, протокол работает, как и обещал его автор. Теперь можно попробовать изменить его таким образом, чтобы в нём проявились нетривиальные черты.

Атаки

Первой и самой очевидной атакой на протокол является *сговор* нескольких участников. Пользуясь словами автора протокола, *некоторые участники могут объединить свои усилия в попытке уличить других (т.е. отследить их сообщения)* [20]. Чтобы проверить некоторое подмножество участников, необходимо иметь информацию обо всех дугах, входящих в него. На рис. 4.2 проверяемое подмножество обведено, необходимые ключи выделены жирным, участники сговора отмечены стрелками. Нет смысла ограничивать выводы какими бы то ни было предположениями о способе обмена информацией внутри сговора, поэтому анализ ведётся таким образом, как будто сговор формирует некую новую сущность, обладающую всей необходимой информацией. Есть сеть представляет из себя полный граф, в сговор должны будут войти все вершины, кроме входящих в проверяемое подмножество. Но такие случаи чересчур банальны (как, например, сговор всех против одного) и ввиду своей простоты неинтересны. В неполных графах сговор бывает нетривиальным, когда



Всего модель содержит примерно с сотню позиций, с сотню переходов и пять сотен дуг. Она слишком велика для рисования её здесь, однако даже её размер не сравнится с размером того автомата, что сгенерировал для нас Спин. Отдавая дань его неоптимальности, мы не можем не быть благодарными ему за освобождение нас от этого бремени.

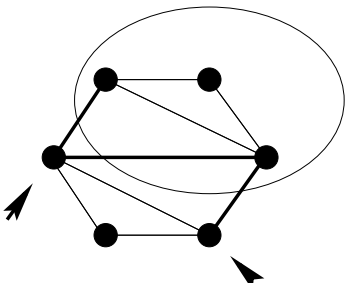


Рис. 4.2: Сговор

содержит разрезающее множество дуг[4]. При этом заплативший (или пославший сообщение, в случае реальной сети) однозначано локализуется в одном из двух полученных подграфов.

Интересный частный случай — нечестный сговор, когда один из его участников снабжает других неверной информацией. (Например, в случае вынужденного сговора всех против всех (скажем, по решению суда) истинный отправитель может попытаться замести следы). К сожалению, эта возможность легко устраняется в помощью механизма цифровых подписей.

Второй тип атаки, рассмотренный в [20] — *срыв* работы системы, например, узлом, безостановочно посылающим новые и новые сообщения. Он относится к хорошо исследованному классу DoS-атак (denial-of-service = отказ обслуживания) и здесь далее не обсуждается.

Третий тип, ранее до нас не рассматривавшийся, описан ниже в том же стиле, в каком было дано условие задачи:

> Один из криптографов слишком любопытен и хочет точно знать, кем был оплачен счёт. Количество сообщений о разных монетах нечётно (то есть платит кто-то из сидящих за столом). Сам любопытный криптограф также не имеет отношения к оплате (иначе его любопытство было бы удовлетворено). Так как с имеющимися у него на руках данными (все устные сообщения и две монеты) он не может достичь своей цели, он пытается подглядеть одну монетку из тех, что не должны быть ему известны.

Случай трёх криптографов очевиден: подсматривая одну монету, любопытный криптограф получает полную информацию о системе,

[4]То есть такое множество, что если каждую дугу из него убрать из графа, тот перестанет быть связным.

следовательно, узнаёт и заплатившего. Четыре хуже пяти своей несимметричностью, и все они лучше шести небольшим размером. Итак, остановимся опять на системе из пяти обедающих криптографов, один из которых — любопытный криптограф.

Изменим модель таким образом, чтобы проверить новый тип атаки. Перед последним утверждением вставим часть, отвечающую любопытству. Сначала любопытный криптограф случайным образом выбирает, какую монету он хочет подсмотреть:

```
if
:: s = 1;       :: s = 2;       :: s = 3;
fi;
```

Далее используется следующее свойство: если любопытный криптограф подсмотрел монету с номером 2 (то есть ту, что лежит на столе напротив него), задача становится эквивалентной двум подзадачам размера 3 (более подробные объяснения даны ниже, на стр. 55). В случае же монет с номером 1 или 3 любопытный криптограф может закончить догадку:

```
if
:: s==3 -> check(4);
:: s==1 -> check(1);
:: else;
fi;
```

где `check()` — это подключённая функция Промелы (см. раздел 2.4):

```
inline check(n)
{
 if
 :: (coin[n-1]==coin[n])&&  diff[n]  -> who = n;
 :: (coin[n-1]!=coin[n])&&(!diff[n]) -> who = n;
 :: else;
 fi;
}
```

Полный текст модели теперь принимает вид:

```
bit  coin [5]; /* массив битов-монет */
bool diff [5]; /*массив реплик криптографов о равенстве монет*/
```

```
int pay;       /* номер заплатившего или -1 для NSA */
byte s, i, diffs;

/* подключенная функция проверки */
inline check(n)
{
 if
 :: (coin[n-1]==coin[n])&&  diff[n]  -> who = n;
 :: (coin[n-1]!=coin[n])&&(!diff[n]) -> who = n;
 :: else;
 fi;
}

/* основная часть модели */
init
 {
  /* Решить, кто платит - недетерминизм модели */
  if
  :: pay =-1;  /* NSA */
  :: pay = 0;  /* "первый" криптограф    */
  :: pay = 1;  /* "второй" криптограф    */
  :: pay = 2;  /* "третий" криптограф    */
  :: pay = 3;  /* "четвертый" криптограф */
  :: pay = 4;  /* "пятый" криптограф     */
  fi;
  /* Кинуть монеты */
  i=0;
  do  /* недетерминированный цикл с параметром */
  :: i<5  -> coin[i] = 0; i++;
  :: i<5  -> coin[i] = 1; i++;
  :: i==5 -> break;
  od;
  /* Реплика каждого криптографа */
  diff[0] = ((coin[0]==coin[4] && !pay) ||
             (coin[0]!=coin[4] &&  pay));
  if
  :: diff[0] -> printf("C0 говорит: упали по-разному!\n");
  :: else    -> printf("C0 говорит: упали одинаково!\n");
  fi;
  i=1;
  do
```

```
  :: i<5  -> diff[i] = ((coin[i]==coin[i-1] && pay==i)
                     || (coin[i]!=coin[i-1] && pay!=i));
             if
             :: diff[i] ->
                printf("C%d говорит: упали по-разному!\n",i);
             :: else    ->
                printf("C%d говорит: упали одинаково!\n",i);
             fi;
             i++;
  :: i==5 -> break;
  od;
  if
  :: s = 1; check(1);
  :: s = 2;
  :: s = 3; check(4);
  fi;
  assert(...);
 }
```

Условие в последней строке может быть различным. Например, `who<1` — при этом мы проверим, действительно ли любопытный криптограф никогда не определит точно заплатившего. В этом случае после половины секунды Спин даёт простейший контр-пример, когда сосед любопытного криптографа платит по счёту и попадает между одной из честных монет и подсмотренной. Это и позволяет ему быть уличённым. Формулу можно придумать более изощрённую: например, `(who<1)||(who==pay)`. Это означает: либо любопытный криптограф не узнаёт заплатившего, либо узнаёт его верно, то есть в любом случае он не ошибается. Такая верификация идёт несколько секунд и даёт положительный результат.

Дальнейшее рассмотрение

Как видно, в случае подсматривания любопытным криптографом монеты, лежащей на столе напротив него, он не получает и не может никак получить информацию о заплатившем. Тем не менее, он может точно определить, где сидит заплативший: справа или слева от него. При этом исходная задача оказывается разрезанной на две подзадачи (рис. 4.3), причём заплативший есть только в одной из них (ведь не разрезали же его, в самом деле, вместе с задачей!).

Теорема. *Чётность количества реплик о неравенстве соседних монет D в системе из N криптографов равна чётности суммы числа*

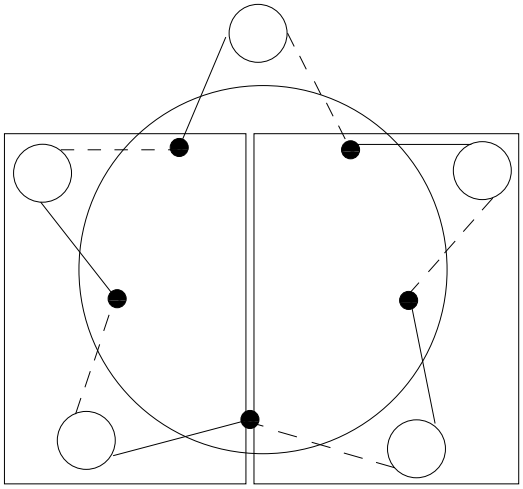


Рис. 4.3: Деление задачи о пяти на две подзадачи о трёх

реплик о неравенстве соседних монет в двух подсистемах, полученных разрезанием исходной.

Доказательство.

1. Построение двух согласованных систем из частей разрезанного графа

   Зная как две честные монеты (которые ему положено знать), так и одну подсмотренную, любопытный криптограф может быть рассмотрен как две различные сущности (или одна двуликая). Каждая часть получает в своё владение одну честную монету и обе — подсмотренную. Таким образом, добавив своё мнение о равенстве правой честной и подсмотренной монет, любопытный криптограф составляет согласованную систему справа от себя. Аналогично он поступает и с левой.

2. Изменение чётности D

   (a) Предположим, у любопытного криптографа две равных честных монеты. Тогда либо подсмотренная монета равна им обеим, либо не равна ни одной. В первом случае в обе части добавится реплика любопытного криптографа о равенстве монет, то есть общее число реплик о неравенстве останется равным D. Во втором случае обе части получат реплику о неравенстве, то есть их общее число возрастёт до $D + 2$. В любом случае чётность очевидным образом сохраняется.

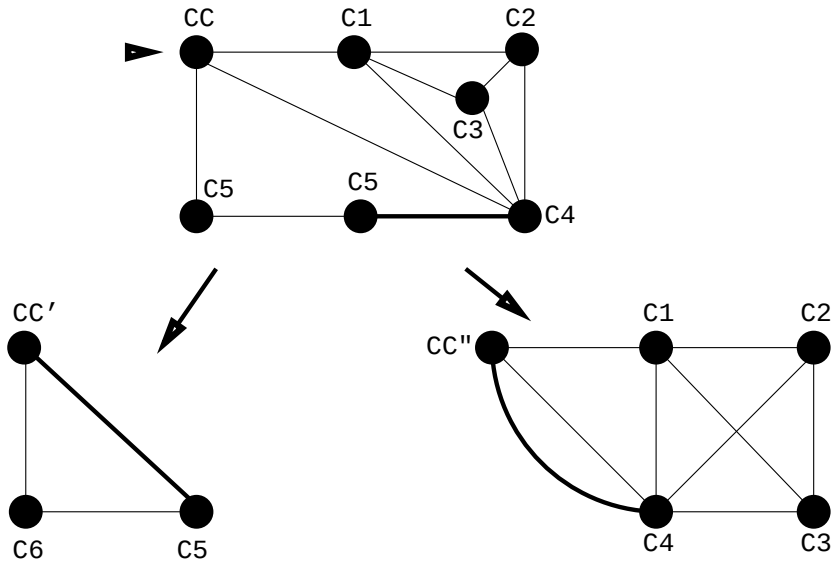


Рис. 4.4: Деление на подзадачи в общем случае

(b) Предположим, у любопытного криптографа две неравных честных монеты, то есть без него в системе было бы $D-1$ реплик о неравенстве. В любом случае подсмотренная монеты может совпасть только с одной из честных. Говоря правду, любопытный криптограф в одну часть подаст реплику за равенство, а в другую — за неравенство видимых монет, что даст общее число реплик о неравенстве $D-1+1=D$, сохранив чётность. ■

В общем случае разрезание производится тем же способом (рис. 4.4), только может потребовать большего количества подсмотренных монет. Очевидно, что чем ближе граф сети к полному, тем больше дуг требуется разорвать для его разрезания. В полном графе с N вершинами нужно знать $N-1$ дуг (рис. 4.5).

Ещё более интересной задачей является нахождение лучшей выигрышной стратегии любопытного криптографа, если считать, что при желании он может подсмотреть любую монету и может принимать решения на основе полученной на очередном шаге информации. Ясно, что наилучшим *с точки зрения минимальности максимального числа шагов* будет метод дихотомии, позволяющий гарантированно достичь успеха за $\lceil \log_2(N-1) \rceil$ шагов[5]. С другой стороны, это же число становится и

[5]Интересно заметить, что эта оценка равна минимальному количеству битов, нужных для хранения числа монет, среди которых делает первый выбор любопытный

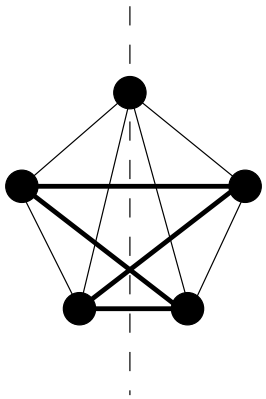


Рис. 4.5: Разрезание полного графа

нижней границей для $N = 2^k + 1$, $k \in \mathbb{N}$. Другой крайностью является последовательный метод, требующий от 1 до $N-2$ шагов. Правильность метода дихотомии не требует доказательства при наличии теоремы со стр. 55.

Разговор об обедающих криптографах можно завершить сравнением двух атак: сговора (рассмотренного автором задачи) и подглядывания (описанного выше). С точки зрения первоначального эстетического описания системы они отличаются количеством вовлечённых сущностей. Пользуясь терминологией теории графов, сговор означает выбор вершины (коих всего $N-1$), подглядывание же означает выбор дуги (коих в кольце $N-2$, а в полном графе $\frac{N(N-3)}{2} - 1$). Кроме того, сговор всегда делается против *потенциального* заплатившего (пославшего сообщение), тогда как подглядывание не столь направлено. Любопытный криптограф действует против *истинного* заплатившего (пославшего сообщение), локализуя его местонахождение.

На практике сговор означает неавторизованное разглашение информации, а подглядывание монет — криптографическую утечку, позволяющую одному участнику коммуникации получать сообщения, предназначенные для другого и расшифровывать их содержимое. Ни та, ни другая атака не ложится в рамки модели Долева—Яо — наиболее распространённой модели в области верификации протоколов безопасности. Этот факт нисколько не мешает их успешному исследованию с помощью проверки моделей.

криптограф.

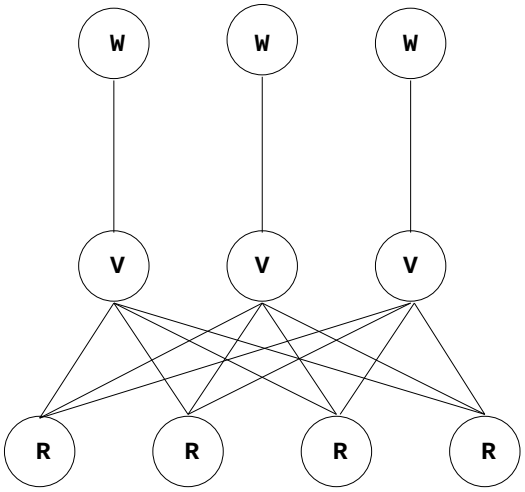


Рис. 4.6: Логическая структура системы доступа

4.2 Верификация алгоритма доступа: распределённое использование ресурсов

Описание задачи

Классическая задача разделённого доступа носит название *readers–writers system*, система пишущих и читающих процессов (см., напр., [24]). Её простейшая формулировка такова: имеется N объектов, к которым может осуществляться доступ, и некоторое количество процессов, которые могут читать или изменять содержимое этих объектов. Существует бесчисленное множество вариаций этой задачи. Мы будем считать, что:

1. К каждому изменяемому объекту прикреплён пишущий процесс.
2. Пишущий процесс может в любой момент изменить содержимое своего объекта.
3. Читающий процесс может в любой момент запросить содержимое любого объекта.

Система схематически изображена на рис. 4.6 (W — Writer, пишущий процесс, V — Value, изменяемый объект, R — Reader, читающий процесс). Не нарушая общности, предположим, что число читающих процессов M больше числа пишущих процессов (оно же число объектов), то есть $N < M$. Этим мы отсечём тривиальные случаи превалирования числа объектов над числом сущностей, ими пользующихся.

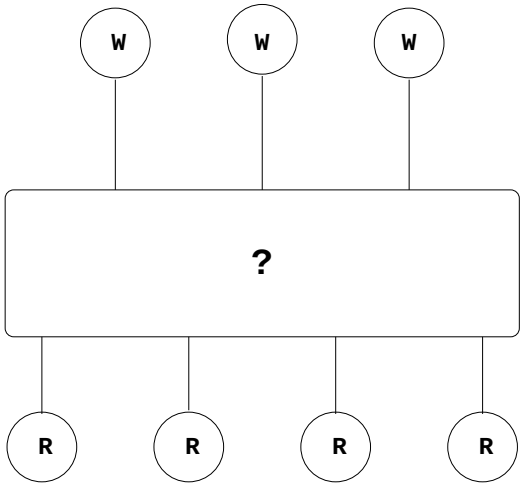


Рис. 4.7: Желаемая структура системы доступа

Предлагаемая нами система имеет вид, изображённый на рис. 4.7. Мы вводим новую сущность, некий координационный сервер, хранящий значения всех изменяемых объектов и предоставляющий читающим процессам возможность получать их содержимое и пишущим — изменять его.

Спецификацию (список требований к поведению системы) можно составить следующим образом:

- Пишущий процесс может в любой момент подать сигнал о желании изменить значение своего объекта и этот сигнал должен быть обслужен.
- Перед входом в секцию, использующую содержимое объекта, читающий процесс заявляет своё желание им пользоваться.
- Пока читающий процесс не отказался от этого желания (то есть пока он не покинул секцию использования), сервер информирует его о каждом изменении объекта.
- Если изменения не происходит, сервер не беспокоит читающий процесс.
- Читающий процесс получает новые значения только о тех объектах, о которых он заявлял своё желание в использовании.

Такая система называется *бессрочным абонементом* или подпиской — читающий процесс подписывается на объект и получает информацию о нём, пока не отпишется.

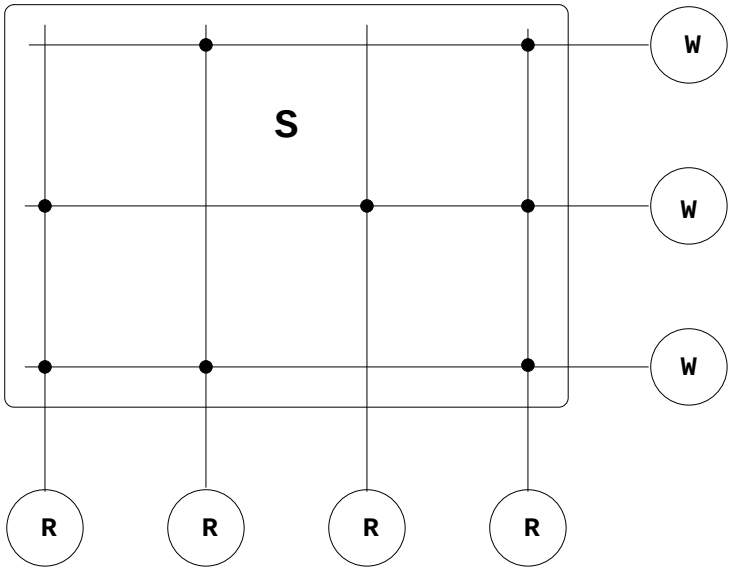


Рис. 4.8: Структура системы доступа с использованием матрицы соединений (матрицы коммуникаций)

Видно, что эта спецификация попадает точно между классами *протоколов* и *транзакций*, сочетая в себе свойства и того, и другого, но служа своей цели, стоящей особняком.

Моделирование

Мы реализуем простейшее соединение читающих процессов с объектами, носящее название *матрицы коммуникаций* (рис. 4.8). Подобные структуры используются в некоторых маршрутизирующих сетевых устройствах (в тех случаях, когда количество соединяемых сторон невелико и требуется быстрая реализация).

Абстрагируясь от некритичных деталей реализации, по данной спецификации мы можем построить следующую модель пишущего процесса:

· **повторять вечно**
  · **изменить значение**

Это абстракция высокого уровня, поэтому мы можем опустить периодичность изменения значения, его вероятность и тому подобные детали, включая и собственно новое значение (для модели критически важно, есть изменение или его нет, но какое именно изменение — безразлично). Модель читающего процесса выглядит несколько более сложно:

- **повторять вечно**
  - **получить обновление используемого объекта, если есть**
  - **и**
    - **либо подписаться на неиспользуемый объект**
    - **либо отписаться от используемого объекта**

Но всё ещё достаточно понятно и просто. Реализация сервера также не представляет собой сверхсложную задачу:

- **повторять вечно**
  - **по команде читающего процесса оформить подписку**
  - **по команде читающего процесса окончить подписку**
  - **по команде пишущего процесса изменить объект**
    - **и разослать его значение всем подписанным**

Реализуем это на Промеле так, чтобы числа N и M можно было легко изменять без необходимости добавления лишних строк в программу. Нам понадобятся два массива каналов: один между сервером и читающими процессами, другой между сервером и пишущими. Каналы, безусловно, должны быть рандеву-каналами, дабы не придумывать лишние проблемы верификатору:

```
mtype = {write, stop, start, update};
chan rs[M] = [0] of {mtype,byte};
chan ws[N] = [0] of {mtype};
```

Пишущий процесс реализуется буквально в три строчки:

```
proctype writer(byte c)
{
 do
 :: ws[c]!write;
 od;
}
```

Читающему процессу можно устроить вечный цикл с параметром, а информацию о подписках хранить в массиве длиной N. Очень важно не заставлять иметь дело с подпиской на каждом шаге (для этого используется всегда разрешённая ветвь `::skip`).

```
proctype reader(byte c)
{
 bool use[N];
 byte i=0;
```

```
 do
 :: i<N -> if
           ::  use[i] -> rs[c]!stop,i;
                         use[i]=false;
           :: !use[i]-> rs[c]!start,i;
                        use[i]=true;
           :: rs[c]?update,i;
           :: skip;
           fi;
           i++;
 :: i>=N -> i=0;
 od;
}
```

Основные усилия построения модели у нас будут сосредоточены на описании спецификации сервера в рамках Промелы. Из соображений простоты и согласно бритве Оккама Промела не содержит многомерных массивов, поэтому наша матрица коммуникаций будет в лучших традициях ассемблера представлена одномерным массивом. Кроме того, получение сообщения по каналу мы должны объединить вместе с производимым действием в один атомарный шаг. В противном случае нашу запись `:: ch?val -> action;` Спин поймёт как `:: skip -> ch?val; action;`, что нам совсем не нужно — так как сервер должен *руководствоваться* посылаемыми ему сообщениями, а не командовать их потоком.

```
bool C[M*N];
proctype server()
{
 byte n,k=0,j;

 do
 :: k<N -> if
           :: atomic{rs[k]?start,n; C[M*n+k]=true;}
           :: atomic{rs[k]?stop,n;  C[M*n+k]=false;}
           :: atomic{ws[k]?write;   broadcast(k);}
           :: else;
           fi;
           k++;
 :: (k>=N)&&(k<M) -> if
                     :: atomic{rs[k]?start,n; C[M*n+k]=true;}
```

```
                    :: atomic{rs[k]?stop,n;  C[M*n+k]=false;}
                    :: else;
                    fi;
                    k++;
 :: else -> k=0;
 od;
}
```

где `broadcast()` — подключенная функция промелы:

```
inline broadcast(ii)
{
 n=0;
 do
 :: n<M -> if
           :: C[M*ii+n] -> rs[n]!update,ii;
           :: else;
           fi;
           n++;
 :: n>=M -> break;
 od;
}
```

Запуск всех процессов можно легко осуществить в следующем цикле (если помнить наше предположение на странице 59 о числах N и M).

```
init
{atomic{
 byte a=0;
 run server();
 do
 :: a<N -> run writer(a); run reader(a); a++;
 :: (a>=N)&&(a<M) -> run reader(a); a++;
 :: else -> break;
 od;
}}
```

Верификация и анализ результатов

Через несколько секунд после начала верификации Спин выдаёт ошибку: система зависла, зашла в тупик. Анализ контр-примера позволяет

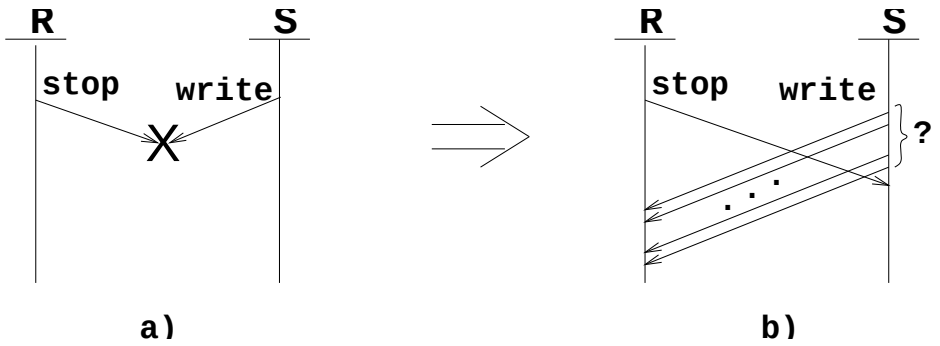


Рис. 4.9: Проблема в системе и невозможность её решения с помощью буфера конечной длины

установить, что в нашей модели возможна ситуация, когда читающий процесс входит в противоречие с сервером: один хочет послать новое значение, другой — отписаться (рис. 4.9 а).

Первый вопрос, который мы должны себе задать: где лежит источник ошибки? В требованиях, сформулированных в самом начале этого раздела; в алгоритме, который мы сделали позже, реализуя, как нам казалось, эти требования; или же только в нашей модели.

Возвращаясь на страницу 60, мы можем легко увидеть, что в спецификацию ситуация столкновения сообщений никак не укладывается. Возможно, проблема в нашей модели? Можно попробовать разделить дуплексный канал между каждым пишущим процессом и сервером на два симплексных. Но это ликвидирует только один случай с рис. 4.9 а, при котором сообщения хотят быть посланным одновременно. Однако в практике они могут отставать друг от друга на шаг. Итак, проблемы это не решает. Можно перейти от рандеву-каналов к асинхронному каналу, надеясь, что буфер поглотит ошибку. Этого также не может произойти по причине, схематически изображённой на рис. 4.9 б: мы не знаем, сколько раз успеет сервер послать изменение, пока не получит остановку подписки. Следовательно, мы не можем ограничить длину буфера никаким конечным числом. Мы можем заставить верификатор работать сколь угодно медленно, но мы не в силах спрятать от него явную ошибку.

Эти и другие методы изменения модели не имеют смысла и не могут быть успешными. Рис. 4.9 б прямо нарушает последнее положение спецификации со стр. 60, потому что читающий процесс получает информацию о том объекте, который ему НЕ НУЖЕН.

Итак, мы должны изменить реализацию, и лишь потом возвратиться к модели. Где её изменять: в читающем процессе, пишущем, или в сервере? Очевидно, что правильным будет являться только последний

вариант. Ни пишущие, ни читающие процессы не виноваты, что сервер плохо координирует их совместную работу. Итак, подробнее и правильнее расписываем действия сервера:

- **повторять вечно**
  - **по команде читающего процесса оформить подписку**
  - **по команде читающего процесса окончить подписку**
  - **по команде пишущего процесса изменить объект**
    - **и для всех подписчиков**
      - **повторять**
        - **по команде оформить новую подписку**
        - **по команде окончить новую подписку**
        - **по возможности послать новое значение**
      - **пока либо значение не отправлено,**
      - **либо соответствующая подписка не окончена**

За изменённую часть в нашей модели отвечает функция `broadcast()`, которая теперь должна принять вид:

```
inline broadcast(ii)
{
 n=0;
 do
 :: n<M -> if
           :: C[M*ii+n] -> do
                           :: rs[n]!update,ii -> break;
                           :: rs[n]?stop,j -> C[M*j+n]=false;
                                              if
                                              :: j==ii->break;
                                              :: else;
                                              fi;
                           :: rs[n]?start,j -> C[M*j+n]=true;
                           od;

           :: else;
           fi;
           n++;
 :: n>=M -> break;
 od;
}
```

Теперь мы можем надеяться на то, что модель заработает. Эта надежда обоснована лишь отчасти: ликвидирована одна ошибка, но никто

не знает их общее число. Пессимистический прогноз полностью оправдывается: повторная верификация находит ещё один контр-пример, на этот раз указывающий на ошибку в нашей модели.

Этот контр-пример связан с атомарностью, которую мы должны были навязать оператору выбора с условиями по получению сообщений из каналов. При этом получилось так, что рассылка нового значения всем подписчикам должна происходить также атомарно. Этого нет ни в спецификации, ни в реализации — рассылка может произойти и квазиатомарно[6], но не обязана. Спецификация допускает случай, когда некоторые процессы не готовы принять сообщение и должны совершить подготовительные шаги. Модель такого случая не допускает. Приоритет безусловно отдаётся спецификации, поэтому модель должна быть изменена.

Это легко делается путём установки флага (которая может быть выполнена атомарно) и последующей его проверки и рассылки значения (которая находится вне оператора ветвления и поэтому не должна быть атомарна). Так как в эту рассылку мы включили и возможность получения запроса на завершение подписки, не следует забывать и об атомарности условий в этом операторе ветвления.

Версия без ошибок

Возможно, это не **версия без ошибок**, а версия с ошибками, которые друг друга компенсируют. Математически более точным было бы название **«версия, полностью удовлетворяющая спецификации»**. Однако тогда пришлось бы заметить, что эта версия утверждает кое-что новое относительно той самой спецификации (например, то, что спецификация не содержит тупиков).

```
#define N   2
#define M   3
#define MxN 6

bool C[MxN];
mtype = {write, stop, start, update};
chan rs[M] = [0] of {mtype,byte};
chan ws[N] = [0] of {mtype};
```

[6] То есть некоторое время все процессы будут заниматься только этим. Квазиатомарная последовательность действий может быть объединена в атомарный шаг, не нарушая общности.

```
inline broadcast(ii)
{
 n=0;
 do
 :: n<M -> if
           :: C[M*ii+n] ->
              do
              :: atomic{rs[n]!update,ii -> break;}
              :: atomic{rs[n]?stop,j    -> C[M*j+n]=false;
                                           if
                                           :: j==ii -> break;
                                           :: else;
                                           fi;}
              :: atomic{rs[n]?start,j -> C[M*j+n]=true;}
              od;
           :: else;
           fi;
           n++;
 :: n>=M -> break;
 od;
}

proctype writer(byte c)
{
 do
 :: ws[c]!write;
 od;
}

proctype reader(byte c)
{
 bool use[N];
 byte i=0;

 do
 :: i<N -> if
           :: use[i] -> rs[c]!stop,i;
                        use[i]=false;
           :: !use[i]-> rs[c]!start,i;
                        use[i]=true;
           :: rs[c]?update,i;
```

```
           :: skip;
           fi;
           i++;
 :: i>=N -> i=0;
 od;
}

proctype server()
{
 byte n,k=0,j;
 bool flag=false,fk;

 do
 :: flag -> broadcast(fk);
            flag=false;
 :: (!flag)&&(k<N) -> if
                      :: atomic{rs[k]?start,n; C[M*n+k]=true;}
                      :: atomic{rs[k]?stop,n;  C[M*n+k]=false;}
                      :: atomic{ws[k]?write;   fk=k;flag=true;}
                      :: else;
                      fi;
                      k++;
 :: (!flag)&&(k>=N)&&(k<M) ->
    if
    :: atomic{rs[k]?start,n; C[M*n+k]=true;}
    :: atomic{rs[k]?stop,n;  C[M*n+k]=false;}
    :: else;
    fi;
    k++;
 :: else -> k=0;
 od;
}

init
{atomic{
 byte a=0;
 run server();
 do
 :: a<N -> run writer(a); run reader(a); a++;
 :: (a>=N)&&(a<M) -> run reader(a); a++;
 :: else -> break;
```

```
 od;
}}
```

Результат верификации

```
zaytsev@ewi144> time ./rw2x3 -m10000000 -w20 -A -c1
Depth=  632521 States=   1e+06 Transitions= 1.91393e+06 Memory= 274.916
Depth=  988998 States=   2e+06 Transitions= 4.20857e+06 Memory= 304.101
(Spin Version 3.5.3 -- 8 December 2002)
        + Partial Order Reduction
        + Compression

Full statespace search for:
        never-claim             - (not selected)
        assertion violations    - (disabled by -A flag)
        cycle checks            - (disabled by -DSAFETY)
        invalid endstates       +

State-vector 84 byte, depth reached 988998, errors: 0
2.38772e+06 states, stored
3.49142e+06 states, matched
5.87914e+06 transitions (= stored+matched)
   19103 atomic steps
hash conflicts: 2.83907e+06 (resolved)
(max size 2^20 states)

Stats on memory usage (in Megabytes):
229.221 equivalent memory usage for states (stored*(State-vector + overhead))
70.860  actual memory usage for states (compression: 30.91%)
        State-vector as stored = 18 byte + 12 byte overhead
4.194   memory used for hash-table (-w20)
240.000 memory used for DFS stack (-m10000000)
314.956 total actual memory usage

nr of templates: [ globals procs chans ]
collapse counts: [ 65 2 120 162 2 ]
unreached in proctype writer
        line 37, state 5, "-end-"
        (1 of 5 states)
unreached in proctype reader
        line 56, state 18, "-end-"
        (1 of 18 states)
unreached in proctype server
        line 82, state 64, "-end-"
        (1 of 64 states)
unreached in proctype :init:
        (0 of 15 states)
33.400u 0.700s 0:34.10 100.0%   0+0k 0+0io 117pf+0w
```

Здесь, кроме очевидно интересных нам чисел (использованная память: 315 Мб, время работы: 34 с, и т. д.), можно почерпнуть и другие данные. Модель пишущего процесса может находиться только в пяти состояниях, читающего — в 18, сервера — в 64, часть из которых атомарные. Числа небольшие, совсем не впечатляющие. Но стоит запустить эти процессы параллельно, как числа мгновенно увеличиваются. Всего состояний у системы оказалось 2387720, а последовательности выполнения команд имели длину до 988998.

Проверка других свойств из спецификации

Из спецификации (стр. 60) мы можем выделить ещё два момента, которые хотелось бы проверить в нашей модели:

1. Пишущий процесс может в любой момент подать сигнал о желании изменить значение своего объекта и этот сигнал должен быть обслужен.

2. Читающий процесс получает новые значения только о тех объектах, о которых он заявлял своё желание в использовании.

Пункт 1 мы проверить не можем. С одной стороны, наша модель построена таким образом, что сигнал от пишущего процесса всегда получаем сервером (так уж работают каналы Промелы). С другой стороны, если мы, например, изменим пишущий процесс следующим образом:

```
proctype writer(byte c)
{
 written[c]=false;
 do
 :: ws[c]!write;
    written[c]=true;
 od;
}
```

и попытаемся проверить PLTL формулу «в любом случае когда-нибудь p», где $p = written_0 \wedge written_1$ (то есть `[]<>p` в нотации Спина), то получим ответ, который видно на рис. 4.10. Он означает, что найден цикл, по которому ходит система, не допуская при этом пишущий процесс до нужного нам места. Иными словами, вся система может работать и без пишущего процесса. Но это мы и знали и без верификатора, это мы сами же и написали в модели. Итак, на данном этапе мы не может проверить

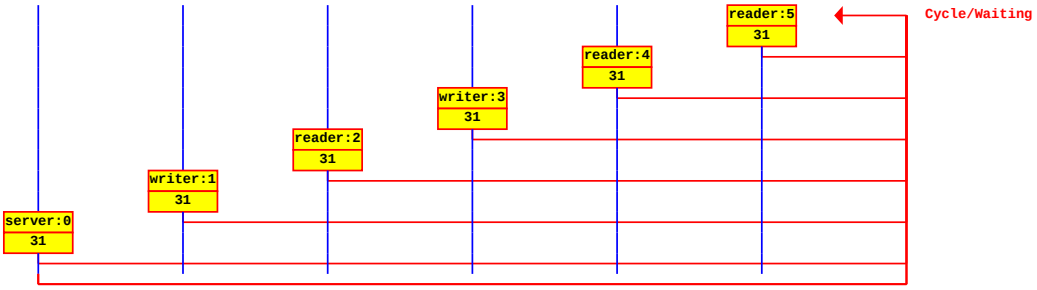


Рис. 4.10: Цикл в системе

этот пункт спецификации, так как явно внесли его в модель. На практике такой пункт считался бы удовлетворённым.

Второй пункт мы проверить можем. Для этого достаточно вставить `assert(use[i])` после `rs[c]?update,i` в читающем процессе. За 31.82 секунды Спин нам может сообщить, что это утверждение никогда не нарушается. Можно ли это сказать самому, глядя на модель? Нет. (То есть, можно, но для этого нет оснований). Кстати, проверить тот же факт с помощью PLTL-формулы сложнее. Мы можем оттуда видеть все переменные и динамику выполнения всех процессов, но мы не можем использовать *значения* переменных. Поэтому формула выглядела бы так:

$$\mathsf{G}\left(P@label \to \bigvee_{j=0}^{N-1}((i=j)\wedge(use_j))\right)$$

или, в нотации Спина (для $N=2$),

```
[](P@label -> ((i=0)&&(use[0]))||((i=1)&&(use[1])) )
```

Подведём итоги данному разделу.

Спин позволяет проводить моделирование и верификацию систем, состоящих из большого числа разнородных объектов. Верификация помогает находить невидимые ошибки и даёт контр-пример, изучение которого позволяет их устранить. Задача проверки соответствия модели своей спецификации нетривиальна и должна проводиться адекватными методами. Эти методы не обязаны ложиться в заранее заданный класс и даже могут решаться помимо моделирования.

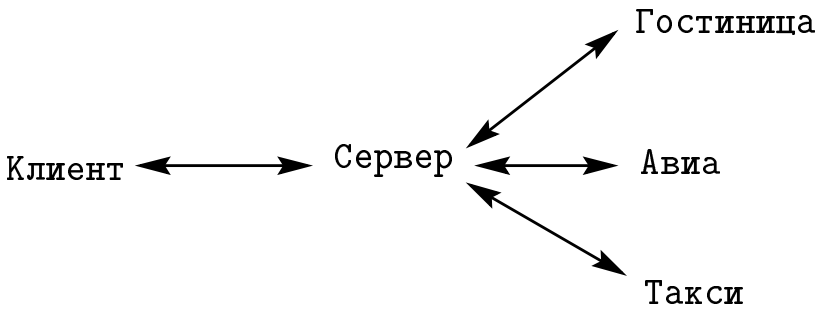


Рис. 4.11: Туристическое агентство как система обработки транзакций

4.3 Верификация транзакций: туристическое агентство

Описание задачи

Рассмотрим распределённую систему, состоящую из пяти частей (рис. 4.11). Первая часть — это сервер, предоставляющий услуги, обычные для туристического агентства (бронирование билетов на самолёт, резервирование гостиницы и вызов такси из аэропорта в гостиницу). Вторая часть — это собственно гостиница, которая предоставляет услуги по резервированию мест и может в том числе и отказать в запросе, если на требуемые даты мест нет. Третья часть — это агентство бронирования авиабилетов, которое также может отказать по причине отсутствия билетов в нужный день. Четвёртой частью будет служба вызова такси, которая никогда не отказывает своим клиентам (тем более сообщающим свою волю заранее). Пятой и самой важной частью будет сам клиент, общающийся с сервером туристического агентства, например, через сайт последнего.

Сначала определим, какие шаги могут происходить в нашей системе, какие сообщения могут посылаться между участниками ассоциации (см. тж. рис. 4.12–4.22).

1. *Осуществление заказа* — клиент обращается к серверу с заказом.
2. *Первый запрос в гостиницу* — сервер сначала оформляет запрос в наиболее важную службу.
3. *Изменение дат из-за гостиницы* — если в гостинице нет свободных мест, сервер обговаривает с клиентом другие даты.

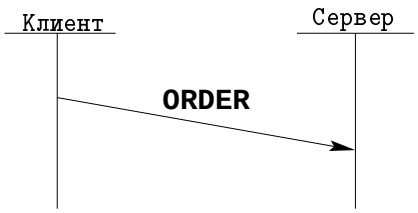


Рис. 4.12: Шаг 1. Осуществление заказа

4. *Повторный запрос в гостиницу* — отличается от первого запроса лишь тем, что теперь сам факт запроса не является неожиданностью для гостиницы, он ожидаем.

5. *Бронь авиабилетов* — установив даты с гостиницей, но не дав последнего подтверждения (то есть они всё ещё могут быть изменены), сервер обращается в авиаагентство.

6. *Изменение дат из-за авиабилетов* — авиабилеты также могут оказаться уже распроданными, требуется вмешательство клиента и изменение договора с гостиницей.

7. *Повторный запрос в авиаагентство* — если то, что устраивало авиаагентство, не устраивает гостиницу (или клиента), оформляется повторный запрос.

8. *Заказ такси* — когда клиент, авиаагентство и гостиница договорились о датах (при посредстве сервера), последний может заказывать такси.

9. *Подтверждение от клиента* — теперь слово только за клиентом, все остальные готовы к сделке.

10. *Завершение сделки* — если после всех изменений клиент всё ещё доволен результатом, сделка может состояться. При этом билеты и места окончательно бронируются, отказ не предусмотрен.

11. *Отказ от сделки* — если клиент не удовлетворён, он может отказаться и уйти, при этом сервер должен отменить все его заказы.

Сообщения в рамках протокола, изображенные на диаграммах, разъяснены ниже. Следующие сообщения используются для обмена данными между клиентом и сервером:

- ORDER — клиент посылает заказ

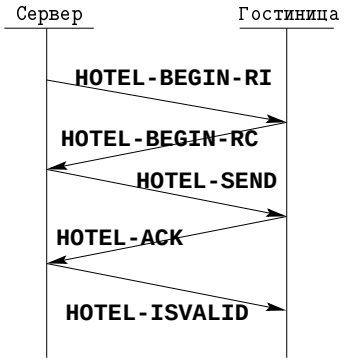


Рис. 4.13: Шаг 2. Первый запрос в гостиницу

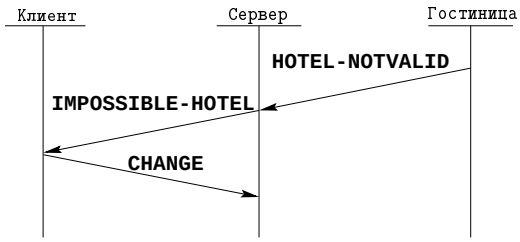


Рис. 4.14: Шаг 3. Изменение дат из-за гостиницы

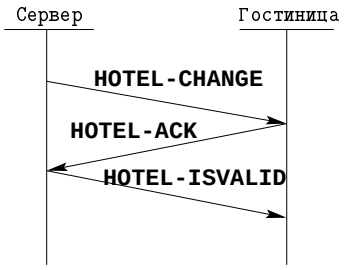


Рис. 4.15: Шаг 4. Повторный запрос в гостиницу

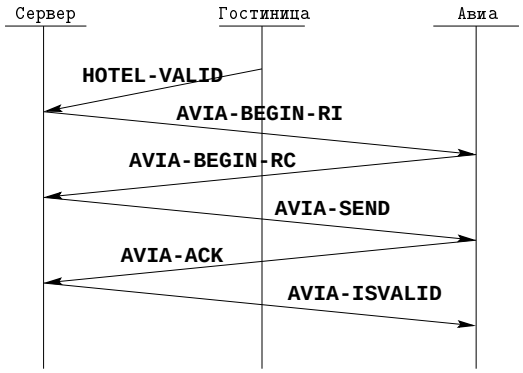


Рис. 4.16: Шаг 5. Бронь авиабилетов

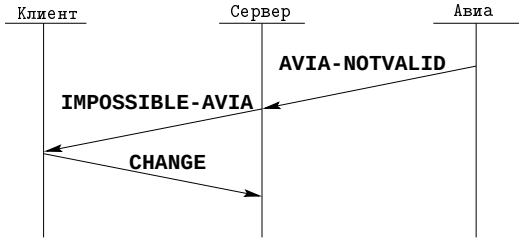


Рис. 4.17: Шаг 6. Изменение дат из-за авиабилетов

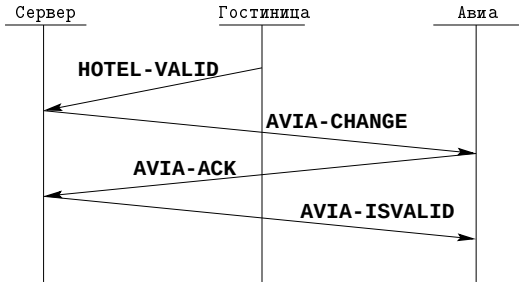


Рис. 4.18: Шаг 7. Повторный запрос в авиаагентство

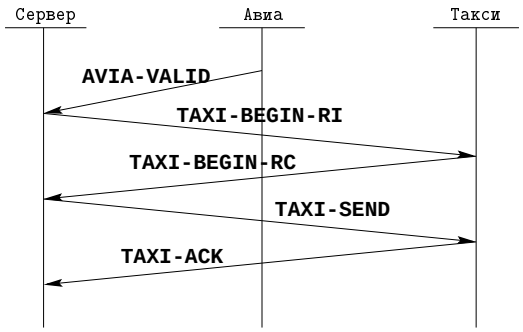


Рис. 4.19: Шаг 8. Заказ такси

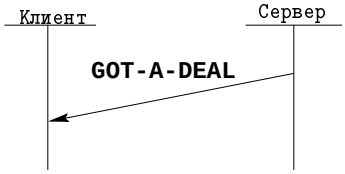


Рис. 4.20: Шаг 9. Подтверждение от клиента

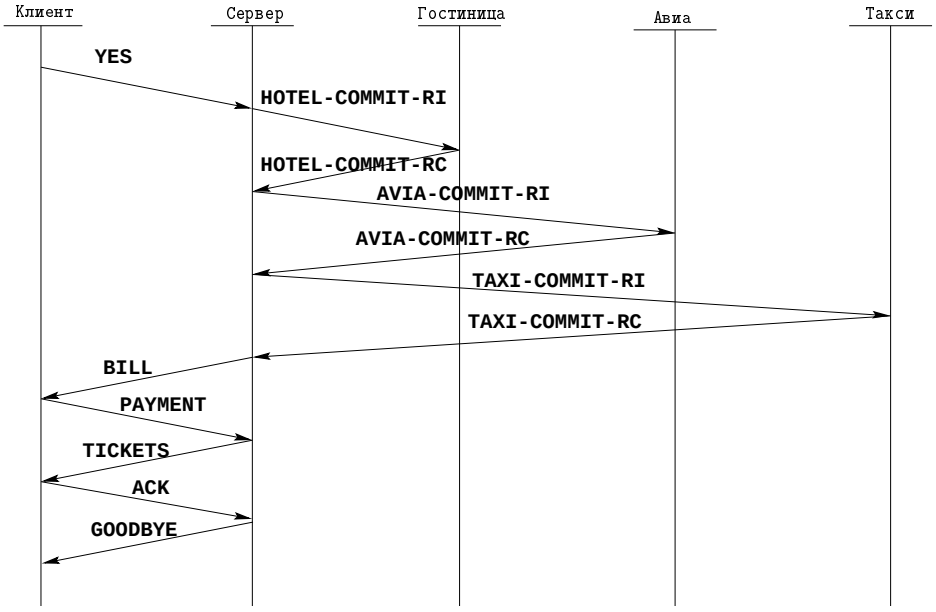


Рис. 4.21: Шаг 10. Завершение сделки

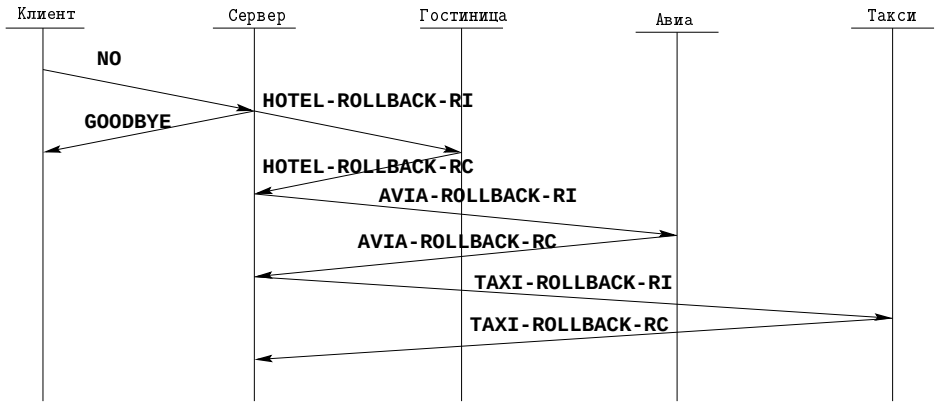


Рис. 4.22: Шаг 11. Отказ от сделки

- Impossible-Hotel — невозможность из-за гостиницы
- Impossible-Avia — невозможность из-за авиабилетов
- Change — клиент изменяет даты
- Got-a-Deal — запрос на завершение сделки клиенту
- Yes — клиент согласен на сделку
- No — клиент не согласен на сделку
- Bill — клиент получает чек
- Payment — клиент оплачивает услуги
- Tickets — клиент получает билеты
- Ack — клиент подтверждает получение
- Goodbye — завершение сеанса с клиентом

Следующие сообщения используются для обмена данными между серверами:

- Hotel-Begin-RI, Avia-Begin-RI, Taxi-Begin-RI — начало нового сеанса связи с гостиницей, авиаагентством и службой заказа такси. RI — от request/indication — запрос/индикация.

- Hotel-Begin-RC, Avia-Begin-RC, Taxi-Begin-RC — подтверждение начала нового сеанса связи с гостиницей, авиаагентством и службой заказа такси. RC — от response/confirmation — ответ/подтверждение.

- Hotel-Send, Avia-Send, Taxi-Send — первое посылание данных в гостиницу, авиаагентство и службу заказа такси.

- Hotel-Ack, Avia-Ack, Taxi-Ack — подтверждение полученных данных от гостиницы, авиаагентства и службы заказа такси.

- Hotel-Change, Avia-Change — повторное посылание данных (изменение дат) в гостиницу и авиаагентство.

- Hotel-IsValid, Avia-IsValid — запрос на возможность утвердить переданные данные (даты) в гостинице и авиаагентстве.

- Hotel-Valid, Avia-Valid — ответ из гостиницы и авиаагентства о возможности утвердить последние переданные данные (даты).

- Hotel-NotValid, Avia-NotValid —ответ из гостиницы и авиаагентства о невозможности утвердить последние переданные данные (даты).

- Hotel-Commit-RI, Avia-Commit-RI, Taxi-Commit-RI — запрос на полное утверждение последних переданных данных (дат) в гостиницу, авиаагентство и службу заказа такси. Транзакция состоялась.

- Hotel-Commit-RC, Avia-Commit-RC, Taxi-Commit-RC — полное утверждение (ответ на запрос) последних переданных данных (дат) в гостинице, авиаагентстве и службе заказа такси. Транзакция завершена положительно.

- Hotel-Rollback-RI, Avia-Rollback-RI, Taxi-Rollback-RI — запрос на полный откат, все переданные до этого момента данные могут быть удалены в гостинице, авиаагентстве и службе заказа такси. Транзакция не состоялась.

- Hotel-Rollback-RC, Avia-Rollback-RC, Taxi-Rollback-RC — подтверждение полного отката, все переданные до этого момента данные были удалены в гостинице, авиаагентстве и службе заказа такси. Транзакция завершена отрицательно.

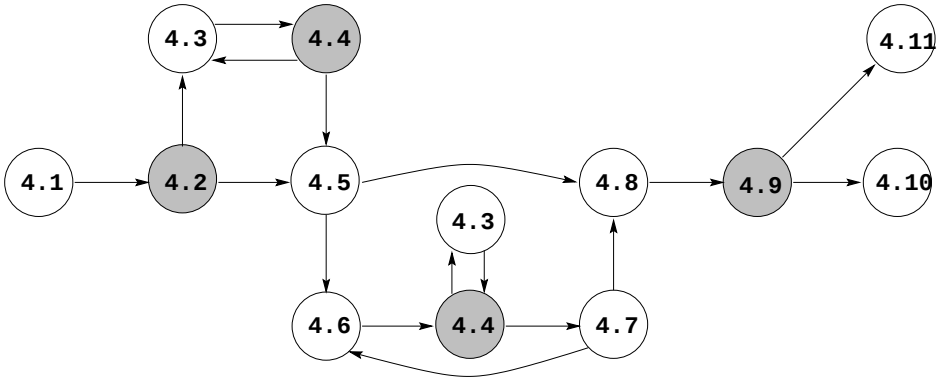


Рис. 4.23: Поведение системы с точки зрения сервера

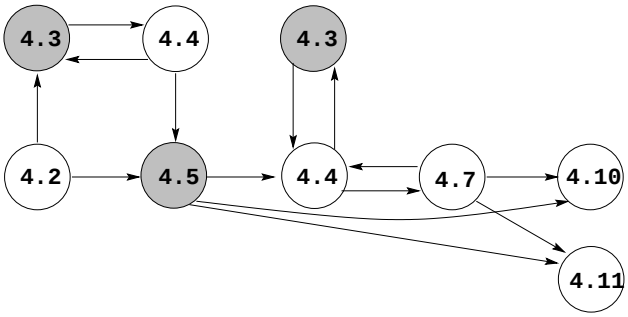


Рис. 4.24: Поведение системы с точки зрения гостиницы

Проектирование модели

Для построения модели мы должны построить общую схему работы системы с различных точек зрения. Будет логично начать с точки зрения сервера, потому как именно сервер координирует работу всех остальных частей. Это означает, что общение всех остальных частей системы идёт только через сервер (рис. 4.11), то есть локальная точка зрения сервера совпадает с глобальной схемой работы системы. Рис. 4.23 содержит схему работы сервера, на которой кружками обозначены шаги с рисунков 4.12–4.22, причем серым цветом выделены те, которые инициируются сервером. Например, начало шага 1 определяется не сервером, а клиентом, а первое сообщение шага 2 должно быть передано со стороны сервера.

Гостиница не участвует в некоторых шагах (а именно: в 1, 6 и 8–9), но также имеет сложную внутреннюю структуру. Возврат к смене дат возможен даже после их подтверждения со стороны гостиницы (из-за других сторон ассоциации). Схема изображена на рис. 4.24.

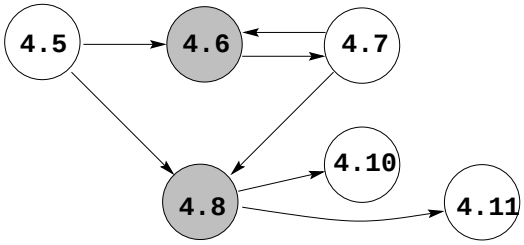


Рис. 4.25: Поведение системы с точки зрения авиаагентства

Рис. 4.26: Поведение системы с точки зрения службы заказа такси

Авиаагентство (рис. 4.25) идёт вторым в очереди после гостиницы, и после него идёт только служба вызова такси (рис. 4.26). Поэтому схемы их работы имеют существенно более упрощённый вид.

С точки зрения клиента система представляет собой нечто:

1. начинающее работу только по его заказу,
2. требующее от него изменения дат по различным внутренним причинам,
3. ждущее его окончательного решения по поводу принятия или непринятия транзакции.

Именно такая схема представлена на рис. 4.27.

Дальше накатанная технология предельно проста: объяснённые выше схемы работы переводятся на Промелу практически один к одному. Мы можем объявить один перечислимый тип для всех сообщений (22 штуки) и четыре канала, им пользующиеся. Нам не будет нужна секция инициализации, потому что вся система состоит из пяти различных

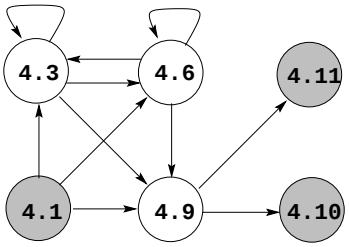


Рис. 4.27: Поведение системы с точки зрения клиента

процессов — мы можем сделать их при описании *активными*. Процессы `hotel()` и `avia()` практически дублируют друг друга, `taxi()` не содержит блока с проверкой дат. В целях краткости и эффективности в процессе `client()` используются не метки (стандарт реализации конечных автоматов), а оператор цикла. Это следует рассматривать именно как небольшую оптимизацию, а не как неоправданную дань структурному программированию. Полный текст спецификации приведён ниже.

Спецификация

```
mtype = {beginri, beginrc, send, ack, change, isvalid, valid,
         notvalid, commitri, commitrc, rollbackri, rollbackrc,
         order, yes, no, goodbye, bill, payment, tickets,
         deal, imphotel, impavia};
chan cs = [0] of {mtype};
chan sh = [0] of {mtype};
chan sa = [0] of {mtype};
chan st = [0] of {mtype};

active proctype avia()
{
 sa?beginri;
 sa!beginrc;
 sa?send;
backavia:
 sa!ack;
 sa?isvalid;
 if
 :: sa!notvalid;
    sa?change;
    goto backavia;
 :: sa!valid;
 fi;
 if
 :: sa?commitri   -> sa!commitrc;
 :: sa?rollbackri -> sa!rollbackrc;
 fi;
}

active proctype hotel()
{
```

```
 sh?beginri;
 sh!beginrc;
 sh?send;
backhotel:
 sh!ack;
 sh?isvalid;
 if
 :: sh!notvalid;
    sh?change;
    goto backhotel;
 :: sh!valid;
 fi;
backhotelb:
 if
 :: sh?change;
   backhotela:
    sh!ack;
    sh?isvalid;
    if
    :: sh!notvalid;
       sh?change;
       goto backhotela;
    :: sh!valid;
    fi;
    goto backhotelb;
 :: sh?rollbackri -> sh!rollbackrc;
 :: sh?commitri   -> sh!commitrc;
 fi;
}

active proctype taxi()
{
 st?beginri;
 st!beginrc;
 st?send;
 st!ack;
 if
 :: st?rollbackri -> st!rollbackrc;
 :: st?commitri   -> st!commitrc;
 fi;
}
```

```
active proctype server()
{
 cs?order;
 sh!beginri;
 sh?beginrc;
 sh!send;
backserv:
 sh?ack;
 sh!isvalid;
 if
 :: sh?notvalid;
    cs!imphotel;
    cs?change;
    sh!change;
    goto backserv;
 :: sh?valid;
 fi;
 sa!beginri;
 sa?beginrc;
 sa!send;
backservb:
 sa?ack;
 sa!isvalid;
 if
 :: sa?notvalid;
    cs!impavia;
    cs?change;
    sh!change;
   backserva:
    sh?ack;
    sh!isvalid;
    if
    :: sh?notvalid;
       cs!imphotel;
       cs?change;
       sh!change;
       goto backserva;
    :: sh?valid;
    fi;
    sa!change;
```

```
    goto backservb;
 :: sa?valid;
 fi;
 st!beginri;
 st?beginrc;
 st!send;
 st?ack;
 cs!deal;
 if
 :: cs?yes;
    sh!commitri;
    sh?commitrc;
    sa!commitri;
    sa?commitrc;
    st!commitri;
    st?commitrc;
    cs!bill;
    cs?payment;
    cs!tickets;
    cs?ack;
    cs!goodbye;
 :: cs?no;
    cs!goodbye;
    sh!rollbackri;
    sh?rollbackrc;
    sa!rollbackri;
    sa?rollbackrc;
    st!rollbackri;
    st?rollbackrc;
 fi;
}

active proctype client()
{
 cs!order;
 do
 :: cs?imphotel;
    cs!change;
 :: cs?impavia;
    cs!change;
 :: cs?deal -> break;
```

```
 od;
 if
 :: cs!yes;
    cs?bill;
    cs!payment;
    cs?tickets;
    cs!ack;
 :: cs!no;
 fi;
 cs?goodbye;
}

init{skip;}
```

Результат верификации

```
zaytsev@ewi144> time ./trans -m100000 -w19 -A -c1
(Spin Version 3.5.3 -- 8 December 2002)
        + Partial Order Reduction

Full statespace search for:
        never-claim             - (not selected)
        assertion violations    - (disabled by -A flag)
        cycle checks            - (disabled by -DSAFETY)
        invalid endstates       +

State-vector 52 byte, depth reached 67, errors: 0
      59 states, stored
       4 states, matched
      63 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^19 states)

2.542   memory usage (Mbyte)

unreached in proctype avia
        (0 of 18 states)
unreached in proctype hotel
        (0 of 28 states)
unreached in proctype taxi
        (0 of 11 states)
unreached in proctype server
        (0 of 66 states)
unreached in proctype client
        (0 of 20 states)
```

```
unreached in proctype :init:
        (0 of 2 states)
0.04u 0.05s 0:00.26 34.0%   0+0k 0+0io (146j+590n)pf+0w
```

Видно, что данная модель явно не выделяется размером. Длина вектора состояний в 52 байта вполне приемлем, не говоря уже о общем размере модели в 59 состояний. Можно сравнить наши результаты с [18], например, где создаётся новая программа проверки моделей Piper: в той статье вводится специально заточенный под задачу формализм (на основе π-исчисления и взаимодействующих последовательных процессов Хоара) и приводятся 17 огромных формул вывода, предшествующих собственно началу работы над моделью.

Итоги раздела

Далее можно систему усложнить, предъявить к ней целую серию разнообразных требований и заняться их проверкой. Наиболее интересным кажется подход, в котором каждый канал заменяется на тройку канал-процесс-канал с возможностью потери (или ещё сложнее: дублирования) сообщений, после чего система проверяется на стойкость с различного рода ограничениями по времени. Но мы остановимся на уже достигнутом, потому что *основной целью данного раздела было показать, что даже для очень большой и сложной распределённой системы возможно подобрать адекватный уровень абстракции, на котором она будет легко верифицируема (размер получившейся модели, как видно из предыдущей страницы, очень и очень небольшой). При этом чёткая и скрупулёзная проработка формулировки задачи делает построение модели простым и чисто техническим занятием.*

Глава 5

Заключение

В данной работе было осуществлено теоретическое и практическое приложение методов построения и проверки моделей (model checking) к области проектирования распределённых систем. Обзор всех основных современных методов верификации программного обеспечения был приведён в главе 2, вместе с описанием достоинств модельных подходов и специфических для них проблем.

На основе этой уже существовавшей теории был проведен глубокий анализ архитектур распределённых систем и способов их построения с целью выделения наиболее перспективных для верификации частей. Такие части были выделены и подробно рассмотрены в главе 3:

1. Спецификация: исходные требования, выраженные в формальном виде, могут быть проверены на адекватность истинным требованиям, предъявляемым к системе

2. Подзадачи: начиная с некоторого уровня сложности, задача может быть разбита на несколько связанных подзадач с понижением сложности каждой из них и всей получившейся системы в целом

3. Алгоритмы: для типично возникающих проблем можно сформулировать и верифицировать шаблонные алгоритмы их решения, то же относится и к распределённым алгоритмам

4. Протоколы: то, что используется для поддержки сервиса и связи общающихся сущностей в ассоциации, наиболее уязвимо и должно представлять первоочередную цель валидации

5. Транзакции: будучи только разновидностью протоколов с лежащими в основе алгоритмами обеспечения требуемых свойств, системы

обработки транзакций могут быть выделены ввиду долгой истории развития их теории

Результаты их анализа были использованы на практике при построении и проверке трёх больших систем (см. главу 4):

1. Протокол: бесследовый протокол из задачи об обедающих криптографах [20], изменённый автором с внесением принципа «любопытного криптографа»

2. Алгоритм: управление доступом к разделяемым ресурсам в задаче о пишущих и читающих процессах, в несколько усложнённом варианте

3. Транзакции: система обработки транзакций туристического агенства как параллельная композиция конечных взаимодействующих последовательных процессов

Некоторые результаты работы в означенном направлении докладывались на миниконференции «Распределённые технологии электронного бизнеса» (27 февраля 2003, г. Энсхеде, Нидерланды), где и статья, и доклад заняли первое место, по этой же теме в настоящее время находится в стадии подготовки журнальная публикация.

Итоги работы рекомендуются к практическому применению.

Литература

[1] Н. Вирт. *Алгоритмы и структуры данных:* Пер. с англ. — М.: Мир, **1989**.

[2] Э. В. Дейкстра. *Дисциплина программирования.* — М.: «Мир», **1978**.

[3] Я. М. Ерусалимский. *Дискретная математика: теория, задачи, приложения.* — М.: «Вузовская книга», **1998**.

[4] Д. Е. Кнут. *Искусство программирования.* 3-е изд. — М.: Издательский дом «Вильямс», **2000**.

[5] M. Abadi, M. Tuttle. *A Semantics for a Logic of Authentication.* Материалы десятого симпозиума ACM по принципам распределённых вычислений, стр. 201–216, август **1991**.

[6] L. C. Aiello, F. Massacci. *Verifying Security Protocols as Planning in Logic Programming.* ACM Transactions on Computational Logics, 2(4):542–580, **2001**.

[7] J. Alves-Foss. *The Use of Belief Logics in the Presence of Causal Consistency Attacks.* Материалы национальной конференции по безопасности информационных систем, **1997**.

[8] K. R. Apt, N. Francez, W. P. de Roever. *A Proof System for Communicating Sequential Processes.* ACM Transactions on Programming Languages and Systems, 2(3):359–385, **1980**.

[9] J. L. Bentley. *Writing Efficient Programs.* Prentice-Hall, Englewood Cliffs, NJ, **1982**.

[10] K. Bhargavan, C. A. Gunter, D. Obradovic. *Fault Origin Adjudication.* Материалы третьего семинара по формальным методам в практике, **2000**.

[11] B. Blanchet. *An Efficient Cryptographic Protocol Verifier Based on Prolog Rules.* Материалы 14го семинара IEEE по основам компьютерной безопасности, **2001**.

[12] B. Boehm, V. R. Basili. *Software Defect Reduction Top 10 List.* IEEE Computer, 34(1):135–137, **2001**.

[13] M. Bozzano, G. Delzanno. *Automated Protocol Verification in Linear Logic.* Материалы четвёртой конференции ACM SIGPLAN по принципам и практическому применению декларативного программирования (PPDL4), стр. 38–49, **2002**.

[14] G. Brat, K. Havelund, S. Park, W. Visser. *Java PathFinder — a Second Generation of a Java Model-Checker.* Материалы семинара по успехам верификации, **2000**.

[15] J. J. D. Bull, P. J. A. de Villiers. *Using SPIN to Verify Protocols at the Implementation Level.* Материалы ежегодной конференции исследователей, проводимой Южноафриканским Институтом Информатики и Информационных Технологий (SAICSIT) по созданию возможностей с помощью технологии, стр. 195–204, Порт Элизабет, ЮАР, **2002**.

[16] M. Burrows, M. Abadi, R. Needham. *A Logic of Authentication.* ACM Transactions on Computer Systems, 8(1):18–36, **1990**.

[17] I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, A. Scedrov. *Relating Strands and Multiset Rewriting for Security Protocol Analysis.* Материалы тринадцатого семинара IEEE по основам компьютерной безопасности, стр. 35–51, IEEE Computer Society Press, **2000**. Содержит ошибку. Расширеные и исправленные тезисы доступны через авторов.

[18] S. Chaki, S. K. Rajamani, J. Rehof. *Types as Models: Model Checking Message-Passing Programs.* Материалы конференции ACM SIGPLAN-SIGACT по принципам языков программирования, стр. 45–57, **2002**.

[19] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, J. D. Reese. *Model Checking Large Software Specifications.* IEEE Transactions on Software Engineering, 24(7):498–520, **1998**.

[20] D. Chaum. *The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability.* Journal of Cryptography, 1:65–75, **1988**.

[21] D. Chaum. *Security without Identification: Transaction Systems to Make Big Brother Obsolete.* Communications of the ACM, 28(10):1030–1044, **1985**.

[22] D. Chaum. *Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms.* Communications of the ACM, 24(2):84–88, **1981**.

[23] D. Chaum, C. Crépeau, I. Damgård. *Multiparty Unconditionally Secure Protocols.* Материалы 12го ежегодного симпозиума ACM по теории вычислений, **1988**.

[24] J. C. Corbett. *An Empirical Evaluation of Three Methods for Deadlock Analysis of Ada Tasking Programs.* Материалы международного симпозиума по тестированию и анализу программного обеспечения, стр. 204–215, **1994**.

[25] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, H. Zheng. *Bandera: Extracting Finite-State Models from Java Source Code.* Материалы 22й международной конференции по разработке программного обеспечения, стр. 439–448, **2000**.

[26] R. Corin, S. Etalle. *An Improved Constraint-Based System for the Verification of Security Protocols.* Материалы 12го международного симпозиума по статическому анализу, LNCS 2477, стр. 326–341, Springer-Verlag. **2002**.

[27] O. Dahl, E. W. Dijkstra, C. A. R. Hoare. *Structured Programming.* Academic Press, New York, **1972**.

[28] G. Delzanno, S. Etalle. *Proof Theory, Transformations, and Logic Programming for Debugging Security Protocols.* Материалы 11го международного семинара по синтезу и трансформации логических програм (LOPSTR'01), стр. 76–91. Springer-Verlag, **2002**.

[29] R. De Nicola, M. C. B. Hennessy. *Testing Equivalences for Processes.* Theoretical Computer Science, 34:83–133, **1984**.

[30] G. Denker, J. Meseguer, C. Talcott. *Protocol Specification and Analysis in Maude.* Материалы семинара по формальным методам и протоколам безопасности, **1998**.

[31] E. W. Dijkstra. *A Correctness Proof for Communicating Sequential Processes—a Small Exercise.* EWD-607, Burroughs, Nuenen, **1977**.

[32] E. W. Dijkstra. *Guarded Commands, Nondeterminacy and Formal Derivation of Programs.* Communication of the ACM, 18(8):453–457, **1975**.

[33] E. W. Dijkstra. *Hierarchical Ordering of Sequential Processes.* Acta Informatica, 1:115–138. **1971**.

[34] S. Dolev, R. Ostrovsky. *Xor-Trees for Efficient Anonymous Multicast and Reception.* ACM Transactions on Information and System Security, 3(2):63–84, **2000**.

[35] D. Dolev, A. C. Yao. *On the Security of Public Key Protocols.* IEEE Transactions on Information Theory, 29(2):198–208, **1983**.

[36] W. Emmerich. *Distributed Component Technologies and their Software Engineering Implications.* Материалы 24й международной конференции по разработке программного обеспечения, стр. 537–546, **2002**.

[37] A. Gargantini, C. Heitmeyer. *Using Model Checking to Generate Tests from Requirements Specifications.* Материалы седьмого международного симпозиума ACM по основам разработки программного обеспечения, стр. 146–162, **1999**.

[38] L. Gong, R. Needham, R. Yahalom. *Reasoning about Belief in Cryptographic Protocols.* Материалы симпозиума IEEE по исследованиям в области безопасности и конфиденциальности, стр. 234–248, **1990**.

[39] J. Hatcliff, M. Dwyer. *Using the Bandera Tool Set to Model-check Properties of Concurrent Java Software.* Concurrency Theory, vol.2154 of Lecture Notes in Computer Science, стр. 39–58. Springer-Verlag, **2001**.

[40] K. Havelund, M. Lowry, J. Penix. *Formal Analysis of a Space-Craft Controller Using SPIN.* IEEE Transactions on Software Engineering, 27(8):749–765, **2001**.

[41] C. A. R. Hoare. *An Axiomatic Basis for Computer Programming.* Communications of the ACM, 12:576–583, **1969**.

[42] C. A. R. Hoare. *Communicating Sequential Processes.* Communcations of the ACM, 21(8):666–677, **1978**.

[43] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, **1991**.

[44] G. J. Holzmann. *Design and Validation of Protocols: a tutorial.* Computer Networks and ISDN Systems, 25:981-1017, **1993**.

[45] G. J. Holzmann. *The Model Checker SPIN.* IEEE Transaction on Software Engineering, 23(5):279-295, **1997**.

[46] F. Huch. *Verification of Erlang Programs Using Abstract Interpretation and Model Checking.* Материалы четвёртой международной конференции ACM SIGPLAN по функциональному программированию, стр. 261–272, **1999**.

[47] D. Ince. *Developing Distributed and E-Commerce Applications.* Addison-Wesley, an imprint of Pearson Education, **2002**.

[48] F. Jacquemard, M. Rusinowitch, L. Vigneron. *Compiling and Verifying Security Protocols.* Материалы международной конференции по применению логики в программировании и автоматизированной аргументации (LPAR'00), стр. 131–160, Springer-Verlag, **2000**.

[49] J.-P. Katoen. *System Validation Using Model Checking.* Учебник. http://fmt.cs.utwente.nl/courses/systemvalidation/

[50] D. E. Knuth. *The Art of Computer Programming.* Volumes 1–3, Addison-Wesley, Reading, MA, **1968, 1969, 1973**.

[51] J. S. Ostroff. *Composition and refinement of discrete real-time systems.* ACM Transactions on Software Engineering and Methodology, 8(1):1–48, **1999**.

[52] L. C. Paulson. *The Inductive Approach to Verifying Cryptographic Protocols.* Journal of Computer Security, 6:85–128, **1998**. Через автора доступна изменённая и исправленная версия за ноябрь **2000**.

[53] J. Penix, W. Visser, E. Engstrom, A. Larson, N. Weininger. *Verification of Time Partitioning in the DEOS Scheduler Kernel.* Материалы 22ой международной конференции по разработке программного обеспечения, **2000**.

[54] L. B. S. Raccoon. *Fifty Years of Progress in Software Engineering.* ACM SIGSOFT, Software Engineering Notes, 22(1):88–104, **1997**.

[55] P. Syverson, P. van Oorschot. *On Unifying Some Cryptographic Protocol Logics.* Материалы симпозиума IEEE по исследованиям в области безопасности и конфиденциальности, стр. 14–28, **1994**.

[56] B. Wegbreit. *Verifying Program Performance.* Journal of the ACM, 23(4):691–699, **1976**.