Н. И. Поликарпова, А. А. Шалыто

Автоматное программирование

Книга написана в Санкт-Петербургском государственном университете информационных технологий, механики и оптики – победителе конкурса инновационных образовательных программ вузов Российской Федерации, в рамках программы «Инновационная система подготовки специалистов нового поколения в области информационных и оптических технологий» по приоритетному научно-образовательному направлению «Технологии программирования и производства программного обеспечения».

Санкт-Петербург

2008

УДК 681.3.06

Рецензенты:

Мелехин В. Ф., докт. техн. наук, профессор, заведующий кафедрой автоматики и вычислительной техники Санкт-Петербургского государственного политехнического университета.

Сергеев М. Б., докт. техн. наук, профессор, заведующий кафедрой вычислительных систем и сетей Санкт-Петербургского государственного университета аэрокосмического приборостроения.

Поликарпова Н. И., Шалыто А. А. Автоматное программирование. 2008. — 167 с.: ил.

В книге рассматривается автоматное программирование – подход к разработке программных систем со сложным поведением, основанный на модели автоматизированного объекта управления (расширении конечного автомата). Предлагаемый подход позволяет создавать качественное программное обеспечение для ответственных систем, охватывая все этапы его жизненного цикла и поддерживая его спецификацию, проектирование, реализацию, тестирование, верификацию и документирование.

Книга предназначена для специалистов в области программирования, информатики, вычислительной техники и систем управления, а также аспирантов и студентов, обучающихся по специальностям «Прикладная математика и информатика», «Управление и информатика в технических системах» и «Вычислительные машины, системы, комплексы и сети».

Оглавление

Предисловие 4
Благодарности 5
Глава 1. Введение в автоматное программирование....7
1.1. Области применения автоматного подхода....7
1.2. Основные понятия....11
1.3. Парадигма автоматного программирования....12
1.4. Автоматные модели 17
Глава 2. Процедурное программирование с явным выделением состояний 42
2.1. Проектирование....43
2.2. Спецификация 64
2.3. Реализация 76
Глава 3. Объектно-ориентированное программирование с явным выделением состояний 95
3.1. Проектирование....97
3.2. Спецификация 106
3.3. Реализация 116
Глава 4. Автоматное программирование. Новые задачи 134
4.1. Автоматы и алгоритмы дискретной математики....134
4.2. Проверка правильности автоматных программ 137
4.3. Автоматы и параллельные вычисления 140
4.4. Автоматы и генетическое программирование....141
Заключение 151
Список источников 155
Предметный указатель....166

Предисловие

Предметом этой книги является парадигма *автоматного программирования*. Автоматное программирование, иначе называемое *«программирование от состояний»* или *«программирование с явным выделением состояний»* – это метод разработки программного обеспечения (ПО), основанный на расширенной модели конечных автоматов и ориентированный на создание широкого класса приложений. Вопреки распространенному мнению, здесь речь идет не только и не столько о использовании конечных автоматов в программировании, сколько о методе создания программ в целом, поведение которых описывается автоматами.

Программирование с использованием автоматов имеет достаточно богатую историю развития. Различные аспекты и понятия, связанные с этой идеей, рассматривались в работах многих авторов с самых разных точек зрения и применительно к различным конкретным вопросам. Программирование от состояний является одним из основных стилей программирования [1].

Как целостная парадигма разработки ПО, автоматное программирование сформировалось, в основном, благодаря усилиям одного из авторов этой книги, который в 1991 г. предложил технологию для поддержки этого стиля программирования [2]. В его работах можно найти обсуждение различных аспектов этого метода программирования, а краткое описание предлагаемой парадигмы программирования содержится, например, в работе [3]. Однако полное и исчерпывающее изложение сути автоматного программирования как парадигмы и метода разработки программных систем в целом в настоящий момент отсутствует. Эта книга может считаться первым шагом к восполнению этого пробела.

Термин «автоматное программирование» родился в 1997 г. в ходе беседы одного из авторов этой книги с *Д. А. Поспеловым* на конференции по мультиагентным системам, проходившей в поселке *Ольгино* под *Санкт-Петербургом*, и был впервые использован во введении к работе [4]. На английский язык этот термин переводится как *automata-based programming*. Англоязычное название было предложено в работе [5].

Цель книги состоит в определении терминов, образующих словарь парадигмы автоматного программирования, и систематическом изложении ее основных концепций. Книга содержит ряд примеров конкретного применения автоматного программирования для решения разнообразных прикладных задач. Примеры призваны продемонстрировать действенность, плодотворность и перспективность данной парадигмы.

Книга имеет следующую структуру. В первой главе излагаются основные идеи и понятия, вводятся специфические обозначения, описываются математические основы автоматного программирования. Знакомство с материалом первой главы необходимо для эффективного освоения остального материала.

Во второй главе описывается традиционный взгляд на автоматный подход к разработке программного обеспечения. Изложение охватывает все аспекты создания программной системы: проектирование, спецификацию и реализацию. Значительная часть второй главы посвящена задачам логического управления. Опыт решения этих задач послужил отправной точкой развития автоматного подхода и его

распространения в других областях программирования. Эта глава намеренно названа «Процедурное программирование с явным выделением состояний»: обычно слово «процедурное» в этом словосочетании упускают, поскольку процедурный подход исторически является для автоматного программирования традиционным. Такое название призвано подчеркнуть смысловое отличие от третьей главы, названной «Объектно-ориентированное программирование с явным выделением состояний».

В третьей главе устанавливается связь между объектной и автоматной парадигмами. Материал этой главы призван показать, что автоматное программирование – это естественное, а не принудительное развитие объектно-ориентированного подхода. Центральная концепция автоматного программирования – *автоматизированный объект управления* – является по своей природе глубоко объектно-ориентированной.

Четвертая глава представляет собой краткий обзор нетрадиционных областей применения и актуальных проблем автоматного программирования. Здесь рассматривается использование автоматов для решения классических задач дискретной математики, вопросы верификации и параллелизма в автоматных программах, а также методы совместного применения генетического и автоматного программирования.

Отметим, что термин *«программирование»* в этой книге употребляется в широком смысле. Как процесс он означает то же, что и *«разработка»*, *«создание»* программного обеспечения. Как разновидность (в словосочетаниях «автоматное программирование», «объектно-ориентированное программирование» и т. п.) он является синонимом терминов *«метод»*, *«подход»*, *«парадигма»*. Для обозначения программирования в узком смысле (написания понятного компьютеру кода) в книге используется термин *«реализация»*.

Материал книги используется в учебном курсе «Теория автоматов в программировании» в Санкт-Петербургском университете информационных технологий механики и оптики (СПбГУ ИТМО) на кафедре «Компьютерные технологии», широко известной своими успехами в области олимпиадного программирования [6, 7].

Более пяти лет назад создан сетевой ресурс http://is.ifmo.ru, посвященный автоматному программированию, на котором, в частности, опубликовано более 150 студенческих работ, включающих проектную документацию, которые иллюстрируют различные аспекты автоматного подхода к программированию.

Автоматное программирование используется в настоящее время при проектировании программного обеспечения систем автоматизации ответственных объектов управления [8, 9], а в соответствии со стандартом *IEC 61499*, который предназначен для унификации правил создания распределенных управляющих систем, базовые функциональные блоки предлагается описывать с помощью конечных автоматов.

Благодарности

Авторы признательны *В. Н. Васильеву* и *В. Г. Парфенову* за многолетнюю поддержку и помощь в развитии парадигмы программирования, описанной в этой книге. Авторы благодарны студентам и аспирантам кафедр «Компьютерные технологии» и «Технологии программирования» СПбГУ ИТМО, которые внесли свой вклад в

теорию и практику автоматного программирования, а также всем специалистам, которые внедряют эту парадигму программирования в промышленности.

Глава 1. Введение в автоматное программирование

1.1. Области применения автоматного подхода

В соответствии с классификацией, введенной *Д. Харелом* [10], любую программную систему можно отнести к одному из следующих классов.

- **Трансформирующие системы** осуществляют некоторое преобразование входных данных, и после этого завершают свою работу. В таких системах, как правило, входные данные полностью известны и доступны на момент запуска системы, а выходные – только после завершения ее работы. К трансформирующим системам относятся, например, архиваторы и компиляторы.
- **Интерактивные системы** взаимодействуют с окружающей средой в режиме диалога (например, текстовый редактор). Характерной особенностью таких систем является то, что они могут контролировать скорость взаимодействия с окружающей средой – заставлять среду «ждать».
- **Реактивные системы** взаимодействуют с окружающей средой путем обмена сообщениями в темпе, задаваемом средой. К этому классу можно отнести большинство телекоммуникационных систем, а также системы контроля и управления физическими устройствами.

Читателю, наверняка, известно, что конечные автоматы в программировании традиционно применяются при создании компиляторов [11], которые относятся к классу трансформирующих систем. Автомат здесь понимается как некое вычислительное устройство, имеющее входную и выходную ленты. Перед началом работы на входной ленте записана строка, которую автомат далее посимвольно считывает и обрабатывает. В результате обработки автомат последовательно записывает некоторые символы на выходную ленту.

Другая традиционная область использования автоматов – задачи логического управления [4] – является подклассом реактивных систем. Здесь автомат – это, на первый взгляд, совсем другое устройство. У него несколько параллельных входов (чаще всего двоичных), на которые в режиме реального времени поступают сигналы от окружающей среды. Обрабатывая эти сигналы, автомат формирует значения нескольких параллельных выходов.

Таким образом, даже традиционные области применения конечных автоматов охватывают принципиально различные классы программных систем. Авторы книги убеждены, что в действительности круг задач, при решении которых целесообразно использовать автоматный подход, значительно шире и включает создание программных систем, принадлежащих всем трем перечисленным классам. Однако, автоматные модели, используемые при создании различных видов программных систем, могут отличаться друг от друга. Различия автоматных моделей подробно описаны в разд. 1.4.

По мнению авторов, критерий применимости автоматного подхода лучше всего выражается через понятие *«сложное поведение»*. Неформально можно сказать, что сущность (объект, подсистема) обладает сложным поведением, если в качестве

реакции на некоторое *входное воздействие* она может осуществить одно из нескольких *выходных воздействий*. При этом существенно, что выбор конкретного выходного воздействия может зависеть не только от входного воздействия, но и от предыстории. Для сущностей с *простым поведением* реакция на любое входное воздействие зависит только от этого воздействия[1] (рис. 1.1).

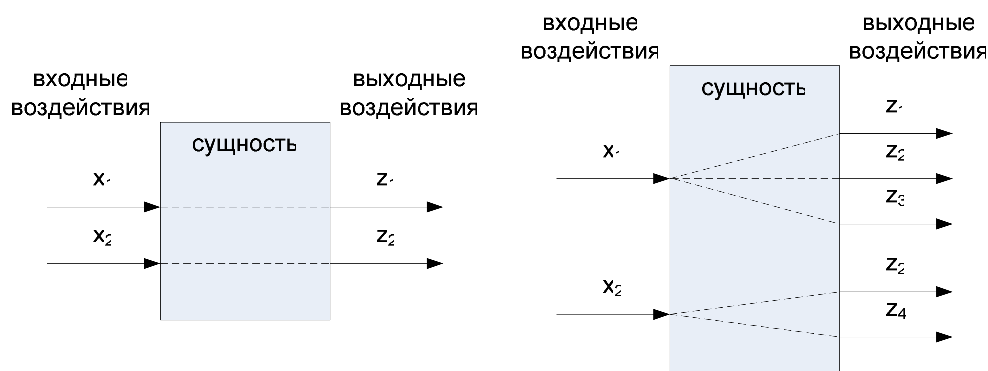

Рис. 1.1. Сущность с простым поведением (слева) и со сложным поведением (справа)

Рассмотрим в качестве примера электронные часы (рис. 1.2). Пусть у них имеется только две кнопки, которые предназначены для установки текущего времени: кнопка «H» (*Hours*) увеличивает на единицу число часов, а кнопка «M» (*Minutes*) – число минут. Увеличение происходит по модулю 24 и 60 соответственно. Такие часы обладают простым поведением, поскольку каждое из двух входных воздействий (нажатие первой или второй кнопки) приводит к единственной, заранее определенной реакции часов.

Рис. 1.2. Электронные часы

Рассмотрим теперь электронные часы с будильником (рис. 1.3). Дополнительная кнопка «A» (*Alarm*) предназначена в них для включения и выключения будильника. Если будильник выключен, то кнопка «A» включает его и переводит часы в режим, в котором кнопки «H» и «M» устанавливают не текущее время, а время срабатывания будильника. Повторное нажатие кнопки «A» возвращает часы в обычный режим.

---

[1] Сложное поведение также называют *поведением, зависящим от состояния* (в англоязычной литературе используется термин *state-dependent behavior*). Соответственно, простое поведение можно назвать *поведением, не зависящим от состояния*. Смысл этих терминов станет более ясным в разд. 1.3, когда будет рассмотрено понятие управляющих состояний.

После этого, если текущее время совпадает со временем будильника, включается звонок, который отключается либо нажатием кнопки «A», либо самопроизвольно через минуту. Наконец, нажатие кнопки «A» в обычном режиме при включенном будильнике приводит к выключению будильника.

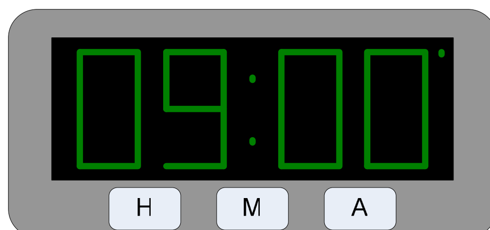

Рис. 1.3. Электронные часы с будильником

Поведение часов с будильником уже является сложным, поскольку одни и те входные воздействия (нажатие одних и тех же кнопок) в зависимости от режима инициируют различные действия.

В программных и программно-аппаратных вычислительных системах сущности со сложным поведением встречаются очень часто. Таким свойством обладают устройства управления, сетевые протоколы, диалоговые окна, персонажи компьютерных игр и многие другие объекты и системы.

Распознать сущность со сложным поведением в исходном коде программы можно следующим образом: при традиционной реализации таких сущностей используются логические переменные, называемые *флагами*, и многочисленные запутанные конструкции ветвления, условиями в которых выступают различные комбинации значений флагов. Такой способ описания логики сложного поведения плохо структурирован, труден для понимания и модификации, подвержен ошибкам. Даже для такого элементарного примера как часы с будильником, реализация при помощи флагов выглядит громоздко и непонятно (листинг 1.1).

Листинг 1.1. Реализация электронных часов с будильником на языке программирования *C++* при помощи флагов

```
class Alarm_Clock {
public:
    void h_button() // Нажатие кнопки H
    {
        if (is_in_alarm_time_mode) {
            alarm_hours = (alarm_hours + 1) % 24;
        } else {
            hours = (hours + 1) % 24;
        }
    }

    void m_button() // Нажатие кнопки M
    {
        if (is_in_alarm_time_mode) {
            alarm_minutes = (alarm_minutes + 1) % 60;
```

```
        } else {
            minutes = (minutes + 1) % 60;
        }
    }

    void a_button() // Нажатие кнопки A
    {
        if (is_alarm_on) {
            if (is_in_alarm_time_mode) {
                is_in_alarm_time_mode = false;
            } else {
                bell_off();
                is_alarm_on = false;
            }
        } else {
            is_alarm_on = true;
            is_in_alarm_time_mode = true;
        }
    }

    void tick() // Срабатывание минутного таймера
    {
        if (is_alarm_on && !is_in_alarm_time_mode) {
            if ((minutes == alarm_minutes - 1) && (hours == alarm_hours) ||
    (alarm_minutes == 0) && (minutes == 59) && (hours == alarm_hours - 1))
                bell_on();
            else if ((minutes == alarm_minutes) && (hours == alarm_hours))
                bell_off();
        }
        minutes = (minutes + 1) % 60;
        if (minutes == 0) hours = (hours + 1) % 24;
    }
private:
    int hours;          // Часы текущего времени
    int minutes;        // Минуты текущего времени
    int alarm_hours;    // Часы срабатывания будильника
    int alarm_minutes;  // Минуты срабатывания будильника

    bool is_alarm_on;   // Включен ли будильник?
    bool is_in_alarm_time_mode;
        // Активен ли режим установки времени будильника?

    void bell_on() {...} // Включить звонок
    void bell_off() {...} // Выключить звонок
};
```

Одна из центральных идей автоматного программирования состоит в отделении описания *логики* поведения (при каких условиях необходимо выполнить те или иные действия) от описания его *семантики* (собственно смысла каждого из действий). Кроме того, описание логики при автоматном подходе жестко структурировано. Эти свойства делают автоматное описание сложного поведения наглядным и ясным.

Основная рекомендация по применению автоматного программирования очень проста: *используйте автоматный подход при создании любой программной системы, в которой есть сущности со сложным поведением*. Опыт показывает, что таким свойством обладает практически любая серьезная система. Однако обычно не все компоненты системы характеризуются сложным поведением. Поэтому данную выше рекомендацию можно дополнить еще одной: *используйте автоматный подход при создании только тех компонентов системы, которые являются сущностями со сложным поведением*. Это дополнение призывает не перегружать спецификации и код описанием логики там, где в этом нет необходимости.

Следование этим простым рекомендациям позволяет создавать корректные и расширяемые программные системы со сложным поведением. Однако сделать это не так просто. Идентификация сущностей со сложным поведением на этапе проектирования системы, а также выделение логики их поведения – сложные творческие задачи. Эти нетривиальные проектные решения принимаются разработчиком для каждой из них индивидуально.

1.2. Основные понятия

Базовым понятием автоматного программирования является *«состояние»*. Это понятие в том смысле, как оно используется в описываемой парадигме, было введено *А. Тьюрингом* и с успехом применяется во многих развитых областях науки, например, в теории управления и теории формальных языков.

Основное свойство состояния системы в момент времени t_0 заключается в «отделении» будущего ($t > t_0$) от прошлого ($t < t_0$) в том смысле, что текущее состояние несет в себе всю информацию о прошлом системы, необходимую для определения ее реакции на любое входное воздействие, формируемое в момент времени t_0.

В разд. 1.1 при описании понятия *«сложное поведение»* упоминалось, что реакция сущности со сложным поведением на входное воздействие может зависеть, в том числе, и от предыстории. При использовании понятия *«состояние»* знание предыстории более не требуется. Состояние можно рассматривать как особую характеристику, которая в неявной форме объединяет все входные воздействия прошлого, влияющие на реакцию сущности в настоящий момент времени. Реакция зависит теперь только от входного воздействия и текущего состояния.

ПРИМЕЧАНИЕ

> По распространенному мнению, эффективность объектно-ориентированного подхода к разработке ПО объясняется тем, что для человека *естественно* мыслить в терминах объектов (сущностей) и взаимодействия между ними. У автоматного подхода, по мнению авторов, также есть все шансы стать эффективным, так как люди живут в состояниях (например, спят или бодрствуют, сыты или голодны), и в

> зависимости от текущего состояния по-разному реагируют на внешние раздражители. Вспомните пословицу: «Сытое брюхо к учению глухо».

Понятие *входное воздействие* также является одним из базовых для автоматного программирования. Чаще всего, входное воздействие – это вектор. Его компоненты подразделяются на *события* и *входные переменные* в зависимости от смысла и механизма формирования. Совокупность конечного множества состояний и конечного множества входных воздействий образует (конечный) *автомат без выходов*. Такой автомат реагирует на входные воздействия, определенным образом изменяя текущее состояние. Правила, по которым происходит смена состояний, называют *функцией переходов* автомата.

То, что в автоматном программировании собственно и называется (конечным) *автомат* (рис. 1.4), получается, если соединить понятие автомата без выходов с понятием *«выходное воздействие»*. Такой автомат реагирует на входное воздействие не только сменой состояния, но и формированием определенных значений на выходах. Правила формирования выходных воздействий называют *функцией выходов* автомата.

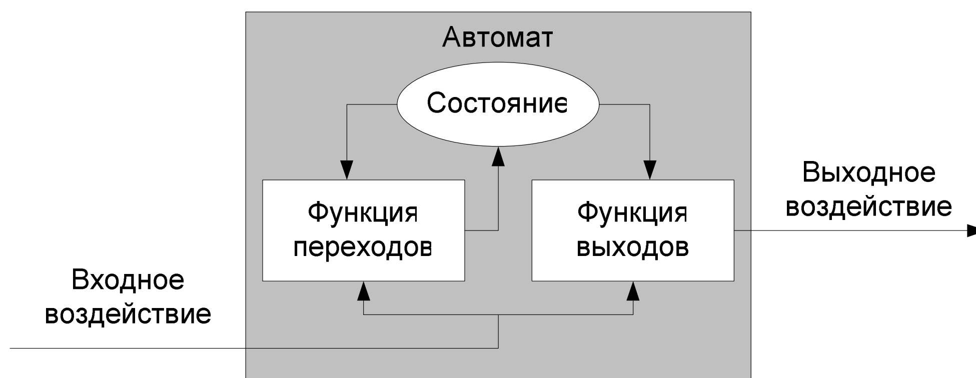

Рис. 1.4. Конечный автомат

1.3. Парадигма автоматного программирования

Для того чтобы лучше разобраться в основных концепциях автоматного программирования, рассмотрим сначала один из абстрактных вычислителей, широко применяемых в теории формальных языков – *машину Тьюринга* [12, 13]. Эта абстрактная машина была предложена *А. Тьюрингом* в 1936 г. в качестве формального определения понятия «алгоритм». Тезис Черча-Тьюринга [13] гласит, что все, что можно «вычислить», «запрограммировать» или «распознать» в любом смысле (из формально определенных в настоящее время), можно вычислить, запрограммировать или распознать с помощью подходящей машины Тьюринга.

Машина Тьюринга состоит из двух частей: устройства управления и запоминающего устройства – ленты (рис. 1.5). Лента содержит бесконечное число ячеек, в которых могут быть записаны символы некоторого конечного алфавита. В каждый момент времени на одной из ячеек ленты установлена головка чтения-записи, позволяющая устройству управления считывать или записывать символ в этой ячейке.

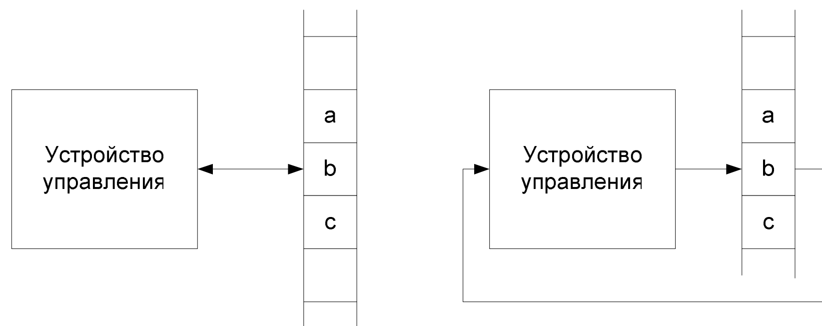

Рис. 1.5. Машина Тьюринга: традиционное изображение (слева) и изображение в традициях теории управления (справа)

Устройство управления представляет собой конечный автомат. У него имеется единственное входное воздействие: символ, считанный с ленты – и два выходных воздействия: символ, записываемый на ленту, и указание головке сдвинуться на одну ячейку в ту или иную сторону, либо остаться на месте.

Тезис Черча-Тьюринга означает, что в терминах операций машины Тьюринга можно записать все те же программы, что и на любом существующем языке программирования.

Как же программировать на машине Тьюринга? Пусть, например, необходимо реализовать функцию *инкремент* (увеличение целого числа на единицу). Пусть исходное число записано на ленте в двоичном виде слева направо, во всех остальных ячейках находится пустой символ (`'blank'`) и головка указывает на самый старший разряд числа. Тогда для увеличения числа на единицу можно предложить следующий алгоритм:

1. Двигаться вправо, пока не встретится пустой символ.
2. Сдвинуться на одну ячейку влево.
3. Пока в текущей ячейке находится символ `'1'`, изменять его на `'0'` и двигаться влево.
4. Если в текущей ячейке находится `'0'` или `'blank'`, записать в ячейку `'1'` и завершить работу.

Этот алгоритм необходимо «ввести» («закодировать») в устройство управления машины Тьюринга. Другими словами, необходимо задать состояния, а также функции переходов и выходов ее управляющего автомата. Удобный и наглядный способ сделать это предоставляют *графы переходов* автоматов, иначе называемые *диаграммами переходов*. Подробнее язык графов переходов будет обсуждаться в разд. 2.2, а пока достаточно знать, что вершины в этом графе соответствуют состояниям автомата, а дуги – переходам между состояниями. Каждая дуга помечается *условием перехода* (значениями входных воздействий, которые инициируют этот переход) и *действием на переходе* (значениями выходных воздействий).

На рис. 1.6 представлен граф переходов управляющего автомата машины Тьюринга, реализующей функцию инкремент. Здесь символ ‘b’ – сокращение от `blank`, символ ‘*’ на месте записываемого символа означает «Записать тот же самый символ, который был считан». Команды головке обозначаются стрелками (стрелка вниз означает «Остаться на месте»). В метке перехода над чертой записывается его условие, а под чертой – действие.

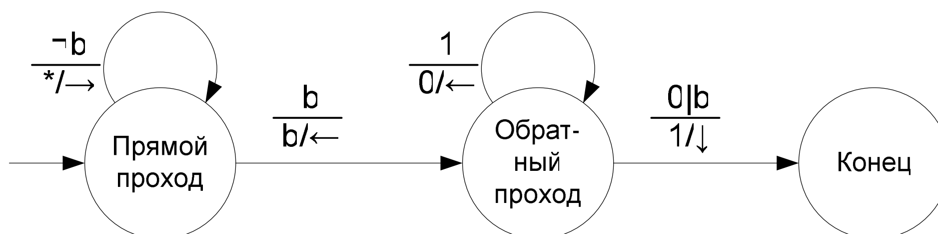

Рис. 1.6. Увеличение числа на единицу с помощью машины Тьюринга

Отметим, что в графе переходов обозначены имена состояний управляющего автомата. Эти имена отражают смысл состояния и являются кратким описанием поведения машины в этом состоянии.

Итак, управляющий автомат машины Тьюринга, реализующей функцию инкремент, имеет три состояния. Сколько же состояний у этой машины в целом? Ее действия в каждый момент времени полностью определяются совокупностью состояния управляющего автомата, строки на ленте и положения головки. Отметим, что символы на ленте для устройства управления представляют собой входные воздействия, однако, относительно машины в целом они не являются входными (внешними), а формируют часть внутреннего состояния вычислителя. Всевозможных строк на ленте, как и положений головки, бесконечно много, поэтому и у машины Тьюринга бесконечное число состояний.

Однако, если задуматься, состояния управляющего устройства и состояния ленты имеют принципиально различные значения. В приведенном примере оказалось, что для того чтобы задать алгоритм для машины Тьюринга, достаточно описать ее поведение в каждом из трех состояний управляющего автомата (рис. 1.6). Нам не потребовалось задавать реакцию машины для каждой из бесконечного числа возможных входных строк. Неформально можно сказать, что состояния управляющего автомата определяют *действия* машины, а состояние ленты – лишь *результат* этих действий.

ПРИМЕЧАНИЕ

> Здесь уместна аналогия из объектно-ориентированного программирования. Вызывая компонент (метод) некоторого класса, клиент указывает имя компонента и, если требуется, его фактические аргументы. Различных компонентов в классе обычно всего несколько, в то время как различных аргументов может быть необозримо много. При этом имя компонента определяет *действие* (алгоритм вычислений), а значения аргументов – только *результат действия* (результат вычислений).

Теперь очевидно, что состояния устройства управления и состояния ленты – совершенно разные понятия с точки зрения программирования на машине Тьюринга, и смешивать их не стоит. Первые следует явно перечислять, отображать на графе переходов, описывать алгоритм поведения в каждом из них. Вторые в программе в явном виде не участвуют, построить граф переходов между ними невозможно, а если бы это и удалось, то для понимания программы такой граф был бы бесполезен. Первые можно назвать качественными состояниями машины, а вторые – количественными. В автоматном программировании для этих двух классов состояний приняты термины, заимствованные из теории управления. Состояния автомата называются *управляющими*, а состояния ленты – *вычислительными*[2].

ПРИМЕЧАНИЕ

> Понятие *«состояние»*, так же как и деление на управляющие и вычислительные состояния, не является «чужеродным» для программирования. Традиционно под состоянием программы подразумевают множество текущих значений всех используемых в ней переменных [14, 15]. И хотя число переменных, равно как и число потенциальных значений каждой переменной, можно считать конечным, получающееся пространство состояний оказывается необозримо большим.
>
> Однако не все исследователи трактуют понятие состояния программы столь прямолинейно. Так, *А. Дж. Перлис* в 1966 г. [16] предложил в описания языка, среды и правил вычислений включать состояния, которые могут подвергаться мониторингу во время исполнения, позволяя диагностировать программы, не нарушая их целостности. В этом же году *Э. Дейкстра* [17] предложил ввести так называемые переменные состояния, с помощью которых можно описывать состояния системы в любой момент времени (*Э. Дейкстра* использовал для этих целей целочисленные переменные). При этом им были поставлены вопросы о том, какие состояния должны вводиться, как много значений должны иметь переменные состояния и что эти значения должны означать. Он предложил сначала определять набор подходящих состояний, а лишь затем строить программу. По мнению *Э. Дейкстры*, диаграммы переходов между состояниями могут оказаться мощным средством для проверки программ. Это обеспечивает поддержку его идеи о том, что программы должны быть с самого начала составлены правильно, а не отлаживаться до тех пор, пока они не станут правильными.

Понятия управляющих и вычислительных состояний применимы не только к машине Тьюринга, но и к любой сущности со сложным поведением. Однако, если в машине Тьюринга отличить управляющие состояния от вычислительных не составляет труда (поскольку она по определению состоит из устройства управления и ленты), то для произвольной сущности *явное выделение управляющих состояний* – сложная задача. Об этом уже упоминалось в разд. 1.1. Причина сложности состоит в том, что различия между управляющими и вычислительными состояниями в общем случае трудно формализовать. Неформально основные различия сформулированы в табл. 1.1.

---

[2] В автоматном программировании (и, в частности, в этой книге), говоря без уточнения о *состоянии* некоторой автоматной модели, сущности или системы со сложным поведением, подразумевают управляющее состояние. Если речь идет о вычислительном состоянии, это оговаривается особо.

Таблица 1.1. Управляющие и вычислительные состояния

Управляющие состояния	Вычислительные состояния
Их число не очень велико	Их число либо бесконечно, либо конечно, но очень велико
Каждое из них имеет вполне определенный смысл и качественно отличается от других	Большинство из них не имеет смысла и отличается от остальных лишь количественно
Они определяют действия, которые совершает сущность	Они непосредственно определяют лишь результаты действий

Представьте себе, что в машине Тьюринга не было бы управляющего автомата. Алгоритм ее работы может быть закодирован на ленте в виде последовательности команд. Машина точно также продолжала бы вести себя по-разному на разных шагах алгоритма, однако, из ее описания это было бы уже не ясно. Логика ее поведения была бы потеряна среди не столь существенных деталей, а управляющие состояния смешались бы с вычислительными. Программировать на такой машине и разбираться в уже существующих программах стало бы практически невозможно.

Переход от ленты, головки и простейших команд к языкам высокого уровня, конечно, упрощает программирование, но при реализации сущностей со сложным поведением полностью проблемы не решает. Вспомните электронные часы из разд. 1.1. Точно так же, как в нашей «воображаемой» машине, состоящей только из ленты, в листинге 1.1 логика затеряна среди деталей. Опыт рассмотрения машины Тьюринга подсказывает, что для того, чтобы сделать программу простой и понятной, необходимо *явно выделить управляющие состояния* (идентифицировать их, дать им имена) и *описать поведение сущности в каждом из них*.

Например, при реализации электронных часов с будильником можно выделить три управляющих состояния: «Будильник выключен», «Установка времени будильника» и «Будильник включен». В каждом из этих состояний реакция будильника на нажатие любой кнопки будет однозначной и специфической.

Как отмечено выше, в случае с машиной Тьюринга выделение управляющих состояний тривиально, так как логика в ней априори вынесена в отдельное устройство – управляющий автомат. Подобная идея используется при построении систем автоматизации, в которых всегда выделяют управляющие устройства и управляемые объекты. Следуя этой концепции, сущность со сложным поведением естественно разделить на две части:

- управляющую часть, ответственную за логику поведения – выбор выполняемых действий, зависящий от текущего состояния и входного воздействия, а также за переход в новое состояние;
- управляемую часть, ответственную за выполнение действий, выбранных для выполнения управляющей частью, и, возможно, за формирование некоторых компонентов входных воздействий для управляющей части – *обратных связей*.

В соответствии с традицией теории управления, управляемая часть здесь и далее называется *объект управления*, а управляющая часть – *система управления*.

Поскольку для реализации управляющей части используются автоматы, то она часто называется *управляющий автомат* или просто *автомат*.

После разделения сущности со сложным поведением на объект управления и автомат реализовать ее уже несложно, а главное, ее реализация становится понятной и удобной для модификации. Вся логика поведения сущности сосредоточена в управляющем автомате. Объект управления, в свою очередь, обладает *простым поведением* (а следовательно, может быть легко реализован традиционными «неавтоматными» методами). Он не обрабатывает непосредственно входные воздействия от внешней среды, а только получает от автомата команды совершить те или иные действия. При этом каждая команда всегда вызывает одно и то же действие (это и есть определение простого поведения).

Таким образом, в соответствии с автоматным подходом, сущности со сложным поведением следует представлять в виде *автоматизированных объектов управления* – так в теории управления называют объект управления, интегрированный с системой управления в одно устройство.

> Парадигма автоматного программирования состоит в представлении сущностей со сложным поведением в виде автоматизированных объектов управления

1.4. Автоматные модели

В этом разделе формализуются концепции, на которых основано автоматное программирование. Цель раздела – исследовать свойства автоматизированного объекта управления как основной модели, используемой в автоматном программировании, и определить его место среди традиционных автоматных моделей. Материал данного раздела способствует более глубокому пониманию философии автоматного программирования, однако, из-за более формального изложения, он труднее для понимания, чем остальные разделы книги. При первом прочтении его можно пропустить.

В настоящее время предложены, исследованы и с успехом применяются различные математические модели, в названиях которых используется слово «автомат». Они имеют много общего в своих основах, но различаются, часто существенно, в деталях. Причина разнообразия автоматных моделей объясняется широтой области их применения (пример различного понимания природы автоматов при создании компиляторов и в задачах логического управления приведен в разд. 1.1). Автоматы являются незаменимым инструментом в таких далеких друг от друга областях как, например:

- математическая лингвистика;
- логическое управление;
- моделирование поведения человека;
- коммуникационные протоколы;
- теория формальных языков, вычислимости и вычислительной сложности.

Выявить общее в различных автоматных моделях можно, если рассмотреть автоматы как частный случай *динамических систем*. Обычно термином «динамическая

система» в технике, природе, жизни и т. д. обозначают систему, процессы которой развиваются во времени. Состояние системы в каждый момент времени характеризуют некоторым множеством обобщенных координат. Процессы в динамической системе описываются уравнениями разных типов относительно обобщенных координат [18].

Динамические системы можно подразделить на несколько классов в зависимости от следующих факторов.

- **Модель времени.** Время может считаться текущим непрерывно или дискретно. В первом случае время изменяется на континууме, во втором — на счетном множестве, элементы которого называются *тактами*.
- **Размерность системы.** Число обобщенных координат может быть конечным или счетным.
- **Мощность множеств координат.** Каждая из обобщенных координат может принимать значения из конечного, счетного или континуального множества.

В физике и технике часто используются системы, в которых время непрерывно и обобщенные координаты также изменяются на континууме. Для описания процессов в таких системах используются дифференциальные уравнения.

Если же время дискретно, но обобщенные координаты принимают значения из континуальных множеств, то для описания процессов используются разностные уравнения.

Те системы, в которых время дискретно, число обобщенных координат конечно и каждая координата может принимать значения из конечного множества, называют *конечными динамическими системами*. Конечные автоматы принадлежат к классу конечных динамических систем.

Системы, которые отличаются от конечных тем, что число обобщенных координат или же множество значений координат может быть бесконечно, образуют более общий класс. К этому классу принадлежат, например, машины Тьюринга.

Рассмотрим конечную динамическую систему с дискретным временем, состояние которой в каждый такт t характеризуется конечным числом обобщенных координат $Y(t)$ ($|Y| = n$). На вход системе подается конечное число входных воздействий $X(t)$ ($|X| = m$). Такая конечная динамическая система называется конечным автоматом, если состояние системы в каждый такт однозначно определяется состоянием системы в предыдущий такт и значениями входных воздействий либо в текущий (1), либо в предыдущий (2) такт:

$$Y(t) = f(X(t), Y(t-1)); \qquad (1)$$

$$Y(t) = f(X(t-1), Y(t-1)). \qquad (2)$$

Таким образом, для описания поведения конечных автоматов используются рекуррентные соотношения определенного вида. Если используется рекуррентное соотношение (1), то автомат называется *автоматом первого рода*, а если соотношение (2) – *автоматом второго рода* [19].

Можно показать, что понятие «конечный автомат» охватывает и те конечные системы, состояния которых определяются предысторией любой наперед заданной конечной длины. Однако оно не охватывает системы, в которых состояние определяется статистически или же зависит от всей предыстории. Таким образом, класс конечных динамических систем, которые «помнят» конечное число предыдущих тактов не шире класса систем, которые «помнят» только один такт. В этой книге рассматриваются только автоматы с задержкой не более чем на один такт.

Любую автоматную модель можно описать как динамическую систему, однако, такое описание слишком абстрактно. Для того чтобы исследовать свойства автоматизированного объекта управления, необходимо рассмотреть различные существующие автоматные модели в деталях.

Далее будут рассмотрены две практически независимые «системы» автоматных моделей: одна заимствована из теории формальных языков, а другая – из области логического управления. Как упоминалось выше, в обеих областях традиционно используются конечные автоматы, однако, понимаются они очень по-разному: применяются различные системы терминов, различные классификации, сферы интересов двух областей практически не пересекаются.

В теории формальных языков изучаются, так называемые, *абстрактные автоматы* (называемые также *абстрактными машинами* или *абстрактными вычислителями*). Внутренняя структура абстрактных автоматов не раскрывается. Интерес при их изучении представляет только вычислительная мощность – класс языков, которые может распознавать машина. Модели абстрактных автоматов рассматриваются в разд. 1.4.1.

В логическом управлении рассматриваются *структурные автоматы*. Они выступают в роли управляющих устройств в системах управления. Интерес здесь представляет число параллельных входов и выходов автомата, его связи с объектом управления и другими элементами системы, простота реализации. Модели структурных автоматов рассматриваются в разд. 1.4.2.

Автоматное программирование, с одной стороны, объединяет в себе опыт двух указанных разделов теории автоматов [20], а с другой – является совершенно самостоятельной областью. При использовании автоматов в программировании не имеют значения ни исследования вычислительной мощности, ни, например, минимизация числа состояний. В этой области нас интересует только то, как использовать конечные автоматы для построения корректных программ.

В разд. 1.4.3 приводится формальное описание модели автоматизированного объекта управления. Эта модель является, в некотором смысле, обобщением других автоматных моделей, и соединяет в себе черты как абстрактных, так и структурных автоматов. В то же время, рассмотрению подлежат именно те свойства модели, которые имеют значение для ее применения в программировании.

1.4.1. Абстрактные автоматы

Абстрактные конечные автоматы принято описывать в следующих терминах. Задано конечное множество символов X, которое называется (входным) *алфавитом*. Множество всех возможных цепочек (последовательностей, строк, слов), составленных из символов алфавита X обозначается X^*. Пустая

последовательность символов обозначается ε, $\varepsilon \in X^*$. Подмножество L множества всех цепочек над алфавитом X, $L \subset X^*$, называется *языком*. Рассматривается следующая проблема: задан язык $L \subset X^*$ и цепочка $\xi \in X^*$. Определить, принадлежит ли цепочка языку ($\xi \in L$)?

Если абстрактный вычислитель способен решить эту проблему для определенного языка $L \subset X^*$ и произвольной строки $\xi \in X^*$, то говорят, что вычислитель *распознает* язык L. Таким образом, абстрактные автоматы описываются в терминах тех языков, которые они распознают. Различные автоматные модели могут распознавать разные классы языков или, другими словами, обладают разной *вычислительной мощностью* (вычислительная мощность модели абстрактных автоматов тем больше, чем шире класс распознаваемых ими языков).

Детерминированный конечный автомат-распознаватель. Детерминированный конечный автомат (ДКА) – это пятерка $\langle X, Y, \delta, y_0, F \rangle$, где X – конечный алфавит входных символов, Y – конечное множество состояний, $\delta : X \times Y \to Y$ – функция переходов, $y_0 \in Y$ – начальное (стартовое) состояние, $F \subset Y$ – множество допускающих состояний.

Расширенная функция переходов $\hat{\delta} : X^* \times Y \to Y$, сопоставляющая новое состояние текущему состоянию и цепочке символов, определяется индуктивно следующим образом [12]:

$$\forall y \in Y \left(\hat{\delta}(\varepsilon, y) = y\right);$$

$$\forall y \in Y \ \forall \xi \in X^* \ \forall x \in X \left(\hat{\delta}(\xi x, y) = \delta(x, \hat{\delta}(\xi, y))\right).$$

В таком случае, если $\hat{\delta}(\xi, y_0) \in F$, то есть, стартуя в начальном состоянии и обработав цепочку ξ, автомат оказывается в одном из допускающих состояний, то говорят, что он допускает эту цепочку. Множество допускаемых цепочек образует язык L, распознаваемый ДКА:

$$L = \left\{\xi \in X^* \mid \hat{\delta}(\xi, y_0) \in F\right\}.$$

Класс языков, распознаваемых ДКА, называют *регулярными языками*. Известно, что он совпадает с классом языков, описываемых регулярными выражениями и автоматными грамматиками [12].

Детерминированный конечный автомат-преобразователь. Для того чтобы наделить модель абстрактного конечного автомата способностью не только давать ответ типа «да/нет», но и выполнять какие-то преобразования, в модель добавляют конечный алфавит выходных символов Z и функцию выхода φ. Если функция выхода имеет вид $\varphi : X \times Y \to Z$, то вычислитель называется *автоматом Мили*, а если $\varphi : Y \to Z$ – *автоматом Мура*. Таким образом, рассматривается шестерка

$\langle X, Y, Z, \delta, \varphi, y_0 \rangle$. Она определяет автоматное отображение $f : X^* \to Z^*$ (преобразование, выполняемое автоматом) следующим образом:

$$f(\varepsilon) = \varepsilon ;$$

$$\forall \xi \in X^* \ \forall x \in X \left(f(\xi x) = f(\xi)\varphi(x, \hat{\delta}(\xi, y_0)) \right).$$

Автоматные отображения — это отображения «без предсказания»: перерабатывая цепочку слева направо, они «не заглядывают вперед». Например, отображение, которое сопоставляет цепочке ее саму, записанную в обратном порядке, не является автоматным.

Недетерминированный конечный автомат. Довольно часто используют модели, в которых считается, что автомат может на каждом такте находиться не в одном, а в нескольких состояниях одновременно. Такие автоматы называют недетерминированными конечными автоматами (НКА) и определяют аналогично ДКА, как пятерку $\langle X, Y, \delta, s_0, F \rangle$, где X — конечный алфавит входных символов, Y — конечное множество состояний, $\delta : X \times Y \to 2^Y$ — функция переходов, которая входному символу и состоянию сопоставляет уже не единственное состояние, а некоторое подмножество множества состояний, $y_0 \in Y$ — начальное состояние, $F \subset Y$ — множество допускающих состояний.

По аналогии с ДКА, расширенная функция переходов $\hat{\delta} : X^* \times Y \to 2^Y$ в этом случае определяется индуктивно:

$$\forall y \in Y \left(\hat{\delta}(\varepsilon, y) = \{y\} \right);$$

$$\forall y \in Y \ \forall \xi \in X^* \ \forall x \in X \left(\hat{\delta}(\xi x, y) = \bigcup_{q \in \hat{\delta}(\xi, y)} \delta(x, q) \right).$$

Язык L, распознаваемый НКА, определяется следующим образом:

$$L = \left\{ \xi \in X^* \mid \hat{\delta}(\xi, y_0) \cap F \neq \varnothing \right\}.$$

Известно, что недетерминизм не увеличивает вычислительной мощности модели и для каждого НКА существует эквивалентный (распознающий тот же язык) ДКА [12, 21].

Автоматы со спонтанными переходами. С точки зрения программирования важным является случай, когда в автоматной модели допускается смена состояния, причиной которого не является внешнее воздействие. В моделях абстрактных автоматов такие переходы называют *ε-переходами*, поскольку буква ε обычно резервируется для обозначения пустого слова – отсутствия какого-либо входного символа. Используют также эквивалентные термины: «спонтанный переход», «немотивированный переход», «переход по завершении».

Из теории формальных языков известно, что допущение ε-переходов не расширяет класса распознаваемых языков автоматных моделей так же, как и допущение недетерминизма [12]. Таким образом, любой автомат с ε-переходами может быть сведен к эквивалентному ДКА.

В программных системах часто используются автоматные модели, в которых *все* переходы являются спонтанными (в разд. 1.4.3 такие модели названы *активными*). Отметим, что немотивированный переход, в общем случае, необязательно является безусловным. Спонтанность выражается в том, что переход совершается по инициативе самого автомата. При этом автомат может проверить истинность некоторого условия и принять решении о совершении перехода, исходя из результата проверки. В этом случае говорят, что переход *помечен* (или *охраняется*) условием.

Еще одно значение спонтанных переходов для программирования состоит в том, что их использование позволяет считать традиционные *блок-схемы* (*схемы алгоритма* [22]) частным случаем диаграмм переходов автоматов. Действительно, схема алгоритма – это граф переходов, в котором из каждого состояния (кроме тех, которые соответствуют заключительным шагам алгоритма) исходит либо единственный безусловный ε-переход (рис. 1.7), либо несколько ε-переходов, помеченных одним или несколькими независимыми условиями.

Рис. 1.7. Элемент схемы алгоритма с единственным спонтанным переходом (слева) и его эквивалент на диаграмме переходов (справа)

Обычно в схемах алгоритмов множество переходов, помеченных независимыми условиями, изображают как сегментированный переход: условия проверяются не одновременно, а одно за другим. Пример приведен на рис. 1.8: на схеме алгоритма (слева) сначала проверяется *«Условие 1»*, а затем, в случае его ложности, *«Условие 2»*. На диаграмме переходов (справа) оба условия проверяются одновременно.

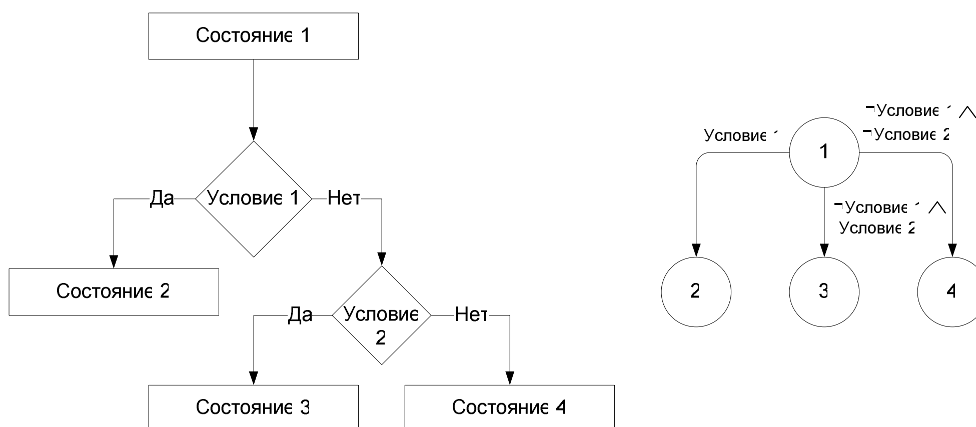

Рис. 1.8. Элемент схемы алгоритма с несколькими исходящими переходами (слева) и его эквивалент на диаграмме переходов (справа)

Именно для схем алгоритмов применяется термин «переход по завершении». Смысл этого термина состоит в том, что причиной перехода из текущего состояния в следующее является завершение выполнения действий, предписанных данному состоянию.

Модели абстрактных конечных автоматов, описанные выше, пригодны лишь для распознавания регулярных языков. Однако существуют и используются нерегулярные языки. Наиболее употребительный пример – распространенные языки программирования. В случае если вычислительной мощности ДКА оказывается недостаточно, применяются различные расширения данной модели. Расширения производятся за счет добавления в модель дополнительной *памяти* того или иного вида.

Автоматы с магазинной памятью. Автоматы с магазинной памятью (МП-автоматы) широко используются для синтаксического анализа. В этой модели к конечному автомату добавляется *магазин* (стек). В нем хранятся символы некоторого магазинного алфавита, который, в общем случае, не совпадает с входным алфавитом. На каждом такте работы автомат снимает один символ с вершины магазина и, в зависимости от его значения, переходит в новое состояние и заталкивает в стек некоторую строку магазинных символов. Такого механизма оказывается достаточно для распознавания *контекстно-свободных языков* – класса языков, в который попадает большая часть конструкций современных языков программирования.

Так же как и конечные автоматы, МП-автоматы могут быть детерминированными и недетерминированными, иметь или не иметь ε-переходы. При этом, в отличие от конечных автоматов, соответствующие модификации МП-автоматов не являются вполне эквивалентными и описывают близкие, но различные подклассы класса контекстно-свободных языков. В частности, детерминированные МП-автоматы без спонтанных переходов распознают те контекстно-свободные языки, которые могут быть описаны синтаксически однозначной порождающей грамматикой. С практической точки зрения это наиболее востребованный класс языков.

Более формально детерминированный МП-автомат (ДМПА) определяется как семерка $\langle X, Y, W, \delta, y_0, w_0, F \rangle$, где X – конечный алфавит входных символов, Y – конечное множество состояний, W – конечный алфавит магазинных символов, $\delta : X \times Y \times W \to Y \times W^*$ – функция переходов, которая по входному символу, состоянию и символу на вершине магазина определяет новое состояние и цепочку символов, которые необходимо записать в магазин, $y_0 \in Y$ – начальное состояние, w_0 – специальный магазинный символ *маркер дна*, $F \subset Y$ – множество допускающих состояний [12].

Для описания работы МП-автомата удобно использовать понятие конфигурации. Конфигурация – это тройка $\langle y, \xi, \omega \rangle$, где $y \in Y$ – состояние, $\xi \in X^*$ – непрочитанная часть входной цепочки, $\omega \in W^*$ – содержимое магазина (по соглашению вершина магазина записывается слева, а дно – справа). Переход МП-автомата – это отношение $\to$ на множестве конфигураций, которое определяется следующим образом:

$$\forall y \in Y \, \forall x \in X \, \forall w \in W;$$
$$(q, \psi) = \delta(x, y, w) \Rightarrow \forall \xi \in X^* \, \forall \omega \in W^* \; (y, x\xi, w\omega) \to (q, \xi, \psi\omega).$$

Отношение $\to^*$ на множестве конфигураций определяется как рефлексивное транзитивное замыкание отношения $\to$.

Заметим, что для ДКА использовать понятие конфигурации (пары – состояние и остаток входа) можно, но в этом нет необходимости, поскольку

$$\hat{\delta}(\xi, y) = q \Leftrightarrow \forall \rho \in X^* \, (x, \xi\rho) \to^* (q, \rho).$$

Достижимость конфигураций полностью описывается расширенной функцией переходов.

Используется два способа определения языка, распознаваемого МП-автоматом:

$L_1 = \left\{ \xi \in X^* \,|\, \exists y \in F : (y_0, \xi, w_0) \to^* (y, \varepsilon, \omega) \right\}$ – язык, распознаваемый по допускающему состоянию. В этом определении содержимое магазина (цепочка ω) в момент исчерпания входной цепочки и достижения допускающего состояния не имеет значения;

$L_2 = \left\{ \xi \in X^* \,|\, (y_0, \xi, w_0) \to^* (y, \varepsilon, \varepsilon) \right\}$ – язык, распознаваемый по опустошению магазина. В этом определении множество допускающих состояний F не имеет значения и его можно исключить из определения МП-автомата.

Известно [12], что данные определения эквиваленты, и при реализации можно использовать то, которое более удобно.

Машина Тьюринга. Машина Тьюринга (МТ) – самый «мощный» из рассматриваемых абстрактных вычислителей. Его неформальное описание было

приведено в разд. 1.3. Образно машину Тьюринга можно назвать «автоматом с ленточной памятью».

Более формально МТ – это семерка $\langle W, Y, \delta, y_0, b, F \rangle$, где W – конечный алфавит ленточных символов, Y – конечное множество состояний, $\delta : W \times Y \rightarrow W \times Y \times \{\leftarrow, \rightarrow, \downarrow\}$ – функция переходов, которая по считанному с ленты символу и состоянию определяет новое состояние, новый символ, который необходимо записать на ленту, и направление сдвига головки (влево, вправо, либо остаться на месте), $y_0 \in Y$ – начальное состояние, $b \in W$ – специальный ленточный символ «пробел», $F \subset Y$ – множество допускающих состояний.

Кроме рассмотренных вычислителей подобную структуру (наличие устройства управления в виде ДКА и некоторого вида дополнительной памяти) имеют еще несколько абстрактных машин. Например, многоленточная машина Тьюринга и счетчиковые машины [12].

Отметим, что только *бесконечная* дополнительная память (которая может хранить бесконечное множество значений) увеличивает вычислительную мощность абстрактной машины. Примерами являются стек МП-автомата и лента машины Тьюринга, приспособленные для хранения бесконечного числа различных строк $\omega \in W^*$. Если же множество возможных значений памяти (обозначим его V) конечно, то вычислитель всегда может быть преобразован в эквивалентный ДКА. Множество состояний этого ДКА будет декартовым произведением $Y \times V$ множества состояний исходного вычислителя и множества значений его дополнительной памяти.

1.4.2. Структурные автоматы

Структурные модели автоматов, используемые в задачах логического управления, описываются совсем в других терминах. Автомат здесь рассматривается не как распознаватель языка, а как устройство управления. Поэтому у всех структурных моделей помимо множества входных воздействий X и множества состояний Y существует и множество выходных воздействий Z (все три множества являются конечными).

По этому признаку структурные автоматы больше всего похожи на абстрактный конечный автомат с выходом. Однако имеются и значительные отличия: ДКА с выходом рассматривается как автоматное отображение входных строк конечной длины в выходные строки той же длины. В случае со структурным автоматом число входных воздействий заранее неизвестно и необязательно конечно. Входными воздействиями для системы управления могут быть, например, значения сигнализаторов. В отсутствие непредвиденных ситуаций работа такой системы никогда не завершается. Здесь значение имеет не вся бесконечная выходная «строка», а каждое выходное воздействие в отдельности и именно в тот момент времени (такт), когда оно сгенерировано. Другими словами, модели абстрактных автоматов применяются в трансформирующих системах (вычисляющих функции над строками), а структурные модели – в реактивных системах (формирующих реакцию на входные воздействия от внешней среды).

В задачах логического управления значение имеет не только конечность множеств X, Y и Z, но и их точная размерность, поскольку она влияет на сложность реализации устройства управления. Кроме того, элементы этих множеств принято считать битовыми векторами (иногда вместо битов удобнее использовать, например, целые числа из небольшого диапазона). Теоретически это несущественно, так как символы любого конечного алфавита можно закодировать битовыми строками одинаковой длины. Такой выбор представления определяется скорее практическими соображениями: удобно считать, что устройство управления имеет несколько параллельных двоичных входов и выходов, причем каждый из них имеет определенный смысл. Например, на один из двоичных входов могут подаваться различные значения в зависимости от того, является ли текущая температура среды допустимой, а на второй – в зависимости от того, является ли допустимым давление. Значения отдельных входов и выходов (компоненты входных и выходных воздействий) называют, соответственно, *входными* и *выходными переменными*, а компоненты состояния – *внутренними переменными*.

Для того чтобы подчеркнуть векторную природу состояний, входных и выходных воздействий далее в этом разделе будем обозначать их как $\overline{y}$, $\overline{x}$ и $\overline{z}$ соответственно.

Перейдем к описанию структурных моделей конечных автоматов.

Автоматы без памяти. Автомат, значения выходов $\overline{z}$ которого зависят только от значений входов $\overline{x}$ в данный момент времени и не зависят от предыстории, называется *комбинационным* (*однотактным*), или *автоматом без памяти*.

Отметим, что в этом случае термин «память» используется в другом смысле, чем при описании абстрактных автоматов в разд. 1.4.1. Здесь речь идет об основной памяти автомата: его управляющих состояниях, которые накапливают информацию о предыстории. В связи с МП-автоматом и машиной Тьюринга говорилось о дополнительной памяти: той, что дана вычислителю помимо его управляющих состояний.

Строго говоря, автомат без памяти не является автоматом в смысле определения, данного в начале разд. 1.4, где автоматом была названа конечная *динамическая* система. Более корректный термин для этого класса автоматов – *комбинационные схемы* (КС). Стандартное обозначение комбинационной схемы приведено на рис. 1.9.

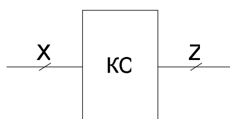

Рис. 1.9. Комбинационная схема

Функционирование комбинационных схем описывается соотношением вида:

$$\overline{z} = f(\overline{x}).$$

Функция f в этом соотношении обычно булева (ее аргументы и результат двоичны).

Булевы функции принято задавать с помощью полностью или не полностью определенных таблиц, которые в первом случае называются *таблицами истинности*,

а во втором – *таблицами решений*. Более компактной формой представления булевых функций являются *булевы формулы*, которые всегда определяют функцию полностью.

Возвращаясь к терминам разд. 1.1, можно сказать, что комбинационная схема (в отличие от автоматов с памятью, рассматриваемых далее) является моделью сущности с простым поведением, поскольку ее выходное воздействие не зависит от состояния.

Автоматы с памятью. Автомат, значения выходов $\bar{z}$ которого зависят не только от значений входов $\bar{x}$ в данный момент времени, но и от предыстории, называется *последовательностным* (*многотактным*), или автоматом с памятью.

Автомат с памятью в отличие от комбинационной схемы является автоматом в обозначенном выше смысле – он представляет собой конечную динамическую систему. Отметим, что способ описания последовательностных автоматов, принятый в логическом управлении, отличается от уравнений динамических систем (формулы (1) и (2)). В логическом управлении время в явном виде не используется. Автоматы представляются в виде структурных схем, которые состоят из элементов двух типов: комбинационных схем и *элементов задержки* (ЭЗ). Для каждого элемента схемы отдельно записывается уравнение преобразования, которое он осуществляет (зависимость выходного сигнала от входного). Элементы задержки на самом деле не преобразуют сигнал: они передают на выход то же, что получили на вход, но через некоторый заданный промежуток времени (в данном случае, один такт), однако, в уравнениях эта задержка в явном виде не отражается.

Ниже рассмотрены наиболее важные классы последовательностных автоматов: автоматы без выходного преобразователя, автоматы Мура, Мили и смешанные автоматы.

Автоматы без выходного преобразователя. Структурная схема автомата без выходного преобразователя первого рода приведена на рис. 1.10 (слева), а второго рода – на рис. 1.10 (справа). Здесь комбинационная схема реализует функцию переходов автомата, а элемент задержки и *обратная связь*, которая передает сигнал с выхода автомата обратно на вход, обеспечивают его память.

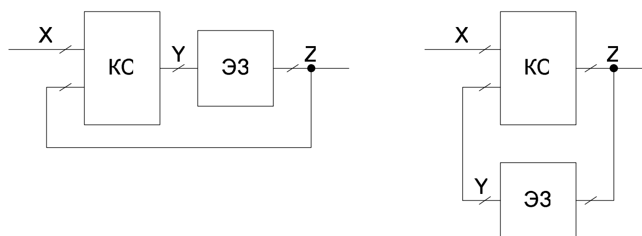

Рис. 1.10. Автоматы без выходного преобразователя: первого рода (слева) и второго рода (справа)

Элементы рассмотренных схем описываются следующими соотношениями (уравнения (3) и (4) описывают автоматы первого и второго рода соответственно):

$$\begin{aligned}\bar{y} &= \delta(\bar{x}, \bar{z}) \\ \bar{z} &= \bar{y}\end{aligned} \qquad (3) \qquad\qquad \begin{aligned}\bar{z} &= \delta(\bar{x}, \bar{y}) \\ \bar{y} &= \bar{z}\end{aligned} \qquad (4)$$

Здесь функция δ, вычисляемая комбинационной схемой КС, имеет смысл функции переходов автомата.

Можно показать, что эти соотношения эквивалентны уравнениям динамических систем (1) и (2). Если в соотношение (3) в явном виде ввести время, то получим:

$$\begin{aligned}\bar{y}(t) &= f(\bar{x}(t), \bar{z}(t)) \\ \bar{z}(t+1) &= \bar{y}(t)\end{aligned} \Rightarrow \bar{y}(t) = f(\bar{x}(t), \bar{y}(t-1)).$$

Аналогично и для соотношения (4):

$$\begin{aligned}\bar{z}(t) &= f(\bar{x}(t), \bar{y}(t)) \\ \bar{y}(t+1) &= \bar{z}(t)\end{aligned} \Rightarrow \bar{y}(t+1) = f(\bar{x}(t), \bar{y}(t)) \Rightarrow \bar{y}(t) = f(\bar{x}(t-1), \bar{y}(t-1).$$

Обратим внимание читателя на различие терминов «автомат без выхода» и «автомат без выходного преобразователя». Рассматриваемый структурный автомат формирует выходные воздействия. Отсутствие выходного преобразователя означает лишь то, что значения выходных переменных совпадают со значениями внутренних переменных автомата (на языке абстрактных автоматов это значит, что функция выходов зависит только от состояний и является им тождественной). Иначе говоря, в этом случае каждое состояние автомата обозначается (*кодируется*) значением выходного воздействия в этом состоянии. Такое кодирование состояний называется *принудительным* [4].

Автоматы без выходного преобразователя имеют ограниченную область применения: они пригодны лишь тогда, когда число управляющих состояний автомата не превышает числа различных выходных воздействий. Приведем простой пример автоматизированного объекта управления, который не обладает таким свойством.

Рассмотрим счетный триггер, состоящий из кнопки, лампочки и управляющего автомата (рис. 1.11). Управляющий автомат в данном случае имеет одну двоичную входную переменную x, принимающую значение единица, если кнопка нажата, и ноль в противном случае, и одну двоичную выходную переменную z, устанавливаемую в единицу, если лампочку требуется включить, и в ноль – если выключить.

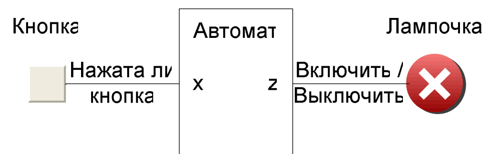

Рис. 1.11. Счетный триггер

Алгоритм работы счетного триггера таков: каждое нажатие кнопки переводит лампочку из выключенного состояние во включенное и наоборот (отпускание кнопки

не влияет на состояние лампочки). Попытаемся изобразить граф переходов автомата без выходного преобразователя, реализующего этот алгоритм (рис. 1.12).

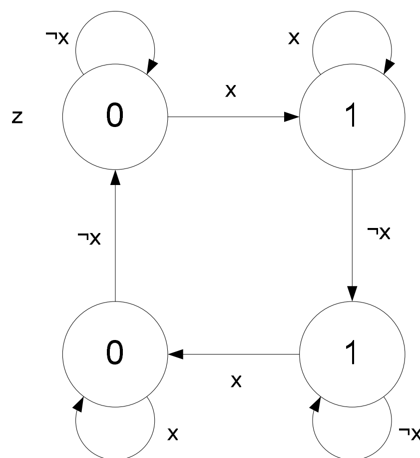

Рис. 1.12. Граф переходов счетного триггера (принудительное кодирование состояний)

При построении графа было использовано принудительное кодирование состояний: код состояния совпадает со значением выходной переменной z в этом состоянии. Это и стало источником проблемы: полученный граф переходов невозможно реализовать, поскольку состояния, соответствующие каждой паре вершин, обозначенных одинаково, неразличимы.

ПРИМЕЧАНИЕ

В действительности, у этой проблемы есть еще один источник: рассматриваемая система не является *событийной*. В событийных системах некоторые или все входные воздействия представляют собой события: они инициируются внешней средой и сигнализируют об *изменениях ее вычислительного состояния*. Если бы входное воздействие счетного триггера состояло из двух событий (первое возникало бы один раз при нажатии кнопки, а второе – один раз при отпускании), то для управления лампочкой достаточно было бы автомата без выходного преобразователя с двумя состояниями, соответствующими состояниям лампочки, и проблемы бы не возникло.

Работать с событиями удобно, однако, это лишь абстракция, причем достаточно высокого уровня (например, прикладное ПО взаимодействует посредством событий с операционной системой). Задачи логического управления предполагают либо аппаратную, либо низкоуровневую программную реализацию. При этом обеспечить взаимодействие автомата с внешней средой посредством событий удается далеко не всегда. Поэтому чаще всего в логическом управлении используется альтернативный способ взаимодействия: автомат опрашивает *вычислительное состояние* внешней среды. При применении вместо событий входной булевой переменной x счетному триггеру требуется четыре управляющих состояния для того, чтобы различать одиночные нажатия кнопки.

Изменим кодирование состояний так, чтобы они стали различимыми. Самый «экономичный» способ сделать это – добавить к коду каждого состояния еще по

одному биту так, чтобы два состояния, обозначаемые одинаково при принудительном кодировании, различались в этом бите (рис. 1.13).

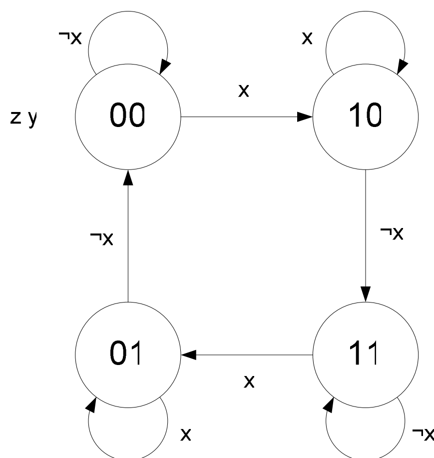

Рис. 1.13. Граф переходов счетного триггера (принудительно-свободное кодирование состояний)

Такое кодирование состояний называется *принудительно-свободным* (один бит кода навязывается значением выходной переменной *z*, а другой определяется свободно, и для его хранения вводится переменная *y* [4]).

Для реализации такого автомата, как и автомата с принудительным кодированием состояний, нет необходимости в выходном преобразователе. Поскольку первый бит кода состояния совпадает со значением выходной переменной *z* , то эту переменную можно без изменений подать на выход автомата (рис. 1.14). Использование принудительно-свободного кодирования состояний расширяет область применения автоматной модели без выходного преобразователя. Поэтому такие автоматы называются *универсальными автоматами без выходного преобразователя*.

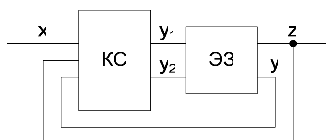

Рис. 1.14. Универсальный автомат без выходного преобразователя (первого рода), реализующий счетный триггер

Можно пойти еще дальше и ввести вместо двух логических внутренних переменных одну многозначную (целочисленную, символьную, строковую). В этом случае коды состояний выбираются произвольно, например {1, 2, 3, 4}, {‘a’, ‘b’, ‘c’, ‘d’} или {«Кнопка опущена, лампа выключена», «Кнопка нажата, лампа включена», «Кнопка отпущена, лампа включена», «Кнопка нажата, лампа выключена»}. Такой способ кодирования состояний называется *свободным*. Он удобен: выходные и внутренние переменные автомата независимы, состояниям можно присваивать мнемонические строковые имена, однако для формирования выходного воздействия в этом случае требуется выходной преобразователь. Именно такой способ кодирования состояний

автомата и было предложено в работе [4] применять в программировании в качестве основного.

Автомат Мура. Для преобразования свободно выбранных кодов состояний в значения выходных воздействий введем в автомат *выходной преобразователь* – еще одну комбинационную схему $КС_2$, реализующую функцию выходов автомата (в отличие от первой комбинационной схемы $КС_1$, реализующей функцию переходов).

Понятие структурного автомата Мура аналогично понятию абстрактного автомата Мура, введенному в разд. 1.4.1. В таком автомате выходное воздействие зависит только от состояния и не зависит от входного воздействия. Структурные схемы автоматов Мура первого и второго рода приведены на рис. 1.15. Автомат первого рода формирует выходные воздействия на основе текущих (не обновленных) значений внутренних переменных, а автомат второго рода, наоборот, сначала обновляет состояние, а затем использует его новое значение для вычисления выходного воздействия.

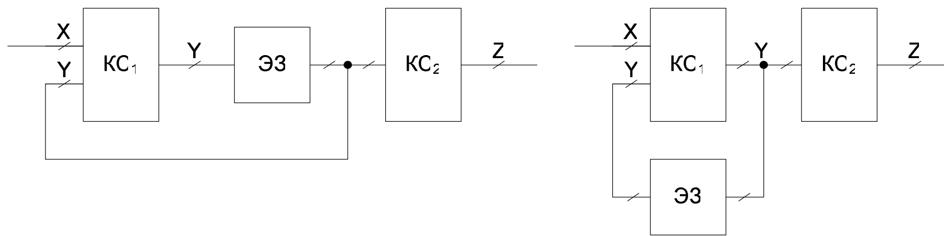

Рис. 1.15. Автоматы Мура: первого рода (слева) и второго рода (справа)

Автоматы Мура первого и второго рода описываются соотношениями (5) и (6) соответственно:

$$\begin{aligned} \bar{y}' &= \delta(\bar{x}, \bar{y}) \\ \bar{z} &= \varphi(\bar{y}) \\ \bar{y} &= \bar{y}' \end{aligned} \qquad (5) \qquad\qquad \begin{aligned} \bar{y}' &= \delta(\bar{x}, \bar{y}) \\ \bar{z} &= \varphi(\bar{y}') \\ \bar{y} &= \bar{y}' \end{aligned} \qquad (6)$$

Здесь функция переходов δ и функция выходов φ вычисляются соответственно схемами $КС_1$ и $КС_2$.

На рис. 1.16 приведен граф переходов автомата Мура счетного триггера с многозначным кодированием состояний.

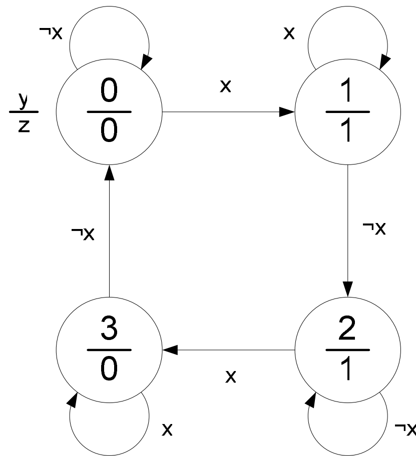

Рис. 1.16. Граф переходов автомата Мура счетного триггера

Возможность применения *одной переменной* для хранения номера состояния при любом числе состояний в автомате позволяет сделать процесс его работы *наблюдаемым* [4]. Наблюдаемость является одним из важнейших свойств, рассматриваемых в теории автоматического управления [23]. Кроме того, переменная состояния может применяться для обеспечения взаимодействия между автоматами. Для этого при реализации автомата необходимо, чтобы значение переменной состояния после *каждого* такта работы было доступно всем автоматам системы.

Автомат Мили. По аналогии с абстрактными автоматами, структурный автомат, выходные воздействия которого зависят не только от состояния, но и от входных воздействий, называется автоматом Мили (рис. 1.17).

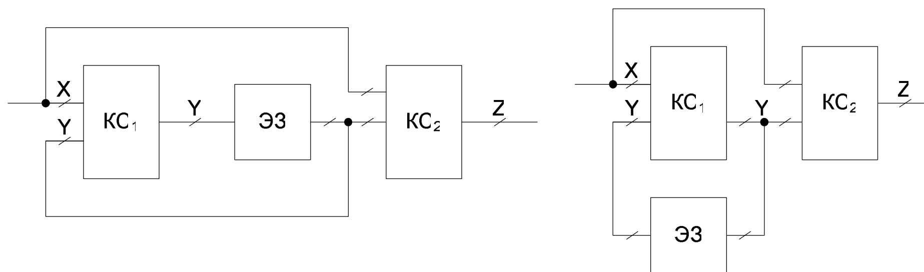

Рис. 1.17. Автоматы Мили: первого рода (слева) и второго рода (справа)

Автоматы Мили первого и второго рода описываются соотношениями (7) и (8) соответственно:

$$\begin{aligned} \bar{y}' &= \delta(\bar{x},\bar{y}) \\ \bar{z} &= \varphi(\bar{x},\bar{y}) \\ \bar{y} &= \bar{y}' \end{aligned} \qquad (7)$$

$$\begin{aligned} \bar{y}' &= \delta(\bar{x},\bar{y}) \\ \bar{z} &= \varphi(\bar{x},\bar{y}') \\ \bar{y} &= \bar{y}' \end{aligned} \qquad (8)$$

Известно [24], что для любого автомата Мили можно построить эквивалентный ему автомат Мура. Число состояний в таком автомате будет не меньше, чем в исходном.

В качестве примера рассмотрим *последовательный двоичный одноразрядный сумматор*, который выполняет сложение одноименных бит двух двоичных чисел с учетом переноса. Результатом работы сумматора является одноименный бит суммы и перенос в следующий разряд. На рис. 1.18 приведен автомат Мили, реализующий это «устройство».

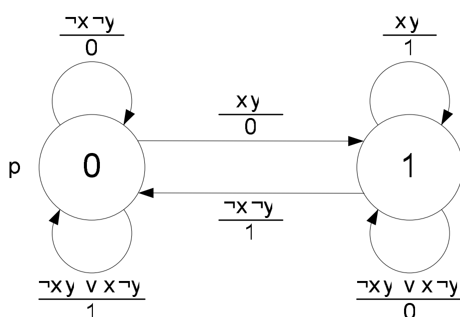

Рис. 1.18. Автомат Мили последовательного двоичного одноразрядного сумматора

На этом рисунке x_i и y_i – складываемые биты, состояния соответствуют значениям переноса, а значение бита суммы формируется как выходное воздействие на переходах.

Смешанные автоматы. Если часть выходных переменных автомата зависит только от состояний, а остальные – также и от входных воздействий, удобно разделить функцию выходов φ на две составляющие: $\varphi_1(y)$ и $\varphi_2(x,y)$. В структурную схему автомата в этом случае вводится не один, как в автоматах Мура и Мили, а два выходных преобразователя (рис. 1.19). Такие автоматы называются *смешанными*, автоматами Мура-Мили или С-автоматами.

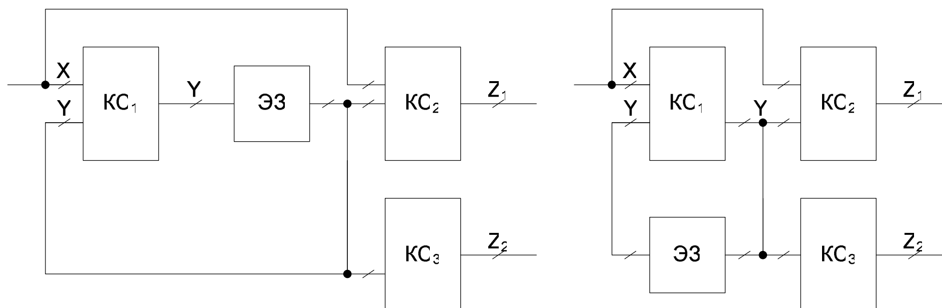

Рис. 1.19. Смешанные автоматы: первого рода (слева) и второго рода (справа)

1.4.3. Автоматы в программировании

Рассмотрев два типа автоматов: абстрактные, применяемые в теории формальных языков, и структурные, которые используются в логическом управлении – попытаемся обобщить полученные сведения, указать сходства и различия автоматных моделей и выделить те их черты, которые важны при применении автоматов в программировании.

Цель этого раздела – построить модель автоматизированного объекта управления (для краткости называемого просто *автоматизированным объектом*, АО). Это понятие уже было введено неформально (разд. 1.3) как совокупность управляющего автомата и объекта управления. Предпосылкой для создания этой концепции была необходимость проектирования и реализации систем со сложным поведением. Теперь попробуем придти к понятию автоматизированного объекта управления с другой стороны: путем обобщения традиционных автоматных моделей.

Вспомним абстрактные модели автоматов, рассмотренные в разд. 1.4.1. Как отмечалось выше, все они имеют похожую структуру: состоят из устройства управления (представляющего собой ДКА с выходом) и хранилища данных того или иного вида (лента, магазин). Для теории формальных языков вид хранилища и набор элементарных операций с данными имеют решающее значение: они определяют вычислительную мощность машины. При моделировании и высокоуровневой программной реализации сущностей со сложным поведением, удобнее заменить конкретное хранилище данных *объектом управления* (ОУ), множество (*вычислительных*) состояний которого может быть любым и определяется спецификой решаемой задачи (рис. 1.20).

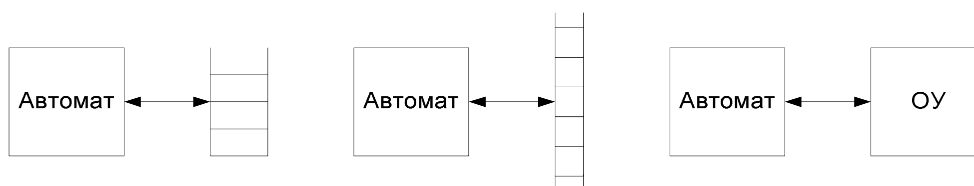

Рис. 1.20. Слева направо: автомат с магазинной памятью, автомат с ленточной памятью (машина Тьюринга), автомат с произвольной памятью (автоматизированный объект)

Вместо ограниченного набора элементарных операций с данными будем использовать произвольные *запросы* и *команды*[3]. В соответствии с традицией теории формальных языков такой вычислитель можно было бы назвать *автоматом с произвольной памятью*.

Важная черта устройства управления всех абстрактных машин – его конечность. Управляющий автомат не только имеет конечное число состояний, но, кроме того, реализуемые им функции переходов и выходов оперируют исключительно конечными множествами. Именно это свойство позволяет описывать логику поведения машины явно: в виде таблицы или графа переходов. Поэтому свойство конечности устройства управления необходимо сохранить при построении модели автоматизированного объекта. Более того, это свойство целесообразно усилить следующим неформальным требованием: число управляющих состояний, входных и выходных воздействий должно быть небольшим (*обозримым*). Управляющий автомат с тысячей состояний, безусловно, является конечным, однако, изобразить его

[3] В объектно-ориентированном программировании *запросами* (или *чистыми функциями*) называются компоненты класса, возвращающие значение и не имеющие побочных эффектов. Такие компоненты не изменяют вычислительные состояния объекта, но позволяют получить некоторую информацию об этих состояниях. Другой тип компонентов класса (*команды*) напротив, предназначен для изменения состояний объектов.

граф переходов практически невозможно, что сводит на нет преимущества явного выделения управляющих состояний.

Напротив, число вычислительных состояний может быть сколь угодно большим (при переходе от модели к программной реализации оно с необходимостью станет конечным, однако, может остаться необозримым). В процессе работы управляющему автомату требуется получать информацию о вычислительном состоянии и изменять его. Однако в силу свойства конечности автомат не может напрямую считывать и записывать вычислительное состояние. Для этого и требуются операции объекта управления. Небольшое число запросов, возвращающих конечнозначные результаты, позволяет автомату получать информацию о вычислительных состояниях, которую он способен обработать. Небольшое число команд используется для «косвенного» изменения вычислительных состояний.

Операции с памятью в традиционных абстрактных вычислителях также можно считать командами и запросами. Рассмотрим, например, автомат с магазинной памятью. Его множество вычислительных состояний бесконечно: это множество всех возможных конфигураций стека. Для управления стеком автомат использует один запрос `top`, возвращающий символ на вершине стека, и две команды `pop` и `push`, первая из которых снимает символ со стека, а вторая заталкивает символ в стек[4].

В вопросах, связанных с программированием, важна не только структура АО, но и особенности процесса его работы. Работа всех рассмотренных автоматных моделей разбита на такты. За такт они успевают считать входное воздействие, вычислить функции переходов и выходов и обновить значения выходных и внутренних переменных.

Некоторые из рассмотренных моделей (например, ДКА и детерминированный МП-автомат) начинают следующий такт работы только в том случае, если получают на вход очередной символ. Их работа всегда заканчивается по достижении конца входной строки. Обобщая такое поведение, введем понятие *пассивной* автоматной модели: каждый такт ее работы инициируется внешней средой, а по окончании такта управление передается обратно этой среде.

У автоматных моделей, например, машины Тьюринга и структурных автоматов, напротив, после окончания текущего такта немедленно начинается следующий. Процесс работы продолжается до тех пор, пока автомат может совершить очередной переход[5]. Такие автоматные модели можно назвать *активными*, поскольку, будучи однажды запущенными, они не нуждаются в дальнейших «стимулах» для продолжения работы.

---

[4] `Push` можно считать как одной командой, аргументом которой является заталкиваемый символ, так и набором команд без аргументов – по одной для каждого символа магазинного алфавита. Это не имеет значения, поскольку магазинный алфавит конечен.

[5] Указанный критерий завершения работы используется при описании машин Тьюринга. При спецификации поведения программных систем в целях наглядности удобно выделять на графах переходов специальные *конечные* (*завершающие*) *состояния*. Переход в любое из таких состояний приводит к немедленному завершению работы автомата.

ПРИМЕЧАНИЕ

> На самом деле, при аппаратной реализации синхронных структурных автоматных моделей начало нового такта инициируется *тактовым генератором*. Однако предполагается, что он является «внутренним» для автомата по сравнению с внешней средой, подающей входные воздействия. Поэтому, с точки зрения среды, такой автомат является активным.

Если активная автоматная модель в качестве входного воздействия считывает вычислительное состояние внешней среды, то для пассивной характерно событийное взаимодействие: среда сама (асинхронно) сигнализирует о своем изменении, вызывая автоматную модель с некоторым событием.

В программировании целесообразно использовать как активные, так и пассивные автоматные модели в зависимости от решаемой задачи. В пассивной модели, в общем случае, лишь некоторые компоненты входного воздействия являются событиями, а остальные представляют собой «традиционные» входные переменные: их значения опрашиваются самим автоматом, а их изменения не инициируют начало такта.

Вернемся к абстрактным вычислителям. Заметим, что в некоторых из них (таких как ДКА) автомат взаимодействует только с внешней средой, получая от нее входные символы. В других (например, в машине Тьюринга) – автомат общается лишь со своим объектом управления, или, в терминах теории абстрактных автоматов, со своей дополнительной памятью. В третьих (таких как МП-автомат) – устройство управление взаимодействует и с внешней средой и с объектом управления, причем от среды оно получает лишь входные воздействия, тогда как взаимодействие с объектом управления имеет двунаправленный характер. При построении модели автоматизированного объекта целесообразно использовать третий вариант как наиболее общий (рис. 1.21).

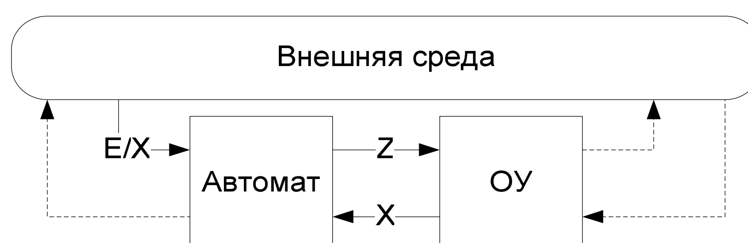

Рис. 1.21. Взаимодействие компонентов модели автоматизированного объекта

На этом рисунке сплошными стрелками обозначены традиционные и наиболее типичные для программных реализаций виды взаимодействия между автоматом, объектом управления и внешней средой. Автомат получает входные воздействия как со стороны среды, так и от объекта управления. В событийных системах часть или все компоненты входного воздействия со стороны среды могут быть событиями (множество событий обозначено на рисунке буквой E). Входное воздействие со стороны объекта управления формирует в модели *обратную связь* (от управляемого объекта к управляющему). Это воздействие может отсутствовать, тогда модель является *разомкнутой* – так в теории управления называются системы управления без обратной связи [23]. В противном случае модель называется *замкнутой*.

Автомат, в свою очередь, воздействует на объект управления.

Пунктирными стрелками обозначены менее распространенные, хотя и возможные, варианты взаимодействия. Так, автомат может оказывать выходное воздействие и на внешнюю среду. Однако таких связей обычно можно избежать, включив все управляемые автоматом сущности в состав его объекта управления. Отметим, что в программировании, в общем случае, различие между объектом управления и внешней средой носит скорее концептуальный, а не формальный характер. Создавая модель системы со сложным поведением, *разработчик производит ее декомпозицию на автоматизированные объекты*, определяя тем самым объект управления каждого автомата. В целях минимизации связей между модулями программной системы целесообразно проводить декомпозицию таким образом, чтобы автомат оказывал выходные воздействия только на собственный объект управления.

Кроме того, объект управления может взаимодействовать с внешней средой напрямую.

Напомним, что в абстрактных автоматных моделях входные и выходные воздействия обычно представляют собой символы некоторого конечного алфавита или цепочки таких символов, а в структурных моделях – битовые строки заданной длины. В программировании на вид входных и выходных воздействий нет ограничений: это могут быть символы, числа, строки, множества, последовательности, произвольные объекты – все зависит от специфики поставленной задачи и инструментов, используемых для ее решения. Кроме того, могут различаться способы передачи входных воздействий автомату и интерпретации выходных воздействий в объекте управления.

Если по назначению сущность близка к традиционной системе управления, то представление входных и выходных воздействий битовыми строками будет удобным по тем же причинам, что и для структурных автоматных моделей. Однако интерпретация этого представления может быть различной. В примере со счетным триггером (разд. 1.4.2) каждое из двух значений выходной переменной соответствовало определенному вычислительному состоянию: включенной или выключенной лампочке. В программировании чаще используется другая интерпретация: каждой выходной переменной сопоставляется определенное *изменение* вычислительного состояния (действие, команда). При этом единица обозначает наличие действия, а ноль – его отсутствие. В этом случае вектору из нулей соответствует отсутствие каких-либо команд. Такой вид выходного воздействия может привести к недетерминизму в том случае, если результат зависит от последовательности выполнения команд. Поэтому в качестве выходного воздействия вместо множества команд часто используется последовательность команд.

При рассмотрении всевозможных деталей использования автоматных моделей в программировании становится ясно, что выбрать одну конкретную модель, подходящую для всех задач, невозможно. При программной реализации сущностей со сложным поведением применение могут найти активные и пассивные, разомкнутые и замкнутые модели, различные формы представления и интерпретации входных и выходных воздействий. Модель автоматизированного объекта управления должна быть применима для любой сущности со сложным поведением, и поэтому целесообразно сформулировать ее довольно абстрактно. Примеры программной

реализации сущностей со сложным поведением, которые будут приведены в последующих главах, являются конкретными воплощениями этой модели.

Итак, приведем формальное определение автоматизированного объекта управления.

Пара $\langle A, O \rangle$, состоящая из управляющего автомата и объекта управления, называется *автоматизированным объектом управления*.

Управляющий автомат представляет собой шестерку $\langle X, Y, Z, y_0, \varphi, \delta \rangle$, где $X = X_E \times X_O$ – конечное множество входных воздействий, причем каждое входное воздействие x состоит из компоненты x_E, порождаемой внешней средой, и компоненты x_O, порождаемой объектом управления; Y – конечное множество управляющих состояний; Z – конечное множество выходных воздействий; $y_0 \in Y$ – начальное состояние; $\varphi = (\varphi', \varphi'')$ – функция выходов (выходных воздействий), состоящая, в общем случае, из двух компонент: функции выходных воздействий в состояниях $\varphi' : Y \to Z$ и функции выходных воздействий на переходах $\varphi'' : X \times Y \to Z$; $\delta : X \times Y \to Y$ – функция переходов.

Объект управления – это тройка $\langle V, f_q, f_c \rangle$, где V – потенциально бесконечное множество вычислительных состояний (или значений), $f_q : V \to X_O$ – функция, сопоставляющая входное воздействие вычислительному состоянию, $f_c : Z \times V \to V$ – функция, изменяющая вычислительное состояние в зависимости от выходного воздействия.

Функции f_q и f_c являются математическими эквивалентами набора запросов и набора команд соответственно.

Графическое представление описанной модели приведено на рис. 1.22.

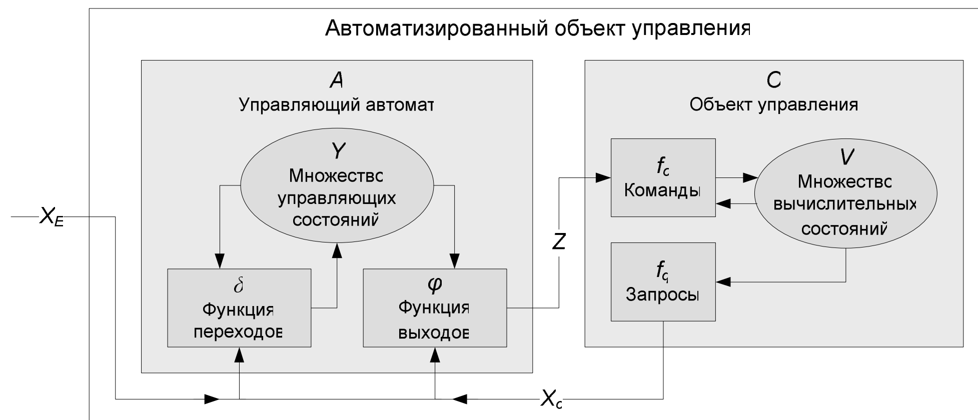

Рис. 1.22. Автоматизированный объект управления

Таким образом, с позиций теории формальных языков автоматизированный объект – это автомат с произвольной памятью. По вычислительной мощности он, в общем случае, эквивалентен машине Тьюринга. Формально, для обеспечения эквивалентности необходимо потребовать, чтобы запросы и команды объекта управления являлись *вычислимыми* функциями (их можно было вычислить с помощью машины Тьюринга). В определении это свойство подразумевается.

В разомкнутой модели автоматизированного объекта входные воздействия поступают только от внешней среды ($X = X_E$), а запросы объекта управления отсутствуют (рис. 1.23). Такой автоматизированный объект по вычислительной мощности эквивалентен ДКА: его объект управления уже не является дополнительной памятью, а скорее представляет собой разновидность выходной ленты.

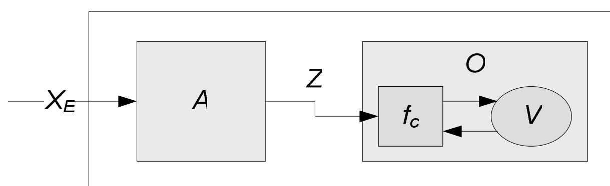

Рис. 1.23. Разомкнутый автоматизированный объект управления

В терминах теории логического управления автоматизированный объект – это, в общем случае, замкнутая система, управляемая автоматом первого рода с выходным преобразователем, в качестве которого может использоваться автомат Мура, Мили или смешанный автомат.

Как было упомянуто выше, автоматизированный объект эквивалентен по вычислительной мощности машине Тьюринга. Иначе говоря, для любого АО можно построить машину Тьюринга, которая решает ту же задачу, и наоборот. С одной стороны, из этого свойства следует важное достоинство автоматного подхода: с помощью автоматизированного объекта можно описать любой алгоритм, любую программу, которая только может быть выполнена компьютером. С другой стороны, возникает вопрос: зачем было изобретать автоматизированный объект вместо того, чтобы выбрать на роль модели сущности со сложным поведением более простую и столь же мощную машину Тьюринга?

Ответ приходит сам собой, если вспомнить пример с машиной Тьюринга, реализующей функцию инкремент (разд. 1.3). Для вычисления простейшей функции понадобился управляющий автомат из трех состояний. Автомат машины Тьюринга, выполняющей умножение двух чисел [12] (преобразование строки «0^n10^m1» в строку «0^{nm}»), содержит уже 12 состояний! Такая модель не только не упрощает описание сложного поведения, но и значительно усложняет описание простого.

Программирование на машине Тьюринга (или *тьюрингово программирование* [25]) чрезвычайно сложно и непрактично по той причине, что набор элементарных операций этой машины с дополнительной памятью очень ограничен. Поэтому автомату приходится не только управлять, но и *выполнять не свойственную ему функцию вычисления*. Переход к модели автоматизированного объекта управления позволяет использовать в качестве элементарных операций произвольные запросы и

команды. При этом на управляющий автомат ложится лишь часть ответственности по реализации алгоритма: та, что связана с логикой (управлением) – то, для чего автоматы, собственно, и предназначены.

Можно сказать, что при программировании на машине Тьюринга любое поведение (кроме алгоритма, состоящего из единственного шага, на котором необходимо записать символ и сдвинуть головку) является сложным. В автоматном программировании грань между простым и сложным поведением определяется разработчиком. Умножение двух чисел может быть сложным, если вы собираетесь эмулировать и визуализировать счеты. Построение сбалансированного дерева поиска [26] может быть элементарной операцией, если в вашем распоряжении есть соответствующая библиотека, предоставляющая готовую функцию. Таким образом, переход от тьюрингова программирования к автоматному состоит в повышении уровня абстракции операций с памятью, причем этот уровень при автоматном подходе не фиксирован, а зависит от решаемой задачи.

Более того, низкоуровневый автоматизированный объект может быть инкапсулирован в объекте управления, существующем на более высоком уровне абстракции. Благодаря такому вложению автоматизированных объектов автоматное программирование поддерживает концепцию выделения уровней абстракции, распространенную в современной методологии разработки ПО.

Выбор уровня абстракции элементарных операций при моделировании сущности со сложным поведением определяет разделение поведения на логику и семантику, а состояний на управляющие и вычислительные. Это и есть наиболее творческий и нетривиальный шаг в автоматном подходе к разработке ПО. Выбор чересчур простых элементарных операций приводит к разрастанию и усложнению автомата, логика становится менее понятной и перегруженной деталями, которые эффективнее было бы реализовать в объекте управления. Машина Тьюринга – крайний случай такого «злоупотребления логикой».

Напротив, выбор слишком абстрактных запросов и команд ведет к усложнению их реализации, перегруженности флагами и условными конструкциями. Вырожденный случай такого «злоупотребления семантикой» – использование традиционных (неавтоматных) стилей программирования, где любое поведение считается простым (рис. 1.24).

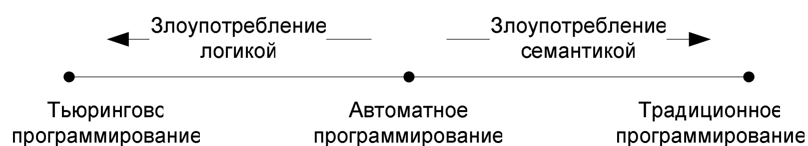

Рис. 1.24. Золотая середина в разделении поведения на логику и семантику

> Нахождение компромисса между сложностью автомата и сложностью операций объекта управления, примирение тьюрингова программирования с традиционным и есть «миссия» автоматного подхода в мире разработки программного обеспечения

Защитники традиционных парадигм программирования могут возразить, что борьба со сложностью программ осуществляется там путем декомпозиции: функциональной

(в процедурном программировании) или объектной (при объектно-ориентированном подходе). При наличии сложной логики декомпозиция лишь распределяет ее по различным компонентам системы, но не делает ее явной и, конечно же, не устраняет ее. Таким образом, декомпозиция отчасти выполняет одну функцию автоматного подхода – упрощение операций, однако совершенно не справляется с другой – формированием у разработчика целостной картины поведения сущности.

С другой стороны, борьба со сложностью автоматов оставалась (и отчасти, остается до сих пор) одной из главных задач автоматного программирования. Эта задача по-разному решается в процедурной и объектно-ориентированной разновидностях парадигмы. Два различных решения подробно обсуждаются во второй и третьей главах этой книги. Пока отметим лишь то, что объектно-ориентированный подход поддерживает понятие автоматизированного объекта управления более непосредственно. Поэтому борьба со сложностью как автомата, так и объекта управления, осуществляются при этом подходе гораздо проще и эффективнее.

Глава 2. Процедурное программирование с явным выделением состояний

Данная глава посвящена описанию автоматного программирования в том виде, в котором оно было предложено в работах [2, 27]. Автоматное программирование в таком традиционном понимании было названо в этих работах *программированием с явным выделением состояний*. Более точным названием было бы *процедурное программирование с явным выделением состояний*, поскольку описываемая разновидность автоматного программирования сочетает в себе концепции автоматного подхода (разделение сложного поведения на логику и семантику, применение конечных автоматов для описания логики) и технику традиционного процедурного программирования.

Автоматное программирование невозможно без поддерживающей его графической нотации, поскольку именно графическое описание автоматов позволяет сделать логику системы понятной. Читатель отчасти уже знаком с нотацией *графов переходов* конечных автоматов. Она была представлена в разд. 1.3 при описании машины Тьюринга, реализующей функцию инкремент, и использована в разд. 1.4.2 при обсуждении счетного триггера и последовательного двоичного одноразрядного сумматора. В этой главе графы переходов применяются как наглядные иллюстрации уже в разд. 2.1 при описании основных концепций автоматного проектирования. Подробное описание нотации приведено в разд. 2.2, который полностью посвящен языку спецификации, применяемому в автоматном программировании. Разд. 2.3 посвящен вопросам реализации автоматов и систем со сложным поведением в целом с использованием традиционных процедурных языков программирования.

Наиболее близкий аналог описываемого подхода – метод под названием *Statemate*, который был предложен *Д. Харелом* и *М. Полити* в книге [28]. Далее в этой главе приводятся сравнения программирования с явным выделением состояний и метода *Statemate* по нескольким параметрам.

Во избежание неоднозначности опишем вкратце, что понимается в этой книге под процедурным программированием. Легче всего определить эту парадигму от противного: это не декларативное, не функциональное и не объектно-ориентированное программирование, причем с практической точки зрения, наиболее важным является последнее противопоставление. Для последующего обсуждения интерес представляют две главные черты процедурного стиля: использование подпрограмм (процедур, функций) в качестве модулей, составляющих архитектуру программной системы, и функциональная декомпозиция *сверху вниз* как основной метод проектирования.

Проектирование сверху вниз начинается с максимально общего и краткого ответа на вопрос «Что делает система?» Далее шаг за шагом описания действий системы уточняются: каждое из них заменяется последовательностью, условием или циклом, в которых участвуют действия, находящиеся на более низком уровне абстракции (рис. 2.1). Процесс заканчивается, когда на очередном шаге описания всех действий оказываются на достаточно низком уровне абстракции, допускающем непосредственную реализацию с помощью примитивов языка программирования или имеющихся библиотек.

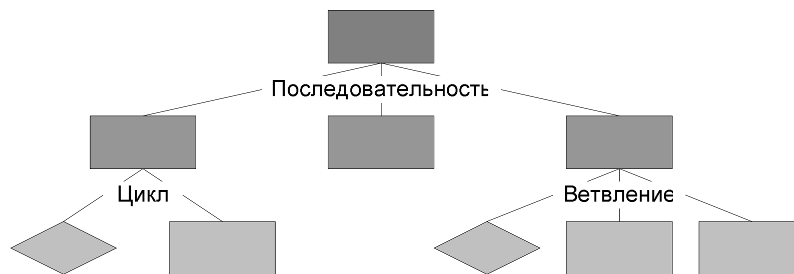

Рис. 2.1. Архитектура системы при проектировании сверху вниз: дерево подпрограмм

На этапе реализации все действия превращаются в подпрограммы (процедуры и функции). Подпрограммы, соответствующие действиям более высокого уровня абстракции, вызывают подпрограммы более низкого уровня. Наконец, самый абстрактный ответ на вопрос «Что делает система?» становится главной программой системы.

2.1. Проектирование

В программировании с явным выделением состояний, как и в традиционном процедурном подходе, может применяться проектирование сверху вниз, однако на верхних уровнях абстракции модулями здесь являются не подпрограммы, а *автоматы*.

ПРИМЕЧАНИЕ

> Обычно в программировании с явным выделением состояний автоматы *реализуются* как подпрограммы, однако на этапе проектирования смешивать эти понятия не следует.

2.1.1. Программные системы, управляемые одним автоматом

Процесс *автоматного проектирования сверху вниз* небольшой программной системы можно кратко описать следующим образом.

1. По словесному описанию поведения системы строится набор *управляющих состояний*.
2. Строится *управляющий автомат* программной системы: управляющие состояния связываются между собой переходами, добавляются входные и выходные переменные, необходимые для реализации заданного в словесном описании поведения. Если система является событийной, определяется набор событий, обрабатываемых автоматом, они включаются в условия переходов наряду с входными переменными.
3. Входным переменным автомата сопоставляются *запросы*: булевы функции или (реже) двоичные переменные. Выходным переменным – *команды*: процедуры. Если упомянутые подпрограммы являются недостаточно простыми для непосредственной реализации, для каждой из них производится цикл традиционного проектирования сверху вниз.

4. Вводятся переменные, необходимые для корректной реализации упомянутых запросов и команд. Совокупность значений этих переменных определяет множество *вычислительных состояний* системы.

Этот процесс приводит к структуре, изображенной на рис. 2.2.

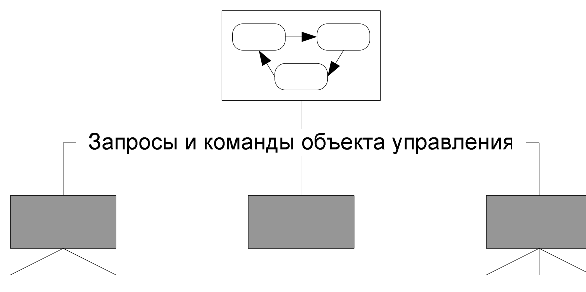

Рис. 2.2. Архитектура системы, управляемой одним автоматом

Рассмотрим в качестве примера, как применить этот подход для проектирования программного эмулятора часов с будильником, рассмотренных в разд. 1.1.

1. Из словесного описания поведения часов можно заключить, что они ведут себя по-разному в зависимости от того, включен или выключен будильник. Кроме того, у часов есть *режим* установки времени будильника. Следовательно, в данной системе целесообразно выделить три управляющих состояния: «Будильник выключен», «Установка времени будильника», «Будильник включен» (рис. 2.3).

Рис. 2.3. Управляющие состояния эмулятора часов с будильником

СОВЕТ

В процессе выделения управляющих состояний приходится внимательно исследовать описание сущности со сложным поведением в поисках набора «ситуаций», в которых поведение сущности имеет качественные особенности. Однако некоторые формулировки в описании сущности могут упростить задачу поиска состояний. Например, понятие *режим* является синонимом понятия *управляющее состояние* (по крайней мере, в контексте программных и программно-аппаратных систем). Если в описании поведения системы упоминаются несколько режимов, то каждому из них наверняка будет соответствовать отдельное состояние.

2. В данном случае целесообразно спроектировать событийную систему. Автомат будет обрабатывать четыре события, соответствующие нажатию трех различных кнопок и срабатыванию минутного таймера. На графе переходов (рис. 2.4) для обозначения трех первых событий используются названия кнопок (рис. 1.3), а для четвертого – символ «Т».

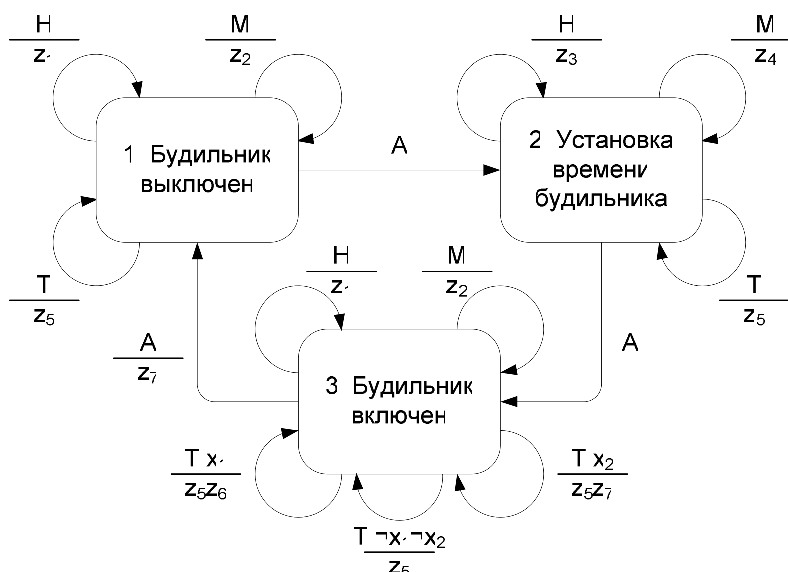

Рис. 2.4. Управляющий автомат эмулятора часов с будильником

Переходы между состояниями осуществляются по нажатию кнопки «А». При возникновении других событий смены состояний не происходит, но выполняются соответствующие действия (изменение текущего времени или времени срабатывания будильника, включение и отключение звонка). Эти действия обозначим выходными переменными z_1–z_7.

В качестве альтернативы событиям, можно было бы ввести три входные переменные, принимающие значения истина или ложь в зависимости от того, нажата ли соответствующая кнопка. Такое решение усложнило бы автомат, поэтому и был выбран событийный подход. В событийном варианте остается только две входных переменных x_1 и x_2, которые позволяют сравнивать текущее время со временем срабатывания будильника.

3. Входным переменным автомата, введенным на предыдущем шаге, сопоставим запросы: x_1 – «Превышает ли время срабатывания будильника на одну минуту текущее время?», x_2 – «Совпадает ли текущее время со временем срабатывания будильника?». Выходным переменным сопоставим команды: z_1 – «Увеличить число часов текущего времени», z_2 – «Увеличить число минут текущего времени», z_3 – «Увеличить число часов времени срабатывания будильника», z_4 – «Увеличить число минут времени срабатывания будильника», z_5 – «Увеличить текущее время на минуту», z_6 – «Включить звонок», z_7 – «Выключить звонок». Перечисленные запросы и команды достаточно просты и не требуют дальнейшей конкретизации.

4. Наконец, введем четыре целочисленные переменные: «число часов», «число минут», «число часов будильника», «число минут будильника». Таким образом, совокупность значений текущего времени и времени срабатывания будильника определяет текущее вычислительное состояние системы.

После выполнения указанных выше шагов начинается стадия реализации эмулятора часов. Подпрограммы самого низкого уровня абстракции реализуются в терминах переменных, хранящих вычислительное состояние. Реализация автомата становится главной программой[6]. В большинстве случаев эту программу можно сгенерировать автоматически из более высокоуровневого и наглядного представления автомата (подробнее о технологии реализации в разд. 2.3).

В приведенном примере логика поведения эмулятора часов с будильником описана с помощью автомата Мили. Выбор этой автоматной модели сделан на основе постановки задачи: из словесного описания поведения часов следует, что они совершают различные действия в зависимости от возникающих событий. В общем случае в программировании с явным выделением состояний могут применяться автоматы Мура, Мили или смешанные автоматы.

Изложенный метод проектирования имеет ряд преимуществ перед традиционной функциональной декомпозицией сверху вниз. Помимо общих достоинств, характерных для любого «автоматного» метода (таких как наглядное представление логики сложного поведения), явное выделение состояний попутно привело к частичному решению ряда проблем традиционного процедурного программирования. Один из основных недостатков метода сверху вниз состоит в том, что при проектировании в первую очередь задается вопрос «Что делает система?», а ведь именно этот ее аспект более всего подвержен изменениям [29]. В результате архитектура построенной таким образом системы обладает недостаточной расширяемостью.

В программировании с явным выделением состояний главный вопрос – «В каких состояниях может находиться система?» Множество управляющих состояний – более устойчивая характеристика системы, чем ее главная функция. Поэтому архитектура, построенная на основе управляющих состояний, является более расширяемой.

Кроме того, программирование с явным выделением состояний непосредственно использует концепцию события. В событийной архитектуре вообще нет понятия главной функции. Такая архитектура полностью соответствует современным представлениям о программной системе, предоставляющей некоторое множество услуг (сервисов).

Отметим, что в программировании с явным выделением состояний метод сверху вниз является лишь одной из альтернатив. Известное ограничение этого метода состоит в том, что его целесообразно применять только при проектировании всей системы от начала до конца – с нуля. Еще одно ограничение появляется из-за специфики задач, решаемых с помощью автоматного подхода. Объекты управления в разрабатываемых системах часто реализуются аппаратно. В этом случае к ним нецелесообразно

---

[6] В событийных системах говорить о главной программе не совсем корректно. В этом случае реализация автомата играет роль главной процедуры обработки событий. В программах для *Windows*, использующих *WinAPI*, такая процедура традиционно называется *WndProc*.

применять те же технологии проектирования, что и к программным компонентам системы, так как критерии оптимальности проектирования для аппаратных и программных компонент различны.

Таким образом, часто при проектировании системы со сложным поведением необходимо исходить из уже имеющихся объектов управления с определенным набором операций и заданного множества событий, которые могут возникать во внешней среде. По этой причине, хотя метод «сверху вниз» хорошо обоснован теоретически, на практике чаще применяется проектирование *от объектов управления и событий*:

1. Исходными данными задачи считается не только словесное описание целевого поведения системы, но и (более или менее) точная спецификация набора *событий*, поступающих в систему из внешней среды, и множеств *запросов* и *команд* всех объектов управления.

2. Строится набор *управляющих состояний*.

3. Каждому запросу объектов управления ставится в соответствие входная переменная автомата, каждой команде – выходная. На основе управляющих состояний, событий, входных и выходных переменных строится автомат, обеспечивающий заданное поведение системы.

В качестве примера рассмотрим проектирование системы управления клапаном [4]. Клапан – физический объект, созданный до того, как начала разрабатываться обсуждаемая программная система. Поэтому при проектировании системы управления необходимо исходить из имеющихся способов взаимодействия ее с клапаном, иначе говоря, необходимо применить метод «от объектов управления и событий».

Проектируемая система относится к области логического управления. В этой области принято описывать взаимодействие с объектом управления в терминах *исполнительных механизмов* (совершающих определенные действия при подаче на них управляющего сигнала) и *сигнализаторов* (подающих сигнал, если объект управления находится в определенном вычислительном состоянии). В терминологии этой книги исполнительным механизмам соответствуют команды, а сигнализаторам – запросы объекта управления.

В соответствии с традициями логического управления не будем использовать в системе события, а среди всех автоматных моделей отдадим предпочтение автомату Мура.

Итак, предположим, что аппаратная часть системы содержит клапан с памятью, снабженный исполнительными механизмами открытия и закрытия, а также сигнализаторами открытого и закрытого положений. Кроме того, в системе имеются три кнопки без памяти (после нажатия кнопки возвращаются в исходное состояние) и три индикатора. Пусть задано следующее словесное описание поведения системы:

- в исходном состоянии клапан закрыт, горит индикатор «Закрыт»;
- при нажатии кнопки «Открыть» клапан начинает открываться;
- после его открытия срабатывает сигнализатор открытого положения, зажигается индикатор «Открыт» и управляющий сигнал с клапана снимается;

- при нажатии кнопки «Закрыть» клапан начинает закрываться;
- после его закрытия срабатывает сигнализатор закрытого положения, зажигается индикатор «Закрыт» и управляющий сигнал с клапана снимается;
- если в течение трех секунд клапан не откроется или не закроется, то управляющий сигнал с клапана снимается и зажигается индикатор «Неисправность»;
- сброс сигнала с индикатора «Неисправность» осуществляется нажатием кнопки «Разблокировка».

Опишем процесс проектирования системы управления клапаном.

1. Как было упомянуто выше, события в этой системе не используются. Перечислим запросы и команды всех имеющихся объектов управления и сопоставим им входные и выходные переменные автомата.

   Каждой имеющейся в системе кнопке («Открыть», «Закрыть» и «Разблокировка») соответствует запрос, возвращающий значение «истина», если кнопка нажата. Этим трем запросам поставим в соответствие входные переменные автомата x_1, x_2 и x_3. У клапана имеется два сигнализатора – открытого и закрытого положений. Этим сигнализаторам сопоставим входные переменные x_4 и x_5.

   Двум исполнительным механизмам клапана (открывающему и закрывающему) сопоставим выходные переменные автомата z_1 и z_2. Будем считать, что индикаторы «Открыт» и «Закрыт» напрямую связаны с сигнализаторами открытого и закрытого положений клапана соответственно. Работа этих индикаторов не требует участия управляющего автомата. Третий индикатор («Неисправность») управляется автоматом. Подаче сигнала на этот индикатор сопоставим выходную переменную z_3.

   Кроме того, для измерения заданного в условии задачи интервала времени (три секунды) введем дополнительный объект управления – *элемент задержки* (*таймер*). У таймера имеется один исполнительный механизм и один сигнализатор. В момент подачи сигнала на исполнительный механизм, таймер включается и через заданный промежуток времени активируется его сигнализатор (таймер срабатывает). Будем считать, что повторная подача сигнала после включения таймера, но до его срабатывания, не производит никакого эффекта. Включению таймера сопоставим выходную переменную z_4, а его срабатыванию – входную переменную x_6.

2. Построим множество управляющих состояний, взяв за основу набор качественно различных состояний клапана, как устойчивых («Открыт» и «Закрыт»), так и неустойчивых («Открывается» и «Закрывается»). Исходя из описания поведения системы, добавим к этому набору еще одно чисто логическое состояние – «Неисправность» (рис. 2.5).

Рис. 2.5. Состояния системы управления клапаном

3. Поскольку в автомате Мура выходное воздействие зависит только от состояния, удобно сразу добавить описания выходных воздействий в вершины графа (рис. 2.6), а потом уже определять условия переходов между состояниями. Выходные воздействия можно записывать в виде битовых векторов, составленных из значений выходных переменных. Например, в такой записи воздействие «0101» означает «Открыть клапан и запустить таймер». Видимо, более наглядным является другой способ записи, когда в вершине перечисляются только те выходные переменные, значение которых «истина». Например, то же самое воздействие «0101» можно записать как z_2 z_4.

Рис. 2.6. Выходные воздействия системы управления клапаном

Теперь добавим переходы между состояниями, пометив их соответствующими условиями (рис. 2.7).

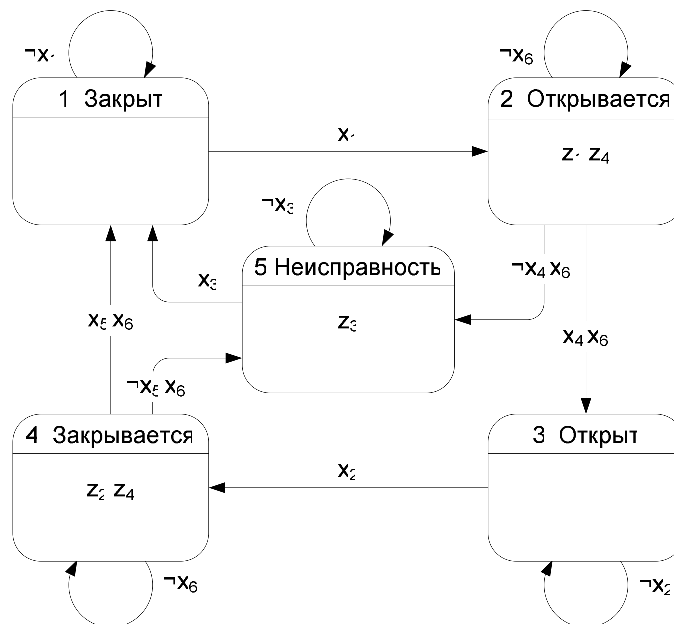

Рис. 2.7. Автомат системы управления клапаном

Условия на переходах могут иметь вид произвольных булевых формул, состоящих из входных переменных. Запись условия в виде формулы эквивалентна записи в виде множества входных воздействий, на которых эта формула принимает значение «истина». Например, в рассматриваемой системе формула

$$x_1 \wedge x_4 \wedge \neg x_6$$

эквивалентна множеству входных воздействий

$$\{100100,\ 100110\ 101100,\ 101110,\ 110100,\ 110110,\ 111100,\ 111110\}.$$

Конечно, запись в виде булевой формулы гораздо более удобна и наглядна. На графе переходов символ операции логического «И» обычно опускается для краткости.

Отметим одно важное отличие данного примера от предыдущего (эмулятора часов с будильником). Предыдущая система была событийной, очередной такт работы автомата инициировался в ней исключительно возникновением события. Другими словами, автоматизированный объект, спроектированный в предыдущем примере, был *пассивным*. Система управления клапаном, напротив, *активна*. Здесь автомат работает не «тогда, когда что-то произошло», а непрерывно, такт за тактом. Даже на неустойчивые состояния объекта управления, такие как «Открывается» и «Закрывается», может приходиться большое число тактов работы автомата. Поэтому в данной задаче во всех вершинах графа переходов имеются петли, назначение которых – поддерживать автомат в одном и том же состоянии до тех пор, пока не выполнится условие

перехода в какое-либо другое состояние. Условие на петле строится как отрицание дизъюнкции условий на всех остальных исходящих дугах.

2.1.2. Программные системы, управляемые взаимодействующими автоматами

Применяя метод «сверху вниз» или метод «от объектов управления и событий», можно спроектировать в автоматном стиле небольшую систему со сложным поведением. Однако с ростом размера системы использование изложенных выше подходов становится затруднительным. В чем же их недостаток?

Вспомним формулировку парадигмы автоматного программирования, приведенную в разд. 1.3. Эта парадигма состоит в представлении сущностей со сложным поведением в виде автоматизированных объектов управления. В большой системе сущностей со сложным поведением обычно много. Следовательно, часть системы, обладающая сложным поведением, должна быть представлена в виде *множества взаимодействующих автоматизированных объектов управления*. Соответствуют ли изложенные методы проектирования заявленной парадигме? Не совсем.

При использовании любого из двух перечисленных выше подходов система представляется в виде *автомата, управляющего множеством объектов управления*.

Более того, четкая структура множества объектов управления существует, как правило, только в программно-аппаратных системах. Каждый сигнализатор или исполнительный механизм в такой системе относится к одному конкретному объекту управления и не имеет непосредственного доступа к состоянию других объектов.

В полностью программных системах множество объектов управления не предопределено, и оно выявляется при анализе предметной области. Однако на стадиях проектирования и реализации эта информация теряется: в соответствии с традициями процедурного программирования все запросы и команды в системе равноправны, они могут напрямую обращаться ко всем данным системы (переменным, описывающим ее вычислительное состояние).

С некоторыми оговорками можно утверждать, что программная система, спроектированная с использованием любого из указанных выше подходов, всегда состоит ровно из *одного* автоматизированного объекта, содержащего один объект управления и один автомат.

Конечно, такое решение применимо только для очень простых и небольших систем. С ростом размера системы обычно растет сложность, как объекта управления, так и автомата. Если со сложностью объекта управления помогает справиться традиционная функциональная декомпозиция, то для борьбы со сложностью автомата приходится вводить специальную *автоматную* декомпозицию и вносить изменения в методы проектирования.

Таким образом, на вызов, связанный с ростом сложности программных систем, процедурное программирование с явным выделением состояний отвечает предложением представлять системы в виде *множества взаимодействующих автоматов, совместно управляющих множеством объектов управления*.

ПРИМЕЧАНИЕ

Современные социологи считают, что форма семейных отношений в человеческом обществе прошла эволюцию от неограниченной *полигамии* (когда каждый член общества мог иметь детей от любого числа людей противоположного пола) через *полигинию* или *гаремную семью* (когда мужчина мог вступить в брак с несколькими женщинами, однако каждая женщина имела не более одного мужа) к *моногамии* (в которой ячейкой общества является семья из двух человек противоположного пола). Моногамия на сегодняшний день считается наиболее развитой и здоровой формой семейных отношений и, более того, является единственной законной формой семьи в большинстве стран мира [30].

Автоматное программирование в процессе своего развития повторяет ошибки человечества, но в другом порядке (рис. 2.8). Начав с гарема (один автомат, управляющий множеством объектов управления), мы пришли к полигамии (множество автоматов, совместно управляющих множеством объектов управления). В главе 3 станет ясно, как принципы объектной технологии приведут к современной формулировке парадигмы автоматного программирования и провозглашению в автоматном обществе ценностей здоровой моногамной семьи – автоматизированного объекта управления.

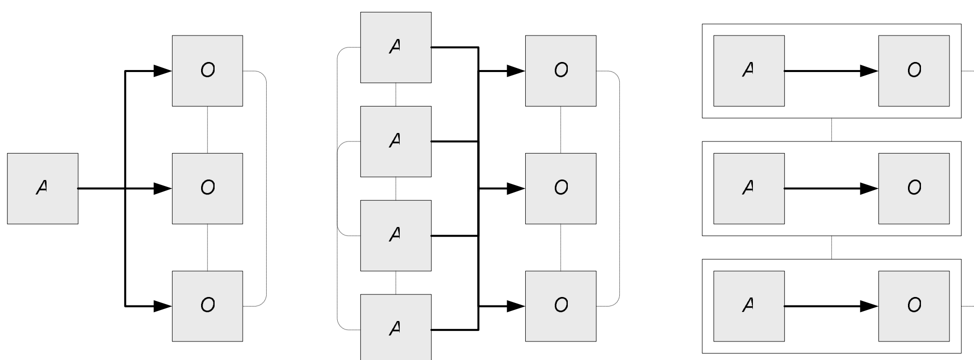

Рис. 2.8. Слева направо: путь к современной парадигме автоматного программирования

Результатом традиционной декомпозиции сверху вниз является дерево подпрограмм. Если применить те же принципы к декомпозиции логики системы, получим дерево (или иерархию) автоматов. С точки зрения архитектуры системы в целом, это означает, что автоматы могут находиться не только на самом верхнем уровне абстракции, но и на нескольких следующих уровнях (рис. 2.9).

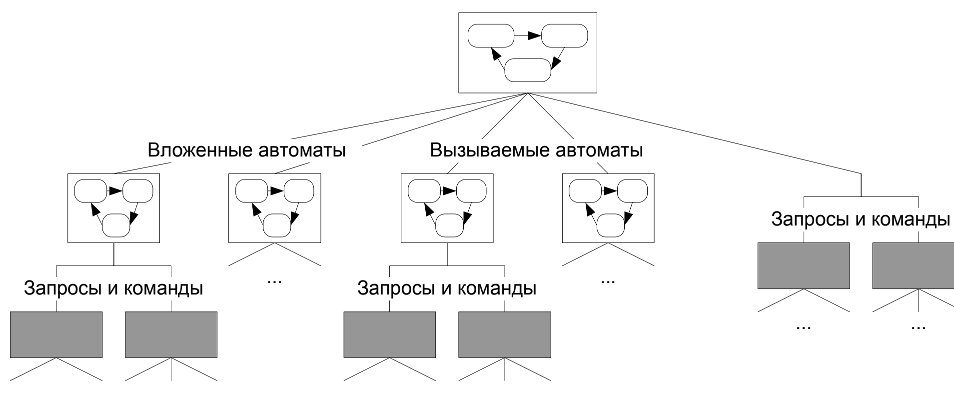

Рис. 2.9. Архитектура системы, управляемой множеством автоматов

В традиционном процедурном программировании существует только один вид отношений между подпрограммами, находящимися на соседних уровнях абстракции: более конкретная подпрограмма является *вызываемой* по отношению к более абстрактной (иначе говоря, вторая *вызывает* первую). В программировании с явным выделением состояний принято различать две разновидности отношений между автоматами: подчиненный автомат по отношению к вышестоящему может быть или *вызываемым* или *вложенным* [27].

Обращение к *вызываемому* автомату подобно обыкновенному синхронному вызову подпрограммы, в том смысле, что вызываемому автомату передается управление. При каждом вызове автомат начинает работу в начальном состоянии. Лишь по окончании его работы (после перехода в конечное состояние) управление вновь возвращается к вызывающему автомату, и тот возобновляет свою работу. Сколько шагов (тактов) совершит вызываемый автомат, заранее неизвестно. Более того, он может никогда не перейти в конечное состояние, и, таким образом, управление никогда не будет передано обратно вызывающему автомату.

Подобно подпрограмме, которой при вызове передаются аргументы, автомат может быть вызван с некоторым событием. Это событие будет обработано вызываемым автоматом в первую очередь. Все последующие события, возникающие в системе в процессе работы этого автомата, будут обрабатываться только им.

Обращение к *вложенному* автомату инициирует один такт его работы. Между последовательными обращениями состояние вложенного автомата сохраняется (по этому признаку вложенные автоматы подобны сопрограммам, однако, аналогия не очень точная, поскольку сопрограммы сами решают, когда приостановить работу, а вложенному автомату разрешается сделать только один шаг). Вложенному автомату нельзя передать произвольное событие, им будет обработано событие, существующее в системе на момент обращения к нему. Таким образом, одно событие, возникшее в системе, может быть обработано объемлющим автоматом и несколькими вложенными.

Различие между вызываемым и вложенным автоматом по отношению к объемлющему автомату аналогично различию между активной и пассивной автоматными моделями по отношению к внешней среде.

В большинстве случаев в системе целесообразно использовать либо только вложенные автоматы, либо только вызываемые – это делает логику системы более понятной. Формально говоря, эти механизмы взаимозаменяемы: каждый из них можно эмулировать с помощью другого. При этом отметим, что для эмуляции сохранения состояния между обращениями к вложенному автомату может потребоваться ввести дополнительную переменную в объект управления. Практически же, обычно несложно определить, какой из механизмов более удобен в данной задаче при выбранном критерии автоматной декомпозиции (ниже будет приведен пример влияния критерия декомпозиции на выбор того или иного вида отношений между автоматами).

С другой стороны, при использовании обоих видов отношений совместно возникают семантические неопределенности. Например, если автомат A_3 вложен в автомат A_2, который вызывается из автомата A_1, должен ли A_3 сохранять свое состояние между вызовами A_2? На подобные вопросы автоматное программирование универсальных ответов не дает. Поэтому разработчикам, которые решаются использовать механизмы вложенности и вызываемости в одной системе, необходимо оговаривать семантику их сочетания.

К вложенным и вызываемым автоматам можно обращаться везде, где разрешено формировать выходные воздействия: в состояниях и на переходах. На практике к вложенным автоматам чаще всего обращаются в состояниях.

Если в некоторое состояние вложен автомат и по совершении им шага объемлющий автомат остается в том же состоянии (переходит по петле), снова происходит обращение к этому же вложенному автомату. Таким образом, в этом случае вложенный автомат работает почти так же, как если бы он был вызываемым, за тем исключением, что вызываемый автомат заканчивает работу только по собственной инициативе, а вложенный – как только после очередного шага выполнится условие перехода объемлющего автомата в другое состояние. Несколько автоматов, вложенных в одно состояние, работают псевдопараллельно (совершают шаги по очереди).

Автоматы в системе могут обмениваться информацией посредством передачи события как аргумента при вызове автомата и через общий объект управления. Последний механизм подобен взаимодействию потоков в параллельной архитектуре с общей памятью или обмену информацией между подпрограммами через глобальные переменные. Этот механизм очень мощный, он мог бы быть единственным способом взаимодействия автоматов. Однако он имеет те же серьезные недостатки, что и глобальные переменные: отсутствие явного интерфейса межмодульного взаимодействия и, как следствие, чрезмерная зависимость между модулями и неспособность системы эволюционировать. Кроме того, в рассматриваемом случае имеется еще и специфический «автоматный» недостаток. Информация, которой обмениваются автоматы, имеет управляющую, логическую природу. В соответствии с принципами автоматного программирования такую информацию не следует смешивать с вычислительным состоянием объекта

управления. По этим причинам взаимодействием через общий объект управления не следует злоупотреблять[7].

Иногда передачи событий вызываемым автоматам оказывается недостаточно, и возникает потребность в дополнительных механизмах обмена информацией. Механизм, который исторически появился первым (еще до передачи событий) и широко применяется в задачах логического управления – это *обмен номерами состояний* [4, 17]. При использовании этого механизма в условиях переходов могут участвовать не только события и входные переменные, но и состояния других автоматов системы. Иначе говоря, условие перехода может содержать предикат вида «автомат i находится (не находится) в состоянии j». Этот механизм прост в реализации, решает задачу разделения управляющей и вычислительной информации. Он идеален для задач логического управления, однако в больших программных системах его применять не рекомендуется, так как он требует доступа к внутренней структуре автомата, а это усиливает межмодульные зависимости.

В событийных системах логично предложить другой способ взаимодействия: разрешить автоматам (а не только внешней среде) *инициировать события*. Инициированное событие становится известным всей системе и будет обработано тем автоматом, который будет выполняться непосредственно после инициации. Этот механизм является обобщением вызова автомата с событием.

Наиболее мощный механизм взаимодействия между автоматами – это *обмен сообщениями*. Сообщения подобны событиям, но кроме уникального идентификатора они могут содержать дополнительную информацию любого объема и структуры. Обычно механизм обмена сообщениями поддерживает отправку сообщения одному конкретному автомату, группе автоматов или всей системе [31, 32]. За удобство этого механизма приходится платить усложнением его программной реализации.

Рассмотрим, как автоматная декомпозиция включается в процесс автоматного проектирования на примере метода «от объектов управления и событий» (изменения в методе проектирования «сверху вниз» аналогичны):

1. Исходные данные – это по-прежнему словесное описание поведения системы а также спецификация событий и объектов управления.
2. Строится набор наиболее абстрактных управляющих состояний системы. Например, у системы управления микроволновой печью такими состояниями могут быть «Выключено» и «Включено» (рис. 2.10), у автопилота самолета – «На земле», «Взлет», «Круиз» и «Посадка».

[7] Этот вид взаимодействия всегда остается, пока у нескольких автоматов есть общий объект управления. Злоупотребление здесь означает введение в объект управления «лишних» переменных, которых относятся больше к управлению, чем к вычислительному состоянию.

Рис. 2.10. Состояния системы управления микроволновой печью на верхнем уровне абстракции

3. На основе набора управляющих состояний строится *головной* автомат системы. Если в выходных воздействиях головного автомата участвуют только выходные переменные (соответствующие командам объектов управления), то автоматная декомпозиция не требуется, и процесс проектирования на этом завершается. В общем случае в описании автомата будут присутствовать *абстрактные действия*, не соответствующие никаким командам объектов управления. Например, автомат, управляющий микроволновой печью, в состоянии «Включено» должен обеспечивать «Приготовление пищи без вреда для здоровья пользователя» (рис. 2.11). Это абстрактное действие, требующее конкретизации.

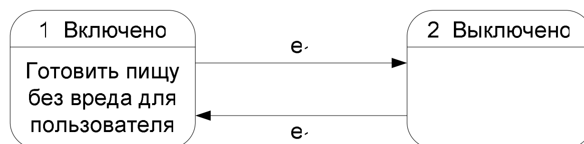

Рис. 2.11. Головной автомат системы управления микроволновой печью с абстрактным действием (событие e_1 соответствует нажатию кнопки «Вкл/Выкл»)

4. Каждое абстрактное действие необходимо конкретизировать, заменив его обращением к одному или нескольким вложенным или вызываемым автоматам. При построении этих автоматов также могут использоваться абстрактные действия (однако уровень их абстракции должен быть ниже, чем у исходного).

5. Конкретизация (шаг 4) повторяется до тех пор, пока в описаниях автоматов системы остаются абстрактные действия. По окончании этого шага в качестве компонентов выходных воздействий всех автоматов могут выступать только выходные переменные и обращения к уже описанным вложенным или вызываемым автоматам.

Как и другие виды декомпозиции, применяемые при проектировании ПО, автоматная декомпозиция – это творческая задача, которая может быть решена различными способами даже для одной конкретной системы. Невозможно предложить алгоритм автоматной декомпозиции, который подходил бы для любой задачи. Однако существуют *критерии автоматной декомпозиции*, следуя которым в большинстве случаев можно получить логичную архитектуру системы.

- Декомпозиция *по режимам* уместна тогда, когда в поведении системы можно выделить несколько качественно различных режимов (каждый из которых при необходимости можно конкретизировать, выделив режимы более низкого уровня абстракции). В этом случае логично сопоставить автомат каждому из режимов, в которых поведение системы является сложным, или, иными словами, сопоставить отдельный автомат каждому абстрактному действию.

Например, в системе управления микроволновой печью действию «Готовить пищу без вреда для здоровья» сопоставим вложенный автомат A_2. Предположим, что у проектируемой печи есть два режима приготовления пищи: с помощью микроволн и на гриле. Тогда у автомата A_2 логично выделить два состояния («Микроволны» и «Гриль»), в которых он должен выполнять абстрактные действия «Готовить пищу с помощью микроволн» и «Готовить пищу на гриле», соответственно. Следуя предложенному методу декомпозиции, каждому из этих абстрактных действий, в свою очередь, сопоставим отдельный автомат: A_3 и A_4 (рис. 2.12), и т. д.

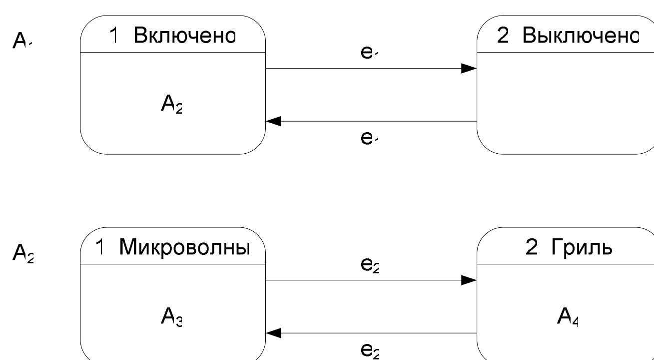

Рис. 2.12. Два верхних уровня абстракции системы управления микроволновой печью

- Декомпозиция *по объектам управления* применима в том случае, когда в системе присутствует несколько объектов управления. В этом случае логично поручить управление каждым из объектов отдельному автомату или поддереву в иерархии автоматов (рис. 2.13).

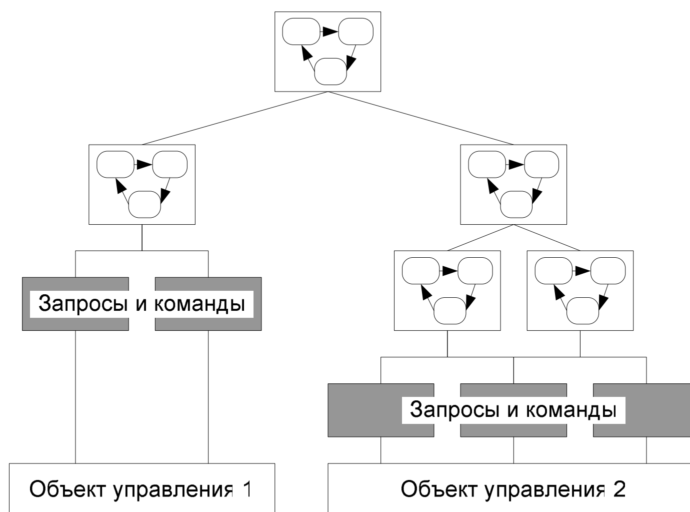

Рис. 2.13. Автоматная декомпозиция по объектам управления

Следование такому критерию декомпозиции приводит к выделению в архитектуре системы пар автомат – объект управления (или группа автоматов –

объект управления) и значительно приближает ее к «идеалу» парадигмы автоматного программирования – *множеству взаимодействующих автоматизированных объектов управления*.

Можно сказать, что концепция автоматной декомпозиции по объектам управления стала источником тех идей, которые легли в основу *объектно-ориентированного программирования с явным выделением состояний* (речь о котором будет идти в главе 3), а также самого понятия автоматизированного объекта управления и формулировки парадигмы автоматного программирования в том виде, как они изложены в этой книге.

Возвращаясь к процедурному программированию с явным выделением состояний, отметим, что декомпозиция по объектам управления обычно приводит к большей модульности и расширяемости, чем декомпозиция по режимам. Однако последняя незаменима в тех случаях, когда объект управления только один и на текущем уровне абстракции нет возможности представить его в виде совокупности нескольких объектов[8].

Сравним два предложенных выше критерия автоматной декомпозиции на примере проектирования системы управления двумя клапанами. Отметим, что эта задача довольно проста, и ее не составит труда решить и с использованием одного автомата. Однако она позволяет продемонстрировать основные принципы автоматной декомпозиции. Конечно, на практике декомпозиция применяется для решения гораздо более сложных задач, и дает более очевидную выгоду в простоте и понятности архитектуры системы.

Эта задача, как и задача управления одним клапаном, рассмотренная в разд. 2.1.1, относится к области логического управления и имеет ту же специфику (отсутствие событий, использование автоматов Мура и т. д.).

Аппаратная часть системы содержит два клапана с памятью и две кнопки без памяти. Задано следующее словесное описание поведения системы:

- в исходном состоянии обе кнопки («Открыть клапаны» и «Закрыть клапаны») не нажаты, клапаны закрыты;
- при нажатии кнопки «Открыть клапаны» подается управляющий сигнал на открытие первого клапана;
- после открытия первого клапана (при срабатывании его сигнализатора открытого положения) с него снимается управляющий сигнал и начинает открываться второй клапан;
- после открытия второго клапана, с него снимается управляющий сигнал;
- когда оба клапана открыты, при нажатии кнопки «Закрыть клапаны» подается управляющий сигнал на закрытие второго клапана;

---

[8] Так, у микроволновой печи есть подобъекты: экран, замок дверцы, нагревательный элемент. Однако поведение на двух верхних уровнях абстракции (на уровне режимов «Включено» – «Выключено» и «Микроволны» – «Гриль») относится ко всей печи в целом, а не к отдельным ее составляющим.

- после закрытия второго клапана (при срабатывании его сигнализатора закрытого положения) с него снимается управляющий сигнал и начинает закрываться первый клапан;
- после закрытия первого клапана с него снимается управляющий сигнал, и система возвращается в исходное состояние.

Сопоставим кнопке «Открыть клапаны» входную переменную x_1, кнопке «Закрыть клапаны» – переменную x_2, сигнализаторам открытого и закрытого положений первого клапана – переменные x_3 и x_4 соответственно, сигнализаторам открытого и закрытого положений второго клапана – переменные x_5 и x_6 соответственно. Управляющим воздействиям на открытие и на закрытие первого клапана сопоставим выходные переменные z_1 и z_2, а для второго клапана – z_3 и z_4.

Попробуем сначала спроектировать систему, управляемую одним автоматом, без использования автоматной декомпозиции (рис. 2.14).

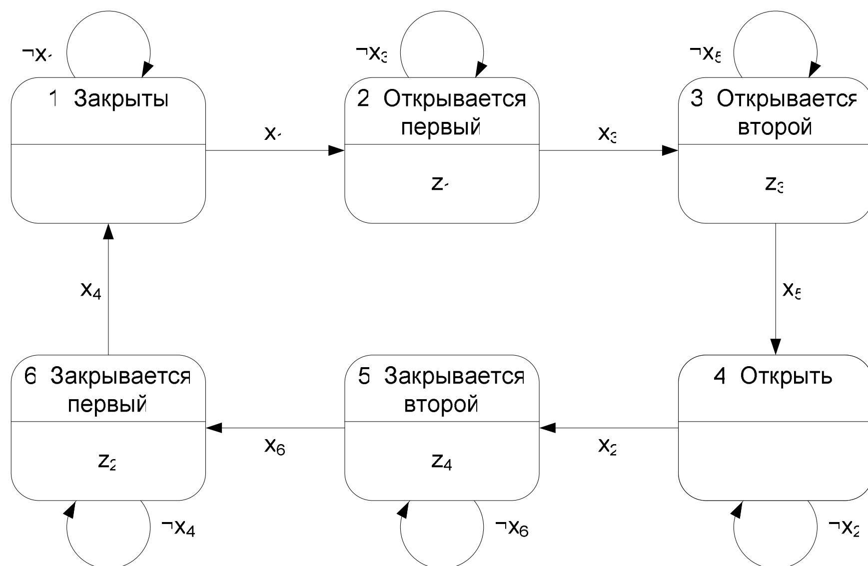

Рис. 2.14. Управление двумя клапанами с помощью одного автомата

Теперь попробуем заново спроектировать управляющую часть системы. На верхнем уровне абстракции можно выделить четыре состояния: «Закрыты», «Открываются», «Открыты» и «Закрываются». С использованием этих состояний построим *головной* автомат системы, в описании которого пока определены не все условия переходов и содержатся абстрактные действия, требующие конкретизации (рис. 2.15).

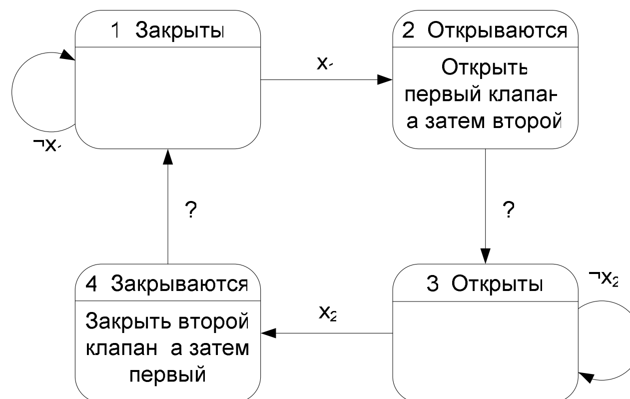

Рис. 2.15. Головной автомат системы управления двумя клапанами (в процессе проектирования)

Теперь настал момент выбрать критерий автоматной декомпозиции. Сначала попробуем применить метод декомпозиции по *режимам*. В соответствии с этим методом сопоставим отдельный автомат каждому абстрактному действию. Таким образом, один автомат (A_2) будет отвечать за открытие клапанов, а другой (A_3) – за их закрытие. Головному автомату, как всегда, присвоим идентификатор A_1.

Задача автоматов A_2 и A_3 – совершить определенную последовательность действий, выполнение которой не должно прерываться до ее завершения. В таком случае эти автоматы удобно сделать *вызываемыми* из соответствующих состояний головного автомата. Напомним, что вызываемым автоматам необходимы конечные состояния, для того чтобы они имели возможность по окончании работы вернуть управление вызывающему автомату. На рис. 2.16 для конечных состояний используется наиболее распространенное обозначение – так называемый «бычий глаз» [33] (круг с рамкой). Переходы головного автомата из второго состояния в третье и из четвертого в первое в этом случае можно сделать безусловными. Головной автомат сможет совершить каждый из этих переходов только по окончании работы соответствующего вложенного автомата, а это именно то поведение, которое требуется по условию задачи.

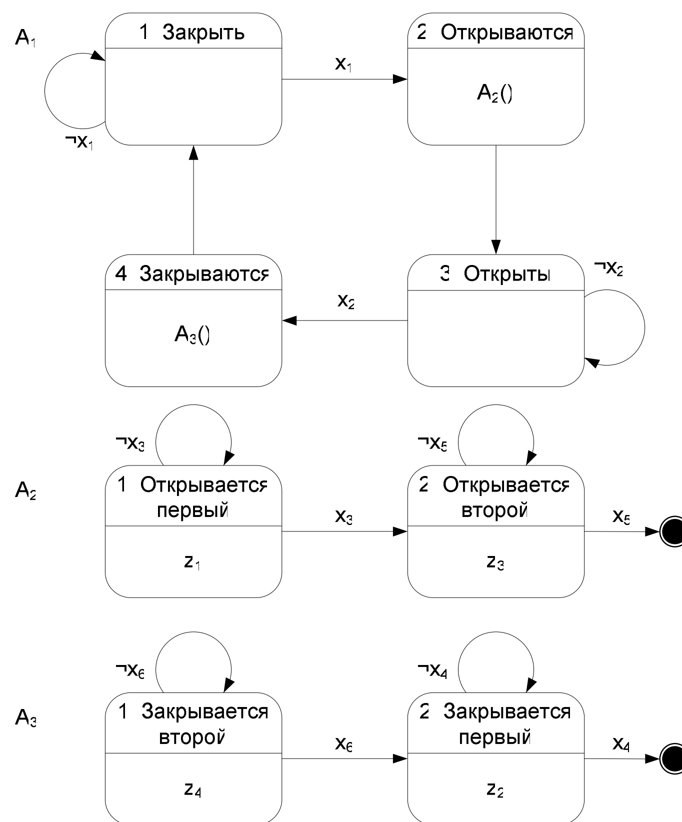

Рис. 2.16. Система управления двумя клапанами: декомпозиция по режимам

Отметим, что в графической нотации автоматного программирования нет общепринятых соглашений о различении вложенных и вызываемых автоматов. В этой книге принято соглашение, по которому при обращении к вложенному автомату просто записывается его идентификатор, а при обращении к вызываемому – к идентификатору добавляются круглые скобки, в которых при необходимости указывается передаваемое автомату событие. Такая запись аналогична синтаксису вызова подпрограмм в языках программирования семейства *C*.

Теперь попробуем применить другой критерий декомпозиции: *по объектам управления*. Построим отдельный управляющий автомат для каждого клапана. Оба этих автомата (обозначим их A_2 и A_3) теперь будут *вложены* в состояния два и четыре головного автомата. Для обеспечения корректной работы системы необходимо предоставить автоматам дополнительный способ обмена информацией. Поскольку проектируемая система относится к области логического управления, применим традиционный для этой области способ взаимодействия – обмен номерами состояний. Состояние автомата A_i на графе переходов принято обозначать символом y_i (рис. 2.17).

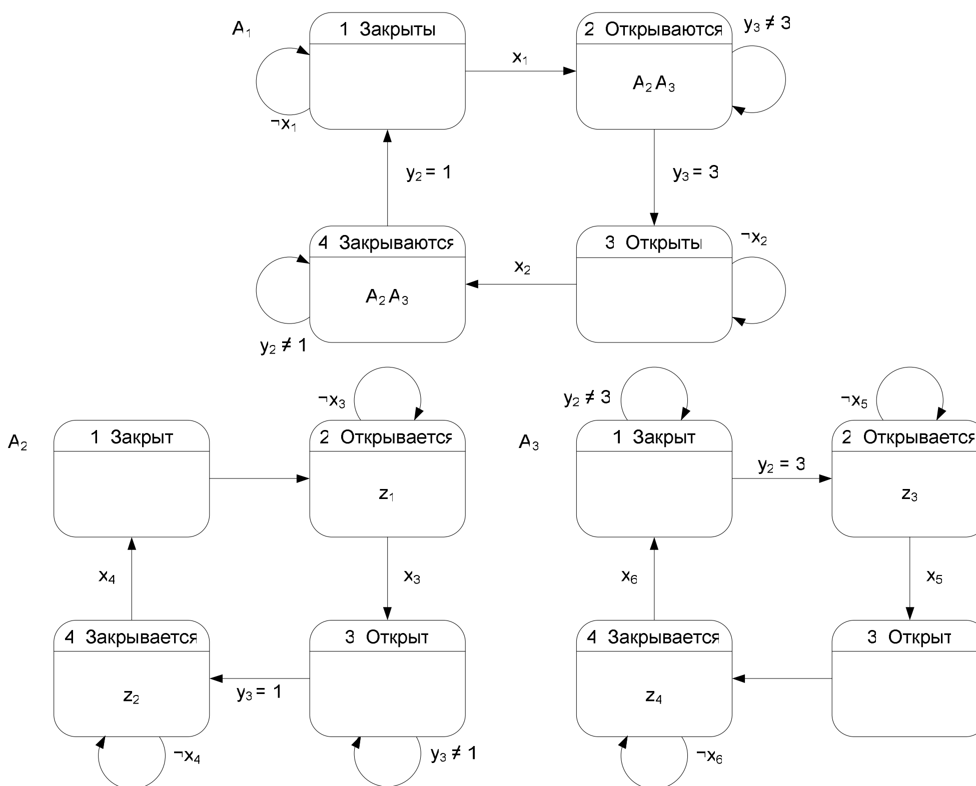

Рис. 2.17. Система управления двумя клапанами: декомпозиция по объектам управления

Система, построенная с помощью декомпозиции по объектам управления, получилась больше и сложнее для понимания. Ее сложность во многом объясняется использованием механизма обмена номерами состояний. Лучшим решением было бы ограничить интерфейс взаимодействия между автоматами, определив для автоматов A_2 и A_3 по два события, инициирующих открытие и закрытие клапана, и сделав эти автоматы вызываемыми (с передачей соответствующего события).

Если заменить обмен номерами состояний обменом событиями станет ясно, что автоматы A_2 и A_3 похожи с точностью до имен событий и переменных. Если же ослабить требование уникальности этих имен в пределах системы (потребовать их уникальности только в пределах каждого автомата), то автоматы A_2 и A_3 можно сделать совершенно идентичными. Поскольку их объекты управления (клапаны) также идентичны, получается, что в системе имеется два идентичных автоматизированных объекта, которые во время работы системы могут находиться в разных состояниях (как управляющих, так и вычислительных). В терминах объектно-ориентированного программирования можно сказать, что они являются экземплярами одного и того же класса.

Одна из сильных сторон декомпозиции по объектам управления, которая будет в полной мере раскрыта и использована в объектно-ориентированном программировании с явным выделением состояний, в том, что при наличии идентичных автоматизированных объектов достаточно описать только один из них

(точнее, описать класс таких объектов). Из этого описания во время выполнения системы можно создать сколько угодно экземпляров. Преимущества такого подхода очевидны. В частности, в случае рассматриваемой системы, если требования изменятся, и возникнет необходимость управлять пятью клапанами вместо двух, не придется добавлять описания еще трех автоматов.

У декомпозиции по объектам управления есть еще одно важное преимущество. Напомним, что метод автоматной декомпозиции, изложенный выше в данном разделе, был построен по аналогии с традиционным методом декомпозиции сверху вниз. Вследствие этой аналогии, множество автоматов, порождаемое изложенным методом, всегда имеет иерархическую структуру.

Однако опыт показывает, что при использовании декомпозиции по объектам управления роль головного автомата, который координирует работу автоматов, управляющих отдельными объектами, часто довольно незначительна. В этом случае его можно вообще исключить из системы, сохранив при этом логику поведения ценой небольших изменений в подчиненных автоматах. В результате получается децентрализованная система: уже не дерево, а множество параллельно работающих автоматов (рис. 2.18, слева) или, в более сложном случае, лес автоматов, деревья которого работают параллельно (рис. 2.18, справа). Такую разновидность автоматной декомпозиции будем называть *параллельной* (или *неиерархической*), в противоположность *иерархической* декомпозиции сверху вниз, рассмотренной ранее.

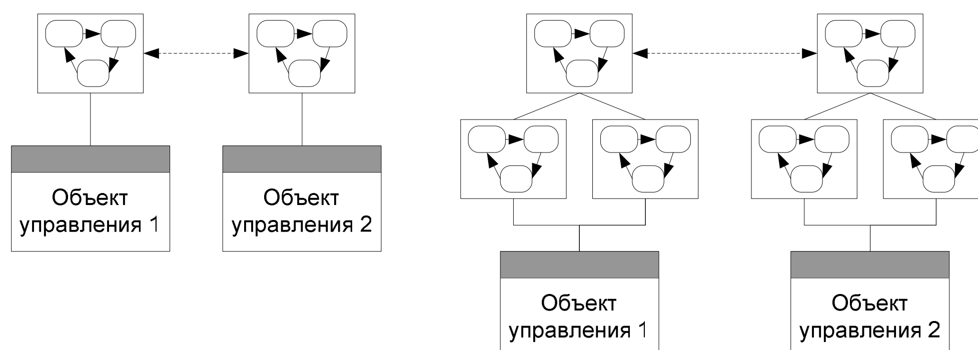

Рис. 2.18. Децентрализованная архитектура системы: параллельно работающие автоматы (слева) и параллельно работающие деревья автоматов (справа)

Децентрализация архитектуры системы особенно важна в задачах логического управления. Здесь каждый автомат или набор автоматов, управляющих одним объектом, часто удобно реализовать на отдельном вычислительном устройстве (логическом контроллере, микроконтроллере) и «встроить» в объект управления. В результате такой агрегации получается *автоматизированный объект управления* в первоначальном смысле этого термина. Поскольку устройства управления различными объектами разделены на аппаратном уровне и работают параллельно, то распределенная архитектура в виде множества параллельно работающих автоматов является для таких систем наиболее естественной.

В вычислительных задачах, отличных от задач логического управления, роль параллелизма и распределенных архитектур традиционно была менее значительной, однако в настоящее время она стремительно растет. Поэтому неиерархическая

автоматная декомпозиция может применяться и для систем, предназначенных для исполнения на персональном компьютере. Подробнее связь автоматного программирования и параллельных вычислений рассмотрена в разд. 4.3.

В качестве примера рассмотрим, как можно осуществить параллельную автоматную декомпозицию в системе управления двумя клапанами, рассмотренной выше. Поскольку головной автомат больше не требуется, система теперь состоит из двух управляющих автоматов (автомат A_1 управляется первым клапаном, автомат A_2 – вторым). Взаимодействие между ними, как и раньше, осуществляется путем обмена номерами состояний. Диаграммы переходов автоматов приведены на рис. 2.19.

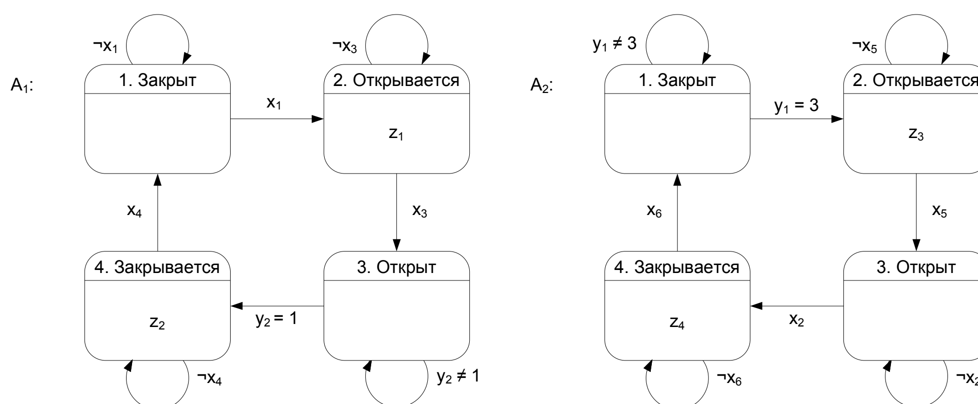

Рис. 2.19. Система управления двумя клапанами: параллельная декомпозиция

Сравнив рис. 2.17 и рис. 2.19, читатель может убедиться, что при переходе к децентрализованной архитектуре автоматы, управляющие клапанами, практически не усложнились.

2.2. Спецификация

2.2.1. Спецификация структуры

Как было упомянуто выше и продемонстрировано на нескольких примерах, одним из главных достоинств автоматного программирования является представление логики поведения системы в наиболее понятном и наглядном виде – с использованием графической нотации автоматов в виде *графов* (или *диаграмм*) *переходов*.

Однако не только логика поведения требует наглядного представления. Спецификация *структуры* системы является, наряду со спецификацией поведения, одним из двух основных результатов процесса проектирования и также требует подходящей нотации.

В примерах, приводимых до сих пор, структура системы (объекты управления, автоматы и связи между ними) описывалась словесно. Как могли убедиться читатели, такое описание громоздко и не наглядно. В программировании с явным выделением состояний для описания структуры системы используется графическая нотация *схемы связей* автоматов [4].

Схема связей строится отдельно для каждого автомата системы. На ней в виде прямоугольника изображается сам автомат (рис. 2.20). Слева от него изображаются *источники информации* (в событийных системах их также называют *поставщиками событий*) – сущности из системы или внешней среды, которые формируют входные воздействия автомата. Для каждого события, входной переменной или предиката с номером состояния между автоматом и источником информации проводится линия, помеченная идентификатором и словесным описанием этого события (переменной, предиката) [27]. Таким образом, схема связей предназначена, в том числе, и для расшифровки сокращенных идентификаторов событий и переменных, используемых на диаграмме переходов. Это позволяет изображать указанную диаграмму весьма компактно даже при наличии большого числа входных и выходных переменных. Такие диаграммы проще понимаются человеком, так как могут быть охвачены целиком одним взглядом.

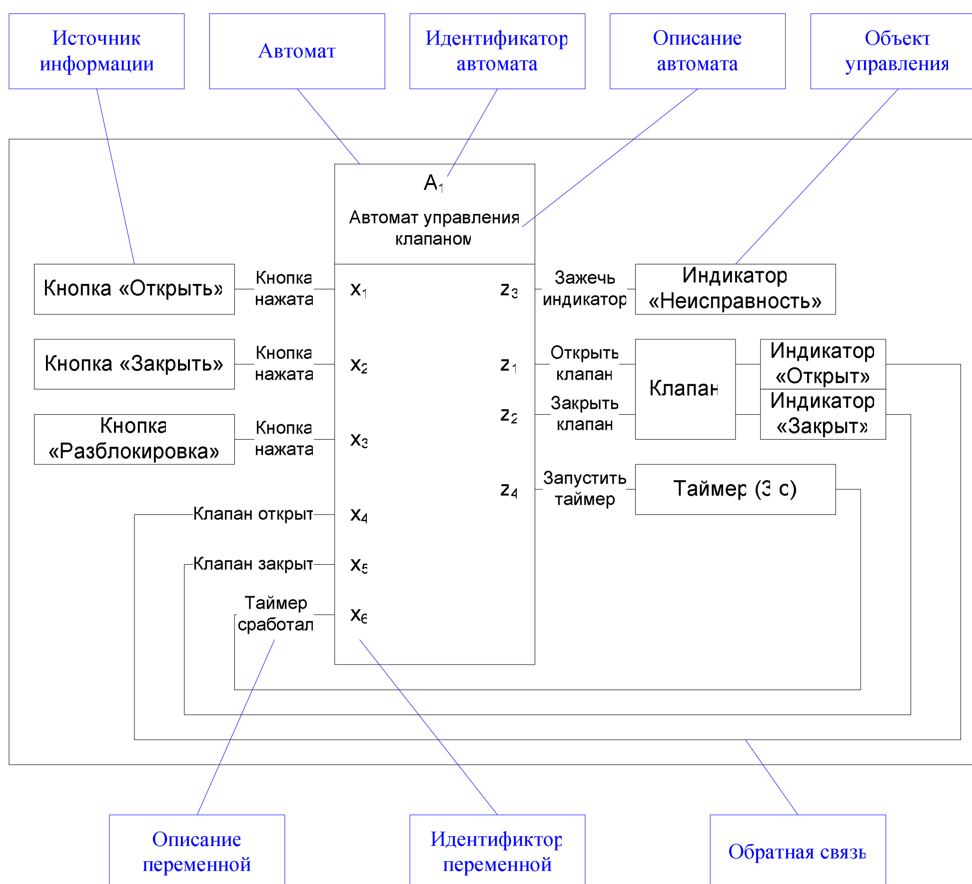

Рис. 2.20. Нотация схемы связей автомата

Справа от автомата изображаются его объекты управления и подчиненные (вложенные или вызываемые) автоматы. Для каждой выходной переменной между автоматом и соответствующим объектом управления проводится линия, помеченная идентификатором и описанием переменной. Если вызываемому автомату передается

одно или несколько событий, то для каждого из этих событий также проводится линия с пометками. Если объекты управления являются также и источниками информации – формируют часть входных переменных автомата, то они изображаются справа, а линии, соответствующие входным переменным, изображаются в виде обратных связей.

На рис. 2.20 изображена схема связей автомата, управляющего клапаном, процесс проектирования которого был описан в разд. 2.1.1.

Приведем схемы связей и для других примеров систем со сложным поведением из предыдущих разделов.

Один из вариантов схемы связей управляющего автомата часов с будильником изображен на рис. 2.21. Объект управления в этой системе (в отличие от системы управления клапаном) имеет чисто программную природу, поэтому его структура полностью определяется разработчиком.

Рис. 2.21. Схема связей управляющего автомата эмулятора часов с будильником

Рассмотрим теперь систему управления двумя клапанами, спроектированную с применением автоматной декомпозиции по объектам управления. В этой системе три управляющих автомата. Структуру системы естественно изобразить в виде трех схем связей (рис. 2.22).

ПРИМЕЧАНИЕ

При спецификации таких простых систем допустимо изображать все управляющие автоматы на одной, общей схеме связей, которая в этом случае отражает также и взаимодействие автоматов между собой. Однако решение с отдельной схемой для каждого автомата более масштабируемо и в общем случае следует использовать его.

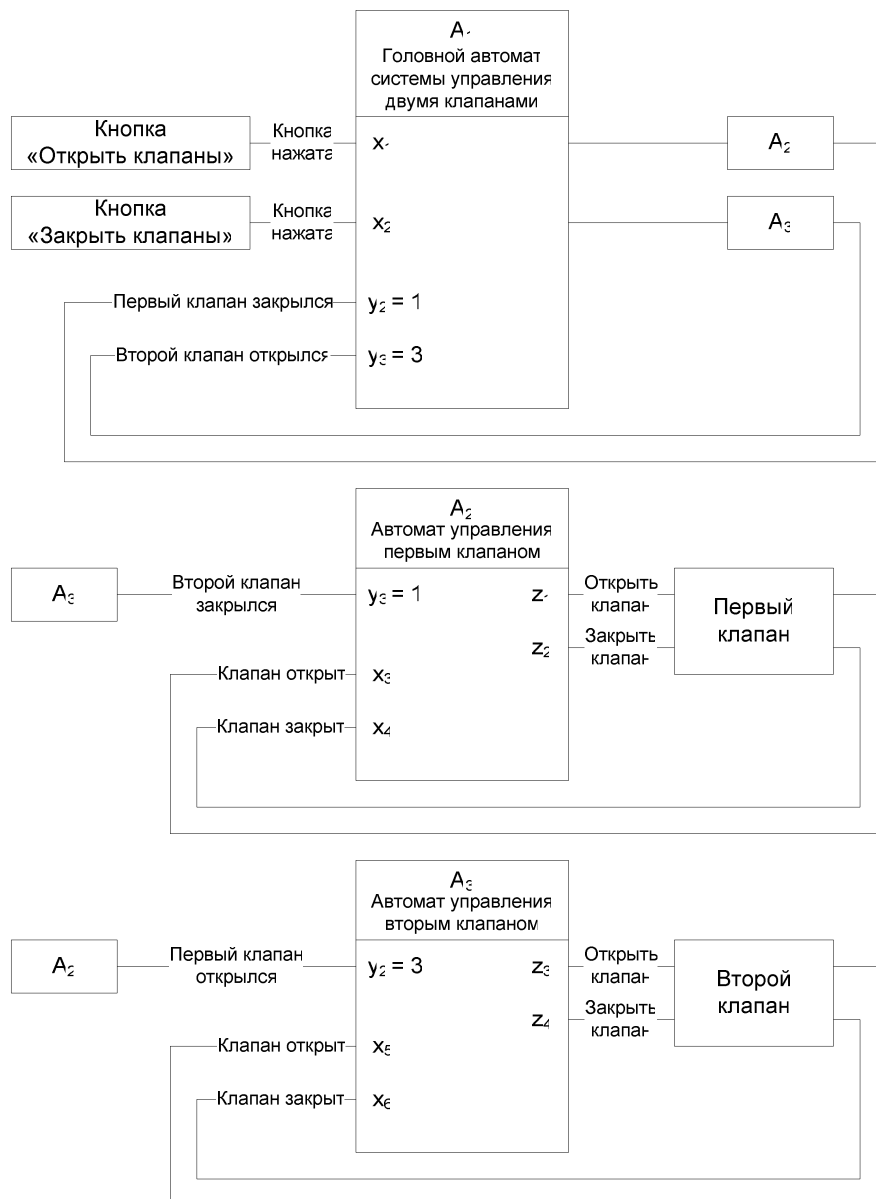

Рис. 2.22. Схемы связей трех автоматов системы управления двумя клапанами (декомпозиция по объектам управления)

Этот пример показывает, что схема связей позволяет наглядно отобразить взаимодействие автомата не только с объектами управления и элементами внешней среды, но и с другими автоматами системы. Автоматы A_2 и A_3 вложены в автомат A_1, и на схеме связей последнего они изображены справа, как объекты управления. В данной системе взаимодействие автоматов осуществляется путем обмена номерами состояний. Поэтому связи между автоматами системы помечены предикатами вида

$y_i = n$ и $y_i \neq n$, а также словесными пояснениями этих предикатов. Такая нотация позволяет сгладить недостатки механизма обмена номерами состояний и является шагом на пути к четко определенному интерфейсу взаимодействия между автоматами. Поскольку каждому предикату сопоставляется отдельная связь и определенный смысл, для перехода к событийной модели взаимодействия достаточно заменить каждый из предикатов отдельным событием.

Для небольших систем, управляемых множеством автоматов, схем связей вполне достаточно, для того чтобы наглядно описать взаимосвязи и обмен информацией между автоматами. Для систем бóльшего размера со сложными взаимосвязями может быть удобно построить *диаграмму взаимодействия* автоматов [27], на которой изображаются все автоматы системы и отношения вложенности и вызываемости между ними (рис. 2.23). При необходимости на такой диаграмме можно отобразить обмен номерами состояний, событиями или сообщениями между автоматами. Диаграмма взаимодействия автоматов может использоваться для получения представления о системе как о целом, в качестве разновидности *диаграммы потока данных*, а также для выявления специфических свойств (например, если рассматривать диаграмму взаимодействия как граф вызываемости, то контуры в нем могут приводить к зацикливанию).

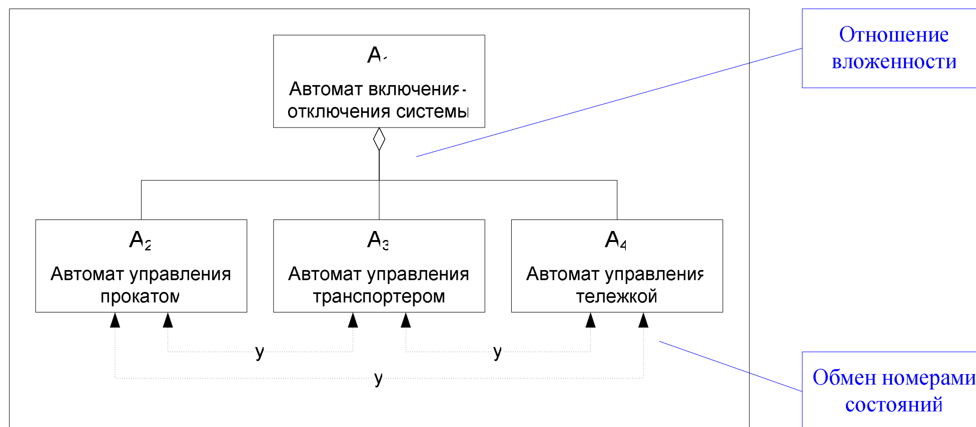

Рис. 2.23. Нотация диаграмм взаимодействия автоматов

2.2.2. Спецификация поведения

Познакомившись с языком спецификации структуры системы, вернемся к языку спецификации ее поведения: уже известным читателю графам переходов. Поскольку неформальное знакомство читателя на примерах с этой нотацией уже состоялось, в данном разделе опишем графы переходов более формально и уделим внимание тем деталям и возможностям, которые ранее не упоминались.

Диаграмма переходов автомата представляет собой ориентированный граф, вершинам которого соответствуют управляющие состояния, а дугам – переходы между состояниями. Вершины графа принято изображать в виде прямоугольников со скругленными углами, внутрь которых помещается номер, название состояния и описание выходного воздействия в этом состоянии. Выходное воздействие

описывается путем перечисления идентификаторов выходных переменных, вложенных или вызываемых автоматов.

Дуга графа переходов помечается условием перехода и выходным воздействием на переходе. Условие перехода имеет вид произвольной булевой формулы над входными переменными, событиями и (в случае взаимодействия путем обмена номерами состояний) предикатами от номеров состояний других автоматов. Громоздкими булевыми формулами бывает неудобно помечать переходы. В таком случае формулу можно заменить сокращенным идентификатором, а его расшифровку дать рядом с графом переходов. Условие перехода отделяется от выходного воздействия горизонтальной чертой. В целях повышения наглядности графов переходов необходимо по возможности выполнять их плоскую укладку.

На диаграммах переходов к переменным, событиям и автоматам системы принято обращаться через их краткие символические идентификаторы. Причина использования этого соглашения в том, что на больших диаграммах длинные, мнемонические имена либо вообще не помещаются, либо «засоряют» граф, мешая формированию у разработчика целостной картины поведения системы. Как уже упоминалось, расшифровка идентификаторов производится на схеме связей. Автоматизированное средство проектирования может избавить разработчика от необходимости обращаться к схеме связей, например, показывая словесное описание символа при наведении на него курсора мыши.

Обычно для автоматов используются идентификаторы вида A_i, для событий – e_i, для входных переменных – x_i, для состояний автоматов – y_i, для выходных переменных – z_i, для сокращенных условий переходов – C_i (где i – натуральное число). Можно использовать и любые другие идентификаторы, лишь бы они были короткими: из одной или двух букв. Так в примере с эмулятором часов события, соответствующие нажатиям кнопок, назывались буквами «H», «M» и «A». Обозначить их так было логично, так как по условию задачи теми же буквами назывались кнопки часов.

Одна из тех возможностей обсуждаемой нотации, которые не были до сих пор использованы ни в одном примере – это *группировка состояний*. Если несколько состояний автомата имеют одинаковые исходящие переходы (с одним и тем же условием, выходным воздействием и целевой вершиной), то такие состояния можно объединить в группу и заменить несколько одинаковых переходов одним – групповым. Группировку применяют в тех случаях, когда некоторый набор состояний логически объединен общим поведением, однако выделять эти состояния в отдельный автомат нецелесообразно. Графически группа состояний обозначается рамкой.

Другая неиспользованная возможность нотации – *переходы с приоритетами*. Она позволяет задать приоритеты на дугах, исходящих из некоторого состояния. Приоритеты определяют порядок, в котором проверяется возможность перехода по каждой из этих дуг (первым будет проверяться условие на дуге с высшим приоритетом, последним – на дуге с низшим приоритетом). Назначение приоритетов – один из способов борьбы с неоднозначностью (*противоречивостью*) в описании

автомата[9]. Граф переходов, в котором условия на нескольких дугах, исходящих из одного состояния, могут быть одновременно истинными, является некорректным. Во время выполнения возможна ситуация, в которой автомат, описываемый таким графом, не сможет определить, по какой из дуг ему перейти. Использование переходов с приоритетами разрешает это противоречие в пользу дуги с б*о*льшим приоритетом. Несколько исходящих дуг могут иметь одинаковый приоритет, но в этом случае между ними не должно быть противоречия.

На графе переходов приоритет обозначается натуральным числом перед меткой перехода. При этом по соглашению меньшие числа соответствуют более высоким приоритетам (иначе говоря, числа соответствуют порядку, в котором проверяются переходы).

Наконец, упомянем еще одно полезное соглашение в нотации диаграмм переходов. Допустим, имеется один или несколько переходов из некоторого состояния, таких, что противоречия между ними не возникает. Однако для обеспечения *полноты*[10] требуется добавить еще один переход, который осуществится в том случае, если ни одно из условий других переходов не будет выполнено. Новую дугу следовало бы пометить отрицанием дизъюнкции условий, написанных на всех остальных исходящих дугах. Такая неприятная необходимость уже встречалась: вспомните петли в автоматах, управляющих клапанами.

Громоздкие условия, обеспечивающие полноту, не только скучно и неприятно писать, они еще и покушаются на святая святых автоматного программирования – наглядность графов переходов. Вместо этого можно было бы ввести безусловный переход и присвоить ему более низкий приоритет, чем у всех остальных. Однако вводить приоритеты для всех остальных дуг (если их не было) неудобно. Кроме того, при модификации автомата может потребоваться изменять приоритет дуги, обеспечивающей полноту, поскольку он всегда должен быть наименьшим. Поэтому для обозначения такой дуги в нотации диаграмм переходов введена специальная конструкция: пометка «иначе». Переход с такой пометкой всегда выполняется, только если условия всех остальных переходов из данного состояния ложны. Это его свойство сохраняется при добавлении любых новых дуг и любом изменении приоритетов.

На рис. 2.24 приведен пример графа переходов, в котором использованы почти все возможности нотации.

---

[9] Множество условий переходов, исходящих из одного состояния, называется *противоречивым*, если при некотором значении входного воздействия более одного условия может иметь значение «истина».

[10] Множество условий переходов, исходящих из одного состояния называется *неполным*, если при некотором значении входного воздействия ни одно из условий не имеет значения «истина».

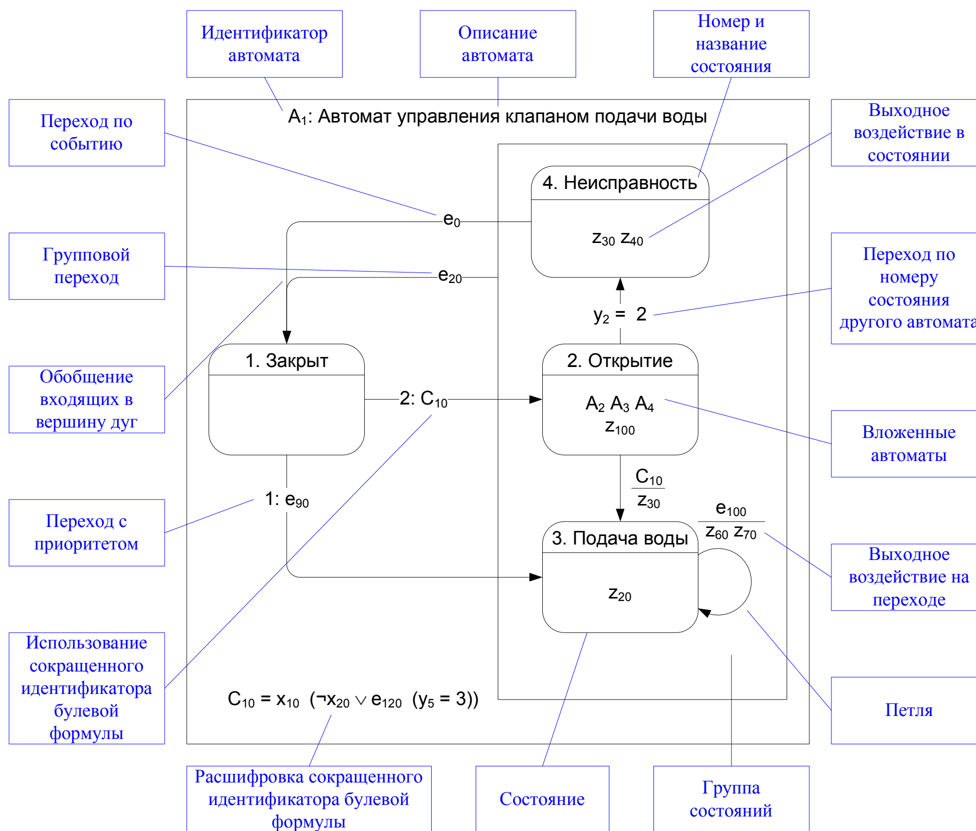

Рис. 2.24. Нотация диаграмм переходов

2.2.3. Использование спецификаций

Теперь, когда читатель знаком с графической нотацией, которая используется в программировании с явным выделением состояний, обсудим назначение спецификаций, построенных с помощью этой нотации.

Проектирование программных систем, и особенно систем со сложным поведением, «в уме» практически невозможно. Спецификации фиксируют результат процесса проектирования, освобождая ум разработчика для решения других задач. Спецификациями можно обмениваться с другими разработчиками, а также непосредственно использовать в качестве входных данных для следующего этапа разработки системы – реализации (рис. 2.25).

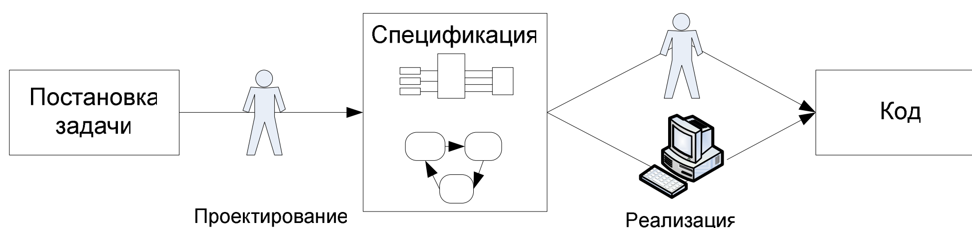

Рис. 2.25. Этапы разработки программной системы со сложным поведением

В рамках предлагаемого подхода спецификация с использованием графов переходов является не набором «картинок», поясняющих работу системы [34], а математической моделью логики поведения системы.

Поэтому имея схему связей и диаграмму переходов, код, реализующий автомат на языке программирования, можно построить с помощью формального преобразования[11]. Эта задача уже не является творческой. Ее несложно выполнить разработчику, однако гораздо лучше, если код генерируется автоматически. Ряд инструментальных средств, которые обсуждаются далее в этой книге, предоставляют такую возможность.

Что же происходит со схемами связей и диаграммами переходов после этапа реализации? Наличие работающего кода не делает спецификации бесполезными. Они становятся частью *проектной документации* системы и залогом успеха ее эволюции [35].

Как упоминалось выше, автоматное программирование берет свое начало от логического управления, которое является в чистом виде инженерной дисциплиной. *В инженерной практике проект или его этап обязательно завершается выпуском проектной документации.* Эта полезная традиция перешла «по наследству» к автоматному программированию: создание документации в рамках автоматного подхода является такой же неотъемлемой частью процесса разработки, как проектирование и реализация.

Проектная документация составляется на естественном языке и обычно содержит постановку задачи, описание структуры и поведения системы, примеры ее использования. Для каждого автомата документация должна включать его словесное описание, схему связей и диаграмму переходов, а возможно, и фрагмент кода, ее реализующего.

2.2.4. Сравнение с методом *Statemate*

Проведем краткое сравнение нотации графов переходов, применяемых в программировании с явным выделением состояний, и *диаграмм состояний* (*Statecharts*) [36], предложенных *Д. Харелом* в рамках метода разработки реактивных систем *Statemate* [28][12].

---

[11] Процесс преобразования спецификации автомата в код обсуждается в разд. 2.3.

[12] За создание этого метода его авторы были удостоены премии *ACM Software Systems Award 2007*.

Авторы считают уместным сравнение двух методов разработки (и, в частности, двух графических нотаций), поскольку оба метода предназначены для создания систем со сложным поведением, оба основаны на конечных автоматах и принципах процедурного программирования. Следует отметить, что, хотя подход *Statemate* является в чистом виде процедурным, диаграммы состояний входят также в состав объектно-ориентированного языка спецификации *UML* и используются в объектно-ориентированном методе разработки *Unified Rational Process* [37].

В соответствии с подходом *Statemate* предлагается рассматривать проектируемую систему с трех точек зрения и, соответственно, использовать для спецификации три различных графических нотации. Структурный вид системы предлагается описывать с помощью *диаграмм модулей*, функциональный вид – с помощью *диаграмм деятельностей*, а поведенческий – с помощью *диаграмм состояний*.

На диаграмме модулей изображается структура системы в виде множества взаимодействующих аппаратных и программных компонент (модулей, подсистем) и элементов внешней среды. На рис. 2.26 приведен пример диаграммы модулей для системы раннего оповещения. Все примеры настоящего раздела заимствованы из книги [28].

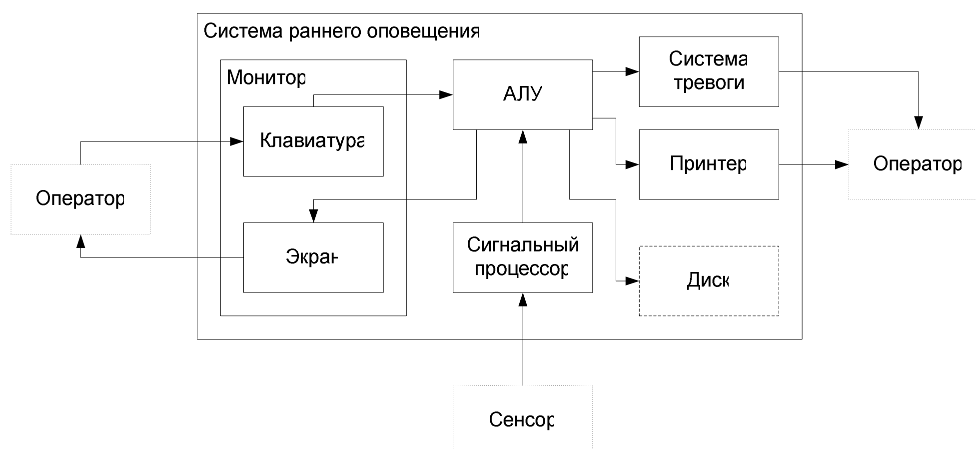

Рис. 2.26. Диаграмма модулей для системы раннего оповещения

На диаграмме деятельностей[13] отображается иерархия функциональных возможностей (функций, деятельностей) системы, порожденная путем декомпозиции сверху вниз. Пример такой диаграммы приведен на рис. 2.27.

[13] Не следует путать с диаграммой с тем же названием в языке *UML*.

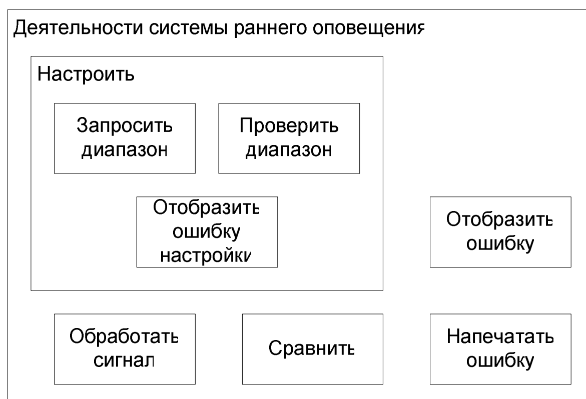

Рис. 2.27. Диаграмма деятельностей для системы раннего оповещения

Диаграммы состояний – это графическая нотация, как и диаграммы переходов, основанная на графах и предназначенная для отображения определенной автоматной модели. Если автоматная модель, используемая в программировании с явным выделением состояний, расширяет традиционное понятие конечного автомата такими дополнительными возможностями как вложенные и вызываемые автоматы, группы состояний, переходы с приоритетами и т. д., то в подходе *Statemate* используются другие расширения. Среди наиболее употребительных: *вложенные состояния*, *ортогональные состояния*, *исторические состояния*, *действия при входе* в состояние и *при выходе* из него, *деятельности*. С концепциями и терминологией диаграмм состояний можно познакомиться в работах [28, 30, 36]. Пример диаграммы состояний приведен на рис. 2.28.

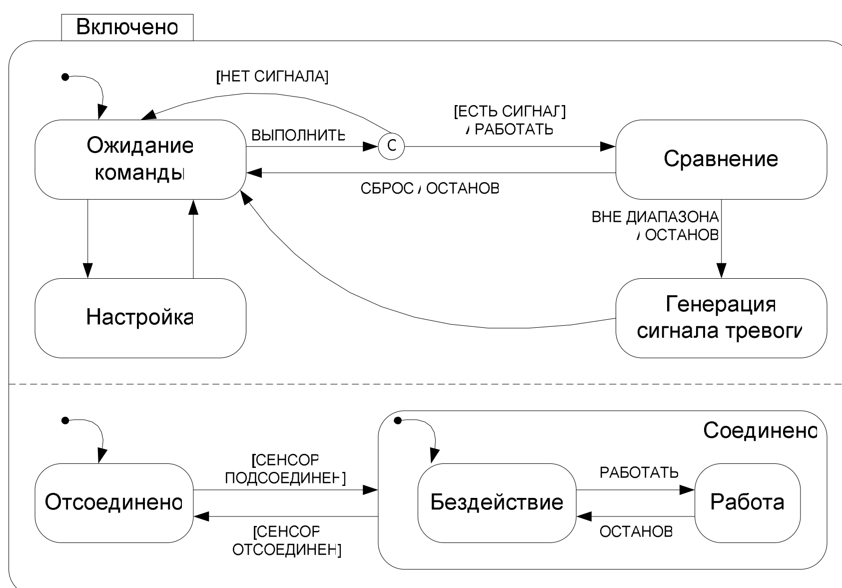

Рис. 2.28. Диаграмма состояний для системы раннего оповещения

В программировании с явным выделением состояний, в отличие от метода *Statemate*, используются всего два основных вида диаграмм: на схемах связей отражают структурные и функциональные свойства системы, а на диаграммах переходов – ее поведение. Трудно судить объективно, какой из подходов к описанию системы лучше. Отметим только, что в подходе *Statemate* диаграммы модулей и деятельностей, в некотором смысле, конфликтуют друг с другом, поскольку первые навязывают разработчику объектную декомпозицию, а вторые – функциональную декомпозицию сверху вниз. Во избежание противоречия, авторы подхода рекомендуют брать за основу архитектуры диаграмму деятельностей, а с помощью диаграммы модулей изображать систему только на самом верхнем уровне абстракции. При использовании схем связей такого противоречия не возникает.

Многие различия между языками спецификации метода *Statemate* и автоматного программирования, обусловлены тем, что автоматное программирование имеет более широкую область применения. Подход *Statemate* предназначен для создания реактивных систем, а это частный случай систем со сложным поведением. По этой причине диаграммы состояний изобилуют понятиями, специфичными для таких систем: сторожевое условие, деятельность, действие при входе в состояние и при выходе из состояния. Аналогов этих понятий нет ни в теории формальных языков, ни в теории автоматов, ни в теории управления. С другой стороны, автоматное программирование основывается на классических автоматных моделях (автоматах Мили и Мура) и вносит только два основных расширения (вложенные и вызываемые автоматы), которые, однако, являются весьма выразительными. Таким образом, нотация диаграмм переходов является более простой по сравнению с языком *Statecharts*, но при этом позволяет выражать те же поведенческие свойства. Среди ограничений, которые характерны для диаграмм состояний и отсутствуют у диаграмм переходов, отметим следующие:

- ориентированность на проектирование событийных систем;
- условием перехода может быть только единичное событие, охраняемое сторожевым условием;
- выходное воздействие на переходе может состоять только из единичного действия.

Эти ограничения представляются авторам неоправданными.

Одно из главных свойств любой графической нотации, предназначенной для описания сложных программных систем – это поддержка декомпозиции. В диаграммах переходов этой цели служат вложенные и вызываемые автоматы, а в языке *Statecharts* – *вложенные* и *ортогональные* состояния. Вложенные состояния используются для последовательного уточнения логики: внутрь более абстрактного, так называемого, *суперсостояния* могут быть вложены несколько более конкретных состояний. Ортогональные состояния служат для задания независимых логических «измерений». Если система находится в состоянии, разбитом на несколько ортогональных компонентов, это означает, что она находится во всех этих компонентах одновременно[14]. На рис. 2.28 состояния «Бездействие» и «Работа»

[14] Поэтому в некоторых источниках такие состояния называют не ортогональными. а параллельными.

вложены в состояние «Соединено». Наиболее абстрактное состояние «Включено» разделено пунктирной линией на два *ортогональных* компонента.

В рамках нотации диаграмм переходов обе концепции: и последовательное уточнение логики, и задание независимых логических измерений – поддерживаются механизмом вложенных автоматов (хотя в графах переходов для компактного изображения однотипных переходов могут использоваться групповые состояния (рис. 2.24), которые в этом смысле эквивалентны суперсостояниям). Семантически суперсостояние с несколькими вложенными состояниями на диаграмме переходов эквивалентно состоянию с единственным вложенным в него автоматом, а суперсостояние с несколькими ортогональными компонентами – состоянию с несколькими вложенными автоматами (или нескольким параллельно работающим автоматам на верхнем уровне иерархии).

С точки зрения синтаксиса, решение, используемое в диаграммах переходов, имеет важное преимущество. В отличие от вложенных состояний, вложенные автоматы специфицируются отдельно от объемлющего автомата, и для каждого из них строится свой граф переходов. Это решение более масштабируемое (при росте размера системы диаграммы не становятся неограниченно большими) и более приспособленное для повторного использования (обращение к одному и тому же вложенному автомату в разных местах не требует повторения его спецификации). При этом отметим, что в языке *UML* поддерживаются вложенные диаграммы состояний, однако их поддержка появилась только в *UML* 2.0 [38].

В заключение отметим, что, несмотря на разночтения в нотациях, подход *Statemate* и программирование с явным выделением состоянии концептуально очень близки. К сожалению, попав в объектно-ориентированный мир – в язык *UML*, диаграммы состояний не заняли подобающего им положения. Вместо формальной спецификации логики сложного поведения, достаточно выразительной для автоматической генерации по ней кода, диаграммы состояний превратились в еще один вид иллюстраций, которые иногда включают в техническую документацию системы. Причина этого, по мнению авторов, в недостаточно глубоком осмыслении роли автоматного описания поведения в объектно-ориентированном программировании. Попытка такого осмысления, результатом которой являются концепции *объектно-ориентированного программирования с явным выделением состояний*, сделана авторами в главе 3. При этом следует отметить, что возможность автоматической генерации кода с учетом диаграмм поведения появилась в рамках концепции *исполняемый UML* [39, 40], что потребовало значительной доработки языка *UML*.

2.3. Реализация

В данном разделе обсуждаются вопросы программной реализации различных автоматных моделей, а также рассматривается реализация систем со сложным поведением в целом в рамках процедурного программирования с явным выделением состояний. В разд. 2.3.1 внимание уделяется задачам логического управления, а в разд. 2.3.2 класс рассматриваемых задач расширяется до произвольных систем со сложным поведением. Задачи логического управления снова, как и в предыдущих разделах, рассматриваются отдельно по нескольким причинам. Во-первых, они имеют ярко выраженные особенности: отсутствие событий, автоматы активны, взаимодействие путем обмена номерами состояний. Во-вторых, логическое управление – традиционная область применения автоматного программирования и,

рассматривая приемы, используемые в этой области, отдельно от остальных, можно проследить эволюцию автоматной парадигмы.

Подчеркнем, что приемы реализации, рассматриваемые в этом разделе, являются частью общего метода разработки систем со сложным поведением (*программирования с явным выделением состояний*) и не могут рассматриваться самостоятельно. В частности, будем полагать, что до начала реализации некоторой системы, уже закончен этап ее проектирования (или, по крайней мере, очередная итерация этого этапа): выделены объекты управления и автоматы, для каждого автомата простроена схема связей и граф переходов.

2.3.1. Задачи логического управления

В системах управления логика может быть реализована как программно [41], так и аппаратно [42]. Вопросы аппаратной реализации конечных автоматов достаточно хорошо изучены, известен ряд эффективных методов решения этой задачи. Однако эти методы не относятся напрямую к предмету этой книги (задачи логического управления интересны нам не сами по себе, а как источник концепций, которые легли в основу парадигмы автоматного программирования). Поэтому в данном разделе обсуждается только программная реализация автоматов в задачах логического управления. На практике программная реализация применима в тех случаях, когда управление осуществляется программируемым вычислительным устройством (программируемым логическим контроллером, микроконтроллером, персональным компьютером и т. п.).

В данном разделе предполагается, что объекты управления, их сигнализаторы и исполнительные механизмы реализованы аппаратно. Поэтому обсуждаться будет только технология программной реализация управляющей части системы: автомата или множества взаимодействующих автоматов.

Для иллюстрации предлагаемой технологии выбран язык программирования *C*. С одной стороны, этот язык традиционно используется для реализации систем управления, а с другой – является относительно высокоуровневым и достаточно популярным. Отметим, что это всего лишь пример: обсуждаемая технология может быть использована совместно с большинством существующих императивных языков программирования.

Каковы критерии оптимальности технологии программной реализации автоматов в системах логического управления? Исходя из специфики данной области (сложное поведение, жесткие ограничения на время выполнения и объем занимаемой памяти), основными критериями можно считать следующие.

- **Возможность формального преобразования графа переходов автомата в программный код**. Проектирование логики – одна из главных и самых трудных задач при создании систем со сложным поведением. Графы переходов помогают справиться с этой задачей, делая логику наглядной. Было бы нецелесообразно, уже построив граф переходов, однозначно описывающий логику, предпринимать еще одно творческое усилие, чтобы преобразовать диаграмму в код. Это преобразование должно быть формальным и однозначным. В этом случае его можно поручить машине.

- **Изоморфизм программного кода графу переходов автомата**. Если программный код предназначен для чтения и модификации человеком, то каждому структурному элементу графа переходов (состоянию, переходу) должна соответствовать отдельная конструкция языка программирования. Такое однозначное соответствие в работе [4] названо *изоморфизмом*.

- **Эффективность по времени и по памяти**. Большинство управляющих систем являются системами реального времени. В качестве вычислительных устройств, осуществляющих управление, чаще всего используются микроконтроллеры, в которых объем памяти, доступной пользователю, невелик. В связи с этим следует отдавать предпочтение менее гибким, но более простым и быстрым решениям.

Пусть на стадии проектирования для управления системой был построен активный автомат Мура первого рода с s состояниями, n входными и m выходными переменными. Работу этого автомата можно описать с помощью схемы алгоритма (рис. 2.29).

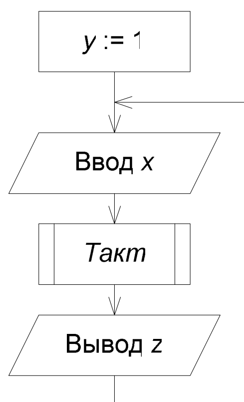

Рис. 2.29. Схема алгоритма, реализующего активный автомат

Алгоритм начинается с блока инициализации, в котором внутренней переменной присваивается номер стартового состояния. Далее следует бесконечный цикл, в котором последовательно выполняются ввод входного воздействия, вызов функции, реализующей один такт работы автомата, и вывод выходного воздействия.

Реализация блоков ввода и вывода зависит от конкретной системы и не относится напрямую к предмету этой книги. Поэтому далее в этом разделе обсуждается только реализация функции «Такт».

В течение каждого такта работы автомату необходимо:

- в зависимости от текущего состояния установить некоторые выходные переменные в единицу, а остальные – в ноль;

- в зависимости от текущего состояния и значений входных переменных обновить состояние.

Эту последовательность действий можно описать с помощью схемы алгоритмов так, как это показано на рис. 2.30 [43].

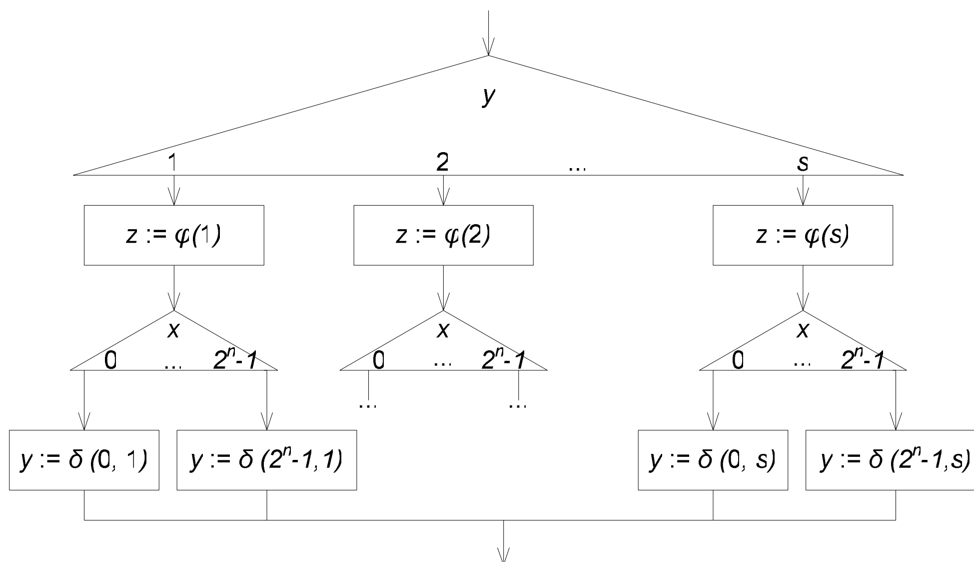

Рис. 2.30. Схема алгоритма, реализующего один такт работы автомата Мура

Вверху схемы расположен *дешифратор состояний*, в котором поток управления разветвляется в зависимости от значения текущего состояния. В каждой из ветвей находится блок формирования выходного воздействия и *дешифратор входных воздействий* (в нем поток управления разветвляется в зависимости от значения текущего входного воздействия). Для обозначения входных воздействий в схеме для удобства вместо битовых векторов используются эквивалентные им натуральные числа. В каждой из ветвей, исходящих из дешифраторов входных воздействий, выполняется обновление текущего состояния.

В языке *C* существуют два основных способа реализации такого алгоритма: с помощью таблиц и с помощью инструкции выбора.

В первом случае функции выходов и переходов автомата представляются в виде таблиц (массивов) в памяти. Для того чтобы представить функцию выходов требуется массив из s элементов, сопоставляющий вектор значений выходных переменных каждому состоянию. Для представления функции переходов требуется таблица из s строк и 2^n столбцов, сопоставляющая каждому состоянию и вектору значений входных переменных новое состояние. Таблицы действий и переходов строятся динамически во время выполнения программы. Поэтому данный способ является более гибким по сравнению с использованием инструкции выбора: автомат можно изменять в ходе выполнения программы. Однако эта гибкость достигается ценой некоторой потери производительности.

Кроме того, при описанном способе хранения размер таблицы переходов растет экспоненциально с увеличением числа входных переменных автомата. В большинстве случаев нет объективной необходимости хранить всю эту таблицу целиком: значение функции переходов в каждом состоянии зависит не от всех компонентов входного воздействия, а лишь от небольшого набора *значимых* входных переменных. Это наблюдение можно использовать для сокращения размера таблиц переходов и действий в том случае, если требуется компактное представление

автоматов, которое можно легко создавать и модифицировать в процессе выполнения программы. Например, такое представление необходимо для оптимизации автоматов, которая обсуждается в разд. 4.4. В контексте настоящего раздела динамическая модификация структуры автомата не требуется. Поэтому в целях повышения быстродействия предпочтение следует отдать второму из упомянутых выше способов реализации (с помощью инструкции выбора), так как он является статическим.

Для обозначения инструкции выбора в *C* используется ключевое слово `switch` [44]. В других императивных языках программирования высокого уровня имеются аналогичные инструкции.

Инструкции выбора можно использовать для реализации дешифраторов состояний и входных воздействий (рис. 2.30). В результате шаблон реализации такта автомата Мура примет следующий вид (листинг 2.1).

Листинг 2.1. Шаблон реализации такта автомата Мура на языке *C* (с использованием двух инструкций выбора)

```
void A() {
    switch (y) {            // Дешифратор состояний
        case 1:
            // Блок формирования выходных воздействий
            z1 = <0|1>; ...; zm = <0|1>;
            switch (x) {    // Дешифратор входных воздействий
                case 0: y = <1..s>; break;  // Обновление состояния

                ...
                case <2^n - 1>: y = <1..s>; break;
            }
            break;
        case 2:
        ...
        case <s>:
            ...
    }
}
```

При таком подходе автомат описывается статически: его структура закодирована в тексте программы. Это свойство обеспечивает требуемое быстродействие. Однако остается другая проблема: экспоненциальный рост описания автомата с увеличением числа входных переменных. Действительно, число меток `case` во вложенной инструкции выбора, а, следовательно, и объем текста программы, экспоненциально зависит от n (числа входных переменных).

Для решения этой проблемы вспомним наблюдение, сделанное в связи с таблицами: в большинстве случаев только небольшой набор входных переменных является значимым в каждом состоянии. Поэтому в указании значения функции переходов для всех возможных входных воздействий нет необходимости. Как правило, только некоторые сочетания значений нескольких входных переменных приводят к смене состояния. По-видимому, наиболее компактный способ записи функции переходов

используется в нотации графов переходов: для каждого состояния задается множество исходящих переходов, помеченных условиями в виде булевых формул. Опыт показывает, что в реальных задачах размер этих формул практически не увеличивается с ростом числа входных переменных.

Предложенный выше шаблон реализации легко модифицировать так, чтобы в нем использовалось компактное представление функции переходов. Для этого достаточно заменить внутреннюю инструкцию выбора несколькими инструкциями ветвления (по числу дуг, исходящих из данного состояния на графе переходов). Условиями ветвления будут метки исходящих дуг. Отметим, что петлям в автомате Мура сопоставлять инструкции ветвления нет необходимости. Модифицированный шаблон реализации представлен в листинге 2.2.

Листинг 2.2. Шаблон реализации такта автомата Мура на языке *C* (с использованием инструкций выбора и ветвления)

```
void A() {
    switch (y) {            // Дешифратор состояний
        case 1:
            // Блок формирования выходных воздействий
            z1 = <0|1>; ...; zm = <0|1>;
            if (<метка_1>) {            // Дешифратор входных воздействий
                y = <1..s>;             // Обновление состояния
            } else if (<метка_2>) {
                y = <1..s>;
            } ... else if (<метка_k>) {
                y = <1..s>;
            }
            break;
        case 2:
        ...
        case <s>:
            ...
    }
}
```

Решение проблемы неоправданного большого объема представления автомата попутно привело к еще одному улучшению: формальное преобразование графа переходов в код стало проще.

Однако действительно ли описание переходов из состояния в виде последовательности инструкций ветвления изоморфно описанию в виде множества исходящих дуг? Источником несоответствия может быть тот факт, что инструкции ветвления всегда выполняются в порядке, заданном текстом программы, а порядок проверки условий на дугах в графе не определен. Однако в корректном (непротиворечивом) графе переходов порядок проверки условий на дугах не имеет значения. Если же в графе переходов дугам назначены приоритеты, то и в тексте программы соответствующие инструкции ветвления должны располагаться в порядке, заданном приоритетами.

В качестве примера использования рассмотренного шаблона приведем реализацию автомата системы управления клапаном, проектирование которого обсуждалось в разд. 2.1.1.

Листинг 2.3. Реализация системы управления клапаном на языке C

```
void A() {
    switch (y) {
        case 1:      // Закрыт
            z1 = 0; z2 = 0; z3 = 0; z4 = 0;
            if (x1)                  { y = 2; }
            break;
        case 2:      // Открывается
            z1 = 1; z2 = 0; z3 = 0; z4 = 1;
            if (x4 && x6)            { y = 3; }
            else if (!x4 && x6)      { y = 5; }
            break;
        case 3:      // Открыт
            z1 = 0; z2 = 0; z3 = 0; z4 = 0;
            if (x2)                  { y = 4; }
            break;
        case 4:      // Закрывается
            z1 = 0; z2 = 1; z3 = 0; z4 = 1;
            if (x5 && x6)            { y = 1; }
            else if (!x5 && x6 )     { y = 5; }
            break;
        case 5:      // Неисправность
            z1 = 0; z2 = 0; z3 = 1; z4 = 0;
            if (x3)                  { y = 1; }
            break;
    }
}
```

Отметим, что ключевые слова `else` в данной реализации введены исключительно для повышения быстродействия программы. Поскольку условия переходов из каждого состояния в этом случае ортогональны, указанные ключевые слова можно не писать.

Выше обсуждалась реализация систем со сложным поведением, управляемых одним автоматом. Перейдем к вопросам реализации систем, управляемых взаимодействующими автоматами.

В этом случае такт работы каждого из автоматов удобно реализовать в виде отдельной функции. Если автоматы в системе взаимодействуют путем обмена номерами состояний, то в условиях ветвления, кроме входных переменных, будут участвовать внутренние переменные других автоматов.

В блоках формирования выходных воздействий помимо инструкций присваивания значений выходным переменным появятся обращения к вложенным и вызываемым автоматам. Обращение к вложенному автомату выполняется путем вызова функции, реализующей такт его работы. Обращение к вызываемому автомату – более сложная

операция. Один из способов ее реализации предполагает создание для каждого вызываемого автомата A_i (описанного с помощью функции `void A<i>()`), дополнительной функции, которая имеет следующую структуру:

Листинг 2.4. Шаблон функции обращения к вызываемому автомату

```
void Call_A<i>() {
    y<i> = 1;
    while (y<i> != <завершающее_состояние>) {
        <Ввод x>
        Ai();
        <Вывод z>
    }
}
```

После этого в точке обращения к вызываемому автомату выполняется вызов функции `Call_A<i>()`.

Если при проектировании системы была произведена параллельная автоматная декомпозиция, то существует два основных варианта реализации.

1. Вся система выполняется на одном процессоре. В этом случае необходимо обеспечить псевдопараллельную работу автоматов. Здесь проявляется одно из достоинств автоматного программирования: даже в том случае, если среда выполнения не поддерживает многозадачность, псевдопараллельное выполнение автоматов легко обеспечить, поместив вызовы соответствующих им функций подряд в теле цикла главной программы (рис. 2.31).

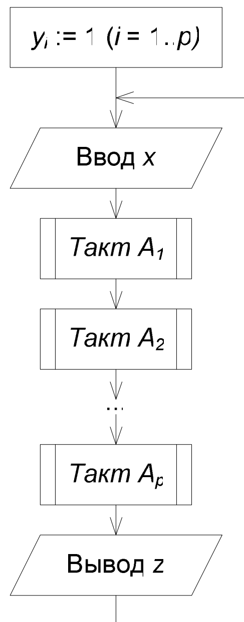

Рис. 2.31. Схема алгоритма, реализующего псевдопараллельную работу *p* активных автоматов

2. Каждый автомат выполняется на отдельном процессоре. В этом случае каждая из функций, реализующих такты работы автоматов системы, помещается в свой собственный цикл внутри отдельной главной программы. Здесь на первый план выступают вопросы взаимодействия и синхронизации. В частности, необходимо обеспечить согласованность данных, которыми обмениваются автоматы в системе. Выбор того или иного решения этих проблем зависит от специфики конкретной системы.

В качестве примера приведем реализацию системы управления двумя клапанами. Из нескольких вариантов архитектуры, предложенных в разд. 2.1.2, выберем тот, в котором осуществлялась параллельная автоматная декомпозиция. Предположим, что система должна выполняться на одном процессоре. Листинг 2.5 содержит реализации функций, отвечающих за такт работы каждого из двух автоматов, а также шаблон реализации главной программы.

Листинг 2.5. Реализация системы управления двумя клапанами на языке *C*

```
void A1() {     // Автомат, управляющий первым клапаном
    switch (y1) {
        case 1:     // Закрыт
            z1 = 0; z2 = 0;
            if (x1)         { y1 = 2; }
            break;
        case 2:     // Открывается
            z1 = 1; z2 = 0;
            if (x3)         { y1 = 3; }
            break;
        case 3:     // Открыт
            z1 = 0; z2 = 0;
            if (y2 == 1)    { y = 4; }
            break;
        case 4:     // Закрывается
            z1 = 0; z2 = 1;
            if (x4)         { y1 = 1; }
            break;
    }
}

void A2() {     // Автомат, управляющий вторым клапаном
    switch (y2) {
        case 1:     // Закрыт
            z3 = 0; z4 = 0;
            if (y1 == 3)    { y2 = 2; }
            break;
        case 2:     // Открывается
            z3 = 1; z4 = 0;
            if (x5)         { y2 = 3; }
            break;
        case 3:     // Открыт
```

```
            z3 = 0; z4 = 0;
            if (x2)         { y2 = 4; }
            break;
        case 4:     // Закрывается
            z3 = 0; z4 = 1;
            if (x6)         { y2 = 1; }
            break;
    }
}

void main() {       // Главная программа
    y1 = 1; y2 = 1;
    while (1) {
        <Ввод x>
        A1(); A2();
        <Вывод z>
    }
}
```

2.3.2. Другие классы задач

В данном разделе рассматриваются задачи, которые не относятся к области логического управления. В сфере прикладного программирования для персональных компьютеров системы со сложным поведением чаще всего являются событийными [45], автоматизированные объекты в них, как правило, пассивны. Что касается автоматных моделей, то удобнее всего использовать автоматы Мили или (реже) смешанные автоматы.

Какие изменения в структуре алгоритма повлечет эта специфика?

1. Поскольку используются другие автоматные модели, блок формирования выходных воздействий должен быть расположен по-другому.

2. Команды объекта управления реализованы уже не аппаратно, а программно, как обыкновенные процедуры. Поэтому в блоке формирования выходных воздействий вместо присваивания значений выходным переменных следует поместить вызовы процедур, соответствующих тем выходным переменным, которые принимают значение «истина».

3. Поскольку рассматриваемые автоматизированные объекты пассивны, изменилась и главная программа.

Рассмотрим схему алгоритма, описывающего один такт работы автомата Мили (рис. 2.32). Здесь блок формирования выходных воздействий находится за дешифратором входных воздействий и содержит вызовы определенных команд объекта управления [43]. Набор команд, которые должны быть вызваны, определяется значением функции выходов автомата для данного состояния и входного воздействия.

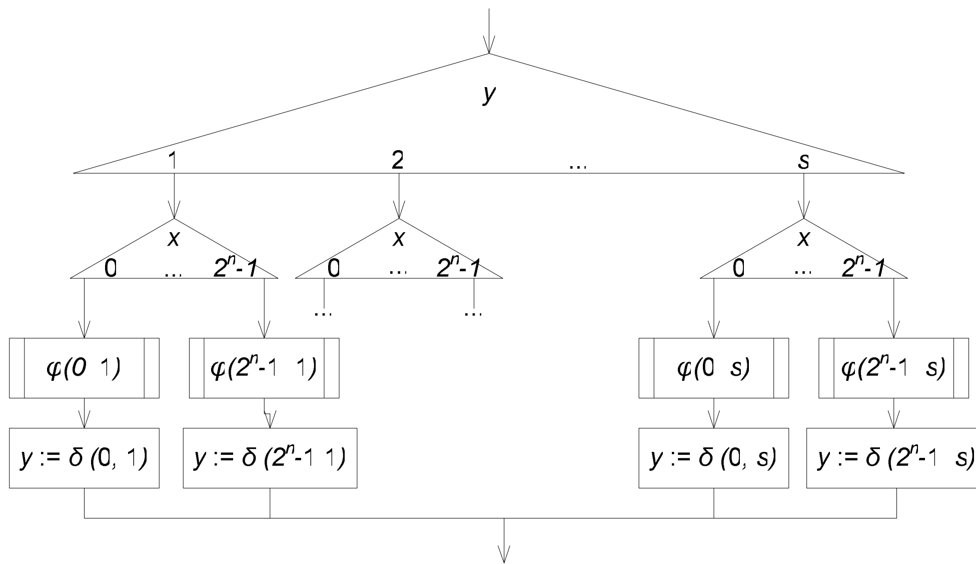

Рис. 2.32. Схема алгоритма, реализующего один такт работы автомата Мили

Объединив этот алгоритм с алгоритмом такта работы автомата Мура из предыдущего раздела, получим алгоритм для смешанного автомата (рис. 2.33), в котором присутствуют два блока формирования выходных воздействий: до и после дешифратора входных воздействий.

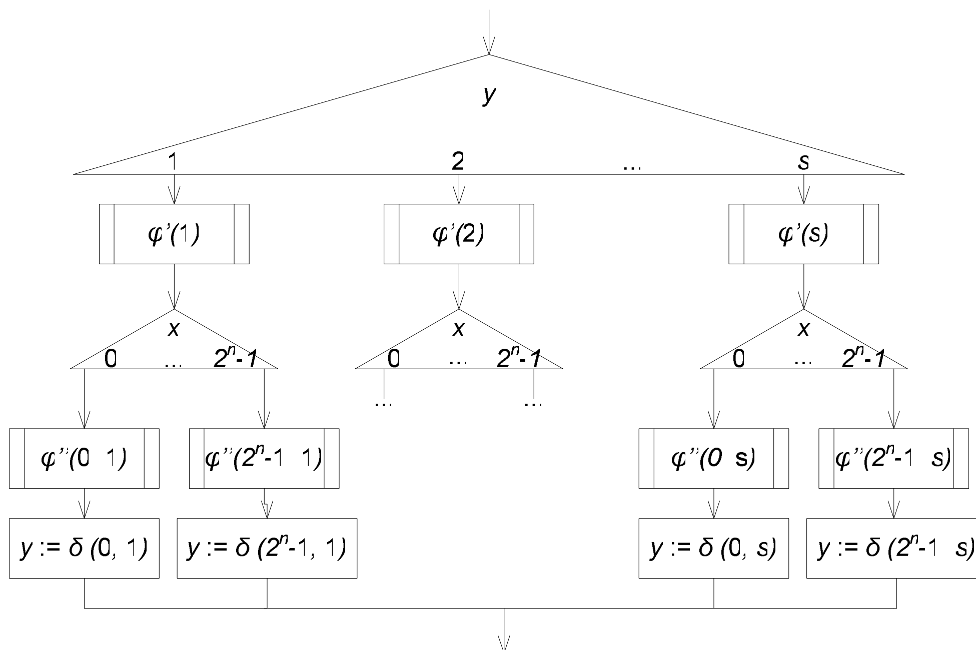

Рис. 2.33. Схема алгоритма, реализующего один такт работы смешанного автомата

Главная программа изменилась гораздо сильнее. Как уже известно читателю, пассивная автоматная модель работает не постоянно (такт за тактом), а совершает очередной такт работы по инициативе внешней среды. Поэтому для таких моделей главная программа уже не является частью реализации автомата. Эта программа описывает функционирование внешней среды, и из нее в определенных местах может вызываться подпрограмма, реализующая такт работы автомата. В прикладных событийных системах функция главной программы, как правило, состоит в извлечении сообщений из очереди и передаче их соответствующим обработчикам. Однако это лишь вершина айсберга: прикладная программа работает под управлением операционной системы, которая берет на себя всю работу по приему событий от внешних устройств и передачи их программе. Поэтому само понятие главной программы в современных прикладных программных системах является относительным и неопределенным.

Кроме того, при переходе к прикладному программированию событийных систем изменились и критерии оптимальности реализации автоматов. Изоморфизм программного кода графу переходов по прежнему остается на первом месте, однако эффективность по времени и памяти для данного класса задач не столь критична. Напротив, большее значение приобретают гибкость и способность эволюционировать, адаптируясь к новым требованиям.

Напомним, что в данном разделе обсуждаются вопросы реализации в контексте процедурного программирования. Объектно-ориентированный подход обладает более широким арсеналом методов и шаблонов для реализации автоматов, часть из них обсуждаются далее в этой книге (разд. 3.3). А пока мы остаемся наедине с языком *C* и двумя способами реализации автоматов, перечисленными выше: с помощью таблиц и с помощью инструкций выбора.

Существует ряд задач, в которых необходимо модифицировать автомат во время выполнения программы. Для таких случаев следует использовать реализацию автомата в виде таблицы. Однако опыт показывает, что такие задачи встречаются довольно редко. Поэтому наиболее употребительным остается шаблон реализации, описанный в предыдущем разделе: использование инструкции выбора в сочетании с инструкцией ветвления (тем более что такой способ упрощает изоморфное преобразование графа переходов в программный код).

Шаблон реализации такта работы автомата изменился по сравнению с листингом 2.2 несущественно. Поскольку вместо автомата Мура теперь используется автомат Мили, то блок формирования выходного воздействия переместился внутрь инструкции ветвления. Кроме того, в этом блоке вместо присваивания значений выходным переменным теперь выполняются вызовы подпрограмм, соответствующих командам объекта управления.

Более интересный вопрос – взаимодействие функции такта автомата с вызывающим контекстом, иначе говоря, передача автомату событий. Существуют три основных варианта такого взаимодействия.

1. События, как и входные переменные, являются **глобальными переменными** программы или модуля программы, в котором находится автомат (отметим, что второй вариант менее вероятен, так как в отличие от входных переменных, которые являются локальными для автоматизированного объекта, события, как правило, предназначены для межмодульного взаимодействия). В том случае,

если события *исключительны* (два события не могут произойти одновременно), для хранения текущего события достаточно ввести одну целочисленную переменную, значение которой в дешифраторе входных воздействий будет сравниваться с идентификаторами конкретных событий. Если же события в разрабатываемой системе не обладают свойством исключительности, для хранения множества текущих событий можно использовать битовый вектор. В таком случае представление событий ничем не отличается от представления входных переменных. Шаблон реализации функции такта автомата при этом способе представления событий отличается от описанного в листинге 2.2 только положением блока формирования выходных воздействий.

2. События являются **аргументами** функции такта автомата. Это проектное решение отражает различие между событиями и входными переменными: здесь автомат *обрабатывает* событие (или набор событий), в то время как значения входных переменных лишь при необходимости опрашиваются автоматом по его собственной инициативе. Кроме того, при реализации автомата вручную такое решение предотвращает программные ошибки, возможные при первом подходе: когда программист забывает установить переменную для хранения события перед вызовом такта автомата. Изменения в шаблоне реализации в этом случае незначительны и затрагивают только сигнатуру функции такта:

   - `void A (int e)`, если события исключительны;
   - `void A (bool[] e)`, если события неисключительны.

   Этот вариант реализации был предложен в работах [27, 42, 46]. Описание примера его использования на практике приведено в работе [8].

3. Каждому событию сопоставляется отдельная **функция**. Это решение подходит только для реализации исключительной модели событий (отметим, что эта модель является наиболее распространенной). Кроме того, оно отражает активную роль событий, тот факт, что именно возникновение события инициировало вызов автомата. К тому же, это решение лучше всего согласовано с традиционной структурой программного модуля в событийных системах, где любая самостоятельная часть программы представляет собой множество обработчиков связанных по смыслу событий.

   Программа, построенная в соответствии с данным решением, уже не совсем изоморфна схеме алгоритма на рис. 2.32, где сначала дешифруется состояние, а затем входное воздействие. В этом случае сначала происходит ветвление по событиям, затем по состояниям, и, наконец, по входным переменным. Это может показаться неоправданным усложнением. Однако в большинстве событийных систем внешняя среда с необходимостью производит дешифрацию события перед отправкой его автомату (хотя бы для того, чтобы понять, какому автомату следует его передать). В этом случае при использовании других решений дешифрация событий на каждом такте без всякой необходимости происходит два раза.

   Шаблон реализации функции такта для третьего способа взаимодействия автомата с внешней средой описан в листинге 2.6.

Листинг 2.6. Шаблон реализации такта автомата Мили на языке *C* (каждому событию соответствует отдельная функция)

```
void e1() {
    switch (y) {            // Дешифратор состояний
        case 1:
            // Дешифратор наборов входных переменных
            if (<формула_от входных переменных_1>) {
                // Блок формирования выходных воздействий
                z<i_1>(); ...; z<i_t>();
                y = <1..s>;             // Обновление состояния
            } else if (<формула_от входных переменных_2>) {
                z<i_1>(); ...; z<i_t>();
                y = <1..s>;
            } ... else if (<формула_от входных переменных _k>) {
                z<i_1>(); ...; z<i_t>();
                y = <1..s>;
            }
            break;
        case 2:
        ...
        case <s>:
            ...
    }
}

void e2() {
    switch (y) {
        case 1:
            ...
        case 2:
        ...
        case <s>:
            ...
    }
}
…
void e<r>() {
    switch (y) {
        case 1:
            ...
        case 2:
        ...
        case <s>:
            ...
    }
}
```

Вариант реализации, близкий к описанному выше, был использован в работе [47].

Рассмотрим все три предложенных варианта реализации взаимодействия автомата с внешней средой на примере часов с будильником – сущности со сложным поведением, управляющий автомат для которой был построен в разд. 2.1.1.

В листинге 2.7 команды и запросы объекта управления часов с будильником в целях повышения модульности реализованы в виде отдельных функций, а не как часть реализации управляющего автомата [42]. В этом листинге используется первый из предложенных шаблонов (события описываются глобальными переменными).

Листинг 2.7. Реализация часов с будильником (события представлены глобальными переменными)

```
const int h = 1;
const int m = 2;
const int a = 3;
const int t = 4;

int e; // Текущее событие
int y; // Текущее управляющее состояние

// Реализация объекта управления
int hours;
int minutes;
int alarm_hours;
int alarm_minutes;

bool x1() {
    if ((minutes == alarm_minutes - 1) && (hours == alarm_hours) ||
    (alarm_minutes == 0) && (minutes == 59) && (hours == alarm_hours - 1))
        return true;
    else
        return false;
}

bool x2() {
    if ((minutes == alarm_minutes) && (hours == alarm_hours))
        return true;
    else
        return false;
}

void z1() {
    hours = (hours + 1) % 24;
}

void z2() {
    minutes = (minutes + 1) % 60;
```

```
}

void z3() {
    alarm_hours = (alarm_hours + 1) % 24;
}

void z4() {
    alarm_minutes = (alarm_minutes + 1) % 60;
}

void z5() {
    minutes = (minutes + 1) % 60;
    if (minutes == 0) hours = (hours + 1) % 24;
}

void z6() {
    ... // Включить звонок
}

void z7() {
    ... // Выключить звонок
}

// Реализация управляющего автомата
void A1() {
    switch (y) {
        case 1: // Будильник выключен
            if (e == h)                 { z1(); }
            else if (e == m)            { z2();  }
            else if (e == a)            { y = 2; }
            else if (e == t)            { z5(); }
            break;
        case 2: // Установка времени будильника
            if (e == h)                 { z3(); }
            else if (e == m)            { z4();  }
            else if (e == a)            { y = 3; }
            else if (e == t)            { z5(); }
            break;
        case 3: // Будильник включен
            if (e == h)                 { z1(); }
            else if (e == m)            { z2();  }
            else if (e == a)            { z7(); y = 1; }
            else if ((e == t) && x1())  { z5(); z6(); }
            else if ((e == t) && x2())  { z5(); z7(); }
            else if (e == t)            { z5(); }
            break;
    }
```

```
}
```

В листинге 2.8 представлена реализация управляющего автомата часов с будильником, использующая второй шаблон: события передаются автомату в качестве аргументов. Реализация объекта управления при этом не отличается от первого варианта, и поэтому в листинге не приведена.

Листинг 2.8. Реализация часов с будильником (события передаются в качестве аргументов)

```
// Реализация управляющего автомата
void A1(int e) {
    switch (y) {
        case 1: // Будильник выключен
            if (e == h)                 { z1(); }
            else if (e == m)            { z2();  }
            else if (e == a)            { y = 2; }
            else if (e == t)            { z5(); }
            break;
        case 2: // Установка времени будильника
            if (e == h)                 { z3(); }
            else if (e == m)            { z4();  }
            else if (e == a)            { y = 3; }
            else if (e == t)            { z5(); }
            break;
        case 3: // Будильник включен
            if (e == h)                 { z1(); }
            else if (e == m)            { z2();  }
            else if (e == a)            { z7(); y = 1; }
            else if ((e == t) && x1())  { z5(); z6(); }
            else if ((e == t) && x2())  { z5(); z7(); }
            else if (e == t)            { z5(); }
            break;
    }
}
```

Листинг 2.9 содержит реализацию управляющего автомата часов с будильником с помощью третьего шаблона: каждому событию сопоставляется отдельная функция.

Листинг 2.9. Реализация часов с будильником (отдельная функция для каждого события)

```
// Реализация управляющего автомата
void h() {  // Нажатие кнопки «H»
    switch (y) {
        case 1: // Будильник выключен
            z1();
            break;
        case 2: // Установка времени будильника
            z3();
            break;
```

```
        case 3: // Будильник включен
            z1();
            break;
    }
}

void m() {  // Нажатие кнопки «M»
    switch (y) {
        case 1: // Будильник выключен
            z2();
            break;
        case 2: // Установка времени будильника
            z4();
            break;
        case 3: // Будильник включен
            z2();
            break;
    }
}

void a() {  // Нажатие кнопки «A»
    switch (y) {
        case 1: // Будильник выключен
            y = 2;
            break;
        case 2: // Установка времени будильника
            y = 3;
            break;
        case 3: // Будильник включен
            z7();
            y = 1;
            break;
    }

void t() {  // Срабатывание минутного таймера
    switch (y) {
        case 1: // Будильник выключен
            z5();
            break;
        case 2: // Установка времени будильника
            z5();
            break;
        case 3: // Будильник включен
            if (x1())       {z5(); z6(); }
            else if (x2())  {z5(); z7(); }
            else z5();
            break;
```

```
    }
}
```

2.3.3. Инструментальные средства

Как было отмечено выше, основной критерий оптимальности реализации управляющих автоматов – возможность формального и изоморфного преобразования графа переходов в программный код. Это свойство позволяет создать программу (генератор кода), которая будет выполнять указанное преобразование автоматически. Таким образом, программисту останется только реализовать объект управления – его запросы, команды и множество вычислительных состояний. Код, реализующий логику программной системы, будет сгенерирован автоматически по более высокоуровневому описанию – графу переходов. Программу, выполняющую такое преобразование, логично назвать *инструментальным средством* автоматного программирования, потому что она позволяет получить максимальную выгоду из применения автоматного подхода.

Известны различные программы, позволяющие генерировать код по графам переходов конечных автоматов [48]. Однако мы рассмотрим только те из них, которые разрабатывались в контексте автоматного программирования и, соответственно, ориентированы на создание программных систем в целом, а не на реализацию отдельных автоматов.

Исторически первым инструментальным средством был конвертор *Visio2Switch* [49]. Он предоставляет возможность генерации кода на языке *C* по графам переходов, изображенным с помощью пакета *Microsoft Visio* в нотации, которая описана в работе [27] и близка к нотации, введенной в разд. 2.2.2. Конвертор *Visio2Switch* используется в настоящее время при создании программного обеспечения ряда ответственных систем реального времени.

Это направление исследований получило развитие в работе [50], в которой показано, как по графу переходов генерировать код на любом наперед заданном языке программирования. Авторами работы было создано инструментальное средство *MetaAuto* для поддержки предложенного ими подхода. В основе этого инструментального средства лежит один из известных методов порождающего программирования [51]: использование преобразований *XML*-документов, описанных на языке *XSLT*. Универсальность средства определяется, во-первых, возможностью обработки различных обозначений на графах переходов, а, во-вторых, возможностью генерации текстов программ на различных языках программирования. Для того чтобы иметь возможность генерировать код на определенном языке, необходимо создать для него один или несколько *XSLT*-шаблонов. В процессе разработки инструментального средства для иллюстрации предложенного подхода были созданы шаблоны для языков *C*, *C#* и *Turbo Assembler*.

Глава 3. Объектно-ориентированное программирование с явным выделением состояний

В предыдущей главе рассматривалась технология программирования, образованная сочетанием автоматного и процедурного подходов к разработке ПО. Надо сказать, что когда эта технология зарождалась, никто и не думал о сочетании подходов, оно возникло само собой.

Теперь, спустя несколько лет, полезно разобраться, почему автоматы так хорошо вписались в процедурный подход и на каких ролях сочетаются в программировании с явным выделением состояний указанные парадигмы. Ведущую роль здесь сыграло понятие «автомат как подпрограмма». При таком понимании автоматов они органично вписываются в процесс проектирования «сверху вниз», в котором модульная структура системы строится из подпрограмм. Если автомат – это всего лишь частный случай подпрограммы, то никакого конфликта парадигм не возникает. Архитектура системы остается почти такой же, как и раньше, за исключением того, что некоторые подпрограммы в ней являются реализациями автоматов.

Возможно ли так же органично объединить автоматный подход с какой-либо другой парадигмой программирования и получить от этого сочетания какие-либо преимущества? Вероятно, было бы интересно исследовать возможность использования автоматных концепций в контексте функционального или логического программирования, однако такое исследование, скорее всего, имело бы чисто теоретическую ценность. Если же речь идет о пользе для практического программирования, целесообразно рассмотреть совместное использование автоматного подхода и объектно-ориентированного программирования.

Вот уже около двух десятилетий объектно-ориентированное программирование является наиболее широко используемым стилем разработки ПО в мире. Про этот стиль написано столько книг, а поддерживающие его языки настолько различны, что на данный момент среди разработчиков не существует однозначного понимания, что такое объектно-ориентированное программирование. Если попытаться обобщить все имеющиеся мнения на этот счет, получится нечто слишком абстрактное. Поэтому, по аналогии с предыдущей главой, прежде всего опишем вкратце, что авторы этой книги понимают под объектно-ориентированным программированием. Отметим, что мнение авторов нельзя считать наиболее распространенным, скорее, оно совпадает со взглядами, изложенными в книге [29]. Для подробного знакомства с объектно-ориентированным программированием авторы рекомендуют эту книгу.

Объектно-ориентированная парадигма создания программного обеспечения (далее в качестве синонима используется термин *объектная технология*) характеризуется следующими основными концепциями.

- **Объектная декомпозиция**. Этот вид декомпозиции основан не на вопросе «Что делает система?», как декомпозиция сверху вниз, а на вопросе «Кто действует в системе?». Поскольку этот аспект является более устойчивым, то и построенная таким образом архитектура хорошо приспособлена к эволюции. В результате модули системы рождаются из *сущностей*, которые, однако, характеризуются не внутренним содержанием, а теми сервисами, которые они предоставляют. Такие

сущности называются *абстрактными типами данных* (АТД). Формально абстрактный тип данных состоит из множества операций, снабженных описаниями их областей определения – *предусловиями*, и множества аксиом, которые задают свойства операций.

- **Классы**. Класс – это полная или частичная реализация АТД и единственный вид модуля в объектно-ориентированный системе. Такие модули взаимодействуют между собой через *интерфейсы* (с точностью до обозначений интерфейс соответствует понятию АТД, на котором основан класс). Интерфейс класса содержит сигнатуры его *компонентов*: *запросов* и *команд*, а также семантические свойства: *предусловия* и *постусловия* запросов и команд, *инварианты класса*. Компоненты класса соответствуют операциями АТД, предусловия компонентов – предусловиям операций, постусловия и инварианты – аксиомам АТД.

  Предусловия, постусловия и инварианты также называются *утверждениями*, в совокупности они образуют *контракт* класса или его компонента. Метод разработки программного обеспечения, в котором утверждения являются неотъемлемой частью текста программной системы, называется *проектированием по контракту* [52]. В соответствии с этим методом система считается корректной только в том случае, если перед вызовом любого компонента выполняется его предусловие, а по завершении работы компонента – его постусловие. Инвариант класса должен выполняться во всех *устойчивых состояниях* любого объекта этого класса (в тех состояниях, которые могут наблюдать клиенты класса).

  В отличие от модулей в структурной парадигме программирования (например, в языках *Pascal* и *Ada*), класс одновременно является модулем и типом. Любой тип в объектно-ориентированной программной системе основан на некотором классе.

- **Объекты**. Если класс полностью реализует АТД, то он может иметь экземпляры – объекты. Связь между объектом и классом имеет двоякую природу, как и сам класс. С одной стороны, класс – это тип объекта (и понятие типа здесь практически ничем не отличается от понятия типа переменной в не объектно-ориентированных языках), а с другой – класс как модуль определяет те операции, которые применимы к объекту.

  Объект – это единственный вид сущности, имеющийся в объектно-ориентированной программной системе во время выполнения. Вызов компонента на некотором объекте – основной вид операции в объектно-ориентированных вычислениях.

- **Наследование** – это механизм, позволяющий создавать повторно используемые программные модули, «не обладая бесконечной мудростью» [29]. С точки зрения класса как модуля, наследование – это механизм расширения, позволяющий определять новые классы, частично используя определения уже существующих. С помощью этого механизма возможно, во-первых, построить модифицированный класс без необходимости внесения изменений в существующий класс (следовательно, и в описания его клиентов), а во-вторых, вносить такие модификации, которые изначально не предполагались автором класса.

> С точки зрения класса как типа, наследование – это механизм порождения *подтипов*. Подтипы, в свою очередь, обеспечивают один из видов *полиморфизма*, при котором к сущности некоторого типа, может быть во время выполнения присоединен объект его подтипа. Полиморфизм позволяет клиентам одинаково работать с множеством «похожих» классов, и поэтому является мощным механизмом повторного использования кода.

Исследования, в которых вместе фигурируют автоматы и объектно-ориентированное программирование, ведутся различными авторами в течение длительного времени. Например, известно множество шаблонов для объектно-ориентированной реализации автоматов [53, 54], а диаграммы состояний стали частью объектно-ориентированного языка моделирования *UML* [30]. Однако, судя по тому, что автоматы на сегодняшний день редко используются при проектировании и реализации сложного поведения, результаты исследований нельзя назвать удовлетворительными.

В работах, посвященных шаблонам (образцам) проектирования, зачастую утверждается, что концепция сложного поведения несовместима с объектной парадигмой. Поэтому существует очень много шаблонов объектно-ориентированной реализации автоматов, все они довольно сложные, и выбор подходящего для конкретной задачи представляет целую проблему [55, 56].

Что касается языка *UML*, в нем существует три основных препятствия использованию диаграмм состояний как общепризнанного и распространенного средства для описания сложного поведения. Во-первых, кроме диаграмм состояний для описания поведения предлагается использовать и другие типы диаграмм и не существует общих правил, определяющих, когда и какие диаграммы следует применять [57]. Во-вторых, в рамках унифицированного процесса разработки программ [37] не было предложено подходов для совместного использования диаграмм, описывающих структурные и поведенческие свойства программ. Наконец, в-третьих, диаграммы для описания поведения в среде пользователей *UML* в основном используются как язык общения между разработчиками и для документирования программ, в то время как подобающая им роль – точная спецификация, которая может служить источником для ее программной интерпретации и генерации текста программы.

Вместе с развитием процедурного программирования с явным выделением состояний, по инициативе одного из авторов этой книги было положено начало развитию другой разновидности автоматного программирования, названной *объектно-ориентированным программированием с явным выделением состояний* [58]. Это направление с самого начала было ориентировано на создание объектно-ориентированных программных систем со сложным поведением в целом, а не на описание конечных автоматов с помощью классов, как большинство образцов проектирования. Однако в вопросах технологии не все сразу оказалось так однозначно: это направление претерпело существенную эволюцию, которая не завершилась и до сих пор. Суть этой эволюции и ее нынешнее состояние будут отражены в данной главе.

3.1. Проектирование

Первый вопрос, который следует обсудить: для чего понадобилось рассматривать объектно-ориентированный вариант автоматного программирования, какие

проблемы традиционного программирования с явным выделением состояний способна решить объектная технология?

Первая проблема – недостаточная инкапсуляция. В системе, построенной с помощью метода, который описан в предыдущей главе, вычислительные состояния всех объектов управления и управляющие состояния всех автоматов глобальны – любой автомат или подпрограмма в системе имеет доступ к любым ее данным. Такие избыточные зависимости делают систему неспособной к эволюции.

Вторая проблема может быть незаметна в процессе изучения автоматного программирования и чтения этой книги, где все примеры сущностей и систем со сложным поведением довольно просты и проектируются «с чистого листа». Однако эта проблема становится существенной при переходе к промышленному программированию. На практике чаще всего возникает необходимость интегрировать реализацию сущностей со сложным поведением в уже существующие системы, а ведь огромная часть существующих промышленных программных систем написана в объектно-ориентированном стиле.

Теперь подойдем к вопросу с другой стороны: если объектная технология так хороша, как о ней пишут, то есть ли необходимость внедрять в нее автоматный подход? Дело в том, что вопросы проектирования, спецификации и реализации сложного поведения лежат за рамками того круга проблем, которые обсуждаются и решаются в объектно-ориентированном программировании. Однако на эти вопросы нельзя закрывать глаза. Как было упомянуто выше, сущности со сложным поведением встречаются в программировании повсеместно. В последнее время часто приходится слышать от гуру индустрии разработки ПО, что в связи со все возрастающей сложностью программ, необходимо использовать для их описания автоматы [59], однако широкое распространения автоматов в программировании сдерживается их конфликтом с господствующей объектно-ориентированной парадигмой.

В чем же заключается этот конфликт? В объектной технологии сущности моделируются при помощи классов. Входное воздействие для класса – это вызов компонента. Если ненадолго оставить в стороне полиморфизм, можно утверждать, что класс всегда реагирует на определенное входное воздействие выполнением одного и того же алгоритма действий, закодированного в теле соответствующего компонента. А это, как читатель, наверное, помнит, определение простого поведения. Таким образом, приходим к следующему выводу.

Класс является моделью сущности с простым поведением

Это и есть глубинная причина непригодности объектной технологии в ее первоначальном виде для моделирования сущностей со сложным поведением. Логично предположить, что моделирование сложного поведения станет более простым и непосредственным, если добавить в объектную технологию элементы автоматного программирования. Однако, как именно следует объединить две парадигмы – нетривиальный вопрос, и общепризнанного ответа на него пока не существует.

Почти все обсуждения автоматных образцов проектирования строятся вокруг вопроса «Как с помощью объектов реализовать автоматы?» Такую поверхностную постановку задачи можно объяснить тем, что авторы образцов не знакомы с

парадигмой автоматного программирования и не знают, что автоматные модели следует применять при разработке ПО систематически, для описания сложного поведения, а не от случая к случаю, как например, при программировании компиляторов и решении еще нескольких специфических задач.

Однако и в *объектно-ориентированном программировании с явным выделением состояний* на первых этапах его развития акценты были расставлены точно таким же образом. В первых работах (например, в работе [58]) по этой парадигме предполагалось, что процесс проектирования должен происходить так же, как при процедурном программировании с явным выделением состояний. Результатом этого процесса, как и раньше, должно было быть множество взаимодействующих автоматов и подпрограмм. И только после этого, уже на стадии реализации, в разработку вовлекался объектно-ориентированный подход: предпринимались попытки «втиснуть» эту совершенно чуждую объектной технологии древовидную архитектуру, основанную на функциях, в множество классов.

При этом архитектура система получается неприспособленной к модификации, непонятной, да и просто некрасивой – но это полбеды. Настоящая же беда начинается тогда, когда в стройную объектно-ориентированную систему, которая разрабатывалась долго и тщательно и имеет структуру на основе объектной декомпозиции, вдруг понадобится ввести сущность со сложным поведением. Для объектно-ориентированных систем введение новой сущности – это обыденная и безболезненная операция. Она требует добавления нового класса, и небольших изменений в тех классах, которые должны стать его клиентами. Однако при описанном подходе к сочетанию автоматной и объектно-ориентированной парадигм, вместо добавления одного класса придется перепроектировать всю систему с нуля в автоматном стиле. Не самая радужная перспектива!

Таким образом, в настоящее время в индустрии разработки программного обеспечения сложилась ситуация, когда автоматы используются для описания поведения только в крайне сложных и ответственных системах, которые проще перепроектировать с нуля с применением автоматов, чем заставить правильно работать без них. В то же время, за использование автоматов при разработке некритичного ПО приходится платить слишком высокую цену.

Корни этой проблемы в том, что автомат продолжает восприниматься как подпрограмма. Даже если эта подпрограмма «обернута» в класс и переменная с номером текущего состояния инкапсулирована в этом классе (что само по себе неплохо) [58], сути дела такая «обертка» не меняет.

ПРИМЕЧАНИЕ

> В истории объектной технологии уже была похожая попытка построить объектно-ориентированный вариант некоторой концепции, просто обернув подпрограмму в класс. Речь идет о так называемых *активных объектах* – одной из моделей параллельных вычислений, которая основывается на отождествлении объекта и процесса (или потока). У класса, порождающего активные объекты, есть главная подпрограмма, выполнение которой отождествляется с работой процесса (потока).
>
> Модель активного объекта применяется в некоторых, в том числе, популярных языках программирования, однако, ее использование связано с рядом проблем (например, известная проблема *аномалии наследования*). Причина состоит в том,

> что эта модель не соответствует объектно-ориентированным концепциям. Класс должен описывать сущность, предоставляющую набор сервисов, а не программу.

Таким образом, модель «автомат как подпрограмма» оказалась несостоятельной в рамках объектно-ориентированного подхода.

Исторически следующей в объектно-ориентированном программировании с явным выделением состояний появилась концепция «автоматы и объекты управления как классы». Такая модель принята, например, в инструментальном средстве автоматного программирования *UniMod* (разд. 3.3.2). Архитектура системы со сложным поведением, построенная по такому принципу, изображена на рис. 3.1.

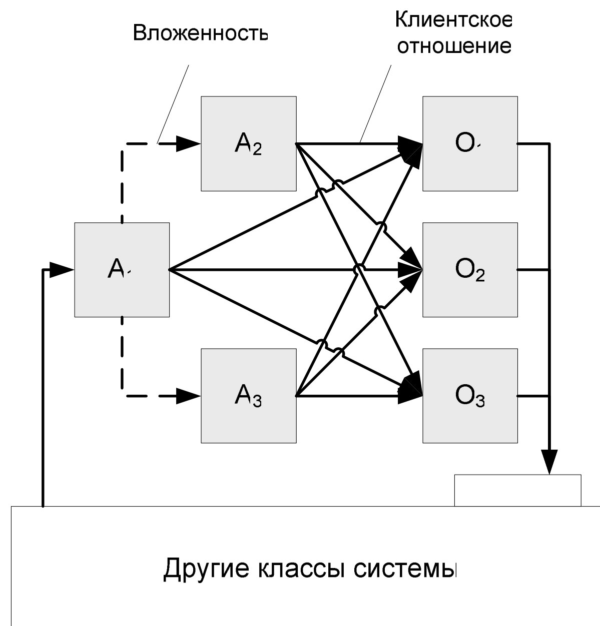

Рис. 3.1. Архитектура системы при подходе «автоматы и объекты управления как классы»

Объекты управления довольно естественно превращаются в классы, у них все для этого есть: запросы, команды и вычислительное состояние, которое можно описать с помощью множества атрибутов класса. Сопоставление отдельного класса каждому объекту управления восстанавливает справедливость: теперь усилия разработчиков по выделению объектов управления на стадии моделирования не пропадают даром на этапе реализации. Каждый запрос или команда теперь имеют доступ только к строго определенной части вычислительного состояния.

Гораздо интереснее вопрос, как сделать классы из автоматов. Вспомним, что существует несколько различных автоматных моделей. Например, активные автоматы довольно плохо согласуются с концепцией класса и являются, скорее, подпрограммами (или даже активными объектами, о которых было упомянуто выше). Лучше всего подходят на роль классов пассивные автоматы Мили. Сервисы, которые они предоставляют – это события, которые они обрабатывают. Тела

компонентов представляют собой реализацию функций переходов и выходов. При этом классы автоматов являются клиентами классов, реализующих объекты управления, и вызывают запросы и команды последних из своих компонентов. Клиентское отношение на рис. 3.1 изображено в соответствии с нотацией *BON* [60] – стрелками, направленными от клиента к поставщику.

Изложенный метод проектирования систем со сложным поведением, конечно, является более объектно-ориентированным, однако и он обладает рядом недостатков. Во-первых, он поддерживает автоматную декомпозицию и отношение вложенности между автоматами. Это отношение обладает специфической семантикой и не имеет аналогов среди межмодульных отношений в объектно-ориентированном мире. Введение такого отношения в структуру системы неоправданно усложняет ее.

Во-вторых, в соответствии с концепциями объектной технологии классы должны описывать самостоятельные, четко определенные абстракции, которые имеют смысл и вне того контекста, где они были первоначально определены. Но является ли автомат самостоятельной абстракцией? Может ли автомат управления клапаном в другой системе начать управлять кофеваркой? При внимательном изучении становится ясно, что самостоятельной абстракцией в автоматной архитектуре является не отдельно взятый автомат, а вся сущность со сложным поведением в целом.

Для того, чтобы реализация сущности со сложным поведением органично вписывалась в объектно-ориентированную систему, модель такой сущности должна быть согласована с объектно-ориентированными концепциями, а следовательно, как и модель любой другой сущности, она должна быть классом.

Моделью сущности со сложным поведением должна быть специальная разновидность класса

Так же, как и обыкновенный класс, эта модель должна предоставлять клиентам набор сервисов, соответствующих событиям, которые обрабатывает автомат. Однако в подводной части айсберга, скрытой от клиентов (в своей реализации) модель сущности со сложным поведением должна отличаться от обыкновенного класса. При вызове компонента выбор выполняемого действия (тела компонента) должен осуществляться в зависимости от управляющего состояния.

В разд. 1.4.3 была построена математическая модель *автоматизированного объекта управления*. Для сущности со сложным поведением эта модель – практически то же, что и абстрактный тип данных для сущности с простым поведением. Различие состоит лишь в том, что в автоматизированном объекте отсутствует в явном виде описание семантики операций объекта управления. Возможно, этот недочет в будущем будет исправлен. Пожалуй, пора раскрыть карты и признаться читателю, что модель автоматизированного объекта управления строилась [61, 62] как частный случай абстрактного типа данных. Это всего лишь АТД со специфическим внутренним устройством (АТД, содержащий автомат).

Поскольку реализацией абстрактного типа данных в объектной технологии является класс, то автоматизированный объект управления естественно реализовать с помощью класса. Так мы приходим к следующей (и последней на сегодняшний день) концепции объектно-ориентированного программирования с явным выделением

состояний, которую можно назвать «автоматизированные объекты управления как классы» (рис. 3.2).

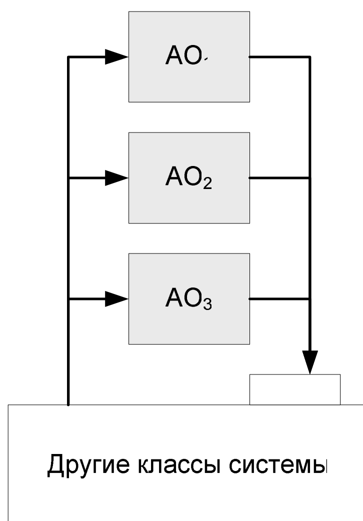

Рис. 3.2. Архитектура системы при подходе «автоматизированные объекты управления как классы»

ПРИМЕЧАНИЕ

> Здесь может возникнуть путаница с терминами: объектами в объектно-ориентированном программировании называются экземпляры класса. В то же время автоматизированный *объект* управления (как и просто *объект* управления) – это скорее класс, чем объект. Эта путаница возникла в результате конфликта в терминов в объектной технологии и теории управления. В данной главе, когда будем говорить о программном, а не о математическом описании, будем употреблять для модели сущности со сложным поведением термин *автоматизированный класс*.

Эта концепция полностью согласуется и с объектно-ориентированными принципами, и с парадигмой автоматного программирования. Автоматы здесь используется только для описания логики сущностей со сложным поведением, а сущности с простым поведением, присутствующие в той же системе, описываются обыкновенными классами.

До сих пор речь шла лишь о том, что автоматизированный класс должен выглядеть как обыкновенный класс с точки зрения клиентов[15], но о внутреннем устройстве этого класса пока ничего не известно. В частности, не утверждается, что объекту управления не следует сопоставить отдельный класс. Напротив, объект управления

[15] Для модулей объектно-ориентированной системы так же естественно общаться с другими классами посредством вызова компонентов, как для вас – общаться с другими людьми с помощью речи, мимики и жестов. Пытаясь заставить класс взаимодействовать не с другим классом, а с автоматом, представьте, каково было бы вам общаться с пришельцем с другой планеты, зная лишь понаслышке о логике его мышления и способах коммуникации.

часто представляет собой самостоятельную абстракцию. В этом случае можно предложить следующий образец проектирования автоматизированных объектов управления (рис. 3.3) .

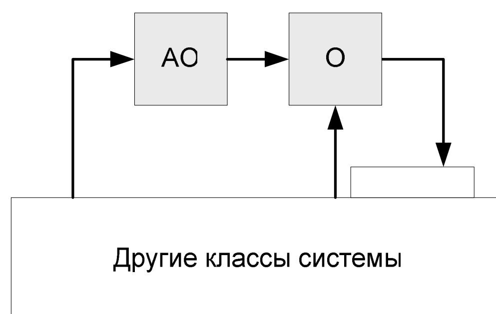

Рис. 3.3. Автоматизированный объект управления и объект управления как классы

Здесь и объект управления и автоматизированный объект управления представлены классами, между которыми существует клиентское отношение. При этом классы-клиенты сущности со сложным поведением обращаются к автоматизированному классу (как к обыкновенному). Если же на более низком уровне абстракции в системе есть сущности, которые нуждаются в сервисах объекта управления, они обращаются напрямую к нему.

Этот образец проектирования не является единственно возможным. В том случае, если объект управления не представляет собой достаточно самостоятельной абстракции так, что его нецелесообразно выделять в отдельный класс, реализацию запросов и команд объекта управления можно поместить непосредственно в автоматизированный класс. Более того, часть или все операции объекта управления могут быть реализованы в других классах системы. Другими словами, у автоматизированного класса может быть не один непосредственный поставщик (как на рис. 3.3), а несколько (рис. 3.4). В этом случае объект управления не представлен непосредственно модулем программной системы, он существует на более высоком уровне абстракции. При необходимости отдельный класс для объекта управления легко ввести: это будет всего лишь адаптер, транслирующий вызовы компонентов в автоматизированном классе его поставщикам.

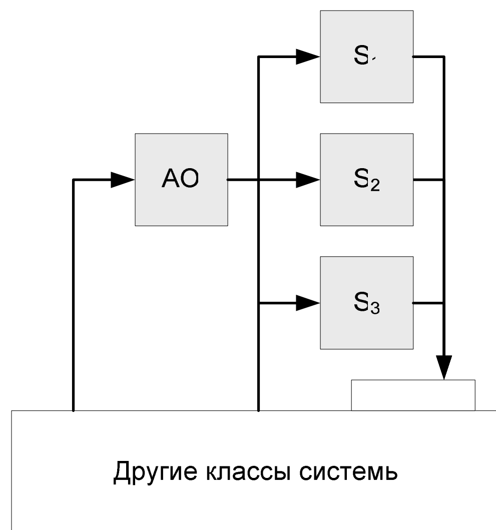

Рис. 3.4. Автоматизированный объект управления и несколько классов-поставщиков

У тех разработчиков, которые мыслят категориями программирования с явным выделением состояний, может возникнуть вопрос: всегда ли возможно обойтись без автоматной декомпозиции? Что если в результате выделения сущностей со сложным поведением автомат, управляющий некоторым объектом управления, получился слишком сложным?

В рамках подхода «автоматизированные объекты управления как классы» предлагается всегда использовать только один вид декомпозиции, который уже существует в объектной технологии – объектную декомпозицию. Это означает, что если автоматизированный класс слишком сложен, его необходимо разбить на несколько классов, обыкновенных или автоматизированных.

В связи с этим обычно возникают возражения двух типов. К первому типу относятся замечания, что объект управления может быть спроектирован заранее (разработан другой компанией, реализован аппаратно) и в этом случае нет возможности провести его декомпозицию. Возразим, что в этом случае объект управления неделим на уровне реализации, однако при построении архитектуры системы со сложным поведением ничто не мешает разработчикам представить его как составной. Например, пусть заранее разработан низкоуровневый эмулятор микроволновой печи, представляющий собой один класс, который, в том числе, имеет компоненты «Открыть замок дверцы» и «Запустить таймер». Это не мешает при разработке системы управления печью считать, что в составе печи есть замок дверцы и таймер, и разработать по автомату для управления каждым из этих объектов в отдельности. На этапе реализации необходимо учесть, что обе эти абстракции были совмещены в одном классе (например, можно ввести в систему классы-адаптеры или просто сопоставить сервисам замка и таймера различные компоненты класса микроволновой печи). Эта проблема – следствие выбора не совсем подходящих компонент для повторного использования. Она не специфична для сложного поведения и применения автоматов. Такое часто бывает в объектно-ориентированном

программировании, если повторно используемые компоненты были спроектированы неидеально.

Ко второму типу относятся замечания о том, что, если логика поведения сущности сложна, причем во всех управляющих состояниях происходит обращение ко всем компонентам объекта управления, то разбить автоматизированный объект на несколько независимых автоматизированных объектов невозможно. Такая ситуация возникает редко, но и эту проблему можно решить, не прибегая к изменению модели сущности со сложным поведением и не нарушая стройности объектной технологии.

В разд. 1.4.3 упоминалось, что автоматное программирование поддерживает концепцию выделения уровней абстракции, позволяя объекту управления самому быть автоматизированным. В процедурном программировании с явным выделением состояний эта конструкция косвенно могла быть реализована с помощью иерархической автоматной декомпозиции: вложенные и вызываемые автоматы, как бы, являлись объектами управления для автомата более высокого уровня. Такой способ выделения уровней абстракции лишен единообразия: автоматы и объекты управления – это все-таки разные сущности и взаимодействовать с ними приходится по-разному, что ведет к усложнению модели.

При объектно-ориентированном подходе концепция вложенных автоматизированных объектов управления реализуется непосредственно. Интерфейс автоматизированного класса ничем не отличается от любого другого. Поэтому ничто не мешает классу объекта управления быть автоматизированным. Таким образом, здесь для выделения уровней абстракции используется стандартное клиентское отношение между классами, модель не усложняется дополнительными сущностями и отношениями (вложенными и вызываемыми автоматами), а результирующая архитектура получается более модульной (готовый автоматизированный класс можно использовать в качестве объекта управления в любом другом автоматизированном классе).

В завершение обсуждения опишем процесс проектирования системы со сложным поведением в соответствии с концепцией «автоматизированные объекты как классы» в виде алгоритма (по аналогии с главой 2, где подобный алгоритм был приведен для процедурного программирования с явным выделением состояний).

1. Производится объектная декомпозиция: программная система представляется в виде множества взаимодействующих сущностей, каждая из которых является самостоятельной, четко определенной абстракцией.
2. Каждой сущности сопоставляется класс, определяются интерфейсы классов и отношения между ними.
3. Из числа сущностей выделяются те, которые обладают сложным поведением: для описания именно этих сущностей будет применяться автоматный подход.
4. Для каждой сущности со сложным поведением в отдельности:
   - строится набор управляющих состояний;
   - компоненты, заданные в интерфейсе, сопоставляются событиям управляющего автомата;

- запросы и команды объекта управления сопоставляются, соответственно, входным и выходным переменным управляющего автомата;
- на основе управляющих состояний, событий, входных и выходных переменных строится управляющий автомат.

5. Обыкновенные, неавтоматизированные классы системы реализуются непосредственно на выбранном объектно-ориентированном языке программирования. Код автоматизированных классов может либо генерироваться автоматически на основе диаграмм переходов, либо строиться вручную с помощью одного из известных образцов проектирования (разд. 3.3).

Обратим внимание читателя на то, что в приведенном выше алгоритме существенными являются лишь аспекты проектирования и реализации сложного поведения. Предложенный метод не ограничивает программиста в выборе модели процесса разработки (водопадная, итеративная, кластерная и т. п.). Приведенный выше алгоритм легко модифицировать таким образом, чтобы разработка системы происходила в несколько итераций, а также чтобы это была не разработка «с нуля», а внесение изменений в уже существующую объектно-ориентированную систему.

3.2. Спецификация

В объектно-ориентированном мире существуют различные методы спецификации. Среди них есть полноценные языки моделирования, поддерживаемые графическими нотациями, такие как *UML* и *BON* [60]. Кроме того, известны и чисто текстовые нотации: самостоятельные (например, *Larch* [63]) или встроенные в универсальные языки программирования (такие как *Eiffel* [64]). Попробуем разобраться, какие языки и методы спецификации из всего этого разнообразия могут быть полезными для автоматного описания сложного поведения.

В процедурном программировании с явным выделением состояний используется всего два вида спецификаций (структуры и поведения), и они позволяют достаточно полно описать систему со сложным поведением. Для спецификации структуры использовались схемы связей и диаграммы взаимодействия автоматов, а для спецификации поведения – графы переходов.

Как отмечено в предыдущем разделе, при объектно-ориентированном подходе к проектированию сущностей со сложным поведением часто следует учитывать контекст – уже существующую объектно-ориентированную систему, в которую требуется внедрить новую сущность. Важно, чтобы не только модель сущности со сложным поведением, но и язык ее спецификации был согласован с существующей системой. Поэтому лучшим решением будет являться применение уже существующих и широко используемых в объектной технологии языков.

Практически все существующие языки спецификации обладают развитыми средствами для описания статической модульной структуры системы: классов и отношений между ними. В наиболее богатом и популярном в настоящее время языке *UML* помимо средств описания структуры предлагаются также средства моделирования использования и поведения. С другой стороны, текстовые языки формальной спецификации в дополнение к структуре, как правило, обеспечивают средства декларативного описания семантики программных модулей (например, в виде *контрактов*). Отметим, что возможность описания семантики предусмотрена и

для моделей *UML*: для этих целей разработан специальный *объектный язык ограничений* (*OCL*), который, однако, в настоящее время не пользуется большой популярностью.

Вопросы моделирования использования выходят за рамки этой книги. Этот вид моделирования рассматривает систему с точки зрения ее пользователей. При этом сложное поведение не играет никакой роли (по крайней мере, к этому следует стремиться).

Спецификацию структуры так или иначе осуществляют все разработчики объектно-ориентированных программных систем вне зависимости от применяемого языка и метода. Не выявив классы и не задав отношения между ними, систему просто невозможно реализовать. Различие состоит в том, что для одних разработчиков процесс проектирования и описания модели представляет собой отдельную фазу создания программной системы, а для других он происходит незаметно, в уме, и результаты этого процесса не фиксируются.

Должна ли быть спецификация структуры системы текстовой или графической – это в большой степени дело вкуса. Конечно, графическое представление структуры (во всех известных авторам нотациях оно называется *диаграммой классов*) более наглядно. Однако пользовательский интерфейс всех существующих на сегодняшний день инструментов визуального проектирования программ [65] оставляет желать много лучшего. Поэтому визуальное конструирование модели оказывается процессом гораздо более трудным и длительным по сравнению с ее текстовым описанием. Пока инструментов с лучшим интерфейсом не существует, оптимальным представляется вариант среды разработки, которая по модели системы может строить различные ее виды (в том числе, текстовые и графические), причем через любой из них можно вносить изменения в модель. В таком случае наглядность диаграммы можно совместить с удобством редактирования текстового описания, а кроме того, угодить различным вкусам разработчиков.

Как изменится описание статической структуры системы с появлением в ней автоматизированных классов? Выше было неоднократно упомянуто, что внешне автоматизированные классы ничем не отличаются от других. Поэтому та часть диаграммы классов, которая описывает связи автоматизированных классов с их клиентами, никак не изменится. С другой стороны, на диаграмме классов можно особым образом отобразить взаимодействие автоматизированного класса с его объектом управления. Богатые языки спецификации, такие как *UML*, позволяют превратить этот участок диаграммы классов, фактически, в схему связей (рис. 3.5).

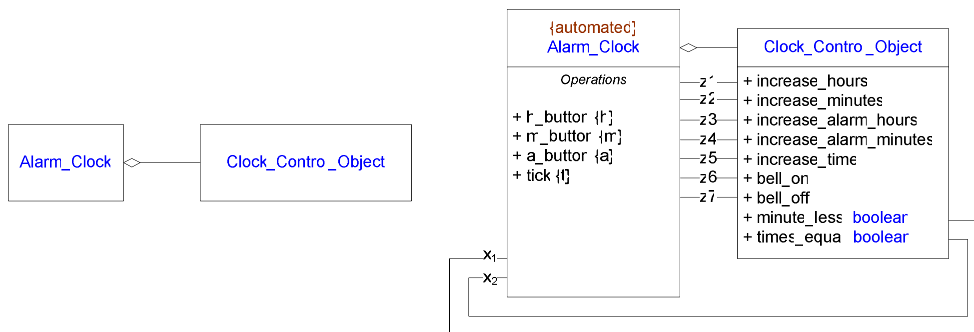

Рис. 3.5. Спецификация взаимодействия часов с будильником и их объекта управления на диаграмме классов: кратко (слева) и по образцу схемы связей (справа)

На этой диаграмме автоматизированный класс связан с запросами и командами объекта управления с помощью ассоциаций, помеченных идентификаторами входных и выходных переменных[16]. Компоненты самого автоматизированного класса помечены краткими идентификаторами событий. Для наглядности к классу `Alarm_Clock` добавлено ограничение `automated`. При этом из диаграммы видно, что этот класс является автоматизированным. Напомним, что краткие идентификаторы событий, входных и выходных переменных в автоматном программировании вводятся для обеспечения компактности графа переходов, и они необязательно будут фигурировать в тексте программы, особенно если этот текст генерируется по графическому описанию автоматически. Располагая графом переходов и приведенной выше диаграммой классов, генератор кода сможет поместить переходы, помеченные событиями, в тела соответствующих компонентов автоматизированного класса и заменить обращения автомата к входным и выходным переменным вызовами соответствующих компонентов объекта управления.

В предыдущем разделе было упомянуто, что объект управления не всегда следует выделять в отдельный класс. Это замечание актуально для часов с будильником, так как их объект управления представляет собой недостаточно четко определенную абстракцию. Возможно, лучший вариант архитектуры представлен на рис. 3.6.

[16] На самом деле, компоненты классов по правилам *UML* не являются сущностями, и соединить их ассоциациями невозможно. Поэтому формально можно считать, что каждая ассоциация связывают два класса (автоматизированный и класс объекта управления), имеет имя, совпадающее с идентификатором входной или выходной переменной, и, кроме того снабжена помеченным значением в виде имени соответствующего компонента объекта управления.

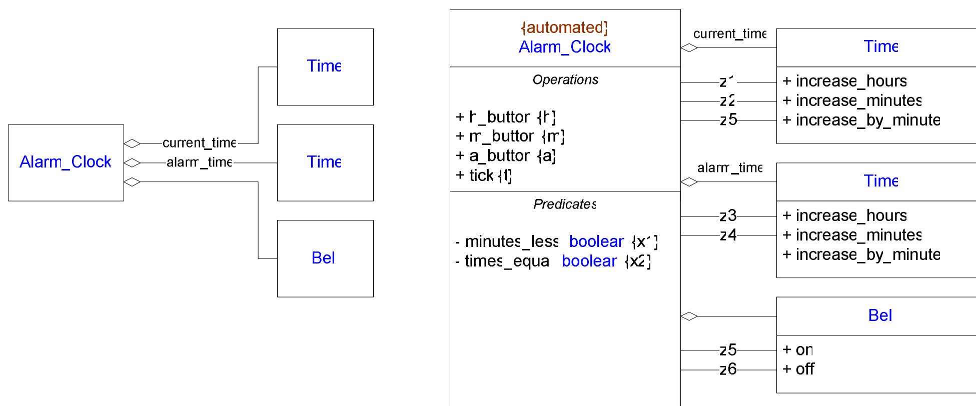

Рис. 3.6. Краткая (слева) и полная (справа) диаграммы классов часов с будильником: объект управления не выделен в отдельный класс

Здесь поставщиками событий для часов с будильником являются классы `Time` (время) и `Bell` (звонок). Запросы объекта управления специфичны для часов с будильником и поэтому реализованы в самом автоматизированном классе. Их связь с краткими идентификаторами входных переменных, по аналогии с событиями, описана с помощью помеченных значений.

Итак, схеме связей нашлось место в объектно-ориентированном программировании с явным выделением состояний. Что касается диаграммы взаимодействия автоматов, то она здесь бесполезна, поскольку взаимодействие автоматизированных объектов управления можно отобразить на стандартной диаграмме классов.

Перейдем к вопросам моделирования поведения, которые относятся к теме этой книги самым непосредственным образом. Здесь следует различать *декларативную* и *императивную* спецификацию поведения. Декларативная спецификация описывает поведение сущности с точки зрения ее клиентов, отвечает на вопрос «Что можно ожидать от этой сущности?» Примером такой спецификации являются контракты.

Императивная спецификация, напротив, рассматривает поведение с точки зрения самой сущности и отвечает на вопрос: «Как достичь того, что ожидают клиенты?» Простейшим примером такой спецификации является текст программы на императивном языке. Однако, как читатель уже имел возможность убедиться, текстовое описание сложного поведения не обладает достаточной наглядностью. Поэтому императивная спецификация сложного поведения должна выполняться в виде графа переходов, диаграммы состояний или другой аналогичной графической нотации.

Для того чтобы не нарушать модульную структуру системы, созданную при объектной декомпозиции, необходимо строить отдельную диаграмму переходов для каждой сущности со сложным поведением (каждого автоматизированного класса). Отметим, что в языке *UML* (по крайней мере, в первой его версии) такой рекомендации по использованию диаграмм состояний дано не было, несмотря на то, что язык по своей сути является объектно-ориентированным. В результате диаграммы состояний применялись для описания поведения системы в целом, или произвольной ее части, что не позволяло органично встроить эти описания в общую

модель системы и интегрировать с другими диаграммами. По мнению авторов, это и явилось основной причиной, по которой диаграммы состояний долгое время использовались в объектной технологии только в качестве документации, а не как полноценные формальные спецификации. При этом в результате опроса, проведенного *С. Орликом* (*Borland*), было установлено, что даже в тех случаях, когда архитекторы программной системы создают диаграммы состояний, при переходе к реализации разработчики их обычно не используют.

Если следовать приведенной выше рекомендации, исчезает разница между графами переходов, диаграммами состояний и другими нотациями, поддерживающими различные расширения базовой модели конечного автомата. Поскольку объектная декомпозиция заменила автоматную, больше нет необходимости ни во вложенных и вызываемых автоматах, ни в ортогональных и суперсостояниях (хотя группировку состояний, безусловно, можно использовать как косметическое средство для улучшения внешнего вида диаграмм). Все, что требуется от графической нотации – это состояния и переходы между ними с условиями (в виде булевых формул от событий и входных переменных) и действиями (в виде одиночных выходных переменных или их последовательностей).

Поговорим немного о декларативном описании сложного поведения. Этот вопрос в настоящее время изучен гораздо меньше. Причина этого, по мнению авторов этой книги, в том что, круг исследователей, которые интересуются формальными спецификациями и контрактами, практически не пересекается с кругом тех, кто озабочен проблемами сложного поведения.

Контракт является частью интерфейса класса и, в основном, предназначен для его клиентов. Если обыкновенные классы в системе снабжены контрактами, то и автоматизированные классы должны иметь контракты точно такого же вида. Однако составление содержательного контракта для сущности со сложным поведением может быть непростой задачей. Спецификации компонентов этой сущности должны отражать ее реакцию во всех возможных управляющих состояниях.

Например, если вспомнить уже знакомые читателю часы с будильником, каким должно быть постусловие компонента `h`? Оно может быть, например, следующим: «Если часы в первом или в третьем состоянии, то увеличилось число часов текущего времени, а если во втором – то увеличилось число часов срабатывания будильника». Такой контракт очень содержателен, однако он раскрывает внутреннюю информацию об управляющих состояниях. Для клиентов класса такой контракт, скорее всего, будет являться *чрезмерной спецификацией*. Можно предложить другой вариант: «Увеличилось либо число часов текущего времени, либо число часов срабатывания будильника», или: «Число минут не изменилось, а число часов текущего времени и число часов срабатывания будильника не могли оба увеличиться». Будет ли такая спецификация достаточной? Что именно необходимо знать клиенту о работе компонента `h`? Стандартных рекомендаций по этому поводу не существует, по крайней мере, на данный момент.

Каким бы ни был контракт автоматизированного класса, его поведение в каждом из управляющих состояний в отдельности является частным случаем общего, сложного поведения. Поэтому если составить контракт на поведение сущности в одном конкретном состоянии, то он должен быть сильнее, чем общий контракт на сложное поведение. Это условие аналогично *принципу подстановочности*, который

используется при определении наследования подтипов: поскольку потомок является частным случаем предка, то спецификация потомка должна быть сильнее спецификации предка. Из этой аналогии следуют два интересных вывода. Во-первых, сложное поведение в объектно-ориентированном контексте можно считать особой формой полиморфизма. Сущность со сложным поведением подобна полиморфной сущности некоторого типа, к которой во время выполнения могут быть присоединены объекты различных его подтипов (каждый подтип соответствует одному управляющему состоянию).

Во-вторых, такое понимание сложного поведения приводит напрямую к образцу проектирования *State* [66], который как раз и предполагает реализацию сложного поведения с помощью классического полиморфизма подтипов. Единственное различие состоит в том, что в этом образце сложное поведение реализуется за счет порождения подтипов класса, описывающего управляющее состояние, а не саму сущность со сложным поведением. Этот дополнительный уровень косвенности необходим, так как иначе невозможно реализовать механизм переходов между состояниями. Подробнее образец проектирования *State* рассмотрен в следующем разделе, посвященном вопросам реализации.

Вопрос о том, имеются ли противоречия между моделью автоматизированного класса и концепциями проектирования по контракту (особенно с учетом механизма наследования), до конца еще не изучен. Ряд авторов занимался вопросами наследования автоматов [67, 68], поскольку проблема повторного использования описания логики сложного поведения очень актуальна. Однако говорить о наследовании *автоматов* не совсем корректно, так как автомат не является самостоятельной абстракцией. Напротив, очень плодотворным полем для исследования является наследование автоматизированных классов.

Подведем итоги обсуждения вопросов спецификации сущностей со сложным поведением в объектно-ориентированном контексте в виде следующей общей рекомендации.

Для каждого автоматизированного класса в системе необходимо построить:

- описание его интерфейса (в текстовом и/или графическом виде с использованием того языка, на котором описаны интерфейсы остальных классов системы);
- схему связей: сопоставление кратких идентификаторов событий, входных и выходных переменных компонентам автоматизированного класса и объекта управления (в том случае, если на диаграмме переходов используются краткие идентификаторы);
- диаграмму переходов управляющего автомата;
- спецификацию объекта управления.

Такую спецификацию возможно автоматически преобразовать в текст на объектно-ориентированном языке программирования или непосредственно интерпретировать во время выполнения программной системы.

Для того чтобы проиллюстрировать все основные особенности предложенной технологии проектирования и спецификации, рассмотрим несколько более сложный пример, чем те, что приводились в этой книге ранее.

Пусть требуется разработать клиентскую часть системы онлайн-бронирования авиабилетов. Бронирование происходит в три этапа. Пользователь:

- задает набор критериев, которым должен удовлетворять рейс;
- выбирает из списка всех рейсов, удовлетворяющих критериям, наиболее подходящий;
- вводит личные данные, для того чтобы выполнить бронирование билетов.

Каждому из этих этапов в пользовательском интерфейсе системы сопоставим отдельный экран. Внешний вид экранов может быть, например, таким, как показано на рис. 3.7–3.9.

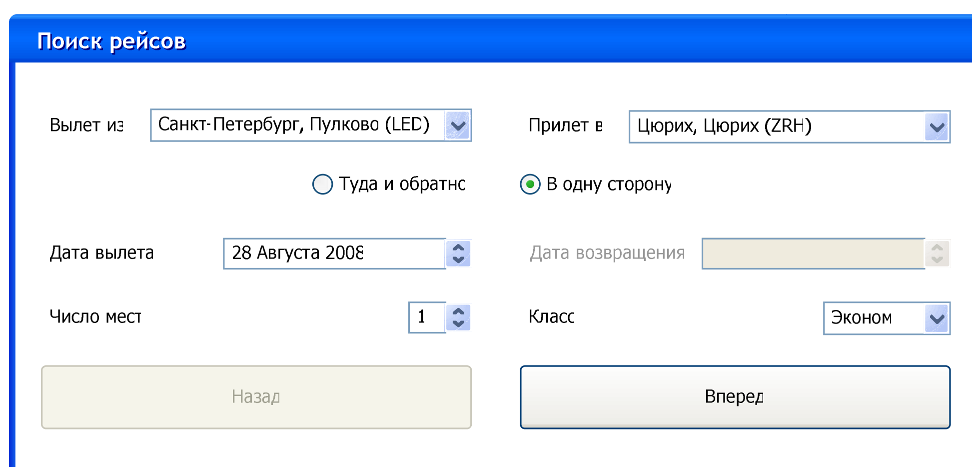

Рис. 3.7. Экран «Поиск рейсов» в системе онлайн-бронирования авиабилетов

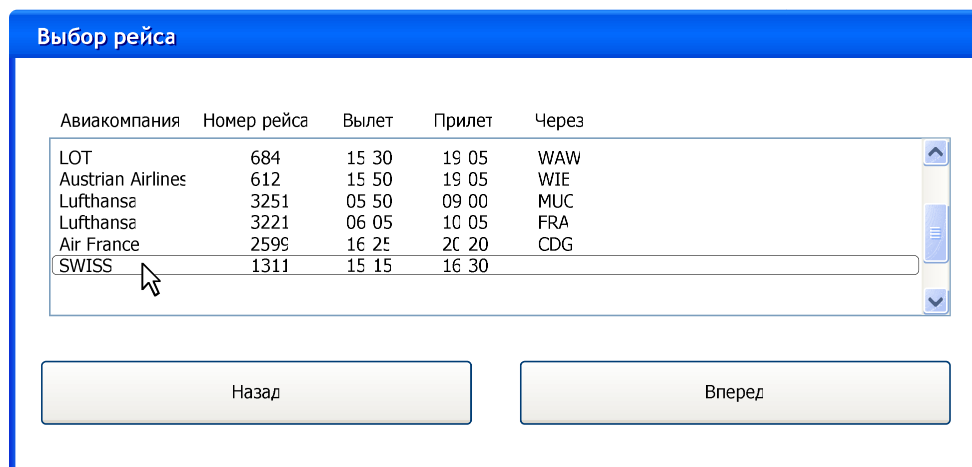

Рис. 3.8. Экран «Выбор рейса» в системе онлайн-бронирования авиабилетов

Бронирование

Имя Надежда

Фамилия Поликарпова

Сохранить Загрузить

Назад Вперед

Рис. 3.9. Экран «Бронирование» в системе онлайн-бронирования авиабилетов

Система рассчитана на постоянных пользователей. Поэтому в целях экономии времени на этапе бронирования предлагается возможность сохранения личных данных пользователя. При следующем входе пользователя в систему сохраненные значения будут использованы в качестве исходных. Если пользователь изменит личные данные на экране бронирования, то у него появится возможность либо сохранить изменения, либо отменить их, загрузив предыдущую сохраненную версию.

В соответствии с предлагаемым методом проектирования, проведем объектную декомпозицию. В системе можно выделить сущность `Application` (приложение[17]), которая отвечает за поведение сервиса в целом и за переходы между этапами. Выделим также сущность `Reservation_Data`, в которой инкапсулируем все данные о текущем заказе: идентификатор рейса, число мест, класс и личные данные пользователя. Кроме того, пусть эта сущность также осуществляет запросы к базе данных рейсов и заказов. Поскольку приложение имеет графический пользовательский интерфейс, выделим сущность `GUI`, которая обладает стандартной функциональностью, предоставляемой библиотекой поддержки графического интерфейса. Посредством этой сущности можно делать запросы к элементам управления и давать им команды. Кроме того, эта сущность сообщает приложению обо всех действиях пользователя. Наконец, сопоставим отдельную сущность каждому этапу процесса бронирования или *экрану*: `Enquiry` – для поиска рейсов, `Choice` – для выбора рейса, `Reservation` – для собственно бронирования. Общий взгляд на архитектуру системы отражен на диаграмме классов (рис. 3.10).

---

[17] Разрабатываемая система является, скорее, веб-сервисом. Термин «приложение» использован как более привычный.

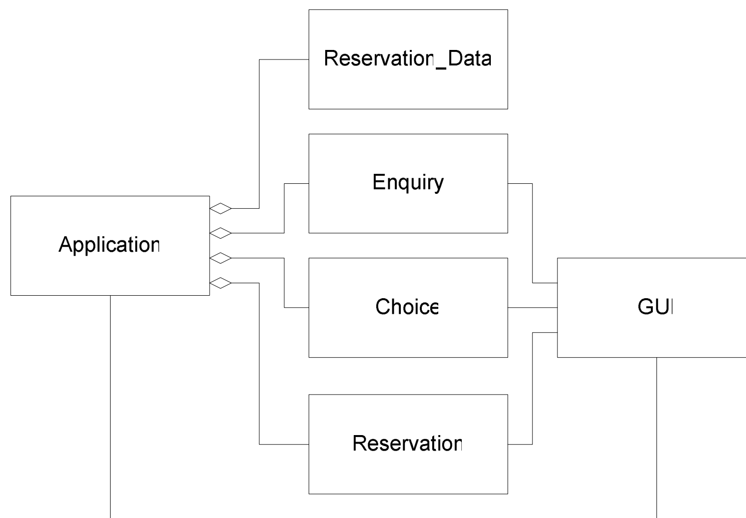

Рис. 3.10. Диаграмма классов системы онлайн-бронирования авиабилетов

Какие из выделенных сущностей обладают сложным поведением? Хороший кандидат – сущность `Application`. Она отвечает за переходы между этапами, кроме того, в зависимости от этапа нажатие одних и тех же кнопок («Вперед» и «Назад») производит разный эффект. Сложное поведение также наблюдается на этапе бронирования (сущность `Reservation`). Здесь возможность сохранить или загрузить личные данные зависит от предыстории (от того, изменял ли пользователь эти данные в текущей сессии работы с приложением). Этим двум сущностям сопоставим автоматизированные классы, а остальным – обыкновенные.

Начнем с описания автоматизированного класса `Application`. Он имеет достаточно богатый интерфейс, поскольку именно этому классу сущность `GUI` сообщает обо всех действиях пользователя. Большинство вызовов компонентов класс `Application` без изменения передает на обработку текущему экрану. Реализация такого поведения тривиальна, но громоздка. Ниже при спецификации структуры и поведения будем учитывать только те компоненты класса `Application`, которые представляют особый интерес. Этот класс управляет переключением экранов, а также данными о заказе. Такой объект управления не является достаточно самостоятельной абстракцией. Поэтому не будем выделять его в отдельный класс. Часть команд объекта управления будет реализована непосредственно в автоматизированном классе `Application`, а остальные – в его классах-поставщиках. Схема связей для этого автоматизированного класса приведена на рис. 3.11, а диаграмма переходов – на рис. 3.12.

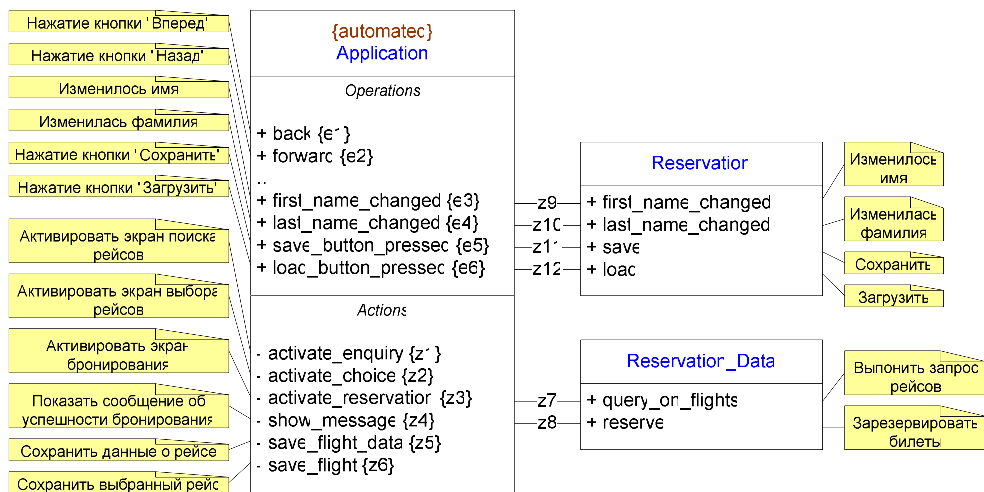

Рис. 3.11. Схема связей автоматизированного класса `Application`

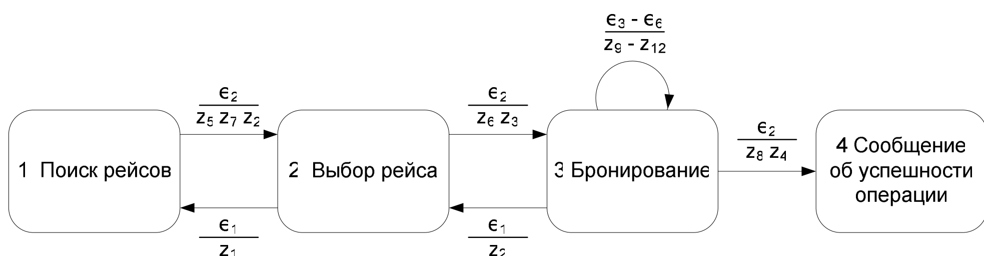

Рис. 3.12. Диаграмма переходов автоматизированного класса `Application`

На диаграммах показано, как действия пользователя транслируются классом `Application` в вызовы компонентов экрана `Reservation`: события e_3–e_6 в третьем состоянии напрямую сопоставляются выходным переменным z_9–z_{12}. Подобная трансляция имеет место и для других экранов, однако на диаграммах для краткости она не отображена.

Перейдем к рассмотрению автоматизированного класса `Reservation`. Он управляет некоторыми элементами пользовательского интерфейса и личными данными пользователя. Такой объект управления также не стоит выделять в отдельный класс. Схема связей и диаграмма переходов класса `Reservation` изображены, соответственно, на рис. 3.13 и рис. 3.14.

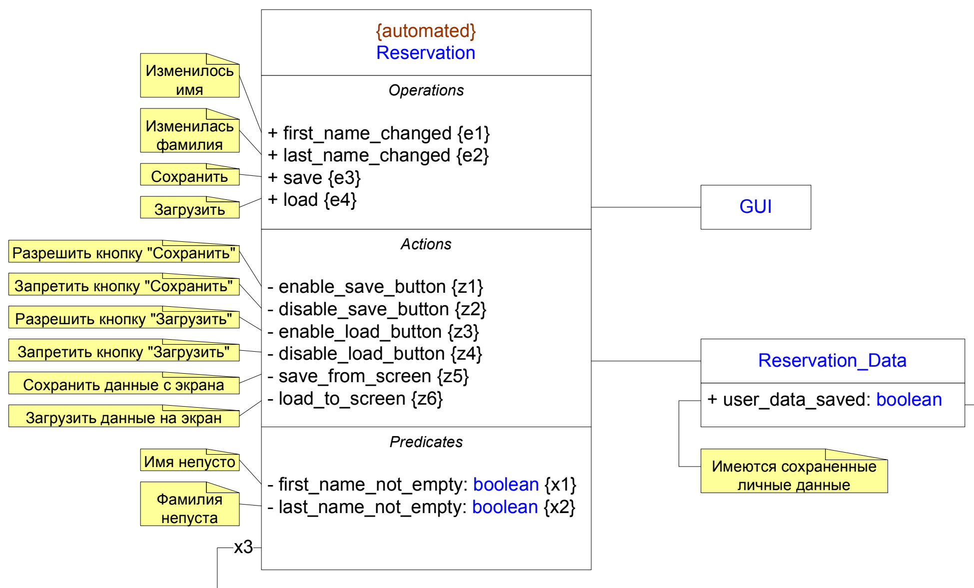

Рис. 3.13. Схема связей автоматизированного класса `Reservation`

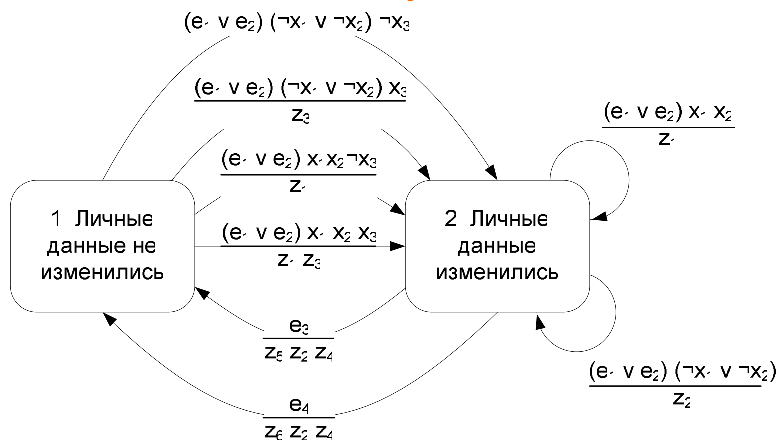

Рис. 3.14. Диаграмма переходов автоматизированного класса `Reservation`

Как мог убедиться читатель, подход к проектированию и спецификации, основанный на автоматизированных классах, позволил построить простую и элегантную архитектуру в чисто объектно-ориентированном стиле.

3.3. Реализация

Конечно, лучший подход к реализации сущностей со сложным поведением – это автоматическая генерация кода по диаграммам. Однако не для всех языков программирования и не для всех случаев жизни есть подходящие инструментальные средства. Поэтому в разд. 3.3.1 рассмотрены примеры шаблонов, которые можно использовать для реализации автоматизированных классов вручную. В разд. 3.3.2

описано наиболее мощное из существующих на сегодняшний день инструментальных средств, поддерживающих объектно-ориентированное программирование с явным выделением состояний – *UniMod* [69].

3.3.1. Шаблоны реализации автоматизированных классов

Как было упомянуто выше, существует большое число образцов проектирования, предназначенных для объектно-ориентированной реализации конечных автоматов в частности и сущностей со сложным поведением в целом. В их основе лежат уже известные читателю два способа описания автоматов в программе: статический (при помощи инструкций выбора и ветвления) и динамический (при помощи таблиц). Появление полиморфизма подтипов позволяет вместо явного создания таблицы переходов и выходов автомата использовать для этих целей таблицу полиморфных вызовов, в том или ином виде встроенную в любой объектно-ориентированный язык программирования для диспетчеризации вызовов компонентов. С точки зрения разработчика использование полиморфизма для реализации сложного поведения гораздо удобнее, чем создание таблицы вручную.

Образцы проектирования сложного поведения отличаются также способом представления элементов автоматной модели: состояний, событий, действий, переходов. С этой точки зрения одним крайним случаем является «обертывание» автоматной функции в класс, знакомое нам по предыдущей главе. При этом состояния и события описываются целочисленными константами. Это решение обладает высоким быстродействием, но крайне низкой гибкостью.

На другом полюсе находится, так называемый, шаблон *FSM Framework* [53], в соответствии с которым для каждого элемента автоматной модели строится отдельный класс. При этом класс состояния содержит таблицу, ключами в которой являются события, а значениями – переходы. Класс перехода содержит ссылку на целевое состояние и действие. Этот шаблон обеспечивает наибольшую гибкость, однако построенная таким способом программа громоздка и не очень эффективна по времени и памяти. Между двумя полюсами лежат различные комбинации описания элементов автоматной модели с помощью целочисленных констант, подпрограмм и классов. Изучить все многообразие образцов проектирования можно, обратившись к работам [53, 54].

В этой книге рассмотрим только два шаблона реализации сложного поведения, которые кажутся авторам наиболее универсальными и согласованными с моделью автоматизированного объекта управления. В контексте этой модели довольно естественно отождествлять события с компонентами автоматизированного класса. Таким образом, вопрос о представлении событий не возникает. Кроме того, как показывает опыт, для большинства задач не требуется обеспечивать возможность динамической конфигурации автомата. Поэтому отдается предпочтения статическому описанию функций переходов и действий. Методы реализации будем рассматривать на примере автоматизированного класса `Alarm_Clock`, описывающего часы с будильником, причем в целях повышения модульности выделим объект управления в отдельный класс.

Наш первый пример показывает, как можно превратить реализацию сущности со сложным поведением из процедурной в объектно-ориентированную, сделав минимум изменений. Часы с будильником теперь представляют собой отдельный класс,

компоненты которого соответствуют событиям. Их реализация выполнена уже знакомым читателю способом: с помощью инструкции выбора.

В листинге 3.1 приведена реализация объекта управления, а в листинге 3.2 – реализация самого автоматизированного класса на языке *C++*. Текст вспомогательных классов `Time` и `Bell` не относится напрямую к вопросам сложного поведения и в листингах не приводится.

Листинг 3.1. Реализация объекта управления часов с будильником

```
// Реализация объекта управления
class Clock_Control_Object {
public:
    void increase_hours() { current_time.increase_hours(); }

    void increase_minutes() { current_time.increase_minutes(); }

    void increase_alarm_hours() { alarm_time.increase_hours(); }

    void increase_alarm_minutes() { alarm_time.increase_minutes(); }

    void increase_time() { current_time.increase_by_minute(); }

    void bell_on() { bell.on(); }

    void bell_off() { bell.off(); }

    bool minute_less() {
        Time temp(current_time);
        temp.increase_by_minute;
        return temp == alarm_time;
    }

    bool times_equal() {
        return current_time == alarm_time;
    }
private:
    Time current_time;
    Time alarm_time;
    Bell bell;
}
```

Листинг 3.2. Реализация часов с будильником с помощью инструкции выбора

```
// Реализация автоматизированного класса
class Alarm_Clock {
public:
    // Инициализация внутренней переменной стартовым состоянием
    Alarm_Clock() : state(alarm_off) {}
```

```
void h_button() {
    switch (state) {
        case alarm_off:
            co.increase_hours();
            break;
        case alarm_setting:
            co.increase_alarm_hours();
            break;
        case alarm_on:
            co.increase_hours();
            break;
    }
}

void m_button() {
    switch (state) {
        case alarm_off:
            co.increase_minutes();
            break;
        case alarmSetting:
            co.increase_alarm_minutes();
            break;
        case alarm_on:
            co. increase_minutes;
            break;
    }
}

void a_button() {
    switch (state) {
        case alarm_off:
            state = alarm_setting;
            break;
        case alarm_setting:
            state = alarm_on;
            break;
        case alarm_on:
            co.bell_off();
            state = alarm_off;
            break;
    }

void tick() {
    switch (state) {
        case alarm_off:
            co.increase_time();
```

```
                break;
            case alarm_setting:
                co.increase_time();
                break;
            case alarm_on:
                if (co.minute_less()) {
                    co.bell_on();
                    co.increase_time();
                } else if (co.times_equal()) {
                    co.bell_off();
                    co.increase_time();
                } else {
                    co.increase_time();
                }
                break;
        }
    }
private:
    enum State {alarm_off, alarm_setting, alarm_on};

    State state;
    Clock_Control_Object co;
}
```

Наш второй пример использует в качестве шаблона реализации образец проектирования *State* [66]. В этом образце вариация поведения в зависимости от состояния реализуется за счет полиморфизма подтипов. Кроме класса, описывающего собственно сущность со сложным поведением, здесь создается абстрактный класс «состояние», компоненты которого также соответствуют событиям. Для каждого конкретного управляющего состояния требуется создать класс-потомок абстрактного состояния, и в его компонентах определить поведение сущности, специфичное для данного состояния. Помимо выполнения действий компоненты классов-состояний обновляют значение атрибута `next_state` (следующее состояние) – так реализована функция переходов. Схема образца проектирования *State* в форме диаграммы классов приведена на рис. 3.15.

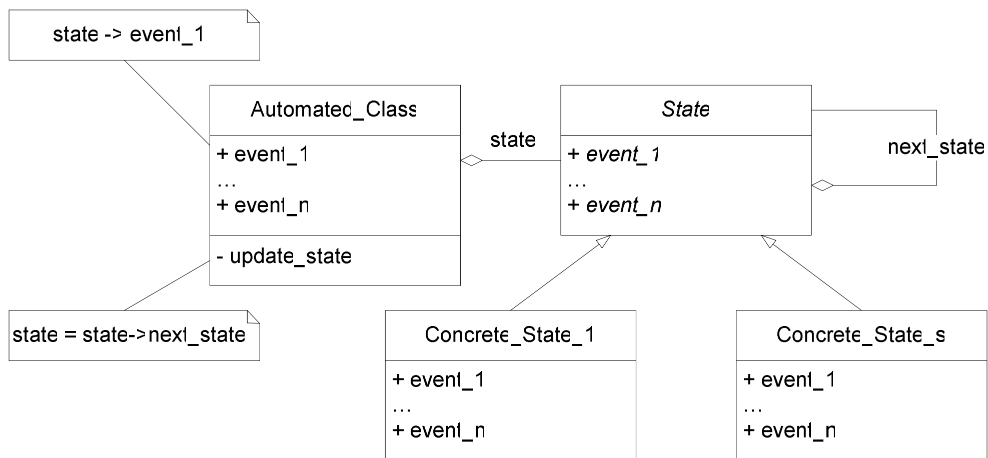

Рис. 3.15. Диаграмма классов образца проектирования State

На рис. 3.16 приведена диаграмма состояний для конкретного примера часов с будильником.

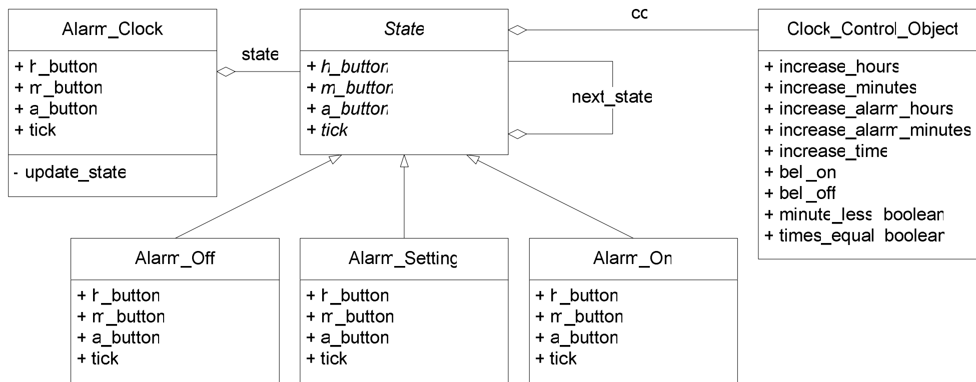

Рис. 3.16. Диаграмма классов образца проектирования State на примере часов с будильником

В листинге 3.3 содержится реализация автоматизированного класса, описывающего часы с будильником, и классов состояний. Реализация объекта управления остается прежней и поэтому не приведена.

Листинг 3.3. Реализация часов с будильником на основе образца проектирования *State*

```
// Реализация автоматизированного класса
class Alarm_Clock {
public:
    Alarm_Clock(Clock_Control_Object* control_object) :
        state(new Alarm_Off(control_object))    {}

    void h_button() {
```

```
        state->h_button();
        update_state();
    }

    void m_button() {
        state->m_button();
        update_state();
    }

    void a_button() {
        state->a_button();
        update_state();
    }

    void tick() {
        state->tick();
        update_state();
    }
private:
    State* state;

    void update_state() {
        State* next_state = state->next_state();
        delete state;
        state = next_state;
    }
}

// Абстрактный класс состояния
class State {
public:
    virtual void h_button() = 0;
    virtual void m_button() = 0;
    virtual void a_button() = 0;
    virtual void tick() = 0;
    State* next_state;
private:
    Clock_Control_Object* co;
}

// Конкретное состояние «Будильник выключен»
class Alarm_Off : public State {
public:
    Alarm_Off(Clock_Control_Object* control_object) : co(control_object) {}

    virtual void h_button() {
        co->increase_hours();
```

```
        next_state = new Alarm_Off(co);
    }

    virtual void m_button() {
        co->increase_minutes();
        next_state = new Alarm_Off(co);
    }

    virtual void a_button() {
        next_state = new Alarm_Setting(co);
    }

    virtual void tick() {
        co->increase_time();
        next_state = new Alarm_Off(co);
    }
}

// Конкретное состояние «Установка времени будильника»
class Alarm_Setting : public State {
    ... // Аналогично Alarm_Off
}

// Конкретное состояние «Будильник включен»
class Alarm_On : public State {
    ... // Аналогично Alarm_Off
}
```

В заключение раздела приведем два примера реализации с помощью образца проектирования *State* классических задач логического управления, которые были рассмотрены в разд. 1.4.2, в рамках парадигмы объектно-ориентированного программирования с явным выделением состояний. Эти примеры не имеют значительной ценности с практической точки зрения (напомним, что объектно-ориентированный подход наиболее эффективен для реализации событийных систем), однако они способствуют более глубокому пониманию связи между истоками автоматного программирования и его положением сегодня.

Итак, вновь рассмотрим последовательный двоичный одноразрядный сумматор. Граф переходов автомата Мили, реализующего сумматор, приведен на рис. 1.18. В соответствии с объектно-ориентированным подходом, выделим в отдельный класс `Position` объект управления сумматора. Объектом управления в этой задаче можно считать совокупность текущих двоичных разрядов первого и второго слагаемого, а также результата.

В этой системе изначально нет событий, а управляющий автомат является активным. В связи с необходимостью перехода к пассивной автоматной модели искусственно введем единственное событие `next_position`, которое будет сигнализировать о том, что очередные разряды первого и второго слагаемых готовы, и можно вычислять очередной разряд суммы. Диаграмма классов реализации сумматора с

помощью образца проектирования *State* приведена на рис. 3.17, а программный текст – в листинге 3.4.

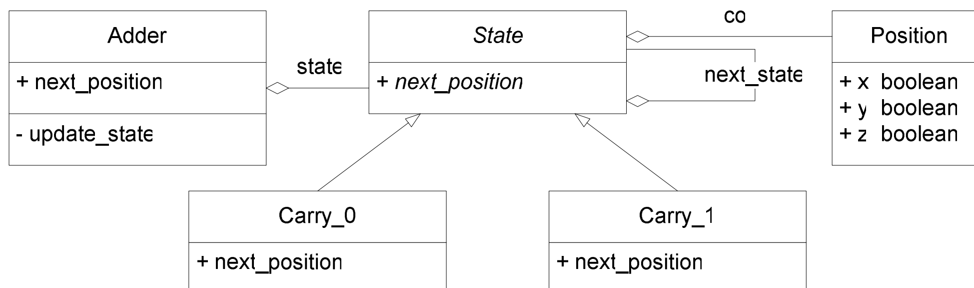

Рис. 3.17. Диаграмма классов образца проектирования State на примере последовательного двоичного одноразрядного сумматора

Листинг 3.4. Реализация последовательного двоичного одноразрядного сумматора на основе образца проектирования *State*

```
// Реализация объекта управления
struct Position {
    bool x; // Текущий разряд первого слагаемого
    bool y; // Текущий разряд второго слагаемого
    bool z; // Текущий разряд суммы
}

// Реализация автоматизированного класса
class Adder {
public:
    Adder(Position* co) : state (new Carry_0(co)) {};

    void next_position() {
        state->next_position();
        update_state();
    }
private:
    State* state;

    void update_state() {
        State* next_state = state->next_state();
        delete state;
        state = next_state;
    }
}

// Абстрактный класс состояния
class State {
public:
    virtual void next_position() = 0;
```

```
    State* next_state;
private:
    Position* co;
}

// Конкретное состояние «В переносе 0»
class Carry_0 : public State {
public:
    Carry_0 (Position* control_object) : co(control_object) {}

    virtual void next_position() {
        if (!co->x && !co->y) {
            co->z = 0;
            next_state = new Carry_0(co);
        } else if (co->x && co->y) {
            co->z = 0;
            next_state = new Carry_1(co);
        } else {
            co->z = 1;
            next_state = new Carry_0(co);
        }
    }
}

// Конкретное состояние «В переносе 1»
class Carry_1 : public State {
public:
    Carry_1 (Position* control_object) : co(control_object) {}

    virtual void next_position() {
        if (!co->x && !co->y) {
            co->z = 1;
            next_state = new Carry_0(co);
        } else if (co->x && co->y) {
            co->z = 1;
            next_state = new Carry_1(co);
        } else {
            co->z = 0;
            next_state = new Carry_1(co);
        }
    }
}
```

Перейдем к рассмотрению счетного триггера, для которого в разд. 1.4.2 был построен автомат Мура (рис. 1.16). В этой системе не будем выделять объект управления в отдельный класс. Вместо этого введем два класса, представляющих, соответственно, кнопку и лампу. Кроме того, по аналогии с предыдущим примером, введем единственное событие `tick`, которое можно отождествить с приходом сигнала от

тактового генератора. Поскольку образец проектирования *State* предназначен для реализации автоматов Мили, необходимо модифицировать автомат Мура счетного триггера, переместив действия из каждого состояния на все переходы, которые ведут в это состояние. Диаграмма классов реализации счетного триггера на основе образца проектирования *State* приведена на рис. 3.18, а программный текст – в листинге 3.5.

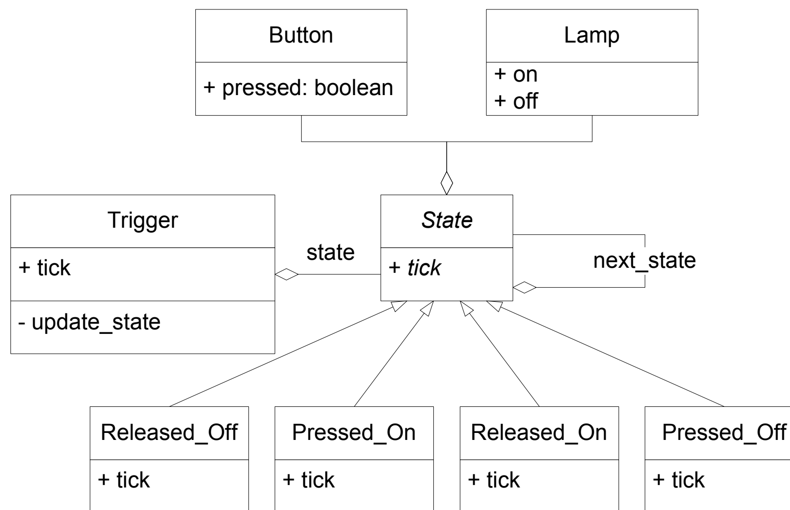

Рис. 3.18. Диаграмма классов образца проектирования State на примере счетного триггера

Листинг 3.5. Реализация счетного триггера на основе образца проектирования *State*

```
// Реализация объекта управления
struct Button {
    bool pressed;   // Нажата ли кнопка?
}

struct Lamp {
    void on();  // Подавать питание на лампу
    void off(); // Не подавать питание на лампу
}

// Реализация автоматизированного класса
class Trigger {
public:
    Trigger(Button* b, Lamp* l) : state (new Released_Off(b, l)) {};

    void tick() {
        state->tick();
        update_state();
    }
private:
    State* state;
```

```
    void update_state() {
        State* next_state = state->next_state();
        delete state;
        state = next_state;
    }
}

// Абстрактный класс состояния
class State {
public:
    virtual void tick() = 0;
    State* next_state;
private:
    Button* b;
    Lamp* l;
}

// Конкретное состояние «Кнопка отпущена, лампа выключена»
class Released_Off : public State {
public:
    Released_Off (Button* button, Lamp* lamp) : b(button), l(lamp) {}

    virtual void tick() {
        if (b->pressed) {
            l->on();
            next_state = new Pressed_On(b, l);
        } else {
            l-off();
            next_state = new Released_Off(b, l);
        }
    }
}

// Конкретное состояние «Кнопка нажата, лампа включена»
class Pressed_On : public State {
public:
    Pressed_On (Button* button, Lamp* lamp) : b(button), l(lamp) {}

    virtual void tick() {
        if (b->pressed) {
            l->on();
            next_state = new Pressed_On(b, l);
        } else {
            l->on();
            next_state = new Released_On(b, l);
        }
    }
```

```
}

// Конкретное состояние «Кнопка отпущена, лампа включена»
class Released_On : public State {
    ...     // Аналогично
}

// Конкретное состояние «Кнопка нажата, лампа выключена»
class Pressed_Off : public State {
    ...     // Аналогично
}
```

3.3.2. Инструментальное средство *UniMod*

Переходя к обсуждению инструментального средства для поддержки объектно-ориентированного программирования с явным выделением состояний, отметим, что если для генерации программ по автоматам кроме средств, рассмотренных в разд. 2.3.3, известны также и многие другие, то рассматриваемое здесь решение задачи об автоматизации построения объектно-ориентированных программных систем со сложным поведением *в целом* единственное в своем роде. Это объясняется тем, что по сравнению с другими средствами визуального конструирования объектно-ориентированных программ (*IBM Rational Rose*, *Borland Together*, *Telelogic Rhapsody*) их проектирование в инструментальном средстве *UniMod* осуществляется так же, как выполняется автоматизация объектов управления в промышленности. Кроме того, по сравнению с аналогами оно является открытым и бесплатным [70].

Авторы проекта *UniMod* [69, 71] предложили, во-первых, метод и язык моделирования для описания систем со сложным поведением (язык является подмножеством *UML*), а во-вторых, реализовали инструментальное средство, которое позволяет описывать системы на этом языке, проверять их корректность, интерпретировать или компилировать их, и даже отлаживать.

Процесс разработки системы со сложным поведением с помощью этого инструментального средства состоит в следующем.

1. На основе анализа предметной области производится объектная декомпозиция системы, определяются сущности и отношения между ними.

2. В отличие от традиционных для объектно-ориентированного программирования подходов, из числа сущностей выделяются *источники событий*, *объекты управления* и *автоматы*. Источники событий активны – они по собственной инициативе воздействуют на автомат. Объекты управления пассивны и выполняют действия по команде от автомата. Они также формируют значения входных переменных для автомата. Автомат активируется источниками событий и на основании значений входных переменных и текущего состояния воздействует на объекты управления и переходит в новое состояние.

3. С использованием нотации *диаграммы классов*, строится схема связей автомата, которая задает его интерфейс. На этой схеме слева отображаются источники событий, в центре – автоматы, а справа – объекты управления. Источники событий с помощью *UML*-ассоциаций связываются с автоматами, которым они

поставляют события. Автоматы связываются с объектами, которыми они управляют.

4. Каждый объект управления содержит два типа компонентов, реализующих входные (`xj`) и выходные переменные (`zk`).
5. Для каждого автомата с помощью нотации диаграммы состояний строится граф переходов, в котором дуги могут быть помечены событием (`ei`), булевой формулой из входных переменных и выходным воздействием на переходе. В вершинах могут быть указаны выходные воздействия в состояниях и имена вложенных автоматов. Каждый автомат имеет одно начальное и произвольное число завершающих состояний.
6. Состояния на графе переходов могут быть простыми и составными (суперсостояния). Если в состояние вложено другое состояние, то оно является составным. В противном случае состояние простое. Основной особенностью составных состояний является то, что дуга, исходящая из такого состояния, заменяет однотипные дуги из каждого вложенного состояния.
7. Компоненты объекта управления, соответствующие входным и выходным переменным, реализуются *вручную* на целевом языке программирования.

Как может убедиться читатель, подход к разработке, предлагаемый авторами инструментального средства *UniMod*, отличается от того, который был описан в разд. 3.1. В частности, в основе этого подхода лежит концепция «автоматы и объекты управления как классы». Скорее всего, такое решение было принято по той причине, что во время создания проекта *UniMod*, концепции «автоматизированные объекты управления как классы» еще не существовало.

На рис. 3.19 приведена схема связей автомата, а на рис. 3.20 – его граф переходов, построенные с помощью инструмента *UniMod*.

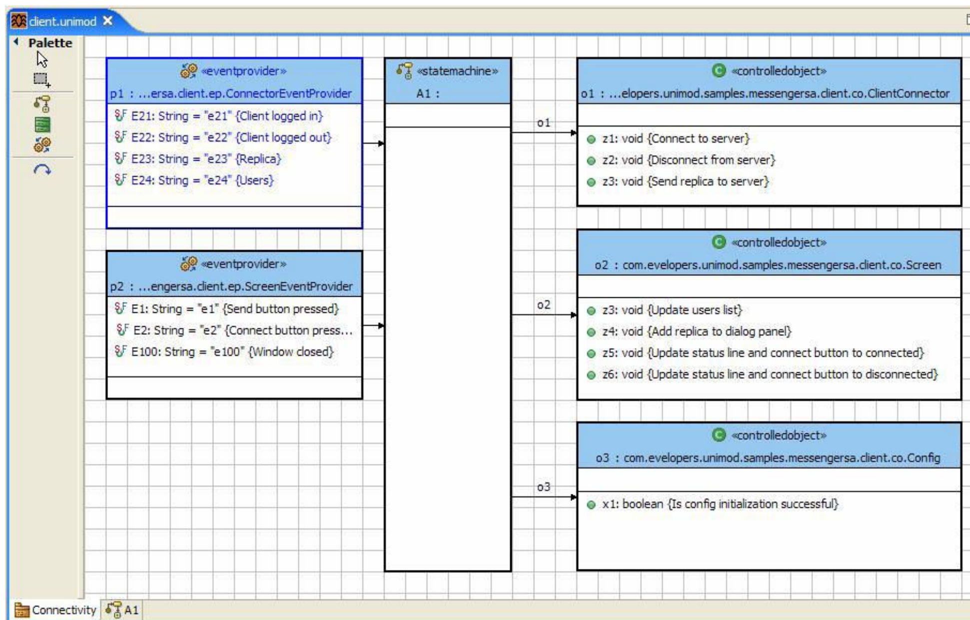

Рис. 3.19. Пример схемы связей автомата

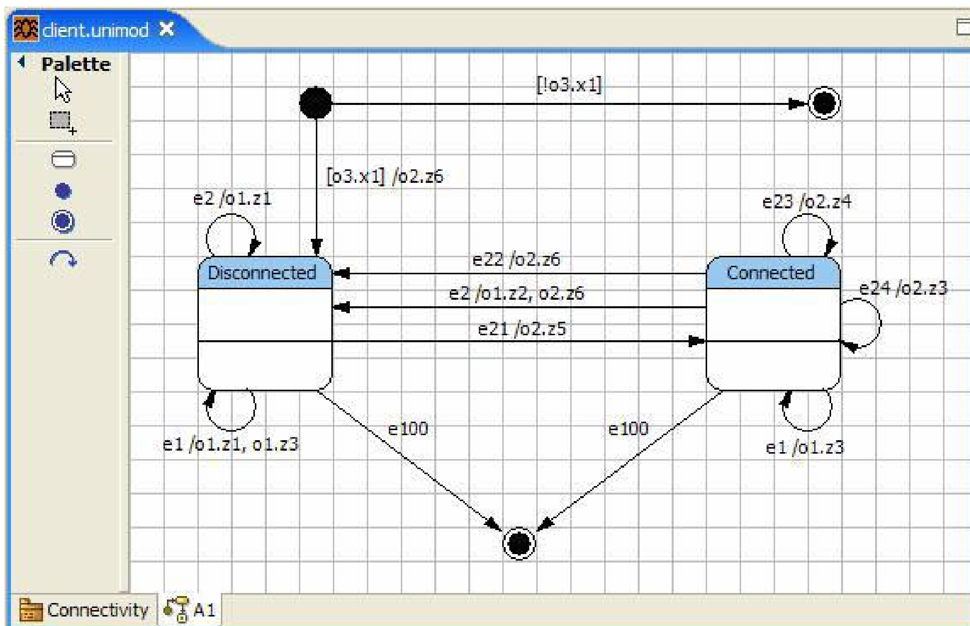

Рис. 3.20. Пример графа переходов автомата

Поскольку модель системы в рассматриваемом средстве предназначена для компиляции в исполняемый код или интерпретации, ей необходима точная операционная семантика. Перечислим наиболее важные ее аспекты.

- При запуске модели инициализируются все источники событий. После этого они начинают воздействовать на связанные с ними автоматы.
- Каждый автомат начинает свою работу из начального состояния, а заканчивает – в одном из завершающих.
- При получении события автомат выбирает все исходящие из текущего состояния переходы, помеченные символом этого события.
- Автомат перебирает выбранные переходы и вычисляет булевы формулы, записанные на них, до тех пор, пока не найдет формулу со значением «истина».
- Если переход с такой формулой найден, автомат выполняет выходное воздействие, записанное на дуге, и переходит в новое состояние. В нем автомат выполняет выходное воздействие, указанное для этого состояния, а также запускает вложенные автоматы.
- Если переход не найден, то автомат продолжает поиск перехода у состояния, в которое вложено текущее состояние.
- При переходе в завершающее состояние автомат останавливает все источники событий. После этого работа системы прекращается.

Инструмент *UniMod* создан на основе среды разработки *Eclipse* [72]. Он позволяет создавать и редактировать схемы связей и графы переходов управляющих автоматов, которые создаются на основе диаграмм классов и состояний языка *UML*. Инструмент выполняет *валидацию* автоматов: проверяет достижимость состояний, полноту и непротиворечивость условий переходов. Кроме того, он поддерживает отладку программ в терминах автоматов, когда при пошаговом выполнении на диаграмме состояний подсвечивается текущее состояние, переход, выходная переменная и т. п. И, наконец, инструмент совместно с известными или вновь разработанными верификаторами позволяет производить проверку корректности построенных с его помощью программ на основе метода *Model Checking* [73].

Инструментальное средство *UniMod* может осуществлять как компиляцию диаграмм в код на различных языках программирования (*Java*, *C++*), так и непосредственную интерпретацию с промежуточным преобразованием диаграмм в описание на языке *XML*. В настоящее время ведутся работы по совершенствованию этого инструментального средства [74].

Рассматриваемое инструментальное средство весьма эффективно при разработке небольших программных систем «с нуля», что подтверждается экспериментально [75].

Читатель, знакомый с материалом разд. 3.1 и 3.2, мог заметить ряд недостатков модели сложного поведения и подхода к разработке, которые используются в инструментальном средстве *UniMod*.

- Концепция «автоматы и объекты управления как классы» нарушает стройность объектно-ориентированной структуры системы и приводит к появлению в ней нового типа сущностей – автоматов. Это ограничивает модульность (поскольку автоматы в действительности не являются самостоятельными сущностями) и усложняет выделение уровней абстракции.

- Как следствие предыдущего пункта, требуется не только объектная, но и автоматная декомпозиция, что еще больше усложняет структуру системы.
- В графической нотации описания поведения смешиваются элементы из различных языков: используются как составные состояния, так и вложенные автоматы.
- Модули системы не являются в достаточной степени самостоятельными и не предназначены для повторного использования. Например, имена компонентов объектов управления совпадают с краткими идентификаторами входных и выходных переменных автомата. Это довольно странно, если допустить, что класс объекта управления мог разрабатываться отдельно от управляющего автомата и может быть повторно использован в другой системе. Кроме того, для автоматов множество их клиентов ограничивается заранее заданными поставщиками событий, что препятствует использованию разработанных автоматов в другом контексте.
- В инструменте *UniMod* используется довольно странная автоматная модель. С одной стороны, все автоматы являются пассивными: совершают переходы только при возникновении события. Кроме того, работа системы начинается с запуска источников событий. Однако все автоматы снабжены завершающими состояниями, и окончание работы системы происходит по инициативе автомата. Такая модель находится где-то между пониманием автомата как главной программы (унаследованного из процедурного программирования с явным выделением состояний) и как сущности, предоставляющей набор сервисов. В доказательство наличия пережитков процедурного подхода, в большинстве проектов, созданных с помощью инструмента *UniMod* [75], существует единственный главный автомат, в который вложены все остальные.

Каким же должно быть инструментальное средство автоматного программирования, поддерживающее подход «автоматизированные объекты управления как классы»? Для того чтобы средство было востребованным среди широкого круга программистов, его использование должно требовать минимальных изменений в процессе разработки по сравнению с программированием «без автоматов».

В соответствии с этим требованием, новый инструмент должен быть встроен в некоторую популярную среду разработки, а лучше – в несколько сред для различных объектно-ориентированных языков программирования. При создании нового класса инструмент должен обеспечивать возможность указания, что этот класс является автоматизированным. В этом случае при редактировании класса помимо обычного текста (который понадобится в том случае, если запросы и команды объекта управления полностью или частично реализуются в автоматизированном классе) должны появляться схема связей и диаграмма переходов. При этом отметим, что диаграмму переходов, как правило, удобнее редактировать в графической форме. Необходимо очень аккуратно подойти к разработке пользовательского интерфейса инструмента, для того чтобы трудности создания и модификации диаграмм не свели на нет все преимущества автоматного подхода. Разумеется, в новом инструменте следует сохранить такие достоинства инструментального средства *UniMod*, как проверка правильности и отладка в терминах автоматов.

Разработка инструментального средства, удовлетворяющего перечисленным выше требованиям, в настоящее время проводится в рамках международного проекта

EiffelState (http://code.google.com/p/eiffel-state/), выполняемого на кафедре «Программная инженерия» Высшей политехнической школы в Цюрихе и кафедре «Технологии программирования» Санкт-Петербургского государственного университета информационных технологий, механики и оптики.

Изложенный в настоящей главе подход обладает при указанных выше достоинствах и одним недостатком: обеспечить верификацию программ, построенных с помощью него, существенно сложнее, чем для программ, в которых логика централизована, как в случае использования инструментального средства *UniMod*. Этот подход с точки зрения верификации занимает промежуточное положение между программами, построенными традиционными путем, и автоматными *UniMod*-программами.

Глава 4. Автоматное программирование. Новые задачи

В последней главе книги собраны краткие очерки о нетрадиционных областях применения автоматного подхода и новых задачах, возникающих в связи с развитием автоматного программирования. Глава является обзорной и не претендует на полноту изложения. Ее цель – ознакомить читателя с последними достижениями в области автоматного программирования и показать, что эта область является современной и активно развивающейся, и в ней есть еще много нерешенных задач.

4.1. Автоматы и алгоритмы дискретной математики

Все предыдущее изложение было посвящено интерактивным и реактивным системам (разд. 1.1). Однако во введении было упомянуто, что автоматы, по мнению авторов, целесообразно использовать во всех классах программных систем. В этом разделе поговорим о применении автоматов в трансформирующих системах, а более конкретно – в задачах дискретной математики.

Здесь автоматы могут использоваться как непосредственно при построении некоторых алгоритмов (например, поиск подстрок [76]), так и для визуализации алгоритмов, реализованных традиционным способом. Методы преобразования традиционных алгоритмов в автоматные описаны в работах [77, 78]. Отметим, что в трансформирующих системах чаще всего применяются активные автоматы без событий.

При построении алгоритмов дискретной математики наибольшее внимание, как правило, уделяется эффективности по времени и по памяти, и автоматы в этом вряд ли могут помочь. Однако автоматные алгоритмы часто являются более структурированными, а их представление с использованием диаграмм переходов – более наглядным. Эти свойства приобретают особенно большое значение при обучении дискретной математике.

В качестве примера использования автоматного подхода при построении алгоритма рассмотрим одну из задач дискретной математики: обход двоичных деревьев. Автоматное решение этой задачи, которое рассматривается ниже, заимствовано из работы [79]. Исходным данным в задаче является двоичное дерево, вершины которого пронумерованы (пример такого дерева приведен на рис. 4.1).

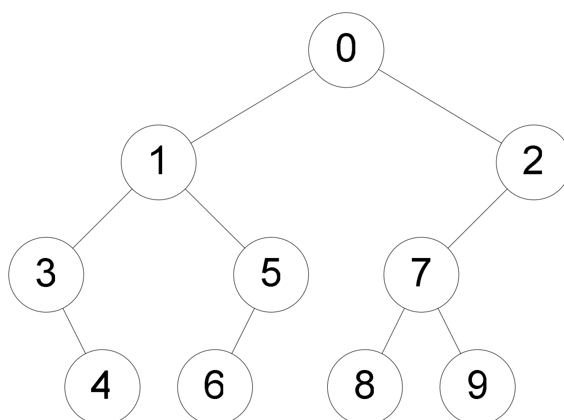

Рис. 4.1. Пример двоичного дерева

Требуется осуществить обход дерева – сформировать последовательность номеров его вершин. При обходе дерева в глубину используются три стандартных порядка обхода: прямой (*preorder*), обратный (*postorder*) и центрированный (*inorder*) [76, 80]. В качестве примера приведем последовательности, порождаемые обходами дерева, которое представлено на рис. 4.1:

- прямой обход: 0, 1, 3, 4, 5, 6, 2, 7, 8, 9;
- центрированный обход: 3, 4, 1, 6, 5, 0, 8, 7, 9, 2;
- обратный обход: 4, 3, 6, 5, 1, 8, 9, 7, 2, 0.

Известно несколько классических решений этой задачи: рекурсивное [76, 80], итеративное с применением стека и итеративное без использования стека [81]. Рекурсивное решение является наиболее простым и понятным: в нем процедура обхода дерева вызывает саму себя для левого и правого поддерева. Однако, как и все рекурсивные алгоритмы, это решение обладает сравнительно низким быстродействием. Итеративный алгоритм без использования стека предполагает хранение в каждой вершине дерева информации о родительском узле. Такая структура хранения избыточна. Наиболее реалистичным является алгоритм с применением стека, однако он же является и наиболее сложным. Рассмотрим автоматную модификацию этого алгоритма, которая, как сможет убедиться читатель, является не только наглядной, но и эффективной.

Будем считать, что в каждой вершине дерева содержится ее номер и указатели на левую и правую дочерние вершины. На языке программирования *C++* эту структуру можно представить следующим образом:

```
struct Node {
    int id;         // Номер узла
    Node* left;     // Левый дочерний узел
    Node* right;    // Правый дочерний узел
};
```

Кроме того, будем считать, что в системе существует функция `put (int x)`, которая позволяет вывести номер вершины в выходную последовательность, и класс, реализующий стек, со следующим интерфейсом:

```
template <typename T>
class Stack {
    void push(const T& x);   // Поместить значение x в стек
    void pop();              // Удалить из стека верхний элемент
    const T& top() const;    // Верхний элемент стека
};
```

Идея предлагаемого алгоритма состоит в том, что при обходе двоичного дерева могут быть выделены три направления движения: влево, вправо и вверх. В зависимости от текущего направления движения, свойств текущей вершины и верхнего символа в стеке можно определить, куда двигаться дальше. Поэтому удобно сопоставить каждому направлению движения управляющее состояние автомата.

Схема связей автомата, осуществляющего обход двоичных деревьев, приведена на рис. 4.2, а его диаграмма переходов – на рис. 4.3.

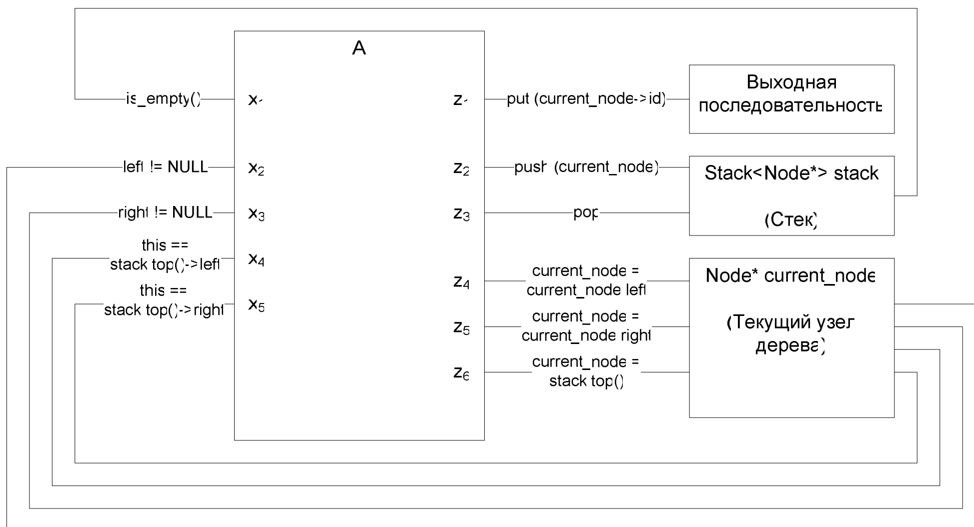

Рис. 4.2. Схема связей автомата, реализующего обход дерева

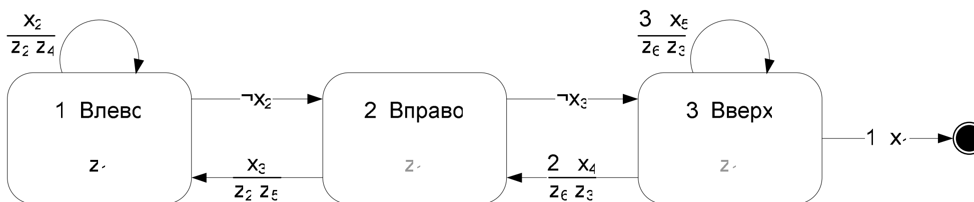

Рис. 4.3. Диаграмма переходов автомата, реализующего обход дерева

На диаграмме переходов действие z_1 следует поместить в одно из трех состояний в зависимости от требуемого порядка обхода:

- для прямого порядка обхода – в состояние «Влево»;
- для центрированного – в состояние «Вправо»;
- для обратного – в состояние «Вверх».

Отметим, что в графе переходов легко выделить общую часть, которая не зависит от порядка обхода. Таким образом, предложенная реализация является довольно универсальной.

Рассмотрим вкратце применение автоматов для построения визуализаторов алгоритмов дискретной математики. *Визуализатор* – это программа, в процессе работы которой на экране компьютера динамически демонстрируется применение алгоритма к выбранному набору данных. Визуализаторы позволяют изучать работу алгоритмов, как в автоматическом, так и в пошаговом режиме, аналогичном режиму трассировки программ. Визуализация шагов алгоритма сопровождается текстовыми комментариями.

Автоматный подход оказался очень удобным при создании визуализаторов алгоритмов дискретной математики для описания логики их работы. Введение в «поточные» алгоритмы состояний значительно упрощает проектирование визуализаторов, позволяя выделить в алгоритме значимые для наблюдателя точки, в которых и следует осуществлять визуализацию.

На основе автоматного подхода были разработаны метод ручного проектирования визуализаторов [82], а также технология автоматизированного построения визуализаторов *Vizi* и инструментальное средство с тем же названием, поддерживающее эту технологию [83]. Примеры визуализаторов, построенных с применением автоматного подхода, и проектная документация к ним опубликованы в сети Интернет [84].

4.2. Проверка правильности автоматных программ

Корректность – одно из ключевых свойств ПО, а методы проверки и обеспечения корректности – важная и актуальная область исследований в инженерии программного обеспечения [85, 86]. Выделяются два основных подхода к проверке корректности: динамический и статический.

При динамическом подходе программа проверяется в процессе исполнения. Наиболее употребительным примером применения этого подхода является тестирование программ. В настоящее время в области тестирования проводится множество исследований, цель которых – автоматизация процесса тестирования, повышение вероятности нахождения ошибок и т. д. Таким образом, тестирование позволяет находить все больше и больше ошибок, однако, как отметил Э. Дейкстра, оно никогда не сможет гарантировать корректность программы.

Напротив, статические методы проверки гарантируют правильность программы в том или ином смысле. Простейший вид статической проверки (проверка синтаксиса программного текста) существует в любом компиляторе. На следующем уровне находится, так называемая, *валидация*: контроль за выполнением более сложных свойств программной системы, таких как, например, типовая корректность. Валидация выполняется большинством современных сред разработки, если язык программирования это предусматривает (в частности, имеет статическую систему

типов). Наиболее мощным видом статической проверки корректности является формальная *верификация* – доказательство соответствия текста программной системы набору свойств, описанному на некотором формальном языке. Этот набор свойств обычно называют формальной спецификацией системы.

В рамках формальной верификации выделяются два основных направления: *доказательная верификация* [87, 88] и *проверка модели* программы (*Model Checking*) [89].

Доказательная верификация, в общем случае, требует большого объема ручной работы, что делает ее малоприменимой для программных систем большого размера.

Для верификации модели программы требуется решить следующие задачи: построить по программе модель с конечным числом состояний и формально описать требования к программе терминах одного из видов темпоральной логики [90]. В результате верификации модели либо подтверждается, что модель удовлетворяет формализованным требованиям, либо строится контрпример. В последнем случае требуется определить причину некорректности: ошибка в исходной программе, в спецификации или при построении модели. При этом необходимо решить задачу переноса контрпримера из модели в исходную программу. Для программ общего вида решение этих задач плохо поддается автоматизации.

Переходя к автоматным программам, остановимся сначала на динамической проверке их корректности. Большинство существующих библиотек и инструментальных средств, поддерживающих автоматное программирования, позволяют вести протокол выполнения программы в терминах автоматов. В этом протоколе записывается последовательность состояний, в которых пребывал автомат, обработанные события, значения входных и выходных переменных, а при помощи специального форматирования (например, отступов) наглядно отражаются отношения вложенности и вызываемости между автоматами [91]. Кроме того, как было упомянуто выше, инструментальное средство *UniMod* позволяет осуществлять пошаговую отладку в терминах автоматов. Применяя описанные высокоуровневые средства отладки гораздо проще находить ошибки в логике сложного поведения, чем с использованием традиционных методов.

Что касается статической проверки корректности, для автоматных программ существуют специфические методы, которые также можно отнести к нескольким уровням. На простейшем, синтаксическом уровне выделяются такие правила как, например, «Любой переход должен вести из одного состояния в другое». На уровне валидации существует целый ряд поведенческих свойств, которые несложно проверить по графу переходов автомата. Это позволяет устранить программные ошибки, которые было бы трудно обнаружить без использования автоматного подхода.

- **Достижимость.** В любое состояние автомата должен вести путь из начального состояния, составленный из переходов. В противном случае это состояние не имеет смысла, так как автомат никогда не сможет в нем оказаться. Само по себе недостижимое состояние безвредно, однако оно, скорее всего, свидетельствует об ошибке при построении автомата.
- **Непротиворечивость.** Множество переходов из каждого состояния автомата должно быть непротиворечивым: не должно существовать такого входного

воздействия, при котором условия нескольких переходов из этого множества будут одновременно истинными. При использовании распространенных методов программной реализации автоматов наличие противоречия приведет к тому, что будет выбран один из переходов, однако какой именно, невозможно определить по диаграмме. Такой недетерминизм не несет практической пользы и, как правило, также свидетельствует об ошибке.

- **Полнота.** Множество переходов из каждого состояния автомата должно быть полным: для любого входного воздействия должен существовать хотя бы один переход из этого множества, условие которого будет истинным. При отсутствии этого свойства невозможно гарантировать правильную работу автоматной программы, так как во время выполнения может сложиться ситуация, когда автомат не сможет совершить переход. Таким образом, функционирование всей автоматной модели будет нарушено.

Отметим, что перечисленные выше свойства проверяются формально, без учета семантики входных и выходных переменных. Обычно этого оказывается достаточно. Однако, если объект управления наделен формальной спецификацией, ее можно принять во внимание при проверке свойств автомата. Например, в спецификации может быть указано, что некоторые входные переменные не могут быть одновременно истинными. Такая информация способна повлиять на результаты проверки всех трех перечисленных свойств.

Перейдем теперь к формальной *верификации* автоматных программ. В этой области в настоящее время активно ведутся исследования в России [92, 93]. Специфическая структура автоматных программ, в которых запросы и команды объекта управления представляют собой очень простые подпрограммы, а логика сложного поведения описана конечными автоматами, как это делается при использовании инструментального средства *UniMod,* позволяет совместно использовать оба указанных выше подхода к формальной верификации.

Для простых подпрограмм (компонентов объекта управления) целесообразно использовать доказательный подход, а управляющие автоматы могут быть формально преобразованы в модель с конечным числом состояний, которая подлежит верификации методом *Model Checking*. Это преобразование может быть автоматизировано. Также может быть автоматизировано и обратное преобразование в случае, если верификатор построил контрпример. Для автоматных *UniMod*-программ не существует семантического разрыва между требованиями к программе и модели — он преодолевается в процессе проектирования автоматов. В настоящее время существуют экспериментальные реализации инструментов, которые позволяют весьма эффективно проводить верификацию автоматных программ указанного класса [94–96].

Результаты работ по верификации автоматных программ на основе метода *Model Checking*, полученные под руководством одного из авторов книги в ходе выполнения государственного контракта, приведены по адресу http://is.ifmo.ru/verification/.

Упрощение формальной верификации автоматных программ по сравнению с программами, написанными без явного выделения управляющих состояний, позволяет авторам надеяться, что со временем в технических заданиях на проектирование ответственных программных систем будет указано на необходимость использования автоматного подхода.

4.3. Автоматы и параллельные вычисления

В настоящее время, когда быстродействие процессоров традиционной архитектуры уже почти достигло своего максимума, все более важную роль играют параллельные и распределенные вычисления. Для того чтобы иметь возможность оптимальным образом увеличить быстродействие программной системы за счет наличия нескольких вычислителей, необходимо тщательно спроектировать архитектуру системы с учетом параллелизма. Разработка простых и эффективных методов «распараллеливания» программ является в настоящее время одной из самых актуальных проблем программной инженерии.

Существует целый ряд моделей параллельных и распределенных вычислений, а также стандартов, языков и инструментальных средств, их поддерживающих:

- классическая модель «ветвление/слияние» потоков для параллельных вычислений с общей памятью, реализованная, например, в программном интерфейсе *OpenMP* [97];
- модель, основанная на процессах и обмене сообщениями между ними (например, стандарт *MPI* [98]);
- активные объекты, присутствующие в большинстве распространенных объектно-ориентированных языков программирования;
- объектно-ориентированная модель *SCOOP* (*Simple Concurrent Object-Oriented Programming*) [29].

Параллельная автоматная декомпозиция, которая была рассмотрена в разд. 2.1, порождает еще одну модель параллельных вычислений: параллельно работающие автоматы, взаимодействие которых может осуществляться путем обмена номерами состояний, событиями или сообщениями [32, 99]. В чем же преимущество этой модели перед перечисленными выше?

В автоматном программировании управляющие автоматы и объекты управления всегда выделяются в явном виде. Если объекты управления двух автоматов не пересекаются (или область их пересечения невелика), то такие автоматы с большой вероятностью смогут эффективно работать параллельно. В результате при применении автоматного подхода параллелизм часто возникает естественно, без дополнительных преобразований.

В качестве примера рассмотрим задачу декодирования файлов формата *GIF* (сравнение автоматного и традиционного подходов к решению этой задачи подробно обсуждается в работе [100]). Задача состоит в том, чтобы преобразовать *GIF*-файл в матрицу цветов пикселей, составляющих изображение.

При декодировании таких файлов сначала декодируется заголовок изображения, а затем оставшаяся часть файла разбирается на блоки. Каждый блок, в свою очередь, состоит из заголовка и тела, представляющего собой *LZW*-код искомой последовательности цветов пикселей.

В работе [100] описано решение этой задачи с использованием программирования с явным выделением состояний. При этом автоматная декомпозиция приводит к

выделению трех автоматов: первый отвечает за декодирование заголовка, второй – разбивает основную часть файла на блоки, а третий – расшифровывает *LZW*-коды. В результате такой декомпозиции оказывается, что второй и третий автоматы могут работать параллельно: каждый раз, когда второй автомат выделяет тело очередного блока, ему необязательно дожидаться, пока третий автомат преобразует это тело в последовательность цветов пикселей. Ему достаточно отправить выделенный *LZW*-код третьему автомату в качестве сообщения или передать его через очередь с параллельным доступом.

4.4. Автоматы и генетическое программирование

При использовании автоматного подхода для реализации сложного поведения построение управляющего автомата во многих случаях является шагом, наиболее сложным для программиста и порождающим наибольшее число ошибок. Кроме того, существуют задачи, для которых построить автомат вручную практически невозможно. В других случаях построенный автомат бывает неоптимальным. Возможное решение всех этих проблем – перепоручить построение управляющего автомата компьютеру.

Одним из наиболее эффективных методов автоматизированного конструирования программ является *генетическое программирование* [101]. Основная идея генетического программирования состоит в построении программ путем применения *генетических алгоритмов* [102] к некоторой *модели вычисления*. При этом разработчику программы остается лишь задать *оценочную функцию*, определяющую для каждого результата вычисления в выбранной модели численное значение, называемое его *приспособленностью*.

Генетические алгоритмы – это метод оптимизации, основанный на концепциях естественной эволюции. Он оперирует *поколениями*, состоящими из *особей*. Первое поколение заполняется особями, сгенерированными случайным образом. Каждое последующее поколение формируется из особей предыдущего поколения в зависимости от их приспособленности путем перенесения их без изменений, либо применения к ним с некоторыми вероятностями генетических операторов: *мутации* и *скрещивания*. Результатом оптимизации считается особь с максимальной оценкой из последнего поколения.

Для того чтобы применение генетического программирования для оптимизации автоматизированных объектов управления стало возможным, необходимо разработать представление автомата в виде особи, определить операции мутации и скрещивания автоматов и подобрать параметры генетического алгоритма, подходящие для автоматов. После этого для каждой задачи остается только реализовать объект управления и определить функцию приспособленности. Значение приспособленности автоматизированного объекта управления целесообразно задавать как функцию вычислительного состояния объекта управления после заданного числа тактов работы автомата.

Обычно в качестве моделей вычислений в генетическом программировании используют деревья, графы, команды процессора – «низкоуровневые» модели, имеющие ограниченный набор элементарных операций (таких как, например, запись и чтение ячеек памяти, арифметические операции, вызовы подпрограмм и т. д.). Достоинство низкоуровневых моделей состоит в их универсальности: с их помощью

можно построить любую программу целиком, единообразно, вне зависимости от специфики решаемой задачи. Однако такие модели обладают и серьезными недостатками. Во-первых, построенная программа из-за отсутствия высокоуровневой структуры редко бывает понятна человеку, что исключает возможность ее дальнейшей модификации вручную, обобщения полученного решения и т. п. Во-вторых, из-за того, что пространство допустимых программ в этом случае очень велико, «генетическая оптимизация» требует длительного времени и может использоваться только для небольших задач.

С другой стороны, если использовать в качестве модели вычислений автоматизированный объект, причем реализацию объекта управления производить вручную, пространство поиска значительно сокращается, и генетический подход становится реалистичным. При этом используется разумное разделение труда: компьютер выполняет ту часть работы, которая лучше всего получается у него, а человек – то, что под силу только ему.

Первые работы в области генерации конечных автоматов с помощью генетических алгоритмов были выполнены около 40 лет назад [103]. Однако в этой работе автоматы рассматривались как одна из моделей человеческого интеллекта. Эта модель оказалась непродуктивной (по крайней мере, при существовавшем в то время уровне развития вычислительной техники), и исследования в этом направлении практически прекратились. Они возобновились только через 20 лет [104–106]. При этом круг рассматриваемых задач расширился. В частности, стали изучаться задачи управления, однако в большинстве этих задач управляющие автоматы имели только одну входную переменную (разд. 4.4.1). С развитием автоматного программирования стала актуальной задача генерации управляющих автоматов с большим числом входных переменных (разд. 4.4.2).

4.4.1. Задача об «Умном муравье»

В качестве примера, в котором автоматическое построение логики сущности со сложным поведением имеет решающее значение, рассмотрим одну из классических задач из области совместного использования генетических алгоритмов и конечных автоматов – задачу об «Умном муравье» [106].

Муравей «живет» на поверхности тора размером 32 на 32 клетки (рис. 4.4). В некоторых клетках (обозначены на рис. 4.4 черным цветом) находится еда. Она расположена вдоль некоторой ломаной, но не во всех ее клетках. Клетки ломаной, в которых нет еды, обозначены серым цветом. Белые клетки не содержат еду и не принадлежат ломаной. Всего на поле 89 клеток с едой.

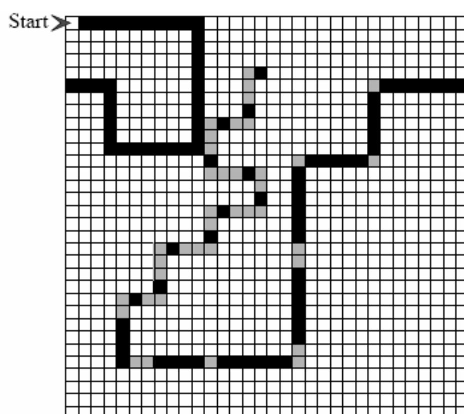

Рис. 4.4. Поле в задаче об «Умном муравье»

В клетке, помеченной меткой *Start*, в начале игры находится муравей. Он занимает одну клетку и смотрит в одном из четырех направлений (север, юг, запад, восток). В начале игры муравей смотрит на восток.

Муравей умеет определять, находится ли непосредственно перед ним еда. За один игровой ход муравей может совершить одно из четырех действий:

- сделать шаг вперед, съедая еду, если она там находится;
- повернуть налево;
- повернуть направо;
- ничего не делать.

Съеденная муравьем еда не восполняется, муравей жив на протяжении всей игры, еда не является необходимым ресурсом для его жизни. Ломаная не случайна, а строго фиксирована (рис. 4.4). Муравей может ходить по любым клеткам поля.

Игра длится 200 ходов, на каждом из которых муравей совершает одно из четырех действий. По истечении 200 ходов подсчитывается количество еды, съеденной муравьем. Это значение и есть результат игры.

Цель игры – создать муравья, который за 200 ходов съест как можно больше еды (желательно, все 89 единиц).

Один из способов описания поведения муравья – автомат Мили, у которого имеется одна входная переменной (находится ли еда перед муравьем), а множество выходных воздействий состоит из четырех упомянутых выше элементов. Схема связей этого автомата приведена на рис. 4.5.

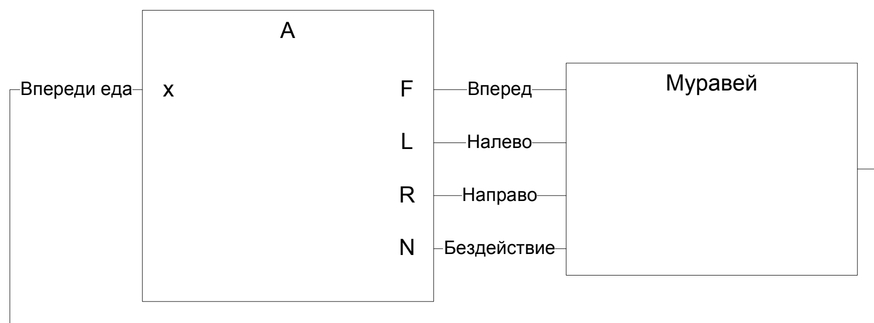

Рис. 4.5. Схема связей автомата «Умного муравья»

Автомат, решающий описанную задачу, крайне трудно построить вручную. Например, эвристически построенный автомат Мили с пятью состояниями из работы [104], граф переходов которого изображен на рис. 4.6, задачу не решает – он описывает поведение муравья, который съедает всего 81 единицу еды за 200 ходов, а всю еду – только за 314 ходов.

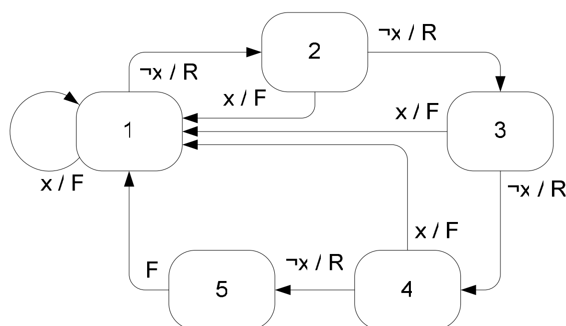

Рис. 4.6. Простой автомат «Умного муравья»

Эта задача была решена с применением генетических алгоритмов. При этом был построен автомат с восьмью состояниями [106]. Ниже описывается алгоритм, предложенный в работе [107], который позволил построить для решения рассматриваемой задачи автомат с семью состояниями.

Пусть представление автомата в виде особи состоит из номера начального состояния и описания каждого состояния. Описание (в терминах генетического программирования – *хромосома*) состояния содержит описания двух переходов, соответствующих двум значениям входной переменной. Описание перехода состоит из номера состояния, в которое он ведет, и действия, выполняемого на этом переходе. Начальное поколение содержит фиксированное число случайно сгенерированных автоматов. Все автоматы в поколении имеют одинаковое наперед заданное число состояний.

Теперь опишем генетические операторы мутации и скрещивания в терминах предложенного представления автоматов.

При мутации случайно выбирается один из четырех равновероятных вариантов:

- **изменение начального состояния** – в этом случае новое начальное состояние выбирается случайно и равновероятно;
- **изменение действия на переходе** – случайно и равновероятно выбирается переход, и действие на нем изменяется на случайное;
- **изменение состояния, в которое ведет переход** – случайно и равновероятно выбирается переход, после этого целевое состояние перехода изменяется на случайно выбранное;
- **изменение условия на переходе** – случайно и равновероятно выбирается состояние, после этого переходы из этого состояния, соответствующие различным значениям входной переменной, меняются местами.

Оператор скрещивания принимает на вход две родительские особи, а результатом также являются две особи. При этом первый ребенок может наследовать начальное состояние от первого родителя, а второй – от второго, или наоборот. Что касается переходов из заданного состояния, каждый ребенок может наследовать как переход по значению входной переменной «истина», так и переход по значению «ложь» равновероятно от обоих родителей, но оба ребенка не могут наследовать один и тот же переход. Схема скрещивания изображена на рис. 4.7.

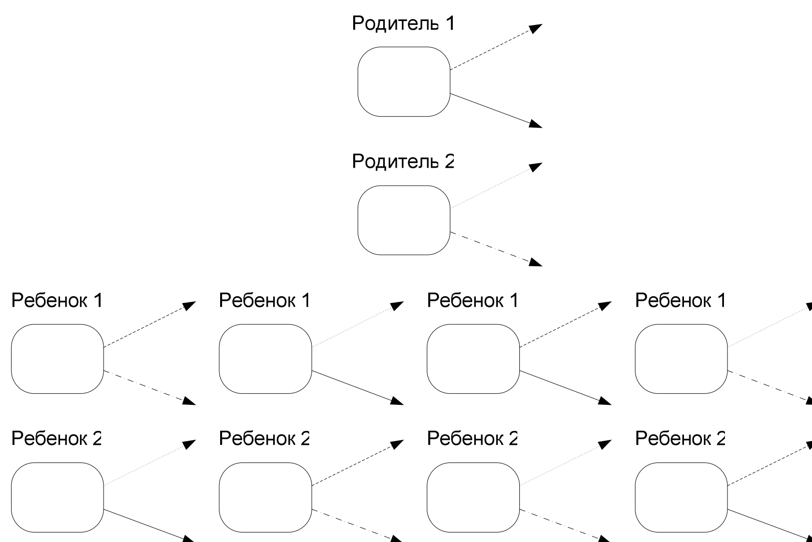

Рис. 4.7. Простая схема скрещивания состояний автомата

Перейдем к функции приспособленности. Определим ее следующим образом:

$$A + \frac{200 - T}{200},$$

где A – суммарное количество еды, съеденной муравьем за игру, T – номер хода, на котором муравей съедает последнюю единицу еды.

С использованием приведенных выше генетических операторов и функции приспособленности удалось построить автомат с семью состояниями, который

позволяет муравью съесть все 89 единиц еды за 190 ходов. Граф переходов этого автомата приведен на рис. 4.8. Для того чтобы найти этот автомат, потребовалось перебрать 130 000 поколений и 160 миллионов особей, что совсем немного по сравнению с общим числом автоматов с семью состояниями, одной входной и четырьмя выходными переменными.

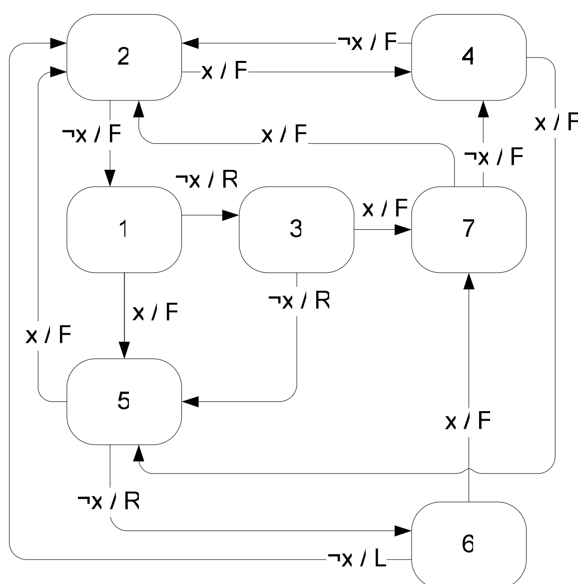

Рис. 4.8. Автомат «Умного муравья», построенный методом генетического программирования

Еще один автомат с семью состояниями был построен после перебора 250 миллионов особей, что также представляет незначительный процент от общего числа автоматов рассматриваемого вида.

В работе [108] приведено решение задачи об «Умном муравье» с помощью автомата Мура и системы из двух автоматов Мили.

Еще одна интересная задача, решаемая простым автоматом, который генерируется с помощью генетического алгоритма, описана в работе [109].

4.4.2. Методы генерации автоматов с большим числом входных переменных

В задаче об «Умном муравье» представление автомата в виде особи и реализация генетических операторов не составляли проблемы. Функции переходов и выходов для *каждого состояния* автомата были представлены в виде *полной таблицы* переходов и выходов: в этой таблице каждому возможному входному воздействию сопоставляется целевое состояние и выходное воздействие (рис. 4.9).

x_1	x_2	s	z_1	z_2	z_3
0	0	0	0	0	1
0	1	1	1	0	1
1	0	0	0	1	0
1	1	2	1	1	1

Рис. 4.9. Представление функций переходов и выходов для одного состояния автомата в виде полной таблицы

При использовании полных таблиц, размер представления автомата растет экспоненциально с увеличением числа входных переменных. Это не позволяет использовать такие таблицы для автоматов с большим числом входных переменных. К счастью, реальные автоматы обладают свойством, отмеченным в разд. 2.3: обычно только небольшая часть входных переменных является *значимыми* в каждом состоянии. Исходя из этого свойства, можно предложить различные способы упрощения представления автоматов.

В настоящее время известно два способа компактного представления автоматов с большим числом входных переменных: метод *сокращенных таблиц* [110] и метод *деревьев решений* [111].

Идея метода сокращенных таблиц состоит в том, что число значимых входных переменных в каждом состоянии ограничивается константой. При этом представление функции переходов и выходов для состояния автомата содержит описание множества значимых входных переменных и собственно сокращенную таблицу, в которой каждой комбинации значений выбранных переменных сопоставляется целевое состояние и выходное воздействие (рис. 4.10).

x_1	x_2	x_3	x_4	x_5	x_6
0	1	0	1	0	0

x_2	x_4	s	z_1	z_2	z_3
0	0	0	0	0	1
0	1	1	1	0	1
1	0	0	0	1	0
1	1	2	1	1	1

Рис. 4.10. Представление функций переходов и выходов для одного состояния с помощью сокращенной таблицы

Опыт показывает, что даже в системах со сложным поведением, которые содержат несколько десятков входных переменных, число значимых переменных в каждом

состоянии обычно не превышает нескольких единиц. Поэтому для сложных задач размер сокращенной таблицы остается небольшим.

Операции скрещивания и мутации для сокращенных таблиц несколько сложнее, чем для полных. При мутации измениться может не только сама таблица, но и множество значимых переменных. При этом каждая из таких переменных с некоторой вероятностью заменяется другой, которая не принадлежит множеству.

При скрещивании сокращенных таблиц родительские хромосомы состояний могут иметь различные множества значимых переменных. Поэтому сначала необходимо выбрать, какие из этих предикатов будут значимы для хромосом детей. В настоящее время используется вариант, при котором переменные, значимые для обоих родителей, наследуются обоими детьми, а каждая из тех переменных, которые были значимы лишь для одной родительской особи, равновероятно достаются любому из двух детей. После этого скрещиваются сами сокращенные таблицы переходов. При этом на значения каждой строки таблицы ребенка влияют значения нескольких строк родительских таблиц. Конкретное значение, помещаемое в ячейку таблицы ребенка, определяется «голосованием» всех влияющих на нее ячеек таблиц родителей.

Для реализации метода сокращенных таблиц была разработана библиотека *AutoGen* [112]. Эта библиотека была успешно использована для генерации автомата верхнего уровня управления самолетом [113].

Перейдем ко второму методу компактного представления автоматов – методу деревьев решений. Дерево решений является удобным способом задания дискретной функции, зависящей от конечного числа дискретных переменных. Оно представляет собой помеченное дерево, метки в котором расставлены по следующему правилу:

- внутренние узлы помечены символами переменных;
- ребра – значениями переменных;
- листья – значениями искомой функции.

Для определения значения функции по значениям переменных необходимо спуститься от корня до листа, и сформировать значение, которым помечен полученный лист. При этом из вершины, помеченной переменной x, переход производится по тому ребру, значение на котором совпадает со значением переменной x.

Пример дерева решений изображен на рис. 4.11. Это дерево реализует булеву функцию $\neg x_1 \neg x_2 \vee x_1 x_3$. Сокращение размера дерева по сравнению с полной таблицей истинности булевой функции достигается за счет использования ее специфики: если первая переменная ложна, то не играет роли значение третьей переменной, а если истинна – то второй.

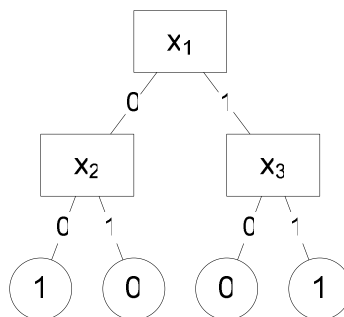

Рис. 4.11. Пример дерева решений

Управляющий автомат можно представить в виде упорядоченного набора деревьев решений, каждое из которых описывает множество переходов из одного конкретного состояния и содержит в листьях не логические значения, а пары <целевое состояние, выходное воздействие>.

Определим генетические операции над деревьями решений. При мутации дерева выбирается случайный узел. После этого поддерево с корнем в этом узле изменяется на случайно сгенерированное. При скрещивании деревьев в каждой из родительских особей случайно выбирается по одному узлу. После этого поддерево с корнем в выбранном узле из первого родителя заменяется поддеревом с корнем в выбранном узле из второго родителя. Результатом этой операции скрещивания, в отличие от предыдущих вариантов, является одна новая особь.

Этот метод был успешно использован при создании автоматов, управляющих беспилотными летательными объектами [114]. В работе [115] описано решение этой же задачи, в котором компоненты объекта управления не реализуются вручную, а представлены нейронными сетями и оптимизируются вместе с управляющим автоматом.

Кроме системы управления беспилотными летательными объектами с помощью генетических алгоритмов были созданы системы управления танком для игры *Robocode* [116] и моделью вертолета [117].

В работе [118] описано инструментальное средство для генерации автоматов с использованием генетических алгоритмов. В частности, это средство позволяет сократить время генерации автоматов за счет использования распределенных вычислений.

Отметим также работу *В. Каширина* (http://is.ifmo.ru/papers/kashirin/), посвященную оптимизации функций приспособленности.

Результаты работ по генерации автоматов для систем со сложным поведением на основе генетического программирования, полученные при участии авторов этой книги в ходе выполнения государственного контракта, приведены по адресу http://is.ifmo.ru/genalg/.

Для обучения генетическому программированию применительно к генерации управляющих автоматов под руководством одного из авторов книги создана

виртуальная лаборатория, размещенная по адресу http://svn2.assembla.com/svn/not_instrumental_tool.

В заключение раздела отметим, что при использовании генетического программирования для генерации автоматов, применяемых в *UniMod*-программах, обеспечивается высокий уровень автоматизации построения этого класса программ, так как, автоматически генерируются не только автоматы, но и программный код по ним. Эксперименты [75] показали, что для рассматриваемого класса программ только 20–30% программного кода строится вручную.

Заключение

Итак, мы завершаем изложение парадигмы, метода и технологии программирования, основанных на идее применения конечных автоматов для описания сложного поведения в программных системах. Авторы осознают, что это изложение было неполным. Это связано с тем, что автоматное программирование является современной и динамично развивающейся технологией. В ситуации, которая сложилась на данный момент в индустрии разработки ПО, есть все предпосылки для ее развития.

Теоретики программирования еще в 1968 году открыто признали кризис программного обеспечения [119], который, по мнению многих ученых, продолжается и по сей день [120]. Главный симптом этого кризиса – неспособность разработчиков обеспечить основное свойство программного обеспечения: его *корректность*. В книге [29] по этому поводу сказано следующее: «если система не делает того, что она должна делать, то все остальное – ее быстродействие, хороший пользовательский интерфейс – не имеет особого значения». На практике же корректности программ долгое время уделяли слишком мало внимания. В результате некорректность ПО не только стала привычным делом для пользователей, но и, можно сказать, вошла в поговорку: достаточно вспомнить такие примеры сетевого фольклора, как эссе «Если бы программисты строили дома» [121] или спор «Почему глючат программы?» [122]. За примерами программных ошибок, имевших серьезные последствия, не надо далеко ходить: вспомните провал единой государственной автоматизированной информационной системы (ЕГАИС) в 2006 г., который привел к многомиллионным убыткам алкогольных компаний, или недавний скандал в пятом терминале лондонского аэропорта *Хитроу*, вызванный техническими сбоями в суперсовременной и высокотехнологичной автоматизированной системе сортировки багажа.

Многие теоретики и практики программирования отмечают, что кризис методов разработки ПО проявляется, в основном, при создании систем со сложным поведением. Так, например, профессор *Н. Вирт* (создатель языков программирования *Pascal*, *Modula-2* и *Oberon*) и его коллега *Ю. Гуткнехт* во время визита в Россию в 2005 г. [123] утверждали, что они не видят проблем в программировании, оговорившись при этом, что сказанное не относится к созданию драйверов, которые, как правило, обладают сложным поведением. Другой широкий класс программно-аппаратных систем со сложным поведением – логическое управление. Трудности, возникающие при традиционном подходе к решению задач из этой области, описаны в работе [124].

В последние годы проблема обеспечения корректности является одной из наиболее актуальных в программной инженерии [125]. Об этом говорит большое число исследований и разработок в области тестирования, доказательной верификации программ, проверки моделей. Признанием важности исследований в этом направлении является присуждение премии *А. Тьюринга* за 2007 год создателям метода *Model Checking* – *Э. Кларку*, *А. Эмерсону* и *Д. Сифакису*.

Неблагополучное положение в области обеспечения корректности ПО привело к созданию сообщества исследователей и практиков, озабоченных будущим программной инженерии – *ISEN* (*Interdisciplinary Software Engineering Network*).

Особое внимание это сообщество уделяет междисциплинарным исследованиям и подходам, особенно тем, которые появились в дисциплинах, не связанных с информационными технологиями, задолго до появления компьютеров. Исследования в этом направлении привели к выводу, что при разработке ПО может быть полезен опыт создания систем автоматического управления и теоретические основы кибернетики. В результате родилось целое направление в разработке ПО, названное *программной кибернетикой* [126]. К этому направлению можно отнести и автоматное программирование.

Таким образом, как может убедиться читатель, *автоматное программирование*, обсуждаемое в этой книге, *является ответом на многие наиболее острые и актуальные проблемы индустрии разработки ПО*. В рамках этой парадигмы программирования впервые в явном виде формулируется проблема спецификации и реализации сложного поведения, предлагается использовать традиционные методы программирования совместно с идеями теории автоматов и теории автоматического управления. Корректность автоматных программ закладывается еще на этапе проектирования, благодаря наглядной графической нотации для описания сложного поведения. Кроме того, соответствие автоматной программы ее спецификации может быть проверено формально: использование метода *Model Checking* в этом случае требует от разработчика значительно меньших усилий, чем для программ, написанных без явного выделения состояний, так как модель программы с конечным числом состояний строится уже на этапе проектирования. Здесь имеет место та же ситуация, что и, например, при контроле схем: если схема не спроектирована с самого начала как *контролепригодная*, то ее и не удается проверить [127].

В последние годы все больше программистов-практиков по всему миру приходят к выводу, что в тех задачах, которыми они занимаются, целесообразно использовать конечные автоматы. Например, в 2007 году были опубликованы статьи ведущего разработчика компании *IBM Э. Принга* о реализации всплывающих подсказок с применением автоматов [128, 129]. Парадигма же автоматного программирования призывает использовать автоматные модели при наличии сущностей со сложным поведением всегда, а не только при решении некоторых специфичных задач. Авторы предполагают, что им удалось объяснить, чем отличается парадигма автоматного программирования от программирования *с использованием* автоматов, которое известно начиная с 60-х годов прошлого века [130].

Автоматное программирование все шире используется на практике, особенно при создании программного обеспечения для ответственных систем [131, 132], и обеспечивает значительное сокращение затрат на их сопровождение. Его применение при разработке систем управления судовыми техническими средствами во ФГУП «НПО «Аврора» описано в работе [4]. В последнее время автоматное программирование начало использоваться в нетрадиционных областях, таких как, например, программирование смарт-карт [133].

Идеи автоматного программирования, изложенные в книге, могут в том или ином виде использоваться не только для текстовых и визуальных языков программирования, как описано выше, но и для программируемых логических контроллеров [131, 132], а также различных средств автоматизации [9] и имитационного моделирования [18, 134, 135].

Авторы предполагают, что области применения автоматного программирования будут еще расширяться, так как «более 99% всех микропроцессоров, проданных в 1998 году, использовались во встраиваемых системах, а в 2000 году число микроконтроллеров в высококачественном автомобиле достигало 60» [136].

Не следует недооценивать роль рассмотренной парадигмы в образовании. В частности, на ее основе можно проводить *первоначальное обучение проектированию программ*, не только в университетах [137], но и в средних школах [138]. При этом отметим, что, поскольку концепции автоматного программирования существенно отличаются от традиционных, начинать обучение программированию в этом стиле следует как можно раньше [139]. Начальное представление о работе автоматов можно получить по адресу http://is.ifmo.ru/projects/life_app/.

Авторы также выражают надежду, что технология использования автоматов при разработке программных систем со сложным поведением будет развиваться: появятся новые модели, нотации, инструментальные средства. Например, при участии авторов ведутся работы по созданию текстовых языков автоматного программирования [140–142], декларативных методов описания автоматов на императивных языках программирования [143], методов динамической верификации автоматных программ [144], инструментальных средств на основе концепции предметно-ориентированных языков программирования [145]. Также проводятся работы по применению автоматов при создании ПО для мобильных роботов [146] и автоматизации документооборота [147].

Авторы также надеются, что парадигма автоматного программирования, изложенная в этой книге, станет «каркасом» для дальнейших исследований в области использования автоматов в программировании.

В заключение работы отметим, что создание автоматных программ предполагает их проектирование, названное во введении к книге [4] «автоматным проектированием программ».

Так как при проектировании этого класса программ основное внимание уделяется управлению, то можно говорить о *парадигме управления*, названной в книге [4] «автоматное управление». Эта парадигма, как отмечалось выше, неоднократно апробировалась на практике, в том числе и при проектировании программного обеспечения сложных систем [8, 131, 132].

Отметим также, что использование автоматов при проектировании систем управления [148] до последнего времени в основном рассматривалось в рамках применения гибридных автоматов [149]. Дополнительный интерес к использованию автоматов в управлении возник у специалистов после пленарного доклада *Р. Брокетта* на конгрессе *IFAC* [150], в котором обсуждались вопросы упрощения проектирования сложных систем управления.

Авторы предполагают, что интерес к автоматному программированию будет возрастать и в дальнейшем, чему поможет не только эта первая в мире книга по рассматриваемой тематике, но и выход в свет тематического сборника по автоматному программированию [151], содержащего 28 статей по различным аспектам этого подхода к программированию.

В заключение отметим, что существенное влияние на создание автоматного подхода к программированию оказали работы по логическому управлению, которые

проводились в лаборатории член-корреспондента АН СССР *Михаила Александровича Гаврилова* (1903–1979) в Институте проблем управления РАН [152–156].

Список источников

1. *Непейвода Н. Н.* Стили и методы программирования. М.: Интернет-университет информационных технологий, 2005.

2. *Шалыто А. А.* Программная реализация управляющих автоматов // Судостроительная промышленность. Сер. «Автоматика и телемеханика». 1991. Вып. 13, с. 41, 42. http://is.ifmo.ru/works/switch_prr/

3. *Шалыто А. А.* Парадигма автоматного программирования //Научно-технический вестник СПбГУ ИТМО. Автоматное программирование. 2008. Вып. 53, с. 3–23. http://is.ifmo.ru/science/sbornik_53_automata

4. *Шалыто А. А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб: Наука, 1998. http://is.ifmo.ru/books/switch/1

5. *Shalyto A. A.* Technology of Automata-Based Programming. 2004. http://www.codeproject.com/KB/architecture/abp.aspx

6. *Ежегодные сборники «Командный чемпионат мира АСМ. Северо-Восточный Европейский регион»* / Под редакцией В. Н. Васильева и В. Г. Парфенова. СПбГУ ИТМО. 1996–2008.

7. *Ежегодные сборники «Всероссийская олимпиада школьников по программированию»* / Под редакцией В. Н. Васильева, В. Г. Парфенова и А. С. Станкевича. СПбГУ ИТМО. 2000–2008.

8. *Туккель Н. И., Шалыто А. А.* Система управления дизель-генератором (фрагмент). Программирование с явным выделением состояний. Проектная документация. СПб.: 2002. http://is.ifmo.ru/projects/dg

9. *Вавилов К. В.* LabVIEW и SWITCH-технология. Методика алгоритмизации и программирования задач логического управления. СПб.: 2005. http://is.ifmo.ru/progeny/_vavilov2.pdf.zip

10. *Harel D., Pnueli A.* On the development of reactive systems / In «Logic and Models of Concurrent Systems». NATO Advanced Study Institute on Logic and Models for Verification and Specification of Concurrent Systems. Springer Verlag, 1985. pp. 477–498.

11. *Ахо А., Сети Р., Ульман Д.* Компиляторы. Принципы, технологии, инструменты. М.: Вильямс, 2002.

12. *Хопкрофт Д., Мотвани Р., Ульман Д.* Введение в теорию автоматов, языков и вычислений. М.: Вильямс, 2002.

13. *Ахо А., Ульман Дж.* Теория синтаксического анализа, перевода и компиляции. М.: Мир, 1978.

14. *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами на С++. М.: Бином; СПб.: Невский диалект, 1998.

15. *Лавров С. С.* Программирование. Математические основы, средства, теория. СПб.: БХВ-Петербург, 2001.

16. *Перлис А.* Синтез алгоритмических систем / Лекции лауреатов премии Тьюринга за первые двадцать лет 1966–1885. М.: Мир, 1993.

17. *Дейкстра Э.* Взаимодействие последовательных процессов / Языки программирования. М.: Мир, 1972.

18. *Колесов Ю. Б., Сениченков Ю. Б.* Моделирование систем. Динамические и гибридные системы. БХВ-Петербург, 2006.

19. *Айзерман А. А., Гусев Л. А., Розоноэр Л. И., Смирнова И. М., Таль А. А*. Логика. Автоматы. Алгоритмы. М.: Физматгиз, 1963.

20. *Глушков В. М.* Синтез цифровых автоматов. М.: Физматлит, 1962.

21. *Пестов А. А., Шалыто А. А.* Преобразование недетерминированного конечного автомата в детерминированный. СПбГУ ИТМО. 2003. http://is.ifmo.ru/projects/determ

22. *ГОСТ 19.002-80*. Схемы алгоритмов и программ. Правила выполнения. Госкомстандарт, 1982.

23. *Заде Л., Дезоер Ч.* Теория линейных систем. Метод пространства состояний. М.: Наука, 1970.

24. *Фридман А., Меннон П.* Теория и проектирование переключательных схем. М.: Мир, 1978.

25. *Туккель Н. И., Шалыто А. А.* От тьюрингова программирования к автоматному // Мир ПК. 2002. № 2, с. 144–149. http://is.ifmo.ru/works/turing/

26. *Дворкин М., Станкевич А., Шалыто А.* О применении автоматов при реализации алгоритмов дискретной математики (на примере АВЛ-деревьев). СПбГУ ИТМО. 2008. http://is.ifmo.ru/works/_avl.pdf

27. *Туккель Н. И., Шалыто А. А.* SWITCH-технология – автоматный подход к созданию «реактивных» систем // Программирование. 2001. № 5, с. 45–62. http://is.ifmo.ru/works/switch/1/

28. *Harel D., Polity M.* Modeling Reactive Systems with Statechart. The Statemate Approach. NY: McGraw-Hill, 1998.

29. *Мейер Б.* Объектно-ориентированное конструирование программных систем. М.: Интернет-университет информационных технологий, 2005.

30. *Полигамия.* http://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B8%D0%B3%D0%B0%D0%BC%D0%B8%D1%8F

31. *Гуисов М. И., Кузнецов А. Б., Шалыто А. А.* Интеграция механизма обмена сообщениями в Switch-технологию. СПбГУ ИТМО. 2003. http://is.ifmo.ru/projects/memech

32. *Альшевский Ю. А., Раер М. Г., Шалыто А. А.* Механизм обмена сообщениями для параллельно работающих автоматов (на примере системы управления турникетом). СПбГУ ИТМО. 2003. http://is.ifmo.ru/projects/turn

33. *Буч Г., Рамбо Дж., Якобсон И.* Язык UML: Руководство пользователя. СПб.: Питер, 2004.

34. *Зюбин В. Е.* Программирование информационно-управляющих систем на основе конечных автоматов. Новосибирск: НГУ, 2006.

35. *Шалыто А. А.* Новая инициатива в программировании. Движение за открытую проектную документацию // PC WEEK/RE. 2003. № 40, с. 38, 39, 42. http://is.ifmo.ru/works/open_doc/

36. *Harel D.* Statecharts: A Visual Formalism for Complex Systems // Scientific Computer Programming. 1987. Vol. 8, pp. 231–274.

37. *Рамбо Дж., Якобсон А., Буч Г.* Унифицированный процесс разработки программного обеспечения. СПб.: Питер, 2002.

38. *Буч Г., Рамбо Дж., Якобсон И.* UML. 2-е издание. СПб.: Питер, 2005.

39. *Mellor S., Balcer M.* Executable UML: A Foundation for Model-Driven Architecture. NJ: Addison-Wesley, 2002.

40. *Гуров В. С., Нарвский А. С., Шалыто А. А.* Исполняемый UML из России // PC WEEK/RE. 2005. № 26, с. 18, 19. http://is.ifmo.ru/works/_umlrus.pdf

41. *Шалыто А. А.* Автоматное проектирование программ. Алгоритмизация и программирование задач логического управления // Известия РАН. Теория и системы управления. 2000. № 6, с. 63–81. http://is.ifmo.ru/works/switch/1/

42. *Шалыто А. А.* Логическое управление. Методы аппаратной и программной реализации алгоритмов. СПб.: Наука, 2000. http://is.ifmo.ru/books/log_upr/1

43. *Шалыто А. А.* Использование граф-схем алгоритмов и графов переходов при программной реализации алгоритмов логического управления // Автоматика и телемеханика. 1996. № 6, с. 148–158 http://is.ifmo.ru/works/gsgp1_/1/; № 7, с. 144–169. http://is.ifmo.ru/works/gsgp2_/1/

44. *Керниган Б., Ричи Д.* Язык программирования *С*. М.: Вильямс, 2005.

45. *Шалыто А. А.* Алгоритмизация и программирование для систем логического управления и «реактивных» систем // Автоматика и телемеханика. 2001. № 1, с. 3–39. http://is.ifmo.ru/works/arew/1/

46. *Туккель Н. И., Шалыто А. А.* Программирование с явным выделением состояний // Мир ПК. 2001. № 8, с. 116–121; № 9, с. 132–138. http://is.ifmo.ru/download/mirpk1.pdf

47. *Martin R.* Designing Object-Oriented C++ Applications Using The Booch Method. NJ: Prentice Hall, 1993.

48. *List of UML tools.* http://en.wikipedia.org/wiki/List_of_UML_tools

49. *Головешин А.* Использование конвертора Visio2Switch. http://is.ifmo.ru/progeny/visio2switch

50. *Канжелев С. Ю., Шалыто А. А.* Автоматическая генерация автоматного кода // Информационно-управляющие системы. 2006. № 6, с. 35–42. http://is.ifmo.ru/works/_autogen.pdf

51. *Чарнецки К., Айзенекер У.* Порождающее программирование: методы, инструменты, применение. СПб.: Питер, 2005.

52. *Meyer B.* Design by Contract, Technical Report TR-EI-12/CO. Interactive Software Engineering Inc., 1986.

53. *Adamczyk P.* The anthology of the finite state machine design patterns / The 10th Conference on Pattern Languages of Programs, 2003.

54. *Adamczyk P.* Selected patterns for implementing finite state machines / The 11th Conference on Pattern Languages of Programs, 2004.

55. *Гранд М.* Шаблоны проектирования в Java. М.: Новое знание, 2004.

56. *Стелтинг С., Маассен О.* Применение шаблонов в Java. Библиотека профессионала. М.: Вильямс, 2002.

57. *Богданов М. С., Шалыто А. А.* Сравнение Sequence Diagram и диаграмм Statechart. СПбГУ ИТМО, 2006. http://is.ifmo.ru/misc/diagrms

58. *Туккель Н. И., Шалыто А. А.* Система управления танком для игры Robocode. Объектно-ориентированное программирование с явным выделением состояний. СПбГУ ИТМО. 2001. http://is.ifmo.ru/projects/tanks/

59. *Кук Д., Урбан Д., Хамилтон С.* Unix и не только. Интервью с Кеном Томпсоном // Открытые системы. 1999. № 4, с. 57–61.

60. *Waldén K., Nerson J.-M.* Seamless Object-Oriented Software Architecture: Analysis and Design of Reliable Systems. Prentice Hall, Hemel Hempstead (U.K.), 1995.

61. *Polikarpova N.* Object-oriented approach to modeling and specification of entities with complex behavior / Материалы конференции «Software Engineering Conference (Russia) – SEC(R) 2006». М.: ТЕКАМА. 2006, с. 10–16.

62. *Поликарпова Н. И.* Объектно-ориентированный подход к моделированию и спецификации сущностей со сложным поведением. СПбГУ ИТМО, 2006. http://is.ifmo.ru/papers/oosusch

63. *Guttag J. V., Horning J. J.* Larch: Languages and Tools for Formal Specification. New York: Springer, 1993.

64. *Standard ECMA-367.* Eiffel: Analysis, Design and Programming Language. ECMA International, 2006. http://www.ecma-international.org/publications/standards/Ecma-367.htm

65. *Новиков Ф. А.* Визуальное конструирование программ // Информационно-управляющие системы. 2005. № 6, с. 9–22. http://is.ifmo.ru/works/visualcons/

66. *Гамма Э., Хэлм Р., Джонсон Р., Влиссидес Дж.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб: Питер, 2001.

67. *Harel D., Kupferman O.* On the Behavioral Inheritance of State-Based Objects / The 34th International Conference on Component and Object Technology. Santa Barbara, C.A. 2000.

68. *Шопырин Д. Г., Шалыто А. А.* Графическая нотация наследования автоматных классов // Программирование. 2007. № 5, с. 127–136. http://is.ifmo.ru/works/_12_12_2007_shopyrin.pdf

69. *Гуров В. С., Мазин М. А., Нарвский А. С., Шалыто А. А.* Инструментальное средство для поддержки автоматного программирования // Программирование. 2007, № 6. http://is.ifmo.ru/works/_2008_01_27_gurov.pdf

70. *UniMod.* http://unimod.sourceforge.net/intro.html

71. *Гуров В. С., Мазин М. А., Шалыто А. А.* Ядро автоматного программирования // Свидетельство о государственной регистрации программы для ЭВМ № 2006613249. Зарегистрировано 14.09.2006.

72. *Сайт проекта Eclipse*. http://www.eclipse.org

73. *Гуров В. С., Шалыто А. А., Яминов Б. Р.* Технология верификации автоматных программ без их трансформации во входной язык верификатора / Международная научно-техническая мультиконференция «Проблемы информационно-компьютерных технологий и мехатроники». Материалы международной научно-технической конференции «Многопроцессорные вычислительные и управляющие системы (МВУС'2007)». Таганрог: НИИ МВС. 2007. Т.1, с. 198–203. http://is.ifmo.ru/verification/_jaminov.pdf

74. *Kochelaev D. Y., Khasanzyanov B. S., Yaminov B. R., Shalyto A. A.* Instrumental Tool for Automata Based Software Development UniMod 2 / Proceedings of the Second Spring Young Researchers’ Colloquium on Software Engineering (SYRCoSE 2008). SPbSU. 2008. V. 1, pp. 55–58.

75. *Unimod-проекты.* http://is.ifmo.ru/unimod-projects

76. *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ. М.: МЦНМО, 2000.

77. *Туккель Н. И., Шалыто А. А.* Преобразование итеративных алгоритмов в автоматные // Программирование. 2002. № 5, с. 12–26. http://is.ifmo.ru/works/iter/

78. *Туккель Н. И., Шамгунов Н. Н., Шалыто А. А.* Реализация рекурсивных алгоритмов на основе автоматного подхода // Телекоммуникации и информатизация образования. 2002. № 5, с. 72–99. http://is.ifmo.ru/works/recurse/

79. *Корнеев Г. А., Шамгунов Н. Н, Шалыто А. А.* Обход деревьев на основе автоматного подхода // Компьютерные инструменты в образовании. 2004. № 3, с. 32–37. http://is.ifmo.ru/works/traverse/

80. *Кнут Д.* Искусство программирования. Т. 1. Основные алгоритмы. М:.: Вильямс, 2003.

81. *Касьянов В. Н., Евстигнеев В. А.* Графы в программировании: обработка, визуализация и применение. СПб.: БХВ-Петербург, 2003.

82. *Казаков М. А., Шалыто А. А.* Автоматный подход к реализации анимации в визуализаторах алгоритмов // Компьютерные инструменты в образовании. 2005. № 3, с. 62–76. http://is.ifmo.ru/works/heapsort/

83. *Корнеев Г. А., Шалыто А. А.* Построение визуализаторов алгоритмов дискретной математики // Научно-технический вестник СПбГУ ИТМО. Высокие технологии в оптических и информационных системах. 2005, Вып. 23. с. 118–129. http://is.ifmo.ru/works/_a_visualizerExample.pdf

84. *Визуализаторы алгоритмов.* http://is.ifmo.ru/vis

85. *Hoffman L.* In Search of Dependable Design // Communications of the ACM. 2008. № 07/08, pp. 14–16.
[русский перевод http://is.ifmo.ru/verification/_v_poiskax_nadejnogo_koda.pdf]

86. *Hoffman L.* Talking Model-Checking Technology // Communications of the ACM. 2008. № 07/08, pp. 110–112.
[русский перевод http://is.ifmo.ru/verification/_razgovori_o_model_checking.pdf]

87. *Грис Д*. Наука программирования. М.: Мир. 1984.

88. *Непомнящий В. А., Рякин О. М.* Прикладные методы верификации программ. М.: Радио и связь. 1988.

89. *Кларк Э. М., Грамберг О., Пелед Д.* Верификация моделей программ. Model Checking. М.: МЦНМО, 2002.

90. *Pnueli A.* The Temporal Logic of Programs / Proceedings of the 18th IEEE Symposium on Foundation of Computer Science. 1977.

91. *Мазин М. А., Шалыто А. А.* Анимация. Flash-технология. Автоматы // Методическая газета для учителей информатики «Информатика». 2006. № 11, с. 36–47. http://is.ifmo.ru/works/flash_aut/

92. *Вельдер С. Э., Шалыто А. А.* О верификации простых автоматных программ на основе метода Model Checking // Информационно-управляющие системы. 2007, № 3. http://is.ifmo.ru/download/27-38.pdf

93. *Кузьмин Е. В., Соколов В. А.* Моделирование спецификация и верификация «автоматных» программ // Программирование. 2008, № 1. http://is.ifmo.ru/download/2008-03-12_verification.pdf

94. *Яминов Б. Р., Шалыто А. А.* Расширение верификатора Bagor для верификации автоматных UniMod-моделей // Свидетельство о государственной регистрации программы для ЭВМ № 2008611055. Зарегистрировано 28.02.2008.

95. *Лукин М. А., Шалыто А. А.* Трансляция UniMod-модели во входной язык верификатора SPIN // Свидетельство о государственной регистрации программы для ЭВМ № 2008611181. Зарегистрировано 06.03.2008.

96. *Kurbatsky E.* Verification of Automata-Based Programs / Proceedings of the Second Spring Young Researchers' Colloquium on Software Engineering (SYRCoSE 2008). SPbSU. 2008. V. 2, pp. 15–17. http://is.ifmo.ru/verification/_kurbatsky_syrcse.pdf

97. *Официальный сайт проекта OpenMP.* http://openmp.org

98. *Сайт проекта MPI.* http://www.mpi-forum.org

99. *Любченко В., Тяжлов Ю.* Осторожно: многоядерный процессор // Открытые системы. 2007. № 6, с. 28–31.

100. *Котов А. Н., Шалыто А. А.* Сравнение различных вариантов реализации на примере задачи о декодировании файлов формата GIF. СПбГУ ИТМО. 2007. http://is.ifmo.ru/projects/gif-parser

101. *Koza J.* Genetic programming. On the Programming of Computers by Means of Natural Selection. MA: The MIT Press, 1998.

102. *Гладков Л. А., Курейчик В. В., Курейчик В. М.* Генетические алгоритмы. М.: Физматлит. 2006.

103. *Фогель Л., Оуэнс А., Уолш М.* Искусственный интеллект и эволюционное моделирование. М.: Мир, 1969.

104. *Angeline P. J., Pollack J.* Evolutionary Module Acquisition / Proceedings of the Second Annual Conference on Evolutionary Programming. 1993. http://www.demo.cs.brandeis.edu/papers/ep93.pdf

105. *Jefferson D., Collins R., Cooper C. et al.* The Genesys System. 1992. www.cs.ucla.edu/~dyer/Papers/AlifeTracker/Alife91Jefferson.html

106. *Chambers L.* Practical Handbook of Genetic Algorithms. Complex Coding Systems. Volume III. CRC Press, 1999.

107. *Царев Ф. Н., Шалыто А. А.* О построении автоматов с минимальным числом состояний для задачи об «Умном муравье» / Сборник докладов X Международной конференции по мягким вычислениям и измерениям. СПбГЭТУ. 2007. Т. 2, с. 88–91. http://is.ifmo.ru/download/ant_ga_min_number_of_state.pdf

108. *Давыдов А. А., Соколов Д. О., Царев Ф. Н., Шалыто А. А.* Применение островного генетического алгоритма для построения автоматов Мура и систем взаимодействующих автоматов Мили на примере задачи об «Умном муравье» / Сборник докладов XI Международной конференции по мягким вычислениям и измерениям (SCM'2008). СПбГЭТУ. 2008, с. 266–270. http://is.ifmo.ru/genalg/_scm2008_sokolov.pdf

109. *Лобанов П. Г., Шалыто А. А.* Использование генетических алгоритмов для автоматического построения конечных автоматов в задаче о флибах // Известия РАН. Теория и системы управления. 2007. № 5, с. 127–136. http://is.ifmo.ru/works/_15_11_2007_lobanov_shalyto.djvu

110. *Поликарпова Н. И., Точилин В. Н., Шалыто А. А.* Применение генетического программирования для реализации систем со сложным поведением / IV Международная научно-практическая конференция «Интегрированные модели и мягкие вычисления в искусственном интеллекте». Том 2. М.: Физматлит. 2007, с. 598–604. http://is.ifmo.ru/genalg/_polikarpova.pdf

111. *Данилов В. Р., Шалыто А. А.* Метод генетического программирования для генерации автоматов, представленных деревьями решений / Научно-техническая конференция «Научное программное обеспечение в образовании и научных исследованиях». СПбГПУ. 2008, с. 174–181. http://is.ifmo.ru/download/2008-03-07-danilov.pdf

112. *Поликарпова Н. И., Точилин В. Н.* Генетический генератор автоматов // Свидетельство о государственной регистрации программы для ЭВМ № 2008610473. Зарегистрировано 25.01.2008.

113. *Точилин В. Н.* Метод сокращенных таблиц для генерации автоматов с большим числом входных воздействий на основе генетического программирования. СПбГУ ИТМО. 2008. http://is.ifmo.ru/papers/tochilin

114. *Технология генетического программирования для генерации автоматов управления системами со сложным поведением.* Экспериментальные исследования поставленных в контракте задач. СПбГУ ИТМО. 2008. http://is.ifmo.ru/genalg/_2007_03_report-genetic.pdf

115. *Царев Ф. Н., Шалыто А. А.* Совместное применение генетического и автоматного программирования для построения мультиагентной системы / Материалы XII Всероссийской конференции по проблемам науки и высшей школы «Фундаментальные исследования и инновации в технических университетах». СПбГПУ, с. 213–215. http://is.ifmo.ru/download/tsarev_slides.ppt

116. *Бедный Ю. Д., Шалыто А. А.* Применение генетических алгоритмов для создания системы управления танком в игре «Robocode» / Сборник докладов XI Международной конференции по мягким вычислениям и измерениям (SCM'2008). СПбГЭТУ. 2008, с. 261–265. http://is.ifmo.ru/genalg/_robocode.pdf

117. *Лобанов П. Г., Сытник С. А.* Использование генетических алгоритмов для построения автопилота для простейшего вертолета / Материалы XV Международной научно-методической конференции «Высокие интеллектуальные технологии и инновации в образовании и науке». СПбГПУ. 2008, с. 298. http://is.ifmo.ru/download/2008-03-01_vertolet.pdf

118. *Мандриков Е. А., Кулев В. А.* Применение распределенных вычислений для автоматической генерации конечных автоматов с использованием генетических алгоритмов / Сборник докладов XI Международной конференции по мягким вычислениям и измерениям (SCM'2008). СПбГЭТУ. 2008, с. 255–260.

119. *Дейкстра Э.* Смиренный программист // Лекции лауреатов премии Тьюринга за первые двадцать лет. 1966–1985. М.: Мир, 1993.

120. *Черняк Л.* Адаптируемость и адаптивность // Открытые системы. 2004. № 9, с. 27–29.

121. *Если бы программисты строили дома*. http://bayanov.net/text-komp/programm_dom.php

122. *Почему глючат программы?* http://bayanov.net/text-komp/mir_rel_gluki.php

123. *Шалыто А. А.* Никлаус Вирт – почетный доктор СПбГУ ИТМО // Компьютерные инструменты в образовании. 2005. № 5, с. 3–7. http://is.ifmo.ru/belletristic/_wirth_poch.pdf

124. *Вавилов В. К., Шалыто А. А.* Что плохого в неавтоматном подходе к программированию контроллеров? // Промышленные АСУ и контроллеры. 2007. № 1, с. 49–51. http://is.ifmo.ru/works/_Asu-2007-01.pdf

125. *Мейер Б.* Речь на торжественном вручении мантии и диплома почетного доктора СПбГУ ИТМО // Компьютерные инструменты в образовании. 2006. № 3, с. 7–9.

126. *Cai Kai-Yuan, Chen T. Y., Tse T. H.* Towards Research on Software Cybernetics / 7th IEEE International Conference on High-assurance Systems Engineering (HASE 2002). Los Alamitos. IEEE Computer Society Press, 2002.

127. *Мосин С.* Г. Структурные решения тестопригодного проектирования заказных интегральных схем // Информационные технологии. 2008. № 11, с. 2 – 10.

128. *Принг Э.* Конечные автоматы в *JavaScript*, Часть1: Разработаем виджет. http://www.ibm.com/developerworks/ru/library/wa-finitemach1/index.html

129. *Принг Э.* Конечные автоматы в *JavaScript*, Часть2: Реализация виджета. http://www.ibm.com/developerworks/ru/library/wa-finitemach2/wa-finitemac_ru.html

130. *Johnson W., Porter J., Ackley, S., Ross D.* Automatic generation of efficient lexical processors using finite state techniques / Communications of the ACM. 1968. N 12, pp. 805–813.

131. *Вавилов К. В.* Программируемые логические контроллеры SIMATIC S7-200 (SIEMENS). Методика алгоритмизации и программирования задач логического управления. СПб.: 2005. http://is.ifmo.ru/progeny/_metod065.pdf

132. *Вавилов К. В.* Контроллеры SIMATIC S7-300 (SIEMENS). Организация взаимодействия локальных систем управления на основе автоматного подхода и функционального разделения автоматов управления. СПб.: 2005. http://is.ifmo.ru/progeny/_s7300.pdf

133. *Klebanov A.* Automata-Based Programming Technology Extension for Generation of JML Annotated Java-Card Code / Proceeding of the Second Spring Young Researchers' Colloquium on Software Engineering (SYRCoSE'2008). SPbSU. 2008. Vol.1, pp. 41–44. http://is.ifmo.ru/articles_en/_klebanov_spbsu.pdf

134. *Карпов Ю. Г.* Имитационное моделирование систем. Введение в моделирование с AnyLogic 5. БХВ-Петербург, 2006.

135. *Дьяконов В. П.* Simulink 5/6/7. Самоучитель. М.: ДМК Пресс, 2008.

136. *Ослэндер Д., Риджли Д., Ринггенберг Д.* Управляющие программы для механических систем. Объектно-ориентированное проектирование систем реального времени. М.: БИНОМ. Лаборатория знаний. 2004.

137. *Васильев В. Н., Казаков М. А., Корнеев Г. А., Парфенов В. Г., Шалыто А. А.* Применение проектного подхода на основе автоматного программирования при подготовке разработчиков программного обеспечения / Труды первого Санкт-Петербургского конгресса «Профессиональное образование, наука, инновации в XXI веке». СПбГУ ИТМО. 2007, с. 98–100. http://is.ifmo.ru/download/2008-02-25_comp_proekt.pdf

138. *Красильников Н. Н., Парфенов В. Г., Царев Ф. Н., Шалыто А. А.* Виртуальная лаборатория для первоначального обучения проектированию программ // Компьютерные инструменты в образовании. 2007. № 5, с. 62–67. http://is.ifmo.ru/download/2008-02-25_virtual_laboratory.pdf

139. *Кузнецов Б. П.* Психология автоматного программирования // BYTE/Россия. 2000. № 11, с. 28–32. http://www.softcraft.ru/design/ap/ap01.shtml

140. *Гуров В. С., Мазин М. А., Шалыто А. А.* Текстовый язык автоматного программирования / Тезисы докладов Международной научной конференции, посвященной памяти профессора А. М. Богомолова «Компьютерные науки и технологии». Саратов: СГУ. 2007, с. 66–69. http://is.ifmo.ru/works/_2007_10_05_mps_textual_language.pdf

141. *Степанов О. Г., Шалыто А. А., Шопырин Д. Г.* Предметно-ориентированный язык автоматного программирования на базе динамического языка Ruby // Информационно-управляющие системы. 2007. № 4, с. 22–27. http://is.ifmo.ru/works/_2007_10_05_aut_lang.pdf

142. *Лагунов И. А.* Разработка текстового языка автоматного программирования и его реализация для инструментального средства UniMod на основе автоматного подхода. СПбГУ ИТМО, 2008. http://is.ifmo.ru/papers/fsml

143. *Астафуров А. А., Шалыто А. А.* Декларативный подход к вложению и наследованию автоматных классов при использовании императивных языков программирования / Материалы конференции «Software Engineering Confernece (Russia) – SEC(R) 2007». М.: TEKAMA. 2007, с. 230–238. http://is.ifmo.ru/works/_2007_10_05_aut_lang.pdf

144. *Stepanov O., Shalyto A.* A Method for Automatic Runtime Verification of Automata-Based Programs / Proceeding of the Second Spring Young Researchers' Colloquium on Software Engineering (SYRCoSE'2008). SPbSU. 2008. Vol.2, pp.19–23.

145. *Решетников Е. О.* Инструментальное средство для визуального проектирования автоматных программ на основе Microsoft Domain-Specific Language Tools. СПбГУ ИТМО. 2007. http://is.ifmo.ru/papers/reshetnikov_bachelor

146. *Клебан В. О., Шалыто А. А.* Использование автоматного программирования для построения многоуровневых систем управления мобильными роботами / Сборник тезисов 19 Всероссийской научно-технической конференции «Экстремальная робототехника». СПб: ЦНИИ РТК, 2008, с. 85–87.

147. *Клебан В. О.*, *Новиков Ф. А.* Применение конечных автоматов в документообороте // Научно-технический вестник СПбГУ ИТМО. Автоматное программирование. 2008, Вып. 53, с. 286–294.

148. *Гудвин Г.К., Гребе С.Ф., Сальгадо М.Э.* Проектирование систем управления. М.: Бином, 2004.

149. *Alur R., Courcoubetis C., Henzinger A., Ho P.* Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems //Lecture notes in computer science. 1993. V.736, pp. 209–229.

150. *Brokett R.* Reduced Complexity control systems /Proceedings of the 17-th World Congress The International Federation of Automatic Control. Seoul. 2008.

151. *Научно-технический вестник СПбГУ ИТМО.* Автоматное программирование. 2008. Вып.53. 314 с. http://books.ifmo.ru/ntv/ntv/53/ntv_53.pdf

152. *Гаврилов М. А., Девятков В. В., Пупырев Е. И.* Логическое проектирование дискретных автоматов. М.: Наука, 1977.

153. *Михайлов Г. И., Руднев В. В.* Простейшие системы взаимосвязанных графов и расширение практических возможностей конечных автоматов // Автоматика и телемеханика. 1979. № 10, с. 45–52.

154. *Иванов Н. Н.* О поведении взаимодействующих автоматов, моделирующих систему «объект управления – управляющее устройство» // Автоматика и телемеханика. 1979. № 7, с. 47–53.

155. *Амбарцумян А. А., Искра С. А., Кривандина Н. Ю. и др.* Проблемно-ориентированный язык описания поведения систем логического управления ФОРУМ–М / Проектирование устройств логического управления. М.: Наука, 1974.

156. *Кузнецов О. П., Макаревский А. Я., Марковский А. В., Шипилина Л. Б. и др.* ЯРУС – язык описания работы сложных автоматов // Автоматика и телемеханика. 1972. № 6, с. 58–67; № 7, с. 53–63.

Предметный указатель

—M—

Model Checking. см. верификация модели

—S—

State, 120

Statecharts, 72

Statemate, 72

—А—

абстрактный тип данных, 96

автомат, 12, 17

  абстрактный, 19

  без выходного преобразователя, 27

    универсальный, 30

  без выходов, 12

  без памяти, 26

  вложенный, 53

  второго рода, 18

  вызываемый, 53

  Мили, 20, 32

  Мура, 20, 31

  недетерминированный, 21

  первого рода, 18

  преобразователь, 20

  распознаватель, 20

  с магазинной памятью, 23

  с памятью, 27

  смешанный, 33

  со спонтанными переходами, 21

  структурный, 19

автоматизированные объекты управления как классы, 102

автоматизированный объект управления, 17, 38

  активный, 35

  замкнутый, 36

  пассивный, 35

  разомкнутый, 36

автоматная декомпозиция

  параллельная, 63

  по объектам управления, 57

  по режимам, 56

автоматное проектирование

  от объектов управления и событий, 47

  сверху вниз, 43

автоматы и объекты управления как классы, 100

активный объект, 99

алфавит, 19

—Б—

блок-схема. см. схема алгоритма

—В—

валидация, 131, 137

верификация, 138

  автоматных программ, 139

  доказательная, 138

  модели, 138

визуализатор, 137

внутренняя переменная, 26

входная переменная, 12, 26

входное воздействие, 12

выходная переменная, 26

выходное воздействие, 12

вычислительная мощность, 20

—Г—

генетический алгоритм, 141

граф переходов, 13, 68

—Д—

декомпозиция

автоматная, 51

объектная, 95

сверху вниз, 42

диаграмма

взаимодействия, 68

деятельностей, 73

классов, 107

модулей, 73

переходов. см. граф переходов

состояний, 73

динамическая система, 17

конечная, 18

достижимость, 138

—И—

изоморфизм кода графу переходов, 78

инструментальные средства

MetaAuto, 94

UniMod, 128

исполнительный механизм, 47

—К—

класс, 96

автоматизированный, 102

кодирование состояний

принудительное, 28

принудительно-свободное, 30

свободное, 30

комбинационная схема. см. автомат без памяти

—М—

машина Тьюринга, 12, 24

метод деревьев решений, 148

метод сокращенных таблиц, 147

мутация, 141

—Н—

наблюдаемость, 32

наследование, 96

непротиворечивость, 69, 138

—О—

обмен номерами состояний, 55

обмен сообщениями, 55

обратная связь, 16, 27, 36

обход дерева, 134

объект, 96

объект управления, 16, 34

—П—

полнота, 70, 139

принцип подстановочности, 110

программирование

автоматное, 4

парадигма, 17

генетическое, 141

объектно-ориентированное, 95

процедурное, 42

с явным выделением состояний

объектно-ориентированное, 97

процедурное, 42

тьюрингово, 39

программная кибернетика, 152

программные системы

интерактивные, 7
реактивные, 7
трансформирующие, 7
проектирование по контракту, 96
проектная документация, 72
простое поведение, 8
протоколирование, 138
—С—
сигнализатор, 47
скрещивание, 141
сложное поведение, 7
событие, 12, 29
состояние, 11, 15
вложенное, 75
вычислительное, 15
завершающее, 35
ортогональное, 75
управляющее, 15
явное выделение, 16
спецификация
поведения, 68, 109
декларативная, 110
императивная, 109
структуры, 64, 107
чрезмерная, 110
суперсостояние, 75
схема алгоритма, 22
схема связей, 64
—Т—
таблица
истинности, 26
решений, 27
такт, 18
тезис Черча-Тьюринга, 12
темпоральная логика, 138
—Ф—
функция
выходов, 12
переходов, 12
приспособленности, 141
—Э—
элемент задержки, 27
—Я—
язык, 20
контекстно-свободный, 23
регулярный, 20