

**Санкт-Петербургский государственный университет
Информационных технологий, механики и оптики**

Кафедра «Компьютерные технологии»

Н. И. Поликарпова, А. А. Шалыто

Автоматное программирование
Учебно-методическое пособие

Санкт-Петербург

2007

Оглавление	
Предисловие	3
Глава 1. Введение в автоматное программирование	4
1.1. Области применения автоматного подхода	4
1.2. Основные понятия	8
1.3. Парадигма автоматного программирования	9
1.4. Автоматные модели	14
Глава 2. Процедурное программирование с явным выделением состояний	36
2.1. Проектирование	37
2.2. Спецификация	58
2.3. Реализация	70
Глава 3. Объектно-ориентированное программирование с явным выделением состояний	87
3.1. Проектирование	89
3.2. Спецификация	95
3.3. Реализация	99
Литература	107

Предисловие

Предметом настоящей работы является парадигма *автоматного программирования*. Автоматное программирование, иначе называемое «*программирование от состояний*» или «*программирование с явным выделением состояний*» — это метод разработки программного обеспечения, 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 построении и реализации конечных автоматов для использования в программах, сколько о методе создания программ в целом, поведение которых описывается автоматами.

Автоматное программирование 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 рассматривается как один из основных стилей программирования в книге [1]. Однако полного и исчерпывающего изложения сути автоматного программирования как парадигмы и метода разработки программных систем в целом в настоящий момент нет. Данная работа может считаться первым шагом к восполнению этого пробела.

Цель работы состоит в определении терминов, образующих словарь парадигмы автоматного программирования, и систематическом изложении ее основных концепций. Настоящая работа также содержит ряд примеров конкретного применения автоматного программирования для решения разнообразных прикладных задач. Эти примеры призваны продемонстрировать действенность, плодотворность и перспективность данной парадигмы.

Настоящая работа 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 созданию программного обеспечения. Изложение охватывает все аспекты создания программной системы: проектирование, спецификацию и реализацию. Значительная часть этой 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 опускают, 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.

Отметим, что термин «*программирование*» в настоящей работе употребляется в широком смысле. Как процесс, деятельность 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 работе используется термин «*реализация*».

Глава 1. Введение в автоматное программирование

1.1. Области применения автоматного подхода

В соответствии с классификацией, введенной Д. Харелом [2], 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 [3], которые относятся к классу трансформирующих систем. Автомат здесь понимается как некое вычислительное устройство, имеющее входную и выходную ленту. 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. Обработывая эти сигналы, автомат формирует значения нескольких параллельных выходов.

Таким образом, даже традиционные области применения конечных автоматов охватывают принципиально различные классы программных систем. Авторы 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 текущего состояния сущности и от предыстории. Для сущностей с *простым поведением* реакция на любое входное воздействие зависит только от этого воздействия¹ (рис. 1.1).

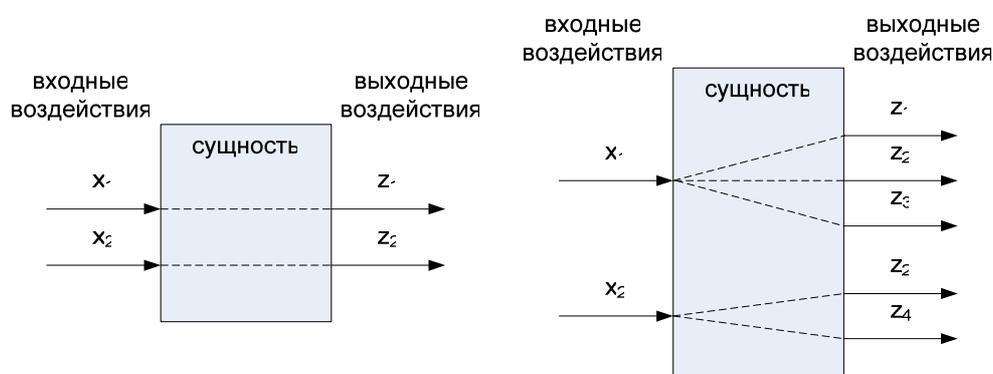


Рис. 1.1. Сущность с простым поведением (слева) и со сложным поведением (справа)

Рассмотрим в качестве примера электронные часы (рис. 1.2). Пусть у них имеется только две кнопки, которые служат для установки текущего времени: кнопка «Н» (*Hours*) увеличивает на единицу число часов, а кнопка «М» (*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.

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.



Рис. 1.2. Электронные часы

Рассмотрим теперь электронные часы с будильником (рис. 1.3). Дополнительная кнопка «А» (*Alarm*) служит в них для включения и выключения будильника. Если будильник выключен, то кнопка «А» включает его и переводит часы в режим, в котором кнопки «Н» и «М» устанавливают не текущее время, а время срабатывания будильника. Повторное нажатие кнопки «А» возвращает часы в обычный режим. Наконец, нажатие кнопки «А» в обычном режиме при включенном будильнике приводит к выключению будильника.

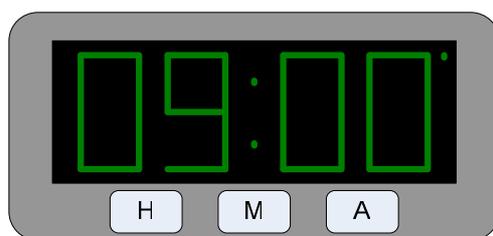


Рис. 1.3. Электронные часы с будильником

Поведение часов с будильником уже является сложным, поскольку одни и те же входные воздействия (нажатия 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 очень просто: 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 C++ при помощи флагов

```
class Clock {  
public:
```

```

...
void H()
{
    if (isInAlarmTimeMode) {
        alarmHrs = (alarmHrs + 1) % 24;
    } else {
        hrs = (hrs + 1) % 24;
    }
}

void M()
{
    if (isInAlarmTimeMode) {
        alarmMins = (alarmMins + 1) % 60;
    } else {
        mins = (mins + 1) % 60;
    }
}

void A()
{
    if (isAlarmOn) {
        if (isInAlarmTimeMode) {
            isInAlarmTimeMode = false;
        } else {
            isAlarmOn = false;
        }
    } else {
        isAlarmOn = true;
        isInAlarmTimeMode = true;
    }
}
...
private:
    int hrs;
    int mins;
    int alarmHrs;
    int alarmMins;

    bool isAlarmOn;
    bool isInAlarmTimeMode;
};

```

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 два свойства делают автоматное описание сложного поведения ясным и удобным.

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 компонент 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 двум 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 задачи индивидуально.

1.2. ОСНОВНЫЕ ПОНЯТИЯ

Базовым понятием автоматного программирования является *«состояние»*. Понятие состояния в том смысле, как оно используется в данной парадигме, было введено А. Тьюрингом. Это понятие 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 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 величину, 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.

Понятие *входного воздействия* 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) *автоматом* (рис. 1.4), получается, если соединить понятие автомата без выходов с понятием *выходного воздействия*. 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.

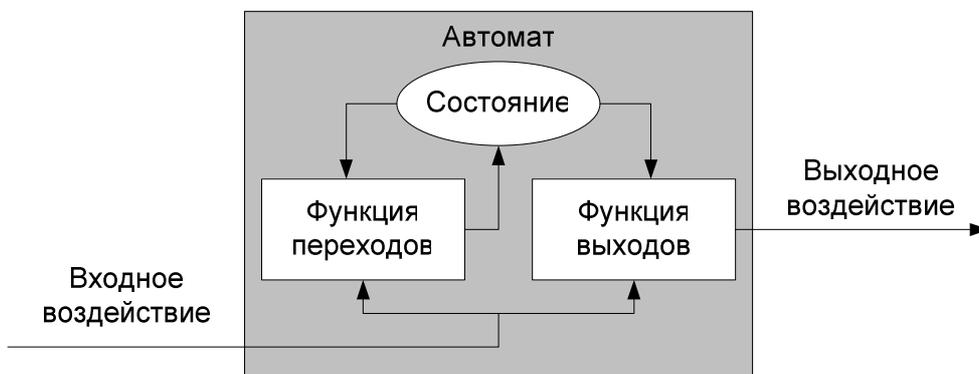


Рис. 1.4. Конечный автомат

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* [5, 6]. Эта абстрактная машина была предложена А. М. Тьюрингом в 1936 г. в качестве формального определения понятия «алгоритм». Тезис Черча-Тьюринга [6] 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 состоит из бесконечного числа 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.

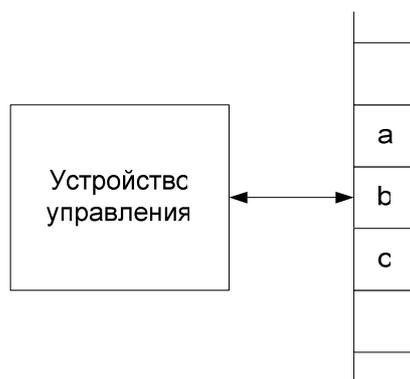


Рис. 1.5. Машина Тьюринга

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 запрограммировать 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 встретиться пустой символ.
2. Сдвинуться на одну ячейку влево.
3. Пока в текущей ячейке находится '1', заменять 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»).

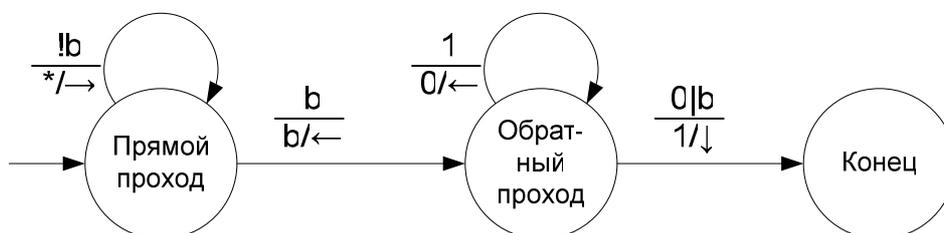


Рис. 1.6. Увеличение числа на единицу с помощью машины Тьюринга

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 различное 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 действия 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 допустимых компонентов 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, позаимствованные из теории управления. Состояния автомата называются *управляющими*, а состояния ленты – *вычислительными*².

² В автоматном программировании (и, в частности, в этом пособии), говоря без уточнения о *состоянии* некоторой автоматной модели, сущности или системы со сложным поведением, подразумевают управляющее состояние. Если речь идет о вычислительном состоянии, это всегда указывается явно.

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 [7, 8]. 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 же прямолинейно. Так, А. Дж. Перлис в 1966 г. [9] 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 [10] предложил ввести так называемые переменные состояния, с помощью которых можно описывать состояния системы в любой момент времени (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.

Таблица 1.1. Управляющие и вычислительные состояния

Управляющие состояния	Вычислительные состояния
Их несколько	Их количество 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. Информацию, содержащуюся в состояниях управляющего автомата, также можно записать на ленте. Машина точно также продолжала бы вести себя по-разному на разных шагах алгоритма, однако, из ее описания это было бы уже не очевидно. 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 управления, 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 входных воздействий, 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 поскольку для реализации управляющей части используются автоматы, то она так и 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 значит, может быть легко реализован традиционными «неавтоматными» методами). Он не обрабатывает непосредственно входные воздействия от внешней среды, а только получает от автомата команды совершить то или иное действие. Каждая команда всегда влечет 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 вместе 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. Его цель – 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 идеологии автоматного программирования, однако, из-за более формального изложения, он труднее для понимания, чем остальные разделы настоящей работы. 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,

- математическая лингвистика;
- логическое управление;
- моделирование поведения человека;
- 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 [11].

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 этого счетного множества 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 $Y(t)$, $|Y| = n$, и на вход которой 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.

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

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

Из приведенных формул видно, что 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*.

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 настоящей работе 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 [12], а с другой – является совершенно самостоятельной областью. При использовании автоматов в программировании не имеют значения ни исследования вычислительной мощности, ни минимизация числа функциональных элементов и вопросы кодирования. В этой области нас интересует только то, как использовать конечные автоматы для построения корректных и надежных 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,

$\delta : Y \times X \rightarrow Y$ — функция переходов, $y_0 \in Y$ — начальное (стартовое) состояние, $F \subset Y$ — множество допускающих состояний.

Расширенная функция переходов $\hat{\delta} : X^* \times Y \rightarrow Y$, сопоставляющая новое состояние текущему состоянию и цепочке символов, определяется индуктивно следующим образом [5]:

$$\begin{aligned} \forall y \in Y \quad (\hat{\delta}(\varepsilon, y) = y); \\ \forall y \in Y \quad \forall \xi \in X^* \quad \forall x \in X \quad (\hat{\delta}(\xi x, y) = \delta(x, \hat{\delta}(\xi, y))). \end{aligned}$$

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 автомат допускает эту цепочку. Множество допускаемых цепочек образует язык L , распознаваемый ДКА:

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

Класс языков, распознаваемых ДКА, называют *регулярными языками*. Известно, что он совпадает с классом языков, описываемых регулярными выражениями и автоматными грамматиками [5].

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 $\varphi : X \times Y \rightarrow Z$, то вычислитель называется *автоматом Мули*, а если $\varphi : Y \rightarrow Z$ — *автоматом Мура*. Таким образом, рассматривается шестерка $\langle X, Y, Z, \delta, \varphi, y_0 \rangle$. Она определяет автоматное отображение $f : X^* \rightarrow Z^*$ (преобразование, выполняемое автоматом) следующим образом:

$$\begin{aligned} f(\varepsilon) = \varepsilon ; \\ \forall \xi \in X^* \quad \forall x \in X \quad (f(\xi x) = f(\xi)\varphi(x, \hat{\delta}(\xi, y_0))). \end{aligned}$$

Автоматные отображения — это отображения «без предвосхищения»: 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, $\delta: X \times Y \rightarrow 2^Y$ — 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 $\hat{\delta}: X^* \times Y \rightarrow 2^Y$ 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:

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

Известно, что недетерминизм не увеличивает вычислительной мощности модели и для каждого НКА существует эквивалентный (распознающий тот же язык) ДКА [5].

Автоматы со спонтанными переходами. С программистской точки зрения 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 допущение недетерминизма [5]. 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 решение о совершении перехода, исходя из результата проверки. В этом случае говорят, что переход *помечен* (или *охраняется*) условием.

Еще одно важное значение спонтанных переходов для программирования состоит в том, что их использование позволяет считать традиционные *блок-схемы* (или *схемы алгоритмов*) 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.

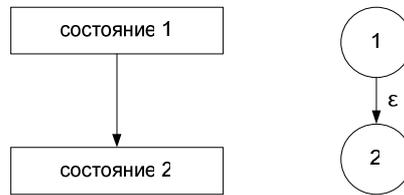


Рис. 1.7. Элемент схемы алгоритма с единственным спонтанным переходом (слева) и его эквивалент на диаграмме переходов (справа)

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

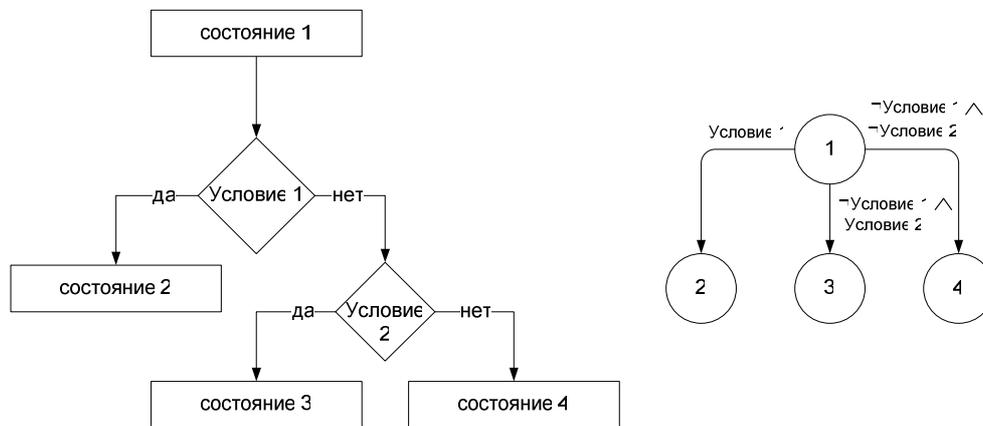


Рис. 1.8. Элемент схемы алгоритма с несколькими исходящими переходами (слева) и его эквивалент на диаграмме переходов (справа)

Именно для блок-схем 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, используются 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 некоторое 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, $\delta: X \times Y \times W \rightarrow Y \times W^*$ — 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 [5].

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

$$\forall y \in Y \forall x \in X \forall w \in W ;$$

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

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

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

$$\mathcal{E}(\xi, y) = q \Leftrightarrow \forall \rho \in X^* (x, \xi\rho) \rightarrow^* (q, \rho).$$

Достижимость конфигураций полностью описывается расширенной функцией переходов.

Используется два способа определения языка, распознаваемого МП-автоматом:

- $L_1 = \{ \xi \in X^* \mid \exists y \in F : (y_0, \xi, w_0) \rightarrow^* (y, \varepsilon, \omega) \}$ – язык, распознаваемый по допускающему состоянию. В этом определении содержимое магазина (цепочка ω) в момент исчерпания входной цепочки и достижения допускающего состояния не имеет значения;
- $L_2 = \{ \xi \in X^* \mid (y_0, \xi, w_0) \rightarrow^* (y, \varepsilon, \varepsilon) \}$ – язык, распознаваемый по опустошению магазина. В этом определении множество допускающих состояний F не имеет значения и его можно исключить из определения МП-автомата.

Известно [5], 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 дано 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 [5].

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 есть 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, показания датчиков. 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 аппаратной 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 наделен определенным смыслом. 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*.

Чтобы подчеркнуть векторную природу состояний, входных и выходных воздействий далее в этом разделе будем обозначать их \bar{y} , \bar{x} и \bar{z} соответственно.

Перейдем к описанию структурных моделей конечных автоматов.

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

Отметим, что здесь 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). Обозначение комбинационной схемы приведено на рис. 1.9.

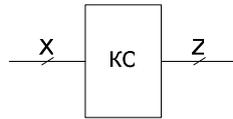


Рис. 1.9. Комбинационная схема

Функционирование комбинационных схем описывается соотношением вида:

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

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 двух типов элементов: 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*, подающая сигнал с выхода автомата обратно на вход, обеспечивают его память.

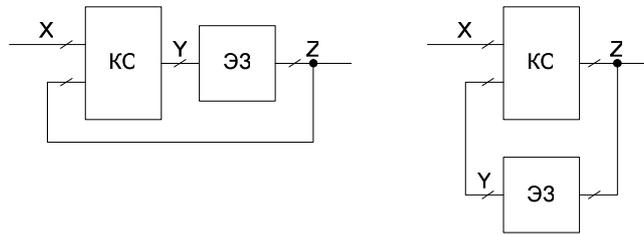


Рис. 1.10. Автоматы без выходного преобразователя: первого рода (слева) и второго рода (справа)

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

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

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

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

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

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

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 «автомата 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 означает, что функция выходов зависит только от состояния и является тождественной). Иначе говоря, в этом случае каждое состояние автомата обозначается (*кодифицируется*) значением выходного воздействия в этом состоянии. Такое кодирование состояний называется *принудительным* [4].

Автоматы без выходного преобразователя имеют ограниченную область применения: они пригодны лишь, 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 состояния 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).

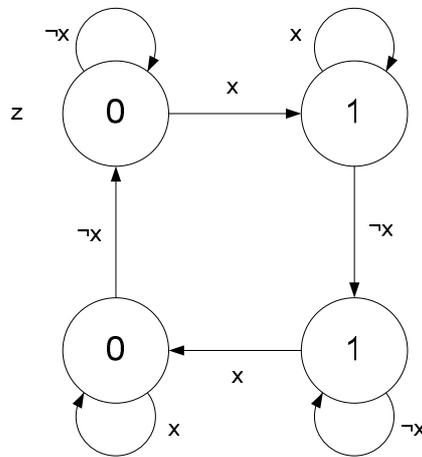


Рис. 1.12. Граф переходов счетного триггера (принудительное кодирование состояний)

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 было событием (единица возникала бы один раз при нажатии кнопки, а ноль – один раз при отпускании), то для управления триггером 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 программное обеспечение 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 удастся далеко не всегда. Поэтому чаще всего в логическом управлении используется альтернативный способ взаимодействия: автомат опрашивает *вычислительное состояние* внешней среды. При отсутствии событий управляющему автомату счетного триггера требуется четыре управляющих состояния, 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). Такое кодирование состояний называется *принудительно-свободным* (один бит кода навязывается значением z , а другой – y – определяется свободно [4]).

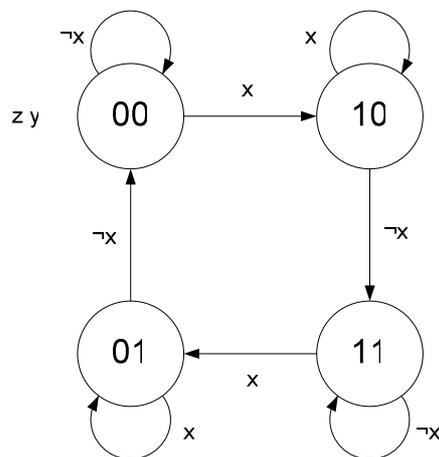


Рис. 1.13. Граф переходов счетного триггера (принудительно-свободное кодирование состояний)

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 , 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*.

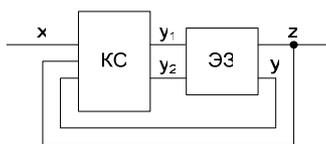


Рис. 1.14. Универсальный автомат без выходного преобразователя (первого рода), управляющий счетным триггером

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 отжата, лампа выключена", "Кнопка нажата, лампа включена", "Кнопка отжата, лампа включена", "Кнопка нажата, лампа выключена"}. Такой способ кодирования состояний называется *свободным*. Он удобен: состояниям можно давать мнемонические строковые имена, однако для формирования выходного воздействия в этом случае требуется выходной преобразователь.

Автомат Мура. Для преобразования свободно выбранных кодов состояний в значения выходных воздействий введем в автомат *выходной преобразователь* – еще одну комбинационную схему КС₂, реализующую функцию выходов автомата (в отличие от первой комбинационной схемы КС₁, 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.

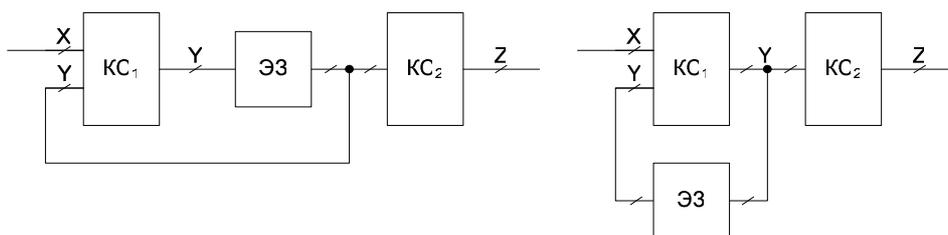


Рис. 1.15. Автоматы Мура: первого рода (слева) и второго рода (справа)

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

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

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

Здесь функция переходов δ и функция выходов φ вычисляются соответственно схемами КС₁ и КС₂.

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. 1.16).

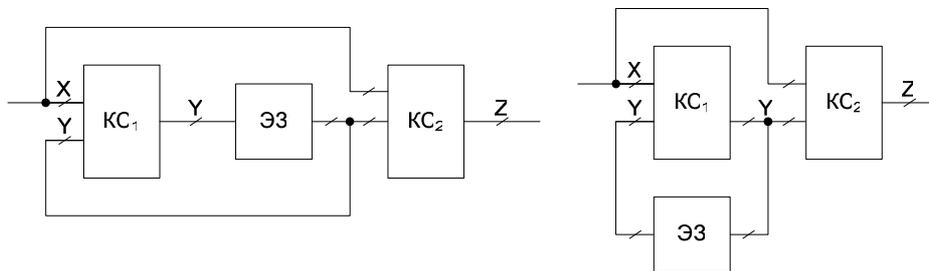


Рис. 1.16. Автоматы Мили: первого рода (слева) и второго рода (справа)

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

$$\begin{aligned}
 \bar{y}' &= \delta(\bar{x}, \bar{y}) & \bar{y}' &= \delta(\bar{x}, \bar{y}) \\
 \bar{z} &= \varphi(\bar{x}, \bar{y}) & \bar{z} &= \varphi(\bar{x}, \bar{y}') & (7) \\
 \bar{y} &= \bar{y}' & \bar{y} &= \bar{y}' & (8)
 \end{aligned}$$

Известно [13], что для любого автомата Мили можно построить эквивалентный ему автомат Мура. Число состояний в таком автомате будет не меньше, чем в исходном.

Смешанные автоматы. Если часть выходных переменных автомата зависит только от состояния, а остальные – также и от входного воздействия, 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. 1.17). Такие автоматы называются смешанными, автоматами Мура-Мили или С-автоматами.

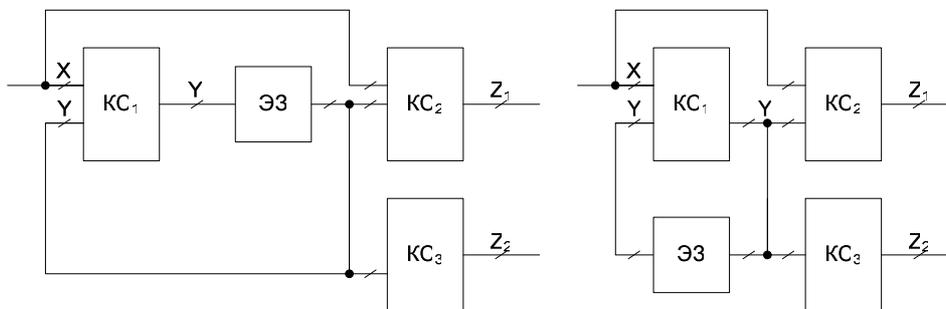


Рис. 1.17. Смешанные автоматы: первого рода (слева) и второго рода (справа)

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, обозначить 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 введения 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: они состоят из устройства управления (представляющего собой ДКА с выходом) и хранилища данных того или иного вида (ленты, магазина). 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. 1.18).

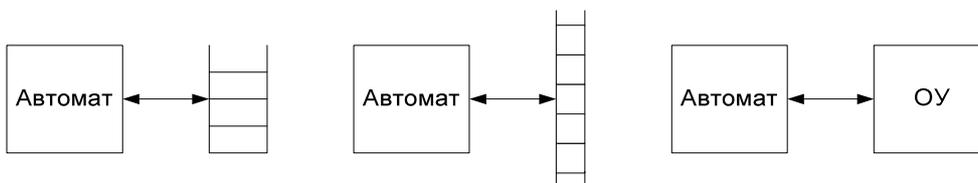


Рис. 1.18. Слева направо: автомат с магазинной памятью, автомат с ленточной памятью (машина Тьюринга), автомат с произвольной памятью (автоматизированный объект)

Вместо ограниченного набора элементарных операций с данными удобно 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 миллиардом состояний, безусловно, является конечным, однако,

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

изобразить его 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 получить информацию о вычислительном состоянии, которую он способен обработать. Небольшое число команд используется для «косвенного» изменения вычислительного состояния.

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 есть 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 стек⁴.

В вопросах, связанных с программированием, важна не только структура АО, но и особенности процесса его работы. Процесс работы всех рассмотренных автоматных моделей разбит 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 таких автоматных моделей, как машина Тьюринга и структурные автоматы, напротив, после окончания текущего такта немедленно начинается следующий. Процесс работы продолжается до тех пор, пока автомат может совершить очередной переход⁵. Такие автоматные модели можно назвать *активными*, поскольку, будучи единожды 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 Такой критерий завершения работы используется при описании машин Тьюринга. При спецификации поведения программных систем в целях наглядности удобно выделять на графах переходов специальные *конечные 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 более характерно событийное взаимодействие: среда сама (асинхронно) сигнализирует о своем изменении, вызывая автоматную модель с некоторым событием.

В программировании активные автоматные модели целесообразно применять при реализации трансформирующих и интерактивных систем, а пассивные – при создании реактивных систем. 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 снова 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 управления 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. 1.19).



Рис. 1.19. Взаимодействие компонентов модели автоматизированного объекта

На 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 [14]. 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 (например, возвращая результат команды, инициированной автоматом).

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 обозначало действие: единица соответствовала включению лампы, а ноль – выключению. В программировании чаще используется другая интерпретация: каждой выходной переменной сопоставляется единственное 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; $\varphi = \varphi' \times \varphi''$ – функция выходов (выходных воздействий), состоящая, в общем случае, из двух компонент: функции выходных воздействий в состояниях $\varphi': Y \rightarrow Z$ и функции выходных воздействий на переходах $\varphi'': X \times Y \rightarrow Z$; $\delta: X \times Y \rightarrow Y$ – функция переходов.

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

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

Детальное графическое представление описанной модели приведено на рис. 1.20.

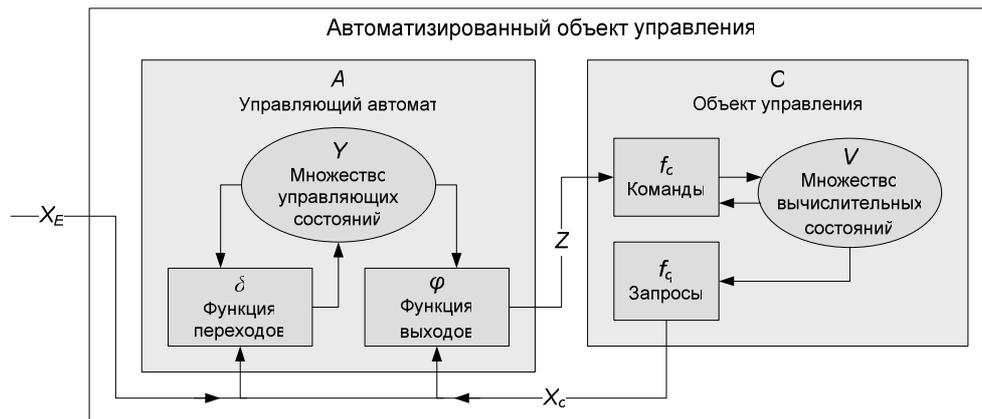


Рис. 1.20. Автоматизированный объект управления

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, чтобы эквивалентность имела место, необходимо потребовать, чтобы запросы и команды объекта управления являлись *вычислимыми* функциями (иначе говоря, чтобы 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. 1.21). Такой автоматизированный объект по вычислительной мощности эквивалентен ДКА: его объект управления уже не является дополнительной памятью, а скорее представляет собой разновидность выходной ленты.

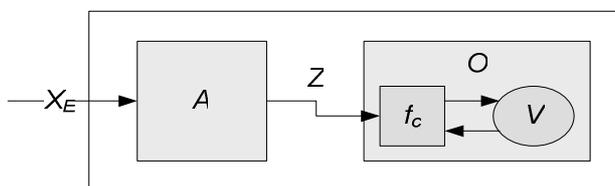


Рис. 1.21. Разомкнутый автоматизированный объект управления

В терминах теории логического управления автоматизированный объект – это, в общем случае, замкнутая система, управляемая автоматом первого рода с выходным преобразователем. Это может быть 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. 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 (преобразование строки «0ⁿ10^m1» в строку «0^{nm}»), содержит уже 12 состояний [5]! Такая модель не только не упрощает реализацию сложного поведения, но и значительно усложняет реализацию простого.

Программирование на машине Тьюринга (или *Тьюрингово программирование*) 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).

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 самобалансирующегося дерева 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 программного обеспечения.

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 программного обеспечения. 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» (рис. 1.22). 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. **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.**



Рис. 1.22. Золотая середина в разделении поведения на логику и семантику

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. Эти два различных варианта подробно обсуждаются в настоящей работе в главах 2 и 3 соответственно. 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 метода автоматного программирования в том виде, каким он был предложен одним из авторов настоящей работы. Автоматное программирование в таком традиционном понимании было названо *программированием с явным выделением состояний*. Более точным названием для этого метода было бы *процедурное программирование с явным выделением состояний*, поскольку описываемый метод 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 немислимо 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. В этой главе графы переходов применяются как наглядные иллюстрации уже в разд. 2.1 при описании основных концепций автоматного проектирования. Подробное описание этой 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 [15]. 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 настоящей работе под процедурным программированием. Легче всего определить эту парадигму от противоположного: это не декларативное, не функциональное и не объектно-ориентированное программирование, причем с практической точки зрения, наиболее важное значение имеет последнее противопоставление. Для последующего

обсуждения интерес представляют две главных 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.

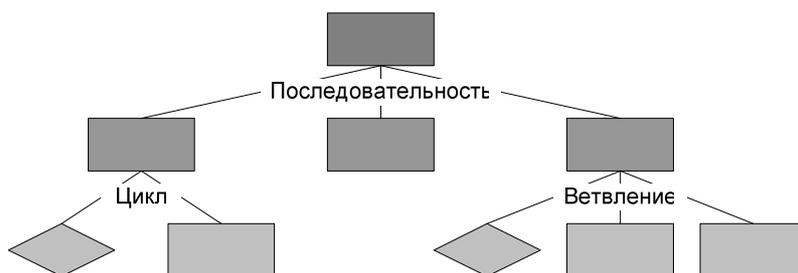


Рис. 2.1. Архитектура системы при проектировании сверху вниз: дерево подпрограмм

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 первый шаг уточнения ответа 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 (рис. 2.2).

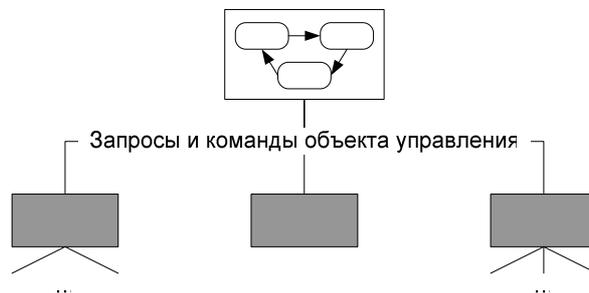


Рис. 2.2. Архитектура системы, управляемой одним автоматом

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 сопоставляются *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 работы упомянутых запросов и команд. Совокупность значений этих переменных составляет *вычислительное состояние* системы.

Для примера рассмотрим, как применить этот метод 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: «1. Будильник выключен», «2. Установка времени будильника», «3. 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.

- В данном случае целесообразно спроектировать событийную систему. Автомат будет обрабатывать три события, соответствующие нажатию трех различных кнопок. На графе переходов (рис. 2.4) для обозначения событий используются названия кнопок. Переходы между состояниями осуществляются по нажатию кнопки «А». При нажатии других кнопок («Н» и «М») переходов между состояниями не происходит, но выполняются соответствующие действия (изменение текущего времени или времени срабатывания будильника). Эти действия обозначим выходными переменными z_1-z_4 .

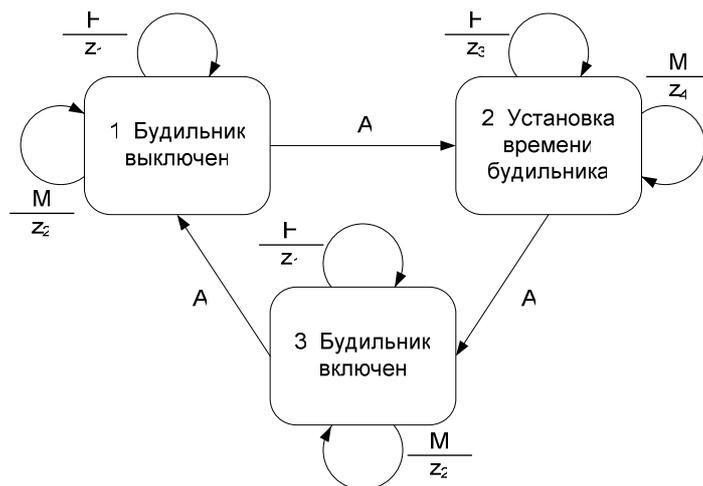


Рис. 2.4. Управляющий автомат эмулятора часов с будильником

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 входные переменные отсутствуют.

3. Выходным переменным автомата, введенным на предыдущем шаге, 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». Перечисленные процедуры являются достаточно простыми 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. Подпрограммы самого низкого уровня абстракции реализуются в терминах переменных, хранящих вычислительное состояние. Реализация автомата становится главной программой⁶. 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 какие-либо действия только при нажатии соответствующих кнопок. 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 основных 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 наиболее всего подвержен изменениям [16]. В результате архитектура 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 вокруг управляющих состояний, является более расширяемой.

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*.

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 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. 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 [4]. В этой области принято описывать взаимодействие с объектом управления в терминах *исполнительных механизмов* (совершающих определенные действия при подачи на них управляющего сигнала) и *сигнализаторов* (подающих сигнал, если объект управления находится в определенном вычислительном состоянии). В терминологии настоящей работы 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, снабженный исполнительными механизмами открытия и закрытия, а также сигнализаторами открытого и закрытого положения. 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, загорается индикатор «Открыт» и управляющий сигнал с клапана снимается;
 - при нажатии кнопки «Закрыть» клапан начинает закрываться;
 - после его закрытия срабатывает сигнализатор закрытого положения, загорается 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 загорается индикатор «Неисправность»;
 - сброс сигнала с индикатора «Неисправность» осуществляется нажатием кнопки «Разблокировка».
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 положения. Этим сигнализаторам поставим в соответствие 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).

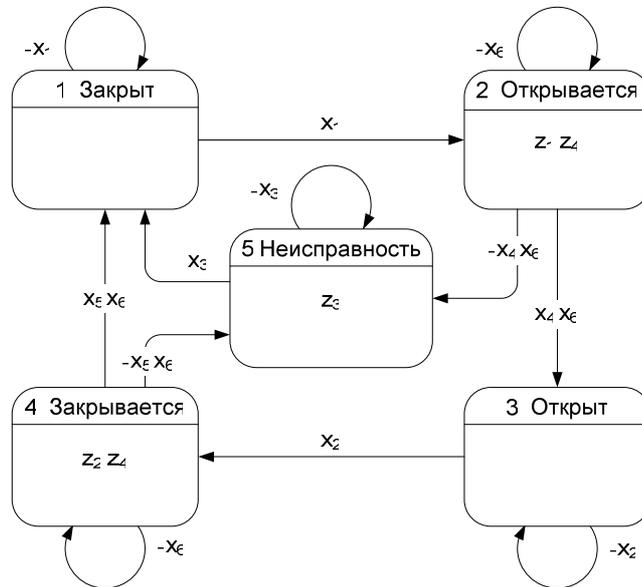


Рис. 2.7. Автомат системы управления клапаном

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 нестабильные 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 отрицании дизъюнкции условий на всех остальных исходящих дугах.

2.1.1. Программные системы, управляемые множеством взаимодействующих автоматов

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

Вспомним формулировку парадигмы автоматного программирования, приведенную в разд. 1.3: парадигма автоматного программирования 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 методов 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, 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 любым из указанных выше способов всегда состоит ровно из *одного* автоматизированного объекта, содержащего один объект управления и один автомат.

Конечно, такое решение применимо только для очень простых и маленьких систем. С ростом размера системы, 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 [17].

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.

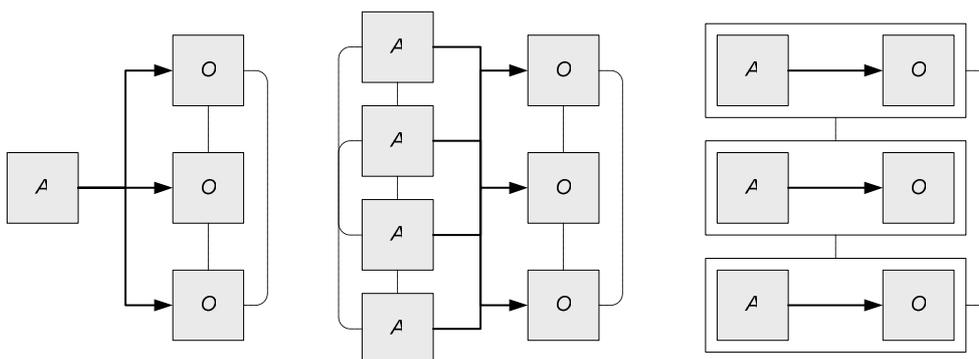


Рис. 2.8. Слева направо: путь к современной парадигме автоматного программирования

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).

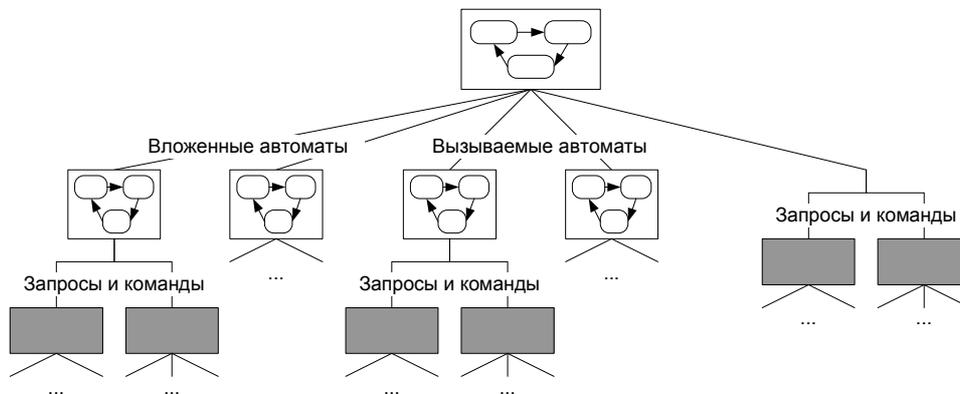


Рис. 2.9. Архитектура системы, управляемой множеством автоматов

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*.

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 объекте 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 использования того или другого 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 все-таки 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. Два автомата, 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 злоупотреблять⁷.

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*. 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 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 следует, так как он требует доступа к внутренней структуре автомата, а это усиливает межмодульную зависимость.

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. За удобство этого механизма приходится платить сложностью его программной реализации.

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

1. Исходные данные – это по прежнему словесное описание поведения системы и спецификация событий и объектов управления.
2. Строится набор наиболее общих, 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, 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 компонент 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 программного обеспечения, 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 действие «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 так далее.

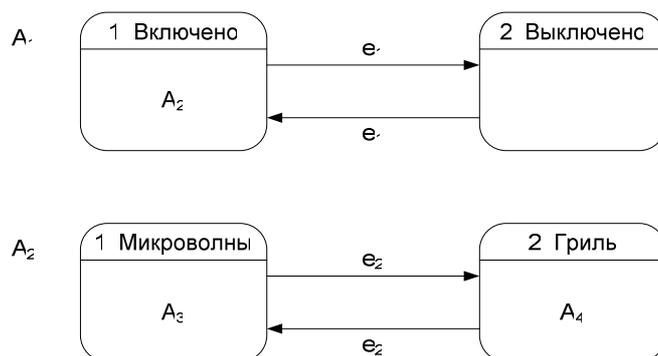


Рис. 2.12. Два верхних уровня абстракции системы управления микроволновой печью

- 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 иерархии автоматов (рис. 2.13).

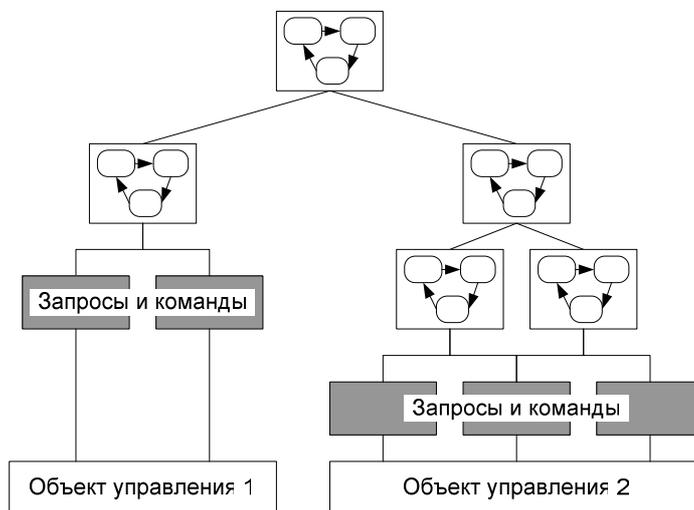


Рис. 2.13. Автоматная декомпозиция по объектам управления

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, в котором они изложены в настоящей работе.

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 объектов⁸.

Сравним два предложенных выше критерия автоматной декомпозиции на примере проектирования системы управления двумя клапанами. Отметим, что эта задача довольно проста, 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 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 положения первого клапана – переменные x_3 и x_4 соответственно, сигнализаторам открытого и закрытого положения 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).

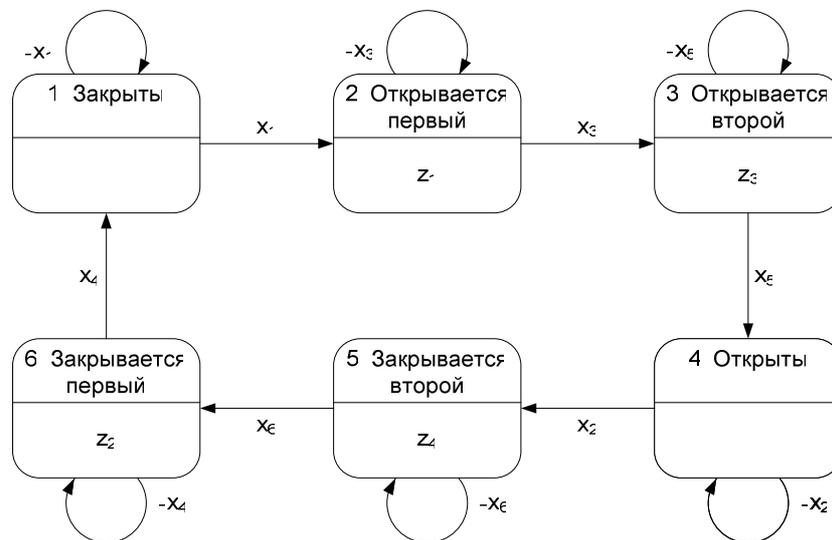


Рис. 2.14. Управление двумя клапанами с помощью одного автомата

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).

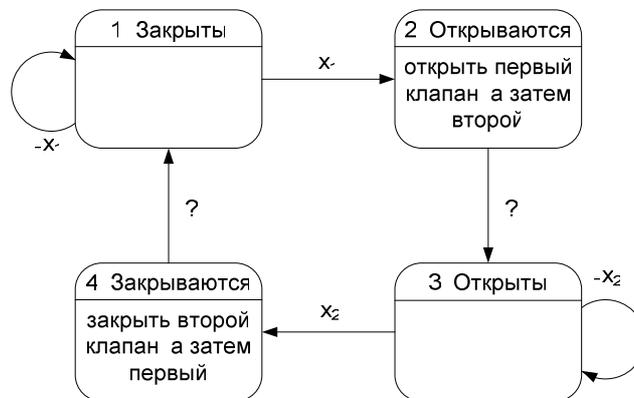


Рис. 2.15. Головной автомат системы управления двумя клапанами (в процессе проектирования)

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, 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» [18] (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.

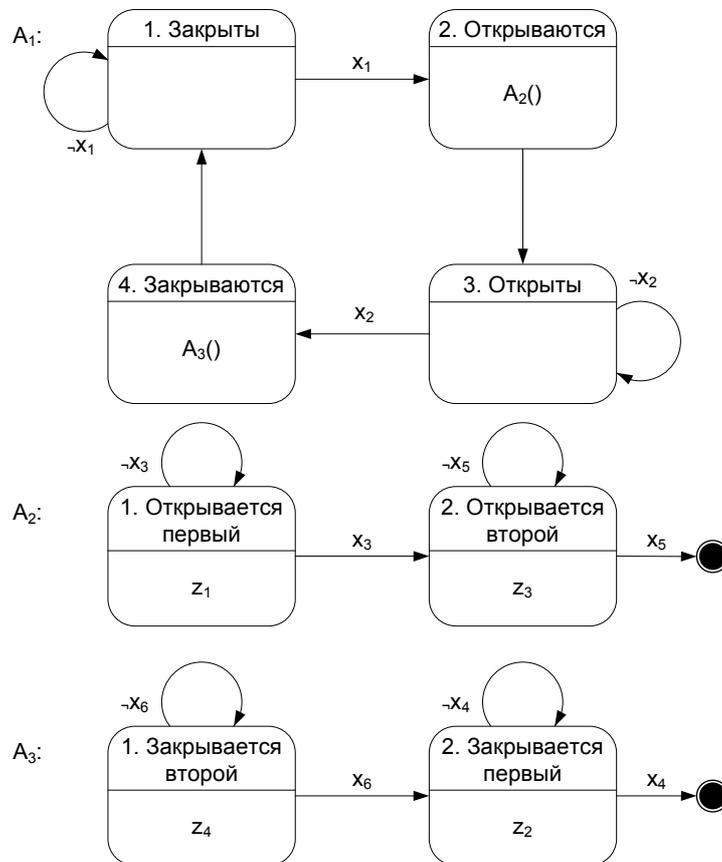


Рис. 2.16. Система управления двумя клапанами: декомпозиция по режимам

Отметим, что в графической нотации автоматного программирования нет общепринятых соглашений о различении вложенных и вызываемых автоматов. В настоящей работе принято соглашение, по которому при обращении к вложенному автомату просто записывается его идентификатор, а при обращении к вызываемому – после идентификатора записываются круглые скобки, в которых при необходимости указывается передаваемое автомату событие. Такая запись аналогична синтаксису вызова функций 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 z_i (рис. 2.17).

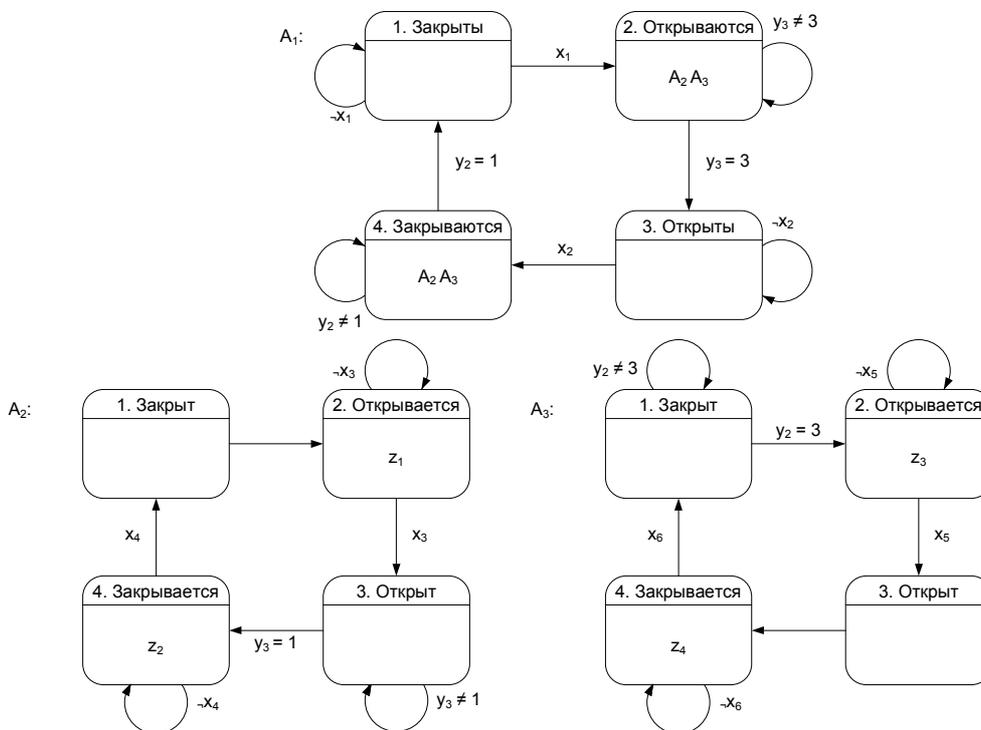


Рис. 2.17. Система управления двумя клапанами: декомпозиция по объектам управления

Система, построенная с помощью декомпозиции по объектам управления, получилась больше и сложнее для понимания. Ее сложность во многом объясняется использованием механизма обмена номерами состояний. Лучшей альтернативой 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 очевидно, 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.

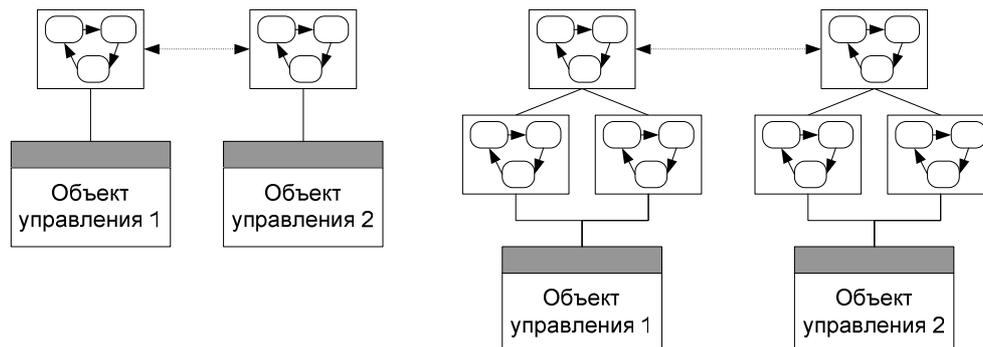


Рис. 2.18. Децентрализованная архитектура системы: параллельно работающие автоматы (слева) и параллельно работающие деревья автоматов (справа)

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.

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.

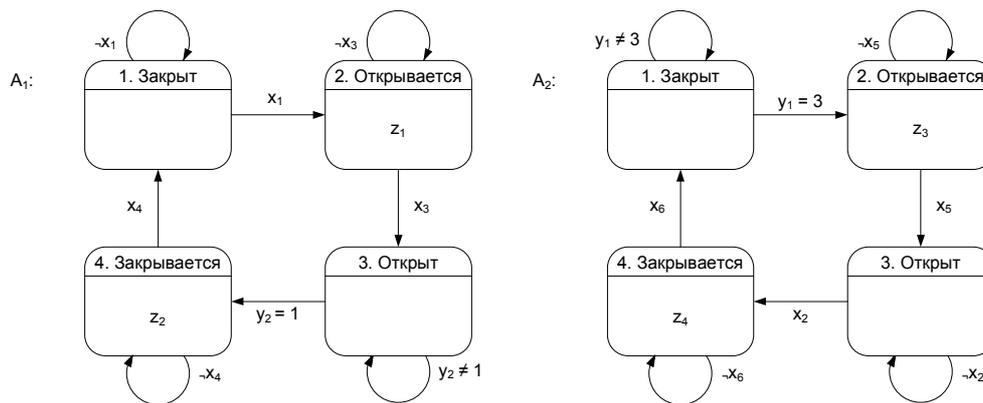


Рис. 2.19. Система управления двумя клапанами: параллельная декомпозиция

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 уже упоминалось и было 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 *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 обладает свойством наглядности. В программировании с явным выделением состояний для описания структуры системы используется графическая нотация *схем связей* автоматов.

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). Таким образом, схема связей служит, в частности, для расшифровки сокращенных идентификаторов событий и переменных, используемых на диаграмме переходов.

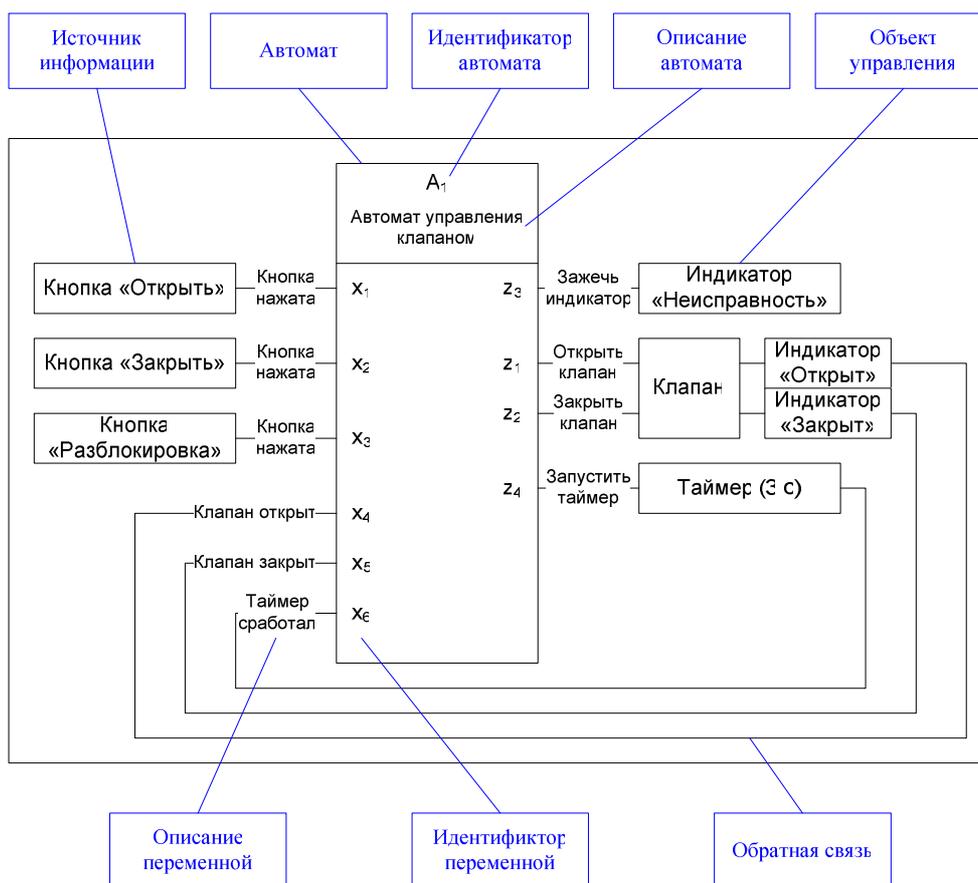


Рис. 2.20. Нотация схем связей автомата

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 эмулятора 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

Обратимся теперь к системе управления двумя клапанами, спроектированной с применением автоматной декомпозиции по объектам управления. В этой системе три управляющих автомата, поэтому изобразим ее структуру с помощью трех отдельных схем связей (рис. 2.22).

ПРИМЕЧАНИЕ

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

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 , поэтому на схеме связей последнего они изображены справа. В данной системе взаимодействие автоматов осуществляется путем обмена номерами состояний, поэтому связи между автоматами системы помечены предикатами вида $y_i = n$ и словесными пояснениями этих предикатов. Такая нотация позволяет сгладить недостатки обмена номерами состояний и является шагом на пути к четко определенному интерфейсу взаимодействия между автоматами. Поскольку каждому предикату сопоставляется отдельная связь и определенный смысл, переход к событийной модели взаимодействия можно обеспечить, просто заменив каждый из предикатов отдельным событием.

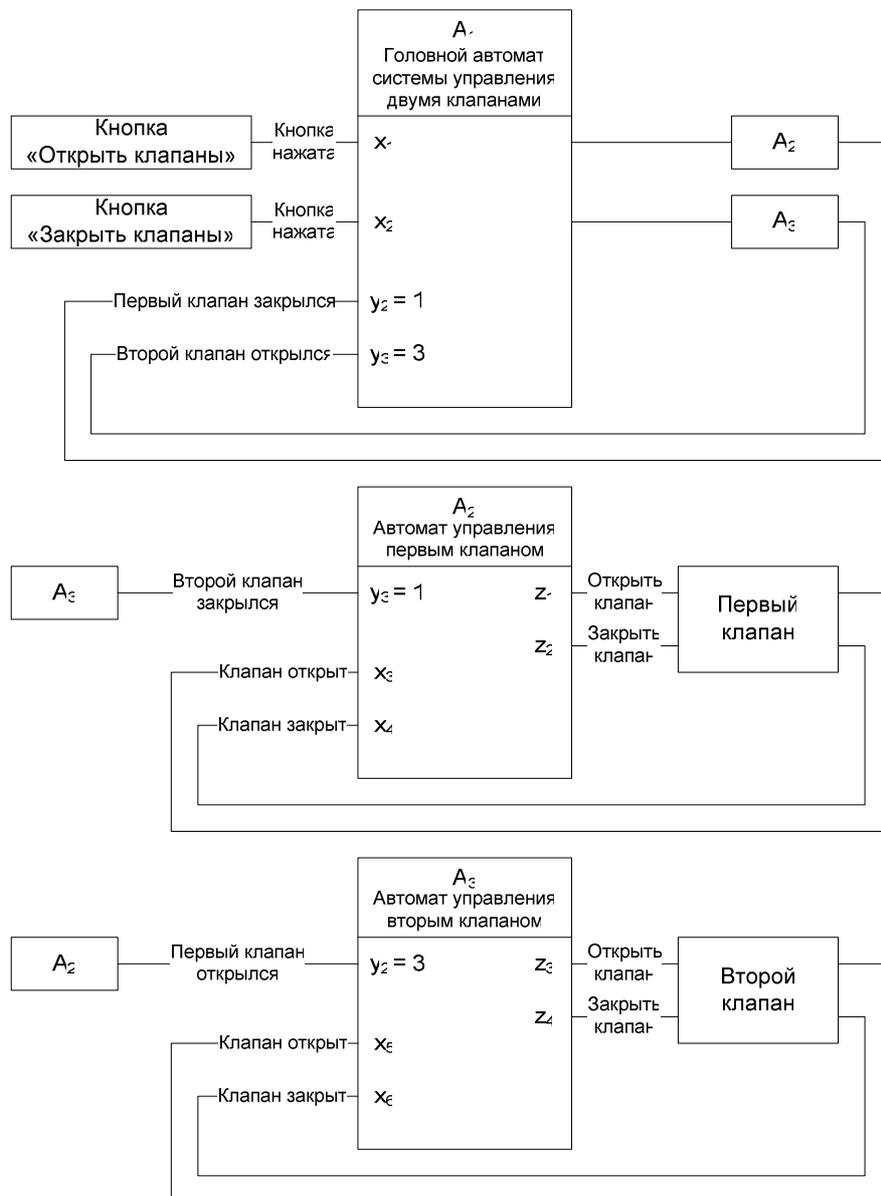


Рис. 2.22. Схемы связей трех автоматов системы управления двумя клапанами (декомпозиция по объектам управления)

Для небольших систем, управляемых множеством автоматов, схем связей вполне достаточно, чтобы наглядно описать взаимосвязи и обмен информацией между автоматами. Для систем *большого* размера со сложными взаимосвязями может быть удобно построить единую *диаграмму взаимодействия*, на которой будут изображены все автоматы системы и отношения вложенности и вызываемости между ними (рис. 2.23). При необходимости на такой диаграмме можно отобразить обмен

номера 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).

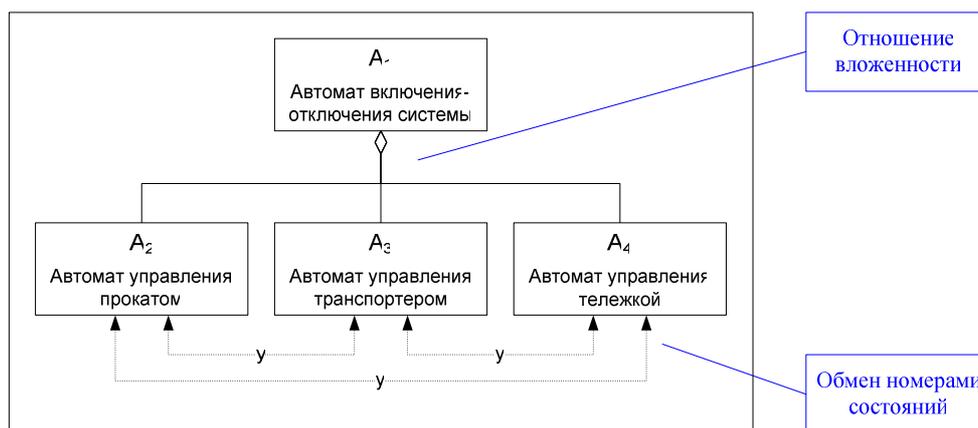


Рис. 2.23. Нотация диаграмм взаимодействия автоматов

2.2.2. Спецификация поведения

Познакомившись с языком спецификации структуры системы, вернемся к языку спецификации ее поведения: уже известным читателю *графам* (или *диаграммам переходов*). Поскольку неформальное знакомство читателя – на примерах – с этой нотацией уже состоялось, в данном разделе опишем графы переходов чуть 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.

Дуги графа переходов помечаются 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.

На диаграммах переходов к переменным, событиям и автоматам системы принято обращаться через их краткие 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 идентификатора 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 назывались буквами «Н», «М» и «А». 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 совершенно одинаковые 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 пунктирной 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 автомата⁹. 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 *большим* 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).

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

Один частный случай использования переходов с приоритетами настолько распространен, что для него существует специальная нотация. 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. На новой дуге следовало бы написать отрицание 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 сделать новый переход безусловным 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.

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

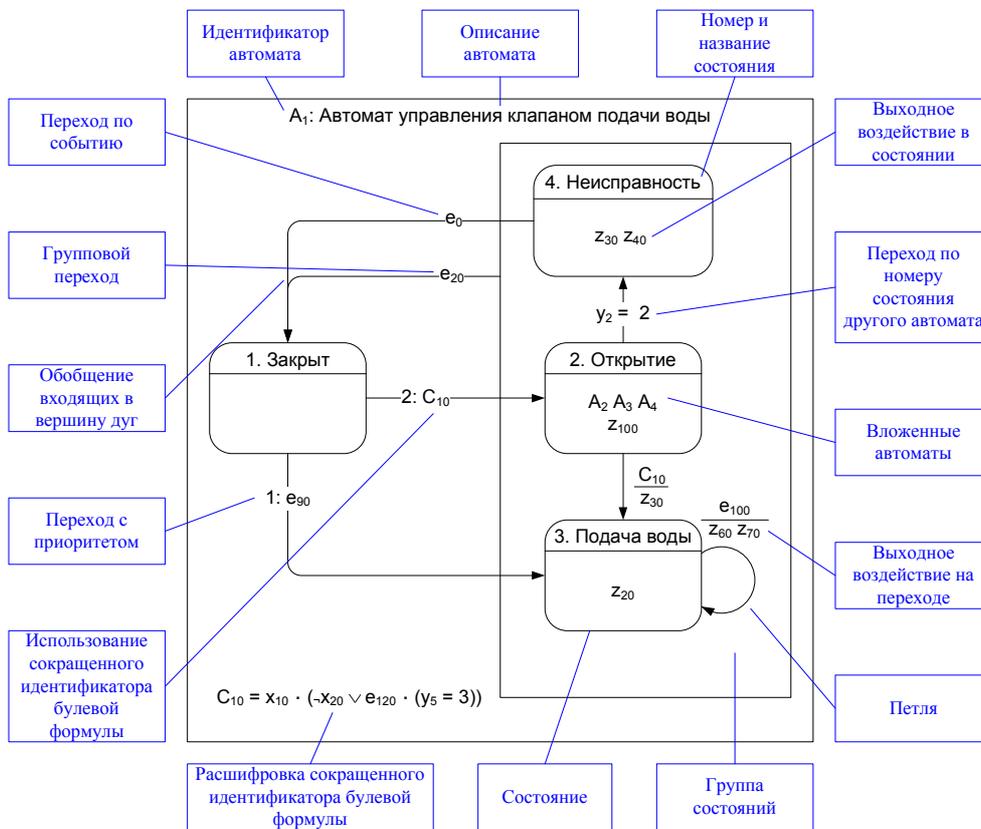


Рис. 2.24. Нотация диаграмм переходов

2.2.3. Использование спецификаций

Теперь, когда читатель полностью 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. Ими можно обмениваться с другими разработчиками, а также непосредственно использовать в качестве входных данных для следующего этапа разработки системы – реализации (рис. 2.25).

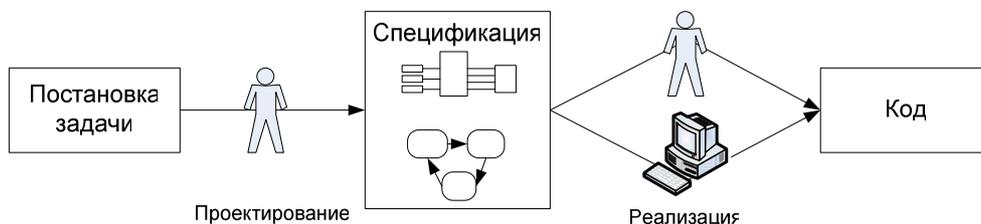


Рис. 2.25. Этапы разработки программной системы со сложным поведением

Имея схему связей и диаграмму переходов, код, реализующий автомат на языке программирования, можно построить с помощью формального преобразования¹¹. Эта задача уже не является творческой. Ее несложно выполнить разработчику, однако гораздо лучше, если код генерируется автоматически. Ряд инструментальных средств, которые обсуждаются далее в настоящей работе, 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.

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.

2.2.4. Сравнение с нотацией *Statechart*

Проведем краткое сравнение нотации графов переходов, применяемых в программировании с явным выделением состояний, и *диаграмм состояний (statecharts)*, предложенных Д. Харелом в рамках метода разработки реактивных систем *StateMate*.

Автор считает уместным сравнение двух методов разработки (и, в частности, двух графических нотаций), поскольку оба метода предназначены для создания систем со сложным поведением, оба основаны на понятии конечного автомата 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 используется в объектно-ориентированном методе разработки *Unified Rational Process* [19].

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

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

системы раннего оповещения. Все примеры настоящего раздела позаимствованы из книги [15].

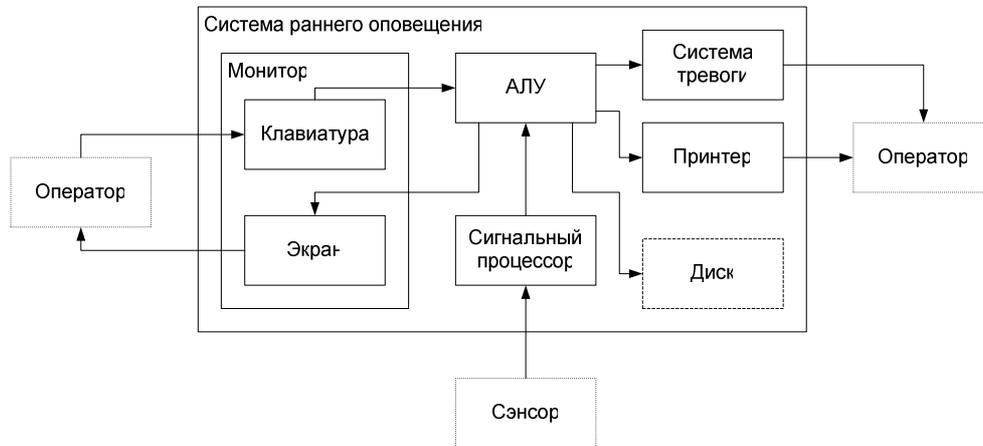


Рис. 2.26. Диаграмма модулей для системы раннего оповещения

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

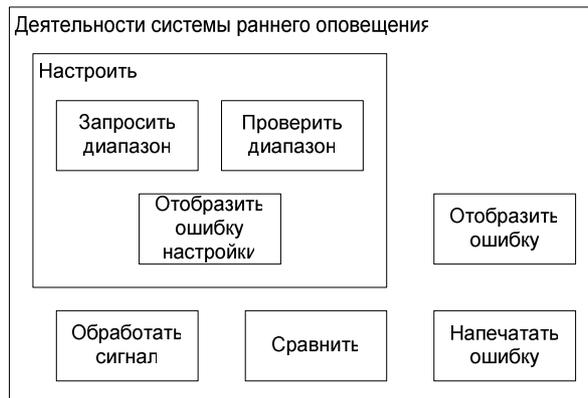


Рис. 2.27. Диаграмма деятельности для системы раннего оповещения

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 [15, 18, 20]. Пример диаграммы состояний приведен на рис. 2.28.

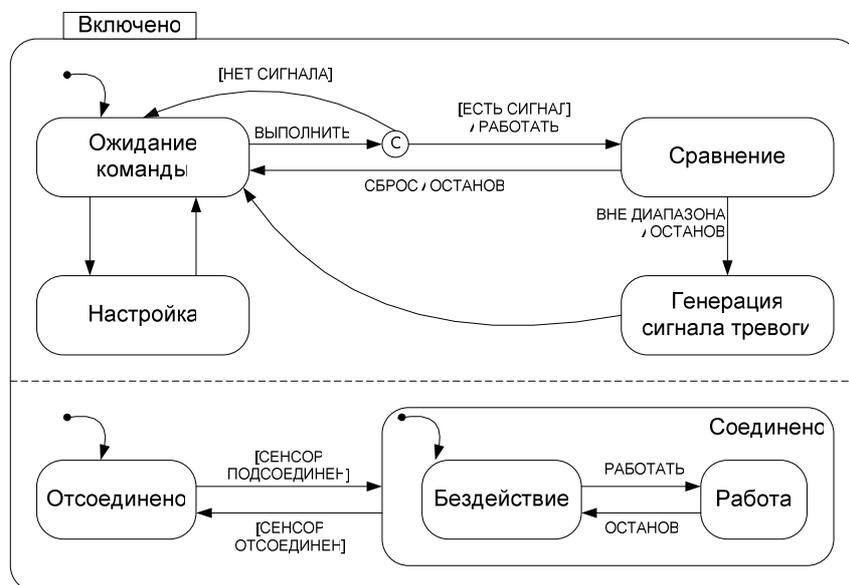


Рис. 2.28. Диаграмма состояний для системы раннего оповещения

В программировании с явным выделением состояний, в отличие от метода *StateMate*, используются всего два основных вида диаграмм: на схемах связей совмещены структурный и функциональный виды системы, диаграммы переходов отображают ее поведенческий вид. 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 событийных 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 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 одновременно более простой и более мощной,

чем язык *statecharts*. Среди ограничений, которые характерны для диаграмм состояний и отсутствуют у диаграмм переходов, отметим следующие:

- возможность проектирования только 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 автору 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. На рис. 2.28 состояния «Бездействие» и «Работа» *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. 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.

С точки зрения синтаксиса, решение, используемое в диаграммах переходов имеет важное преимущество. В отличие от вложенных состояний, вложенные автоматы специфицируются отдельно от объемлющего автомата, 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).

В заключении отметим, что, несмотря на разночтения в нотациях, подход *StateMate* и программирование с явным выделением состояния 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 автора, 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 автором в главе 3.

2.3. Реализация

В данном разделе обсуждаются вопросы программной реализации различных автоматных моделей в частности, и 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 следующем разделе (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 подразумевать, 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 построена схема связей и граф переходов.

2.3.1. Задачи логического управления

В системах управления логика может быть реализована как программно, так и аппаратно. Вопросы аппаратной реализации конечных автоматов достаточно хорошо изучены, известен ряд эффективных методов решения этой задачи [4]. Однако эти методы не относятся напрямую к предмету настоящей работы (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 (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 реализации 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.

- **Изоморфизм¹² программного кода графу переходов автомата.** 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 не требовать творческих усилий. В этом случае его можно поручить машине.
- **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 доступной памяти 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 блок-схемы (рис. 2.29).

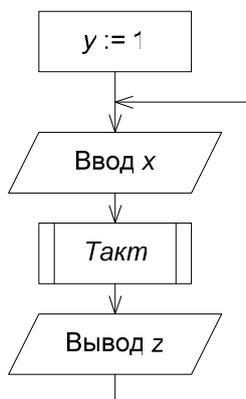


Рис. 2.29. Схема алгоритма, реализующего активный автомат

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.

¹² Изоморфизм здесь понимается в том смысле, что преобразование графа переходов в код и обратно может быть описано формально и выполнено машиной, и кроме того, это преобразование сохраняет поведение (поведение программы, наблюдаемое при ее выполнении, соответствует спецификации, заданной диаграммой переходов).

Реализация блоков ввода и вывода зависит от конкретной системы и не относится напрямую к предмету настоящей работы. 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 блок-схемы так, как это показано на рис. 2.30.

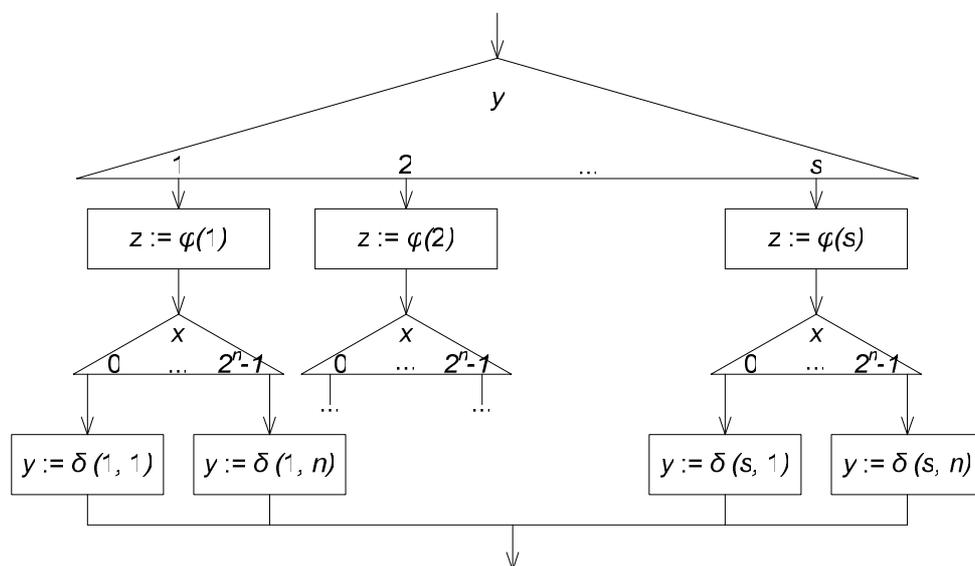


Рис. 2.30. Схема алгоритма, реализующего один такт работы автомата Мура

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 на 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. 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 всего 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. В контексте настоящего раздела динамическая модификация структуры автомата не требуется. Поэтому в целях повышения быстродействия предпочтение следует 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` [21]. 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:
            z1 = <0|1>; ...; zm = <0|1>;
            // Блок формирования выходных воздействий
            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

данный подход в том виде, в котором он был изложен, не решает другую проблему: экспоненциальный рост описания автомата с увеличением числа входных переменных. Действительно, количество меток case во вложенной инструкции выбора, а, следовательно, и объем текста программы, экспоненциально зависит от n (количества 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. Более того, часто 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.

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:
            z1 = <0|1>; ...; zm = <0|1>;
                                // Блок формирования выходных воздействий
            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>:
            ...
    }
}
```

Нетрудно убедиться, что объем текста программы, составленной по такому шаблону, линейно зависит от числа дуг в графе переходов.

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

Но 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 приведенного выше шаблона приведем реализацию автомата системы управления клапаном (листинг 2.3), проектирование которого обсуждалось в разд. 2.1.1.

Листинг 2.3. Реализация системы управления клапаном на языке C

```
void A1 () {
    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;
    }
}
```

Выше обсуждалась реализация систем со сложным поведением, управляемых одним автоматом. Перейдем к вопросам реализации систем, управляемых множеством взаимодействующих автоматов.

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, имеющей структуру, описанную в листинге 2.4.

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).

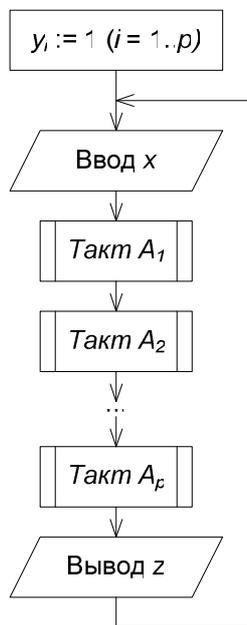


Рис. 2.31. Схема алгоритма, реализующего псевдопараллельную работу p активных автоматов

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 рассмотрим задачи, не относящиеся к области логического управления. Будем рассматривать событийные системы, в которых автоматы, чаще всего пассивны. С внешней средой и между собой они взаимодействуют путем обмена сообщениями. Что касается автоматных моделей, то используются автоматы Мили и смешанные автоматы.

Что изменилось в алгоритмах?

1. Поскольку используются другие автоматные модели, то блок формирования выходных воздействий теперь в другом месте.
2. Поскольку команды объекта управления теперь реализованы программно, как обычные функции, в блоке формирования выходных воздействий теперь не присваивание, а вызов функций, соответствующих тем выходным переменным, значение которых «истина».
3. Поскольку рассматриваются пассивные автоматы, то изменилась и главная программа.

Сначала рассмотрим схему алгоритма, описывающего один такт работы автомата Мили (рис. 2.32). Здесь блок формирования выходного воздействия имеет вид вызова подпрограммы, вместо имени которой указана функция выходов автомата с определенными аргументами. Имеется в виду, что происходит вызов команд объекта управления, а каких именно – определяется значением функции выхода от данных аргументов.

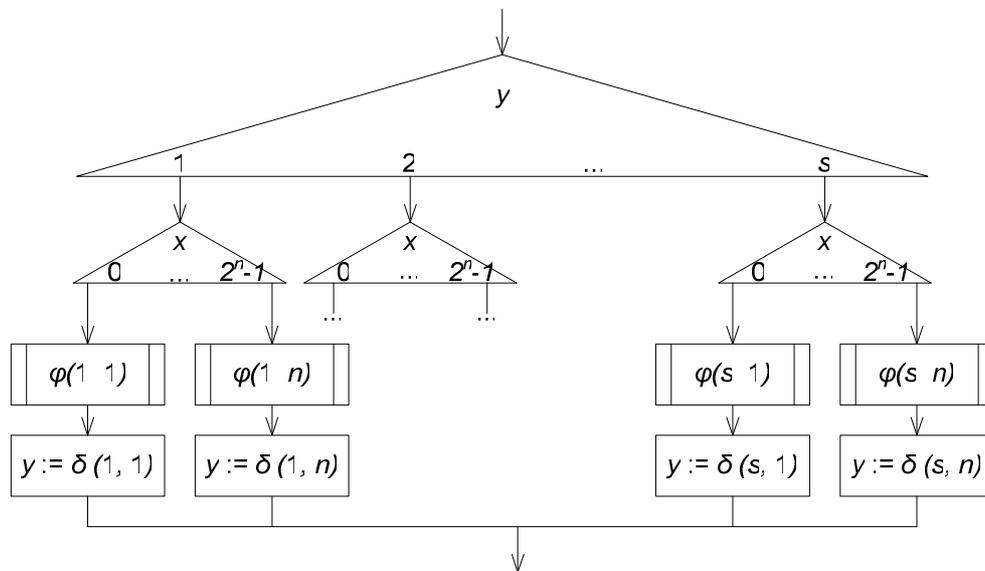


Рис. 2.32. Схема алгоритма, реализующего один такт работы автомата Мили

Объединив этот алгоритм с алгоритмом такта работы автомата Мура, получим алгоритм, описывающий шаг работы смешанного автомата (рис. 2.33).

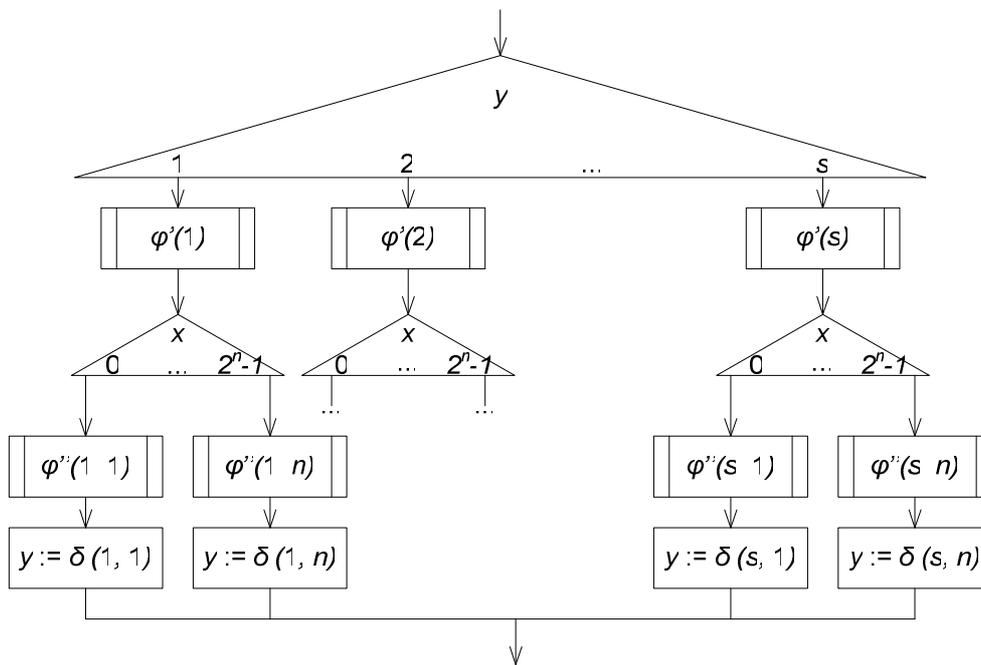


Рис. 2.33. Схема алгоритма, реализующего один такт работы смешанного автомата

Теперь что касается главной программы. Как вы помните, пассивный автомат не работает такт за тактом, а совершает очередной такт работы только по инициативе внешней среды. Поэтому теперь структура главной программы не так однозначна, в ней описано функционирование внешней среды, и в некоторых местах этой программы вызывается функция такта автомата. Например, если программа работает под управлением операционной системы *Windows*, то главная программа – это цикл, в котором достаются события из очереди, а потом вызывается автомат для обработки очередного события. Но, с другой стороны, это только вершина айсберга: ведь операционная система совершает всю работу по приему событий от внешних устройств и передачи их нашей программе. Так что вообще сложно сказать, что тут главная программа.

Как изменились критерии оптимальности реализации по сравнению с областью логического управления? Изоморфизм кода графу переходов по-прежнему на первом месте. Эффективность по времени и памяти здесь не настолько критичны, но всё-таки желательны. Гибкость и адаптируемость решения к новым условиям, напротив, здесь более важны.

Отметим, что в данном разделе обсуждаются вопросы реализации в контексте процедурного программирования. Объектно-ориентированный подход располагает большим арсеналом механизмов для реализации автоматов (часть из них обсуждается в разд. 3.3, а пока что у нас так и остается два способа: таблицы и инструкция выбора. Примеры реализации, как и раньше, будут на языке *C* (для удобства, с некоторыми расширениями), поскольку это самый известный в настоящее время процедурный язык.

В ряде специальных задач нужно модифицировать автомат во время выполнения. В этом случае следует использовать реализацию при помощи таблиц. Но опыт показывает, что такие случаи редки. В сочетании с тем, что, как было показано выше, использование инструкции выбора в сочетании с инструкцией ветвления только упрощает изоморфное преобразование графа переходов в код, всё-таки в большинстве случаев следует выбирать реализацию с помощью инструкции выбора. Ее мы и будем рассматривать.

Вообще, реализация функции такта не сильно изменилась по сравнению с предыдущим разделом. Поскольку был автомат Мура, а стал автомат Мили (или смешанный), то блок формирования выходного воздействия переместился внутрь инструкций ветвления. Кроме того, поскольку команды объекта управления – это теперь не исполнительные механизмы, реализованные аппаратно, а обыкновенные функции, реализованные программно, то вместо присваивания значений выходным переменным выполняются вызовы функций.

Лучше поговорим о взаимодействии этой функции с вызывающим контекстом, а именно о передаче событий. Здесь есть можно придумать три разумных проектных решения.

1. События, как и входные переменные – это глобальные переменные программы (или, в лучшем случае, при декомпозиции по объектам управления, модуля, отвечающего за конкретный объект управления).

Если события исключительны (не могут произойти два сразу), то можно завести одну целочисленную переменную «текущее событие» и сравнивать ее с идентификаторами конкретных событий.

Данное проектное решение допускает реализацию и неисклчительных событий. В этом случае необходимо завести битовый вектор, размер которого равен числу событий. Значение «истина» в некотором разряде будет означать, что событие с номером, равным номеру разряда, произошло. События в таком случае представлены так же, как входные переменные.

Шаблон реализации в этом случае отличается от листинга 2.2 только положением блока формирования выходного воздействия.

2. Событие является аргументом функции такта. Это хорошее проектное решение. Такая архитектура отражает смысл (автомат осуществляет именно обработку этого единственного события) и не дает совершить глупые ошибки (не установить переменную события перед вызовом автомата). События обычно глобальны в рамках, системы, так что, в отличие от входных переменных, они редко могут быть объявлены в том программном модуле, который относится к конкретному объекту управления. Поэтому больше смысла делать их аргументами.

Изменения в шаблоне реализации в этом случае незначительны: меняется только сигнатура функции такта:

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

3. Каждому событию сопоставляется отдельная функция. Это тоже хорошее проектное решение в том случае, когда события исключительны (а так по идее и должно быть). Оно отражает не только исключительность, но и активную роль событий. Оно, как бы, говорит, что вызов автомата инициирован событием, а не просто событие в это время где-то существует.

К тому же, оно лучше всего согласовано со структурой программы, принятой в событийном программировании: программа – это множество обработчиков событий.

Единственный недостаток: оно менее изоморфно схеме алгоритма на рис. 2.33. Там сначала идет дешифратор состояний, а потом дешифратор входных воздействий, а тут, как бы, сначала дешифратор событий, потом состояний, потом остальных компонент входного воздействия (входных переменных). Конечно, усложнение преобразования это плохо, но с другой стороны в большинстве систем вперед передачей события автомату внешняя уже должна была его дешифровать, хотя бы отчасти, чтобы понять, какому автомату на обработку его передать. Если так, то при других проектных решениях идет, как бы, повторная дешифровка события в автомате (а это не только долго, но и некрасиво).

Отметим, на всякий случай, что изоморфизм все-таки никуда не исчезает, он есть и преобразование графа переходов в код, построенный по этому образцу, можно поручить машине.

Шаблон реализации функции такта для этого случая описан в листинге 2.6.

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

```
void e1 () {
    switch (y) {          // Дешифратор состояний
        case 1:
            if (<формула_от входных переменных_1>) {
                // Дешифратор наборов входных переменных
                z<i_1>(); ...; z<i_t>();
                // Блок формирования выходных воздействий
                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>:
            ...
    }
}

```

Рассмотрим все три варианта реализации на примере часов с будильником – сущности со сложным поведением, управляющий автомат для которой был построен в разд. 2.1.1. Напомним, что в этой системе нет входных переменных.

Листинг 2.7 содержит реализацию объекта управления и управляющего автомата часов с будильником с использованием первого из предложенных ранее 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;

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

// Реализация объекта управления
int hrs;
int mins;
int alarmHrs;
int alarmMins;

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

```

```

}

void z2() {
    mins = (mins + 1) % 60;
}

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

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

// 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; }
            break;
        case 2: // Установка времени будильника
            if (e == h) { z3(); }
            else if (e == m) { z4(); }
            else if (e == a) { y = 3; }
            break;
        case 3: // Будильник включен
            if (e == h) { z1(); }
            else if (e == m) { z2(); }
            else if (e == a) { y = 1; }
            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, 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; }

```

```

        break;
    case 2: // Установка времени будильника
        if (e == h) { z3(); }
        else if (e == m) { z4(); }
        else if (e == a) { y = 3; }
        break;
    case 3: // Будильник включен
        if (e == h) { z1(); }
        else if (e == m) { z2(); }
        else if (e == a) { y = 1; }
        break;
    }
}

```

В листинге 2.9 приведена реализация управляющего автомата часов с будильником с помощью третьего шаблона: каждому событию сопоставляется отдельная функция.

Листинг 2.9. Реализация часов с будильником (события передаются в качестве аргументов)

```

// Реализация управляющего автомата
void n () { // Нажатие кнопки «Н»
    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 «М»
    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 «А»
    switch (y) {
        case 1: // Будильник выключен
            y = 2;
            break;
        case 2: // Установка времени будильника
            y = 3;
            break;
        case 3: // Будильник включен
            y = 1;
            break;
    }
}

```

2.3.3. Инструментальные средства

Как неоднократно упоминалось выше, главный критерий оптимальности реализации управляющих автоматов – это изоморфизм текста программы графу переходов. Существование формального преобразования из графа в код позволяет создать программу (генератор кода), которая будет выполнять это 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 [22]. Однако в настоящей работе рассмотрим только те, которые разрабатывались в контексте автоматного программирования и, соответственно, ориентированы на создание программных систем в целом, а не просто программного описания отдельных конечных автоматов.

Исторически первым инструментальным средством был конвертер *Visio2SWITCH* [23], который по графам переходов, представленным в нотации, близкой к изложенной в настоящей работе, и изображенным с помощью пакета *Microsoft Visio*, генерирует код на языке *C*.

Это средство используется в настоящее время при создании программного обеспечения одного класса ответственных систем реального времени.

Развитие это направление исследований получило в работе [24], в которой показано, что аналогичный подход может быть использован для реализации автоматов на любом наперед заданном языке программирования, и было создано инструментальное средство *MetoAuto* для поддержки такого подхода.

В основе этого инструментального средства лежит один из известных подходов к порождающему программированию [25]: 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 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. Когда эта технология разрабатывалась, никто не пытался специально ничего «сочетать», оно как-то само собой получилось.

Почему автоматы так хорошо вписались в процедурный стиль, на каких ролях сочетались парадигмы? Главной здесь была установка «автомат как подпрограмма», которая каким-то образом естественно возникла сама собой. Если так понимать автоматы, то они органично вписываются в процесс проектирования сверху вниз. В этой парадигме модульная структура программной системы строится из подпрограмм. Если автомат – это всего лишь разновидность подпрограммы, то не возникает никакого конфликта парадигм, просто некоторые подпрограммы в системе являются автоматами, а некоторые – нет. Автоматная декомпозиция является всего лишь разновидностью декомпозиции сверху вниз (по определенным критериям).

Но, может быть, так же естественно можно скомбинировать автоматный подход с каким-то еще? Наверное, интересно провести исследования по поводу использования автоматных концепций в контексте функционального и логического программирования, но результаты таких исследований, скорее всего, имели бы исключительно теоретическую ценность. Мы же интересуемся практикой, и поэтому логично рассмотреть сочетание автоматного подхода с объектно-ориентированным программированием.

Уже два десятилетия объектно-ориентированное программирование является наиболее широко используемым стилем программирования в мире. Про этот стиль столько всего написано, поддерживающие его языки настолько разные, что объективного понимания, что такое объектно-ориентированное программирование, можно сказать, не существует. Какие-то черты на очень высоком уровне абстракции, конечно, можно сформулировать путем обобщения, но это будет нечто аморфное и невнятное. Поэтому, по аналогии с предыдущей главой, прежде всего опишем вкратце, что авторы настоящей работы понимают под объектно-ориентированным программированием. Отметим, что мнение авторов нельзя читать наиболее распространенным, скорее, оно совпадает со взглядами, изложенными в книге [16]. 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?», а на вопросе «Кто в системе что-нибудь делает?». Поскольку этот аспект системы является более устойчивым, то и результирующая 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* или АТД).
- *Классы.* Класс – это полная или частичная реализация АТД. Классы – единственный вид модулей в объектно-ориентированной системе. Классы общаются 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* (аналог *операций* АТД), а также семантические свойства: *предусловия* запросов и команд (аналог *предусловий операций* АТД), *постусловия* запросов и команд и *инварианты класса* (аналог аксиом АТД).

В отличие от модулей в структурной парадигме (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, существующей в объектно-ориентированной программной системе во время выполнения. Вызов компонента на некотором объекте – это основной вид операции в объектно-ориентированных вычислениях.

- *Наследование* – это механизм, позволяющий создавать повторно используемые программные модули, «не обладая бесконечной мудростью». С точки зрения класса как модуля, наследование – это механизм расширения, позволяющий определять новые классы частично повторно используя определения уже существующих. Этот механизм позволяет, во-первых, построить модифицированный класс без необходимости внесения изменений в существующий класс (а следовательно, и в описания его клиентов), а во-вторых, позволяет 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 подтипа этого типа. Полиморфизм позволяет клиентам одинаково работать с множеством «похожих» классов, и поэтому, является мощным механизмом повторного использования клиентского кода.

Исследования, в которых вместе фигурируют автоматы и объектно-ориентированное программирование, ведутся давно, в большом количестве и многими авторами. Есть

множество паттернов для объектно-ориентированной реализации автоматов [26], есть диаграммы состояний в UML [18]. Результаты нельзя назвать удовлетворительными.

Авторы и собиратели паттернов заявляют, что концепция сложного поведения несовместима с объектной парадигмой, поэтому паттерны получаются такие сложные и их так много, а выбрать подходящий для конкретного случая так сложно.

В языке UML, во-первых, кроме диаграмм состояний для описания поведения предлагается использовать и другие типы диаграмм и не говорится когда и какие диаграммы следует применять. Во-вторых, в рамках унифицированного процесса разработки программ [19] не было предложено подходов для совместного использования диаграмм, описывающих статические и динамические свойства программ. Наконец, в третьих, диаграммы для описания поведения в основном использовались как язык общения между участниками разработки и для документирования программ, в то время как подобающая им роль – точная спецификация, входные данные для генерации текста программы.

Вместе с развитием процедурного программирования с явным выделением состояний, по инициативе одного из авторов настоящей работы было положено начало другой разновидности автоматного программирования, названной *объектно-ориентированным программированием с явным выделением состояний*. 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 паттернов. Однако в вопросах технологии не все сразу оказалось так однозначно: это направление претерпело существенную эволюцию, которая не остановилась и до сих пор. Суть этой эволюции и ее нынешнее состояние будет отражено в данной главе.

3.1. Проектирование

Первый вопрос, который приходит в голову: а зачем вообще рассматривать объектно-ориентированный вариант автоматного программирования? Вроде, и в процедурном варианте все было неплохо. Во-первых, там был недостаточный уровень модульности: вычислительные состояния всех объектов управления и управляющие состояния всех автоматов были глобальными. Во-вторых, это только в обучающих целях рассматриваются задачи на пустом месте. На практике реализация сущностей со сложным поведением должна интегрироваться в существующие системы, а огромная часть этих систем написано в объектно-ориентированном стиле.

Если в объектно-ориентированном программировании все так хорошо, как сказано во введении к данной главе, зачем запихивать в него автоматы? При ближайшем рассмотрении оказывается, что все хорошо, пока нет сложного поведения. Как обсуждалось выше, сущности со сложным поведением, на самом деле, встречаются очень часто. В последнее время часто приходится слышать от гуру индустрии разработки программного обеспечения, что в связи с возрастающей сложностью программ, необходимо использовать автоматы, но как-то их использование не вяжется с концепциями объектно-ориентированного программирования.

Почему не вяжется? Давайте разберемся. В объектной технологии сущности моделируются при помощи классов. Входное воздействие для класса – это вызов компонента. Если пока оставить в стороне полиморфизм, класс всегда реагирует на вызов определенного входного воздействия выполнением одного и того же набора

действий, закодированных в теле соответствующего компонента. А это, как вы, наверное, помните, определение простого поведения. Итак:

Класс является моделью сущности с простым поведением.

Причина плохой пригодности объектной технологии «в сыром виде» для описания сущностей со сложным поведением найдена. И рецепт, на первый взгляд, прост: если требуется описать сущность со сложным поведением, необходимо привести в традиционную объектную технологию автоматный подход. Но как именно, в каких пропорциях и на каких ролях его привести – об этом спорит множество людей на протяжении пары десятков лет.

В основном все обсуждения автоматов в объектно-ориентированном контексте сконцентрированы вокруг вопросы: «Как с помощью классов реализовать автоматы?». Авторам паттернов это простительно, они ничего не знают об автоматном программировании и о том, что автоматные модели необходимо применять в программных системах систематически, для описания сложного поведения, а не от случая к случаю, при разработке компиляторов и еще для решения нескольких случайных задач.

Как ни удивительно, в первых работах по объектно-ориентированному программированию с явным выделением состояний, акцент был точно такой же. В них предполагалось, что проектирование системы должно происходить точно также, как при процедурном программировании с явным выделением состояний. В результате получается множество взаимодействующих автоматов и подпрограмм, которые они вызывают. И только после этого, уже на стадии реализации, в разработку вовлекался объектно-ориентированный подход, в результате чего предпринимались попытки «втиснуть» эту совершенно чуждую объектной технологии древовидную, основанную на функциях, архитектуру в множество классов.

То, что структура системы при этом получается некрасивая – это полбеды. 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. Но при описанном подходе к сочетанию автоматной и объектно-ориентированной парадигм, вместо добавления одного класса придется перепроектировать всю систему с нуля в автоматном стиле! Даже не стоит говорить, что это смешно.

Корни этой проблемы в том, что автомат продолжает восприниматься как подпрограмма. Даже если его пытаются упаковать в класс и инкапсулируют переменную, хранящую управляющее состояние (что похвально), все равно это всего лишь автоматная функция (а иногда и цикл, который раньше был главной программой), упакованная в класс.

ПРИМЕЧАНИЕ

В объектно-ориентированном программировании уже была похожая история с попыткой построить объектно-ориентированную модель сущности, просто обернув подпрограмму в класс. Так называемые, *активные объекты* – одна из моделей

параллельных вычислений, которая основывается на отождествлении объекта и процесса. У класса, порождающего активные объекты, есть главная подпрограмма, выполнение которой отождествляется с работой процесса.

Модель активного объекта применяется в некоторых языках программирования, однако, ее использование все время наталкивается на препятствия (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.

Следующей в объектно-ориентированном программировании с явным выделением состояний появилась концепция «автоматы и объекты управления как классы» (рис. 3.1).

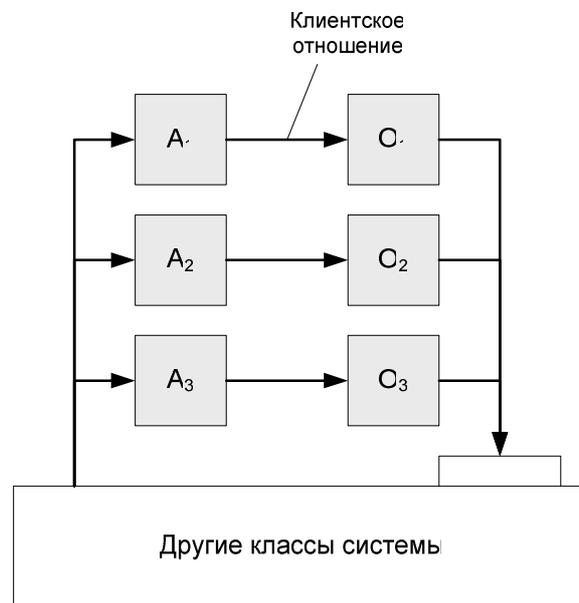


Рис. 3.1. Архитектура системы при подходе «автоматы и объекты управления как классы»

Объекты управления представить в виде классов не сложно, у них все для этого есть: запросы, команды и вычислительное состояние в качестве реализации. Наконец-то восстановилась справедливость: если раньше объекты управления четко отделялись друг от друга только тогда, когда они были реализованы аппаратно, то теперь усилия проектировщиков не пропадают, выделенные объекты управления превращаются в классы и каждый запрос и команда имеют доступ только к своей части вычислительного состояния.

А как сделать классы из автоматов? Из активных никак. Активные автоматы по своей природе являются программами, а не классами. Зато из пассивных можно. Сервисы, которые они предоставляют – это события, которые они обрабатывают. Тела компонентов представляют собой реализацию функции переходов и выходов. При

этом классы автоматов являются клиентами классов, реализующих объекты управления, и вызывают их запросы и команды, когда нужно.

Такая архитектура уже почти идеальна, но кое-что все-таки смущает. В замечании в разд. 2.1.1 было отмечено, что путь к современной парадигме автоматного программирования можно сравнить с движением человечества к современной форме семейно-брачных отношений. Но при подходе «автоматы и объекты управления как классы» получается возврат к полигамии, ведь ничего не запрещает спроектировать систему не так, как показано на рис. 3.1, а так, как показано на рис. 3.2.

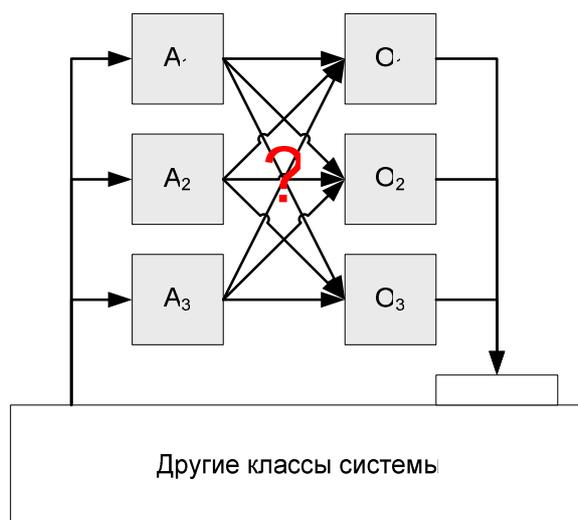


Рис. 3.2. Возможный вариант архитектуры системы при подходе «автоматы и объекты управления как классы»

В объектно-ориентированной программной системе классы должны описывать самостоятельные абстракции. Являются ли автомат и объект управления двумя самостоятельными абстракциями. Может ли автомат управления клапаном в другой системе начать управлять кофеваркой? Нет. Самостоятельной абстракцией является не автомат и не объект управления, а сущность со сложным поведением целиком.

Чтобы реализация сущности со сложным поведением органично вписывалась в объектно-ориентированную систему, модель такой сущности должна быть согласована с объектно-ориентированными концепциями, то есть, как и модель любой сущности, она должна быть классом.

Моделью сущности со сложным поведением должна быть специальная разновидность класса.

Эта модель, так же, как и обыкновенный класс, должна предоставлять клиентам набор сервисов, соответствующих, как и в предыдущем подходе, событиям, обрабатываемым автоматом. Однако внутри, модель сущности со сложным поведением будет отличаться от обыкновенного класса. При вызове компонента выбор выполняемого действия (тела компонента) будет осуществляться в зависимости от управляющего состояния.

В разд. 1.4.3 была построена математическая модель сущности со сложным поведением – автоматизированный объект управления. Автоматизированный объект управления для сущности со сложным поведением, практически то же, что АД для сущности с простым поведением (за тем исключением, что в автоматизированном объекте отсутствует в явном виде описание семантики объекта управления: аксиом и условий запросов и команд; возможно, этот недочет в будущем будет исправлен). Автоматизированный объект управления по сути является разновидностью абстрактного типа данных: это АД со специфическим внутренним устройством, содержащий автомат. Поэтому раз АД в программировании реализуется с помощью класса, то и для автоматизированного объекта управления тоже естественно стать классом. Так 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. 3.3).

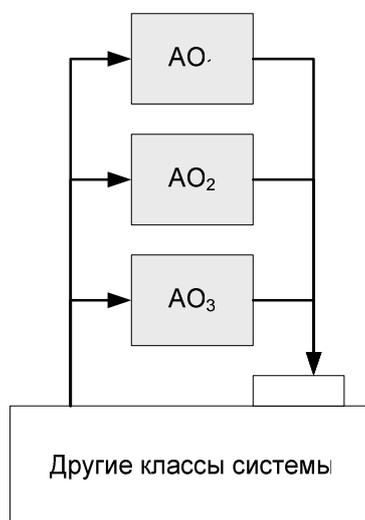


Рис. 3.3. Архитектура системы при подходе «автоматизированные объекты управления как классы»

ПРИМЕЧАНИЕ

Здесь возникает путаница с терминами: объектами называются экземпляры класса, а автоматизированный объект управления – это скорее класс, чем объект. Но такая уж система терминов сложилась в 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 используются только для описания логики сущностей со сложным поведением, а сущности с простым поведением, присутствующие в той же системе, описываются обыкновенными классами.

Отметим, что мы говорим об автоматизированных классах как о концептуально правильной модели для сущности со сложным поведением, но о внутреннем устройстве этого класса пока ничего не утверждалось. В частности, не утверждается, что объекты управления не должны быть классами. Напротив, объект управления, в большинстве случаев, это самостоятельная абстракция, и очень хорошо, если он будет классом. Смысл в том, чтобы для другим сущностям системы, которые хотят получить сервисы от сущности со сложным поведением, не приходилось разбираться с каким-то автоматами, о которых они и знать не хотят. Они должны иметь дело с сущность со сложным поведением как с единым классом. Одно из хороших проектных решений для автоматизированного класса продемонстрировано на рис. 3.4.

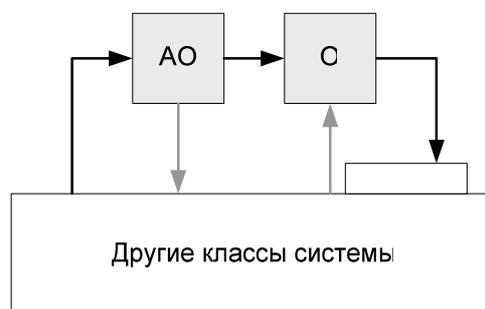


Рис. 3.4. Автоматизированный объект управления и объект управления как классы

Здесь и объект управления и автоматизированный объект управления представлены классами, между которыми существует клиентское отношение. Это корректно, так как оба они являются самостоятельными, хорошо определенными абстракциями. При этом классы, взаимодействующие с сущностью 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.

У тех, кто хорошо знаком с программированием с явным выделением состояний и с автоматной декомпозицией, может возникнуть вопрос: а всегда ли это возможно? Что если после выделения сущностей со сложным поведением получилось так, что автомат, управляющий каким-то объектом управления слишком сложен? Не требуется ли в этом случае осуществить автоматную декомпозицию, а ведь тогда получается, что красивый лозунг «одному объекту управления – один автомат» не работает, объектом будет, как и раньше, управлять целая иерархия автоматов.

Нет, есть лучшее решение. Основным видом декомпозиции в объектной технологии является объектная декомпозиция, и только ее мы и будем применять. С связи с этим обычно возникают возражения двух типов. К первому типу относятся заявления, что объект управления может быть спроектирован раньше (реализован аппаратно, или программно, но другими людьми) и в этом случае нет возможности провести его декомпозицию. Это неправда, если на уровне архитектуры для нас объект составной, то так и следует его воспринимать, даже если на уровне реализации он неделим. Например, кто-то уже спроектировал низкоуровневый (без логики) эмулятор микроволновой печи как один класс, у которого есть, в том числе, компоненты «Открыть замок дверцы» и «Запустить таймер». Это не мешает нам при разработке

системы управления печью считать, что у нас есть отдельно замок дверцы и отдельно таймер, и разработать по автомату для управления каждым из этих объектов. Потом, на этапе реализации, придется учесть, что эти две абстракции были совмещены в одном классе (например, можно ввести адаптеры). Эта проблема – следствие выбора не совсем подходящих компонент для повторного использования. Такое часто бывает в объектно-ориентированном программировании, если повторно используемые компоненты были спроектированы неидеально. Автоматы тут ни причем.

Ко второму типу относятся такие заявления: а что если действительно очень сложная логика, причем все управляющие состояния действительно взаимодействуют со всеми компонентами объекта управления, и поэтому нельзя разбить автоматизированный объект управления на несколько независимых. Такое случается редко, но и эту проблему можно решить, не прибегая к изменению модели сущности со сложным поведением и не нарушая объектно-ориентированных концепций.

В разд. 1.4.3 упоминалось, что автоматное программирование поддерживает концепцию выделения уровней абстракции, разрешая объекту управления самому быть автоматизированным. В процедурном программировании с явным выделением состояний эта конструкция реализовывалась с помощью иерархической автоматной декомпозиции, но довольно криво. Там вложенные и вызываемые автоматы служили, как бы, объектами управления для автомата более высокого уровня. При этом не было единообразия: автоматы все-таки не то же самое, что объекты управления, поэтому взаимодействовать с ними приходилось по-другому и это вело к усложнению модели.

При объектно-ориентированном подходе концепция вложенных автоматизированных объектов управления реализуется непосредственно. Ведь автоматизированный класс снаружи выглядит как обычный, интерфейс у него такой же. Значит ничего не мешает классу объекта управления быть автоматизированным. Так получается почти тот же эффект, как с вложенными автоматами, но без всяких изменений модели (используется одно единственное стандартное клиентское отношение) и гораздо более модульно.

3.2. Спецификация

В объектно-ориентированном мире придумано много методов спецификации. Существуют известные языки моделирования, поддерживаемые графическими нотациями, такие как UML и BON [27]. С другой стороны, есть чисто текстовые нотации: специализированные языки, например Larch [28], и нотации, встроенные в общецелевые языки программирования, например Eiffel [16]. Что из этого может помочь при спецификации автоматизированных классов, что не следует использовать, а что абсолютно ортогонально? Попробуем разобраться.

В процедурном программировании с явным выделением состояний всего два вида спецификаций (структуры и поведения) позволяли достаточно полно описать систему. Для спецификации структуры использовались схемы связей и диаграммы взаимодействия автоматов, а для спецификации поведения – графы переходов. По предыдущему разделу мы уже поняли, что при объектно-ориентированном подходе следует учитывать, что система чаще всего не разрабатывается с нуля, а сущности со сложным поведением внедряются в существующий объектно-ориентированный

контекст. Поэтому не только модель, но и язык ее спецификации должен быть согласован с тем языком, который уже применяется для спецификации системы.

Что же предлагают нам существующие языки? Почти все обладают развитыми средствами для спецификации статической модульной структуры: классов и взаимоотношений между ними. В UML также есть средства для моделирования использования и поведения. С другой стороны, языки формальной спецификации в дополнение к структуре обычно предлагают средства для декларативного описания поведения (например, в форме *контрактов*). В UML, вообще-то, это тоже есть (так называемый, *object constraint language*), но он не очень широко используется. Что из этого нам пригодится?

Моделирование использования нас не будет интересовать: оно представляет систему так, как она видится пользователям, а как стало явно из предыдущего раздела, снаружи сложное поведение вообще не заметно. Поэтому этот вид спецификации ортогонален нашим интересам: если вы его обычно используете, то используйте дальше без изменений, а если нет, то нет, от сложного поведения хуже не станет.

Спецификацию структуры прямо или косвенно используют все, потому что на объектно-ориентированном языке программирования невозможно написать код, не задав при этом классы и отношения между ними. Другое дело, выделяется ли проектирование и построение модели в сознании разработчика в отдельный этап, или происходит походя.

Должна ли быть эта спецификация текстовой или графической – здесь это в большей степени дело вкуса, чем при спецификации сложного поведения. Классы и отношения между ними возможно держать в голове, правда, только в голове их непосредственного разработчика и только на протяжении нескольких месяцев. Конечно, графическое представление статической структуры (во всех известных авторам языках оно называется *диаграммой классов*, хоть и выглядит по-разному) полезно. С другой стороны, существующие на сегодняшний день инструментальные средства для визуального (графического) проектирования, в основном, имеют настолько отвратительный интерфейс, что процесс проектирования при этом неоправданно затягивается. Альтернативный подход, который наиболее симпатичен авторам, это доступность текстового и графического представления одновременно и их автоматическая синхронизация, так что, вне зависимости от того, предпочитает ли разработчик визуальное проектирование или текстовое, он всегда имеет в распоряжении как диаграмму классов, так и программный текст.

Отличается ли спецификация статической структуры системы с автоматизированными классами и без них? И да, и нет. Внешне автоматизированный класс не отличается от обычного, поэтому описание его отношений с клиентами ортогонально предмету разговора: если система описывалась диаграммой классов, то автоматизированный класс в нее отлично впишется и не будет выделяться.

О чем можно поговорить отдельно, так это о спецификации взаимодействия автоматизированного класса с классом объекта управления (если объект управления выделен в отдельный класс, что, как упоминалось выше, является хорошим образцом проектирования). Богатые языки спецификации, например UML, позволяют превратить этот кусок диаграммы классов, фактически, в схему связей (рис. 3.5).

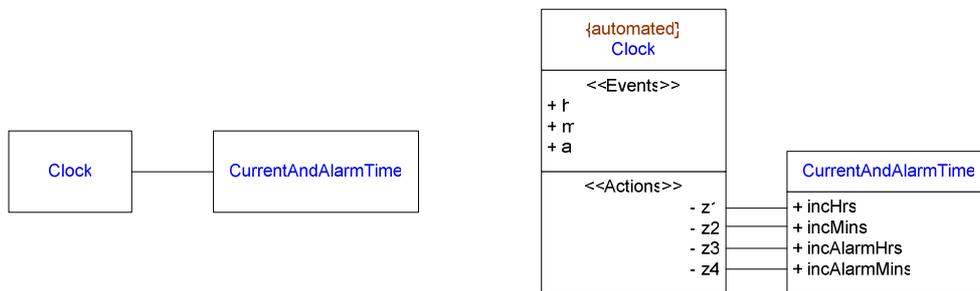


Рис. 3.5. Спецификация взаимодействия часов с будильником и их объекта управления на диаграмме классов: кратко (слева) и по образцу схемы связей (справа)

Здесь с помощью ассоциаций мы связали краткие идентификаторы выходных переменных и команды объекта управления. Напомним, что использование кратких идентификаторов для компонентов входных и выходных воздействий удобно на диаграмме переходов. Не следует воспринимать эту диаграмму буквально, то есть, что для каждой команды объекта управления будет создана дублирующая команда в автоматизированном классе. Эти ассоциации могут восприниматься, например, так: при генерации кода по диаграмме классов и описанию сложного поведения (его мы обсудим ниже) вместо идентификаторов выходных переменных будут подставлены имена команд объекта управления. Или, если язык программирования это позволяет, автоматизированный класс можно сделать не клиентом, а наследником класса объекта управления (разумеется, должно использоваться наследование реализации, или, как его еще называют закрытое наследование), и переименовать его команды.

Итак, схеме связей нашлось место и в объектно-ориентированном контексте. Что же касается диаграммы взаимодействия автоматов, то она, конечно, больше не существует, зато на совершенно стандартной диаграмме классов отлично видно взаимодействие автоматизированных объектов управления.

Вид спецификации, который относится к теме обсуждения самым непосредственным образом – это спецификация поведения. Здесь очень важно различать *декларативную* и *императивную* спецификацию поведения. Декларативная спецификация поведения описывает его, как бы, снаружи, так, как его видят клиенты (если это поведение отдельного компонента, то имеются в виду клиенты компонента), описывая некоторые его свойства. Примером декларативной спецификации служат контракты.

Императивная спецификация поведения – это описание изнутри, с точки зрения поставщика (самой сущности). Примером императивного описания служит код подпрограммы на языке программирования. Однако, это не то описание, которое хорошо подходит для сложного поведения (как мы уже убедились, оно не наглядно). Для сущностей со сложным поведением императивное описание поведение должно быть в форме графов переходов или диаграмм состояний (или подобной нотации).

Основное требование очень просто и логичное: диаграмма должна описывать логику одной сущности со сложным поведением (одного автоматизированного класса). Удивительно, что в языке UML (по крайней мере, в первой его версии) такой рекомендации для объектно-ориентированных систем дано не было, несмотря на то, что язык хорошо приспособлен для объектно-ориентированного моделирования и в нем есть диаграммы состояний. В результате, диаграммы состояний использовались

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 только в качестве документации, а не как полноценные спецификации, по которым можно генерировать код.

Ирония судьбы в том, что при соблюдении правила «одна сущность – один автомат», различия между диаграммами состояний, графами переходов и другими похожими нотациями практически не имеют значения. Так как автоматная декомпозиция отменяется (здесь все автоматы простые), то от всех них остается простая базовая нотация, от которой они все и произошли. Нет вложенных и вызываемых автоматов, вложенные состояния, конечно, могут использоваться, но только для красоты, а не как необходимый механизм.

Вопросы декларативного описания сложного поведения гораздо меньше изучены. Об этом мало кто задумывается: те, кто много знает про контракты, обычно игнорируют проблему сложного поведения, а те, кто занимается сложным поведением, пренебрежительно относятся к контрактам.

Поскольку мы претендуем на то, что автоматизированный класс снаружи выглядит, как обычный, то и контракты у него должны быть такие же, как и у обычного (ведь это описание с точки зрения клиента). Но составить содержательный контракт для сущности со сложным поведением, да так, чтобы клиенты не заподозрили, что поведением сложное, может быть трудно.

Понятно, что контракты компонентов сущности со сложным поведением должны отражать реакцию сущности на определенное входное воздействие в различных состояниях. Например, если рассмотреть часы с будильником, какое может быть постуловие у компонента *h*? Может быть такое: «Если часы в первом или в третьем состоянии, то увеличилось число текущих часов, а если во втором – то часов будильника». Такой контракт несет много информации, но зато раскрывает управляющие состояния. Для внутреннего использования он подойдет, но для клиентов класса, скорее всего, будет *чрезмерной спецификацией (overspecification)*. Можно предложить такой контракт: «Увеличилось или число текущих часов, или число часов будильника», или такой: «Число минут не изменилось, а число текущих часов и число часов будильника не могли оба увеличиться». Будет ли такая спецификация в самый раз? Или это будет *недостаточная спецификация (underspecification)*? Это сложный вопрос, который, надо отметить, часто возникает и без всякого сложного поведения.

В любом случае, получается, что поведения в каждом конкретном управляющем состоянии – это частный случай общего, сложного поведения, а значит контракт на поведение сущности в каждом состоянии должен быть сильнее, чем общий контракт на сложное поведение. А это то же требование, как для отношения подтипизации. Получается, что сущность, пребывающая в конкретном управляющем состоянии является подтипом сущности со сложным поведением самой по себе. Отсюда следуют два интересных вывода. Во-первых, о том, что сложное поведение – это особый вид полиморфизма. По форме этот вид ничем не отличается от сущности отложенного (абстрактного) типа, которая во время выполнения хранит объект, порожденный одним из потомков этого типа. Во-вторых, из такого понимания

сложного поведения напрямую следует образец проектирования *State* [29], 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.

Вопрос о том, есть ли противоречия в модели автоматизированного класса с концепциями проектирования по контракту, и особенно с механизмом наследования, до конца еще не изучен. Ряд авторов занимался вопросами наследования автоматов [30, 31], поскольку проблема повторного использования описания логики поведения очень актуальна. По мнению авторов настоящей работы, говорить о наследовании автоматов не совсем корректно, так как автомат не является самостоятельной абстракцией. Напротив, очень плодотворным полем для исследования является наследование автоматизированных классов.

В заключении обсуждения спецификации отметим, что спецификацию структуры в виде диаграммы классов (такой как на рис. 3.5) и спецификацию поведения в виде графа переходов (рис. 2.4) или диаграммы состояний, можно формально преобразовать в описание автоматизированного класса и «скелет» описания класса объекта управления на объектно-ориентированном языке программирования. После этого останется только добавить описание вычислительного состояния (если оно не было отражено в диаграмме) и реализацию компонентов объекта управления, чтобы получить полноценный код.

3.3. Реализация

Как неоднократно упоминалось выше, лучший подход к реализации сущностей со сложным поведением – автоматизированный. Однако, не для всех языков программирования и не для всех случаев жизни есть подходящие инструментальные средства. Поэтому в разд. 3.3.1 будут рассмотрены примеры шаблонов реализации автоматизированных классов. В разд. 3.3.2 будет рассмотрено наиболее мощное из существующих на сегодняшний день инструментальных средств, поддерживающих автоматное программирование – *UniMod*.

3.3.1. Реализация вручную

Выше упоминалось, что существует огромное количество образцов проектирования, предназначенных для объектно-ориентированной реализации конечных автоматов в частности и сущностей со сложным поведением целиком. Изучить большинство из этих образцов можно, обратившись к работе [26]. Мы же рассмотрим только два шаблона реализации на примере автоматизированного класса `Clock`, описывающего часы с будильником.

Наш первый пример показывает, как можно превратить реализацию сущности со сложным поведением из процедурной в объектно-ориентированную, сделав минимум изменений. Часы с будильником так же, как и их объект управления, превратятся в классы, однако реализация логики останется такой же как в примере из разд. 2.3.2, с использованием инструкции выбора.

В листинге 3.1 приведена реализация объекта управления, а в листинге 3.2 – реализация самого автоматизированного класса на языке *C++*.

Листинг 3.1. Реализация объекта управления часов с будильником

```
// Реализация объекта управления
class CurrentAndAlarmTime {
public:
    void incHrs() {
        hrs = (hrs + 1) % 24;
    }

    void incMins() {
        mins = (mins + 1) % 60;
    }

    void incAlarmHrs() {
        alarmHrs = (alarmHrs + 1) % 24;
    }

    void incAlarmMins() {
        alarmMins = (alarmMins + 1) % 60;
    }

private:
    int hrs;
    int mins;
    int alarmHrs;
    int alarmMins;
}
```

Листинг 3.2. Реализация часов с будильником с помощью инструкции выбора

```
// Реализация автоматизированного класса
class Clock {
public:
    // инициализация внутренней переменной стартовым состоянием
    Clock() : y (alarmOff) {}

    void H () {
        switch (y) {
            case alarmOff:
                co.incHrs();
                break;
            case alarmSetting:
                co.incAlarmHrs();
                break;
            case alarmOn:
                co.incHrs();
                break;
        }
    }
}
```

```

    }

    void M () {
        switch (y) {
            case alarmOff:
                co.incMins();
                break;
            case alarmSetting:
                co.incAlarmMins();
                break;
            case alarmOn:
                co.incMins();
                break;
        }
    }

    void A () {
        switch (y) {
            case alarmOff:
                y = alarmSetting;
                break;
            case alarmSetting:
                y = alarmOn;
                break;
            case alarmOn:
                y = alarmOff;
                break;
        }
    }
private:
    enum State {alarmOff, alarmSetting, alarmOn};

    State y;
    CurrentAndAlarmTime co;
}

```

Наш второй пример использует в качестве шаблона реализации образец проектирования *State*. Как было упомянуто выше, в этом шаблоне вариация поведения в зависимости от состояния реализуется за счет классического полиморфизма. 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 Clock {
public:

```

```

Clock (CurrentAndAlarmTime* controlObj) : y (new AlarmOff(controlObj))
{}

void H () {
    State* newState = y->H();
    delete y;
    y = newState;
}

void M () {
    State* newState = y->M();
    delete y;
    y = newState;
}

void A () {
    State* newState = y->A();
    delete y;
    y = newState;
}
private:
    State* y;
}

// Абстрактный класс состояния
class State {
public:
    virtual State* H() = 0;
    virtual State* M() = 0;
    virtual State* A() = 0;
private:
    CurrentAndAlarmTime* co;
}

// Конкретное состояние «Будильник выключен»
class AlarmOff : public State {
public:
    AlarmOff (CurrentAndAlarmTime* controlObj) : co(controlObj) {}

    virtual State* H() {
        co -> incHrs ();
        return new AlarmOff(co);
    }

    virtual State* M() {
        co -> incMins ();
        return new AlarmOff(co);
    }
}

```

```

    }

    virtual State* A() {
        return new AlarmSetting(co);
    }
}

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

// Конкретное состояние «Будильник включен»
class AlarmOn : public State {
    ... // Аналогично AlarmOff
}

```

3.3.2. Инструментальное средство UniMod

Переходя к инструментальному средству 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 решение задачи об автоматизации построения объектно-ориентированных программных систем со сложным поведением в целом в открытых источниках не излагалось.

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

Предлагаемый метод моделирования системы состоит в следующем:

- ❑ на основе анализа предметной области разрабатывается концептуальная модель системы, определяющая сущности и отношения между ними;
- ❑ 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 — они выполняют действия по команде от автомата. Объекты управления также формируют значения входных переменных для автомата. Автомат активируется источниками событий и на основании значений входных переменных и текущего состояния воздействует на объекты управления, переходя в новое состояние;
- ❑ используя нотацию диаграммы классов, строится схема связей автомата, задающая 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, события которым они поставляют. Автоматы связываются с объектами, которыми они управляют;

- ❑ каждый объект управления содержит два типа методов, реализующих входные переменные (x_j) и выходные воздействия (z_k);
- ❑ для каждого автомата с помощью нотации диаграммы состояний строится граф переходов, в котором дуги могут быть помечены событием (e_i), булевой формулой из входных переменных и формируемыми на переходах выходными воздействиями. В вершинах могут указываться выходные воздействия и имена вложенных автоматов. Каждый автомат имеет одно начальное и произвольное количество конечных состояний;
- ❑ состояния на графе переходов могут быть простыми и сложными. Если в состоянии вложено другое состояние, то оно называется сложным. В противном случае состояние простое. Основной особенностью сложных состояний является то, что наличие дуги, исходящей из такого состояния, заменяет однотипные дуги из каждого вложенного состояния;
- ❑ каждая входная переменная и каждое выходное воздействие являются методами соответствующего объекта управления, которые реализуются вручную на целевом языке программирования;
- ❑ использование символьных обозначений в графах переходов позволяет весьма компактно описывать сложное поведение проектируемых систем. Смысл таких символов задает схема связей. При наведении курсора на соответствующий символ на графе переходов во всплывающей подсказке отображается его текстовое описание.

В терминах предыдущих разделов, предлагаемый метод использует концепцию «автоматы и объекты управления как классы». Скорее всего, такое решение было принято по той причине, что во время создания проекта *UniMod*, более современной концепции «автоматизированные объекты управления как классы» еще не существовало. Кроме того, в предлагаемом методе используется не только объектная, но и автоматная декомпозиция (причины все те же: отсутствие четкого выделения сущностей со сложным поведением).

На рис. 3.6 приведена схема связей автомата, а на рис. 3.7 – его граф переходов, построенные в *UML*-нотации описанным выше способом.

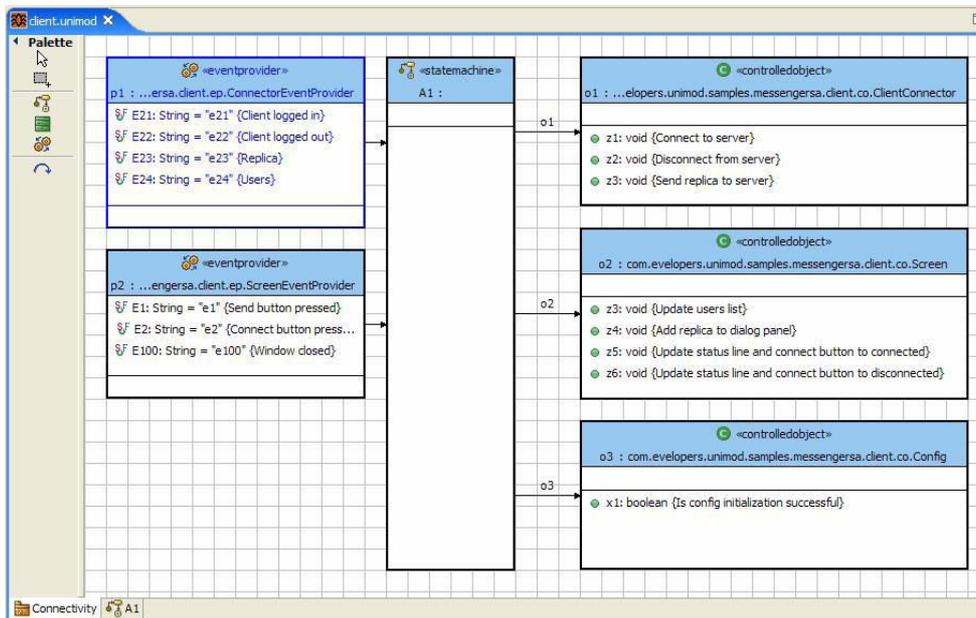


Рис. 3.6. Пример схемы связей автомата

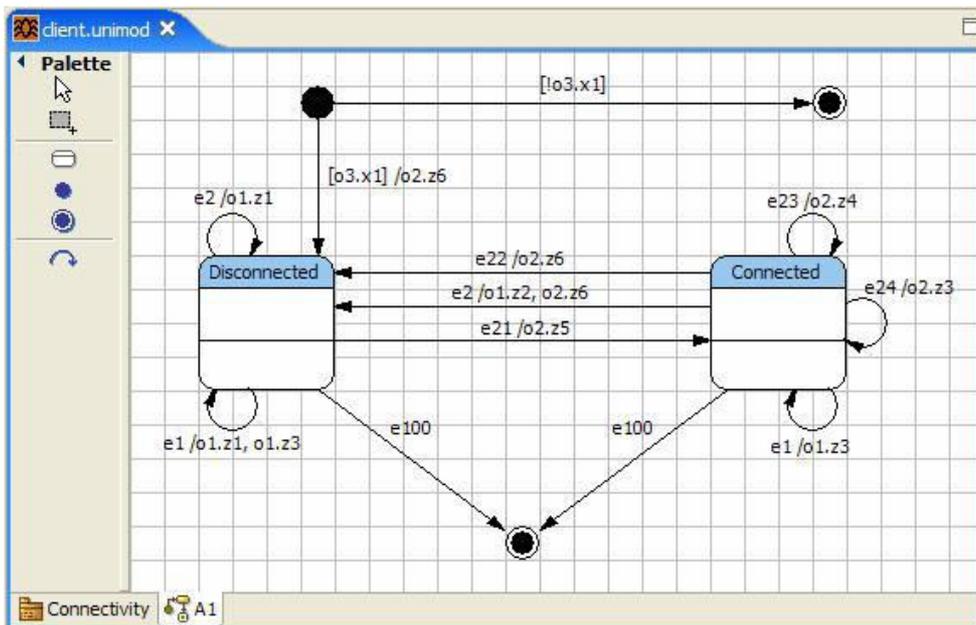


Рис. 3.7. Пример графа переходов автомата

Зададим операционную семантику для модели системы:

- 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 конечных;
- ❑ 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 выходные воздействия, записанные на дуге, и переходит в новое состояние. В нем автомат выполняет выходные воздействия, а также запускает вложенные автоматы;
- ❑ если переход не найден, то автомат продолжает поиск перехода у состояния, в которое вложено текущее состояние;
- ❑ при переходе в конечное состояние автомат останавливает все источники событий. После этого работа системы завершается.

Описав исполняемый графический язык на основе *UML*-нотации и его операционную семантику, перейдем к описанию процесса создания инструмента моделирования, который будет поддерживать активный процесс разработки программ на его основе.

Пакет *UniMod* обеспечивает разработку программных систем со сложным поведением в рамках парадигмы объектно-ориентированного программирования с явным выделением состояний. Он позволяет создавать и редактировать *UML*-диаграммы классов и состояний, которые соответствуют схеме связей и графу переходов, и поддерживает два типа реализации — на основе интерпретации и компиляции. В первом случае имеется возможность:

- ❑ преобразовывать диаграммы в формат *XML*;
- ❑ исполнять полученное *XML*-описание с помощью интерпретатора, созданного на основе набора разработанных базовых классов. Эти классы реализуют, например, такие функции как обработка событий, сохранение текущего состояния, протоколирование.

Во втором случае диаграммы непосредственно преобразуются в код на целевом языке программирования, который впоследствии компилируется и запускается.

Проектирование программ с использованием пакета *UniMod* предполагает следующий подход: логика приложения описывается структурным конечным автоматом, заданным в виде набора указанных выше диаграмм, построенных с использованием *UML*-нотации. Источники событий и объекты управления задаются кодом на целевом языке программирования.

При использовании языка *Java* источникам событий и объектам управления соответствуют классы. Для этого языка в пакете *UniMod* реализован интерпретатор *XML*-описаний структурных конечных автоматов, которые пакет строит на основе указанных диаграмм. При запуске программы интерпретатор загружает в оперативную память *XML*-описание и создает экземпляры источников событий и объектов управления. В процессе указанные источники формируют события и направляют интерпретатору, который обрабатывает их в соответствии с логикой, описываемой автоматом. При этом он вызывает методы объектов управления, реализующие входные переменные и выходные воздействия.

Литература

1. *Ненейвода Н.И.* Стили и Методы программирования. М.: Интернет-университет информационных технологий, 2005. 316 с.
2. *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.
3. *Ахо А., Сети Р., Ульман Д.* Компиляторы. Принципы, технологии, инструменты. М.: Вильямс, 2002. 767 с.
4. *Шальто А. А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб: Наука, 1998. 628 с.
5. *Хопкрофт Д., Мотвани Р., Ульман Д.* Введение в теорию автоматов, языков и вычислений. М.: Вильямс, 2002. 528 с.
6. *Ахо А., Ульман Дж.* Теория синтаксического анализа, перевода и компиляции. М.: Мир, 1978.
7. *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами на С++. М.: Бином; СПб.: Невский диалект, 1998.
8. *Лавров С. С.* Программирование. Математические основы, средства, теория. СПб.: БХВ-Петербург, 2001.
9. *Перлис А.* Синтез алгоритмических систем / Лекции лауреатов премии Тьюринга за первые двадцать лет 1966–1985. М.: Мир, 1993.
10. *Дейкстра Э.* Взаимодействие последовательных процессов / Языки программирования. М.: Мир, 1972.
11. *Колесов Ю. Б., Сенеченков Ю. Б.* Моделирование систем. Динамические и гибридные системы. БХВ-Петербург, 2006.
12. *Глушков В. М.* Синтез цифровых автоматов. М.: Физматлит, 1962.
13. *Фридман А., Меннон П.* Теория и проектирование переключательных схем. — М.: Мир, 1978.
14. *Заде Л., Дезоер Ч.* Теория линейных систем. Метод пространства состояний. М.: Наука, 1970.
15. *Harel D., Polity M.* Modeling Reactive Systems with Statechart. The Statechart Approach. NY: 258 p. McGraw-Hill, 1998.
16. *Мейер Б.* Объектно-ориентированное конструирование программных систем. М.: Интернет-университет информационных технологий, 2005. 1198 с.
17. *Википедия.* <http://ru.wikipedia.org>.
18. *Буч Г., Рамбо Дж., Якобсон И.* Язык UML: Руководство пользователя. СПб.: Питер, 2004. 430 с.

19. *Рамбо Дж., Якобсон А., Буч Г.* Унифицированные процесс разработки программного обеспечения. СПб.: Питер, 2002.
20. *Harel D.* Statecharts: A Visual Formalism for Complex Systems // *Sci. Comput. Programming*, vol. 8, 1987. pp. 231-274.
21. *Керниган Б., Ричи Д.* Язык программирования С. М.: Вильямс, 2005. 304 с.
22. *List of state machine CAD tools.*
http://en.wikipedia.org/wiki/List_of_state_machine_CAD_tools
23. *Головешин А.* Использование конвертора Visio2Switch.
<http://is.ifmo.ru/progeny/visio2switch>
24. *Канжелев С. Ю., Шалыто А. А.* Автоматическая генерация автоматного кода // Информационно-управляющие системы. 2006. № 6.
25. *Чарнецки К., Айзенекер У.* Порождающее программирование: методы, инструменты, применение. СПб.: Питер, 2005.
26. *Adamczyk P.* The anthology of the finite state machine design patterns // The 10th Conference on Pattern Languages of Programs, 2003.
27. *Waldén K., Nerson J.-M.* Seamless Object-Oriented Software Architecture: Analysis and Design of Reliable Systems. Prentice Hall, Hemel Hempstead (U.K.), 1995.
28. *Guttag J. V., Horning J. J.* Larch: Languages and Tools for Formal Specification. New York: Springer, 1993. 571 p.
29. *Гамма Э., Хэлм Р., Джонсон Р., Влиссидес Дж.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб: Питер, 2001.
30. *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.
31. *Шопырин Д. Г.* Методы объектно-ориентированного проектирования и реализации программного обеспечения реактивных систем. Диссертация, 2005.
http://is.ifmo.ru/disser/shopyrin_disser.pdf