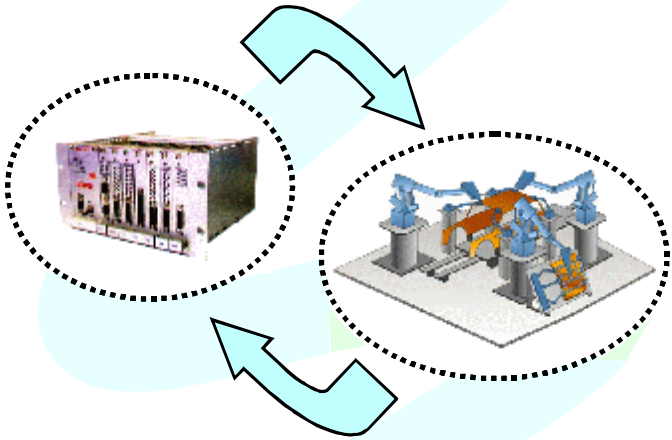
В. Е. Зюбин

ПРОГРАММИРОВАНИЕ
ИНФОРМАЦИОННО-УПРАВЛЯЮЩИХ СИСТЕМ НА ОСНОВЕ КОНЕЧНЫХ АВТОМАТОВ



http://reflex-language.narod.ru

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Факультет информационных технологий
Кафедра информационно-измерительных систем

В. Е. Зюбин

ПРОГРАММИРОВАНИЕ ИНФОРМАЦИОННО-УПРАВЛЯЮЩИХ СИСТЕМ НА ОСНОВЕ КОНЕЧНЫХ АВТОМАТОВ

Учебно-методическое пособие

Новосибирск
2006

ББК В185.12 + З-965
УДК 811.93+62-529+004.434
З-980

Зюбин В. Е. Программирование информационно-управляющих систем на основе конечных автоматов: Учеб.-метод. пособие / Новосиб. гос. ун-т. Новосибирск, 2006. 96 с.

ISBN 5-94356-425-X

В учебно-методическом пособии рассматривается применение модели конечного автомата и его модификаций при создании информационно-управляющих систем. Анализируется специфика задач управления и языки, используемые для описания управляющих алгоритмов. Рассматриваются типовые алгоритмы, используемые при решении задач промышленной автоматизации.

Рецензент
д-р техн. наук, проф. И. Ф. Клисторин

Рекомендовано к изданию в качестве учебно-методического пособия
ученым советом ФИТ НГУ

ISBN 5-94356-425-X

ОГЛАВЛЕНИЕ

Введение

Для понятия «огонь» жрецы Древнего Египта имели более двадцати существительных. Для наводнения – более десяти. И это только дошедшие до нас слова. Та же картина у эскимосов: для снега у них существует более двух десятков различных слов. Перевод каждого такого эскимосского слова на русский язык займет несколько предложений. А ведь мы, в России, хорошо знакомы со снегом и для обозначения замерзшей воды имеем в запасе несколько вариантов: «наст», «иней», «лед», «снег». Африканец при переводе с эскимосского встретится с более серьезной проблемой: ведь у него в языке нет ни одного слова, обозначающего «снег». Разумеется, если хорошенько постарается, он все-таки сможет выразить свою мысль о «холодной-холодной белизне, на огне дающей воду», но результат будет весьма плачевен. Жителю жаркой Африки также непонятны и природные условия Крайнего Севера, и назначение большинства предметов из быта эскимоса, и приемы, необходимые для выживания. Случись ему оказаться в тундре, он, скорее всего, быстро погибнет от нехватки знаний.

В программировании ситуация очень сходна. Кроме королевства персональных компьютеров, где доминируют языки объектно-ориентированного программирования, существуют еще и удаленные, достаточно обширные области, для которых понятийный аппарат языков Си++ и Объектного Паскаля совсем не пригоден. И хотя многие программисты даже не догадываются о существовании этих областей, проводя всю свою жизнь в рамках компьютерного мэйнстрима, специалисты насчитывают более двух тысяч разных языков программирования, отличных по специфике, подходу и области использования.

Одна из таких областей – область промышленной автоматизации и автоматизации научных исследований. Станки, поточные линии, роботы, гибкие производственные комплексы, технологические процессы, сложное диагностическое и экспериментальное оборудование – в настоящее время практически вся промышленная автоматизация реализуется на цифровых системах управления. В качестве базового элемента систем управления используются программируемые логические контроллеры (ПЛК), которые имеют в своем составе микропроцессорный модуль, множество входов и выходов, и *программное обеспечение* – управляющий алгоритм (УА).

УА реализует разнообразные функции: контролирует значения входных / выходных сигналов, выполняет логические / арифметические операции, поддерживает необходимые значения параметров, обеспечивает связь с датчиками и исполнительными устройствами.

УА имеет специфику, которая либо отсутствует, либо не вполне четко проявляется в вычислительных задачах и пользовательских программах для персонального компьютера.

Во-первых, это наличие внешнего мира, *объекта управления* (ОУ), с которым УА постоянно обменивается данными. Через разнообразные датчики в УА непрерывным потоком поступают данные о текущем состоянии ОУ (температуре, давлении, скорости, напряжении, размерах и расстоянии, массе и т. д.) и происходящих событиях. На этот поток данных УА должен реагировать через органы управления (переключатели, клапаны, двигатели, насосы, источники тока, электромоторы), пытаясь привести ОУ в требуемое состояние. Непрерывный характер обмена УА с внешней средой делает неприменимой привычную схему вычисления «дано …, найти …», теряется смысл понятия исходных данных и конечного результата.

Во-вторых, в зависимости от событий на ОУ может в корне меняться характер обработки входных данных, например, при вхождении технологического процесса в критическую зону или при отказе одного из элементов. Поэтому УА должен «уметь» перестраиваться, или, другими словами, быть событийно-управляемым.

В-третьих, реакция УА должна соответствовать динамическим характеристикам ОУ, его физической природе и физическим процессам, протекающим на нем. Таким образом, функционирование УА предполагает большое число операций с временными интервалами: задержками, паузами, тайм-аутами.

В-четвертых, в УА необходимо отражать существование множества физических процессов, независимо протекающих на ОУ. Например, объект может иметь несколько отдельных контуров управления температурой, клапанов и (или) насосов. При наивной попытке организовать монолитное управление возникает необходимость рассматривать все возможные состояния системы целиком, что приводит к так называемому комбинаторному взрыву сложности, экспоненциальной зависимости размеров алгоритма от количества входных / выходных сигналов. В этой ситуации единственная возможность практически решить проблему – обеспечить независимость описания и логический параллелизм его исполнения.

Учебно-методическое пособие, которое вы держите в руках, преследует цель познакомить студента со спецификой информационно-управляющих систем, обеспечить его адекватным понятийным аппаратом для решения задач промышленной автоматизации и обучить его практическим приемам создания управляющих алгоритмов.

В качестве теоретической основы используется модель конечного автомата, очень удобная, но, к сожалению, не используемая в программистской практике так часто, как она того заслуживает.

Курс построен по следующей схеме. Сначала обсуждается общее понятие алгоритма и приводится история создания модели конечных автоматов. Рассматривается специфика задач управления, формулируются необходимые требования к понятийному аппарату. Проводится критический

анализ модели конечного автомата на предмет соответствия этим требованиям. Рассматриваются полезные свойства модели и выявляются концептуальные проблем. Конструируется модель гиперпроцесса (гипераавтомата) для описания управляющих алгоритмов и понятийный аппарат, терминологически приближенный к современным тенденциям в области IT-индустрии. Приводятся способы реализации гиперпроцесса средствами процедурных языков программирования. Обсуждаются языки стандарта МЭК 61131-3 – проблемно-ориентированные языки, которые наиболее часто используются в промышленной практике. Дается описание языка Рефлекс и примеры решения типовых задач, возникающих при автоматизации промышленных объектов.

1. Конечный автомат

1.1. Исторические предпосылки создания модели конечного автомата

Проблема определения понятия «алгоритм». На протяжении многих веков понятие алгоритма связывалось с числами и относительно простыми действиями над ними, да и сама математика была, по большей части, наукой о вычислениях, наукой прикладной. Чаще всего алгоритмы представлялись в виде математических формул. Порядок элементарных шагов алгоритма задавался расстановкой скобок, а сами шаги заключались в выполнении арифметических операций и операций отношения (проверки равенства, неравенства и т. д.). Часто вычисления были громоздкими, а вычисления вручную – трудоемкими, но суть самого вычислительного процесса оставалась очевидной. У математиков не возникала потребность в осознании и строгом определении понятия алгоритма, в его обобщении. Но с развитием математики появлялись новые объекты, которыми приходилось оперировать: векторы, графы, матрицы, множества и др. Как определить для них однозначность или как установить конечность алгоритма, какие шаги считать элементарными? В 1920-х гг. задача точного определения понятия алгоритма стала одной из центральных проблем математики.

Попытки выработать такое определение привели к возникновению теории алгоритмов, в которую вошли труды многих известных математиков – К. Гедель, К. Черч, С. Клини, А. Тьюринг, Э. Пост, А. Марков, А. Колмогоров и др. Несмотря на интересные результаты, полученные в рамках теории алгоритмов, дать четкое определение понятию алгоритма не удалось. И с современной точки зрения «**алгоритм,** алгорифм, – одно из основных понятий (категорий) математики, не обладающих формальным определением в терминах более простых понятий, а абстрагируемых непосредственно из опыта».

Неформальное определение алгоритма. В повседневной жизни каждый человек сталкивается с необходимостью решения задач самой разной сложности. Некоторые из них трудны и требуют длительных размышлений для поиска решений (а иногда его так и не удается найти), другие же, напротив, столь просты и привычны, что решаются автоматически. При этом выполнение даже самой простой задачи осуществляется в несколько последовательных этапов (шагов). В виде последовательности шагов можно описать процесс решения многих задач, известных из школьного курса математики: приведение дробей к общему знаменателю, решение системы линейных уравнений путем последовательного исключения неизвестных, построение треугольника по трем сторонам с помо-

щью циркуля и линейки и т. д. Это всё алгоритмы, рано или поздно приводящие к конечному результату. В то же время можно привести примеры формально бесконечных алгоритмов, широко применяемых на практике. Например, алгоритм работы системы сбора метеорологических данных состоит в непрерывном повторении последовательности действий («измерить температуру воздуха», «определить атмосферное давление»), выполняемых с определенной частотой (через минуту, час) во все время существования данной системы.

Понятие алгоритма близко к другим понятиям, таким, как метод (метод Гаусса решения систем линейных уравнений), способ (способ построения треугольника по трем сторонам с помощью циркуля и линейки).

Дадим неформальное определение алгоритма. Алгоритм – это точно определенная, возможно циклически повторяемая инструкция по преобразованию входных данных в выходные.

Появление первых проектов вычислительных машин стимулировало исследование возможностей практического применения алгоритмов, использование которых ввиду их трудоемкости было ранее недоступно. Дальнейший процесс развития вычислительной техники определил развитие теоретических и прикладных аспектов изучения алгоритмов.

Формы представления алгоритмов. Для записи алгоритмов необходим некоторый язык, при этом очень важно, какой именно язык выбран. Записывать алгоритмы на русском языке (или любом другом естественном языке) неудобно.

Например, описание алгоритма Евклида нахождения НОД (наибольшего общего делителя) двух целых положительных чисел может быть представлено в виде трех шагов. Шаг 1: Разделить m на n. Пусть p – остаток от деления.

Шаг 2: Если p равно нулю, то n и есть исходный НОД.

Шаг 3: Если p не равно нулю, то сделаем m равным n, а n равным p. Вернуться к шагу 1.

Приведенная здесь запись алгоритма нахождения НОД очень упрощенная. Запись, данная Евклидом, представляет собой страницу текста, причем последовательность действий существенно сложней.

Одним из распространенных способов записи алгоритмов является запись на языке блок-схем. Запись представляет собой набор элементов (блоков), соединенных стрелками. Каждый элемент – это «шаг» алгоритма. Элементы блок-схемы делятся на два вида. Элементы, содержащие инструкцию выполнения какого-либо действия, обозначают прямоугольниками, а элементы, содержащие проверку условия, – ромбами. Из прямоугольников всегда выходит только одна стрелка (входить может несколько), а из ромбов – две (одна из них помечается словом «да», другая – словом «нет», они показывают, соответственно, выполнено или нет проверяемое условие).

Построение блок-схем из элементов всего лишь нескольких типов дает возможность преобразовать их в компьютерные программы и позволяет формализовать этот процесс (рис. 1.1).

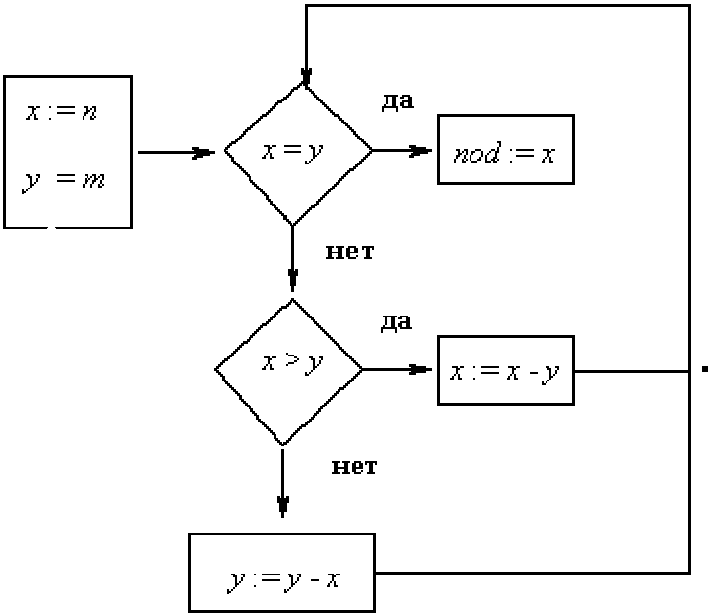


Рис. 1.1. Блок-схема нахождения НОД

Проблема формализации записи алгоритмов. Задача нахождения единообразной формы записи алгоритмов, решающих различные задачи, была одной из основных задач теории алгоритмов. Математические определения фигур, чисел, уравнений, неравенств и многих других объектов достаточно четки. Каждый математически определенный объект можно сравнить с другим объектом, соответствующим тому же определению. Например, прямоугольник можно сравнить с другим прямоугольником по площади или длине периметра. Возможность сравнения математически определенных объектов – важный момент математического изучения этих объектов. Но как сравнивать два произвольных алгоритма? Можно, например, сравнить два алгоритма решения системы уравнений и выбрать более подходящий в данном случае, но невозможно сравнить алгоритм перехода через улицу с алгоритмом извлечения квадратного корня. С этой целью нужно формализовать понятие алгоритма, т. е. отвлечься от сущест-

ва решаемой данным алгоритмом задачи и выделить свойства различных алгоритмов, привлекая к рассмотрению только его форму записи.

В связи с этим в теории алгоритмов была поставлена следующая задача: представить алгоритм в таком виде, чтобы каждый элементарный шаг алгоритма мог быть выполнен достаточно простым устройством (машиной). Желательно, чтобы это устройство было универсальным, т. е. чтобы на нем можно было выполнять любой алгоритм. Механизм работы машины должен быть максимально простым по логической структуре, но настолько точным, чтобы эта структура могла служить предметом математического исследования. Впервые это было сделано математиком из США Эмилем Постом в 1936 г. (машина Поста) еще до создания современных вычислительных машин и (практически одновременно) английским математиком Аланом Тьюрингом (машина Тьюринга).

Обе машины были созданы для уточнения понятия «алгоритм».

Машина Поста. Статья Э. Поста, в которой была впервые описана его машина, называлась «Финитные комбинаторные процессы, формулировка 1». В этой статье Пост доказывал, что предлагаемая система обладает алгоритмической полнотой. В 1967 г. профессор В. Успенский пересказал эти статьи с новых позиций. Он же и ввел в употребление термин «машина Поста» и предложил новую интерпретацию. Машина Поста (МП) – абстрактная машина, которая работает по алгоритмам, разработанным человеком, и обладает следующим свойством: если для решения задачи можно построить машину Поста, то она алгоритмически разрешима.

Э. Пост
(1897–1954)

Машина Поста (рис. 1.2) состоит из каретки (или считывающей / записывающей головки), устройства управления кареткой (УУ) и разбитой на ячейки бесконечной в обе стороны ленты. Каждая ячейка ленты может быть либо пустой, либо помеченной меткой ‘●’. В МП используется унарная система счисления, число Р записывается как Р + 1 единиц: числу 4 соответствует 5 ячеек с метками, расположенных по порядку. За один шаг каретка может сдвинуться на одну позицию влево или вправо, считать, поставить или уничтожить символ в том месте, где она стоит.

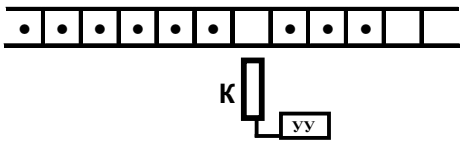


Рис. 1.2. Машина Поста

Работа машины Поста определяется программой, записанной в УУ и состоящей из конечного числа строк. Всего команд шесть:

n. → a	Сдвиг каретки вправо и переход к строке a
n. ← a	Сдвиг каретки влево и переход к строке a
n. v a	Запись метки и переход к строке a
n. × a	Удаление метки и переход к строке a
n. ? a, b	Проверка состояния ячейки; если ячейка пустая, то перейти к строке a, иначе перейти к строке b
n. !	Останов

Здесь *n* – номер строки в программе, *a*, *b* – строка, на которую передается управление.

Для работы машины нужно задать программу и ее начальное состояние (т. е. состояние ленты и позицию каретки). После запуска возможны варианты:

- работа может закончиться ошибкой (невыполнимой командой: стирание несуществующей метки или запись в помеченное поле);
- работа может закончиться командой «останов»;
- работа никогда не закончится.

В качестве примера работы и программирования машины Поста рассмотрим задачу вычитания натуральных чисел P – Q.

Как это принято в машине Поста, будем представлять натуральное (целое неотрицательное) число P набором из P + 1 единиц и разделять числа нулём. Исходное положение каретки помечено символом *v*:

```
          V
  00111110111000
      P     Q
```

Сложение двух чисел тривиально – достаточно поставить 1 между ними и стереть крайний правый символ у Q.

Программа вычитания состоит из последовательного затирания крайних левых меток у Q и правых у P:

1. **x** – стираем левый символ у Q
2. →
3. **? 5, 4**
4. **Stop** – стоп, если затерли Q = 0 (терминальную единицу)
5. ←
6. **? 7,** 5 – цикл поиска P
7. **x** – стираем правый символ у P
8. →
9. **? 1, 8** – ищем Q

Отметим, что номер команды перехода не указывается, если переход происходит на следующую по порядку строку (для наглядности текста).

В шестой строке возможно зацикливание, если Q > P (вы можете добавить проверку сами).

В 1970 г. сотрудники Симферопольского университета разработали машину Поста «в металле».

Важность идей Эмиля Поста в том, что был предложен простейший способ преобразования информации, а именно он построил алгоритмическую систему (алгоритмическая система Поста):

- вся информация, которая должна быть обработана по существу, должна быть обработана по форме, т. е. с помощью двоичного алфавита;
- вся информация должна обрабатываться побуквенно.

Машина Тьюринга (рис. 1.3) представляет собой некоторое автоматически действующее устройство управления (УУ), способное находиться в конечном числе внутренних состояний: $S = \{s_0, s_1, \ldots, s_p\}$ (от англ. *state*), и снабженное бесконечной внешней памятью – лентой (Л). Среди состояний имеются два выделенных – начальное (s_1) и заключительное (s_0). Лента разделена на ячейки (клетки) и не ограничена влево и вправо.

А. Тьюринг
(1912–1954)

В каждой ячейке ленты может быть записан любой из символов, входящих в некоторый заранее заданный перечень $C = \{c_0, c_1, \ldots, c_m\}$ (от англ. *character*). Ради единообразия считают, что в пустой ячейке записана «пустая буква» – c_0.

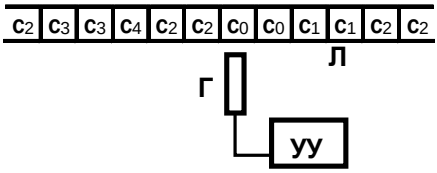


Рис. 1.3. Машина Тьюринга

В каждый момент времени машина Тьюринга находится в одном из своих состояний (s_i) и, рассматривая (посредством специального устройства – головки (Г) для считывания / записи) одну из ячеек своей ленты, воспринимает записанный в ней символ (c_j). Если в текущий момент времени машина находится не в заключительном состоянии (s_0), то в следующий за ним момент:

- она переходит в новое состояние (s_{i+1}), которое в частном случае может совпадать со старым или оказаться заключительным;
- в рассматриваемой ячейке старый символ (c_j) заменяется новым, который в частном случае может быть пустым (c_0) или совпадающим со старым;
- лента машины сдвигается на одну ячейку влево или вправо либо остается на месте. Этот шаг машины вполне определяется её текущим состоянием (s_i) и текущим воспринимаемым символом (c_j).

Программа для машины Тьюринга представляет собой таблицу, в каждой ячейке которой записана команда. Команда представляет собой указание: какой символ записать в текущую ячейку ленты, в какую сторону передвинуть головку чтения / записи и в какое состояние перейти машине.

Текущее полное описание машины Тьюринга задаётся её конфигурацией, которая состоит из указания для данного момента следующей информации:

- конкретного заполнения клеток ленты символами;
- клетки, находящейся в поле зрения машины;
- состояния, в котором машина находится.

Если у конкретной машины Тьюринга взять в качестве исходной какую-либо конфигурацию с не-заключительным состоянием, то работа этой машины будет состоять в последовательном преобразовании исходной конфигурации в соответствии с программой машины. Работа будет продолжаться до тех пор, пока не будет достигнута конфигурация с заключительным состоянием. Эта последняя, если она существует, и считается результатом работы данной машины над исходной конфигурацией.

По своему устройству машина Тьюринга крайне примитивна. Она намного проще самых первых компьютеров. Примитивизм состоит в том, что у нее предельно прост набор элементарных операций, производимых головкой – считывание и запись, а также в том, что доступ к ячейкам памяти (секциям ленты) в ней происходит не по адресу, как в компьютерах, а путем последовательного перемещения вдоль ленты. По этой причине даже такие простые действия, как сложение или сравнение двух символов, машина Тьюринга производит за несколько шагов, а обычные операции сложения и умножения требуют весьма большого числа элементарных действий. Однако машина Тьюринга была придумана не как модель (прототип) реальных вычислительных машин, а для того, чтобы показать принципиальную (теоретическую) возможность построения сколь угодно сложного алгоритма из предельно простых операций, причем сами операции и переход от одной к последующей машина выполняет автоматически.

В 1937 г. сотрудники Малой Крымской академии наук построили машину Тьюринга «в металле».

Обе машины – Поста и Тьюринга – «эквивалентны», т. е. доказано, что класс алгоритмов, представленных в форме машины Поста, и класс алгоритмов, представленных в форме машины Тьюринга, совпадают. В рамках теории алгоритмов Тьюринг выдвинул тезис, получивший название тезис Тьюринга, согласно которому всякий алгоритм представим в форме машины Тьюринга. Это принимаемое без доказательства фундаментальное положение теории алгоритмов может также рассматриваться как неформальное определения понятия «алгоритм».

Важный момент, на который следует обратить внимание, – это характер данных, записываемых и считываемых с ленты. Как и в случае с машиной Поста, используются численные значения фиксированной длины – алфавит.

Машина Поста и машина Тьюринга в дальнейшем получили развитие в качестве модели конечного автомата.

Вопросы для самопроверки

Вопрос 1. В приведенной программе для машины Поста в шестой строке возможно зацикливание, если Q > P. Попытайтесь добавить проверку.

Вопрос 2. Как вы считаете, почему известна лишь одна практическая реализация машины Тьюринга? Зачем это было сделано?

Вопрос 3. Назовите число ячеек машины Поста, необходимых для хранения максимально возможной суммы двух байтовых переменных?

Вопрос 4. Что сложнее, машина Поста или машина Тьюринга? Как вы считаете, для какой из этих машин проще создавать программы?

1.2. Математическая модель абстрактного автомата

В середине XX столетия была создана и получила широкое использование электронная цифровая машина с программным управлением. На повестку дня встал вопрос о рациональном конструировании, или, как было принято говорить, *синтеза* схем таких машин. Первые ЭВМ рассматривались как преобразователи дискретных данных. Поэтому в литературе было достаточно распространено альтернативное название для ЭВМ – *дискретные*, или *цифровые автоматы* (устройства, машины).

Для формализации решения задач синтеза дискретных устройств была предложена математическая модель в виде абстрактного автомата, во многом сходная с машиной Тьюринга. Абстрактный автомат A задается шестеркой:

$A = (S, I, O, Fs, Fo, s_1)$, у которого:

1. $S = \{s_1, \ldots, s_k, \ldots, s_K\}$ – множество состояний (алфавит состояний);
2. $I = \{i_1, \ldots, i_m, \ldots, i_M\}$ – множество входных символов (входной алфавит);
3. $O = \{o_1, \ldots, o_n, \ldots, o_N\}$ – множество выходных символов (выходной алфавит);
4. $Fs : S \times I \to S$ – функция переходов, отображающая $D_{Fs} \subseteq S \times I$ в S. Другими словами, функция Fs некоторым парам состояние – входной символ (s_k, i_m) ставит в соответствие состояние автомата $s_j = Fs\,(s_k, i_m)$, $s_j \in S$;

5. $Fo : S \times I \to O$ – функция выходов, отображающая $D_{Fo} \subseteq S \times I$ в O. Функция Fo некоторым парам состояние – входной символ (s_k, i_m) ставит в соответствие выходные символы автомата, $o_n = Fs\ (s_k, i_m)$, $o_n \in O$;

6. $s_1 \in S$ – начальное состояние автомата.

Обозначения выбраны по следующим соображениям: A – от «automaton», S / s – от «state», I / i – от «input», O / o – от «output», F – от «function».

Автомат называется *полностью определенным*, если $D_{Fs} = D_{Fo} = S \times I$.

1.3. Необходимые пояснения к понятию «конечный автомат»

Поговорим немного об этой модели и попытаемся найти точки соприкосновения этой математической модели с реальной жизнью.

Часто дезориентирует само название «абстрактный автомат». Почему «абстрактный», почему «автомат»? Повторимся: «автомат» – это «аппарат (машина, прибор, устройство), после включения самостоятельно выполняющий ряд заданных операций». Слово «абстрактный» можно переводить как «существующий только с математической точки зрения». Это слово используется потому, что с математической точки зрения множества состояний, входных и выходных символов у автомата могут быть бесконечными множествами. Разумеется, все существующие на практике цифровые устройства, будучи представлены в виде абстрактного автомата, имеют конечные множества S, I, O. В рамках теории автоматов модели таких устройств называются *конечными автоматами*. Впредь мы всегда будем говорить исключительно о конечных автоматах.

Что значит «алфавит»? В теории автоматов это «непустое множество попарно различных символов». Опять получается не очень понятно. Чтобы разобраться с этим вопросом, следует вспомнить некоторые факты из истории.

Дело в том, что при возникновении теории автоматов кодирование в двоичной системе не было стандартом, поэтому вопросы кодирования рассматривались отдельно. На физическом уровне допускалось кодирование не только в виде битов (0 / 1), но и в троичной системе, четверичной и т. д. Также не было использовано понятие «переменная». Эти исторические факты, как правило, неизвестны студентам, и без них понимание модели автомата уже в этой части может вызвать затруднения.

С точки зрения современной практики программирования все значительно проще. Под символами в теории автоматов понимаются просто некоторые значения (переменной). Входной (выходной) алфавит – это просто переменная с заранее определенным набором допустимым значений.

Из сказанного можно сделать один весьма интересный вывод. Поскольку практическая работа в силу чисто физических ограничений

проектировщика может вестись с сотней-другой значений, весь алфавит большинства реальных автоматов может быть представлен байтом, максимум двумя.

Цикличность работы автомата. Часто в полном соответствии с принятым в кибернетике подходом автомат рассматривается как «черный ящик» и изображается так, как показано на рис. 1.4. Автомат имеет один вход и один выход. На вход ему подается последовательность разрешенных входных значений. В теории автоматов эта последовательность называется входным словом. На выходе автомат генерирует выходную последовательность заранее определенных значений, которая называется выходным словом. На уровне абстрактной теории понятие «работа автомата» понимается как преобразование входной последовательности значений в выходную последовательность значений. Автомат получает на своем входе некоторое значение и на основании этого значения и текущего внутреннего состояния формирует выходное значение.

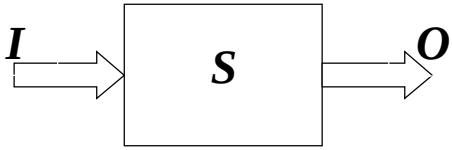


Рис. 1.4. Автомат

Поскольку прозвучало слово «работа» и сделана заявка на кибернетический подход, в котором основное внимание уделяется поведению системы с точки зрения внешней среды, возникает целый ряд вопросов. Например, зависит ли поведение автомата от частоты подачи ему на вход новых значений? Как устроен автомат и что же такое «состояние»? К сожалению, этим важным вопросам редко уделяется должное внимание.

Для начала вспомним, что с помощью автомата моделируется цифровое устройство, т. е. полностью дискретное устройство, в котором все сигналы квантованы по уровню, а все действия квантованы по времени. Квантование работы автомата по времени, необходимость считывать значения со своего входа, обрабатывать их и реагировать на входные значения соответствующим образом, выдавая выходные значения, возможно единственно в соответствии с *циклической* схемой, изображенной на рис. 1.5.

К сожалению, классическая теория автоматов опускает обсуждение этого важного вопроса. Мы же можем сделать по этому поводу следующую ремарку. Поскольку действия цифровой системы требуют временных затрат на исполнение, становится понятно, что у автомата есть еще и некоторая временная характеристика – временной интервал, характеризующий цикл исполнения. Этот временной интервал может быть постоянным или плавающим, но самое главное, что сейчас требуется осознать, – сам факт его существования. Возьмем этот факт на заметку.

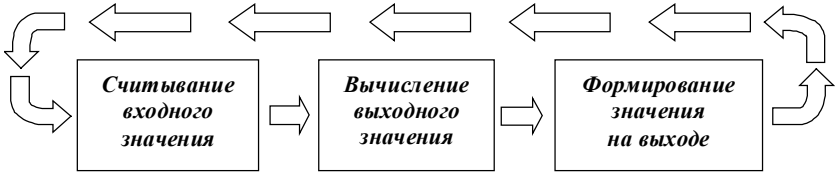


Рис. 1.5. Циклическая схема работы автомата

Понятие «состояние». Про состояние обычно говорится следующее. «Понятие состояния введено в связи с тем, что часто возникает необходимость в описании поведения систем, выходы которых зависят не только от состояния входов в данный момент времени, но и от некоторой предыстории, т. е. от сигналов, которые поступали на вход системы ранее. Состояние как раз и соответствует некоторой памяти о прошлом, позволяя устранить время как явную переменную и выразить выходной сигнал как функцию состояния и входа в данный момент времени». Это требуется прокомментировать. Во-первых, не стоит обращать внимание на заявление о том, что состояние позволяет устранить время как явную переменную. Это не совсем так. Забегая вперед, можно сказать, что оно действительно может сильно помочь и обеспечить работу с временны́ми интервалами, но в рамках классической теории автоматов это не делается. Во-вторых, в приведенном отрывке поясняется, *для чего* служит состояние и *что достигается* с помощью состояния, но ответа на вопрос, *что же такое* состояние, не звучит.

Самая простая интерпретация состояния – это просто некоторое числовое значение. Аналогично входному и входному алфавиту. Соответственно, можно ввести понятие «алфавит состояний», понимая под этим некоторую переменную (ячейку памяти) со специфицированным набором допустимых значений состояний. Функцию переходов *Fs* и функцию выходов *Fo* без потери связи с теорией автоматов можно воспринимать уже как функции в программистском смысле, – функции от двух переменных типа «беззнаковое целое», возвращающие целое значение, – *Fo*(*s*, *i*) и *Fs*(*s*, *i*).

После этого пояснения становится очевидным, что же скрыто за элементом «вычисление выходного значения» на рис. 1.5. В терминах языка Си это будут две строки (рис. 1.6).

Обратим внимание на порядок вызова функций *Fo* и *Fs*, в каком порядке вычисляются эти функции. Вообще говоря, порядок может быть произвольным, но приведенный на рис. 1.6 удобнее с точки зрения использования: в качестве аргументов обеих функций используются одни и те же значения, что минимизирует число рассматриваемых вариантов.

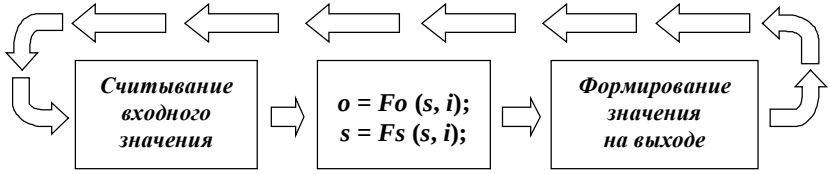


Рис. 1.6. Цикл автомата. Раскрытие элемента
«вычисление выходного значения»

О внутреннем наполнении функций переходов и выходов можно сказать следующее. Если аналитические зависимости между значением входов и значением выходов еще встречаются на практике, то аналитические зависимости между состояниями и выходами – практически никогда, также пренебрежимо редки случаи, когда в аналитическом виде можно задать функцию *Fs*.

Это означает, что в подавляющем большинстве случаев функции *Fo* и *Fs* задаются исключительно таблично. С другой стороны, при создании устройства нас в первую очередь волнует его поведение, реакция устройства на внешнее воздействие в конкретный момент времени, для данного цикла. Внутреннее наполнение, – его состояние, – может выбираться произвольно, к нему не предъявляется никаких жестких требований, в кибернетическом подходе оно малоинтересно. Эти обстоятельства делают более удобным подход, в котором сначала производится абстрагирование от значения переменной «состояние» *s* и, тем самым, функции переходов и выходов редуцируются к функциям от одного переменного. Дополнительный плюс такого подхода в том, что функция выходов, возможно, описывается в компактном аналитическом виде. Именно этот порядок формирования функций негласно подразумевается, когда говорится, что «состояние – это память о прошлом».

На рис. 1.7 отражен такой подход.

Сначала по текущему значению *s* определяется, какие же конкретно функции использовать для вычисления значений выходов и состояния следующего цикла, а затем выбранные функции исполняются. При этом функции выходов и переходов задаются как множества функций *Fo_XX*, *Fs_XX* от одного переменного – значения выходов *i* (*XX* – служит для идентификации принадлежности функции к определенному состоянию, например, F_0*_41*, *Fs_35* и т. п.).

Если взглянуть на состояние и порядок работы с ним под другим углом, то мы придем к понятию *функций-состояний* – функций, объединяющих редуцированные функции выходов и переходов. Эти функции

взаимоисключающие в том смысле, что на одном цикле автомата исполняется только одна функция-состояние – текущая.

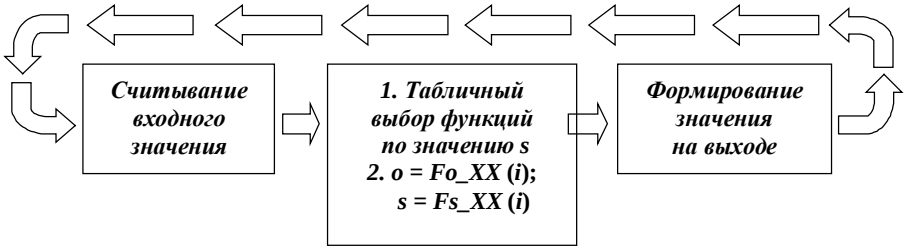


Рис. 1.7. Цикл автомата. Редукция числа аргументов функций выхода и перехода

При вызове функция-состояние определяет, как отреагировать на входное значение – какое выходное значение сформировать на текущем цикле автомата, и вычисляет, какая функция-состояние будет вызвана на следующем цикле автомата – станет текущей.

Понятие функции-состояния существенно для изучения способов задания автоматов.

Автомат с одним состоянием. Для автоматов с одним состоянием необходимость в табличном выборе функций отпадает, так как вариант всегда один. Автоматы такого класса называются *комбинационные схемы*. У таких автоматов для одинакового входного значения выходные значения всегда одинаковы.

Комбинационная схема осуществляет перевод одного множества значений в другие. С помощью комбинаторной схемы можно реализовать взаимно однозначное отображение произвольного конечного множества из N элементов на множество $\{1, 2, 3, \ldots, N\}$ и наоборот. Если после считывания входного значения поставить комбинаторную схему, отображающую M-элементное множество I в множество $\{1, 2, \ldots, M\}$, а перед выдачей выходного значения поставить комбинаторную схему, отображающую N-элементное множество O в множество $\{1, 2, \ldots, N\}$, то множества I и O в модели автомата можно заменить на множества $\{1, 2, \ldots, M\}$ и $\{1, 2, \ldots, N\}$.

Замена множества S на множество $\{1, 2, \ldots, K\}$ также несущественна. Более того, с переходом к понятию функции-состояния теряет смысл использование самого понятия алфавита состояний.

Модернизированная модель конечного автомата. С учетом сказанного можно определить конечный автомат как циклически активизируемый набор альтернативных функций, полностью определяемый пятеркой:

$A = (F, x, y, f_{cur}, f_1)$, у которого:

1. x – входная переменная с допустимыми значениями из множества $I = \{1, 2, 3, \ldots, m, \ldots, M\}$;

2. y – выходная переменная с допустимыми значениями из множества $O = \{1, 2, 3, \ldots, n, \ldots, N\}$;

3. $F = \{f_1, \ldots, f_k, \ldots, f_K\}$ – множество альтернативных функций-состояний, отображающих I в O и I в F. Другими словами, каждая из функций f_k содержит две подфункции: а) подфункцию выходов fo_k, которая некоторому значению входа m ставит в соответствие значение выхода n, и подфункции переходов fs_k, которая вычисляет функцию-состояние для следующего цикла из множества F : $y = fo_k(x)$, $fo_k(x) \in O$; $f_{cur} = fs_k(x)$, $fs_k(x) \in F$;

4. $f_{\text{cur}} \in F$ – текущая функция-состояние автомата для рассматриваемого момента времени;

5. $f_1 \in F$ – начальная функция-состояние автомата, текущая функция-состояние по началу работы.

Текущее полное описание конечного автомата задаётся его текущей функцией-состоянием f_{cur} для рассматриваемого момента времени, точнее – для рассматриваемого цикла.

Дополнительное указание на множества I и O в данном определении и спецификация их вида ($\{1, 2, 3, . . ., N\}$) не дает никаких видимых преимуществ при работе с моделью. Для простоты можно рассматривать и все возможные значения переменных x и y, просто автомат будет не полностью определенным. Смысл этих манипуляций – показать, *что* модель автомата представляет собой и *как* она может быть трансформирована.

В конце параграфа вернемся к математической модели автомата и попытаемся ответить на серию вопросов для самопроверки.

Вопросы для самопроверки

Вопрос 1. В чем отличие начального состояния автомата s_1 от других состояний?

Вопрос 2. Как, по вашему мнению, может быть реализована функция выходов *Fo* на языке программирования Си? То же для функции переходов *Fs*? То же для множества функций-состояний F в модифицированной модели? При выполнении задания вспомните, какой оператор языка Си предназначен для реализации табличного разбора.

Вопрос 3. Назовите максимальный элемент из возможных вариантов множества I, если x – это переменная типа unsigned char (байт)?

1.4. Автоматы Мили и Мура. Способы задания автоматов

На практике в рамках теории автоматов наибольшее распространение получили два класса автоматов – автоматы Мили и автоматы Мура, названные по имени исследовавших эти модели ученых из США G. H. Mealy и E. F. Moore. Автоматы Мили и Мура – это просто две различные формы конечных автоматов. Функционирование конечного автомата в форме автомата Мили задается уравнениями

$s(t+1) = Fs\,(s(t), i(t)); o(t+1) = Fo\,(s(t), i(t)), t = 0, 1, 2, \ldots,$

а закон функционирования автомата Мура – уравнениями

$s(t+1) = Fs\,(s(t), i(t)); o(t+1) = Fo\,(s(t)), t = 0, 1, 2 \ldots$

Формы описания Мура и Мили отражают наше желание изобразить функционирование конечного автомата во времени. Однако с учетом отмеченного в предыдущем параграфе циклического характера функционирования конечного автомата под t следует понимать, разумеется, не время, а порядковый номер цикла (начиная с момента запуска, $t = 0$).

Отличия между этими формами задания автомата следующие. В функциях-состояниях автомата Мура значение выдаваемого сигнала y неизменно ($fo_k\,(x) = \text{const}$), а у автомата Мили $fo_k\,(x) \neq \text{const}$ и вычисляется каждый раз в зависимости от значения входного сигнала x.

Табличное задание автоматов Мили и Мура. Как уже отмечалось, в подавляющем большинстве случаев невозможно описать функции fo_k, fs_k аналитически. Поэтому один из наиболее популярных средств задания автоматов – таблица (табл. 1.1, 1.2).

Таблица 1.1.
Функция переходов автомата Мили. Автомат A_1

Буквы входного алфавита	Состояния		
	s_1	s_2	s_3
i_1	s_2	s_3	s_2
i_2	s_3	s_2	s_1

Таблица 1.2.
Функция выходов автомата Мили. Автомат A_1

Буквы входного алфавита	Состояния		
	s_1	s_2	s_3
i_1	o_1	o_3	o_3
i_2	o_2	o_1	o_1

Строки этих таблиц соответствуют возможным значениям входной переменной x, а столбцы – возможным значениям переменной «состояние» s. На пересечении столбца s_k и строки i_m в левой таблице (таблице переходов) указывается значение состояния для следующего цикла $s_j = Fs\,(s_k, i_m)$, а в

правой таблице (таблице выходов) – значение сигнала $y = Fo\ (s_k, i_m)$, выдаваемого на выход.

Аналогичные таблицы (табл. 1.3, 1.4) для некоторого автомата Мура приведены ниже.

Таблица 1.3. Функция переходов автомата A_2

Буквы входного алфавита	Состояния		
	s_1	s_2	s_3
i_1 i_2	s_2 s_3	s_1 s_2	s_1 s_2

Таблица 1.4. Функция выходов автомата A_2

Буквы входного алфавита	Состояния		
	s_1	s_2	s_3
i_1 i_2	o_1 o_1	o_1 o_1	o_2 o_2

Поскольку в автомате Мура функции fo_k имеют вид y = const (обратите внимание, что в таблице для Fo обе строки с o_i-ми совпадают, зависят только от значения состояния), то описание автомата Мура может быть дано в одной таблице (табл. 1.5).

Таблица 1.5. Совмещенная таблица для автомата A_2

Буквы входного алфавита	Буквы выходного алфавита		
	o_1	o_1	o_2
	Состояния		
	s_1	s_2	s_3
i_1 i_2	s_2 s_3	s_1 s_2	s_1 s_2

Иногда и при задании автоматов Мили используют совмещенную таблицу (табл. 1.6) переходов и выходов, в которой на пересечении столбца s_k и строки i_m записываются сразу и новые значения s и o.

В отличие от машин Тьюринга и Поста автоматы Мили и Мура достаточно широко использовались на практике. Практика показала, что общим

минусом для обеих моделей является экспоненциальный рост размеров их таблиц. В качестве одного из способов снижения сложности таблиц был предложен частичный способ задания автоматов, когда не все ячейки специфицируются, например в случаях, когда требуется сохранить старое значение состояния или выходного сигнала. В таких случаях в ячейке ставится прочерк. Это не снимает проблему экспоненциального роста сложности полностью, но упрощает работу с таблицами и в ряде случаев позволяет сократить число функций-состояний автомата (табл. 1.7, 1.8).

Таблица 1.6. Совмещенная таблица для автомата A_1

Буквы входного алфавита	Состояния		
	s_1	s_2	s_3
i_1	s_2/o_1	s_3/o_3	s_2/o_3
i_2	s_3/o_2	s_2/o_1	s_1/o_1

Таблица 1.7. Функция переходов автомата Мили

Буквы входного алфавита	Состояния		
	s_1	s_2	s_3
i_1	s_2	s_3	s_2
i_2	s_3	–	s_1

Таблица 1.8. Функция выходов автомата Мили

Буквы входного алфавита	Состояния		
	s_1	s_2	s_3
i_1	o_1	o_3	–
i_2	o_2	–	o_1

Или то же с помощью совмещенной таблицы (табл. 1.9):

Таблица 1.9. Совмещенная таблица для автомата Мили

Буквы входного алфавита	Состояния		
	s_1	s_2	s_3
i_1	s_2 / o_1	s_3 / o_3	$s_2 / -$
i_2	s_3 / o_2	–	s_1 / o_1

Графическое задание автоматов Мили и Мура. *Граф автомат* – ориентированный граф, вершины которого соответствуют значениям переменной s, а дуги – переходам между ними. Пример такого графа дан на рис. 1.8 (автомат Мили) и рис. 1.9 (автомат Мура).

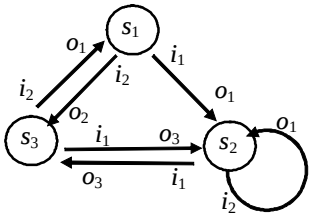


Рис. 1.8. Граф автомата Мили A_1

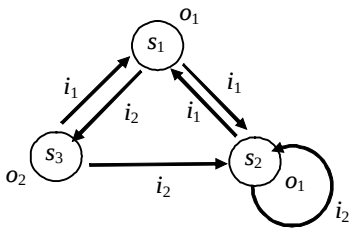


Рис. 1.9. Граф автомата Мура A_2

Преобразование автоматов Мили и Мура. Автоматы Мили и Мура могут быть преобразованы друг в друга. Эти преобразования с некоторой натяжкой можно назвать взаимнооднозначными. Возникающие при этом проблемы связаны с начальным состоянием: для автомата Мура первое выходное воздействие всегда одинаково, в то время как для автомата Мили его можно сделать разным. При преобразованиях выходные сигналы сдвигаются на цикл вперед или назад в зависимости от направления преобразования.

Переход от автомата Мура к автомату Мили осуществляется по схеме, изображенной на рис. 1.10, обратный переход – на рис. 1.11.

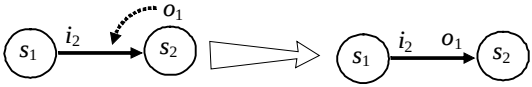


Рис. 1.10. Преобразование автомата Мура к автомату Мили

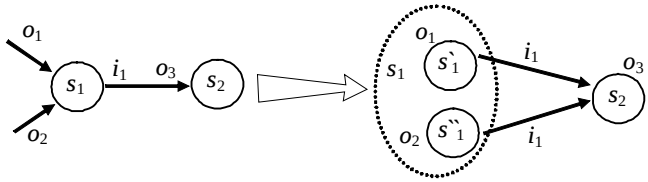


Рис. 1.11. Преобразование автомата Мили к автомату Мура

Совмещенный автомат (автомат Мура + автомат Мили). При использовании на практике автомат Мура и автомат Мили имеют свои специфические особенности. Так, автомат Мура, несомненно, более привлекателен для задач управления, при ожидании сигнала обратной связи. В этом случае установленный сигнал держится до тех пор, пока не придет

сигнал о срабатывании устройства. Ожидание внешнего события удобнее всего реализовывать в одной функции-состоянии, в то время как внутренние вычисления, не требующие обратной связи, удобнее производить с использованием автомата Мили. Как правило, в реальных задачах возникает необходимость в обоих типах операций.

Соображения практического характера говорят о целесообразности использования совмещенной модели автомата Мили и Мура – С-автомата. Проблема заключается в том, что автоматы Мили и Мура при попытке совместного использования начинают конфликтовать: вспомним, что в автомате Мура выходное значение – это неизменная для текущей функции-состояния константа, в то время как в автомате Мили она может меняться в зависимости от состояния входов. В теории автоматов в качестве средства разрешения конфликта предлагается разбивать входной алфавит на два непересекающихся подмножества.

В терминах модифицированной модели конечного автомата это означает появление второй выходной переменной.

Особенности использования конечных автоматов в начале компьютерной эпохи. Теория автоматов активно развивалась на фоне процесса быстрого вытеснения аналоговых управляющих систем дискретными автоматическими устройствами в эпоху зарождения цифровой техники. В начале этого процесса отсутствовали не только высокоразвитые системы программирования и среды разработки, без которых в настоящее время невозможно представить цифровую технику. Отсутствовали языки высокоуровневого программирования. Не было даже концепции ассемблеров. Все «программирование» велось в терминах нулей и единиц, в терминах самых примитивных операций. Машинные команды 8-разрядных процессоров были бы восприняты в конце 1950-х гг. как непозволительная роскошь. Эти обстоятельства следует учитывать при анализе возникавших в то время идей и применявшихся подходов. В рамках теории автоматов конструировались не программы, а устройства. На решение этой проблемы и были направлены основные усилия теоретиков.

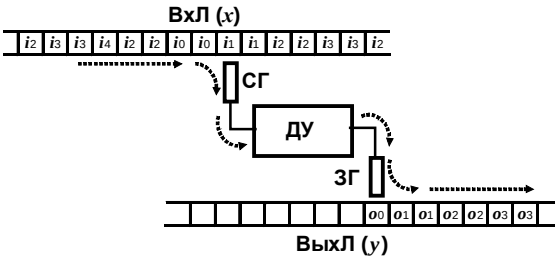


Рис. 1.12. Конечный автомат

Конечный автомат – ближайший родственник машин Поста и Тьюринга – воспринимался в первую очередь как механическое устройство. Это гипотетическое устройство (рис. 1.12) имеет две ленты, которые двигаются синхронно и строго поступательно,

слева направо. Считывающая головка (СГ) считывает символы из ячеек входной ленты (ВхЛ). Эти символы поступают на вход дискретного устройства (ДУ). В соответствии со считанными значениями и внутренним «конечно-автоматным» алгоритмом ДУ формирует выходную последовательность значений, которая записывается в ячейки выходной ленты (ВыхЛ) посредством записывающей головки (ЗГ).

Несмотря на присущий конечному автомату механицизм, его концептуальная схема достаточно универсальна. Например, она позволяет хорошо описывать формальные языки и грамматики. Один из мощных разделов теории автоматов – математическая лингвистика, в частности, широко используемая при создании трансляторов языков программирования.

Другая широчайшая область их применения – конструирование информационно-управляющих систем, в котором «входная» и «выходная» ленты служат для общения с внешней средой, например с объектом управления. «Входная лента» подменяется потоком данных о состоянии объекта управления, а «выходная лента» – сигналами управления, формируемыми управляющим алгоритмом.

Большой вклад в развитие теории автоматов и разработку методов их практического применения в информационно-управляющих системах внесли отечественные ученые-кибернетики, такие как В. М. Глушков, М. А. Гаврилов, С. И. Баранов и многие другие.

Синтез конечного автомата. Схемный подход. В классическом подходе, в создание которого значительный вклад внес Виктор Михайлович Глушков, при построении сложного дискретного устройства выделяется несколько этапов.

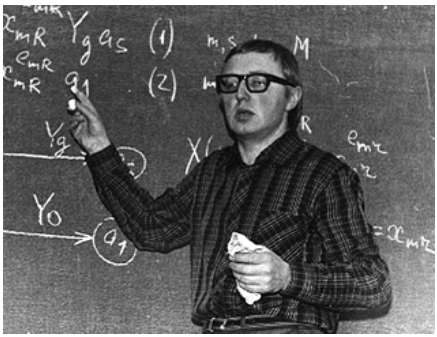
Виктор Михайлович Глушков (1923–1982)

На первом этапе, называемом этапом *блочного синтеза*, осуществляется разбиение схемы автомата на отдельные более простые блоки-автоматы и определяется общий план обмена данными между этими блоками.

На втором этапе, который называется этапом *абстрактного синтеза*, строится алгоритм блоков в виде конечного автомата, выбирается форма описания, определяются функции-состояния без привязки к реализующим автомат элементам.

Третий этап – *структурный синтез*, выбор логических и запоминающих элементов (триггеров, И-НЕ сборок и т. п.) для построения спроектированных блоков, и на основе специальных формальных методик – так называемых «канонических уравнений» – общая задача синтеза автомата-блока сводится к синтезу схем из элементов дискретного действия, не обладающих памятью.

Следующим, четвертым этапом является синтез этих последних схем, называемый обычно *комбинационным синтезом*.



Профессор С. И. Баранов читает лекцию по теории автоматов

Наконец, на пятом и последнем этапе производятся преобразования и дополнения к спроектированным схемам с целью обеспечения надежности и устойчивости их функционирования. В частности, меры по исключению так называемых гонок (конфликтной рассинхронизации работы алгоритма, возникающей из-за параллельности исполняемых операций в логических элементах) – чрезвычайно неприятных и широко распространенных при проектировании электронных схем на «рассыпухе».

В результате теоретических исследований выяснилось, что очень хорошо поддаются формализации третий и четвертый этапы проектирования, частично второй этап. Удовлетворительную формализацию первого и второго этапа получить не удалось. Отсутствие формальных методик в паре с экспоненциальным ростом объема работы при линейном росте размерности объекта (размерности входной / выходной переменных) существенно ограничивали сложность автоматизированных объектов.

Вопросы для самопроверки

Вопрос 1. Что такое $s(t)$, при $t = 0$ в автомате Мили? В автомате Мура?

Вопрос 2. Чему равно $o(t)$, при $t = 0$ в автомате Мили? В автомате Мура?

Вопрос 3. Определить число ячеек таблицы при табличном задании автомата Мили при числе состояний 10, при 8-битовых входной и выходной переменных x, y?

Вопрос 4. Какие из этапов синтеза конечного автомата нетипичны для современных методик программирования? Какие проблемы сходны?

2. Специфика задач промышленной автоматизации

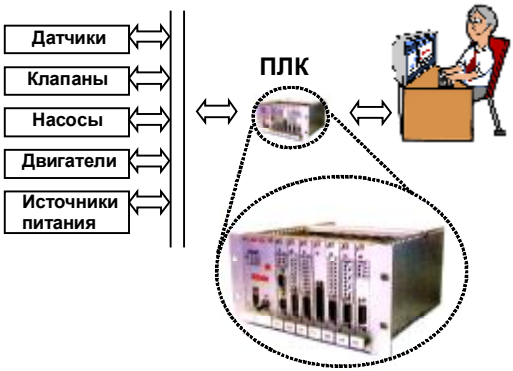


Рис. 2.1. Программируемый логический контроллер – ядро современной системы управления

В настоящее время практически вся промышленная автоматизация реализуется на цифровых системах управления. В качестве базового элемента систем управления используются программируемые логические контроллеры (ПЛК). ПЛК имеют в своем составе микропроцессорный модуль, множество входов и выходов, и *программное обеспечение* – управляющий алгоритм (УА), который реализует разнообразные функции: контролирует значения входных / выходных сигналов, выполняет логические / арифметические операции, осуществляет регулирование, обеспечивает связь с датчиками и исполнительными органами (клапанами, насосами, двигателями, источниками питания, регуляторами). ПЛК могут работать автономно в полностью автоматических системах, или же такая базовая схема может усложняться. Например, ПЛК могут подключаться к автоматизированному рабочему месту (АРМ) оператора для осуществления супервизорного управления или к базам данных для накопления информации и интеграции в автоматизированную систему управления (АСУ) предприятия. Через датчики в ПЛК поступает информация о текущем состоянии управляемого объекта, а через исполнительные органы ПЛК может воздействовать на него – управлять технологическим процессом.

Аркадий Дмитриевич Закревский

Поскольку все ПЛК строятся на базе цифровой техники, естественным образом предполагаются некоторые языковые средства их программирования. Из-за специфики задачи алгоритмические языки программирования, такие как Си, Паскаль, Си++ не годятся для этих целей. Анализ специфики задач промышленной автоматизации, в результате которого эти задачи удалось выделить в

отдельный класс, был проведен Аркадием Дмитриевичем Закревским при разработке специализированного языка ПРАЛУ.

Открытость, цикличность и событийность. Существенное отличие задач автоматизации от задач для персонального компьютера – это *открытость*, наличие внешней (по отношению к системе управления) среды – *управляемого объекта* – и активное воздействие на нее через органы управления. Самый распространенный случай управляемого объекта – техническая система, реализующая некоторую производственную технологию. Воздействия – или, другими словами, *реакция* системы управления, – определяется алгоритмом управления в зависимости от *событий* на объекте управления, информация о которых поступает через датчики обратной связи. Для цифровых систем это обстоятельство обусловливает *цикличность* управляющего алгоритма по схеме, уже обсуждаемой ранее: «считывание состояния входных сигналов через датчики» – «их обработка и формирование выходных сигналов» – «выдача выходных сигналов на исполнительные органы». Открытость и устойчивое функционирование механизмов во внешней среде изучаются в гомеостатике, свойства событийности исследуются в теории реагирующих систем.

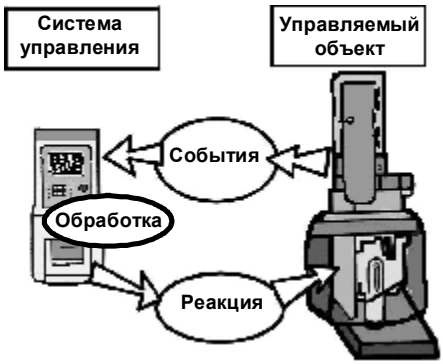


Рис. 2.2. Открытость, событийность, цикличность в задачах управления

Синхронизм и временные интервалы. Управляемый объект принадлежит физическому миру, подчиняется объективным законам физики, химии, биологии. Если, скажем, автоматизируется рост дрожжевых палочек в автоклаве и для выращивания партии дрожжей требуется 250 кВт · час, то эта энергия должна подаваться в систему равномерно и в течение длительного времени, около двух суток. Наивная попытка вкачать эту энергию как можно быстрее приведет к тому, что микроорганизмы погибнут от перегрева. Механические элементы конструкции, например клапаны, не могут отработать управляющий сигнал мгновенно. Для передачи пакета данных требуется время, зависящее от длины пакета и скорости передачи. Увеличив силу тока на массивном нагревательном элементе, ошибочно ожидать мгновенного установления равновесного режима по температуре. Иными словами, существуют вполне объективные временны́е требования к порядку выдачи управляющих воздействий. Таким образом, алгоритм

управления предполагает *синхронизацию* своего исполнения с физическими процессами во внешней среде. Это обусловливает необходимость развитой службы времени и активную работу с временными объектами: задержками, паузами, тайм-аутами. Проблемами синхронизации иногда занимаются в рамках так называемых систем «реального» времени.

Примечание. *Вообще в большинстве случаев, когда разговор заходит о «реальном времени», подразумевается решение проблемы нехватки вычислительных ресурсов и выбора алгоритма планирования для организации исполнения нескольких независимых задач на одном процессоре. Иногда это словосочетание и вовсе не имеет четкого технического смысла, используется чисто в маркетинговых целях, для позиционирования продукта на рынке. Например, достаточно часто встречается слоган «ОС QNX – система реального времени».*

Логический параллелизм. Другая характерная особенность алгоритмов управления – *логический параллелизм*, отражающий существование множества параллельно протекающих процессов в объекте управления. Регулирование температуры, давления, скоростей перемещения и вращения рабочих органов технологической системы по определению допускают и, более того, предполагают параллельность в работе алгоритма управления. Поскольку события, происходящие в различных компонентах системы, возникают независимо и в произвольной последовательности, то попытка задать реакцию системы единым блоком означает комбинаторный перебор большого числа вариантов и неоправданный рост сложности описания. Логический параллелизм предполагает наличие в алгоритме управления независимых или слабозависимых частей.

Следует сразу отметить, что параллелизм алгоритмов управления отличается от параллелизма высокопроизводительных многопроцессорных или многомашинных вычислительных систем. Параллелизм высокопроизводительных систем, или так называемый *физический параллелизм*, направлен на сокращение времени получения результата некоторого вычислительного алгоритма и исследуется в рамках параллельного программирования. Для систем автоматического управления получение результата в минимальные сроки в подавляющем большинстве практических случаев неактуальна (см. выше «синхронизм и временны́е интервалы»), параллелизм используется для того, чтобы упростить логическую структуру алгоритма управления.

Сложность и человеческий фактор. Механизмы структуризации и абстрагирования. Поскольку программы пишутся человеком и исключительно для человека, то на процесс создания алгоритмов сильное влияние оказывает человеческий фактор: человеческая психика устанавливает естественные ограничения на сложность объектов, с которыми человек может работать. Поэтому при создании алгоритмов (в частности, и управ-

ляющих) языки программирования предусматривают различные средства, позволяющие упростить работу с программой. Среди основных средств такого плана следует указать *механизмы структуризации* алгоритма (в нашем случае – средства описания и организации совместного функционирования логически параллельных частей) и *механизмы абстрагирования* (в нашем случае – понятийный переход от датчиков и исполнительных органов к целевому технологическому процессу). Иными словами, программа должна быть организована в виде обозримых, информационно-изолированных компонентов, возможно, иерархически вложенных друг в друга, и на некотором уровне иерархии программирование должно вестись в естественных терминах технологического процесса.

Степень соответствия модели конечного автомата специфике задач управления. Несмотря на то что модель конечного автомата достаточно привлекательна для решения простых задач автоматизации, лежит в основе целого ряда языков и подходов, изучается в физико-технических вузах и исторически разрабатывалась для решения задач логического управления, ряд серьезных недостатков ограничивает ее практическое использование в программировании. Главным образом недостатки обусловлены тем фактом, что специфика алгоритмов управления учтена в модели конечного автомата не в полной мере.

Обсудим эти недостатки, крайне усложняющие работу по описанию реальных алгоритмов управления.

Модель конечного автомата несет на себе груз идеи схемной реализации алгоритмов, причем реализации на базе микросхем малой степени интеграции. Поэтому попытка использовать в современных программных комплексах понятийный аппарат стандартных этапов синтеза автоматов (структурный синтез, комбинационный синтез, надежностный синтез), напрямую связанных с элементной базой реализации, приводит к низкоуровневому программированию – неэффективному, чреватому ошибками, а, значит, и недопустимому для использования на практике.

В модели конечного автомата отсутствуют операции работы со временем.

В рамках теории конечного автомата так и не удалось выработать эффективных и универсальных методов блочного синтеза автоматов, необходимых при конструировании реальных алгоритмов, предполагающих структурную компоновку алгоритма, активное использование логического параллелизма: дивергенции (расхождения) и конвергенции (схождения) потоков управления.

Концепция входного и выходного алфавита крайне затрудняет описание арифметических операций и делает невозможным использование операций сравнения, присваивания. Это ограничивает область применения МКА системами логического управления, которые крайне редко встречаются на практике в чистом виде. Концепция переменной выглядит в дан-

ном случае более предпочтительной в силу гибкости. Другой негативный момент теории автоматов – это сам формальный математический подход. Манипулирование нулями и единицами, не привязанное к семантике автоматизируемого процесса, провоцирует примитивные логические ошибки, крайне усложняет как процесс создания алгоритма, так и процесс его дальнейшего сопровождения (изучение, верификацию, модификацию, в частности, коррекцию обнаруживаемых ошибок).

Модель конечного автомата сложна для понимания современным выпускником школы или вуза, так как ее понятийный аппарат не соответствует текущим тенденциям в образовании, ориентированном на информационные технологии и тотальную компьютеризацию.

Перечисленные недостатки приводят к тому, что практическое применение модели конечного автомата в формате «as is» на современном этапе крайне неудобно в силу сложности и ненадежности. Создание программных управляющих систем характеризуется высокой трудоемкостью и отсутствием унифицированного подхода.

Чтобы создать теоретический базис для высоконадежных методов описания управляющих алгоритмов повышенной сложности, требуется адаптация модели конечного автомата к современным реалиям индустрии и образования, чему и посвящены следующие разделы нашего пособия.

Вопросы для самопроверки

Вопрос 1. Что такое «время»? (Сверьте свой ответ с определением из энциклопедического словаря)

Вопрос 2. Рис. 2.2 изображает ситуацию с точки зрения алгоритма управления. Как, по-вашему, должен выглядеть рисунок с точки зрения объекта управления?

Вопрос 3. Попытайтесь составить таблицу отличий для логического и физического параллелизма. Сравните их по целям использования, типовым аппаратным платформам, достигаемым при их использовании преимуществам, присущим недостаткам и характерным проблемам. Попытайтесь выделить основной отличительный признак для обоих случаев.

3. Гиперавтомат

3.1. Процесс и событийный полиморфизм

Вернемся к рассмотрению модернизированной модели конечного автомата, задаваемой пятеркой $A = (F, x, y, f_{cur}, f_1)$, и поговорим немного о функциях-состояниях. В этой модели мы определили $F = \{f_1, \ldots, f_k, \ldots, f_K\}$ как множество альтернативных функций-состояний, отображающих x в y ($y = fo_k(x)$, $fo_k(x) \in O$) и x в F ($f_{cur} = fs_k(x)$, $fs_k(x) \in F$). Сделаем следующее тривиальное замечание. Нет причин так жестко привязывать функции-состояния к уникальным переменным, как это делается в модели конечного автомата. Переменные для каждой функции-состояния могут быть уникальными. Таким образом, переменные могут привязываться исключительно к функции-состоянию. При этом функция-состояние может трактоваться исключительно в программистском смысле как обычная функция или процедура (функция, работающая с глобальными переменными). Сохраним наименование «функция-состояние» для отражения факта их альтернативности, переключаемости. Также в качестве возможных функций-состояний будем рассматривать и «пустые» функции, которые не производят никаких действий. Такие «пустые» функции, значение которых обсудим чуть позже, мы будем называть «пассивными», а все остальные – «активными».

Сделанные замечания позволяют ввести понятие «процесса». Процесс – это *полиморфная* функция, совокупность альтернативных функций-состояний, представляемых в программе как единая неделимая сущность. При вызове этой полиморфной функции (процесса) исполняется одна и только одна из составляющих его функций-состояний – текущая функция-состояние. Дополнительно к этому каждый процесс снабжен индивидуальными часами – счетчиком времени, который работает постоянно и обнуляется при смене текущей функции-состояния.

Таким образом, процесс p_i задается четверкой элементов, которые отражают статическую и динамическую информацию о процессе:

$$p_i = (F_i,\ f_i^1,\ f_i^{cur},\ T_i^{\text{p}}),$$

где F_i – множество функций-состояний процесса (как активных, так и пассивных); f_i^1 – начальная функция-состояние (активная функция), $f_i^1 \in F_i$; f_i^{cur} – текущая функция-состояние, $f_i^{cur} \in F_i$; T_i^{p} – текущее время нахождения процесса в текущей функции-состоянии, или текущее время отсутствия переходов.

Статика процесса определяется элементами F_i и f_i^1, для рассмотрения процесса в динамике дополнительно используются элементы f_i^{cur} и T_i^{p}.

3.2. Функция-состояние. События и реакция на событие

В свою очередь, *j*-я функция-состояние произвольного *i*-го процесса задается парой элементов:

$$f_i^{\,j} = (X_i^{\,j},\ Y_i^{\,j}),$$

где $X_i^{\,j} = \{x_i^{j1}, \ldots, x_i^{jL}\}$ – *множество событий* $f_i^{\,j}$; $Y_i^{\,j} = \{y_i^{j1}, \ldots, y_i^{jL}\}$ – *множество реакций* $f_i^{\,j}$.

В качестве *события* функции-состояния может рассматриваться произвольная суперпозиция фактов: значение текущей функции-состояния некоторого процесса, некоторое определенное значение переменной и некоторое определенное значение текущего времени отсутствия переходов процесса (оператор *timeout*).

Реакцией называется произвольная суперпозиция действий по изменению значений переменных и текущих функций-состояний процессов, определяемая на основе событий текущей функции.

Среди функций-состояний процесса F_i различаются взаимно непересекающиеся множества *активных и пассивных функций-состояний*, F_i^{a} и F_i^{p} соответственно. Функция-состояние процесса пассивна, если его множество реакций пустое:

$$f_i^{\,j} \in F_i^{\mathrm{p}} : X_i^{\,j} = \{\varnothing\}.$$

Среди пассивных функций-состояний выделены функции f^{NS} «нормальный останов» и f^{ES} «останов по ошибке», которые одинаковы для всех процессов.

3.3. Математическая модель гиперпроцесса (гипераавтомата)

Совокупность совместно функционирующих процессов образует гиперпроцесс (гипераавтомат) – надстройку над (гипер)процессами. Модель управляющего алгоритма в виде *гиперпроцесса* терминологически ориентирована на программную реализацию и позволяет отразить перечисленные свойства алгоритмов управления: открытость, событийность, цикличность, синхронизм и логический параллелизм.

Гиперпроцесс H представляет собой упорядоченный набор процессов, циклически активизируемых с периодом активизации T_H:

$$H = (T_H,\ P,\ p_1),$$

где T_H – *период активизации гиперпроцесса*; P – *множество процессов* ($P = \{p_1,\ p_2,\ \ldots,\ p_M\}$, где M – число процессов гиперпроцесса); p_1 – *начальный процесс*, $p_1 \in P$.

По запуску гиперпроцесса текущая функция-состояние выделенного процесса p_1 – начальное состояние, для всех остальных процессов текущее состояние – состояние нормального останова:

$$f_1^{cur} = f_1^1,$$
$$\forall i \neq 1 \Rightarrow f_i^{cur} = f^{NS}.$$

Пассивные функции служат для организации взаимодействия между процессами и начальной инициализации.

Циклическая схема функционирования гиперпроцесса (рис. 3.1) подобна схеме исполнения конечного автомата, и если не учитывать тот факт, что процессы слабосвязаны (влияют друг на друга), то единственные отличия – это множественность независимых блоков и тактируемость запуска циклов активизации (с периодом T_H).

Строго говоря, вопрос о том, как же активизируются процессы – физически параллельно или же последовательно, открытый, но для простоты

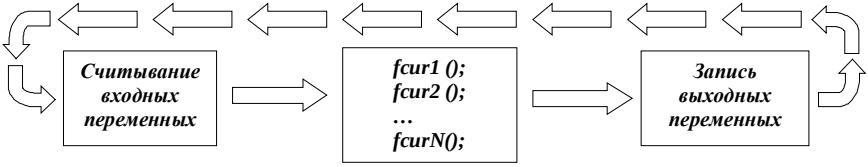


Рис. 3.1. Цикл гиперавтомата. Множественность обрабатывающих блоков

восприятия можно считать, что они активизируется последовательно, в порядке описания, как и показано на рис. 3.1. Заметим, что в этом случае гиперпроцесс представляет собой некоторый вариант кооперативной многозадачности, рассматриваемый в вузовских курсах при изучении операционных систем. В такой кооперативной многозадачности каждый процесс является по сути отдельным потоком управления, а сам гиперпроцесс – *структурой логического параллелизма с гранулярностью (параллелизма) на уровне нескольких инструкций (функций-состояний).*

Формальное описание функций-состояний. Функции-состояния описываются в стиле языка Си с помощью стандартных безусловных и условных *if-else* операторов произвольной вложенности. Например, для гипоте-

тического процесса p_3 описание начального состояния f_3^1 могло бы выглядеть следующим образом:

f_3^1 :

$y_1 = 1;$

if $(f_2^{cur} == f^{NS}) \rightarrow \{y_1 = 0; f_2^{cur} = f_2^1; f_3^{cur} = f_3^2;\}$

timeout $(3\ c) \rightarrow \{f_3^{cur} = f^{ES};\}$

Комментарий к коду. Переменной y_1 присваивается значение 1. Если текущая функция процесса p_2 – это функция нормального останова, то переменной y_1 присваивается значение 0. Процесс p_2 переводится в начальное состояние, а текущая функция-состояние процесса p_3 меняется на f_3^2. В случае отсутствия других событий через три секунды ($T_3^z > 3\ c$) наступает событие тайм-аута и процесс p_3 меняет свою текущую функцию-состояние на пассивную функцию «останов по ошибке».

Для большей наглядности записи используются следующие обозначения. Запуск процесса (установка его текущего состояния в начальное активное) обозначается через одноместный оператор start. Например, запись start f_2 эквивалентна записи $f_2^{cur} = f_2^1$. Одноместные операторы stop и error служат для обозначения перевода в пассивные состояния нормального останова и останова по ошибке соответственно:

stop $f_i^{cur} \equiv \{f_i^{cur} = f^{NS};\}$,

error $f_i^{cur} \equiv \{f_i^{cur} = f^{ES};\}$.

Ниже мы рассмотрим основные приемы, используемые при организации совместного функционирования процессов как единой совокупности – гиперпроцесса.

Организация совместного функционирования процессов. Поясним основные приемы работы с процессами. Поначалу ситуация всегда стандартна: активен только первый процесс, все другие пассивны. Это позволяет полностью контролировать начальное «разворачивание» алгоритма.

Перевод процесса в активное состояние – start p_i – по сути порождает отдельный поток управления. Перевод процесса в пассивное состояние – stop p_i или error p_i – закрывает поток управления, ассоциируемый с процессом. Хотя при этом формально процесс продолжает активизироваться.

Аналогом вызова процедуры служит следующая конструкция.

f_3^1 :

start p_2;

$f_3^{cur} = f_3^2$;

f_3^2 :

if $((f_2^{cur} == f^{NS}) \parallel (f_2^{cur} == f^{ES})) \rightarrow \{<$ *действия* $>\}$

... *etc*.

Комментарий к коду. Из процесса p_3 запускается процесс p_2. Производится переход в следующее состояние. Ожидается событие «переход процесса p_2 в пассивное состояние».

Соответственно сам процесс p_2 предполагает по окончании переход в пассивное состояние. Различение пассивных состояний позволяет организовать передачу некой примитивной информации о возможных причинах завершения: возникновении ошибок при выполнении.

f_2^1 :

< *действия* >

$f_2^{cur} = f_2^2$;

f_2^2 :

if $(x_{35} == 17) \rightarrow$ stop p_2;

else $\rightarrow$ error p_2;

Следует отметить, что при порождении нового потока управления порождающий поток управления не прекращает свою работу. Также при порождении новых потоков управления не возникают какие-либо отношения родственности между потоками управления. Последнее обстоятельство позволяет организовывать очень гибкие иерархические структуры.

Например, в любом процессе можно организовать дивергенцию потока управления произвольной степени. Последующая конвергенция также не вызывает проблем. Ниже приведен пример кода (гипотетический процесс p_3), порождающий два потока (процессы p_4 и p_5) с их последующей конвергенцией.

f_3^1 :

start p_4;

start p_5;

$f_3^{cur} = f_3^2$;

f_3^2 :

if $(((f_4^{cur} == f^{NS}) \parallel (f_4^{cur} == f^{ES}))$ &&

$((f_5^{cur} == f^{NS}) \parallel (f_5^{cur} == f^{ES}))) \rightarrow \{<$ *действия* $>\}$

... *etc*.

Вопросы для самопроверки

Вопрос 1. Какие недостатки с точки зрения восприятия присущи использованным тут формальным нотациям? Каким способом, по-вашему, это можно устранить?

Вопрос 2. Какие способы организации логического параллелизма вам известны?

Вопрос 3. Известно, что любые операции занимают время. Какие, на ваш взгляд, ограничения на ресурсоемкость алгоритма присущи модели гиперпроцесса?

Вопрос 4. Попытайтесь предложить способ для вычисления T_i^{p}. В качестве разминки ответьте, какие значения T_i^{p} будет принимать во время исполнения при предположении, что время отработки алгоритма много меньше периода активизации гиперпроцесса T_H?

4. Реализация гипераавтомата

4.1. Логический параллелизм

Рассмотрим следующий пример. Сначала распараллелим некоторую задачу, а затем выполним ее в рамках однопроцессорной архитектуры. С точки зрения цели физического параллелизма это выглядит как полный абсурд, так как время расчета увеличивается. Но есть ли у такого подхода положительные моменты? Несомненно. С точки зрения удобства программирования в таком распараллеливании легко обнаруживается глубокий смысл.

«Разделяй и властвуй»: старая поговорка имеет для программистов форму закона. Сложность современных программных систем может быть преодолена только путем разбиения проблемы на множество обособленных частей, и наоборот. Программные комплексы произвольной сложности могут строиться только на основе независимых или, в крайнем случае, слабо зависимых компонентов. А это также подразумевает максимальную обособленность частей с целью минимизации перекрестных связей, что необходимо для эффективной организации групповой разработки.

Выделение параллельных (а вернее, независимых) сущностей исходя из соображений удобства программирования – в первую очередь, с целью снизить сложность описания системы, – верный признак логического параллелизма.

Популярность операционных систем во многом обусловлена именно этим – возможностью конфигурирования функциональной среды отдельно взятого компьютера из набора параллельно исполняемых компонентов-задач. При вызове менеджер задач в любой многозадачной операционной системе покажет около десятка задач, выполняющих параллельно незаметную, но необходимую работу. В дополнение к этим задачам пользователь может запустить проигрыватель mp3-файлов, антивирусный мониторинг, редактор, почтовую программу и браузер… причем открытых для редактирования файлов и окон браузера может быть несколько. Попытка связать такую функциональность в целое и описать единым блоком однозначно приводит к комбинаторному взрыву сложности.

Логический параллелизм на уровне операционных систем называется многозадачным логическим параллелизмом. Планировщик – выделенная задача операционной системы – занимается переключением процессора с одной задачи на другую. Алгоритмы таких переключений могут быть как примитивными, так и весьма сложными. Самый простой – алгоритм разделения по времени. Планировщик работает с очередью задач и активизируется по прерываниям от таймера с некоторым заданным периодом (обычно это десятки миллисекунд). После активизации он сохраняет контекст те-

кущей задачи, ставит ее в конец очереди, восстанавливает контекст следующей задачи из очереди и запускает ее на исполнение.

Поскольку задачи различны по исполняемым функциям и ресурсоемкости, возникает проблема, как распределить вычислительную мощность процессора соразмерно задачам. В этом случае базовый алгоритм может усложняться. Например, механизмом приоритетов, когда задачам присваиваются приоритеты и при смене задач управление передается наиболее приоритетной. В системе может быть предусмотрено динамическое изменение приоритетов. Планировщик может вызываться по прерываниям не только от таймера, но и от других источников прерываний. Вызов планировщика может быть предусмотрен и по требованию активной задачи, после выполнения необходимых высокоприоритетных операций. Однако появление гибкости сопровождается повышением накладных расходов на планирование, сильно зависящих от числа обслуживаемых задач.

Преимущества мелкозернистого логического параллелизма. Задача с уникальным набором данных и функций достаточно тяжеловесна для вызова и обслуживания. Многозадачный логический параллелизм предполагает небольшое количество задач (обычно в пределах одного десятка).

Между тем существуют области, в которых необходим мелкозернистый логический параллелизм (разбиение задачи на большое число мелких независимо исполняемых частей). В серверах баз данных, новостных интернет-порталах, мобильных сервисах, обслуживающих распределенные системы из нескольких сотен тысяч абонентов, требуется механизм независимого обслуживания тысяч одновременно поступающих запросов. В таких системах надо не только не пропустить поступивший запрос, но и обслужить его, возможно, организовав интерактивный диалог, контроль достоверности, перезапрос информации в случае ошибки.

Мелкозернистый логический параллелизм также чрезвычайно эффективен в задачах управления сложным промышленным оборудованием. Необходимость отражать параллелизм процессов, протекающих на объекте управления, требует создавать по меньшей мере сотни параллельно исполняемых процессов. Например, алгоритмически эквивалентное «монолитное» решение в виде одного процесса (конечного автомата) для практической задачи управления выращиванием монокристаллического кремния приводит к рассмотрению 10^{30} (sic!) различных ситуаций.

Сжатие информации достигается при логическом параллелизме за счет упрощения описания комбинаций независимых случаев: описание алгоритма как набора независимых частей – это описание суммы ситуаций, в то время как монолитное описание – это описание произведения ситуаций.

Редукция сложности за счет распараллеливания выражается следующей формулой:

$$R = \frac{\prod_{i=1}^{L} N_i}{\sum_{i=1}^{L} N_i}$$ (L – число независимых участков кода, N_i – число состояний).

Невообразимое число ситуаций (10^{30}) может быть получено всего лишь из сотни параллельно исполняемых процессов с двумя состояниями, в то время как независимое описание алгоритма – это рассмотрение всего 200 уникальных случаев.

Многопоточный логический параллелизм. Очевидные преимущества мелкозернистого логического параллелизма заставляют искать пути снижения накладных расходов, присущих многозадачности. Известные решения для операционных систем – это легковесные процессы (light-weight process), – например, в Sun OS 5.x, – облегченный вариант задач, и потоки (thread), состояние которых полностью характеризуется значениями регистров-указателей на код и стек. Переключение процессора на поток минимизировано, таким образом, до операций сохранения / восстановления этих указателей. Планировщик по-прежнему присутствует и активизируется по прерываниям от таймера.

Необходимо отметить, что переключение потоков по времени не всегда удобно и приводит к непроизводительным простоям процессора. Если в некотором потоке для продолжения вычислений требуется внешнее событие (отклик оборудования, реакция абонента на запрос, подтверждение успешной передачи сообщения), то поток честно тратит отведенное ему время на ожидание:

```
Step1();                    /* действие */
while (!EventOnStep1());    /* ожидание реакции на действие */
Step2();                    /* следующее действие */
```

Если для появления реакции на действие требуется значительное время (например, событие – это реакция пользователя на запрос системы), то более приемлемой стратегий была бы передача управления после опроса.

Этот подход используется при организации многопоточности методом циклического опроса (round-robin), иногда обозначаемом термином «кооперативная многозадачность». При циклическом опросе потоки могут быть представлены просто функциями, которые опрашиваются в бесконечном цикле:

```
main () {
    for (;;) {
        thread_1();
        thread_2();
        …
```

```
                thread_N();
        }
    }
```

Необходимость ожидания внешней реакции естественным образом задает разбиение потоков на части. Каждая из таких частей является простейшей функцией. Если присвоить каждой из таких частей число, то реализация потоков естественным образом выражается в терминах языка Си. При этом неожиданно выясняется, что такая реализация не больше и не меньше, чем реализация конечного автомата, в части организации работы с состояниями:

```
thread_i () {
  switch (State_i) {
  …
  case STATE_1:
      Step1();                                    /* действие */
      if (EventOnStep1()) State_i = STATE_2; /* смена состояния */
  break;
  case STATE_2:
      Step2();                              /* следующее действие */
      …
  }
}
```

Чтобы убедиться, что приведенный участок кода соответствует классическому конечному автомату, заключите его в бесконечный цикл **for (;;) { }**.

Таким образом, для многопоточности можно минимизировать или даже полностью исключить накладные расходы, присущие многозадачности. А простота организации делает многопоточность крайне привлекательной для организации мелкозернистого логического параллелизма.

К сожалению, получаемый выигрыш от многопоточности также связан с рисками потенциально возможных ошибок (которые могут возникнуть, например, из-за общей области данных). К счастью, эти риски могут быть радикальным образом снижены за счет специальных механизмов, которые мы обсудим чуть позже.

4.2. Использование процедурных языков

Описанный подход является классическим при программной реализации конечного автомата на процедурных языках. В этом подходе организуется бесконечный цикл вызова конструкции, соответствующей конечному автомату. Конкретизируем запись автомата Мура A_2, заданного таблицей 1.5, и вместо абстрактных идентификаторов подставим конкретные числовые значения (табл. 4.1). (Как мы помним, в силу соображений,

приведенных в п. 1.3, эта подстановка дает эквивалентный автомат, в качестве числовых значений использованы индексы при идентификаторах.)

Таблица 4.1. Численная конкретизация автомата Мура A_2

y	1	1	2
s / x	1	2	3
1 2	2 3	1 2	1 2

Опишем приведенный в таблице алгоритм на языке Си-файла.

```
main () {
      BYTE x, y, s;
      s = 1;
      for (;;) {
          Input(x);
          switch (s) {
                case 1:
                          y = 1;
                          switch (x) {
                            case 1: s = 2; break;
                            case 2: s = 3; break;
                          }

                break;
                case 2:
                          y = 1;
                          switch (x) {
                            case 1: s = 1; break;
                            case 2: s = 2; break;
                          }
                break;
                case 3:
                          y = 2;
                          switch (x) {
                            case 1: s = 1; break;
                            case 2: s = 2; break;
                          }
                break;
          }
          Output(y);
      }
}.
```

Можно построчно убедиться, что приведенный алгоритм полностью идентичен заданному в табл. 4.1.

Чего не хватает для реализации гипераавтомата «в лоб»? Во-первых, автомат только один, а, во-вторых, нет службы времени. Первая проблема решается просто: дополнительные автоматы (процессы) означают дублирование переменных **x**, **y**, **s**. Но как быть со службой времени? Поступим следующим образом. Введем функцию инициализации таймера **InitTimer (int T)**, которая будет настраивать таймер на генерацию прерываний в заданное параметром **T** число миллисекунд, а в процедуре обработки прерывания инкрементировать переменную **InterruptFlag**. Таким образом, тактирование будет обеспечиваться следующей конструкцией.

```
for (;;) {
    if (InterruptFlag) {
        InterruptFlag = 0;
        <… тело алгоритма …>
    }
}.
```

Далее встает вопрос о текущем времени отсутствия переходов T_i^{p} для каждого процесса. Выход очевиден: для каждого процесса заводим дополнительную переменную $\mathbf{t_i}$, которую будем инкрементировать при каждом вызове и обнулять при каждом переходе.

Для краткости приведем фрагмент:

```
main () {
    BYTE x1, y1, s1, t1;        /*первый процесс*/
    BYTE x2, y2, s2, t2;        /*второй процесс*/
    …
    BYTE x<nn>, y<nn>, s<nn>, t<nn>; /*nn-й  процесс*/
    InitTimer(10);     /*инициализация прерываний */
    s1 = 1;          /*инициализация ячеек памяти состояний */
    s2 = 0;
    …
    s<nn> = 0;
    for (;;) {
    if (InterruptFlag) {          /* тактирование */
        InterruptFlag = 0;
        Input(x1); /* считывание входных переменных */
        Input(x2);
        …
        Input(x<nn>);
        switch (s1) {             /* обработка процесса 1*/
            case 0: break;                /* пассивное состояние */
            case 1:
                y1 = 1; t1++;
                switch (x1) {
```

```
                    case 1:
                        s2 = 1;          /* start p2 (запуск p2) */
                        s1 = 2; /* изменение своего состояния */
                        t1 = 0;          /* сброс счетчика */
                    break;
                    case 2: s1 = 3; t1 = 0; break;
                }

            break;
            case 2:
            <…другие состояния…>
        }

        switch (s2) {           /* обработка процесса 2*/
            case 0: break;          /* пассивное состояние */
            case 1:
                    y2 = 1; t2++;
                    switch (x2) {
                      case 1: s2 = 2; t2 = 0; break;
                      case 2: s2 = 3; t2 = 0; break;
                    }

            break;
            case 2:
            <…другие состояния…>
        }
        <… другие процессы …>
        Output(y1);   /* запись выходных переменных */
        Output(y2);
        …
        Output(y<nn>);
    }
    }
}.
```

Составление списка недостатков для приведенного способа реализации автомата и гипераавтомата не представляет особого труда. Их много и они лежат на поверхности. Недостатки связаны с многочисленными рутинными операциями, прямой работой с числами и операциями с большим количеством глобальных переменных. Программы, написанные таким способом, чрезвычайно ненадежны, их составление трудоемко, их дальнейшая модификация практически исключена. При программировании происходит полный разрыв со смыслом задачи, остаются неясными вопросы интерфейса с УСО, реализация допускает различные подходы, что, опять же, затрудняет чтение алгоритмов, написанных в этом стиле. Верификация алгоритма даже с помощью специально разработанных методик не гарантирует отсутствия ошибок. Более того, при проверке экспоненциально рас-

тущих вариантов для сколько-нибудь серьезной программы практически невозможно не только проанализировать лог-файлы, но и вообще получить таковые.

Чтобы лучше прочувствовать методику программирования «в лоб», используем приведенный способ и опишем процесс открытия отсечного вакуумного клапана.

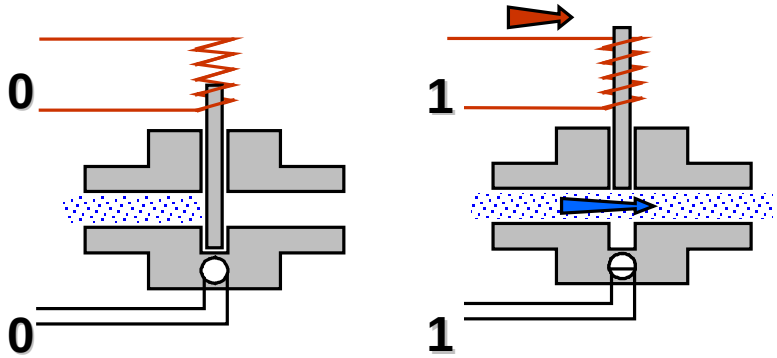


Рис. 4.1. Работа клапана. Слева – клапан в закрытом состоянии, справа – состояние после нормального включения (открытия)

Постановка задачи. Имеется вакуумный клапан, который имеет один управляющий вход (для системы управления – это выходной сигнал **y**) и один сигнал положения (для системы управления – это входной сигнал **x**). В исходном закрытом состоянии значения обоих сигналов – 0 (рис. 4.1). При подаче на вход клапана единицы клапан должен открыться в течение 0,5 секунды (сигнал положения должен принять значение 1). В противном случае диагностируется отказ клапана.

```
main () {
    BYTE x, y, s, t ;
    InitTimer(10);     /* инициализация прерываний 100 Гц */
    s = 1; t = 0;
    for (;;) {
    if (InterruptFlag) {          /* тактирование */
        InterruptFlag = 0;

        Input(x);       /* чтение входа */
        t++;            /* инкрементируем здесь */
        switch (s) {
            case 1:
                    y = 1; /* выдаем управляющий сигнал */
                    s = 2; /* переход в следующее состояние */
```

```
                t = 0;  /* обнуляем счетчик времени */
            break;
            case 2:
                /* или срабатывает сигнал: */
                if (x) { printf ("\n Done!"); s = 3; t = 0;}
                /* или тайм-аут 0,5 сек: */
                if (t > 50) { printf ("\n Error!"); s = 4; t = 0;}
            break;
            case 3: break;
            case 4: break;
        }
        Output(y);
    }
    }
}.
```

В заключение обсудим еще один важный момент предложенной реализации гипераутомата. Кстати, этот момент мы затрагивали в вопросах для самопроверки в предыдущем параграфе. Речь идет о третьем вопросе – вопросе о ресурсоемкости алгоритма. Дело в том, что когда время, требуемое для обсчета цикла гипераутомата, превышает период активизации, наступает очевидное логическое противоречие, связанное с нехваткой ресурсов: необходимость начать новый цикл возникает до того, как предыдущий цикл был обсчитан. Это обстоятельство предполагает выполнение некоторых условий на ресурсоемкость алгоритма. Проблема эта, разумеется, не нова. Например, для решения аналогичных проблем в рамках проекта по языку Esterel была выдвинута *гипотеза идеального синхронизма* (perfect synchrony hypothesis). Гипотеза устанавливает, что в рамках модели языка Esterel накладные расходы на вычисления и связь между компонентами равны нулю. Разумеется, в рамках гипераутомата можно снизить требование этой гипотезы до разумного уровня и переформулировать ее следующим образом. Время, затрачиваемое на организацию и выполнение алгоритма при реализации модели гипераутомата, не должно превышать периода активизации гипераутомата T_H.

Вопросы для самопроверки

Вопрос 1. Какое число состояний возникнет при «монолитном» рассмотрении 10 независимых алгоритмов с четырьмя состояниями в каждом?

Вопрос 2. Предложите альтернативный способ организации тактированного циклического исполнения гипераутомата.

Вопрос 3. Какие методы могут быть использованы в рамках процедурных языков для исключения недостатков реализации конечного автомата и гипераутомата «в лоб»?

4.3. Языки стандарта МЭК 61131-3 и возможные альтернативы

В 1993 г. Международная электротехническая комиссия выпустила в свет третью часть стандарта МЭК 61131-3. Этот международный стандарт входит в группу МЭК 61131-стандартов, которые охватывают различные аспекты использования программируемых логических контроллеров, т. е. предназначены для программирования информационно-управляющих систем. Декларируемые цели МЭК 61131-3 – стандартизация существующих языков ПЛК, а вернее, базовая платформа для такой работы в национальных комитетах стандартизации. Кратко остановимся на особенностях стандарта, затем проанализируем языки стандарта на соответствие специфике задачи и обсудим приемы создания с их помощью управляющих алгоритмов, близких по свойствам к гипераутомату.

Международная электротехническая комиссия. Цели создания стандарта на языки программирования ПЛК

Международная электротехническая комиссия – это международный орган стандартизации, создающий базовые стандарты для последующей адаптации в национальных комитетах. Интересный факт, которым могут гордиться граждане России: в создании и работе этой комиссии принимал активное участие СССР, поэтому русский – один из трех официальных языков МЭК.

Что касается стандартизации языков, используемых для программирования ПЛК, то эта проблема назрела давно. К концу 1980-х гг. несколько базовых концепций на практике были представлены сотней вариаций. Их унификация сулила ощутимый экономический эффект. Для решения этой проблемы была создана рабочая группа, состоящая из представителей ведущих игроков на рынке автоматизации, которая начала работу.

В силу того что общепринятого подхода к программированию ПЛК не существовало (и сейчас не существует), членам комиссии договориться о едином языке не удалось. Поэтому было принято компромиссное решение – включить в стандарт языки, используемые в фирмах, представителям которых посчастливилось оказаться членами комиссии.

Среди языков-«счастливцев» оказались:

1. **SFC** (Sequential Function Chart) – графический язык, используемый для описания алгоритма в виде набора связанных пар: шагов (step) и переходов (transition). Шаг – набор операций над переменными, переход – набор логических условных выражений, определяющий передачу управления к следующей паре шаг-переход. По внешнему виду описание на языке SFC напоминает хорошо известные логические блок-схемы алгоритмов, хотя идеологически SFC близок к сетям Петри. SFC имеет возможность распараллеливания алгоритма, однако не имеет средств для описания ша-

гов и переходов, которые могут быть выражены только средствами других языков стандарта. Происхождение: язык Grafcet фирмы Telemechanique-Groupe Schneider. Пример кода, созданного с использованием языка SFC, приведен на рис. 4.2.

Рис. 4.2. Пример кода на языке SFC

2. **LD** (Ladder Diagram) – графический язык, стандартизованный вариант класса языков релейно-контактных схем (РКС). Основной выразительный элемент – «реле». Реле представлено парой: обмотка (аналог выходной переменной) и контакт (аналог входной переменной). Контакты и обмотки между собой и с левой, и с правой силовой шиной соединены горизонтальными линиями («проводниками»). РКС были очень мощным средством автоматизации в докомпьютерную эпоху. Сильнейшие школы теоретиков автоматизации на базе РКС существовали в СССР, одна из них – школа Михаила Александровича Гаврилова. РКС были настолько популярны в автоматизации, что с появлением ПЛК идеи, заложенные в РКС, были реализованы в виде специализированного языка программирования. Это оказалось очень удобно при внедрении ПЛК: старые кадры очень быстро осваивали язык программирования, поскольку концептуальный уровень (метафора реле) остался прежним.

Михаил Александрович Гаврилов (1903–1979)

Ввиду своих ограниченных возможностей язык дополнен привнесенными средствами: таймерами, счетчиками и т. п. Происхождение: различные варианты языка релейно-контактных схем, поддерживаемые в компаниях Allen-Bradley, AEG Schneider Automation, GE-Fanuc, Siemens. Пример кода, написанного на языке LD, приведен на рис. 4.3.

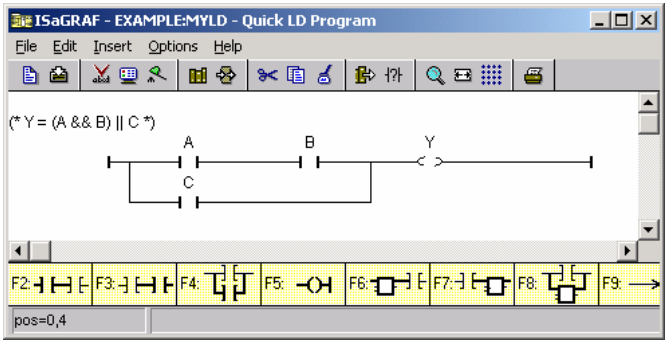


Рис. 4.3. Пример кода на языке LD

3. **FBD** (Functional Block Diagram) – графический язык по своей сути похожий на LD: вместо реле в этом языке используются функциональные блоки. Алгоритм работы некоторого устройства, выраженный средствами этого языке, напоминает функциональную схему электронного устройства: элементы типа «логическое И», «логическое ИЛИ» и т. п., соединенные линиями (рис. 4.4). Корни языка выяснить сложно, однако большинство специалистов сходятся во мнении, что это язык, подобный LD, но использующий в качестве метафоры другую элементную базу.

4. **ST** (Structured Text) – текстовый высокоуровневый язык общего назначения, по синтаксису ориентированный на Паскаль (рис. 4.5). Самостоятельного значения не имеет; используется совместно с SFC. Происхождение: язык Паскаль, язык Grafcet фирмы Telemechanique-Groupe Schneider.

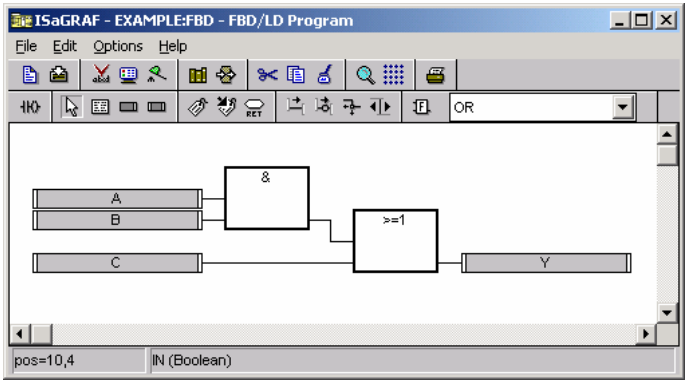


Рис. 4.4. Пример кода на языке FBD

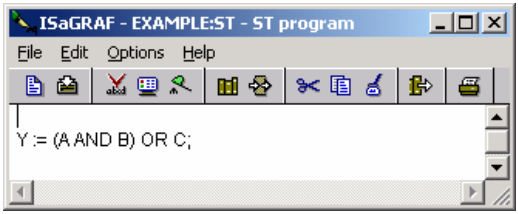


Рис. 4.5. Пример кода на языке ST

5. **IL** (Instruction List) – текстовый язык низкого уровня. Выглядит как язык Ассемблера (рис. 4.6), что объясняется его происхождением: для некоторых моделей ПЛК фирмы Siemens является языком Ассемблера. В рамках стандарта IEC 1131-3 он не привязан к архитектуре конкретного процессора. Самостоятельного значения не имеет: используется совместно с SFC. Происхождение – язык STEP 5 фирмы Siemens.

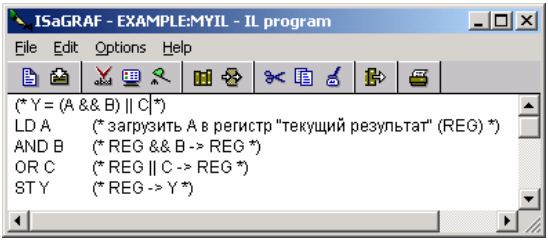


Рис. 4.6. Пример кода на языке IL

Основные недостатки МЭК 61131-3. Мифологемы стандарта. При анализе стандарта выявляются некоторые моменты, которые следует сразу обсудить.

Во-первых, имеется перекос в выборе языков. *С одной стороны*, в стандарт попадает ассемблер (IL), *с другой стороны*, чрезвычайно мощная ветка так называемых языков машин конечных состояний FSM оказывается за рамками рассмотрения.

Во-вторых, некорректным и дезориентирующим пользователей решением МЭК стала группировка языков в едином стандарте, что в корне отличается от общепринятых подходов к стандартизации языков программирования (см., например, стандарты ANSI C, Ada и т. д.).

В-третьих, стандарт исключает из рассмотрения вопросы унифицированного представления графических языков стандарта, что автоматически означает проблемы совместимости продуктов разных производителей. Эта особенность стандарта сводит практически на нет выгоды от его использо-

вания. Кроме того, в стандарте МЭК61131-3 не рассматривается вопрос привязки алгоритма к интерфейсной аппаратуре, которая с необходимостью присутствует в любой системе управления.

Вокруг стандарта МЭК 61131-3 возникло несколько мифологем, основные из которых рассмотрены ниже.

Мифологема 1. Для программирования ПЛК допускается применять только языки стандарта МЭК 61131-3.

Комментарий. Это неверно. Для программирования ПЛК могут применяться произвольные языки, естественно, наиболее адекватные решаемой задаче. Более того, во многих (если не в большинстве) средств программирования, ориентированных на МЭК 61131-3, язык Си – это, по сути, шестой язык программирования. Многие ПЛК по-прежнему программируются и языками класса FSM, и на языках Си / Си++.

Мифологема 2. Среда разработки МЭК 61131-3 должна обеспечивать возможность программирования на всех пяти языках.

Комментарий. Это неверно. Языки стандарта независимы, и разработчиками стандарта предполагалось, что средство программирования МЭК 61131-3 поддерживает только один язык из набора. Более того, *мультиязыковые* программы расцениваются как потенциально ненадежные и затратные в сопровождении. На практике нарушение принципа языковой однородности программ всегда вынужденно и просто отражает ограничения, присущие языкам МЭК 61131-3.

Мифологема 3. МЭК 61131-3 обеспечивает кросс-брендовую и кросс-платформенную переносимость пользовательских программ.

Комментарий. Это неверно. Стандарт специфицирует лишь внешний вид, синтаксис языков, что недостаточно для обеспечения переносимости графических языков (для текстовых – достаточно). Выбрав фирму-поставщика, пользователь с большой вероятностью навсегда связывает с ней свою судьбу. Также стандарт допускает создание несовместимых вариаций и регламентирует лишь наличие списка несоответствий стандарту.

Несомненно, перечисленные недостатки и дезориентирующие мифологемы на руку мегакорпорациям, так как крайне затрудняет передел сложившегося рынка и появление на нем новых игроков. Существуют и другие, более жесткие, мнения, в которых стандарт МЭК 61131-3 прямо называется контрпродуктивным. Не останавливаясь подробно на этом аспекте, можно посоветовать интересующимся статью об использовании стандартизации в конкурентной борьбе [6].

PLCopen. Попытка адаптировать стандарт для целей сообщества конечных пользователей

Обеспечение кросс-брендовой переносимости и реальная стандартизация языков программирования ПЛК весьма актуальны для индустрии. Функции сертификации и развития стандарта взяла на себя независимая

организация PLCopen [www.plcopen.org], которая попыталась адаптировать стандарт МЭК 61131-3 к нуждам конечных пользователей.

PLCopen объединила средних и мелких игроков на рынке ПЛК, системных интеграторов и пользователей. Внутри PLCopen долгое время действовала группа по разработке независимого формата, обеспечивающего кросс-брендовую переносимость пользовательских программ. Была развернута активность по формированию и продвижению корректировок стандарта, отвечающих нуждам сообщества. Была разработана процедура и организована работа экспертных комиссий по сертификации CASE-средств, ориентированных на стандарт.

К сожалению, базовые подходы, использованные при разработке стандарта, оказали серьезное сопротивление деятельности PLCopen. Как следствие, работы по созданию переносимых программ были практически прекращены, предполагаемый формат, разрабатываемый для обеспечения переноса программ – FxF (File-eXchange-Format), – не получил поддержки. Наиболее значимым результатом по корректировке стандарта явился, пожалуй, ввод локальных переменных. Сертификация CASE-средств ограничивается языками ST и IL (текстовыми языками, не имеющими самостоятельного значения). В силу этого обстоятельства проблема переносимости программ МЭК 61131-3 остается по-прежнему нерешенной. К большому сожалению, скептические прогнозы относительно будущего стандарта МЭК 61131-3, сделанные при его появлении рядом специалистов, стали реальностью.

Прагматика языков МЭК 61131-3. Выбор языка под конкретную задачу. Несмотря на недостатки, существуют вполне определенные ситуации, когда языки МЭК 61131-3 могут быть использованы на практике. В относительно простых задачах, не предъявляющих строгих требований по надежности, языки МЭК могут оказаться экономически эффективными.

Даже в ситуации, когда языки МЭК слабо подходят для практической задачи, отказ от их использования совсем не очевиден. В первую очередь, это вызвано тем обстоятельством, что конечному пользователю или системному интегратору тяжело конкурировать с мегакорпорациями, разрабатывать и поддерживать альтернативные решения. Ведь, кроме собственно языка, в среду разработки входит набор вспомогательных программ и библиотек, существенно облегчающих работу по тестированию и настройке системы. Поэтому, несмотря на недостатки, языки МЭК 61131-3 вполне допустимо использовать. Но при выборе языка реализации проекта требуется учитывать особенности языков стандарта.

Ниже рассмотрены характеристики языков МЭК 61131-3 с точки зрения прагматики их использования и даны некоторые рекомендации по выбору.

Ladder Diagram (LD). Метафора реле, лежащая в основе концепции, графическая форма описания алгоритма позволяет легко освоить язык не-

профессионалу. При переходе на ПЛК язык обладал вполне объяснимыми преимуществами, так как снимал психологические проблемы переучивания персонала. Отладка логической программы производится естественным образом с использованием исходной записи. Как и любой метафорический язык, LD не обладает должной степенью универсальности, ориентирован на манипуляции с битовыми переменными, имеет ограничения на сложность описываемого алгоритма. Цикличность и логический параллелизм на уровне инструкций – неотъемлемые атрибуты LD.

Синхронизация, операции с аналоговыми сигналами, структуризация алгоритма вызывают большие проблемы. Абстрагирование и отход от ключевой метафоры реле невозможны. Событийность алгоритма может быть обеспечена только инородными для подхода средствами и чрезвычайно сложна в реализации.

Типовой пример использования – реализация аварийных блокировок системы, включающая преимущественно логические сигналы. Вполне приемлем в случаях, когда задачей предполагаются частные коррекции несложного логического алгоритма неквалифицированным (с точки зрения программирования) персоналом (ремонтники, механики и т. п.).

При необходимости реализовать поведенческий алгоритм применяется стандартный прием реализации конечного автомата. Для хранения значения текущего состояния заводится ячейка памяти. Выбор функции-состояния конечного автомата производится с помощью конструкции условного перехода по метке, которая позволяет игнорировать заданный участок кода. Функции-состояния, описанные в терминах реле, обрамляются конструкциями типа:

```
if (s != 2) goto Label_3;
    <код функции-состояния #2>
Label_3:
if (s != 3) goto Label_4;
    <код функции-состояния #3>
Label_4:
```

Functional Block Diagram (FBD) обладает характерным для метафорических языков преимуществом: легкостью начального изучения. Предоставляет достаточно естественную возможность работы с аналоговыми переменными и минимальные средства структуризации (новые функциональные блоки можно компоновать, используя уже существующие). Языку присущ логический параллелизм. Средства синхронизации достаточно естественны для языка.

Существенно меньшая по сравнению с LD наглядность при отладке программы. Проблемы с обработкой логических конструкций: средства управления потоком операций – конструкции типа if-then-else – отсутствуют среди выразительных средств языка, что ограничивает универсальность языка и часто приводит к появлению нестандартных библиотечных

блоков, создаваемых сторонними средствами, например на языке Си. Сильная степень структуризации алгоритма связана с ростом входных / выходных переменных функциональных блоков, что ограничивает сложность описываемых алгоритмов. Соответственно, ограничены и абстрагирующие свойства языка: информация о физических сигналах связи с объектом управления практически неустранима. Подход характеризуется отсутствием событийности.

Типовой пример использования – алгоритмы регулирования, обработка (например, фильтрация) аналоговых сигналов. В качестве пользователей предполагаются специалисты в области автоматического регулирования с привлечением квалифицированных системных программистов в сложных случаях.

При необходимости реализовать поведенческий алгоритм применяется прием, аналогичный используемому в случае LD. Для хранения значения текущего состояния заводится ячейка памяти и используется аналог **goto**. В случае FBD следует обращать особое внимание на расположение гипераватоматных блоков и меток, так как расположение команд на двумерном поле определяет порядок их выполнения. Смещение графического элемента на пару пикселей может привести к некорректной работе всего алгоритма. Поэтому для исключения подобных досадных и чрезвычайно трудных для обнаружения ошибок следует очень аккуратно формировать отступы между функциональными блоками, переходами и метками.

Sequential Function Chart (совместно с Structured Text или Instruction List). *SFC + ST* – это, пожалуй, наиболее мощное средство из состава языков МЭК 61131-3. Графика языка SFC облегчает изучение языка. Наличие общих корней с сетями Петри частично снимает проблемы синхронизации. Программирование операций с аналоговыми и логическими переменными достаточно комфортно за счет использования текстового Паскаль-подобного языка ST. Управление потоком команд не вызывает проблем. Существенное преимущество подхода – событийность, естественным образом поддерживаемая через механизм «шаг-переход». Отладка программ может быть облегчена визуальной трассировкой потока управления. Слабые места языка SFC (как и сетей Петри) – абстрагирование и структурирование, что, как и в предыдущих случаях, негативно сказывается на сложности программируемых алгоритмов и их качестве (сопровождаемости и надежности). По сравнению с LD и FBD подход SFC+ST существенно проигрывает в удобстве программирования параллелизма алгоритма, отличается крайне ненадежным механизмом дивергенции и конвергенции и, соответственно, более подходит для программирования линейных алгоритмических последовательностей.

Типовой пример использования – совмещенные алгоритмы логического и аналогового управления. В качестве пользователей предполагаются про-

граммирующие специалисты, совмещающие знание алгоритмических языков программирования и специфики автоматизируемого технологического процесса.

Аналог конечного автомата реализуется на SFC через механизм альтернативного выбора потока команд (через так называемый механизм дивергенции типа OR), что требует особой внимательности. Проблематична и организация параллелизма, которая необходима во многих практических задачах автоматизации. Особенно трудно в SFC организовать корректную конвергенцию потока управления. В качестве решения рекомендуется для каждого параллельного участка кода заводить флаг окончания операций – аналог пассивного состояния процесса. В этом случае конвергенцию следует контролировать, основываясь на проверке флагов всех потоков, подлежащих слиянию.

Пару *SFC + IL*, по-видимому, не стоит рекомендовать к использованию вообще. Такой экзотический вариант можно обосновать лишь в случае, когда в распоряжении имеется единственный программирующий специалист, знакомый только с ассемблером линейки ПЛК производства Siemens.

Следует добавить, что использование языков МЭК 61131-3 может обеспечить упрощение программирования и системную интеграцию, так как для имеющихся на рынке МЭК-средств, как правило, существуют более-менее апробированные решения, которые можно использовать в качестве прототипов вашей системы. С другой стороны, решение об использовании МЭК-средства имеет смысл предварять тщательным анализом требований задачи, в случае если вы столкнетесь с ограничениями, убедить производителя учесть в среде разработки специфику вашего проекта будет весьма непросто. В качестве начального пособия по изучению МЭК-средств настоятельно рекомендуется книга И. В. Петрова [5].

Возможные альтернативы. В качестве альтернативы языков стандарта МЭК 61131-3 можно попытаться использовать средства, встроенные в SCADA-системы. Такое решение вполне допустимо в некритичных задачах супервизорного контроля.

В большинстве случаев штатная эксплуатация систем управления не предусматривает наличие средств проектирования алгоритмов. Стандартный подход – это создание базового алгоритма с возможностью его настройки через ограниченное число доступных пользователю параметров. Это позволяет проектировать базовый алгоритм управления квалифицированными специалистами и адекватными языковыми средствами, а сопутствующую этому сложность «скрывать» за дружественным интерфейсом оператора, который создается, например, с помощью тех же SCADA-пакетов.

Аналогичный подход можно использовать в средах, ориентированных на «малую» автоматизацию, обслуживание небольшого физико-

технического эксперимента на базе Wintel. В качестве примера такого подхода укажем интегрированную среду LabVIEW компании National Instruments [www.ni.com]. использование встроенного графического языка G. Как и язык FBD стандарта МЭК 61131-3, язык G относится к классу data-flow-языков, в которых отсутствует конструкция управления потоком команд **if-then-else**. Его использование требует коренной смены приемов программирования и очень тяжело дается людям с опытом процедурного программирования. Однако в языке G есть специальные структуры **case** – аналог оператора табличного выбора **switch** в Си. Соответственно реализация конечного автомата не вызывает принципиальных затруднений.

В принципе допустимо решение об использовании для задач управления языков Си / Си++. Такой подход может быть оправдан при наличии штата квалифицированных специалистов, отлаженной культуре разработки ПО и больших объемах тиражируемых изделий. Си++ предоставляет хорошие возможности для адаптации языка к широкому спектру задач, так что создание паттернов и набора классов, ориентированных на приведенную выше специфику, вполне осуществимо. К сожалению, при этом следует смириться с тем, что обеспечить серьезный уровень контроля корректности программ и их семантическую целостность не удастся. Сложность подхода – высокие квалификационные требования к программистам, существенные затраты на обеспечение надежности и низкая сопровождаемость программ, трудности с вовлечением в процесс разработки конечного пользователя.

Анатолий Абрамович Шалыто – автор switch-технологии

Достаточно популярно в России обсуждение так называемой switch-технологии. В основе подхода лежит классическая реализация конечного автомата, в котором состояния автомата (некий набор функций) пронумерованы, а номер текущего состояния хранится в выделенной ячейке памяти. Текущая функция определяется через Си-конструкцию табличного выбора switch (этот факт и был использован при выборе названия). Собственно новизна предлагаемого подхода в наборе приемов по разработке алгоритма (через жестко прописанную процедуру документирования), его отладке (через встраиваемые в код функции записи текущего состояния в лог-файл) и идентификационная система для переменных. Подход может обеспечить цикличность, логический параллелизм, достаточную свободу в организации вычислений и, несомненно, имеет право на рассмотрение как вариант «пишу на Си». Switch-подход успешно используется в учебном процессе в Санкт-Петербургском государственном институте точной механики и оптики. К сожалению, подход нуждается в проработке методов синхронизации, структуризации и абстрагирования. Очевидно, что не про-

писаны процедуры связи с УСО, отсутствует четкий формат кодирования. Вызывает вопросы предлагаемая система для составления идентификаторов, которая заключается в конструировании идентификаторов на основе натуральных чисел.

Как дополнительные варианты Си-подхода, можно рассматривать работы, направленные на адаптацию алгоритмических языков Си, Си++, Java к задачам автоматизации через создание специализированных библиотек, классов и паттернов. Основной недостаток перечисленных подходов заключается в невозможности достигнуть необходимую степень комфортности программирования и безопасности получаемых программ. Остается очень большое число рутинных операций, выполняемых вручную. Контроль специфической семантики программ, появляющейся при расширении языков общего назначения, не автоматизирован. Отсутствует унификация привязки к УСО. Концептуально большинство таких подходов базируется на теории конечных автоматов, разрабатываемой, как уже отмечалось, для синтеза управляющих контроллеров из электронных компонентов. Но такой подход может быть использован в качестве варианта в единичном проекте, привязанном к группе разработчиков.

Особый интерес вызывает работа над стандартом МЭК 61499, в котором разработчики предприняли попытку преодолеть ограничения языков МЭК 61131-3 и скомбинировать в одном языковом средстве и поддержку логического параллелизма, и поддержку событийности. Цель стандарта – предоставить методологию разработки сложных алгоритмов. Программные компоненты представлены функциональными блоками специального вида: кроме обычных для языка FBD входных и выходных данных, интерфейс функционального блока стандарта МЭК 61499 предполагает событийные входы / выходы. Это нововведение частично решает проблему событийности для «классических» функциональных блоков. К сожалению, этот, несомненно, прогрессивный стандарт не поддержан ведущими производителями ПЛК, и известные в настоящий момент реализации стандарта носят скорее исследовательский характер.

Таким образом, использование одного из языков стандарта МЭК 61131-3 в реальных проектах может быть вполне успешным, но при условии тщательной проработки вопроса соответствия выбранного языка и требований задачи. Существенное преимущество МЭК 61131-3-подхода – наличие на рынке развитых сред разработки. При определенных обстоятельствах вполне допустимо использовать альтернативные средства программирования ПЛК либо на основе FSM, либо, при известной осторожности и наличии высококвалифицированных кадров, – на основе чистого Си / Си++ с набором библиотечных функций, созданным под решаемую задачу.

5. Язык Рефлекс

Установка по выращиванию монокристаллического кремния и система управления

Язык Рефлекс, разработанный в Институте автоматики и электрометрии СО РАН, базируется на модели гиперавтомата. Язык доказал свою эффективность в проектах по автоматизации сложных промышленных объек-тов, таких как станки с ЧПУ и системы управления выращивания монокристаллического кремния.

Модель гиперавтомата предполагает существование внешней среды, активное взаимодействие с ней, обеспечивает привязку алгоритма к физическому времени и предоставляет простые механизмы структурной организации параллельно исполняемых процессов. Это позволяет использовать модель гиперавтомата при создании управляющих алгоритмов широкого диапазона: для так называемых систем реального времени, реагирующих (reactive) и гомеостатических систем.

Модель программы определяет базовые приемы и сущности, которые используются при проектировании концептуальной структуры алгоритма управления. Собственно же описание программы в виде, пригодном для дальнейшего преобразования в машинные коды, производится на формальном языке программирования, который задается его синтаксисом (правилами построения допустимых языковых конструкций) и семантикой (смысловой корректностью).

При создании языка Рефлекс было принято важное решение выполнить его в качестве диалекта языка Си. Это позволяет коренным образом снизить сложность изучения языка и свести его освоение к изучению лишь действительно уникальных, концептуально новых языковых конструкций.

5.1. Концептуальная основа языка Рефлекс

Программа на языке Рефлекс описывается в текстовом виде. Как и в языке Си, символы пробела, табуляции, перевода строки и комментарии – это разделительные символы, которые используются для формирования удобочитаемого листинга программы. Несколько имен, чисел и (или) резервированных слов, расположенных последовательно, должны быть отделены по крайней мере одним разделительным символом, чтобы транслятор мог распознавать их как различные объекты. Комментарий в языке Реф-

лекс начинается последовательностью из символов '/' и '*' и заканчивается символами '*' и '/'.

В любом месте, где разрешено использование разделительного символа, допускается включение строк на языке Си. Каждая такая строка помечается последовательностью из трех символов «#СИ».

При написании программ допускается или англоязычный, или русскоязычный синтаксис. Выбор языка производится по написанию резервированного слова начала программы. Слово **PROGR** приводит к переключению на англоязычный синтаксис, а слово **Прогр** – на русскоязычный вариант синтаксиса. В статье синтаксис дается в русскоязычном написании. Таблица соответствия русскоязычного и англоязычного варианта приведена в прил. 1. За резервированным словом начала программы, специфицирующим синтаксис, указывается имя программы, далее следует открывающая фигурная скобка начала тела программы. Тело программы заканчивается закрывающей фигурной скобкой.

5.2. Тело программы. Константы. Функции. Привязка к интерфейсной аппаратуре

Тело программы состоит из двух частей: спецификации общих атрибутов гипераавтомата (периода цикла, констант, разрешенных для вызова Си-функций, портов ввода/вывода) и собственно описания процессов.

В первой части сначала указывается *период цикла активизации* процессов (в миллисекундах).

Затем с помощью резервированного слова **КОНСТ** определяются константы и их значения. Отличие определения констант от прагмы **define** в Си – закрывающая точка с запятой.

Константы допускается определять посредством ключевого слова **ПЕРЕЧИСЛЕНИЕ** – аналога **enum** в Си. В отличие от Си, имя перечисления опускается.

После описания констант в тексте программы дается список Си-функций, допустимых для вызова. Механизм формирования списка аналогичен декларации функций в Си. Начало описания функции помечается резервированным словом **ФУНКЦИЯ**.

После этого производится спецификация портов ввода / вывода. Порты ввода / вывода рассматриваются как отдельные компоненты языка, которые служат для задания привязки переменных к интерфейсной аппаратуре – устройствам связи с объектом (УСО).

Порты различаются по типу (входной или выходной), по размеру и адресу. Указывается тип порта (**ВХОД** или **ВЫХОД**), идентификатор порта, два числовых значения, определяющие его системный адрес, и разрядность (**8** или **16**). Семантика системного адреса определяется целевой вы-

числительной платформой. Формат записи чисел соответствует формату, принятому в Си.

Пример начальной части программы:

```
Прогр ПримерПрограммы {
        ТАКТ 10; /*задание периода активизации, в миллисекундах */
/*=======================================*/
/*= КОНСТАНТЫ ============================*/
/*=======================================*/
КОНСТ ВКЛ       1;
КОНСТ ВЫКЛ      0;
КОНСТ ОТКР      1;
КОНСТ ЗАКР      0;
КОНСТ СЕКУНДА           100; /*соответствует ста тактам (по 10 мс) */
КОНСТ ТРИ_СЕКУНДЫ 3*СЕКУНДА;
КОНСТ МИНУТА            60*СЕКУНДА;
КОНСТ ТРИ_МИНУТЫ        3*МИНУТА;
/*=======================================*/
/*= КОНСТАНТЫ - БАЗОВЫЕ АДРЕСА ПОРТОВ I/O ==*/
/*=======================================*/
КОНСТ ВА_FPGA1_U1       0XA110; /* входного порта */
КОНСТ ВА_FPGA3_U1       0XA910; /* выходного порта */

/*=======================================*/
/*====== ОПИСАНИЕ ФУНКЦИЙ ================*/
/*=======================================*/
ФУНКЦИЯ ЦЕЛ  GetMsg    (ЦЕЛ);
ФУНКЦИЯ ЦЕЛ  SendMsg   (ЦЕЛ);

/*=====================================*/
/*====== ОПИСАНИЕ ПОРТОВ I/O  =========*/
/*=====================================*/
ВХОД            ВХ_ПОРТ         ВА_FPGA1_U1     0       8;
ВЫХОД           ВЫХ_ПОРТ        ВА_FPGA3_U1     1       8;

/*=====================================*/
/*====== ОПИСАНИЕ ПРОЦЕССОВ  =======*/
/*=====================================*/
<описание процессов>
}
```

5.3. Процессы. Описание переменных

Описание процессов – вторая и основная часть программы, определяющая структуру и поведение управляющего алгоритма, представленного в виде гиперавтомата. Процессы описываются последовательно. Сам факт описания процесса в тексте программы автоматически означает его периодическую активизацию во время исполнения.

Процесс, описанный первым, является *начальным выделенным процессом*, т. е. по запуску программы это единственный процесс, находящийся в

активном состоянии (в *начальном состоянии*). Все остальные процессы находятся в *пассивном состоянии* «нормальный останов».

Описание отдельного процесса начинается резервированным словом **ПРОЦ**, за которым следует имя процесса и открывающая описание тела процесса фигурная скобка. Описание тела процесса состоит из описания переменных, доступных для использования внутри текущего процесса, и описания его поведения.

В языке Рефлекс переменные различаются по следующим признакам: тип, наличие отображения на модули УСО, степень доступа.

Каждая используемая в процессе переменная должна быть специфицирована либо непосредственно, либо через ссылку на ее описание в другом процессе.

В дополнение к типам переменных, используемым в Си (целые, короткие целые, плавающие, знаковые, беззнаковые), введены переменные логического типа (резервированное слово **ЛОГ**).

Описание отображаемой на модули УСО (*входной* или *выходной*) переменной устанавливает отношение эквивалентности между идентификатором этой переменной и битом (последовательностью битов) конкретного (входного или выходного) физического порта. При описании привязки к порту указываются имя порта и число битов. Если описание отображения на модули УСО отсутствует, то переменная является *внутренней*.

Степень доступа к переменной может быть трех типов: локальная (видимая только из процесса, в котором она описана), полностью открытая (видимая из любого процесса), частично открытая (видимая из нескольких, но не всех процессов). В случае "полностью" или "частично открытой" переменной достаточно, единожды описав ее в некотором процессе, в дальнейшем использовать ссылку. Степени доступа декларируются следующими резервированными словами – **ЛОКАЛ**, **ДЛЯ ВСЕХ**, **ДЛЯ ПРОЦ** <+ список процессов>. Ссылка на уже описанную переменную начинается резервированными словами **ИЗ ПРОЦ** <+ имя процесса>.

Расширенная по сравнению с Си типизация переменных языка Рефлекс введена, во-первых, для адекватного отражения специфики задач управления (связь с модулями УСО), во-вторых, – из соображений надежности, для обеспечения дополнительного контроля ошибок, в-третьих, из соображений удобства – желания использовать переменные с разными областями видимости. И действительно, переменные, помеченные как **LOCAL** – это не что иное, как глобальные (с точки зрения Си) переменные, но имеющие ограниченную видимость. Цикличность вызова функций-состояний гипераавтомата не позволяет использовать для этих целей динамические переменные, создаваемые в момент вызова функции в стеке. При этом использование одинаковых имен для статических переменных крайне удобно.

Примеры описания переменных:

```
  /*описание входной однобитовой переменной. Изменение состояния 1-го
бита выходного порта ВХ_ПОРТ автоматически приведет к изменению
переменной. Переменная доступна из любого процесса, в котором будет
присутствовать ссылка на нее: */
  ЛОГ СостояниеКлапанаVE1 = {ВХ_ПОРТ[1]} ДЛЯ ВСЕХ;

/*описание выходной однобитовой переменной. Изменение переменной
автоматически приведет к изменению состояния бита выходного порта
ВЫХ_ПОРТ. Переменная может быть доступна из указанных процессов в
случае, если в них будет присутствовать ссылка на нее: */
  ЛОГ УправлениеОткрытиемКлапанаVE1 = {ВЫХ_ПОРТ[1]} ДЛЯ ПРОЦ
                                          ОткрытиеКлапанаVE1,
                                          Закрытие КлапанаVE1;

/*описание внутренней переменной типа целое. Переменная доступна
только в процессе, в котором она описана. Ссылки на нее из других
процессов будут трактоваться транслятором как ошибка в программе: */
  ЦЕЛ Счетчик ЛОКАЛ;

/*ссылка на переменную, описанную в процессе Инициализация. При
описании в процессе Инициализация переменная должна иметь степень
доступа ДЛЯ ВСЕХ или ДЛЯ ПРОЦ. В последнем случае в списке
процессов должно присутствовать имя процесса, в котором дается
ссылка: */
  ИЗ ПРОЦ Инициализация УправлениеОткрытиемКлапанаVE1;
```

Язык Рефлекс, вообще говоря, как и Си, не имеет строгой типизации переменных. В начальных версиях языка использовалась строгая типизация переменных, но практическая отработка не показала явных преимуществ строгой типизации при появлении очевидных алгоритмических ограничений и усложнении синтаксиса. Единственным ограничением является запрет на запись во входные переменные.

5.4. Описание функций-состояний процесса

Событийный полиморфизм процесса задается путем последовательного описания его *функций-состояний* (или далее, для краткости, просто *состояний*). Описание состояния начинается с резервированного слова **СОСТ**, за которым следуют имя состояния и фигурная скобка, открывающая описание тела состояния, состоящей из *событий* и *реакций* на события. События и реакции формируются привычным для программиста способом с помощью условных и безусловных операторов. Описание состояния завершается закрывающей фигурной скобкой.

Стандартные Си-операторы расширены операторами, позволяющими организовать совместное функционирование процессов во времени:

- операторы проверки состояния стороннего процесса – конструкция-предикат **ПРОЦ В СОСТ** <имя состояния>;

- операторы запуска/останова процесса – конструкции **СТАРТ ПРОЦ** <имя процесса>, **СТОП ПРОЦ** <имя процесса>, **ОШИБКА ПРОЦ** <имя процесса> (работа с выделенными пассивными состояниями);
- операторы управления состоянием текущего процесса – конструкции **В СОСТ** <имя состояния>, **В СЛЕДУЮЩЕЕ**;
- оператор контроля времени **ТАЙМАУТ** <спецификатор времени/количество тактов>.

Если оператору проверки состояния в качестве имени состояния указать резервированное слово **АКТИВНОЕ**, то будет проверяться факт нахождения в любом активном состоянии. Резервированное слово **ПАССИВНОЕ** означает проверку факта нахождения в любом пассивном состоянии. Операторы останова процесса (**СТОП**, **ОШИБКА**) имеют редуцированную форму – могут использоваться без операнда (ссылки на процесс). Редуцированная форма означает, что действие оператора направлено на текущий процесс.

Пример, демонстрирующий использование типовых операторов управления текущим состоянием процесса:

```
ПРОЦ ОткрытиеКлапанаVE1{
 ИЗ ПРОЦ Инициализация
                Управление_VE1,
                Состояние_VE1; /*ссылка на переменные управления
                               * клапаном и контроля состоянием
                               * клапана, которые описаны в
                               * процессе Инициализация */
    СОСТ Начало {  /* первое описанное состояние – это функция,
                               * которую  процесс выполняет по
                               * запуску на исполнение */
        Управление_VE1 = ВКЛ;   /* подача управляющего сигнала
                               * на открытие */
        В СЛЕДУЮЩЕЕ;       /* безусловное изменение состояния
                               * процесса */
    }
    СОСТ ПроверкаОткрытия {
        ЕСЛИ (СостояниеКлапанаVE1 == ОТКР) СТОП;  /* если
                               * фиксируется
                               * открытие клапана, то процесс
                               * переходит в состояние «нормальный
                               * останов» */
        ТАЙМАУТ ТРИ_СЕКУНДЫ ОШИБКА;   /* в противном
                               * случае по
                               * истечении трех секунд процесс
                               * переходит в состояние «останов по
                               * ошибке» */
    }
}
```

В примере использован процесс открытия клапана, приведенный в предыдущей главе. Процесс выполняет открытие клапана VE1: выдает сигнал на открытие клапана VE1, затем проверяет его состояние. Если клапан открывается в течение 500 мс, то процесс переходит в состояние “нормального останова”. Если за 500 мс клапан не открылся, то исполняется оператор **ТАЙМАУТ** и процесс переходит в состояние “останов по ошибке”. Идентификаторы ВКЛ, ОТКР, ТРИ_СЕКУНДЫ являются идентификаторами констант (см. фрагмент программы на с. 62).

5.5. Процессы. Свойства структурности и абстрактности

На основе процессов можно гибко структурировать алгоритм. Например, если алгоритм предусматривает выполнение идентичных действий в нескольких местах, то целесообразно выделить эти действия в отдельный процесс и запускать его по мере необходимости с помощью оператора «СТАРТ». Это обеспечивает более компактную запись алгоритма и облегчает его модификации.

Для процесса, приведенного на с. 64, запуск будет выглядеть так:

СТАРТ ПРОЦ ОткрытиеКлапанаVE1.

Поскольку процессы исполняются параллельно, то запуск процесса просто приводит к появлению дополнительного потока управления (происходит дивергенция потока управления). Появившийся параллельный поток управления будет выполнять действия, описанные в запущенном процессе. Вызывающий процесс продолжается обычным образом. В случае необходимости в вызывающем процессе можно проконтролировать окончание выполнения запущенного процесса (произвести конвергенцию управляющего потока):

ЕСЛИ (ПРОЦ ОткрытиеКлапанаVE1 **В СОСТ ПАССИВНОЕ)**.

Успешное окончание процесса контролируется проверкой перехода процесса в состояние нормального останова:

ЕСЛИ (ПРОЦ ОткрытиеКлапанаVE1 **В СОСТ СТОП)**, а ошибочное окончание – в состояние останова по ошибке:

ЕСЛИ (ПРОЦ ОткрытиеКлапанаVE1 **В СОСТ ОШИБКА)**.

Другой эффективный способ структуризации алгоритма – оформление в процесс группы операций, допускающей независимое (параллельное) исполнение. При необходимости выполнить параллельно несколько операций (а в управляющих алгоритмах такая необходимость возникает очень часто) на исполнение запускается сразу несколько процессов. Этот прием позволяет производить произвольную дивергенцию и последующую конвергенцию потока управления:

```
ПРОЦ ВключениеКлапановОткачки {
  СОСТ Начало {
    СТАРТ ПРОЦ ОткрытиеКлапанаVE1; /* запуск процессов на
                                    * параллельное исполнение
                                    * (дивергенция) */
    СТАРТ ПРОЦ ОткрытиеКлапанаVM14;
    В СЛЕДУЮЩЕЕ;
  }
  СОСТ ПроверкаВключения { /*конвергенция  */
    ЕСЛИ ((ПРОЦ ОткрытиеКлапанаVE1 В СОСТ ПАССИВНОЕ) &&
          (ПРОЦ ОткрытиеКлапанаVM14 В СОСТ ПАССИВНОЕ)) {
      /* проверка отказа, и ошибка в случае отказа: */
    ЕСЛИ ((ПРОЦ ОткрытиеКлапанаVE1 В СОСТ ОШИБКА) ||
          (ПРОЦ ОткрытиеКлапанаVM14 В СОСТ ОШИБКА))
          ОШИБКА;
    ИНАЧЕ В СЛЕДУЮЩЕЕ; /* а иначе – продолжаем нормальную
                        * работу */
  }
  }
  СОСТ ДругиеДействия {
    <…другие действия…>
    СТОП; /* нормальный останов */
  }
}
```

Примечательный факт, который обнаруживается в приведенном фрагменте, – отсутствие в рассмотренном процессе «ВключениеКлапановОткачки» спецификаций переменных. Это говорит о весьма интересной возможности, которая предоставляется в рамках рассматриваемого подхода: возможности абстрагироваться при описании алгоритма от исполнительных органов и оперировать в терминах технологических операций и целевых функций, т. е. понятийно совместить описание алгоритма с проблемной областью его использования.

Процессы могут быть организованы в иерархию произвольного порядка. Иерархия, создаваемая по приведенному шаблону, позволяет организовать дополнительные слои информационной изоляции внутри алгоритма. Родственные отношения типа «родительский процесс / дочерний процесс» между запускающим и запускаемым процессом отсутствуют, что позволяет создавать в том числе и замкнутые иерархические структуры.

Явно заложенные тактирование и цикличность исполнения снимают вопрос о синхронности процесса. По сравнению с классическими конечными автоматами устойчивость в рамках модели гипераватомата – основная проблема, которая решается на пятом этапе синтеза конечного автомата, – трактуется более широко как отсутствие возможности смены состояния при отсутствии событий. Поскольку любая реакция обязательно связана с некоторым событием, то устойчивость состояний (а, значит, и отсутствие гонок) обеспечивается конструктивно.

Конструктивная устойчивость состояний позволяет исключить из рассмотрения понятие «частичный автомат». Полнота задания конечного автомата в табличном и графическом виде связана в классическом случае с трудоемкой процедурой перебора всех букв входного алфавита (по сути, всех возможных значений входных переменных). В языке Рефлекс отсутствие событий (а именно это означает отсутствие рассмотрения какого-либо варианта) приводит к сохранению текущего состояния и (или) текущих значений выходных переменных, т. е. к отсутствию реакции.

Необходимо подчеркнуть следующие важные моменты семантики языка.

Во-первых, все переменные процессов, в том числе локальные, гарантированно сохраняют свои значения. Несмотря на то что процесс реализуется циклически вызываемой функцией и локальные переменные могут иметь одинаковые имена (в разных процессах), значения локальных переменных языка Рефлекс, в отличие от локальных переменных процедурных языков, сохраняются при переходах из одного состояния в другое.

Во-вторых, следует особо подчеркнуть, что операторы смены состояний **В СЛЕДУЮЩЕЕ**, **В СОСТ** <имя состояния>, **СТОП**, **ОШИБКА** не являются аналогами Си-оператора перехода по метке **goto**. Равно, оператор **СТАРТ** <имя процесса> не является аналогом вызова функции. Эти операторы воздействуют только на ячейку памяти состояния процесса. Таким образом, действие этих операторов проявится только в момент активизации процесса, при выборе альтернативной функции, соответствующей текущему значению ячейки памяти состояния процесса. Эта особенность приводит невозможности для процесса находиться в двух состояниях на одном цикле активизации и, следовательно, к автоматическому выполнению условия однозначности. В случае выполнения двух операторов перехода действительной силой будет обладать оператор перехода, который исполнился последним (был описан в теле состояния позже).

Итоговые замечания. Таким образом, язык Рефлекс, ориентированный на описание алгоритмов функционирования сложных автоматизированных или полностью автоматических систем, построен как диалект языка Си. Синтаксис языка имеет как англоязычный, так и русскоязычный вариант. Си-подобный синтаксис упрощает изучение языка большинством программистов, русскоязычность делает его особенно привлекательным для отечественных пользователей.

Синтаксис Си расширен концептом параллельно исполняемого процесса (полиморфной функции событийного типа) и состояния (альтернативной функции процесса). Синтаксис исключает описание рутинных операций по циклическому считыванию данных с модулей УСО и периодическую активизацию процессов. Контекстно определяются начальный процесс гипераавтомата и начальное состояние процесса.

Текстовая форма записи позволила снизить трудоемкость задания конечного автомата, присущую классическим табличным и графическим способам его описания. Конструктивно обеспечены полнота определения автомата и выполнение условия однозначности переходов.

В синтаксис языка введены специальные операторы, обеспечивающие простой механизм дивергенции и конвергенции потока управления на основе процессов, а также механизм временной синхронизации. Способ организации алгоритма на основе процессов обеспечивает структуризацию программы и абстрагирование от исполнительных органов системы управления, позволяя понятийно совместить описание алгоритма с предметной областью.

Синтаксис и семантика языка концептуально адекватны модели гиперавтомата, что избавляет пользователя от необходимости изучать внутреннюю организацию событийного полиморфизма.

Семантика языка гарантирует безопасную работу с локальными переменными. Событийность как неотъемлемое свойство процесса конструктивно обеспечивает отсутствие гонок и свободу при выборе стратегии управления.

6. Типовые задачи промышленной автоматизации

6.1. Логическое управление

Разберем простой пример использования языка Рефлекс – алгоритм управления автоматизированной системы загрузки гравия (рис. 6.1).

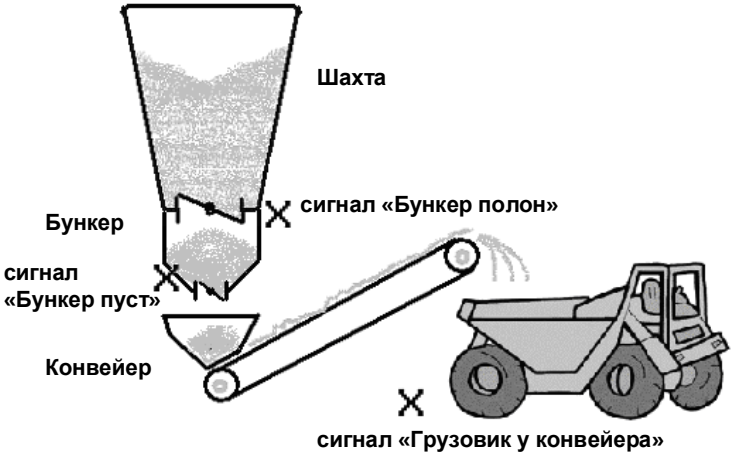


Рис. 6.1. Автоматическая загрузка гравия

Прогр АвтопогрузкаГравия

```
/* Демонстрационная программа управления эталоннным процессом
   Автопогрузки гравия в грузовики

Постановка задачи:

Автопогрузчик гравия состоит из шахты, в которую засыпан гравий.
На дне шахты расположена заслонка, через которую гравий засыпается в
нормировочный бункер. На дне нормировочного бункера также стоит
заслонка, через которую порция гравия поступает на конвейер.
Управление системой осуществляется тремя сигналами:
У_ОТКРЫТЬ_ШАХТУ, У_ОТКРЫТЬ_БУНКЕР, У_ВКЛ_КОНВЕЙЕР.
В системе имеется три датчика:
К_БУНКЕР_ПУСТ, К_БУНКЕР_ПОЛОН, К_ГРУЗОВИК_У_КОНВЕЙЕРА.
Алгоритм работы:
Заполнить бункер. Дождаться грузовика. Открыть бункер и
```

включить конвейер. Дождаться опустошения бункера.
Остановить конвейер. Дождаться, когда грузовик уедет.
Начать снова.

```
  ДАТА СОЗДАНИЯ:              19.09.2006 (8-05)
  ДАТЫ КОРРЕКЦИИ:             -
*/

{

 ТАКТ    10;   /* период активизации равен 100 миллисекундам */

/*#################################################################*/
/*#########  ОПИСАНИЕ КОНСТАНТ  ###############################*/
/*#################################################################*/

 КОНСТ ВКЛ            1;
 КОНСТ ВЫКЛ           0;
/*=================================================================*/
/*======= БАЗОВЫЕ АДРЕСА МОДУЛЕЙ УСО ======================*/
/*=================================================================*/

 КОНСТ BA_FPGA1_U1              0XA110; /* ВХОД */
 КОНСТ BA_FPGA3_U1              0XA910; /* ВЫХОД */

/*#################################################################*/
/*########  ОПИСАНИЕ РЕГИСТРОВ МОДУЛЕЙ УСО        #########*/
/*#################################################################*/

ВХОД  ЛОГ_ВХОДЫ  BA_FPGA1_U1 0 8; /* имя, базовый адрес, № Регистра, 8бит */
ВЫХОД ЛОГ_ВЫХОДЫ BA_FPGA3_U1 0 8;

/*
 * Процесс ЦиклЗагрузки.
 * Реализует весь алгоритм загрузки.
 */

ПРОЦ ЦиклЗагрузки{   /* это единственный процесс в в алгоритме */

/*=================================================================*/
/*========== ОПИСАНИЕ ПЕРЕМЕННЫХ ========================*/
/*=================================================================*/

/* ВХОДНЫЕ СИГНАЛЫ: */
 ЛОГ К_БУНКЕР_ПУСТ = {ЛОГ_ВХОДЫ[1]} ДЛЯ ВСЕХ;
 ЛОГ К_БУНКЕР_ПОЛОН = {ЛОГ_ВХОДЫ[1]} ДЛЯ ВСЕХ;
 ЛОГ К_ГРУЗОВИК_У_КОНВЕЙЕРА = {ЛОГ_ВХОДЫ[1]} ДЛЯ ВСЕХ;

/* ВЫХОДНЫЕ СИГНАЛЫ (т.к. привязаны к модулю выходов): */
 ЛОГ У_ОТКРЫТЬ_ШАХТУ = {ЛОГ_ВЫХОДЫ[1]} ДЛЯ ВСЕХ;
 ЛОГ У_ОТКРЫТЬ_БУНКЕР = {ЛОГ_ВЫХОДЫ[1]} ДЛЯ ВСЕХ;
 ЛОГ У_ВКЛ_КОНВЕЙЕР = {ЛОГ_ВЫХОДЫ[1]} ДЛЯ ВСЕХ;
```

```
СОСТ Начало{
 У_ОТКРЫТЬ_ШАХТУ = ВЫКЛ; /* приводим систему в исходное */
 У_ОТКРЫТЬ_БУНКЕР = ВЫКЛ;
 У_ВКЛ_КОНВЕЙЕР = ВЫКЛ;
 В СЛЕДУЮЩЕЕ;
}
СОСТ ЗаполнениеБункера{
 ЕСЛИ (!К_БУНКЕР_ПОЛОН) {
   У_ОТКРЫТЬ_ШАХТУ = ВКЛ;
 } ИНАЧЕ {
   У_ОТКРЫТЬ_ШАХТУ = ВЫКЛ;
   В СЛЕДУЮЩЕЕ;
 }
}
СОСТ ОжиданиеГрузовика{
 ЕСЛИ (К_ГРУЗОВИК_У_КОНВЕЙЕРА)
   В СЛЕДУЮЩЕЕ;
}
СОСТ ЗагрузкаПорцииГравия{
 ЕСЛИ (К_ГРУЗОВИК_У_КОНВЕЙЕРА && (!К_БУНКЕР_ПУСТ)) {
   У_ОТКРЫТЬ_БУНКЕР = ВКЛ;
   У_ВКЛ_КОНВЕЙЕР = ВКЛ;
 } ИНАЧЕ {
   В СЛЕДУЮЩЕЕ;
 }
}
СОСТ РазборПричиныОстанова{
 ЕСЛИ (!К_ГРУЗОВИК_У_КОНВЕЙЕРА) {
   У_ОТКРЫТЬ_БУНКЕР = ВЫКЛ;
   У_ВКЛ_КОНВЕЙЕР = ВЫКЛ;
   В СОСТ ЗаполнениеБункера;
 } ИНАЧЕ {  /* значит, бункер пуст */
   У_ОТКРЫТЬ_БУНКЕР = ВЫКЛ; /* закроем его */
   В СЛЕДУЮЩЕЕ;
 }
}
СОСТ ЖдемОтъездаГрузовика{
 ЕСЛИ (!К_ГРУЗОВИК_У_КОНВЕЙЕРА) {
   У_ВКЛ_КОНВЕЙЕР = ВЫКЛ;
   В СОСТ ЗаполнениеБункера;
 }
 }
}
} /* 8-20*/
```

Даже в таком простом алгоритме очень сложно обойтись без операций с временными интервалами. Обратите внимание, что если после опустошения бункера сразу остановить конвейер, то существенная часть гравия не попадет в кузов грузовика. Для решения этой проблемы в последнем состоянии ожидается отъезд грузовика.

6.2. Паузы, задержки, тайм-ауты

Программирование временных задержек с использованием оператора **ТАЙМАУТ** продемонстрируем на задаче по управлению дорожным движением на перекрестке (рис. 6.2).

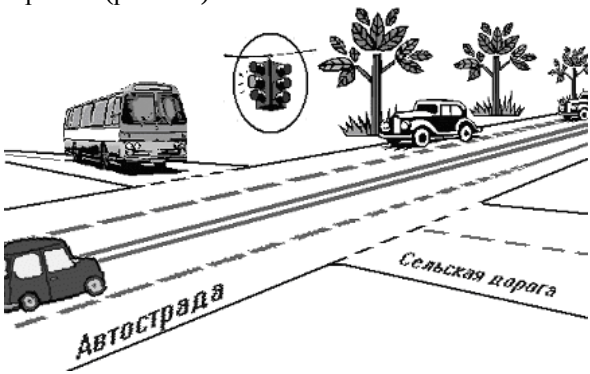


Рис. 6.2. Семафор на перекрестке автострады
и сельской дороги

Прогр СемафорНаПерекрестке

```
/* Демонстрационная программа управления
   сельским семафором

Постановка задачи:

Есть автострада и пересекающая ее сельская дорога.
На автостраде постоянное движение, на сельской дороге
машины появляются очень редко.
На перекрестке установлен «семафор», его штатное состояние –
«зеленый для автострады». Когда у переезда со стороны сельской дороги
появляется автомобиль, семафор начинает регулирование.
В «классической» формулировке управление светофором "может быть
осуществлено схемой управления, получающей сигналы от реле
времени и от параллельно соединенных датчиков давления,
вмонтированных в полотно" сельской дороги.
"Временной сигнал x1 отсутствует (x1=0) в течение 60 секунд
и появляется (x1=1) на 30 секунд. Красный сигнал светофора (z=1)
на автостраде может включаться лишь на интервале, на котором
x1=1. Включившись в начале этого интервала, он должен
оставаться включенным на протяжении всего интервала.
Срабатывание датчиков давления от автомобилей обозначается
в виде условия x2=1 (отсутствию автомобилей соответствует
условие x2=0; z становится равным единице на очередном интервале
включения x1)."
```

==========

Постановка не совсем корректная, т.к. светофор работает с тремя цветами (красный, желтый, зеленый). Но закроем на это глаза.

Итак. Имеется пересечение автострады с сельской дорогой...
один входной сигнал:
К_МАШИНЫ_У_ПЕРЕЕЗДА,
и сигнал управления:
У_ЗАПРЕТ_ДВИЖЕНИЯ_ПО_АВТОСТРАДЕ.

По условию адекватное изменение цвета для сельской дороги реализовано автоматически.

Структура решения задачи:
А. базовый процесс - ЦиклРаботыСветофора, который реализует цикл - 1 мин. пауза, 30 секунд - запрет движения, затем разрешение движения и останов.

Б. всем управляет процесс ОжиданиеМашин. Как появляются машины, он делает следующее - смотрит, если процесс ЦиклРаботыСветофора остановлен (в пассивном состоянии) - то запускает его...

```
  ДАТА СОЗДАНИЯ:          9.03.2006 (время решения задачи 15 минут)
  ДАТЫ КОРРЕКЦИИ:         15.09.2006 – коррекция условия задачи */

{

 ТАКТ    10;   /* период активизации равен 10 миллисекундам */

/*#################################################################*/
/*#########   ОПИСАНИЕ КОНСТАНТ  ##############################*/
/*#################################################################*/

 КОНСТ ВКЛ              1;
 КОНСТ ВЫКЛ             0;
 КОНСТ ОДНА_СЕКУНДА     100;
 КОНСТ ТРИДЦАТЬ_СЕКУНД  30*ОДНА_СЕКУНДА;
 КОНСТ ОДНА_МИНУТА      60*ОДНА_СЕКУНДА;
/*=============================================================*/
/*========= БАЗОВЫЕ АДРЕСА МОДУЛЕЙ УСО ====================*/
/*=============================================================*/

 КОНСТ ВА_FPGA1_U1              0XA110; /* ВХОД */
 КОНСТ ВА_FPGA3_U1              0XA910; /* ВЫХОД */

/*#################################################################*/
/*########  ОПИСАНИЕ РЕГИСТРОВ МОДУЛЕЙ УСО        #########*/
/*#################################################################*/

ВХОД  ЛОГ_ВХОДЫ  ВА_FPGA1_U1 0 8; /* имя, базовый адрес, № Регистра, 8бит */
ВЫХОД  ЛОГ_ВЫХОДЫ ВА_FPGA3_U1 0 8;
```

```
/*#################################################################*/
/*###############          ПРОЦЕССЫ              ###################*/
/*#################################################################*/

/* Процесс Инициализация. Служит для
 * развертывания программы. Этот процесс (описанный
 * первым) - единственно активный процесс по запуску.
 * Также процесс Инициализация содержит описание
 * переменных для ссылок из других
 * процессов. Это удобно: описания локализованы в одном
 * месте.
*/

ПРОЦ ОжиданиеМашин{   /* просто все инициализирует и запускает процесс */

/*=========================================================*/
/*========== ОПИСАНИЕ ПЕРЕМЕННЫХ ==========================*/
/*=========================================================*/

/* ВХОДНЫЕ СИГНАЛЫ: */
 ЛОГ К_МАШИНЫ_У_ПЕРЕЕЗДА     = {ЛОГ_ВХОДЫ[1]} ДЛЯ ВСЕХ;

/* ВЫХОДНЫЕ СИГНАЛЫ (т.к. привязаны к модулю выходов): */
 ЛОГ У_ЗАПРЕТ_ДВИЖЕНИЯ_ПО_АВТОСТРАДЕ = {ЛОГ_ВЫХОДЫ[1]} ДЛЯ ВСЕХ;

 СОСТ Начало{
  У_ЗАПРЕТ_ДВИЖЕНИЯ_ПО_АВТОСТРАДЕ  = ВЫКЛ;
  В СЛЕДУЮЩЕЕ;   /* смена текущей функции-состояния */
 }
 СОСТ КонтрольПоявленияМашин{
  ЕСЛИ (К_МАШИНЫ_У_ПЕРЕЕЗДА) {  /* есть машины ? */
     ЕСЛИ (ПРОЦ ЦиклРаботыСветофора В СОСТ ПАССИВНОЕ){ /* можно запускать?
*/
        СТАРТ ПРОЦ ЦиклРаботыСветофора;
     }
  }
  ЗАЦИКЛИТЬ; /* бесконечная работа в этом состоянии */
 }
}

ПРОЦ ЦиклРаботыСветофора { /* отработка одного цикла светофора:
                                                * 1 мин – зеленый для автострады,
                                                * 30 с - запрет движения по автостраде
                                                * стоп  */
 ИЗ ПРОЦ ОжиданиеМашин У_ЗАПРЕТ_ДВИЖЕНИЯ_ПО_АВТОСТРАДЕ;

 СОСТ Пауза_1мин{
   У_ЗАПРЕТ_ДВИЖЕНИЯ_ПО_АВТОСТРАДЕ = ВЫКЛ;
   ТАЙМАУТ ОДНА_МИНУТА В СЛЕДУЮЩЕЕ;
 }
 СОСТ ЗапретДвиженияНа30сек{
   У_ЗАПРЕТ_ДВИЖЕНИЯ_ПО_АВТОСТРАДЕ = ВКЛ;
   ТАЙМАУТ ТРИДЦАТЬ_СЕКУНД {
```

```
    У_ЗАПРЕТ_ДВИЖЕНИЯ_ПО_АВТОСТРАДЕ = ВЫКЛ; /* снимаем запрет автостраде */
    СТОП;
  }
 }
}
}
```

6.3. Параллелизм

Пример, включающий использование параллелизма и временные интервалы, – программа управления микроволновой печью.

```
Прогр Микроволновка

/* Демонстрационная программа управления микроволновой печью

Постановка задачи:

С точки зрения алгоритма управления автоматизируемая микроволновая печь
имеет два входных сигнала (К_КНОПКА_ПУСКА, К_ДВЕРЬ) и три выходных сигнала
управления и сигнализации (У_РАЗОГРЕВ, У_ЛАМПА, У_ЗВОНОК).

В исходном состоянии, когда дверь закрыта, лампочка выключена.
При открытии дверцы лампочка зажигается. После помещения продукта
в печь и закрытия дверцы лампочка гаснет. По нажатию кнопки
пуска включается силовой элемент (при закрытой двери, конечно),
включается лампочка, таймер устанавливает время приготовления,
равное 1 мин. Каждое следующее нажатие кнопки добавляет
к текущему времени приготовления 1 минуту. Открытие дверцы
во время работы печи прерывает процесс приготовления:
силовой элемент отключается, лампочка остается гореть, таймер
прекращает отсчет времени. Если время приготовления пищи
истекло, а процесс приготовления пищи не был прерван, то
лампочка гаснет и выдается звуковой сигнал.

  ДАТА СОЗДАНИЯ:          23.01.2006
  ДАТЫ КОРРЕКЦИИ:          -
*/
{
 ТАКТ    10;   /* период активизации равен 10 миллисекундам */

/*#############################################################*/
/*#########  ОПИСАНИЕ КОНСТАНТ  ###########################*/
/*#############################################################*/

 КОНСТ ВКЛ            1;
 КОНСТ ВЫКЛ           0;
 КОНСТ ОТКР           1;
 КОНСТ ЗАКР           0;
 КОНСТ ОДНА_СЕКУНДА   100;
```

```
 КОНСТ ОДНА_МИНУТА      60*ОДНА_СЕКУНДА;
/*=========================================================*/
/*========= БАЗОВЫЕ АДРЕСА МОДУЛЕЙ УСО ==================*/
/*=========================================================*/
 КОНСТ BA_FPGA1_U1              0XA110; /* ВХОД */
 КОНСТ BA_FPGA3_U1              0XA910; /* ВЫХОД */

/*#########################################################*/
/*########   ОПИСАНИЕ РЕГИСТРОВ МОДУЛЕЙ УСО        #########*/
/*#########################################################*/

ВХОД  ЛОГ_ВХОДЫ  BA_FPGA1_U1 0 8; /* имя, базовый адрес, № Регистра, 8бит */
ВЫХОД ЛОГ_ВЫХОДЫ BA_FPGA3_U1 0 8;

/*#########################################################*/
/*########          ПРОЦЕССЫ             ##################*/
/*#########################################################*/

/*
 * Процесс Инициализация. Служит для
 * развертывания программы. Этот процесс (описанный
 * первым) - единственно активный процесс по запуску.
 * Процесс Инициализация содержит описание
 * переменных для ссылок из других
 * процессов. Это удобно: описания локализованы в одном
 * месте.
*/

ПРОЦ Инициализация{   /* просто все инициализирует и запускает три процесса */

/*=========================================================*/
/*========== ОПИСАНИЕ ПЕРЕМЕННЫХ ==========================*/
/*=========================================================*/

/* ВХОДНЫЕ СИГНАЛЫ: */
 ЛОГ К_КНОПКА_ПУСКА         = {ЛОГ_ВХОДЫ[1]} ДЛЯ ВСЕХ;
 ЛОГ К_ДВЕРЦА               = {ЛОГ_ВХОДЫ[1]} ДЛЯ ВСЕХ;

/* ВЫХОДНЫЕ СИГНАЛЫ (т.к. привязаны к модулю выходов): */
 ЛОГ У_РАЗОГРЕВ = {ЛОГ_ВЫХОДЫ[1]} ДЛЯ ВСЕХ;
 ЛОГ У_ЛАМПА    = {ЛОГ_ВЫХОДЫ[1]} ДЛЯ ВСЕХ;
 ЛОГ У_ЗВОНОК   = {ЛОГ_ВЫХОДЫ[1]} ДЛЯ ВСЕХ;

 ДЦЕЛ ВремяГотовки ДЛЯ ВСЕХ;

 СОСТ Начало{
  У_РАЗОГРЕВ  = ВЫКЛ;
  У_ЛАМПА     = ВЫКЛ;
  У_ЗВОНОК    = ВЫКЛ;
  ВремяГотовки = 0;
  СТАРТ ПРОЦ КонтрольКнопкиПуска;
  СТАРТ ПРОЦ Разогрев;
  СТАРТ ПРОЦ СветоваяСигнализация;
```

```
  СТОП;
 }
}

ПРОЦ КонтрольКнопкиПуска { /* при нажатии кнопки пуска
                            просто увеличивает время готовки */
 ИЗ ПРОЦ Инициализация К_КНОПКА_ПУСКА,
                       ВремяГотовки;

 СОСТ КонтрольНажатия{
   ЕСЛИ (К_КНОПКА_ПУСКА == ВКЛ) {
      ВремяГотовки += ОДНА_МИНУТА;
      В СЛЕДУЮЩЕЕ;
   }
 }
 СОСТ ПаузаНаДребезгКнопки{
   ТАЙМАУТ ОДНА_СЕКУНДА В СОСТ КонтрольНажатия;
 }
}

ПРОЦ Разогрев{ /* готовит если дверца закрыта и время ненулевое,
                  если открывается дверца -
                  выключает нагреватель и сбрасывает время,
                  если приготовил  - звуковой
                  сигнал 1 сек  */
 ИЗ ПРОЦ Инициализация К_ДВЕРЦА,
                       У_ЗВОНОК,
                       У_РАЗОГРЕВ,
                       ВремяГотовки;
 СОСТ Начало{
   ЕСЛИ (К_ДВЕРЦА = ОТКР){
     ВремяГотовки = 0;
   } ИНАЧЕ {
     ЕСЛИ (ВремяГотовки != 0){
        У_РАЗОГРЕВ = ВКЛ;
        В СЛЕДУЮЩЕЕ;
     }
   }
 }
 СОСТ Разогрев{
   ЕСЛИ (К_ДВЕРЦА = ОТКР){
     У_РАЗОГРЕВ = ВЫКЛ;
     ВремяГотовки = 0;
     В СОСТ Начало;
   }
   ТАЙМАУТ ВремяГотовки {
     У_РАЗОГРЕВ = ВЫКЛ;
     В СЛЕДУЮЩЕЕ;
   }
 }
 СОСТ ЗвуковойСигнал {   /* можно выделить в процесс, а можно и так */
   У_ЗВОНОК = ВКЛ;
   ВремяГотовки = 0;
```

```
    ТАЙМАУТ ОДНА_СЕКУНДА {
      У_ЗВОНОК = ВЫКЛ;
      В СОСТ Начало;
    }
  }
}

ПРОЦ СветоваяСигнализация {  /* автономный процесс - просто
                              зажигает и гасит лампочку когда
                              это требуется */
 ИЗ ПРОЦ Инициализация К_ДВЕРЦА,
                       У_РАЗОГРЕВ,
                       У_ЛАМПА;

 СОСТ КонтрольУсловийВключения{
   ЕСЛИ ((К_ДВЕРЦА == ОТКР) || (У_РАЗОГРЕВ == ВКЛ))
      У_ЛАМПА = ВКЛ;
   ИНАЧЕ
      У_ЛАМПА = ВЫКЛ;
   ЗАЦИКЛИТЬ;
 }
}
}
```

6.4. Сложные алгоритмы

Если для простых алгоритмов управления на языке Рефлекс в общем-то достаточно одной постановки, то для сложных алгоритмов перед кодированием имеет смысл наметить общую структуру программы. Делается это в произвольной форме. Обычно используются либо логические блок-схемы, либо просто рисунки, либо одна из подходящих диаграмм UML. Основная цель этого этапа разработки – лучше усвоить задачу, прочувствовать ее и найти наиболее простое решение. Сделанные записи при необходимости, – скажем, по прошествию длительного времени, – могут быстро восстановить основной ход мыслей при решении задачи.

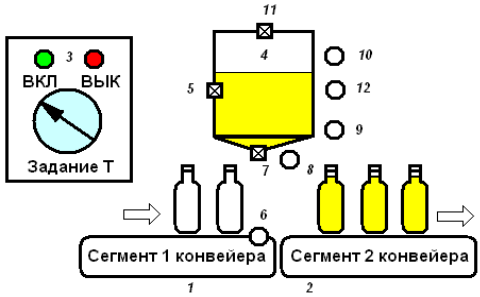


Рис. 6.3. Автоматическая линия розлива жидкости в бутылки

В качестве демонстрационной используем задачу автоматизации линии розлива жидкости в бутылки (рис. 6.3). Ниже приводится условие задачи.

Двухсегментный конвейер (*1*, *2*) используется для перемещения бутылок. Первый сегмент *1* включается и отключается в соответствии с алго-

ритмом, приведенным чуть позже. Второй сегмент *2* по включению системы постоянно находится в работе. Кнопка управления системой *3* включает и отключает систему. Резервуар *4* содержит разливаемую жидкость. По условию задачи разливаемая жидкость находится в нагретом состоянии, температура жидкости поддерживается на уровне 100 °С. Хотя температура может поддерживаться типовым ПИД-регулятором, по условию задачи температура поддерживается паровым клапаном *5*. При падении температуры жидкости ниже 100 °С этот клапан открывается и пускает в систему перегретый пар, который разогревает резервуар. Клапан закрывается по достижению температуры 110 °С. Датчик положения бутылки *6* служит для обнаружения бутылки в заданном положении и срабатывает, когда бутылка находится под соплом резервуара. Когда бутылка обнаружена, первый сегмент конвейера останавливается. Клапан розлива в бутылку *7*, расположенный на дне резервуара, открывается при совпадении следующих четырех условий: а) обнаружена бутылка; б) эта бутылка пуста; в) в резервуаре есть жидкость; г) температура жидкости больше или равна 100 °С. После его открытия жидкость через сопло заливается в бутылку. Количество жидкости в бутылке контролируется специальным фотодатчиком уровня жидкости *8* в бутылке, который срабатывает, когда бутылка заполнена. Уровень жидкости в резервуаре контролируется двумя датчиками: датчиком отсутствия жидкости *9* и датчиком полноты резервуара *10*. Клапан пополнения резервуара жидкостью *11* открывается по срабатыванию датчика отсутствия жидкости (и выключается при срабатывании датчика полноты резервуара). Операции розлива в бутылки (клапан розлива в бутылку) и разогрева (паровой клапан) запрещены во время пополнения резервуара жидкостью (клапан пополнения резервуара). Температура жидкости измеряется аналоговым датчиком *12*.

Анимированный gif-файл, демонстрирующий работу системы, можно посмотреть по адресу [http://reflex-language.narod.ru/bottle/ani_bottle.htm].

Вообще говоря, к постановке задачи имеются претензии. Например, непонятно, почему запрещен разогрев жидкости при пополнении резервуара или почему разогрев включается только при падении температуры ниже 100 °С, что приводит к прекращению розлива, и почему нельзя это делать с упреждением, и т. п. Чтобы избежать дискуссий, предлагается не обсуждать эту классическую постановку задачи.

Определение общей структуры алгоритма. Из условия задачи можно выделить пять более-менее независимых, и главное, естественных для семантики автоматизируемого объекта, процессов:

- процесс «Включение / выключение системы»,
- процесс «Поддержание уровня жидкости в резервуаре»,
- процесс «Поддержание температуры жидкости в баке»,
- процесс «Розлив жидкости в бутылку»,

• процесс «Подача пустой бутылки к соплу» и передача заполненной бутылки по конвейеру.

По условию задачи эти процессы связаны друг с другом следующим образом.

1. Если система не включена, то ни один из процессов не работает.

2. Если в резервуар наливается жидкость, то запрещена работа процессов «Поддержание температуры жидкости в баке» и «Розлив жидкости в бутылку».

3. Если резервуар разогревается, то запрещена работа процесса «Розлив жидкости в бутылку».

Отразим взаимовлияние процессов друг на друга (рис. 6.4).

Эта иерархия процессов и будет соответствовать структуре программы на языке Рефлекс, приведенной ниже.

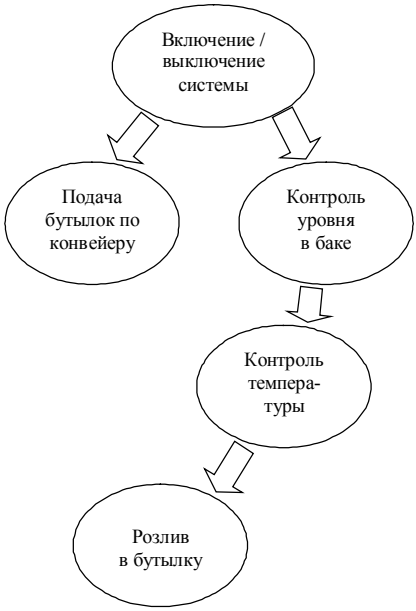


Рис. 6.4. Автоматическая линия розлива жидкости в бутылки. Взаимодействие процессов

Прогр BottleFillingSystem

```
/* Демонстрационная программа управления эталоннным процессом
   Автоматизированная линия розлива

   ДАТА СОЗДАНИЯ:          24.12.2005
   ДАТЫ КОРРЕКЦИИ:          -
*/

{

 ТАКТ    10;   /* период активизации равен 100 миллисекундам */

/*###############################################################*/
/*#########  ОПИСАНИЕ КОНСТАНТ  ##########################*/
/*###############################################################*/

 КОНСТ ВКЛ          1;
 КОНСТ ВЫКЛ          0;

/*=============================================================*/
/*======= БАЗОВЫЕ АДРЕСА МОДУЛЕЙ УСО ======================*/
/*=============================================================*/
```

```
 КОНСТ BA_FPGA1_U1              0XA110; /* ВХОД */
 КОНСТ BA_FPGA3_U1              0XA910; /* ВЫХОД */
 КОНСТ BA_FPGA1_U2              0XA120; /* АНАЛОГОВЫЙ ВХОД 8бит*/

/*#################################################################*/
/*########  ОПИСАНИЕ РЕГИСТРОВ МОДУЛЕЙ УСО          #########*/
/*#################################################################*/

ВХОД  ЛОГ_ВХОДЫ  BA_FPGA1_U1 0 8; /* имя, базовый адрес, № Регистра, 8бит */
ВЫХОД ЛОГ_ВЫХОДЫ BA_FPGA3_U1 0 8;
ВХОД  ВХОД_АЦП          BA_FPGA1_U2 0 8;

/*#################################################################*/
/*########  ПРОЦЕССЫ     ###################################*/
/*#################################################################*/

/*
 * Процесс Инициализация. Служит для
 * развертывания программы. Этот процесс (описанный
 * первым) - единственно активный процесс по запуску.
 * Кроме этого, процесс Инициализация содержит описание
 * переменных для ссылок из других
 * процессов. Это удобно: описания локализованы в одном
 * месте.
*/

ПРОЦ Инициализация{

/*=============================================================*/
/*=========== ОПИСАНИЕ ПЕРЕМЕННЫХ ============================*/
/*=============================================================*/

/* ВХОДНЫЕ СИГНАЛЫ:
тип, имя,               модуль УСО, 1 бит, "видимость" из др. процессов */
 ЛОГ К_КНОПКА_ПУСКА_СИСТЕМЫ  = {ЛОГ_ВХОДЫ[1]} ДЛЯ ВСЕХ;
 ЛОГ К_БУТЫЛКА_ПОД_СОПЛОМ    = {ЛОГ_ВХОДЫ[1]} ДЛЯ ВСЕХ;
 ЛОГ К_БУТЫЛКА_НАПОЛНЕНА     = {ЛОГ_ВХОДЫ[1]} ДЛЯ ВСЕХ;
 ЛОГ К_БАК_ПУСТ              = {ЛОГ_ВХОДЫ[1]} ДЛЯ ВСЕХ;
 ЛОГ К_БАК_ПОЛОН             = {ЛОГ_ВХОДЫ[1]} ДЛЯ ВСЕХ;

/* ВЫХОДНЫЕ СИГНАЛЫ (т.к. привязаны к модулю выходов): */
 ЛОГ У_НАПОЛНЕНИЕ_БАКА    = {ЛОГ_ВЫХОДЫ[1]} ДЛЯ ВСЕХ;
 ЛОГ У_РАЗОГРЕВ_БАКА      = {ЛОГ_ВЫХОДЫ[1]} ДЛЯ ВСЕХ;
 ЛОГ У_ЗАПОЛНЕНИЕ_БУТЫЛКИ = {ЛОГ_ВЫХОДЫ[1]} ДЛЯ ВСЕХ;
 ЛОГ У_ВКЛ_КОНВЕЙЕРА_1    = {ЛОГ_ВЫХОДЫ[1]} ДЛЯ ВСЕХ;
 ЛОГ У_ВКЛ_КОНВЕЙЕРА_2    = {ЛОГ_ВЫХОДЫ[1]} ДЛЯ ВСЕХ;

/* СИГНАЛ ОТ АЦП:
 короткое целое, фактический параметр - температура, 8 битов,
 свободный доступ из любого процесса */
 ЦЕЛ ФП_ТЕМПЕРАТУРА   = {ВХОД_АЦП[8]}  ДЛЯ ВСЕХ;
```

```
/*########  СОСТОЯНИЯ ПРОЦЕССА  #########*/
 СОСТ Начало{  /* именно отсюда все и начинается
                          * после включения питания */
 У_НАПОЛНЕНИЕ_БАКА    = ВЫКЛ; /* выключаем все */
 У_РАЗОГРЕВ_БАКА      = ВЫКЛ;
 У_ЗАПОЛНЕНИЕ_БУТЫЛКИ = ВЫКЛ;
 У_ВКЛ_КОНВЕЙЕРА_1    = ВЫКЛ;
 У_ВКЛ_КОНВЕЙЕРА_2    = ВЫКЛ;
 В СЛЕДУЮЩЕЕ;              /* переходим в следующее состояние */
}
 СОСТ ОжиданиеНажатияКнопкиСтарт{
 ЕСЛИ   (К_КНОПКА_ПУСКА_СИСТЕМЫ   ==   ВКЛ)   {   /*   каждые   100   мс
                                                          * проверяем кнопку */
   У_ВКЛ_КОНВЕЙЕРА_2 = ВКЛ;     /* если нажата - включаем конвейер 2 */
   СТАРТ ПРОЦ НаполнениеБака;
   СТАРТ ПРОЦ ПодачаБутылок;
   В СЛЕДУЮЩЕЕ;                  /* и уходим в следующее */
 }                          /* а иначе - ничего не делаем */
}
 СОСТ ОжиданиеОстанова{
 ЕСЛИ (К_КНОПКА_ПУСКА_СИСТЕМЫ == ВЫКЛ) { /* каждые 100 мс
                                                          * проверяем кнопку */
   СТОП ПРОЦ НаполнениеБака;
   СТОП ПРОЦ ПодогревБака;
   СТОП ПРОЦ РозливБутылок;
   СТОП ПРОЦ ПодачаБутылок;
   В СОСТ Начало;                /* по выключению и уходим в */
 }                          /* Начало, там все выключится */
}
}

ПРОЦ НаполнениеБака{
ИЗ ПРОЦ Инициализация К_БАК_ПУСТ,  /* описывать уже не надо, */
                К_БАК_ПОЛОН, /* просто ссылка         */
                У_НАПОЛНЕНИЕ_БАКА,
                У_РАЗОГРЕВ_БАКА,
                У_ЗАПОЛНЕНИЕ_БУТЫЛКИ;
 СОСТ Начало{
  ЕСЛИ (К_БАК_ПУСТ) В СЛЕДУЮЩЕЕ;
  ИНАЧЕ {
    СТАРТ ПРОЦ ПодогревБака; /*если есть жидкость, то можно нагревать*/
    В СЛЕДУЮЩЕЕ;
  }
 }
 СОСТ КонтрольУровня_в_Баке{
   ЕСЛИ (К_БАК_ПУСТ) {  /* если бак опустел, то */
     СТОП ПРОЦ ПодогревБака;
     СТОП ПРОЦ РозливБутылок;
     У_РАЗОГРЕВ_БАКА  = ВЫКЛ;     /* разогревать нельзя */
     У_ЗАПОЛНЕНИЕ_БУТЫЛКИ = ВЫКЛ; /* в бутылки лить тоже нельзя */
     У_НАПОЛНЕНИЕ_БАКА = ВКЛ;     /* можно только наполнять бак */
     В СЛЕДУЮЩЕЕ;
   }
```

```
 }
 СОСТ КонтрольНаполненияБака{
   ЕСЛИ (К_БАК_ПОЛОН) {  /* если бак наполнился до отказа, то */
     СТАРТ ПРОЦ ПодогревБака;   /* можно разогревать */
     У_НАПОЛНЕНИЕ_БАКА = ВЫКЛ;  /* а наполнять уже не надо */
     В СОСТ КонтрольУровня_в_Баке;
   }
 }
}

ПРОЦ ПодогревБака{
ИЗ ПРОЦ Инициализация ФП_ТЕМПЕРАТУРА,
                У_РАЗОГРЕВ_БАКА,
                У_ЗАПОЛНЕНИЕ_БУТЫЛКИ;

 СОСТ Начало{
  ЕСЛИ (ФП_ТЕМПЕРАТУРА < 100) В СЛЕДУЮЩЕЕ;
  ИНАЧЕ {
    СТАРТ ПРОЦ РозливБутылок; /* если Т >= 100 – можно разливать */
    В СЛЕДУЮЩЕЕ;
  }
 }
 СОСТ КонтрольОстыванияБака{
   ЕСЛИ (ФП_ТЕМПЕРАТУРА < 100) {  /* если бак остыл, то */
     СТОП ПРОЦ РозливБутылок;
     У_ЗАПОЛНЕНИЕ_БУТЫЛКИ = ВЫКЛ; /* в бутылки лить нельзя */
     У_РАЗОГРЕВ_БАКА  = ВКЛ;      /* а надо греть бак */
     В СЛЕДУЮЩЕЕ;
   }
 }
 СОСТ КонтрольНагреваБака{
   ЕСЛИ (ФП_ТЕМПЕРАТУРА > 110) { /* если бак нагрелся, то */
     СТАРТ ПРОЦ РозливБутылок;  /* можно разливать */
     У_РАЗОГРЕВ_БАКА = ВЫКЛ;  /* а греть уже не надо */
     В СОСТ КонтрольОстыванияБака;
   }
 }
}

ПРОЦ РозливБутылок{
ИЗ ПРОЦ Инициализация К_БУТЫЛКА_ПОД_СОПЛОМ,
                К_БУТЫЛКА_НАПОЛНЕНА,
                У_ЗАПОЛНЕНИЕ_БУТЫЛКИ;

 СОСТ Начало{
  ЕСЛИ (К_БУТЫЛКА_ПОД_СОПЛОМ && !К_БУТЫЛКА_НАПОЛНЕНА)
     У_ЗАПОЛНЕНИЕ_БУТЫЛКИ = ВКЛ;
  ИНАЧЕ
     У_ЗАПОЛНЕНИЕ_БУТЫЛКИ = ВЫКЛ;
  ЗАЦИКЛИТЬ; /* снимем контроль ошибки по отсутствию перехода из */
 }
}
```

```
ПРОЦ ПодачаБутылок{
ИЗ ПРОЦ Инициализация К_БУТЫЛКА_ПОД_СОПЛОМ,
                      К_БУТЫЛКА_НАПОЛНЕНА,
                      У_ВКЛ_КОНВЕЙЕРА_1;

 СОСТ Начало{
  ЕСЛИ (К_БУТЫЛКА_НАПОЛНЕНА || !К_БУТЫЛКА_ПОД_СОПЛОМ)
     У_ВКЛ_КОНВЕЙЕРА_1 = ВКЛ;
  ИНАЧЕ
     У_ВКЛ_КОНВЕЙЕРА_1 = ВЫКЛ;
  ЗАЦИКЛИТЬ;
 }
}
}
```

6.5. Верификация и моделирование

Возрастающая сложность автоматизируемых технических систем обусловливает необходимость создания принципиально новых методик проектирования. Ключевое требование, которое предъявляется к таким методикам, – требование обеспечить надлежащее качество, надежность создаваемого программного обеспечения, что особенно актуально для класса задач автоматизации "критичных" объектов на предприятиях металлургической и химической промышленности, атомных электростанциях, предприятиях космической индустрии, а также класса задач автоматизации сложных научно-технических экспериментов, связанных с такими производствами. Для перечисленных случаев возникновение ошибки не только влечет неприемлемые материальные и финансовые потери, но зачастую имеет катастрофические последствия – вплоть до человеческих жертв.

Качество создаваемого ПО в первую очередь обеспечивается за счет среды разработки. На основании этого даже вводится понятие «надежность языкового средства», которое в большинстве случаев трактуется как мера степени обнаружения ошибок на этапе трансляции и исполнения программы. Основной упор при этом делается на формальные методы: строгую типизацию переменных, выявление синтаксических и семантических ошибок. Тем не менее формальный подход имеет вполне очевидные ограничения.

Для построения любой языковой системы имеется предел обнаруживаемых ею ошибок, и ошибки в логическом построении программы принципиально не поддаются автоматическому обнаружению. Причем чем более сложен создаваемый алгоритм, тем более вероятно возникновение таких ошибок. На помощь приходит прагматика и психология программирования: весьма успешно бороться с логическими ошибками помогает использование специализированных языков, наиболее адекватных задаче. Но и это не га-

рантирует 100-процентного успеха. Дальнейшая работа по выявлению возможных ошибок производится через усиленное тестирование созданной программы. Для этого в интегрированные среды разработки включаются дополнительные возможности по трассировке программ и их отладке, прогонке тестов. В особо ответственных проектах тестирование – это ключевая операция, вокруг которой выстраивается весь процесс разработки.

Но насколько универсальны такие средства отладки? Можно ли их использовать для отладки управляющих алгоритмов? Анализ показывает, что встроенные средства таких мощных сред, как .NET или Visual C++ ориентированы исключительно на отладку WIMP-систем (Windows-Icons-Menus-Pointer_Devices). Специфичен не только процесс создания управляющих алгоритмов, но и процесс их отладки, в силу того что для тестирования управляющего алгоритма необходима некоторая модель управляемого объекта.

Управляемый объект и его модель. На настоящий момент общепринятая практика отладки управляющего алгоритма при наличии денежных средств в бюджете проекта – это создание полномасштабных имитаторов управляемого объекта в «железе». Рядом с отлаживаемой системой управления ставится дополнительный контроллер с набором модулей УСО, по сигналам совпадающих с входами / выходами управляемого объекта. Специально созданный алгоритм, исполняемый на контроллере, реализует физические процессы на управляемом объекте. Стоимость такого имитатора приблизительно равна стоимости системы управления.

Если же средства на отладку ограничены, то создаются ручные имитаторы, подключаемые лишь к входам системы управления. В этом случае поведение управляемого объекта задается положением ручных переключателей и переменных резисторов. В некоторых интегрированных средах МЭК 61131-3 имеется возможность программно отключить модули УСО и задавать на входе алгоритма тестовые значения сигналов или даже изменяющиеся во времени последовательности значений.

Рис. 6.5. Симметрия взаимодействия система управления / управляемый объект

Если внимательно проанализировать рис. 2.2, то становится очевидным,

что схема взаимодействия системы управления и управляемого объекта отражена на рисунке однобоко – исключительно со стороны системы управления. В то время как со стороны управляемого объекта ситуация абсолютно симметричная (рис. 6.5). С той разницей, что входные сигналы системы управления – это выходные сигналы управляемого объекта и наоборот, выходные сигналы системы – это входные сигналы управляемого объекта. События на управляемом объекте возникают под воздействием реакции, выдаваемой системой управления. Что, в общем-то, понятно и из общих соображений: управляемый объект – такой же автомат («машина», «устройство»), как и система управления. Таким образом, объект управления может быть промоделирован (заменен) гипераватоматом.

В результате замены управляющего алгоритма и объекта управления гипераватоматами мы получим систему из двух гипераватоматов, входы и выходы которых замкнуты друг на друга (рис. 6.6).

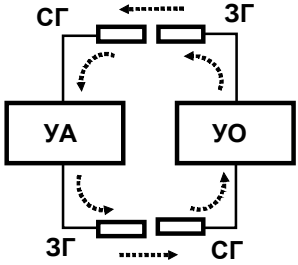


Рис. 6.6. Взаимодействие управляющего алгоритма (УА) и управляемого объекта (УО). Замкнутые друг на друга гипераватоматы

Вывод, который естественным образом вытекает из сказанного, прост – управляемый объект может моделироваться теми же языковыми (и понятийными) средствами. При таком подходе для тестирования алгоритма используется не дорогостоящая физическая модель объекта управления, а программная. Более того, нет необходимости даже в системе управления (программируемом контроллере и наборе модулей УСО): алгоритм может тестироваться на персональном компьютере.

Библиографический список

1. *Баранов С. И.* Синтез микропрограммных автоматов. Л.: Энергия. Ленингр. отд-ние, 1979. 232 с.

2. *Глушков В. М.* Синтез цифровых автоматов. М.: Физматгиз, 1963. 476 с.

3. *Зюбин В. Е.* Язык Рефлекс. Информационный портал // http://reflex-language.narod.ru/.

4. *Кругосвет*: Энциклопедия / Под ред. А. В. Добровольского // http://www.krugosvet.ru/. 2006.

5. *Петров И. В.* Стандартные языки и приемы прикладного программирования. М.: СОЛОН-Пресс, 2004.

6. *Crater K.* When Technology Standards Become Counterproductive. Control Technology Corporation, 1996. http://www.cic-control.com/customer/elearning/whpapers/counter.asp/.

7. *IEC* 61131-3. Programmable controllers. Part 3: Programming languages, 2nd Ed // International Electrotechnic Commission. 2003. http://www.iec.ch/.

Приложение 1

Таблица соответствия русскоязычного и англоязычного варианта синтаксиса языка Рефлекс

№ п\п	*Вариант написания*		*Семантика и использование*
	Русскоязычный	*Англоязычный*	
1	**Прогр**	**PROGR**	Начало программы, выбор варианта написания синтаксиса
2	**ТАКТ**	**TACT**	Задание периода активизации
3	**КОНСТ**	**CONST**	Начало описания константы, примерный аналог в Си - #define
4	**ВХОД**	**INPUT**	Начало описания входного порта
5	**ВЫХОД**	**OUTPUT**	Начало описание выходного порта
6	**ПРОЦ**	**PROC**	1. Начало описания процесса - ПРОЦ <имя процесса> { <тело процесса> } Или, в общем случае, указание на то, что далее следует имя процесса, например, СТОП ПРОЦ <имя процесса>; ДЛЯ ПРОЦ <список имен процессов>;
7	**ЛОКАЛ**	**LOCAL**	Задание степени доступа к переменной – локальная.
8	**ДЛЯ**	**FOR**	Задание степени доступа к переменной: используется в сочетаниях ДЛЯ ВСЕХ или ДЛЯ ПРОЦ <список имен процессов>
9	**ВСЕХ**	**ALL**	Используется при задании степени доступа переменной (см. ДЛЯ)
10	**ИЗ**	**FROM**	Используется для ссылки на описание переменной. ИЗ ПРОЦ <имя процесса> <список переменных>;
11	**ЛОГ**	**LOG**	Задание типа переменной (тип булевая)

Продолжение табл.

№ п\п	*Вариант написания*		*Семантика и использование*
	Русскоязычный	*Англоязычный*	
12	**ЦЕЛ**	**INT**	Задание типа переменной (тип целая), аналог типа int языка Си
13	**СОСТ**	**STATE**	1. Начало описания состояния - СОСТ <имя состояния> { <тело состояния> } 2. Или, в общем случае, указание на то, что далее следует спецификация имени состояния, например, В СОСТ <имя состояния>;
14	**СТОП**	**STOP**	1. Оператор перевода указанного процесса в выделенное состояние «нормальный останов» (нормальный останов процесса) Например, СТОП ПРОЦ <имя процесса>; 2. Или имя состояние нормального останова при операциях контроля состояния процесса ЕСЛИ (ПРОЦ <имя процесса> В СОСТ СТОП) {<действия>}
15	**СТАРТ**	**START**	Оператор перевода указанного процесса в выделенное состояние «начальное состояние» (запуск процесса) СТАРТ ПРОЦ <имя процесса>;
16	**ТАЙМАУТ**	**TIMEOUT**	Оператор контроля времени нахождения текущего процесса в текущем состоянии ТАЙМАУТ ТРИ_СЕКУНДЫ {<действия>}
17	**ЕСЛИ**	**IF**	Условный оператор, аналог оператора if языка Си
18	**ИНАЧЕ**	**ELSE**	Оператор альтернативы, аналог оператора else языка Си

Продолжение табл.

№ п\п	*Вариант написания*		*Семантика и использование*
	Русскоязычный	*Англоязычный*	
19	**В**	**IN**	Совместно с резервированным словом СОСТ в операторах перехода специфицирует состояние перехода, а в операциях контроля события текущее состояние процесса. Совместно с резервированным словом СЛЕДУЮЩЕЕ специфицирует переход в следующее по порядку описания состояние процесса. Например, В СОСТ <имя состояния>; ЕСЛИ (ПРОЦ <имя процесса> В СОСТ ОШИБКА) {<действия>} В СЛЕДУЮЩЕЕ;
20	**СЛЕДУЮЩЕЕ**	**NEXT**	См. резервированное слово В
21	**ОШИБКА**	**ERROR**	1. Оператор перевода указанного процесса в выделенное пассивное состояние «останов по ошибке» (ошибка процесса) Например, ОШИБКА ПРОЦ <имя процесса>; 2. Или спецификация состояния «останов по ошибке» при операциях контроля состояния процесса ЕСЛИ (ПРОЦ <имя процесса> В СОСТ ОШИБКА) {<действия>}
22	**КЦЕЛ**	**SHORT**	Задание типа переменной (тип короткая целая), аналог типа short языка Си
23	**ДЦЕЛ**	**LONG**	Задание типа переменной (тип длинная целая), аналог типа long языка Си
24	**ПЛАВ**	**FLOAT**	Задание типа переменной (тип с плавающей точкой), аналог типа float языка Си
25	**ДПЛАВ**	**DOUBLE**	Задание типа переменной (тип с двойной точностью), аналог типа double языка Си

Продолжение табл.

№ п\п	*Вариант написания*		*Семантика и использование*
	Русскоязычный	*Англоязычный*	
26	**РАЗБОР**	**SWITCH**	Задание табличного разбора, аналог резервированного слова switch языка Си. Например, РАЗБОР (<имя целочисленной переменной>) …
27	**СЛУЧАЙ**	**CASE**	Начало описания действий при табличном разборе, аналог резервированного слова case языка Си. Например, СЛУЧАЙ 5: <действия>
28	**КОНЕЦ**	**BREAK**	Окончание описания действий при табличном разборе, аналог резервированного слова break языка Си при использовании совместно со switch. Например, СЛУЧАЙ 5: <действия> КОНЕЦ;
29	**УМОЛЧАНИЕ**	**DEFAULT**	Действия по умолчанию при табличном разборе, аналог резервированного слова default языка Си.
30	**АКТИВНОЕ**	**ACTIVE**	При операциях контроля состояния процесса замещает любое активное состояние процесса. Служит для сокращения рутинных описаний. Например ЕСЛИ (ПРОЦ <имя процесса> В СОСТ АКТИВНОЕ) {<действия>} – действия выполняются при нахождении процесса в любом из активных состояний. Аналогичное описание: ЕСЛИ (!(ПРОЦ <имя процесса> В СОСТ СТОП) && !(ПРОЦ <имя процесса> В СОСТ ОШИБКА)) – процесс и не в состоянии «нормальный останов» и не в состоянии «останов по ошибке»

Окончание табл.

№ п\п	Вариант написания		Семантика и использование
	Русскоязычный	*Англоязычный*	
31	**ПАССИВНОЕ**	**PASSIVE**	При операциях контроля состояния процесса замещает любое пассивное состояние процесса (состояние нормального останова или состояние контроля по ошибке). Служит для сокращения рутинных описаний. Например ЕСЛИ (ПРОЦ <имя процесса> В СОСТ ПАССИВНОЕ) {<действия>} – действия выполняются при нахождении процесса в любом из пассивных состояний. Аналогичное описание: ЕСЛИ ((ПРОЦ <имя процесса> В СОСТ СТОП) || (ПРОЦ <имя процесса> В СОСТ ОШИБКА)) – процесс в состоянии «нормальный останов» или в состоянии «останов по ошибке».
32	**ЗАЦИКЛИТЬ**	**LOOP**	Указание транслятору на то, что отсутствие выходов из текущего состояния контролируемое. Обход запрета на возможность зацикленных состояний.
33	**ЗНАКОВОЕ**	**SIGNED**	Спецификация знаковости целочисленной переменной, аналог резервированного слова signed языка Си.
34	**БЕЗЗНАКОВОЕ**	**UNSIGNED**	Спецификация знаковости целочисленной переменной, аналог резервированного слова unsigned языка Си.
35	**ПЕРЕЧИСЛЕНИЕ**	**ENUM**	Начало автоматической нумерации констант, аналог резервированного слова enum языка Си. Отличие в том, что имя перечисления не указывается.
36	**ФУНКЦИЯ**	**FUNCTION**	Начало декларации Си-функции, доступной к использованию в программе.
37	**ПУСТО**	**VOID**	Задание свободного типа переменной, или отсутствия аргумента при декларации функции, аналог резервированного слова void языка Си
38	**#СИ**	**#C**	Начало строки на языке Си.

Приложение 2

Экзаменационные билеты

Билет 1

1. Задача: логическое управление.
2. Проблема определения понятия «алгоритм».
3. Типовые задачи промышленной автоматизации.

Билет 2

1. Задача: операции с временными интервалами.
2. Машина Поста. Машина Тьюринга. Цели создания.
3. Специфика отладки управляющих алгоритмов. Типовые подходы.

Билет 3

1. Задача: логический параллелизм.
2. Модель конечного автомата.
3. Математическая модель гипераавтомата. Процессы и функции-состояния.

Билет 4

1. Задача: логическое управление.
2. Автомат Мили и Мура. Способы задания автоматов.
3. Средства обеспечения качества программного обеспечения.

Билет 5

1. Задача: операции с временными интервалами.
2. Совмещенный автомат. Этапы синтеза конечного автомата при схемной реализации.
3. Реализация конечного автомата и гипераавтомата средствами языков МЭК 61131-3.

Билет 6

1. Задача: логический параллелизм.
2. Программируемые логические контроллеры и специфика задач промышленной автоматизации.
3. Язык Рефлекс. Процессы и состояния.

Билет 7

1. Задача: логическое управление.
2. Степень соответствия модели конечного автомата специфике задач промышленной автоматизации
3. Язык Рефлекс. Дивергенция и конвергенция потока управления.

Билет 8

1. Задача: операции с временными интервалами.
2. Модифицированная модель конечного автомата.
3. Язык Рефлекс. Операции со временем.

Билет 9

1. Задача: логический параллелизм.
2. Логический параллелизм особенности и отличия от физического параллелизма.
3. Использование гипераавтомата при создании и отладке управляющих программ.

Билет 10

1. Задача: логическое управление.
2. Реализация логического конечного автомата средствами процедурных языков.
3. Реализация гипераавтомата средствами процедурных языков.

Билет 11

1. Задача: операции с временными интервалами.
2. Языки стандарта МЭК 61131-3.
3. Язык Рефлекс. Цели создания.

Учебное издание

Зюбин Владимир Евгеньевич

ПРОГРАММИРОВАНИЕ ИНФОРМАЦИОННО-УПРАВЛЯЮЩИХ СИСТЕМ НА ОСНОВЕ КОНЕЧНЫХ АВТОМАТОВ

Учебно-методическое пособие

Редактор С. В. Исакова
Верстка Ю. В. Баевой

Подписано в печать 15.09.2006 г.
Формат 60x84 1/16. Офсетная печать.
Усл. печ. л. 5,6. Уч.-изд. л. 6,7. Тираж 100 экз.
Заказ №

Редакционно-издательский центр НГУ
630090, Новосибирск-90, ул. Пирогова, 2.