Карпов В.Э.

# Автоматное программирование и робототехника

## Введение

Автоматное программирование — это некая общая парадигма программирования, суть которой заключается в том, что создаваемая программа рассматривается как реализация некоторого управляющего автомата. Цель работы – донести до читателя эти общие принципы, убедить его в том, что для создания сложных поведенческих программ, каковыми являются управляющие программы для роботов, удобно использовать понятие автомата. Как минимум с той точки зрения, что, во-первых, представление алгоритма работы системы управления в виде автомата очень наглядно (граф автомата – это весьма удобный для восприятия объект). Во-вторых, стараясь придерживаться формальных определений и механизмов, мы заставляем себя писать программу более строго и, главное, строить ее единообразным образом из элементарных составляющих. Если угодно, речь идет о повышении культуры программирования.

Разумеется, автоматное программирование – не единственная парадигма программирования в робототехнике. С одной стороны, существуют и более сложные, развитые, «интеллектуальные» во многих смыслах модели, нежели примитивный автомат. В них используется и ситуационное управление, и интеллектуальное планирование, и глубокие процедуры анализа, оценки и прогноза, другие очень интересные механизмы. С другой стороны, существует и некий радикальный минимализм, в котором все сводится к стимул-реактивному поведению, а вместо управляющего автомата используются, например, процедуры обработки прерываний.

В этом смысле автоматное программирование является «золотой серединой» между сложностью решаемых задач и ограничениями ресурсов бортовой системы управления (да и, откровенно говоря, простоты решения).

Для начала рассмотрим три типичные «робототехнические» задачи, имеющих классическую интерпретацию в терминах поколений систем управления – жесткое программное (первое поколение), стимул-реактивное (второе поколение) и управление в условиях неопределенности (третье, более близкое к «интеллектуальному», поколение).

**Задача 1. Фрагмент сборочной линии**

Пусть имеется техническое устройство (или комплекс устройств), которое должно осуществить некую последовательность операций: (1) взять заготовку и поместить ее под сверло, (2) просверлить отверстие, (3) поместить деталь обратно на ленту транспортера, см. Рис. 1. Необходимо организовать процесс управления таким устройством.

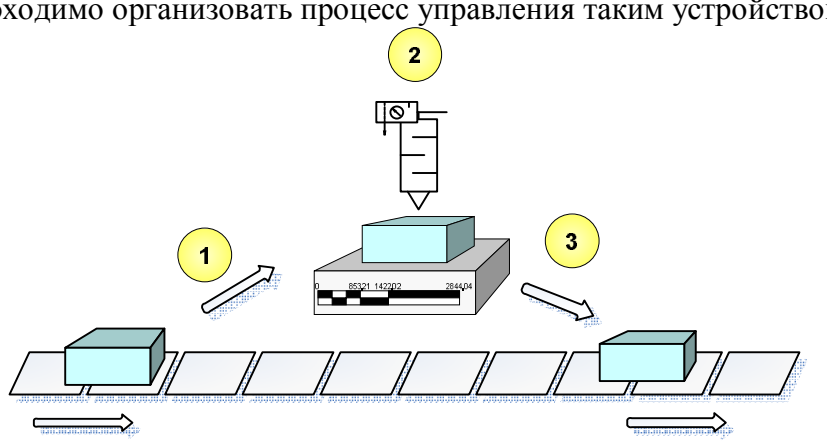



Рис. 1. Сборочная линия

**Задача 2. Робо-сумо**

На полигоне находятся два робота-соперника, см. Рис. 2. Задача «сумоистов» - вытолкнуть противника за пределы ринга. Роботы оснащены датчиками, они анализируют положение противника и осуществляют то или иное действие – ищут, атакуют, реагируют на границы ринга и т.п.

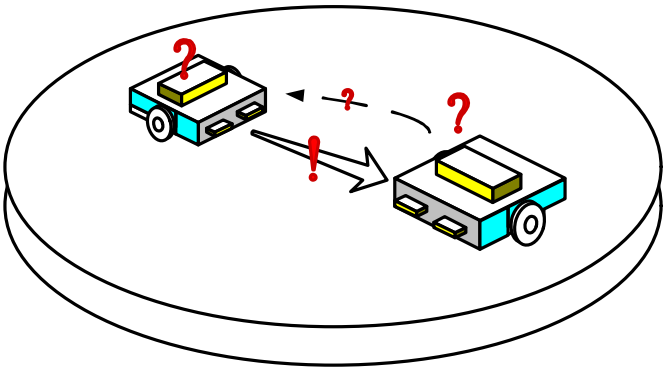


Рис. 2. Робо-сумо

**Задача 3. Соревнования УМНИК-БОТ**

Имеется поле с игровыми элементами (бруски, «свечи», площадки и т.п.), см. Рис. 3. Роботы должны выполнить ряд упражнений, за которые получают очки. Например, робот должен проехать по линии, нажать на «кнопку», задуть «свечи», «эвакуировать пострадавшего» и проч. При этом порядок выполнения операций произволен и, главное, на игровом поле находятся два робота, которые могут мешать друг другу. Необходимо создать алгоритм решения этой задачи, позволяющий за минимальное время заработать максимальное количество очков.

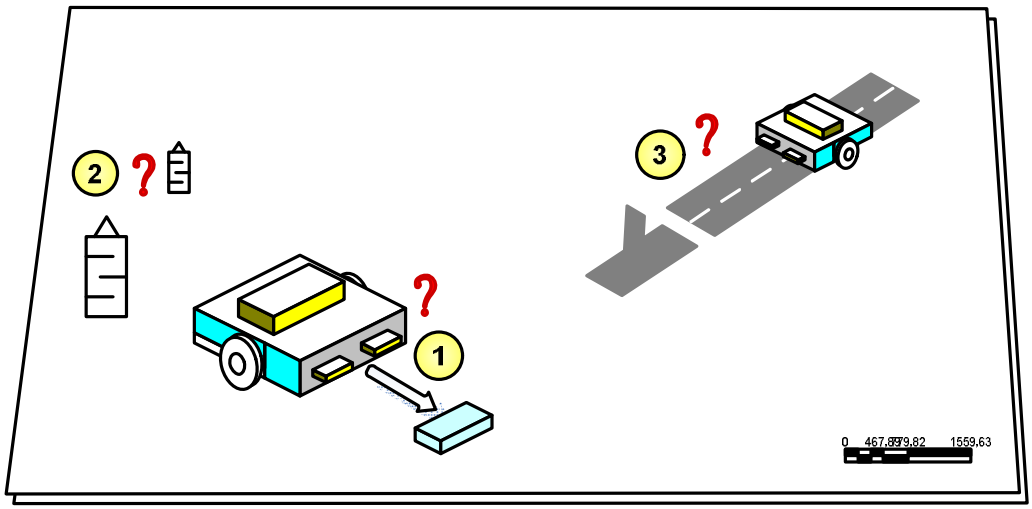



Рис. 3. Соревнования УМНИК-БОТ

Рассмотрим, в чем заключаются принципиальные особенности этих «действительно робототехнических» задач.

**Сборочная линия.** Для первой задачи характерно то, что она решается путем выполнения жесткой последовательности операций. Алгоритм ее решения принципиально линеен, т.к. полностью предсказуемы среда и условия выполнения операций. Фактически, несмотря на сложность ее отдельных компонент, это – задача для автоматического (т.е. работающего без участия человека в контуре управления) устройства. Интуитивно

представляется, что это не «настоящая робототехническая» задача, хотя реализующее ее устройство может называться промышленным роботом.

**Робо-сумо.** Здесь ситуация несколько сложнее. Робот-соперник активен, он тоже перемещается в пространстве, мешает, атакует, убегает и проч. Жесткая, линейная последовательность действий здесь уже не годится. Однако сам алгоритм может быть достаточно простым. Фактически, в основе алгоритма находится цикл (1) «сбор информации» – (2) «анализ и принятие решения» – (3) «выполнение действия». Например:

```
1. Сбор информации от датчиков
2. if(противник_спереди) then атаковать
3. if(противник_слева) then повернуть_налево
4. if(противник_справа) then повернуть_направо
5. ...
6. goto 1
```

Итак, в этой задаче необходим анализ ситуации и, в зависимости от результатов анализа, далее следует выполнение той или иной операции.

**УМНИК-БОТ.** В этой задаче мы имеем принципиально иную ситуацию. «Обычные» управляющие программы жесткой логикой и тем более – с линейной последовательностью действий, уже не годятся. УМНИК-БОТ, несмотря на простые правила и изрядный примитивизм, относится уже к категории сложных поведенческих задач. Следующий раздел будет посвящен именно анализу особенностей подобного рода задач.

## Поведенческие задачи

Начнем с достаточно очевидного тезиса о том, что наш робот представляет собой реальный физический объект, который должен работать в реальном физическом мире. На самом деле, из этого естественного утверждения следует, что:

1. Существуют неизбежные ошибки в управлении - робот не может всегда точно и предсказуемо отрабатывать свои действия.
2. Очевидно, что имеются помехи и периодически возникают различного рода непредвиденные факторы. Это – и неровности и неоднородности поверхности, и неточные показания датчиков, и многое другое.
3. Присутствуют факторы противодействия среды, например, в виде наличия робота-соперника.

Следствие этих факторов является то, что мир, в котором живее робот, сложен для того, чтобы точно формализовать его – мира – условия и поведение робота в нем. Это означает, что описывать поведение робота в виде жесткой, линейной схемы крайне нецелесообразно и неудобно.

Разумеется, общая схема «сбор информации» - «анализ и принятие решения» - «выполнение действия» останется неизменной. Робот должен уметь анализировать текущую ситуацию и, в зависимости от того, как была оценена эта ситуация, выполнять ту или иную процедуру или комплекс действий. Речь идет о том, как организовать анализ и принятие решений.

Характерной особенностью поведенческих задач является то, что прежде всего робот должен рассматривать общую задачу как множество более простых подзадач (как в УМНИК-БОТе) и уметь переключаться между ними. Например, если в данный момент невозможно решить задачу движения по полосе (кто-то или что-то мешает), то можно попробовать поискать свечи и т.п. Либо речь может идти просто о том, что робот должен решать подзадачи последовательно, но при этом ему полезно знать, что именно он сейчас решает или, иными словами, *в каком состоянии* он сейчас находится.

**Состояние**. Мы подошли к центральному понятию в решения поведенческих задач – к понятию состояния. Состояние – это все то, что определяет ситуацию, в которой находится робот. Состояние может определить этап решения задачи, совокупность

значений датчиков, некоторые внутренние параметры системы. Множество состояний образует то, что называется памятью в системе управления (в самом широком, смысле).

Роботам-сумоистам память, вообще говоря, не нужна. Точнее, глубина их памяти может быть равна единице. Дело в том, что роботы-сумоисты могут реализовывать то, что называется системой безусловных рефлексов или стимул-реактивным поведением (увидел – атаковал). Запоминать нечего. Но у них большую роль играет процедура анализа ситуации, которая может быть достаточно нетривиальной.

Управляющее сборочной линией устройство памятью уже обладает. Ему необходимо помнить, какой этап операции выполняется в данный момент. Но состояния изменяются сугубо последовательно, т.е эта память линейна. Анализ ситуации здесь, вообще говоря, большой роли не играет.

Именно тому, как создавать системы со сложной, нелинейной памятью, которые оперируют понятием состояния и, в зависимости от этого состояния и результатов анализа сигналов внешнего мира, выполняют те или иные действия, и посвящена настоящая работа.

Среди множества существующих механизмов и методов решения подобного рода сложных поведенческих задач особое место занимает т.н. *автоматное программирование*. (На самом деле, методов не так уж и много. Просто так принято говорить, что «среди множества…»). Прежде, чем говорить об этом автоматном программировании, рассмотрим, что такое автомат. Не в смысле – автоматическое устройство, а автомат, как математический объект.

## Автомат

Начнем с весьма формального определения основного объекта статьи - автомата.

**Определение**. *Конечный автомат (КА)* – это пятерка $A = (\Sigma, Q, q_0, T, P)$, где

- $\Sigma$ – входной алфавит (конечное множество, называемое также входным словарем);
- Q – конечное множество состояний;
- $q_0$ – начальное состояние ($q_0 \in Q$);
- T – множество терминальных (заключительных) состояний, $T \subset Q$;
- P – подмножество отображения вида $Q \times \Sigma \to Q$, называемое функцией переходов. Элементы этого отображения называются *правилами* и обозначаются как $q_i a_k \to q_j$, где $q_i$ и $q_j$ – состояния, $a_k$ – входной символ: $q_i, q_j \in Q, a_k \in \Sigma$.

В переводе на «человеческий» (инженерно-программистский язык) КА можно рассматривать как машину, которая в каждый момент времени находится в некотором состоянии $q \in Q$ и читает поэлементно последовательность ω символов из $\Sigma$, записанную на некоторой входной ленте. Если автомат в состоянии $q_i$ читает символ $a_k$ и правило $q_i a_k \to q_j$ принадлежит $P$, то автомат воспринимает этот символ и переходит в состояние $q_j$ для обработки следующего символа:

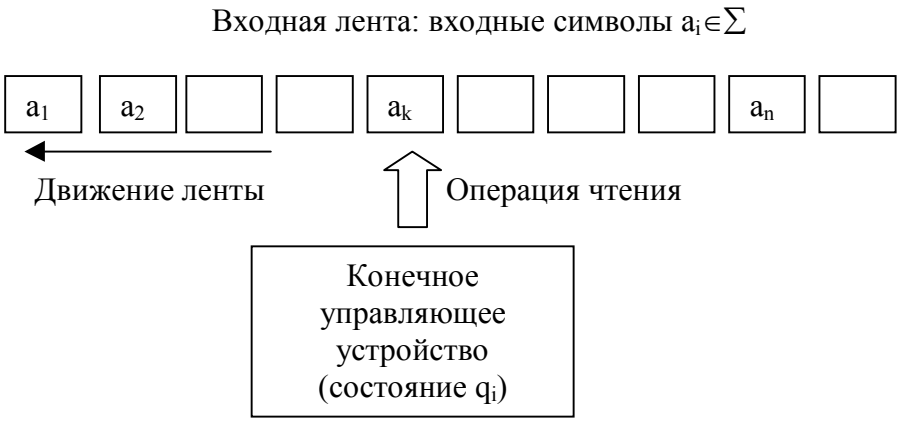



Рис. 4. Абстрактное изображение автомата

Таким образом, суть работы автомата сводится к тому, чтобы прочитать очередной входной символ и, используя соответствующее правило перехода, перейти в другое состояние.

Конечный автомат имеет очень удобное и наглядное представление. Его можно изобразить в виде ориентированного графа, вершинами которого являются состояния *Q*, а на дугах отмечаются символы входного алфавита.

## Задача анализа

На Рис. 5 изображен автомат с пятью состояниями – *S, A, B, C, D*. При этом его начальное состояние – *S*, а конечное – *D*.

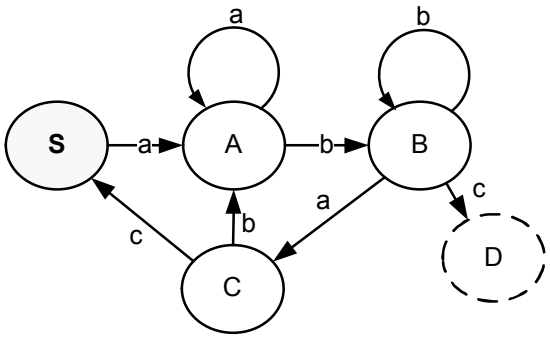



Рис. 5. Конечный автомат в виде графа

Этот автомат воспринимает символы *a, b* и *c*. Дуги изображают, в какое состояние перейдет автомат (какое состояние станет текущим в следующий момент времени) при поступлении соответствующего входного символа. Автомат стартует с состояния *S* и заканчивает работу, попадая в конечное состояние *D*. Если автомат попадет в состояние *D*, то будем считать, что входная последовательность распознана корректно. Если же не удалось совершить очередной переход или входная последовательность исчерпана, а мы не попали в заключительное состояние, то считается, что входная цепочка не распознана.

Например, автомат распознает такие последовательности, как:

*(a, b, c), (a, a, a, b, c), (a, a, a, a. b, b, b, c), ($a^n$, $b^m$, a, b, $b^k$, c), ($a^n$, $b^m$, a, c, $a^r$, $b^s$, c)*

и т.п. При этом автомат в конце оказывается в состоянии *D*.

А вот последовательности вида *(b, c), ($a^n$, $b^m$, a)* и проч. не приводят автомат в *D*. Такие последовательности для автомата являются «неправильными», т.е. он их не распознает.

Итак, конечный автомат является, вообще говоря, анализирующей или распознающей системой. Автомат – это один из основных механизмов анализа, например, в теории компиляторов. На основе конечных автоматов строятся, например, лексические анализаторы (сканеры).

## Автоматы с выходами

Нас, однако, интересуют не абстрактные задачи распознавания, а задачи управления. В конце концов, требуется писать управляющие, а не анализирующие программы (хотя, разумеется, управления без анализа не бывает).

Проблема заключается в том, что рассмотренный выше конечный автомат ничего «полезного» не умеет делать. Для того, чтобы не только анализировать, но и что-то делать, существуют различного рода модификации конечного автомата, среди которых нас интересуют т.н. автомат Мили и автомат Мура (автоматы второго рода).

Формально, речь идет об автоматах с *выходом*, которые определяются так:

$$A_{\text{вых}} = (\Sigma, Q, q_0, T, P, Y, \lambda)$$

Здесь к знакомой пятерке добавляются еще два элемента: выходной алфавит $Y$ и функция выходов $\lambda$. Считается, что автомат умеет не только воспринимать входную последовательность, но и выводить (печатать) некоторые символы (из алфавита $Y$) на т.н. выходной ленте. Функция выходов $\lambda$ определяет, когда и при каких условиях автомат будет эту печать осуществлять.

Собственно, «печатать» что-то нам не интересно. Будем интерпретировать вывод информации, как выполнение некоторых процедур вообще. Для нас это – выполнение двигательных функций, сложных или простых поведенческих процедур и т.п.

В зависимости от того, как устроена функция $\lambda$, различают упомянутые выше автоматы Мили и автоматы Мура.

**Автомат Мили.** Выходная последовательность зависит от состояния автомата и входных сигналов. Иными словами, автомат совершает некоторое действие *на переходе*, т.е. в зависимости от того, в каком состоянии находится автомат и какой символ имеется на входе, автомат переходит в другое состояние и при этом совершает какое-то действие:

$$\lambda : Q \times \Sigma \to Y$$

или:

$$y(t+1) = \lambda(q(t), x(t))$$

Здесь *y(t+1)* – совершаемое действие (вызов процедуры), *q(t)* и *x(t)* – текущие состояние и входной символ соответственно.

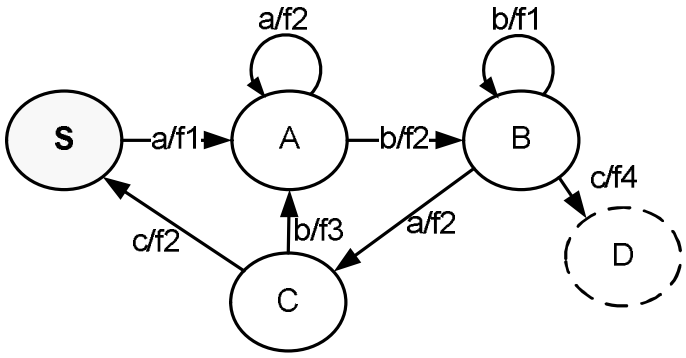



Рис. 6. Автомат Мили

Пометки дуг на Рис. 6 *"x/f"* интерпретируются так: $x$ – входной символ, $f$ – некоторая исполняемая процедура. Например, если мы находимся в состоянии $A$, а на входе – символ $b$, то мы перейдем в состояние $B$ и при этом выполним процедуру *f2*.

**Автомат Мура.** Его функция выходов выглядит иначе:

$$\lambda : Q \to Y,$$

т.е.

$$y(t+1) = \lambda(q(t))$$

Это означает, что выходное значение сигнала зависит лишь от текущего состояния данного автомата.

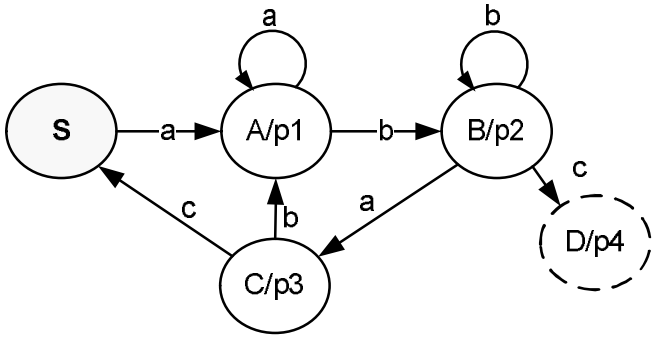



Рис. 7. Автомат Мура

Интерпретация графа такова: в вершинах изображаются не только имена состояний, но и имена выполняемых процедур. Например, при попадании в состояние B, вызывается процедура *p2*.

Важно, что, несмотря на внешнюю похожесть изображений на Рис. 6 и Рис. 7, на них изображены принципиально разные по своему поведению автоматы.

На самом деле, автоматы Мили и Мура равнозначны по своим возможностям. Более того, для любого автомата Мура существует эквивалентный ему автомат Мили и наоборот. Например, автомат Мура может быть преобразован в автомат Мили путем добавления ряда внутренних состояний.

Выбор конкретного вида автомата определяется спецификой решаемых задач, т.е. нашими представлениями о том, в каких терминах нам удобнее рассуждать – выполнять процедуры либо в состояниях (автомат Мура), либо на переходах (автомат Мили).

**Важное замечание.** Прежде, чем переходить к конкретике программной реализации автоматов, отметим один принципиальный момент. Рассмотренные выше автоматы (и с выходом, и без выхода) представляют собой некие формы описания алгоритмов, т.е. являются частными случаями такого фундаментального понятия, как Машина Тьюринга.

Именно в виде автомата мы и будем описывать далее структуру и логику поведения системы управления. Но прежде обратимся к некоторым сугубо техническим вопросам.

# Задача регламента УМНИК-БОТ

## *Правила соревнований*

УМНИК-БОТ – это парные соревнования роботов, заключающиеся в выполнении игроками ряда простых упражнений. За выполнение упражнений - успешное решение задач в наших терминах– роботу начисляются очки. Необходимо выполнить максимальное количество упражнений за ограниченное время. При этом запрещено столкновение с роботом-противником, который параллельно также решает свои задачи. Ниже приведены сами упражнения:

1. «Эвакуация». Сбоку от линии трассы находится брусок («пострадавший»), который необходимо задвинуть в некоторую отмеченную на полигоне область.
1. «Кнопка». Необходимо задвинуть игровой элемент («Кнопка»), толкая его, например, корпусом. К «кнопке» от стартовой зоны ведет трасса – черная линия.
2. «Свеча». На полигоне расположено некоторое количество «свечей», которые необходимо «задуть». Для этого робот должен уметь находить их и, выйдя на цель, включать вентилятор.
3. «Флаг». В специально обозначенной зоне робот должен «поднять флаг», т.е. запустить некий сервопривод. Вне этой зоны понятие флага не засчитывается.

### *Робот*

Предположим далее, что у нас имеется робот, удовлетворяющий условиям соревнований. Кроме того, будем считать, что мы умеем по отдельности решать ряд частных задач. А именно:

- совершать простые действия типа движения в заданном направлении и разворотов;
- ехать по линии, ориентируясь по датчикам;
- считывать значения с установленных на роботе дальномеров;
- реагировать на препятствия;
- обнаруживать источники инфракрасного излучения («свечи»);
- включать вентилятор;
- поднимать флаг;

и т.п.

Это – наше базовое программное обеспечение, набор функций, из которых мы будем строить основную управляющую программу.

Очевидно, что наличие самого робота и его базового программного обеспечения – это уже очень много. В принципе, этим можно было бы попробовать и ограничиться, написав программу с жесткой логикой, особенно, если нам будет известно заранее расположение игровых элементов. Но на практике, как уже говорилось выше, такие жесткие алгоритмы дают обычно плохие результаты. Поэтому продолжим разговор об автоматном программировании.

## Управляющий автомат

Итак, будем описывать нашу управляющую программу в виде автомата. Как уже ясно, автомат – это система с памятью, описывающая поведение некоторого объекта (робота, агента, технического устройства) в терминах состояний.

Подчеркнем еще раз, что мы рассматриваем память автомата в виде определения того, что должен выполнять робот не только и не столько в некоторый момент времени, но прежде всего в терминах состояний.

### *Этапы решения*

Работа автомата заключается в том, чтобы, проанализировав ситуацию, определить, в какое состояние он должен перейти и что ему делать в этом состоянии. Память в виде множества состояний должна определить, что делать в случае изменившихся условий (помехи в виде робота-соперника, сбой датчиков, изменение игровой ситуации), на какую задачу переключиться, куда вернуться после отработки рефлексов и проч.

Существуют две основные стратегии решения задачи:

1. Жесткая стратегия, при которой последовательность решаемых задач строго зафиксирована.

2. Ситуационная стратегия, при которой робот выбирает, какую задачу следует решать в данный момент времени, исходя из анализа текущей ситуации, его приоритетов, планов и проч.

Далее мы будем рассматривать реализацию именно более простой жесткой стратегии.

## Фиксированная стратегия

Определим стратегию поведения робота как следующую жестко заданную последовательность выполнения игровых задач:

1. «Эвакуация».
2. «Кнопка».
3. «Свеча».
4. «Флаг».

Изобразим эту последовательность в виде графа. Это достаточно наглядно, но пока несколько бесполезно.

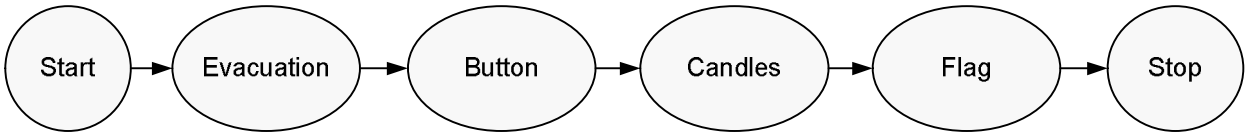



Рис. 8. Граф стратегии решения задачи

А далее мы начнем уточнять это граф, превращая его «крупные» состояния в множество более детальных состояний. Такая последовательная детализация и приведет нас, в конце концов, к конечному управляющему автомату, который мы и реализуем в виде программы (автомат, напомним, – это форма представления алгоритма).

Прежде, чем начать детализацию, мы должны определиться с тем, что умеет делать автомат, т.е. какие он выполняет функции и какие входные сигналы умеет воспринимать. Наш начальный список может выглядеть так:

**Действия**:

1. Движение по линии (Line ()).
2. Процедура эвакуации (Evac()).
3. Процедура нажатия на кнопку (Button()).
4. Поиск свечи (candleSearch()).
5. Задувание свечи (Blow()).
6. Поиск зоны приема (SearchZone()).
7. Процедура поднятия флага (Lift())
8. …

**Обнаруживаемые объекты:**

1. Брусок («пострадавший») (“:victim”).
2. Кнопка (“:button”).
3. Препятствие спереди (“:obstacle”).
4. Свеча (“:candle”).
5. …

В скобках указаны имена процедур и наименования сигналов.

Это – очень условные рассуждения. Мы поместили в один список и сложные поведенческие процедуры (например, процедура эвакуации), и элементарные действия (задувание свечи). Список неизбежно будет далее уточняться, систематизироваться, дополняться. Пока же вернемся к нашему графу.

Распишем для примера два состояния – Evacuation и Button, отвечающих за реализацию процедур «Эвакуация» и «Кнопка» соответственно. Будем считать, нам

удобнее работать с автоматом Мура, у которого выполняемое действие связано с состояниями.

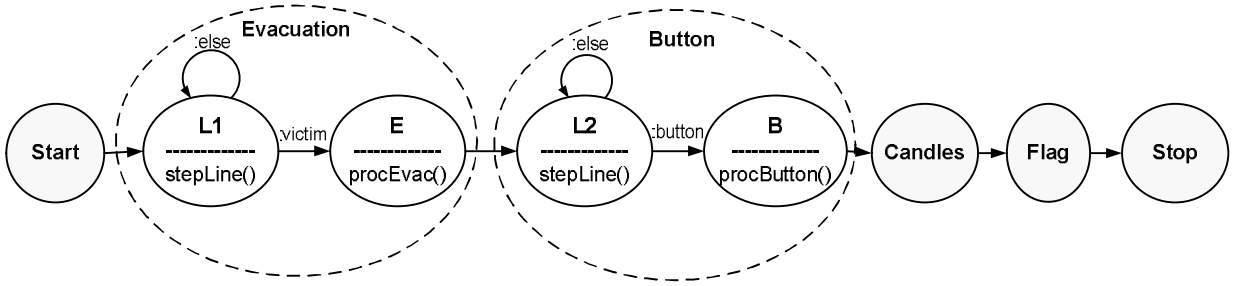



Рис. 9. Более детальное представление управляющего графа

Посмотрим, во что превратились исходные состояния. *Evacuation* на данном этапе мы расписали в виде двух состояний – *L1* и *E*. Логика рассуждений заключается в том, что мы представили в виде последовательности двух операций – движении по полосе *stepLine()* и выполнения собственно процедуры толкания бруска («пострадавшего») *procEvac()*.

Каким именно образом выполняется процедура движения по полосе *stepLine()* – не существенно. Мы договорились о том, что робот делать это как-то умеет. Важнее, что появились пометки на дугах графа. Они обозначают, какой символ имеется на входе автомата. Пометка *":victim"* означает, что робот обнаружил брусок. После этого он переходит в состояние *E*, начиная выполнение процедуры эвакуации. Пометка *":else"* означает, что нас не интересуют никакие остальные входные сигналы. Отсутствие пометок на дуге перехода из состояния *E* в состояние *L2* означает, что мы безусловно осуществим этот переход после завершения процедуры *procEvac()*.

Аналогичным образом интерпретируются переходы и процедуры, связанные с вершинами *L2* и *B*.

**Примечание**. Уже понятно, что множество $Q$ состояний автомата – это вершины *Start, L1, L2* и т.д., а множество $\Sigma$, т.е. входной алфавит автомата – это сигналы *":victim", ":button", ":obstacle", ":candle"* и проч.

Разумеется, такая «детализация» не является достаточной. Действительно, а что делать, если во время движения по полосе будет обнаружено препятствие? Или как поступить, если не сработает датчик обнаружения пострадавшего и сигнал *":victim"* мы не получим? Тогда надо дополнять автомат состояниями, условиями переходов и проч. Но уже сейчас видится одна очень важная и полезная особенность автомата: он (а вместе с ним – и сама управляющая программа) может развиваться *постепенно*, становясь все более гибким и адекватным.

## Особенности программной реализации

Упрощенный алгоритм работы программы выглядит так:

```
Используемые переменные:
      Q – номер или имя состояния автомата
      X – входной вектор

1. Setup()          -- Инициализация системы
2. Q := Start       -- Робот вначале находится в состоянии Start
3. X := ReadSensors() -- Чтение информации от датчиков (формирование вектора
                         входных символов X)
4. Q' := F(Q, X)    -- Определение нового состояния Q', исходя из того, в каком
                         состоянии  находился автомат и каков входной символ.
5. Q := Q'          -- Переход в это состояние
6. ExecuteProc (Q)      -- Выполнение процедуры, связанной с состоянием Q
                         (у нас – автомат Мура)
7. goto 3
```

Рис. 10. Основной алгоритм

Прокомментируем некоторые его шаги.

**П.3. Чтение информации от датчиков.** Эта достаточно очевидная на первый взгляд процедура заключается в том, что мы формируем, на самом деле, не один-единственный входной символ *x*, а некоторое множество входных сигналов, которое обозначено как ***X***. Это связано с тем, что робот воспринимает сигналы от множества датчиков. И в этом – существенное отличие от того, что говорилось в определении автомата.

Вектор ***X*** формируем следующим образом. Введем множество переменных, отвечающих за регистрацию того или иного события, значения сигнала и т.п. При этом будем считать, что нам вполне достаточно будет переменных-флагов (или предикатов), которые принимают значения только 0 или 1. Далее, «пронумеруем» эти флаги. Для этого их можно хранить в виде вектора, порядковые номера элементов которых будут соответствовать номеру входного признака-сигнала. Это и будет вектор ***X***. Например, элемент вектора *X[0]* будет хранить признак того, что обнаружен брусок (сигнал ":victim" на графе), *X[1]* отвечает за признак ":button" и т.п.

**П.4. Определение нового состояния.** Мы должны выяснить, в какое состояние автомат перейдет из текущего. При этом здесь есть одна тонкость. В «каноническом» автомате на входе в каждый момент времени имеется один символ, поэтому проблем с определением перехода нет. Здесь же мы можем иметь дело, как было сказано выше, с несколькими входными сигналами, существующими в векторе ***X*** одновременно. Например, мы решили уточнить поведение робота в состоянии *L1*., введя реакцию на возможное препятствие по ходу движения. В общем-то, это вполне хорошая идея, но может возникнуть ситуация, когда робот увидит одновременно и препятствие, и «пострадавшего». Возникает вопрос, какой переход автомату следует отработать?

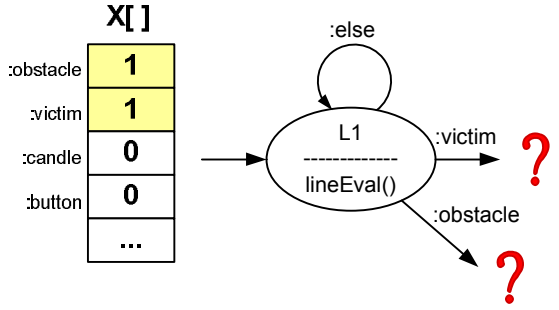



Рис. 11. Одновременное появление сигналов *":obstacle"* и *":victim"*

Повторим, что в «нормальном» автомате такая ситуация невозможна по определению самой модели. Здесь же мы имеем ситуацию, определяемую нашими сугубо практическими соображениями. Исходя из таких же соображений, можно ввести понятие приоритета перехода. Т.е. указать в программе, что переход по символу *":obstacle"* (препятствие) важнее.

И последнее замечание по этому шагу. Сама реализация функции переходов *F(Q, X)* может быть основана на использовании т.н. матрицы переходов ***M***. А именно - массива, содержащего в качестве значений имена (номера) состояний. Строки массива ***M*** соответствуют входным символам, а столбцы – именам (номерам) состояний.

***M:***

| | Start | L1 | E | L2 | … |
|---|---|---|---|---|---|
| :obstacle | | | | | |
| :victim | | E | | | |
| :candle | | | | | |
| :button | | | | B | |
| … | | L1 | | | |
| … | | | L2 | | |

**П. 6. Выполнение процедуры, связанной с текущим состоянием Q.** Особенность этого пункта в том, что наша программа должна работать циклически. Мы не имеем права уйти в выполнение некоторой «долгой» процедуры. Ситуация в мире может поменяться и робот должен вовремя отследить эти изменения, отреагировать на них. Собственно, ради этого и создавался автомат. Цикличность означает, что шаги 3-7 будут выполняться с большой частотой. Скажем, если нас интересует адекватность поведения робота, то, чтобы не пропустить сигнал от датчика, мы должны выполнять этот цикл с частотой, скажем, 100 Гц (0.1 сек. – достаточная скорость реакции робота). А эта цикличность означает т.н. реентерабельность процедур, т.е. повторность входа в процедуры.

## Реентерабельные процедуры

Реентерабельная процедура, или процедура с повторным входом, – это некоторая последовательность операций, которую можно выполнять, используя внешний цикл. Например, процедура движения по линии может выглядеть так:

```
void stepLine(void)
{ if(SensLeft && SensRight) GoFwd();
  else
  if(SensLeft) GoLeft();
  else
  if(SensRight) GoRight();
}
```

Это – типичная реентерабельная процедура, которую мы будем вызывать циклически на шаге 6.

Требования к реентерабельным процедурам очевидны:

1. Процедура должна осуществлять управление путем ее многократного вызова (мы вызываем ее циклически извне).
2. Процедура должна быть «короткой» (чтоб не «застрять» при ее выполнении). Это означает, что внутри нее желательно избегать циклов, вызовов других сложных процедур и проч.

Иногда оказывается, что реализовать чисто реентерабельную процедуру либо сложно, либо вовсе невозможно. Например, иногда перед выполнением основного цикла процедуры управления необходимо проинициализировать какие-нибудь параметры, совершить подготовительные действия. Или необходимо выполнить некоторые полезные действия по завершении управления. Тогда мы можем сказать, что каждое действие (или

поведенческая процедура) в общем случае дополняется «прологом» (процедурой инициализации) и «эпилогом» (завершающей процедурой):

*startProc() – stepProc() – endProc().*

Здесь *stepProc* – это реентерабельная процедура, вызываемая многократно, а *startProc* и *endProc* – однократно вызываемые пролог и эпилог процедуры соответственно.

Пролог и эпилог, равно как и любые прочие процедуры «одноразового» применения (т.е. вызываемые однократно) также прописываются в управляющем автомате. А «платой» за это является введение дополнительных фиктивных состояний. Фиктивность означает, что эти состояние не оказывают влияние на логику работу программы, имеют безусловные переходы и нужны лишь для того, чтобы запустить соответствующие процедуры.

Пусть имеется некоторая процедура *Proc* следующего вида:

```
void Proc(void)
{
  Init(); // Пролог. Выделится в процедуру startProc()
  while(…)
  {
    Ctl(); // Основная часть. Выделится в процедуру stepProc()
  }
  Finish(); // Эпилог. Выделится в процедуру endProc()
}
```

В этой процедуре имеется блок инициализации (пролог) *Init()*, основная часть, выполняемая циклически, и завершающий блок (эпилог) *Finish()*.

На Рис. 12 показан пример того, во что приходится превращать некоторое состояние *C* для реализации этой процедуры управления.

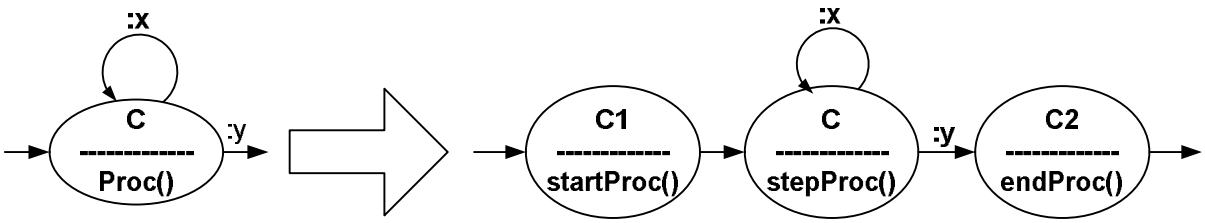



Рис. 12. Введение фиктивных вершин для выполнения пролога и эпилога

## Переходы по условию «иначе»

Реализуя матрицу переходов нашего автомата, мы сталкиваемся с одним весьма неприятным моментом. Речь идет о переходе по условию «иначе» или “:else” в наших обозначениях.

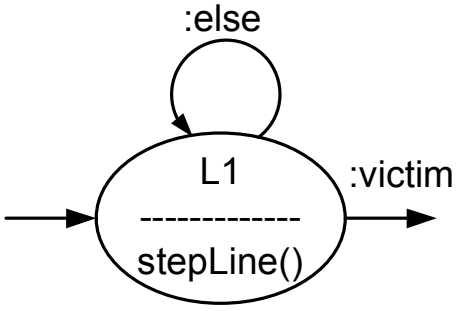
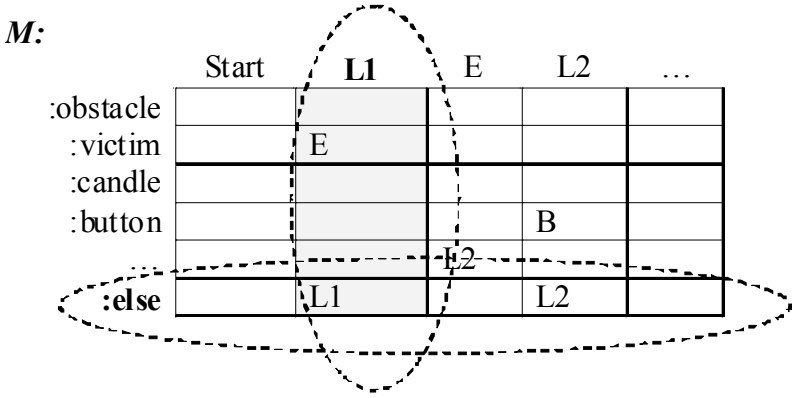



Рис. 13. Переход по входному символу “:else”

С понятийной точки зрения ситуация на Рис. 13 весьма прозрачна: автомат перейдет в следующее состояние, когда придет символ *“:victim”*. В противном случае он останется в текущем состоянии *L1*. Именно это и означает пометка *“:else”*на дуге. Однако

на самом деле *":else"* должен означать некий вполне определенный входной символ, пусть он и является для нас фиктивным. Откуда же этот символ возьмется?

Возможен следующий вариант разрешения этой ситуации. Рассмотрим процедуру определения следующего состояния автомата *F(Q, X)*. В матрицу переходов автомата добавим еще одну строку, которая будет отвечать за это наше «иначе».




Пусть наше текущее состояние – это *L1*. Это означает, что далее мы будем работать только с соответствующим этому состоянию столбцом.

Мы уже обсуждали выше устройство вектора входных сигналов ***X***. В цикле пройдемся по всем его элементам. Если будет обнаружен ненулевой элемент *X[i]*, то следующим состоянием станет то, которое было определено в i-й строке столбца *L1*. Если же мы не обнаружим ни одного ненулевого элемента в векоре *X*, либо значение следующего состояния будет пустым (пустой элемент матрицы ***M[i][L1]***), то тода мы берем содержимое элемента матрицы, соответствующий строке «иначе». Это и будет считаться переходом по символу *":else"*.

Ниже приведен пример функции, определяющей переход в новое состояние (функция F(Q, X) основного алгоритма).

```
int F(int q, int X[])
// q – номер текущего состояния
// X – вектор входных сигналов
{
  for(int i=0;i<DIM_X;i++)
    if(X[i]!=0 && M[i][q]!=0) // Нашли «нормальный» переход
      return M[i][q];
  // Не нашли. Возвращаем переход по условию «иначе»
  return M[DIM_X][Q];
}
```

Здесь предполагается, что вектор входных сигналов содержит `DIM_X` элементов, а количество строк в матрице переходов ***M*** равно (`DIM_X+1`), т.к. необходима еще одна строка для фиктивного перехода по условию «иначе».

## Время

Добавим большей адекватности поведению нашему роботу. Речь идет о том, что, рассматривая фрагмент управляющего графа, отвечающего за движение, мы задумались о времени. Время, так или иначе, должно было появиться в наших рассуждениях. Например, рассмотрим процедуру движения робота по полосе. Выход из состояния L1, согласно графу автомата, произойдет тогда, когда мы обнаружим препятствие (переход *":victim"* на Рис. 14). Но что делать, если препятствие по каким-то причинам не будет обнаружено? Причиной тому может быть, например, сбой датчика или поведение соперника. Как бы то ни было, было бы полезно что-то предпринять, если знать, что истек некий временной интервал. И если такое случилось, то начать выполнение необходимых действий, т.е.

перейти в этом случае в какое-нибудь соответствующее состояние. Иначе мы рискуем ехать прямо «до бесконечности».

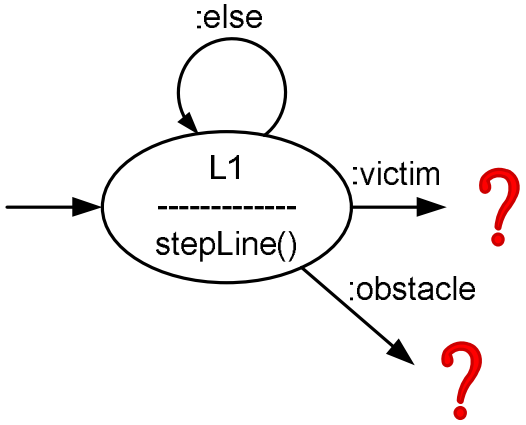



Рис. 14. Время истекло. Надо что-то делать

Введение понятия времени желательно осуществить так, чтобы не нарушать нашей автоматно-процедурной логики. Для этого можно создать некий аналог будильника. Наш будильник основан на реализация нескольких простых механизмов.

Во-первых, время необходимо уметь измерять. Например, можно завести некоторую глобальную переменную (счетчик времени), значение которой будет увеличиваться в некоторой процедуре обработки прерываний от таймера. Или можно воспользоваться неким системным механизмом, связанным с системными часами. В любом случае, можно считать, что нам необходима некоторая глобальная переменная – счетчик *TCNT*.

Во-вторых, изготовим некоторую функцию вида ***SETALARM***, задача которой обнулить этот счетчик, а значение параметра этой функции будет соответствовать моменту срабатывания будильника. Таким образом, *SETALARM* установит будильник на некоторое время.

В-третьих, добавим в вектор входных сигналов ***X*** еще один элемент. Этот элемент вектора и будет хранить признак того, что значение счетчика *TCNT* превысило установленное функцией *SETALARM* значение (сработал будильник - *":ALARM"*). И, в-третьих, после того, как мы «услышали» будильник и отреагировали на него переходом по символу *":ALARM"*, будильник надо выключить, т.е. прописать в ***X[S_ALARM]*** значение 0. Этим будет заниматься процедура ***ALARMRESET***. Тогда мы получим схему, показанную на Рис. 15.

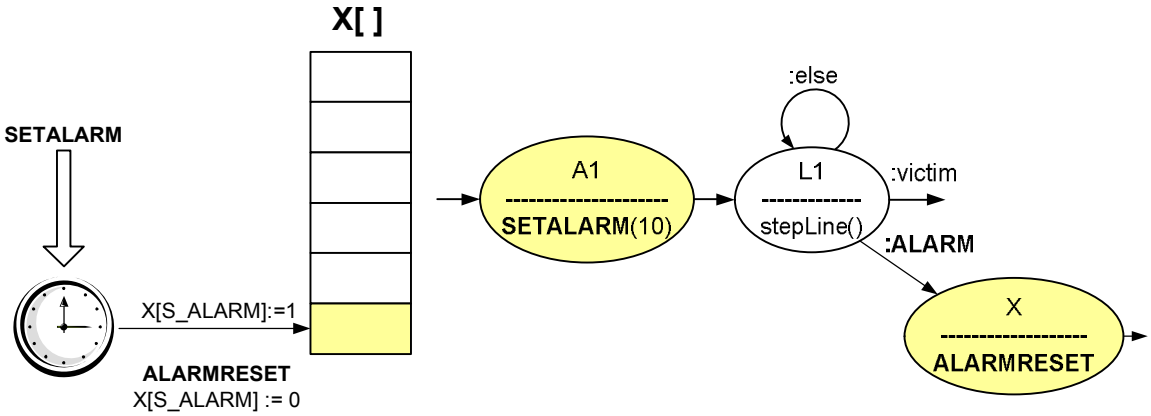



Рис. 15. Добавление «будильника»

## Вызовы подпрограмм

Продолжим наши «улучшения» программы. И вновь рассмотрим поведение робота в состоянии движения по линии L1. Зададимся вопросом: что делать, если по ходу движения обнаружится препятствие? Понятно, что в этом случае необходимо отработать маневр объезда этого препятствия, а затем вернуться обратно к процедуре движения по полосе. И мы получим примерно такую схему:

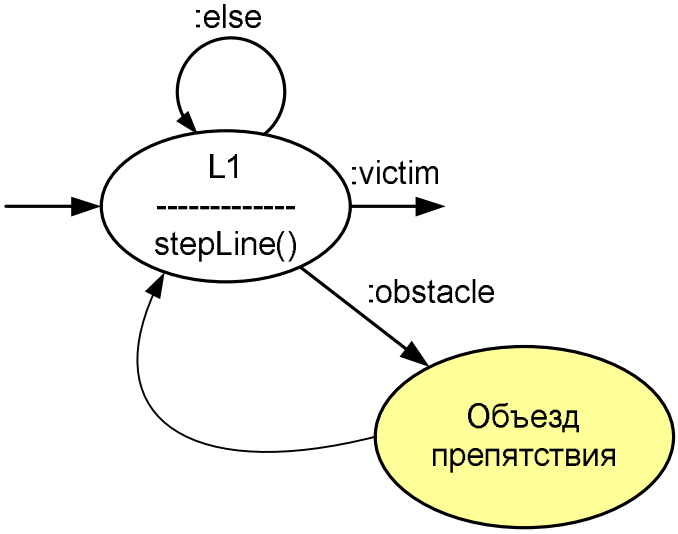



Рис. 16. Дополнение схемы подграфом «Объезд препятствия»

Здесь предполагается, что вершина «Объезд препятствия» - это на самом деле множество вершин управляющего графа. Все достаточно естественно и очевидно. Но проблемы начинаются тогда, когда мы захотим аналогичным образом «улучшить» вершину L2. Очевидно, что нам не остается ничего другого, как продублировать тот же подграф «Объезд препятствия». И сделать то же самое во множестве иных мест. Что, разумеется, не может устроить нас никоим образом. На самом деле, подграф «Объезд препятствия» - это явный аналог того, что в «нормальном» программировании называется подпрограммой. Здесь должно быть то же самое, только в роли подпрограмм выступают подграфы.

Итак, рассмотрим подграф «Объезд препятствия». Это будет изолированный фрагмент, со своим началом – единственной входной вершиной (*DStart*) – и концом – выходной вершиной (*DEnd*), см. Рис. 17.

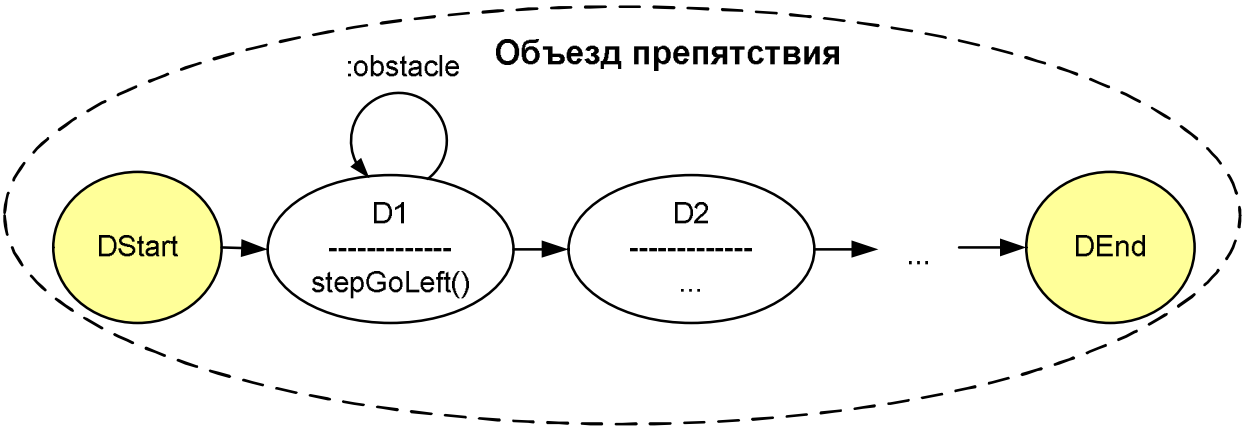



Рис. 17. Граф процедуры «Объезд препятствия»

А теперь надо разобраться с тем, как передать управление этому графу, а затем вернуться в нужное место (состояние или вершину графа).

**Вызов.** Для этого надо написать процедуру ***CALL***, а в качестве параметра указать имя вершины, которая станет текущей на следующем шаге. В данном случае, для отработки процедуры объезда должен быть сделан вызов *CALL(DStart)*. Текущим состоянием станет *DStart* и автомат продолжит свою работу именно с нее. Остается вопрос, куда и каким образом мы будем возвращаться.

**Возврат**. Вообще, для возврата необходимо сначала где-то сохранить имя вершины, в которую нам следует вернуться. И сделать это в самом начале выполнения процедуры *CALL*. Далее, граф-подпрограмму следует завершать состоянием, вызывающим процедуру возврата из подпрограммы ***RETURN***. Процедура *RETURN* смогла бы использовать это сохраненное значение для того, чтобы присвоить его переменной, хранящей номер текущей вершины. В «обычных» системах возврат из подпрограммы осуществляется на *«следующую»* инструкцию. Это достаточно естественно, т.к. «обычная» программа линейна, существует счетчик инструкций и т.п. Но у нас «следующей» инструкции нет. У нас есть множество вершин, из которых ведут дуги. Возвращаться в ту же вершину, из которой мы вызвали «подпрограмму», смысла еще меньше. Проще всего поступить так: в процедуру *CALL* добавить еще один параметр – имя состояния, в которое будет передано управление после отработки процедуры *RETURN*. Тогда вызов *CALL* будет выглядеть так

***CALL(GStart, RetCond)***

где *GStart* – это имя стартовой вершины подграфа, *RetCond* – имя состояния, в которое мы вернемся.

Разумеется, явное указание адреса возврата – это нонсенс и дикость с точки зрения линейной программы, но, во-первых, это будет работать, а во-вторых, реализация такого механизма крайне проста.

**Рекурсии**. Естественно возникнет вопрос о том, что делать в той ситуации, когда нам захочется вызывать подграф внутри подграфа. Т.е. из одной процедуры обработки вызывать другую. Или вдруг возникнет противоестественное желание вызвать обработку себя же, т.е. осуществить рекурсивный вызов. Тогда нам придется завести стек вызовов, и процедуры *CALL* и *RETURN* должны будут работать со *стеком*. В этом стеке будут храниться адреса возвратов.

Итак, описанный механизм действительно похож на то, как происходит вызов подпрограмм в привычном программировании.

## Непредвиденные сложности

**Пропуск перехода.** Предположим, что у нас имеется подграф, изображенный на Рис. 18. В вершине ***A*** происходит передача управления подграфу ***G***, при этом возврат будет осуществлен в вершину ***B*** (процедура *CALL(G, B)*). Причем, обратите внимание, из вершины ***B*** идет *безусловный* переход в вершину ***C***.

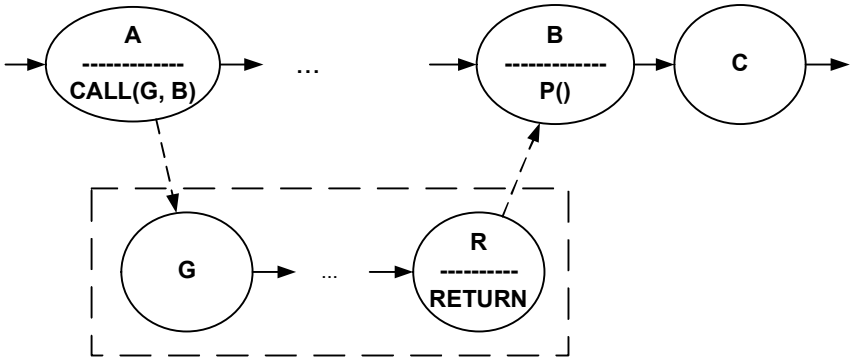



Рис. 18. Вызов «подпрограммы»

Казалось бы, что никаких проблем здесь быть не должно, однако попытка реализации такого управления приведет к неожиданному эффекту: процедура ***P()***,

связанная с вершиной ***B*** не будет выполнена. Мы увидим, что после отработки процедуры RETURN автомат окажется в состоянии ***C***. Иными словами, произойдет пропуск перехода.

Связано это вот с чем. Если внимательно посмотреть на основной алгоритм работы автомата, Рис. 10, то обнаружится такая неприятность. Пусть в данный момент мы находимся в состоянии ***R***. На шаге 6 алгоритма будет отработана процедура *RETURN* и текущим состоянием станет состояние ***B*** (пока все идет хорошо).

```
4. Q' := F(Q, X)
…
6. ExecuteProc(Q)
```

Однако на следующем цикле начнет выполняться сначала шаг 4 – определение нового состояние. И таким состоянием, согласно структуре автомата, станет ***C***, т.е. процедура ***P()*** вызвана не будет. Это связано с тем, что на шаге 6 произошло «нештатное» изменение состояния, что является проявлением т.н. побочного эффекта процедуры *ExecuteProc()*.

С этим двойным изменением состоянием можно справиться следующим образом. Во-первых, шаги 4 и 6 следует поменять местами. Т.е. сначала автомат будет отрабатывать процедуру, связанную с текущим состоянием, а затем только осуществлять переход.

Во-вторых, следует анализировать значение, возвращаемое *ExecuteProc()*. Речь идет о том, что, если внутри этой процедуры происходит изменение состояния, т.е., фактически, изменение порядка выполнения программы, то шаг вычисления нового состояния следует пропустить.

Тогда мы получим такую схему:

```
1. Setup()
2. Q := Start
3. X := ReadSensors()
4. if ExecuteProc(Q) = 1 then
5.   Q := F(Q, X)
6. goto 3
```

Рис. 19. Результирующий алгоритм работы автомата

## От матрицы переходов к списку рёбер

Матрица переходов ***M*** – это вполне удобный и наглядный объект. Однако есть одна сугубо техническая проблема: матрица переходов требует излишне много памяти. Действительно, наш граф алгоритма – очень разреженный. В основном вершины образуют линейные цепочки и если приглядеться к матрице ***M***, то в ней будет очень много пустых позиций. Другой формой представления графа является список рёбер. Вообще, список рёбер – это множество пар: вершина-источник и вершина-приемник. Мы же будем использовать список помеченных рёбер, где пометками являются условия перехода.

Таким образом, в общем виде список рёбер будет описываться множеством троек вида:

*<src, dest, condition>*

«Затраты» на хранение списка рёбер будут существенно ниже, нежели в случае использования матрицы переходов, а работать со списком не многим сложнее, нежели с матрицей.

Для оценки. В одном из автоматов имелось 32 состояния и 14 входных сигналов. Количество элементов в матрице переходов составляло 32*14=448. С другой стороны, всего в этом графе было 32 дуги (ребра). Хранение списка обошлось в 32*3=96 элементов. Разница существенна. И чтобы закончить агитацию за список ребер, приведем фрагменты программы, реализующей автомат, представленный списком рёбер.

Описание рёбер удобно представить в виде структуры:

```
// Ребро графа
struct TArc
{
  byte src, dest; // источник, приемник
  int cond;       // условие перехода (пометка дуги)
};
```

Собственно список рёбер (фрагмент):

```
#define NumArcs 32
TArc arc[NumArcs] =
{
  {qStart,qE0, sigALWAYS},

  // Evacuation
  {qE0,qE1, sigALWAYS},

  {qE1,qE1, sigELSE},
  {qE1,qE2, sigVictim},
  {qE1,qE3, sigAlarm},
  {qE1,qE4, sigObstacle},
  {qE3,qB0, sigALWAYS},
  ...

};
```

Здесь сигнал `sigALWAYS` обозначает безусловный переход.

Функция перехода выглядит достаточно просто. Она приведена целиком:

```
int FM(void)
{
  for(int i=0;i<NumArcs;i++)
    if(arc[i].src==Q)
    {
      int cond = arc[i].cond;
      if(X[cond] && cond!=sigELSE) return arc[i].dest; // Что-то сработало
    }
  // Не нашли. Ищем ELSE
  for(int i=0;i<NumArcs;i++)
    if(arc[i].src==Q && arc[i].cond==sigELSE) return arc[i].dest;
  // Не нашли вообще ничего. Никуда не будем переходить.
  return Q;
}
```

И, наконец, ниже приведен фрагмент функции, выполняющей процедуры, связанные с состояниями. Это тоже достаточно очевидная и простая функция.

```
int ExecuteProc(int q)
{ switch(q)
  {
    case qStart: break;
    case qE0: sysSETALARM(5); break;
    case qE1: stepLine(); break;
    case qE2: sysCALL(qEvacStart, qB0); return 0;
    case qE3: sysALARMRESET(); break;
    case qE4: sysCALL(qObstStart, qB0); return 0;
```

```
    ...
    case qB1: stepLine(); break;
    case qB2: sysCALL(qObstStart, qC0); return 0;
    case qB3: sysALARMRESET(); break;
    ...
  }
  return 1;
}
```

Обратите внимание на то, что эта функция возвращает значение 0, если происходит изменение текущего состояния (побочный эффект, о котором мы говорили выше).

**Приоритеты.** Задание структуры автомата списком ребер имеет еще один приятный момент. Поскольку функция перехода просматривает список ребер последовательно, то приоритет условий переходов будет осуществляться использованием элементарного упорядочивания этих ребер. На самом деле, список дуг является альтернативной формой записи списка продукций (правил вида «ЕСЛИ-ТО»), а в системе продукций существуют точно такие же проблемы с определением приоритетов.

# Обсуждение

Разумеется, реализация управляющей программы для робота в виде автомата – это лишь один из подходов. Со своими особенностями, слабыми и сильными сторонами.

## Сильные стороны АП

К сильным сторонам АП можно отнести следующее.

1. Простота и, как следствие, надежность. Речь идет о том, что стремление держаться рамок единого автоматного формализма заставляет нас более строго относиться к создаваемому программному коду. Этот код строится из небольших по объему «строительных» элементов. В этом и заключается простота. Короткие, унифицированные, построенные по единому принципу процедуры легче контролировать.

2. Наглядность алгоритма. Сложность алгоритма поведения никуда не делась. Просто она перенесена из программного кода вовне, в тот рисунок – граф, – которым пользуется программист, реализуя автомат. А с программной точки зрения сам автомат вместе с логикой его работы представлен простой структурой – матрицей переходов. Анализировать поведение робота, исходя из графа автомата действительно удобно и наглядно. Это в какой-то мере – особенности психологии образного человеческого восприятия.

3. Гибкость алгоритма. Автоматное представление позволяет постепенно наращивать сложность и гибкость поведения робота, не ломая логику работы всей программы. А это – крайне важно.

## Слабые стороны АП

Есть, разумеется, в автоматном программировании и свои сложности.

1. Неизбежное «трюкачество». Формальные рамки иногда становятся излишне тесными. В угоду сугубо практическим программистским или техническим соображениям, приходится отступать от требований модели. Так, например, мы поступали с входным символом $x$, заменяя его вектором сигналов $\boldsymbol{X}$.

2. Громоздкость автомата. Реализация того, что было естественно для «обычного» программирования, зачастую превращается в наращивание структуры автомата различного рода фиктивными состояниями, основная задача которых – свести вместе процедурную необходимость и автоматную возможность.

3. Неестественность некоторых механизмов. Такая неестественность возникает тогда, когда мы пытаемся найти автоматный эквивалент таким механизмам, как работа со

временем, вызовом подпрограмм и т.п. Введение таких механизмов влечет за собой и неизбежное «трюкачество», и увеличение сложности автомата.

### Принципиальная ограниченность автомата

Это, пожалуй, самая критикуемая сторона автоматного программирования. Рано или поздно раздаются упреки в том, что тот или иной механизм плохо укладывается в концепцию АП и ожидаемая (обещанная) красота и простота программы не наблюдается вовсе.

Но дело в том, что АП и не претендует на универсальность. Метод хорош тогда, когда решаемая задача укладывается в рамки его – метода – модели. Конечный автомат, например, плохо умеет считать. Возможности его памяти ограничены количеством и структурой связей его состояний. Это означает, что он может реализовывать управление лишь *регулярным* поведением. Плохо дело обстоит с рекурсивностью поведения. Дело в том, что автомат не может описать и сохранить в памяти такое явление, как *контекст ситуации*. Для всего этого существуют другие, более сложные модели. Поэтому, подчеркнем еще раз, решение о применении принципов АП принимается тогда, когда мы четко сопоставили возможности модели и решаемую задачу.

С практической точки зрения, ограниченность автомата для нас проявляется главным образом в реализации функции переходов *F(Q, X)*. Наше «трюкачество» с вектором **X** так и не смогло привести к реализации такого красивого механизма, как определение **произвольного условия перехода**. Если бы в качестве условия перехода можно было бы указать произвольное логическое выражение, то наш автомат – а вместе с ним и алгоритм, – выглядел бы гораздо привлекательнее.

### Автомат Мили и автомат Мура

Вновь вернемся к вопросу о выборе типа автомата. Можем ли мы сказать, что автомат Мура оказался более удобным для нашей задачи, чем автомат Мили? Уже упоминалось, что теоретически они равноправны. В какой-то степени, выполнение процедуры на переходе (автомат Мили) более универсально и удобно, чем в состоянии (автомат Мура). Это так. И автомат Мили более «лаконичен». В частности, если по автомату Мили построить эквивалентный ему автомат Мура, то у последнего будет больше состояний. И это естественно.

Далее. «Процедурная емкость» автомата Мили выше. Если у обоих автоматов будет по $|Q|$ состояний и $|\Sigma|$ входных символов, то количество процедур у автомата Мура будет ограничено величиной $|Q|$ (состояние-процедура), то у автомата Мили их может быть $|Q| \cdot |\Sigma|$.

Но, поскольку, чудес не бывает и за все надо платить, то платой за гибкость и емкость автомата Мили является усложнение процедур и формы представления. В частности, одной простой матрицей переходов ***M*** уже не обойтись. Надо либо строить дополнительную матрицу выходов (т.е. процедур) с тем же количеством строк (входы) и столбцов (состояния), либо усложнять саму матрицу ***M***. А используя автомат Мура, мы просто переложили эту работу на большее количество элементарных операций.

## Заключение

На этом мы закончим неформальное введение в принципы автоматного программирования (АП), суть которого заключается в том, что мы рассматриваем управляющую программу, как автомат. Если мы говорим, что логика работы управляющей программы не укладывается в линейную схему, то надо выбрать нелинейную структуру для ее представления, а это и есть автомат со своей сложной структурой.

Сильные и слабые стороны АП, его ограниченность и универсализм, особенности его применения мы уже обсудили. Здесь хотелось бы отметить лишь следующее.

На самом деле главную «опасность» для автоматного программирования представляет собой непоследовательность программиста. Речь идет о том, что зачастую программист, устав от необходимости подробного расписывания всяких там состояний «на все случаи жизни» и соответствующих реентерабельных процедур, начинает ставить «костыли», т.е. использует явно непригодные для автомата, «долгие» процедуры. Или вмешивается в логику работы управляющего автомата, используя «для удобства» прерывания, незапланированные механизмы обработки и анализа сигналов, «волюнтаристские» перехватывания переходов и т.п. Впрочем, такая непоследовательность (в самом широком смысле) – это опасность для любого программирования.

# Литература

Ниже приведены некоторые источники, относящиеся к двум категориям: книги, посвященные вопросам поведения технических систем, и книги по теории автоматов и автоматному программированию. Первая категория – идеологическая, вторая – практическая.


**Поведение технических систем:**

1. Варшавский В.И., Поспелов Д.А. Оркестр играет без дирижера: размышления об эволюции некоторых технических систем и управлении ими. - М.:Наука., 1984. - 208 с.
2. Гаазе-Раппопорт М.Г., Поспелов Д.А. От амебы до робота: модели поведения. - М.:Наука, 1987, - 288 с.
3. Цетлин М.Л. Исследования по теории автоматов и моделированию биологических систем. -М.:Наука, 1969. - 316 с.

**Автоматы и автоматное программирование:**

1. Кудрявцев В.Б., Алешин С.В., Подколзин А.С. Введение в теорию автоматов. - М.:Наука, 1985. -320 с.
2. Савельев А.Я. Прикладная теория цифровых автоматов. -М.:Высш.шк., 1987. -272 с.
3. Поликарпова Н.И., Шалыто А.А. Автоматное программирование. СПб.: Питер, 2008. 167 с.



МФТИ, МИЭМ НИУ ВШЭ, 2014