Откорректированная версия статьи:

Вавилов К.В. Программирование за… 1 (одну) минуту // Компьютер Price. — 2002. — № 31. — С. 288–293. (www.comprice.ru/debug)

Константин Вавилов [wawi@narod.ru]

Программирование за... 1 (одну) минуту

Программирование — в широком смысле — все технические операции, необходимые для создания программы, включая анализ требований и все стадии разработки и реализации. В более узком смысле – это ***кодирование*** и тестирование программы в рамках некоторого конкретного проекта.

Кодирование — написание уже спроектированной программы на некотором формальном языке программирования, причем любые решения, принимаемые на этом этапе, нельзя классифицировать как проектные, поскольку они довольно тривиальны.

Алгоритм — заранее заданная последовательность четко определенных правил или команд для получения решения задачи за конечное число шагов.

Из толкового словаря

Заранее выражаю признательность профессору Шалыто А.А. за теорию и Туккелю Н.И. за практические примеры в области автоматного программирования.

Проблема

Наверное давней мечтой всех постановщиков задач и разработчиков программного обеспечения является полное соответствие задуманного решения задачи (алгоритма решения) и программной реализации этого алгоритма. Но что-то не склеивается у постановщиков и программистов. В алгоритмах постоянно не учитывается то, что необходимо программистам для реализации, а текст программы мало похож на алгоритм. Таким образом, существуют два алгоритма: один на бумаге (для отчетности и документирования проектных решений), содержащий обычно некоторый результат проектирования, а не пути для его получения, а второй — в голове программиста (сохраняемый, правда, еще и в текстовом виде программы).

После написания окончательного текста программы зачастую предпринимаются попытки изменения документации, но опять учитывается не все. При этом логическая часть программы скорее всего отличается от логики алгоритма — нет полного соответствия. Я намеренно пишу **скорее всего**, потому что **никто** и **никогда** не собирался проверять **текст** программы. Если программа большая, то проверить по тексту соответствие алгоритму невозможно. Для проверки правильности реализации используется некая процедура под названием "Тестирование". При этом по существу проверяется как программист понял алгоритм (тот, который на бумаге) и как преобразовал его в алгоритм в своей голове, а далее в текст программы. В итоге программист является единственным держателем важнейшей **логической** информации и становится абсолютно не важным, что было придумано до программной реализации. И дело даже не в том, что программист может заболеть (или, не дай Бог, уйти на

другую работу), а в том, что **разные** программисты **по-своему** "строят" текст программы, в зависимости от интеллекта и знания языка программирования. В **любом** случае используется множество промежуточных переменных, которые программист вводит и применяет по своему усмотрению. И если программа большая и логически сложная, то для того, чтобы понять почему она "глючит" (имеется в виду не глюки операционной системы и неправильное использование функций языка, а логически неправильное выполнение), необходим более квалифицированный специалист, которому придется разбираться уже в **тексте** программы.

Большинство программистов, мягко говоря, не любят документировать алгоритмы перед программированием (и даже просто рисовать их на бумаге), скорее всего потому, что все равно придется выдумывать что-то свое по ходу кодирования. И правда, зачем терять время на рисование каких-то прямоугольников, ромбиков и стрелочек, если лучше сначала сразу **попрограммировать**, а потом изобразить примерно похожий или весьма общий алгоритм в документации. Все привыкли к этому — программисты, потому, что так легче, а постановщики задач — так как не всегда владеют знаниями по программированию в нужном объеме, а если и владеют, то уж никак не могут повлиять своевременно на то, что "вытворят" программисты. Удобные среды программирования (даже в DOS, не говоря о визуальных под Windows) тоже способствуют уверенности именно в такой последовательности разработки. Развитые средства отладки и наблюдения за значениями **переменных** позволяют надеяться, что можно выявить любую ошибку в логике.

Время уходит, проект надо закончить к заданному сроку, а исполнитель сидит и придумывает "на коленке" как ему разрешить ту или иную логическую задачу, да еще и успеть реализовать. А уж о логических ошибках, не выявленных при тестировании, и вспоминать не будем, ибо и тестирование аналогично (хаотично)… Таково сегодняшнее положение. Есть ли выход или хотя бы улучшение?

Логическая часть программы

Складывается впечатление, что теряется нечто главное при переходе от изображенного в общепринятом стандарте алгоритма к тексту программы.

Теоретиками автоматного программирования предложена следующая концепция идеальной логической части программы.

Вся логика программы строится на основе селектора (switch в языке Си, case — в Паскале). **Вспомогательно** используется старый добрый оператор условия if.

Упрощенно **любой** алгоритм управления (автомат) может быть реализован так, как показано в Листинге 1 (пока не задумывайтесь над смыслом комментариев, оцените структуру).

Листинг 1. Структура программы на языке Си, реализующей автомат

```
// Первая часть автомата – проверка условий переходов.
switch( Y ) // Переменная состояния автомата.
{
  case 0:
  // Проверка условий на дугах и петлях (условия проверяются по приоритетам),
  // выполнение перехода (изменение значения переменной Y)
  // и действий на дуге или петле (выполнение выходной функции);
  // протоколирование переходов и действий при выполнении условия.
  break ;
  ...
  case n:
  // Проверка условий на дугах и петлях (условия проверяются по приоритетам),
  // выполнение перехода (изменение значения переменной Y)
  // и действий на дуге или петле (выполнение выходной функции);
  // протоколирование переходов и действий при выполнении условия.
  break ;
};
```

```
// Вторая часть автомата – вызов вложенных автоматов и выполнение
// выходных функций в состояниях.
switch( Y )
{
  case 0:
  // Вызов вложенных автоматов.
  // Выполнение выходных функций в состоянии.
  break ;
  ...
  case n:
  // Вызов вложенных автоматов.
  // Выполнение выходных функций в состоянии.
  break ;
};
```

Конкретная реализация на языке Си для среды программирования LabVIEW 6i (программная имитация работы затвора) представлена в Листинге 2. Для непосвященных — затвор это устройство, по действию напоминающее обычный водопроводный вентиль, только намного больше и с электроприводом, а также с возможностью дистанционного управления.

В этой программе для реализации выходных функций используются признаки выполнения, которые в традиционном Си можно было бы заменить непосредственным вызовом выходных функций. В программируемых логических контроллерах эти признаки являлись бы значениями выходных переменных.

Листинг 2. Реализации автомата имитации работы затвора на языке Си для пакета LabVIEW 6i

```
// Входные переменные.
//
// x0 – режим дистанционного управления.
// x1 - сигнал "Включить пускатель на открытие".
// x2 - сигнал "Включить пускатель на закрытие".
// x3 - сигнал "Отключить пускатели".
// T1 - время включения/отключения пускателя истекло.
// T2 - время открытия/закрытия затвора истекло.
// H  – номер имитируемой ситуации.

// Выходные воздействия.
//
// z1   – запись признака "Затвор открыт".
// z2   – запись признака "Затвор закрыт".
// z3   – запись признака "Включен пускатель на открытие".
// z4   – запись признака "Включен пускатель на закрытие".
// zN1  – стирание признака "Затвор открыт".
// zN2  – стирание признака "Затвор закрыт".
// zN3  – стирание признака "Включен пускатель на открытие".
// zN4  – стирание признака "Включен пускатель на закрытие".
// zt1  – фиксация времени начала включения/отключения пускателя.
// zNt1 – сброс зафиксированного времени включения/отключения пускателя.
// zt2  – фиксация времени начала открытия/закрытия затвора.
// zNt2 – сброс зафиксированного времени открытия/закрытия затвора.

// Объявление и определение (начальный "сброс")
// признаков выполнения выходных функций.
int8 z1 = 0,  z2 = 0,  z3 = 0,   z4 = 0,
     zN1 = 0, zN2 = 0, zN3 = 0,  zN4 = 0,
     zt1 = 0, zt2 = 0, zNt1 = 0, zNt2 = 0 ;

// Автомат имитации работы затвора.
// Проверка условий переходов.
```

```
switch( AIPS )
{
  case 0:
    if((H==9))              { z3=1; zt2=1;                AIPS=103; }
    else
    if((H==10))             { z4=1; zt2=1;                AIPS=104; }
    else
    if(x1)                  { zt1=1;                      AIPS=101; }
    else
    if(x2)                  { zt1=1;                      AIPS=102; }
  break;

  case 1:
    if((H==9))              { zN2=1; z3=1; zt2 = 1;       AIPS=103; }
    else
    if(x1 && x0)            { zt1=1;                      AIPS=101; }
  break;

  case 2:
    if((H==10))             { zN1=1; z4=1; zt2=1;         AIPS=104; }
    else
    if(x2 && x0)            { zt1=1;                      AIPS=102; }
  break;

  case 101:
    if(T1)                  { zN2=1; z3=1; zNt1=1; zt2=1;  AIPS=103; }
    else
    if((!x1 || !x0) && x2)  { zNt1=1;                     AIPS=1; }
    else
    if((!x1 || !x0) && !x2) { zNt1=1;                     AIPS=0; }
  break;

  case 102:
    if(T1)                  { zN1=1; z4=1; zNt1=1; zt2=1;  AIPS=104; }
    else
    if((!x2 || !x0) && x1)  { zNt1=1;                     AIPS=2; }
    else
    if((!x2 || !x0) && !x1) { zNt1=1;                     AIPS=0; }
  break;

  case 103:
    if(T2 && (H!=7))        { z1=1; zt1=1; zNt2=1;        AIPS=105; }
    else
    if(T2 && (H==7))        { zN3=1; zNt2=1;              AIPS=0; }
    else
    if(x3)                  { zt1=1;                      AIPS=107; }
  break;

  case 104:
    if(T2 && (H!=8))        { z2=1; zt1=1; zNt2=1;        AIPS=106; }
    else
    if(T2 && (H==8))        { zN4=1; zNt2=1;              AIPS=0; }
    else
    if(x3)                  { zt1=1;                      AIPS=108; }
  break;

  case 105:
    if(T1)                  { zN3=1; zNt1=1;              AIPS=2; }
    else
    if((H==5))              { zNt1=1;                     AIPS=202; }
  break;

  case 106:
    if(T1)                  { zN4=1; zNt1=1;              AIPS=1; }
    else
    if((H==6))              { zNt1=1;                     AIPS=201; }
  break;

  case 107:
    if(!x3)                 { zNt1=1;                     AIPS=103; }
    else
    if(T1)                  { zN3=1; zNt1=1; zNt2=1;      AIPS=0; }
  break;
```

```
  case 108:
    if(!x3)                 { zNt1=1;                       AIPS=104; }
    else
    if(T1)                  { zN4=1; zNt1=1; zNt2=1;        AIPS=0; }
  break;

  case 201:
    if((H!=6))              { zN4=1;                        AIPS=1; }
  break;

  case 202:
    if((H!=5))              { zN3=1;                        AIPS=2; }
  break;

  default: break;
};

// Автомат имитации работы затвора.
// Вызов вложенных автоматов и выполнение
// выходных функций в состояниях.

switch( AIPS )
{
  default: break;
};
```

Как видите, приведенный пример достаточно сложен логически.

"Ну и что? Во-первых, ничего не понятно, какие-то автоматы, во-вторых, опять же знакомые операторы условий и нет никаких комментариев. В чем решение проблемы-то?!!!" — примерно такие вопросы хочется задать автору, не правда ли?

Внимательно прочитайте следующее предложение.

Весь текст программы (включая обеспечение "читабельности"), приведенный в листинге 2, получен **автоматически** (сгенерирован) из графа переходов (рис. 1), построенного в редакторе Word’97.

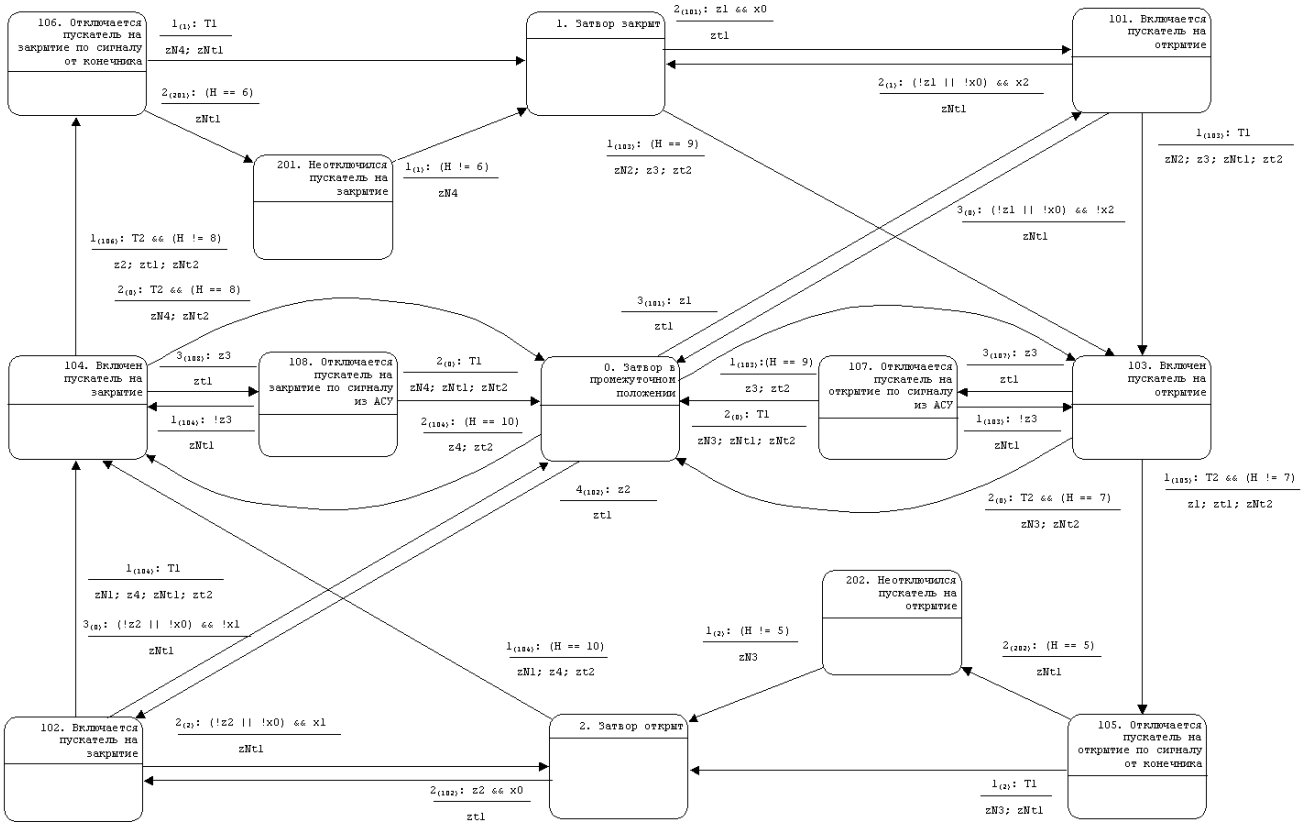

Рис. 1. Граф переходов автомата имитации работы затвора

Однозначное соответствие графа переходов и текста программы (их изоморфизм) — одна из основ автоматного программирования — позволяет обойтись без ручного кодирования, без собственных придумок человека, набирающего текст логической части программы, не говоря уже об ошибках при наборе текста!!! Что задумано в алгоритме, то и будет в программе!!!

Программисту остается сделать только несколько системозависимых "вещей": правильно вызвать функцию, реализующую автомат, правильно передать при этом вызове входную информацию (признаки и значения) и правильно запрограммировать выходные функции.

Алгоритм

Как следует из вышесказанного, главная роль при создании логической части программы отводится алгоритму. Слова **логическая часть** являются ключевыми, это надо сразу запомнить.

Основой всего в данном случае служит нечто — **состояние**. Необходимо добавить еще только одно слово — **ожидание**. Получаем, на мой взгляд, вполне достаточное определение "**состояние ожидания**". Ожидают в состоянии появления входных воздействий (признаков, значений или событий). Ожидание может быть либо кратким, либо продолжительным. Или, по-другому — состояния бывают неустойчивыми и устойчивыми.

Первым свойством состояния является то, что в нем ожидают **ограниченный** набор входных воздействий. Важность этого обстоятельства станет ясна позже.

Любой алгоритм (и программа, естественно) имеет входную и выходную информацию. Выходные воздействия можно разделить на два вида: переменные (например, операции над свойствами объектов) и функции (например, вызов функции открытия окна, запуска приложения, выдачи сообщения в окне диалога и т.п.).

Второе свойство состояния — выдача набора **точно определенных** значений выходных переменных. Это означает очень простое и одновременно необычайно важное обстоятельство — **в любой момент времени определены значения всех выходных переменных**, так как в каждый момент времени алгоритм (программа) находится в каком-то состоянии. Отметим, что это свойство явно используется в автоматах Мура, в которых выходные воздействия формируются в состояниях, и неявно — в автоматах Мили, в которых выходные воздействия формируются на переходах, как в рассмотренном примере.

Число состояний **ограничено и число наборов значений выходных переменных**, соответственно, тоже. Функция протоколирования переходов является органически встроенной в функцию автомата и, следовательно, **всегда** можно определить последовательность переходов между состояниями и выдачи выходных воздействий.

Автор статьи участвует в разработке АСУ ТП (автоматизированных систем управления технологическими процессами). Поэтому и приведенный пример посвящен имитации технологического объекта — затвора. Здесь все просто — "поведение" этого механического чуда техники известно и определить его основные устойчивые и неустойчивые состояния несложно. Далее добавляются всевозможные "нехорошие" условия, переходы и если необходимо "нехорошие" состояния.

Особенности программы

Теперь поговорим об особенностях программы. Сразу необходимо сказать о том, что идеальной средой для исполнения автомата является однопотоковая среда (например, DOS). Думаю, и даже уверен, что это не такая уж большая проблема, решение которой просто требует другого подхода к реализации. Для простоты понимания будем считать, что программа написана на классическом Си под DOS.

Первая особенность программы, использующей автоматы — обязательное наличие цикла while (или do–while). Тут вроде бы ничего нового не сказано, но самое главное, что этот цикл будет для логической части всей программы единственным! Автоматов (напомню — это функции, подпрограммы логики) может быть сколько угодно, но нигде больше в автоматах не будет встречаться ни цикл while, ни тем более оператор goto.

Вторая особенность "вытекает" из первой. Любой автомат содержит конструкцию switch (и фактически состоит из нее), внутри которой содержатся все логические операции. При вызове любого автомата управление передается к одной из меток case, а после выполнения соответствующих этой метке действий работа автомата (подпрограммы) завершается до следующего запуска. Указанные действия заключаются в проверке условий переходов и, если некоторое условие выполняется, то осуществляется вызов соответствующих выходных функций и изменение состояния автомата.

Основным следствием из всего сказанного является не только простота реализации **любого** автомата, а гораздо более важное — **в программе можно обойтись без многих промежуточных логических переменных**, роль которых в каждом автомате выполняет многозначная переменная состояния.

В последнее утверждение трудно поверить, ведь все мы привыкли без всяких особых размышлений применять множество глобальных и локальных переменных (флагов). А куда же без них?!!! Чаще всего это флаги, "сигнализирующие" программе о выполнении какого-либо условия. Флаг устанавливается (принимает значение TRUE) в нужный по мнению программиста момент, а потом (чаще всего только после того как этот флаг вызывает нежелательные последствия своим вечным TRUE) начинается мучительный поиск места в программе для его сброса в значение FALSE. Знакомо, не правда ли? Так вот теперь сообразите, глядя на текст примера: ведь здесь не применяется никаких дополнительных переменных, а просто изменяется значение номера состояния, и изменяется оно при выполнении логического условия. Чем не достойная замена флагов?!

А как же переменные — аргументы условий? В качестве аргументов в условиях выступают переменные, которые являются глобальными **входными** переменными для **всей** программы, их не надо выдумывать. Они есть в постановке задачи, а их значения изменяют не внутренние процессы, а внешние, никак не зависящие от логики программы (например, ввод символа с клавиатуры). Единственным исключением являются временные (от слова "время") переменные, установка и сброс которых осуществляется в определенных состояниях.

Отметим, что входные переменные могут быть реализованы и как функции, возвращающие значение 0 или 1. При такой реализации эти функции будут вызываться только при проверке значений входных переменных в условиях переходов. Кроме того, такая реализация позволяет выполнять протоколирование значений входных переменных при их опросе.

Принципы

Ну хорошо, это DOS, и это явно не Windows-решение, где же обработчики событий и ООП, все логические процедуры делаются там и без всяких автоматов?!!

Вернитесь к началу статьи, где в примере достаточно непростая логическая задача. В статье описан только принцип (успешно реализованный, кстати, в пакете графического программирования Windows-приложений LabVIEW), и этот принцип прост — **отделение логики от программиста-исполнителя** (специалиста, понимающего особенности операционной системы и языка и ответственного, в первую очередь, за программную корректность). Этот принцип поддерживается а**втоматическим получением текста логической части программы по графу переходов** (программирование за одну минуту) и, идеально, сосредоточение логики в одном месте программы. Теоретиками (Шалыто А.А. и

Туккель Н.И.) предложен один из путей применения автоматного подхода в Windows ("Мир ПК" № 8, 9 за 2001 год), как мне кажется, не единственный.

Никто не собирается утверждать, что программист не может строить граф переходов и вообще проектировать в одиночку. Так наверняка и будет в действительности, если человек сам без посторонней помощи правильно разберется в логике решения. В принципе такой путь проектирования следует признать более результативным. Главное тут в желании четко логически описать и понять алгоритм до собственно кодирования. Про тестирование будет сказано ниже.

А теперь **know-how**. Как же так получается, что и программировать практически ничего не надо, а достаточно только алгоритма в виде документа Word?

Все дело в…

- однородности графа переходов;
- особом, но простом написании текста автофигур Word;
- особой, но простой группировке автофигур и линий;
- однородности текста программы, реализующей автомат;
- однозначном соответствии графа переходов и текста программы.

Подумайте немного и все получится. У меня, не считающего себя даже средним программистом, ушло два дня на создание приложения для автополучения текста программы из документа Word. Правда, программировал я в LabVIEW, но можно и нужно, конечно, использовать VBA, VC, Delphi и т.п. Текст программы в принципе генерируется на любом языке: Си, Паскаль, Ассемблер и т.д. То есть везде, где есть аналог оператора switch (возможно также обойтись и применением оператора if).

А вообще-то, я все больше склоняюсь к мысли использовать не Word, а Visio. Кстати, я узнал, что существует программа, преобразующая граф переходов, изображенный в Visio, в текст на языке Си, построенный в соответствии с принципами автоматного программирования (**http://www.geocities.com/goloveshin/v2s_rus.htm**). При использовании Visio, кроме прочих удобств, отпадет необходимость указывать в пометке каждой дуги номер вершины, в которую она ведет.

Тестирование

Что же дальше? Программа написана, надо тестировать, убедиться в том, что то, что задумывалось, успешно разрешено и реализовано. Вспомним одно из свойств состояния автомата, а именно первое — в состоянии ожидают **ограниченный** набор входных воздействий и проверяется **ограниченное** количество условий. Также вспомним об **ограниченном** количестве состояний. Таким образом, получается пусть не маленькое (в случае большого алгоритма), но явно конечное и легко выделяемое число вариантов для проверки. Главное состоит в том, что и тестирование логики будет однородным и, следовательно, его можно автоматизировать! В реальной работе я лично всегда опирался на протоколы, отображающие переходы и выполнение выходных функций. В них отражены все логические "пути" программы, причем можно организовать просмотр протокола в реальном времени. **Ни с чем не сравнимое чувство** возникает, когда ты точно и сразу знаешь место и условия возникновения логической ошибки. А по мере набора опыта, начинаешь уже при проектировании определять "узкие" места и учитывать всевозможные проблемы. Как говорится, главное начать правильно мыслить. Исправлять логику следует за счет изменения графа переходов и повторной генерации программы после этого.

Ну вот, пожалуй, вкратце и все. Советую напоследок обязательно почитать классиков (труды Шалыто А.А. и Туккеля Н.И. представлены также и в Интернете на **http://is.ifmo.ru** и **http://www.softcraft.ru**). Там все будет научно и не так коряво изложено. Успехов!!!

Об авторе

Вавилов Константин Валерьевич — заведующий группой по разработке АСУ ТП,
ЗАО "Тяжпромэлектропроект Санкт-Петербург", wawi@narod.ru