С авторами можно связаться по адресу: mail@avrorasystems.com, "для Шалыто".

Статья опубликована журнале "Мир ПК", 2001. №8, С.116-121; №9, С.132-138.

ПРОГРАММИРОВАНИЕ С ЯВНЫМ ВЫДЕЛЕНИЕМ СОСТОЯНИЙ

Анатолий Шалыто, Никита Туккель

Вне зависимости от методов разработки у любой программы есть состояния, в каждый момент времени определяемые значениями всех ее данных. Гради Буч пишет о состояниях программ: "Внутри большой прикладной программы могут существовать сотни и даже тысячи переменных и несколько потоков управления. Полный набор этих переменных описывает состояние прикладной программы в каждый момент времени" [1].

Можно рассматривать состояние программы и более упрощенно, в виде совокупности значений всех управляющих переменных — переменных, участвующих в условиях переходов. Тогда изменение значения одной из управляющих переменных будет означать изменение состояния программы, а число состояний программы будет определяться максимально возможным количеством комбинаций значений управляющих переменных, возникающим при ее работе. Предположим, что в программе используются только двоичные управляющие переменные (флаги). В этом случае количество состояний программы, содержащей n двоичных управляющих переменных, будет лежать в интервале от n до 2^n.

Возможно, что программист предусмотрел реакции на все комбинации значений управляющих переменных (2^n комбинаций в рассматриваемом случае). Однако более вероятно, что некоторые комбинации значений управляющих переменных (вплоть до 2^n-n) оказались непредусмотренными. Тогда при

возникновении неожиданного сочетания входных воздействий программа может перейти в непредусмотренное состояние. Такие состояния Фредерик Брукс называет "невизуализируемыми": "Сложность служит причиной трудности перечисления, а тем более понимания всех возможных состояний программы, а отсюда возникает ее ненадежность... Сложность структуры является источником невизуализируемых состояний, в которых нарушается система защиты" [2]. Поведение программы в непредусмотренном состоянии может быть различным: от нарушения защиты памяти до продолжения функционирования с созданием различного рода побочных эффектов.

Многие пользователи компьютеров, и, наверняка, все разработчики программного обеспечения не раз сталкивались с ситуацией, в которой используемая или разрабатываемая программа попадает в непредусмотренное состояние.

Чтобы привести пример непредусмотренных состояний, обратимся к области компьютерных игр. Следует отметить, что хотя компьютерные игры принято относить к "несерьезной" разновидности программного обеспечения, их никак нельзя назвать несерьезными с точки зрения программиста. В наши дни многие компьютерные игры максимально используют самые современные программные и аппаратные наработки всей компьютерной индустрии, а по сложности зачастую превосходят крупные промышленные системы управления. К тому же, большинство компьютерных игр относится к категории программ реального времени и характеризуются весьма сложным поведением. Исходя из этого, развитие индустрии игрового программного обеспечения можно рассматривать как показатель развития программирования вообще. Изучение современных компьютерных игр показывает, что при высоком качестве графического и звукового оформления, досадные "проколы"

чаще всего встречаются именно там, где на помощь мог бы прийти пересмотр практики программирования в сторону явного выделения на этапе проектирования всех требуемых состояний программы.

Как пример можно привести компьютерную игру "Starship Troopers" (разработчик — компания "BlueTooth"). Учитывая прекрасный звук и графику, популярный сюжет и великолепный игровой процесс, можно смело назвать эту игру одной из лучших за 2000 год. Эта игра содержит "классический" пример непредусмотренного состояния в программе. Если один из десантников, находящихся под управлением игрока, будет атакован "птичкой" в тот момент, когда он летит при помощи реактивного ранца (что является редко возникающим и, видимо, поэтому непредусмотренным программистом сочетанием входных воздействий), то в результате этого "воздушного боя" игрок сможет наблюдать довольно странную картину: состояние десантника в окне состояний всех бойцов будет отображаться как "погиб", в то время как он, в миг потерявший весь свой запас брони и жизненных сил, будет выглядеть и действовать не хуже своих "более живых" товарищей. Происходит это, по всей видимости, потому что состояние десантника в модулях программы, отвечающих за анимацию модели, обработку команд и отображение состояния, хранится в разных управляющих переменных. Подобной ошибки в программе можно было бы избежать, если хранить состояние каждого бойца не в виде набора управляющих переменных, а как одну многозначную переменную с определенными заранее значениями.

Другая разновидность примеров непредусмотренных состояний в программах наверняка известна любителям игр с элементами квестового (приключенческого) жанра. Многие сталкивались с ситуацией, когда непредусмотренные

разработчиками игры действия игрока приводили к невозможности дальнейшего развития сюжета игры. И в этом случае корректность программы может быть обеспечена явным введением состояний с многозначным их кодированием.

SWITCH-ТЕХНОЛОГИЯ: ОБЩИЕ СВЕДЕНИЯ

Идея использования многозначных переменных состояния для описания как развития сюжета игры, так и взаимодействия игрока с персонажами игры, управляемыми компьютером, напрашивается сама собой. Видимо поэтому, в [3] для описания поведения в играх было предложено применять автоматы, а компания "BioWare" в своих играх (например, "Baldur's Gate 2") стала использовать многозначные переменные состояния для описания последовательностной логики развития сюжета игры. Естественно, что такой подход был известен и ранее. Еще в 1966 году Эдсгер Дейкстра предложил использовать в программировании многозначные переменные состояния [4].

Технология, основанная на многозначных переменных состояния, получившая название "SWITCH-технология", была предложена в 1991 г. А.А.Шалыто для логического управления [5]. Именно в ней был впервые введен этап кодирования состояний, отсутствующий в традиционных технологиях программирования. Позднее авторы развивали SWITCH-технологию применительно к событийным системам [6, 7].

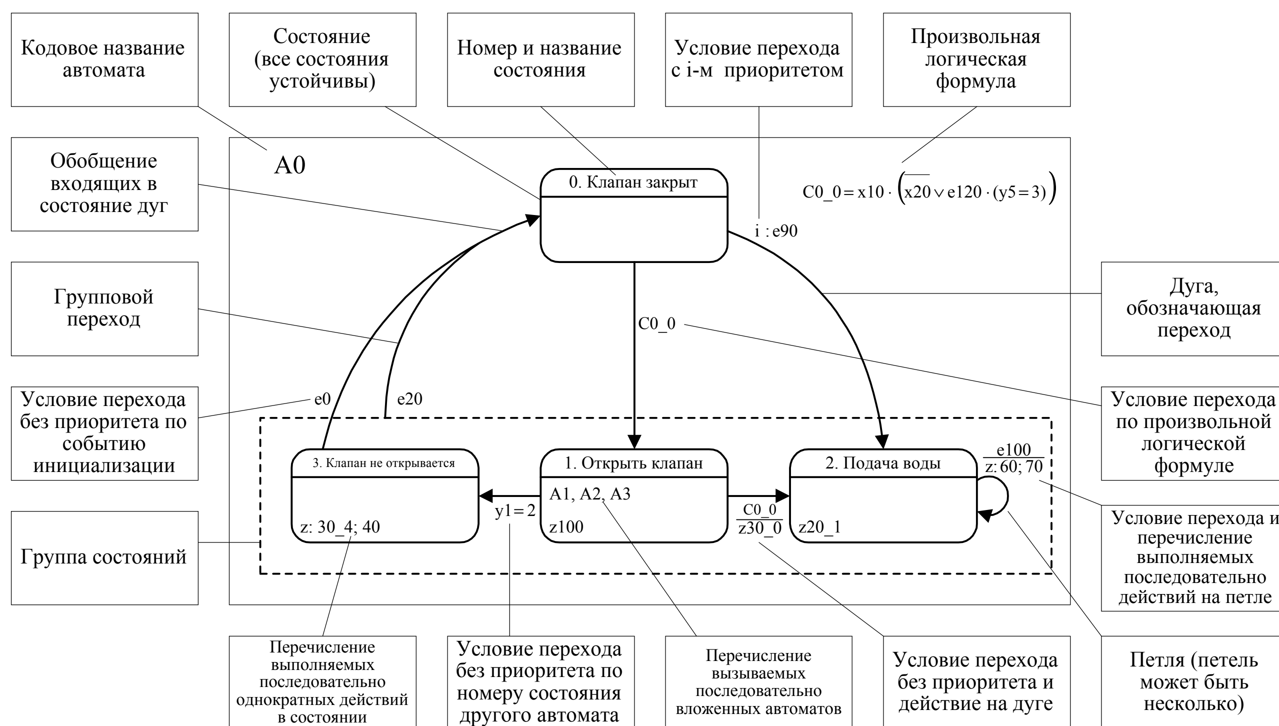
Для устранения самой возможности возникновения в программе непредусмотренных состояний следует еще на этапе проектирования явно определять все требуемые состояния и применять для их различения только одну многозначную управляющую переменную. После этого необходимо явно определить все возможные переходы между

состояниями и построить программу так, чтобы она не могла сойти с проложенных "рельс".

Для достижения строгости при разработке поведения программы необходимы три составляющие: математическая модель, позволяющая однозначно определить состояния программы и возможные переходы между ними, графическая нотация для этой модели и универсальный способ программной реализации алгоритмов, заданных в этой нотации.

В качестве такой математической модели предлагается использовать конечный автомат, базирующийся на понятии "состояние". Следует отметить, что конечные автоматы в программировании традиционно используются при разработке трансляторов и протоколов, в то время как при создании других типов программ они обычно не применяются. В последнее время ситуация в этом вопросе, как показано в обзоре [7], начинает меняться.

В качестве графической нотации выберем графы переходов смешанных автоматов [5], так как они сочетают в себе компактность и наглядность с гибкостью в плане введения дополнительных обозначений (рис.1) и внесения изменений.



Используемые обозначения

An — автомат с номером n.
yn — переменная состояния автомата с номером n.
xj — входная переменная с номером j.
zk — выходное воздействие с номером k.
zd_m — m-ое значение выходного воздействия с номером d.
en — событие с номером n. В условиях переходов "en" является сокращенной записью предиката "e == n", а "$\overline{\text{en}}$" - сокращенной записью предиката "e != n".
yn = p — условие перехода по номеру состояния автомата с номером n (предикат "yn == p").
Cn_r — условие перехода с номером r для автомата с номером n.
i: — обозначение приоритета условия перехода (1 - наивысший приоритет).

Рис. 1

Для программирования автоматов предлагается алгоритм (рис.2), позволяющий реализовать произвольную иерархию автоматов (графов переходов) с любым уровнем вложенности.

Рис. 2

При использовании языка Си каждый автомат реализуется в виде функции, шаблон которой приведен в приложении 1. Этот шаблон для обеспечения изобразительной эквивалентности с графом переходов базируется на конструкции `switch`, что и определило название излагаемой технологии.

Для иллюстрации сказанного рассмотрим пример. Создадим программный модуль управления элементом пользовательского интерфейса "тулбар" (рис.3), реализующий следующие функции:

- перемещение тулбара при нажатой правой кнопке мыши;
- вывод меню тулбара нажатием и отпусканием правой кнопки мыши.

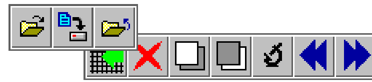


Рис. 3

ТРАДИЦИОННЫЙ ПОДХОД

Сначала построим этот модуль, используя традиционный событийный подход. При этом для выполнения заданных функций необходимо реализовать обработку следующих событий:

– нажатие правой кнопки мыши;

– отпускание правой кнопки мыши;

– перемещение мыши с нажатой правой кнопкой;

– выход курсора мыши за границу тулбара.

Модуль, состоящий из обработчиков перечисленных событий (приложение 2), создан для работы под ОС QNX и графической оболочкой Photon.

На первый взгляд, кажется (как это обычно утверждается в литературе), что эти обработчики независимы. Однако это не так, ввиду того, что они предназначены для управления одним и тем же объектом (тулбаром), и имеет место их взаимосвязь за счет наличия общей управляющей переменной (menu). Из текста программы следует, что логика ее работы рассредоточена по обработчикам событий, что делает поведение модуля априори непредсказуемым.

Можно отметить также и следующий недостаток традиционного подхода. Для обеспечения понятности программы ее стараются делать самодокументирующейся. При этом вводятся комментарии на естественном языке разработчика (в России обычно на русском) или заказчика. Также используются смысловые идентификаторы, которые по

этой причине достаточно длинны и должны быть записаны на английском языке. Поэтому, во-первых, разработчик вынужден использовать английский язык в именах переменных и функций, что весьма неудобно, так как требует постоянного переключения мыслей с одного языка на другой. Во-вторых, при необходимости графического отображения результатов проектирования, что в последнее время все чаще требуется заказчиками, оно становится чрезвычайно громоздким, и поэтому либо все-таки не используется, либо не является досконально точным.

Этот же недостаток имеет место и в тех случаях, когда самодокументирующимся делают визуальный формализм. Примером этого являются диаграммы состояний (Statecharts), используемые в UML [8], в которых запись сложных логических выражений и большого числа выходных воздействий и их особенностей становится практически необозримой. Такая же ситуация имеет место и при использовании SDL-диаграмм, широко применяемых при программной реализации протоколов в телефонии [7].

Отметим также, что при традиционном написании текстов программ реализация функций входных и выходных воздействий обычно выполняется совместно с логикой, затрудняя ее понимание.

SWITCH-ТЕХНОЛОГИЯ

Перейдем теперь к проектированию рассматриваемого модуля с применением описываемой технологии автоматного программирования, особенности и достоинства которой изложены в [6, 7].

При использовании этого подхода для каждого модуля, вместо самодокументирующейся программы, разрабатываются четыре документа, которые в совокупности, как будет

показано ниже, решают вопрос о понятности поведения программы: словесное описание поведения модуля (например, перечень выполняемых модулем функций); схема связей автомата с его окружением (интерфейс автомата); граф переходов, однозначно и математически строго определяющий поведение автомата; текст программного модуля.

Используя словесное описание поведения модуля, формализуем перечень его входных и выходных воздействий в виде схемы связей автомата (рис.4), указывая на ней для каждого из воздействий его источник (приемник), полное название (на языке разработчика) и идентификатор в виде буквы латинского алфавита с номером. Для автомата идентификатор предлагается начинать с буквы 'A', для события — с 'e', для входной переменной — с 'x', для переменной состояния автомата — с 'y', а для выходного воздействия — с 'z'.

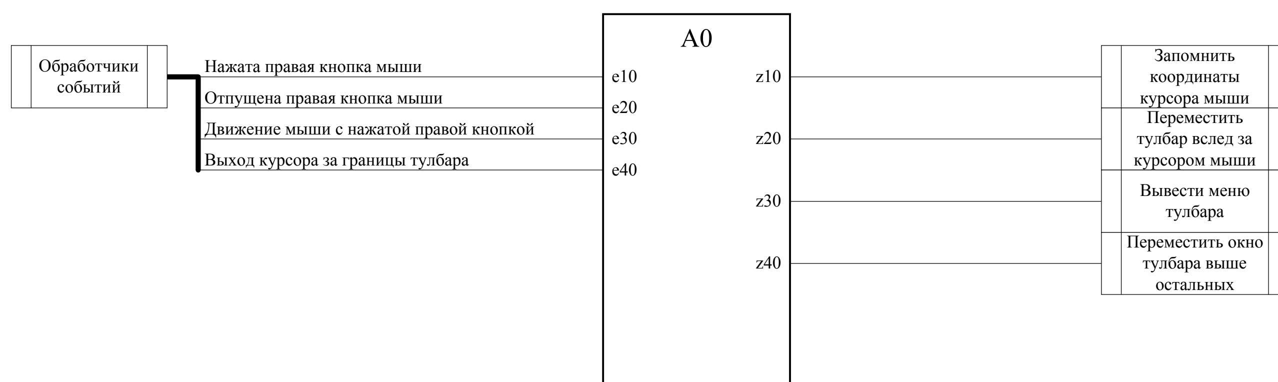


Рис. 4

В начале построения графа переходов определяются его возможные состояния, соответствующие состояниям тулбара. Эти состояния являются "естественными", так как они связаны с физикой процесса управления: ожидание (начальное состояние) и перемещение.

Далее определяются последовательности событий, вызывающие переходы между этими состояниями. После этого вводятся дополнительные состояния, "разделяющие" события

в каждой из последовательностей. Дополнительное состояние, возникающее на пути от состояния "ожидание" к состоянию "перемещение", назовем "готовность". Пронумеровав в графе переходов состояния (начиная с нуля), определяются остальные переходы между состояниями и необходимые петли. Используя схему связей автомата, на графе переходов отображаются условия переходов и действия, выполняемые в вершинах, на дугах и петлях. Построенный таким образом граф переходов смешанного автомата приведен на рис.5.

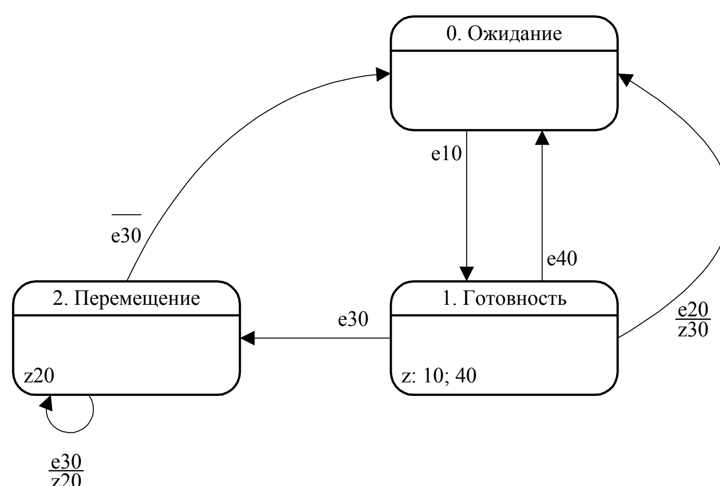


Рис. 5

Наличие схемы связей позволяет понять смысл используемых в графе переходов идентификаторов, несмотря на их очень короткие и весьма абстрактные обозначения. Совместное изучение схемы связей и графа переходов позволяет даже не участвовавшему в разработке специалисту понять поведение создаваемого программного модуля.

Отметим, что при использовании SWITCH-технологии вместо термина "**логика программы**" (предполагающего работу с флагами) предлагается применять термин "**поведение программы**", подразумевающий работу с состояниями.

Далее граф переходов (рис.5) формально и изоморфно реализуется по шаблону в виде функции, в данном случае на

языке Си. Применяемый шаблон позволяет обеспечить внешнюю похожесть текста программного модуля на граф переходов. Построенная функция не требует пояснений (комментариев) к своему поведению, так как при использовании предлагаемой технологии текст программы не является основным (и тем более единственным) программным документом, а поведение однозначно и математически строго задается графом переходов. Эта функция не содержит реализации входных и выходных воздействий, а включает только их вызовы.

Получаемая при этом часть программы называется системонезависимой. Перейдем к построению ее системозависимой части.

В программу добавляются обработчики событий, каждый из которых также реализован в виде функции и содержит вызов построенного автомата с передачей номера соответствующего события.

После этого, добавив в виде "заглушек" функции входных 'x' (если они имеются) и выходных 'z' воздействий, содержащие только вызовы функций протоколирования, можно уже на ранней стадии программной реализации получить действующий макет разрабатываемого модуля, что соответствует принципу пошаговой нисходящей разработки [2].

Программная реализация завершается разработкой располагаемых отдельно функций входных и выходных воздействий и используемых ими вспомогательных модулей (приложение 3). Эти функции обычно могут быть отлажены независимо.

Структура полученной программы (рис.6) отличается от традиционной, так как в ее центре находится система взаимосвязанных автоматов (в рассматриваемом примере — один автомат).

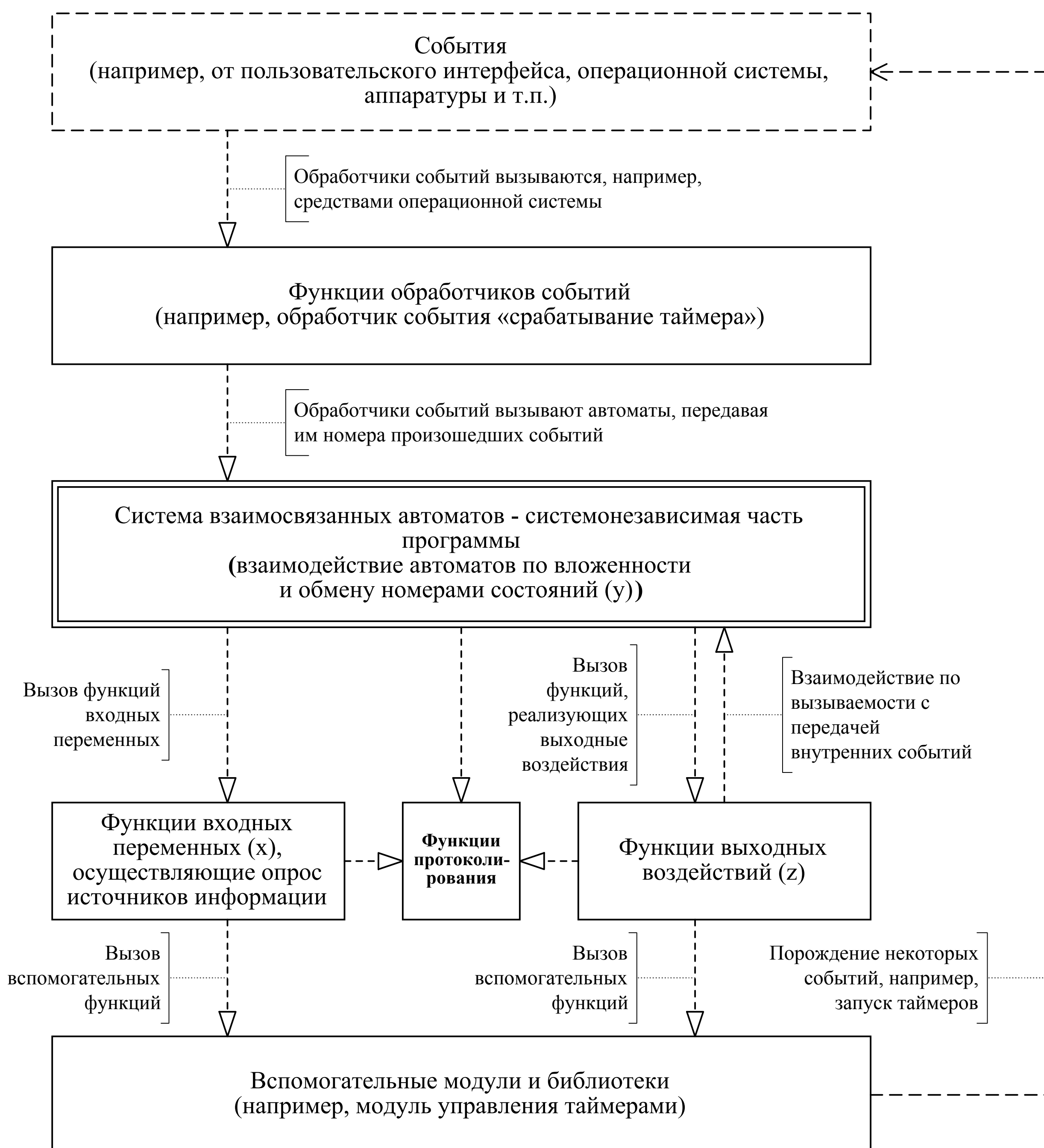


Рис. 6

В силу того, что построенная на основе предложенного подхода программа имеет регулярную структуру, то вводя вызовы функций протоколирования в функции автоматов, входных и выходных воздействий имеется возможность автоматического получения **истории выполнения программы в терминах автоматов** в форме протокола (приложение 4).

ЗАКЛЮЧЕНИЕ

В предыдущие десятилетия большинство программистов имели либо инженерное, либо математическое образование с соответствующей, устоявшейся веками, культурой. Современные программисты воспитываются иначе [9], а дисциплине программирования должного внимания не уделяется.

Предлагаемая технология является новой попыткой введения такой дисциплины и основана на априорном задании требуемых состояний и их визуализации.

Опыт применения предлагаемой технологии подтвердил следующее высказывание: "то, что не специфицировано формально, не может быть проверено, а то, что не может быть проверено, не может быть безошибочным" [10]. Поэтому авторы надеются (особенно учитывая мнение о работе [5], высказанное в [11]), что предложенный подход по крайней мере для систем логического управления и "реактивных" систем является приближением к "серебряной пуле" [2] в части создания качественных программ, тем более, что Ф.Брукс отозвался благосклонно только о подходе Дэвида Харела [8], также основанном на применении автоматов, преимущество по сравнению с которым показано в [7].

Целесообразность применения автоматного подхода подтверждается и тем, что создатель операционной системы UNIX Кен Томпсон на вопрос о текущей работе ответил: "Мы создали язык генерации машин с конечным числом состояний, так как реальный селекторный телефонный разговор — это группа взаимодействующих машин с конечным числом состояний. Этот язык применяется в "Bell Labs" по прямому назначению — для создания указанных машин, а вдобавок с его помощью стали разрабатывать драйверы" [12].

Применение в предлагаемой парадигме понятия "автомат" в качестве центрального, соответствует его месту в теории управления, что принципиально отличает данный подход от других парадигм программирования.

Авторы считают, что предложенный подход для рассматриваемых классов систем, весьма распространенных на практике, в соответствии с принципом Оккама "не размножает сущности без необходимости" (как, например, UML) и обладает "минимализмом" [13], необходимым для обеспечения понимания программ.

Рассматриваемая в настоящей статье парадигма автоматного программирования начинает все шире использоваться [14—16]. Для большей ее популяризации на сайте [17] создан раздел "Автоматные модели".

ЛИТЕРАТУРА

1. *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами приложений на С++. М.: Бином, СПб: Невский диалект, 1998. 560 с.
2. *Брукс Ф.* Мифический человеко-месяц или как создаются программные системы. СПб.: Символ, 2000. 304 с.
3. *Секреты* программирования игр /А. Ла Мот, Д. Ратклифф, М. Семинаторе и др. СПб.: Питер, 1995. 278 с.
4. *Дейкстра Э.* Взаимодействие последовательных процессов // Языки программирования. М.: Мир, 1972. С.9-86.
5. *Шалыто А.А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. 628 с.
6. *Шалыто А.А., Туккель Н.И.* SWITCH-технология — автоматный подход к созданию программного обеспечения "реактивных" систем // Промышленные АСУ и контроллеры. 2000. №10. С.44-48.
7. *Шалыто А.А.* Алгоритмизация и программирование для систем логического управления и "реактивных" систем // Автоматика и телемеханика. 2001. №1. С.3-35.
8. *Буч Г., Рамбо Д., Джекобсон А.* Язык UML. Руководство пользователя. М.: ДМК, 2000. 432 с.
9. *Черняк Л.* Создание программ как инженерная дисциплина //COMPUTERWORLD РОССИЯ. 2000. №37. С.18-20.
10. *Зайцев С.С.* Описание и реализация протоколов сетей ЭВМ. М.: Наука, 1989. 112 с.
11. *Герр Р.* Новый поворот // PC Magazine / Russian Edition. 1998. №10. С.88-90.
12. *Кук Д., Урбан Д., Хамилтон С.* Unix и не только. Интервью с Кеном Томпсоном //Открытые системы. 1999. №4. С.35-47.
13. *Герр Р.* Отладка человечества // PC Magazine / Russian Edition. 2000. №5. С.90-91.
14. *Богатырев Р.* Об асинхронном и автоматном программировании //Открытые системы. 2001. N3.
15. *Любченко В.С.* Мы выбираем, нас выбирают... (к проблеме выбора алгоритмической модели) //Мир ПК. 1999. N3.
16. *Кузнецов Б.П.* Психология автоматного программирования //BYTE/Россия. 2000. N11.
17. http://www.softcraft.ru

ОБ АВТОРАХ

Шалыто Анатолий Абрамович — ученый секретарь Федерального научно-производственного центра (ФНПЦ) — федерального государственного унитарного предприятия (ФГУП) "НПО "Аврора"", профессор кафедры "Компьютерные технологии" Санкт-Петербургского государственного института точной механики и оптики (технического университета) (СПбГИТМО (ТУ)). С ним можно связаться по адресу: mail@avrorasystems.com ("для Шалыто").
Туккель Никита Иосифович — инженер-программист ФНПЦ — ФГУП "НПО "Аврора"". С ним можно связаться по адресу: cynical@mail.ru.

ПРИЛОЖЕНИЕ 1

```
// Шаблон функции, реализующей автомат. Заменить "_i_" на номер автомата.
void A_i_( int e )
{
  int y_old = y_i_ ;

  // Протоколирование запуска автомата.
  #ifdef A_i__BEGIN_LOGGING
    log_begin( "A_i_", y_old, e ) ;
  #endif

  switch( y_i_ )
  {
    case 0:
      // Вызвать вложенные автоматы.
      // Проверить условия на дугах и петлях,
      // выполнить переход и действия на дуге или петле.
    break ;

    ...

    case n:
      // Вызвать вложенные автоматы.
      // Проверить условия на дугах и петлях,
      // выполнить переход и действия на дуге или петле.
    break ;

    default:
      #ifdef A_i__ERRORS_LOGGING
        log_write( LOG_GRAPH_ERROR,
                   "Ошибка в автомате A_i_: неизвестный номер состояния!", 0 ) ;
      #endif
  } ;

  // Если состояние не изменилось - завершить выполнение функции.
  if( y_old == y_i_ ) goto A_i__end ;

  // Протоколирование перехода в автомате.
  #ifdef A_i__TRANS_LOGGING
    log_trans( "A_i_", y_old ) ;
  #endif

  switch( y_i_ )
  {
    case 0:
      // Произвести активизацию вложенных в новое состояние автоматов.
      // Выполнить действия в новом состоянии.
    break ;

    ...

    case n:
      // Произвести активизацию вложенных в новое состояние автоматов.
      // Выполнить действия в новом состоянии.
    break ;
  } ;

  // Протоколирование завершения работы автомата.
  A_i__end: ;
  #ifdef A_i__END_LOGGING
    log_end( "A_i_", y_i_, e ) ;
  #endif
} ;
```

ПРИЛОЖЕНИЕ 2

//==
// Модуль, реализующий управление тулбарами.
// Версия, выполненная с использованием традиционного подхода.

```
//======
// Обработчик события нажатия кнопки мыши на тулбаре.
//
int toolbar_btn_press(PtWidget_t *widget, ApInfo_t *apinfo, PtCallbackInfo_t *cbinfo)
{
  toolbar_t            *tb = tb_data(widget) ;    // Указатель на данные тулбара.
  PhRect_t             *rect ;                    // Координаты курсора мыши.
  PhEvent_t            *event = cbinfo->event ;   // Произошедшее событие.
  PhPointerEvent_t  *edata = PhGetData( event ) ;     // Дополнительная информация о
                                                     // событии.

  if( edata->buttons&Ph_BUTTON_MENU )
  {
    // Была нажата правая кнопка мыши.

    rect = PhGetRects( event ) ;
    tb->drag_pos = rect->ul ;     // Запомнить координаты курсора мыши.
    PtWindowToFront( tb->wgt ) ; // Переместить окно тулбара выше всех
                                 // остальных.
    tb->menu = 1 ;               // Запомнить, что была нажата правая кнопка мыши.
  } ;

  return( Pt_CONTINUE ) ;
} ;

//======
// Обработчик события отпускания кнопки мыши на тулбаре.
//
int toolbar_btn_release(PtWidget_t *widget,ApInfo_t *apinfo,PtCallbackInfo_t *cbinfo)
{
  toolbar_t            *tb = tb_data(widget) ;    // Указатель на данные тулбара.
  PhEvent_t            *event = cbinfo->event ;   // Произошедшее событие.
  PhPointerEvent_t  *edata = PhGetData( event ) ;     // Дополнительная информация о
                                                     // событии.

  if(    event->subtype==Ph_EV_RELEASE_REAL
      && edata->buttons&Ph_BUTTON_MENU )
  // Была отпущена правая кнопка мыши.
    if( tb->menu == 1 )
    // Перед этим была нажата правая кнопка мыши.
      ApCreateModule( ABM_toolbar_menu, NULL, NULL ) ;      // Отобразить меню.

  tb->menu = 0 ;

  return( Pt_CONTINUE ) ;
} ;

//======
// Обработчик события перемещения мыши с нажатой кнопкой.
//
int toolbar_btn_move(PtWidget_t *widget, ApInfo_t *apinfo, PtCallbackInfo_t *cbinfo)
{
  toolbar_t            *tb = tb_data(widget) ;    // Указатель на данные тулбара.
  PhRect_t             *rect ;                    // Координаты события.
  PhEvent_t            *event = cbinfo->event ;   // Произошедшее событие.
  PhPointerEvent_t  *edata = PhGetData( event ) ;     // Дополнительная информация о
                                                     // событии.

  // Если не нажата правая кнопка мыши — ничего не делать.
  if( !edata->buttons&Ph_BUTTON_MENU ) return Pt_CONTINUE ;

  // Переместить тулбар вслед за курсором мыши.
  rect = PhGetRects( event ) ;
  tb->menu = 0 ;
  toolbar_move( tb, rect->ul.x - tb->drag_pos.x,
                    rect->ul.y - tb->drag_pos.y ) ;
```

```
  return( Pt_CONTINUE ) ;
} ;

//======
// Обработчик события пересечения курсором мыши границы тулбара.
//
int toolbar_boundary(PtWidget_t *widget, ApInfo_t *apinfo, PtCallbackInfo_t *cbinfo)
{
  toolbar_t                *tb = tb_data(widget) ;    // Указатель на данные тулбара.

  tb->menu = 0 ;

  return( Pt_CONTINUE ) ;
} ;
```

ПРИЛОЖЕНИЕ 3

```
//===========================================================================
// Модуль, реализующий управление тулбарами.
// Версия, выполненная с использованием автоматного подхода.

//======
// Обработчик события нажатия кнопки мыши на тулбаре.
//
int toolbar_btn_press( PtWidget_t *widget, ApInfo_t *apinfo, PtCallbackInfo_t *cbinfo )
{
  toolbar_t         *tb = tb_data(widget) ;           // Указатель на данные тулбара.
  PhEvent_t         *event = cbinfo->event ;          // Произошедшее событие.
  PhPointerEvent_t  *edata = PhGetData( event ) ;     // Дополнительная информация о
                                                      // событии.

  if( edata->buttons == Ph_BUTTON_MENU )
  {
    // Нажатая кнопка мыши — правая.
    // Вызвать управляющий автомат.
    A0( 10, tb, event ) ;
  } ;

  return( Pt_CONTINUE ) ;
} ;

//======
// Обработчик события отпускания кнопки мыши на тулбаре.
//
int toolbar_btn_release( PtWidget_t *widget, ApInfo_t *apinfo, PtCallbackInfo_t *cbinfo )
{
  toolbar_t         *tb = tb_data(widget) ;           // Указатель на данные тулбара.
  PhEvent_t         *event = cbinfo->event ;          // Произошедшее событие.
  PhPointerEvent_t  *edata = PhGetData( event ) ;     // Дополнительная информация о
                                                      // событии.

  if( event->subtype == Ph_EV_RELEASE_REAL && edata->buttons == Ph_BUTTON_MENU )
  {
    // Отпущенная кнопка мыши — правая.
    // Вызвать управляющий автомат.
    A0( 20, tb, event ) ;
  } ;

  return( Pt_CONTINUE ) ;
} ;

//======
// Обработчик события перемещения мыши с нажатой кнопкой.
//
int toolbar_btn_move( PtWidget_t *widget, ApInfo_t *apinfo, PtCallbackInfo_t *cbinfo )
{
  toolbar_t         *tb = tb_data(widget) ;           // Указатель на данные тулбара.
  PhEvent_t         *event = cbinfo->event ;          // Произошедшее событие.
  PhPointerEvent_t  *edata = PhGetData( event ) ;     // Дополнительная информация о
                                                      // событии.

  if( edata->buttons == Ph_BUTTON_MENU )
  {
    // Правая кнопка нажата.
    // Вызвать управляющий автомат.
    A0( 30, tb, event ) ;
  } ;

  return( Pt_CONTINUE ) ;
} ;

//======
// Обработчик события пересечения курсором мыши границы тулбара.
//
int toolbar_boundary( PtWidget_t *widget, ApInfo_t *apinfo, PtCallbackInfo_t *cbinfo )
{
  toolbar_t         *tb = tb_data(widget) ;    // Указатель на данные тулбара.
  PhEvent_t         *event = cbinfo->event ;   // Произошедшее событие.
```

```
  if( event->subtype == Ph_EV_PTR_LEAVE )
  {
    // Курсор мыши вышел за границу тулбара.
    // Вызвать управляющий автомат.
    A0( 40, tb, event ) ;
  } ;

  return( Pt_CONTINUE ) ;
} ;

//======
// Функция, реализующая автомат управления тулбаром.
//
void A0( int e, toolbar_t *tb, PhEvent_t *event )
{
  int y_old = tb->y0 ;

  #ifdef GRAPH_EVENTS_LOGGING
    log_exec( "A0", y_old, e ) ;
  #endif

  switch( tb->y0 )
  {
    case 0:
      if( e == 10 )                     tb->y0 = 1 ;
    break ;

    case 1:
      if( e == 20 ) { z30(tb) ;         tb->y0 = 0 ; }
      else
      if( e == 30 )                     tb->y0 = 2 ;
      else
      if( e == 40 )                     tb->y0 = 0 ;
    break ;

    case 2:
      if( e == 30 ) { z20(tb, event) ; }
      else
      if( e != 30 )                     tb->y0 = 0 ;
    break ;

    default:
      #ifdef GRAPH_ERRORS_LOGGING
        log_write( LOG_GRAPH_ERROR, "ОШИБКА В A0: неизвестное состояние!", 0 ) ;
      #endif
    break ;
  } ;

  // Если состояние не изменилось - завершить выполнение функции.
  if( y_old == tb->y0 ) goto A0_end ;

  #ifdef GRAPH_TRANS_LOGGING
    log_trans( "A0", y_old, tb->y0 ) ;
  #endif

  switch( tb->y0 )
  {
    case 1:
      z10(tb, event) ; z40(tb) ;
    break ;

    case 2:
      z20(tb, event) ;
    break;
  } ;

  A0_end: ;
  #ifdef GRAPH_ENDS_LOGGING
    log_end( "A0", tb->y0, e ) ;
  #endif
} ;

//======
// Запомнить координаты курсора мыши.
//
```

```
void z10( toolbar_t *tb, PhEvent_t *event )
{
  PhRect_t      *rect = NULL ;   // Координаты курсора мыши.

  #ifdef ACTIONS_LOGGING
    log_write( LOG_ACTION, "z10. Запомнить координаты курсора мыши.", 0 ) ;
  #endif

  rect = PhGetRects( event ) ;
  tb->drag_pos = rect->ul ;
} ;

//======
// Переместить тулбар.
//
void z20( toolbar_t *tb, PhEvent_t *event )
{
  PhRect_t      *rect = NULL ;   // Координаты курсора мыши.

  #ifdef ACTIONS_LOGGING
    log_write( LOG_ACTION, "z20. Переместить тулбар.", 0 ) ;
  #endif

  rect = PhGetRects( event ) ;
  toolbar_move( tb, rect->ul.x - tb->drag_pos.x, rect->ul.y - tb->drag_pos.y ) ;
} ;

//======
// Вызвать меню тулбара.
//
void z30( toolbar_t *tb )
{
  #ifdef ACTIONS_LOGGING
    log_write( LOG_ACTION, "z30. Вызвать меню тулбара.", 0 ) ;
  #endif

  ApCreateModule( ABM_toolbar_menu, NULL, NULL ) ;
} ;

//======
// Переместить окно тулбара выше остальных.
//
void z40( toolbar_t *tb )
{
  #ifdef ACTIONS_LOGGING
    log_write( LOG_ACTION, "z40. Переместить окно тулбара выше остальных.", 0 ) ;
  #endif

  PtWindowToFront( tb->wgt ) ;
} ;
```

ПРИЛОЖЕНИЕ 4

"Полный" протокол с проверкой всех переходов автомата, реализующего алгоритм управления тулбаром

Обработка события "нажатие правой кнопки мыши"

```
16:44:58.543{ A0: в состоянии 0 запущен с событием e10
16:44:58.543T A0: перешел из состояния 0 в состояние 1
16:44:58.543* z10. Запомнить координаты курсора мыши.
16:44:58.543* z40. Переместить окно тулбара выше остальных.
16:44:58.543} A0: завершил обработку события e10 в состоянии 1
```

Обработка события "отпускание правой кнопки мыши" - вызов меню тулбара.

```
16:44:59.903{ A0: в состоянии 1 запущен с событием e20
16:44:59.903* z30. Вызвать меню тулбара.
16:44:59.903T A0: перешел из состояния 1 в состояние 0
16:44:59.903} A0: завершил обработку события e20 в состоянии 0
```

Обработка события "выход курсора мыши за границу тулбара"

```
16:44:59.903{ A0: в состоянии 0 запущен с событием e40
16:44:59.903} A0: завершил обработку события e40 в состоянии 0
```

Обработка события "нажатие правой кнопки мыши"

```
16:45:03.963{ A0: в состоянии 0 запущен с событием e10
16:45:03.963T A0: перешел из состояния 0 в состояние 1
16:45:03.963* z10. Запомнить координаты курсора мыши.
16:45:03.963* z40. Переместить окно тулбара выше остальных.
16:45:03.963} A0: завершил обработку события e10 в состоянии 1
```

Обработка события "выход курсора мыши за границу тулбара"

```
16:45:05.933{ A0: в состоянии 1 запущен с событием e40
16:45:05.933T A0: перешел из состояния 1 в состояние 0
16:45:05.933} A0: завершил обработку события e40 в состоянии 0
```

Обработка события "нажатие правой кнопки мыши"

```
16:45:10.482{ A0: в состоянии 0 запущен с событием e10
16:45:10.482T A0: перешел из состояния 0 в состояние 1
16:45:10.482* z10. Запомнить координаты курсора мыши.
16:45:10.482* z40. Переместить окно тулбара выше остальных.
16:45:10.482} A0: завершил обработку события e10 в состоянии 1
```

Обработка событий "перемещение мыши с нажатой кнопкой"

```
16:45:12.812{ A0: в состоянии 1 запущен с событием e30
16:45:12.812T A0: перешел из состояния 1 в состояние 2
16:45:12.812* z20. Переместить тулбар.
16:45:12.812} A0: завершил обработку события e30 в состоянии 2
16:45:12.852{ A0: в состоянии 2 запущен с событием e30
16:45:12.852* z20. Переместить тулбар.
16:45:12.852} A0: завершил обработку события e30 в состоянии 2
```

Обработка события "отпускание правой кнопки мыши"

```
16:45:15.812{ A0: в состоянии 2 запущен с событием e20
16:45:15.812T A0: перешел из состояния 2 в состояние 0
16:45:15.812} A0: завершил обработку события e20 в состоянии 0
```

Обработка события "выход курсора мыши за границу тулбара"

```
16:45:16.742{ A0: в состоянии 0 запущен с событием e40
16:45:16.742} A0: завершил обработку события e40 в состоянии 0
```

Обработка события "нажатие правой кнопки мыши"

```
16:45:18.992{ A0: в состоянии 0 запущен с событием e10
16:45:18.992T A0: перешел из состояния 0 в состояние 1
```

```
16:45:18.992* z10. Запомнить координаты курсора мыши.
16:45:18.992* z40. Переместить окно тулбара выше остальных.
16:45:18.992} A0: завершил обработку события e10 в состоянии 1
```

Обработка событий "перемещение мыши с нажатой кнопкой"

```
16:45:20.472{ A0: в состоянии 1 запущен с событием e30
16:45:20.472T A0: перешел из состояния 1 в состояние 2
16:45:20.472* z20. Переместить тулбар.
16:45:20.472} A0: завершил обработку события e30 в состоянии 2
16:45:21.192{ A0: в состоянии 2 запущен с событием e30
16:45:21.192* z20. Переместить тулбар.
16:45:21.192} A0: завершил обработку события e30 в состоянии 2
```

Обработка события "выход курсора мыши за границу тулбара" - прекращение перемещения

```
16:45:21.232{ A0: в состоянии 2 запущен с событием e40
16:45:21.232T A0: перешел из состояния 2 в состояние 0
16:45:21.232} A0: завершил обработку события e40 в состоянии 0
```