В.И. Шелехов

Язык и технология автоматного программирования

Разработана технология автоматного программирования в интеграции с технологиями предикатного и объектно-ориентированного программирования. Автоматная программа реализует конечный автомат в виде гиперграфа управляющих состояний. Технология автоматного программирования иллюстрируется на примерах программы моделирования электронных часов с будильником и протокола передачи данных ATM Adaptation Layer уровня Type 2 AAL.

Ключевые слова: понимание программ, автоматное программирование, предикатное программирование, уровень адаптации ATM

1. ВВЕДЕНИЕ

Автоматная программа определяется в виде конечного автомата и состоит из нескольких сегментов кода. В качестве примера автоматной программы рассмотрим модуль операционной системы, реализующий следующий сценарий работы с пользователем, показанный на рис.1.

```
login:   if (exists(get_string()))
                goto passw
         else   goto login
passw:   if (valid(get_string()))
                goto session
         else   goto passw
session: <сессия пользователя>;
         goto login
```

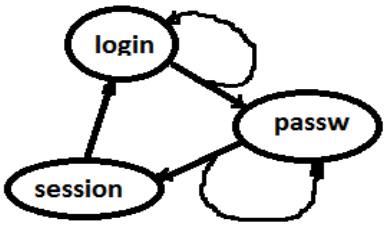

Рис. 1 – Схема работы ОС с пользователем

В управляющем состоянии login операционная система запрашивает имя пользователя. Если полученное от пользователя имя существует в системе, система переходит в управляющее состояние passw, иначе возвращается в состояние login. В состоянии passw система запрашивает пароль. Если поданная пользователем строка соответствует правильному паролю, то пользователь допускается к работе и система переходит в состояние session. При завершении работы пользователя система переходит в состояние login.

Всякая вершина автомата соответствует некоторому *управляющему состоянию*. Ориентированная гипердуга автомата соответствует некоторому *сегменту кода* и связывает одну вершину с одной или несколькими другими вершинами. Исполнение сегмента завершается оператором перехода на начало другого сегмента.

Понятие автоматной программы концептуально не отличается от введенного Анатолием Шалыто [19], однако различия в языке и технологии существенны.

Определим следующие классы программ.

Класс *невзаимодействующих программ* или класс *программ-функций*. Программа принадлежит этому классу, если она не взаимодействует с внешним окружением. Точнее, если возможно перестроить программу таким образом, чтобы все операторы ввода данных находились в начале программы, а весь вывод собран в конце программы, то такая программа относится к классу невзаимодействующих программ. Программа обязана всегда завершаться, поскольку бесконечно работающая и невзаимодействующая программа бесполезна. Следовательно, программа определяет функцию, вычисляющую по набору

входных данных (аргументов) некоторый набор результатов. Класс программ-функций, по меньшей мере, содержит программы для задач дискретной и вычислительной математики.

Программа-функция реализует решение (алгоритм) некоторой математической *задачи*. Решение задачи строится на базе *логики решения* – набора математических свойств (теорем). Программирование – это реализация логики решения в конструкциях языка программирования. Понимание программы реализуется в процессе сопоставления кода программы с логикой решения. В качестве иллюстрации рассмотрим логику решения для простейшей программы умножения натуральных чисел a и b через операцию сложения:

$$a * b = b + (a - 1) * b, \quad 0 * b = 0 . \tag{1}$$

Ниже приведены логическая, функциональная и императивная программы, являющиеся реализациями логики решения (1).

$$0 * b \to 0;\ \ a * b \to b + (a - 1) * b \tag{2}$$

```
nat mult(nat a, b) { if (a = 0) 0 else b + mult(a – 1, b) }   (3)
```

```
nat c = 0; while (a != 0) { c = c + b; a = a – 1 }   (4)
```

Различия логической программы (2) и логики (1) чисто синтаксические. Фактически они тождественны. Функциональная программа (2) и логика (1) отличаются. Однако они также тождественны – программа (2) легко транслируется в логику (1). Императивная программа (4) также построена из логики (1), однако понять ее и установить тождественность с логикой (1), – нетривиальная задача из-за трудности понимания циклов типа **while**[1].

Класс *программ-процессов* (*автоматных программ*). Программа данного класса является автоматной программой, состоящей из набора сегментов кода. Всякий сегмент есть либо программа-функция, либо программа-процесс, декомпозиция которой представлена другим автоматом. Исполнение автоматной программы может быть бесконечным процессом. Взаимодействие с внешним окружением автоматной программы реализуется через прием и посылку *сообщений*, а также через *разделяемые переменные*, доступные в данной программе, а также в других программах из окружения программы. Операторы *ввода* и *вывода* рассматриваются как упрощенная форма операторов посылки и приема сообщений. *Состояние* автоматной программы определяется значениями набора переменных, модифицируемых в программе, за исключением локальных переменных. Управляющее состояние программы идентифицирует текущий исполняемый сегмент.

Важнейшим подклассом автоматных программ являются контроллеры систем управления в аэрокосмической отрасли, энергетике, медицине, массовом транспорте и др. отраслях. На каждом шаге вычислительного цикла контроллер получает входную информацию из окружения и обрабатывает ее. Результаты вычисления используются для передачи управляющего сигнала в окружение контроллера. Значительная часть *программируемых логических контроллеров* PLCs (programmable logic controllers) разрабатывается в соответствии со стандартом IEC 61131-3.

Отметим, что классы программ-функций и программ-процессов не охватывают всего множества программ. Например, компиляторы и операционные системы относятся к более сложным классам программ.

Автоматные программы по своей структуре существенно сложнее программ-функций. Автоматное программирование разработано для программ-процессов, и его применение не оправдано для программ-функций. Технология автоматного программирования должна быть интегрирована с технологиями для класса программ-функций, поскольку автоматная программа строится из программ-функций.

В целях улучшения понимания автоматных программ одной из задач технологии автоматного программирования является применение методов, позволяющих упростить сегменты кода автоматной программы. В частности, здесь применимы методы объектно-ориентированного и предикатного программирования: введение объектов вместо наборов переменных позволяют разгрузить автоматную программу, локализуя часть связей внутри

[1] Косвенным подтверждением является чрезвычайная трудность автоматического построения инвариантов циклов.

классов, а использование функциональных (предикатных) программ вместо аналогичных императивных программ позволяет существенно упростить программу.

Общее описание парадигмы предикатного программирования дано в разделе 2. Базисные конструкции языка автоматного программирования определены в разделе 3. На примере программы моделирования электронных часов с будильником в разделе 4 определяется базис технологии автоматного программирования. В разделе 5 технология автоматного программирования иллюстрируется на примере сложного протокола передачи данных ATM Adaptation Layer уровня Type 2 AAL [1]. Обзор смежных работ представлен в разделе 6. В заключении определяются задачи развития предложенного подхода для различных подклассов программ-процессов.

2. ПРЕДИКАТНОЕ ПРОГРАММИРОВАНИЕ

Предикатная программа относится к классу программ-функций и является предикатом в форме вычислимого оператора, адекватно представляющим логику решения задачи, обеспечивая хорошее понимание программы. Язык предикатного программирования P [3] обладает большей выразительностью в сравнение с языками функционального программирования и по стилю ближе к императивному программированию.

Предикатная программа состоит из набора рекурсивных программ (определений предикатов) на языке P следующего вида:

<имя программы>(<описания аргументов>**:** <описания результатов>)
pre <предусловие>
{ <оператор> }
post <постусловие>

Необязательные конструкции: предусловие и постусловие – являются формулами на языке исчисления предикатов; они используются для улучшения понимания программ и для дедуктивной верификации [4–6, 8]. Ниже представлены основные конструкции языка P: оператор присваивания, блок (оператор суперпозиции), параллельный оператор, условный оператор, вызов программы и описание переменных, используемое для аргументов, результатов и локальных переменных.

<переменная> = <выражение>
{<оператор1>; <оператор2>}
<оператор1> || <оператор2>
if (<логическое выражение>) <оператор1> **else** <оператор2>
<имя программы>(<список аргументов>**:** <список результатов>)
<тип> <пробел> <список имен переменных>

В предикатном программировании запрещены такие языковые конструкции, как циклы и указатели, серьезно усложняющие программу. Известно, что функциональная программа в несколько раз проще в сравнении с императивной программой, реализующей тот же алгоритм, потому что вместо циклов используются рекурсивные функции, а вместо массивов и указателей – списки.

Эффективность предикатных программ достигается применением следующих оптимизирующих трансформаций, переводящих программу на императивное расширение языка P:

- замена хвостовой рекурсии циклом;
- подстановка тела программы на место ее вызова;
- склеивание переменных: замена всех вхождений одной переменной на другую переменную;
- кодирование алгебраических типов (списков, деревьев, …) с помощью массивов и указателей.

Эффективность программы после применения трансформаций обеспечивается оптимизацией, реализуемой программистом, на уровне предикатной программы. Для

приведения рекурсии к хвостовому виду применяется метод обобщения исходной задачи. Далее обычно открывается возможность проведения серии дальнейших улучшений алгоритма. Итоговая программа по эффективности не уступает написанной вручную и, как правило, короче [4–8]. Отметим, что в функциональном программировании (при общеизвестной ориентации на предельную компактность и декларативность [12]) оптимизация программы полностью возлагается на транслятор, в частности, обеспечивается автоматическое приведение рекурсии к хвостовому виду. Разумеется, функциональное программирование существенно уступает в эффективности, поскольку даже применением изощренных методов оптимизации невозможно автоматически воспроизвести серию оптимизаций, свершаемых программистом вручную.

Гиперфункции. Рассмотрим типичное решение задачи: взять второй элемент списка s и обработать его программой B; если второго элемента нет, запустить программу D.

```
Elem2(s: e, exists); if (exists) B(e…) else D(…);        (5)
```

Программа Elem2 на языке P:

```
pred Elem2(list(int) s: int e, bool exists )
{   if (s = nil ∨ s.cdr = nil)    exsts = false
    else  { e = s.cdr.car || exists = true }
} post exists = (s ≠ nil  & s.cdr ≠ nil) &  (exists ⇒ e = s.cdr.car);
```

Использование логической переменной exists для реализации ветвления в (5) – плохое решение, загромождающее и усложняющее программу. Золотое правило программирования[2] состоит в том, чтобы не использовать логических переменных для реализации ветвления в программе[3]. Правильным решением является следующая *гиперфункция* для Elem2:

```
hyper Elem2(list (int) s : int e #1: #2 )
pre 1: s ≠ nil & s.cdr ≠  nil
{   if (s = nil  ∨  s.cdr = nil)   #2
    else  { e = s.cdr.car;   #1  }
} post 1: e = s.cdr.car;
```

Гиперфункция Elem2 имеет две *ветви* результатов: первая ветвь включает переменную e, а вторая ветвь не имеет результатов – она пуста. *Метки* 1 и 2 – дополнительные параметры, обозначающие два различных *выхода* гиперфункции. *Спецификация гиперфункции* состоит из двух частей. Утверждение после "**pre** 1:" есть предусловие первой ветви; предусловие второй ветви – просто отрицание предусловия первой ветви. Утверждение после "**post** 1:" есть постусловие для первой ветви. Фрагмент (5) заменяется следующим:

```
Elem2(s: e #L1: #L2)  case L1: B(e…) case L2: D(…);
```

Исполнение вызова гиперфункции завершается переходом либо по метке L1, либо по метке L2. Операторы перехода #L1 и #L2 внутри вызова гиперфункции явным образом фиксируют точки передачи управления в отличие от вызова вида Elem2(s, e, L1, L2) на языке типа Алгол-60, где переходы по меткам L1 и L2 синтаксически не выделены и их можно не заметить. Эта одна из причин, почему оператор перехода **goto** L1 записывается в виде #L1.

Чтобы дать общее определение гиперфункции, рассмотрим следующее определение предиката:

---

[2] Геннадий Кожухин, разработчик транслятора АЛЬФА, собирал золотые правила программирования в 1960-х гг.

[3] Это противоречит принципу структурного программирования, поскольку структурирование произвольной программы в соответствии с теоремой Боэма и Джакопини реализуется с использований дополнительных логических переменных.

pred A(x**:** y, z, c) (6)
pre P(x)
{ … }
post c = C(x) & (C(x) ⇒ S(x, y)) & (¬C(x) ⇒ R(x, z));

Здесь x, y и z – возможно пустые наборы переменных; P(x), C(x), S(x, y) и R(x, z) – логические утверждения. Предположим, что все присваивания вида c = **true** и c = **false** – последние исполняемые операторы в теле предиката. Определение предиката (6) может быть заменено следующим определением гиперфункции:

hyp A(x**:** y #1**:** z #2) (7)
pre P(x) **pre** 1**:** C(x)
{ … }
post 1: S(x, y) **post** 2: R(x, z);

В теле гиперфункции каждое присваивание c = **true** заменено на #1, а c = **false** – на #2.

Допустим, имеется условный оператор вида

if (C) {A; B} **else** {E; D} (8)

где A, B, E и D – некоторые операторы. Определим гиперфункцию:

hyper H(…**:** …#1**:** …#2) **pre** 1**:** C { **if** (C) {A; #1} **else** {E; #2} } .

Тогда условный оператор (8) эквивалентен следующей композиции:

H(…**:** …#1**:** …#2) **case** 1**:** B **case** 2**:** D .

Данное свойство демонстрирует, что аппарат гиперфункций позволяет проводить гибкую декомпозицию программы, недостижимую другими средствами [8, 4, 21]. Аппарат гиперфункций является более общим в сравнении с обработкой исключений в языках Java и C#. Гиперфункции не представимы монадами[4] функциональных языков. Использование гиперфункций делает программу короче, быстрее и проще для понимания [8, 4, 21]. Отметим, что модель программ-процессов в форме гиперграфа является естественным продолжением аппарата гиперфункций.

3. ЯЗЫК АВТОМАТНОГО ПРОГРАММИРОВАНИЯ

Ограничимся рассмотрением простых процессов без параллелизма и недетерминизма. Язык автоматного программирования строится расширением языка для класса программ-функций. Программа-процесс определяется следующей конструкцией:

process <имя программы>(<описания аргументов и результатов>)
pre <предусловие>
{ <описания переменных состояния процесса>
<сегменты кода>
}

Произвольный <сегмент кода> представляется следующим образом:

<имя управляющего состояния>**:**
inv <инвариант сегмента>;
<оператор>;

<Инвариант сегмента> должен быть истинным перед выполнением <оператора>. Инвариант не является обязательным, он используется для улучшения понимания автоматной программы и для верификации. Исполнение <оператора> завершается исполнением оператора перехода на начало другого сегмента автоматной программы. При нормальном завершении <оператора> исполнение продолжится с начала следующего сегмента.

Для взаимодействия с внешним окружением используются средства ввода и вывода данных, посылки и приема сообщений с возможными параметрами. Оператор **send** m(e)

[4]Возможности, определяемые монадами, явным образом предоставляются в языке P.

посылает сообщение m с параметрами, значения которых определяются набором выражений e. Оператор приема сообщений определяется следующим правилом:

<оператор приема сообщений>::=
receive <имя сообщения>(<параметры>) <оператор> |
receive <имя сообщения> (<параметры>){ <оператор> }
else <оператор>

Для второго варианта правила при отсутствии сообщения в канале исполняется **else**-часть оператора, возможно содержащая прием других сообщений. Если все альтернативы определяют прием сообщений и все эти сообщения отсутствуют, <оператор приема сообщений> исполняется многократно до тех пор, пока в канале не появится одно из сообщений, принимаемых оператором. После получения сообщения переменные, указанные <параметрами>, получают значения параметров сообщения, и выполняется соответствующий <оператор>.

4. ПРИМЕР: ЭЛЕКТРОННЫЕ ЧАСЫ С БУДИЛЬНИКОМ

На примере автоматной программы «Электронные часы с будильником» [19] дадим иллюстрацию технологии автоматного программирования, интегрированной здесь с объектно-ориентированной технологией. Язык предикатного программирования P расширяется описаниями классов и конструкцией <объект>.<имя элемента класса> для доступа к элементу некоторого объекта. Описание класса определяет переменные, константы и методы как элементы класса. Описание метода представляется в виде определения предиката.

Рассмотрим устройство электронных часов с будильником [19]. На корпусе часов имеется три кнопки:

- H (Hours)– увеличивает на единицу число часов;
- M (Minutes)– увеличивает на единицу число минут;
- A (Alarm) – включает и выключает будильник.

Увеличение часов и минут происходит по модулю 24 и 60 соответственно. Если будильник выключен, то кнопка A включает его и переводит часы в режим, в котором кнопки H и M устанавливают не текущее время, а время срабатывания будильника. Повторное нажатие кнопки A переводит часы в режим с включенным будильником, в котором кнопки H и M будут менять время на часах. В режиме с включенным будильником, если текущее время совпадает со временем будильника, включается звонок, который отключается либо нажатием кнопки A, либо самопроизвольно через минуту. Нажатие кнопки A в режиме с включенным будильником переводит часы в нормальный режим без будильника.

Набор различных действий с часами, встречающихся в автоматной программе моделирования функционирования часов, представим в виде класса Часы.

```
class Часы {
  nat  hours, minutes;  // текущее время (часы, минуты)
  nat  alarm_hours, alarm_minutes; // время срабатывания будильника
  inc_h(); {...}  // увеличить время на один час
  inc_m(); {...} // увеличить время на одну минуту
  inc_alarm_h(); {...} // увеличить время будильника на час
  inc_alarm_m(); {...} // увеличить время будильника на минуту
  tick() {...} // приращение времени на тик – минимальный интервал
  bell_on() {...} // Включить звонок
  bell_off() {...} // Выключить звонок
  bool bell_limit() {...};  // звонок звонит уже минуту
}
```

Использование класса Часы позволяет существенно разгрузить и, тем самым, упростить автоматную программу. В качестве состояния автоматной программы используется одна переменная t, объект класса Часы, вместо четырех переменных, которые были бы использованы в версии автоматной программы без применения объектно-ориентированной технологии. Вместе с четырьмя переменными класс Часы также локализует внутри себя набор условий, формулируемых обычно в виде инвариантов, а также предусловий и постусловий методов.

Имеются три управляющих состояния автоматной программы:

- off – режим работы часов без будильника;
- set – режим установки будильника;
- on – режим работы часов с включенным будильником.

Представленная ниже автоматная программа является непосредственной формализацией данного выше содержательного описания функционирования часов. Разумеется, было бы правильным реализовать естественный ход часов независимым параллельным процессом. Данный алгоритм выбран, потому что он ближе к алгоритму в работе [19].

```
process Работа_часов_с_будильником {                                          (9)
  Часы t = Часы();
  off:   receive H          { t.inc_h()    #off }                              (9.1)
         else receive M     { t.inc_m()    #off }
         else receive A             { #set }                                  (9.2)
         else  { tick()  #off }
  set:   receive H          { t.inc_alarm_h()    #set }
         else receive M     { t.inc_alarm_m()  #set }
         else receive A     { t.bell_on()          #on }
         else  { tick()  #set }
  on:    receive H         { t.inc_h()      #on }
         else receive M     { t.inc_m()      #on }
         else receive A     { t.bell_off()   #off }
         else  { tick(); if (t.bell_limit()) { t.bell_off() #off } else #on }   (9.3)
}
```

Сравним данную программу с аналогичной, описанной в работе [19]. С этой целью трансформируем ее так, чтобы получить программу в соответствии со switch-технологией А. Шалыто [2]. Для множества управляющих состояний введем описание типа:

```
type State = enum(off, set, on)
```

Введем переменную state, определяющую значение текущего управляющего состояния. Заменим операторы перехода на соответствующие присваивания переменной state, например, переход #off заменим на state = off; далее часть таких присваиваний можно опустить в случаях, когда управляющее состояние не меняется. Для реализации переходов на нужные управляющие состояния используем оператор **switch** по значению переменной state. Наконец, представим итоговую программу в виде бесконечного цикла.

```
process Работа_часов_с_будильником {                                    (10)
  Часы t = Часы();
  State state = off;
  while (true)
  switch (state) {
    case off:  receive H          { t.inc_h() }                        (10.1)
               else receive M     { t.inc_m() }
               else receive A     { state = set }                      (10.2)
               else               { tick() }
    case set:  receive H          { t.inc_alarm_h() }                  (10.3)
               else receive M     { t.inc_alarm_m() }
               else receive A     { t.bell_on();  state = on }
               else               { tick() }
    case on:   receive H     { t.inc_h() }
               else receive M     { t.inc_m() }
               else receive A     { t.bell_off(); state = off }
               else               { tick(); if (t.bell_limit()) { t.bell_off(); state = off } }   (10.4)
}}
```

Программа (10) похожа на одну из версий этой программы в работе [19, 2.3.2, стр. 92]. Сравним программы (9) и (10), см. рис. 2. В соответствии с присваиванием state = set в строке (10.2) дальнейшее управление через оператор **switch** будет передано на строку (10.3). Аналогичный переход в программе (9) реализуется явным образом оператором #set в строке (9.2). Поскольку в строке (10.1) нет присваиваний переменной state, она сохраняет прежнее значение off; поэтому в соответствии с оператором **switch** управление будет передано снова на строку (10.1). Аналогичный переход в программе (9) реализуется явно оператором #off в строке (9.1). Таким образом, программа (9) проще, поскольку все переходы в ней реализуются явно операторами перехода, тогда как аналогичные переходы в программе (10) опосредованы через присваивания переменной state и механизм действия оператора **switch**.

```
                                          while (true)
                                          switch (state) {
off:  receive H  { t.inc_h()  #off }  (9.1)   case off: receive H  { t.inc_h() }   (10.1)
      else receive M { t.inc_m()  #off }              else receive M  { t.inc_m() }
      else receive A  #set            (9.2)         else receive A { state = set }(10.2)
      else     { tick()  #off }                     else  { tick() }
 set: receive H  { t.inc_alarm_h()  #set }     case set: receive H { t.inc_alarm_h() } (10.3)
```

Рис. 2 – Сравнение программ (9) и (10) на начальном сегменте кода.

Последний сегмент программы (9) имеет структуру гипердуги. Чтобы привести программу (9) в соответствии с классическим конечным автоматом, необходимо добавить четвертое управляющее состояние после вызова tick() на строке (9.3). В программе, представленной листингом 2.8 [19, 2.3.2, стр. 92], удается избежать добавления управляющего состояния, однако сложным и неадекватным образом, подобно втягиванию вызова tick() внутрь условного оператора на строке (10.4) с добавлением в условном операторе еще двух ветвей. Это лишь один показательный пример того, что используемая в нашем подходе гиперграфовая структура обладает лучшими выразительными возможностями и более адекватна в сравнении со структурой классического конечного автомата.

Сформулируем требования, составляющие дисциплину автоматного программирования. Во-первых, в автоматной программе переходы разрешены лишь на начало одного из сегментов, либо на метку ветви завершения процесса, указанную в <описании аргументов

и результатов>. Во-вторых, все сегменты автоматной программы (а, значит, и все переходы автоматной программы) должны быть одновременно визуально доступны. Это важное эргономическое требование, означающее, что автоматная программа должна полностью помещаться на экране дисплея. Требуемое сокращение размера автоматной программы, в частности, достигается переносом ее частей в методы или подпрограммы. В предельном случае, сегмент кода может быть заменен соответствующим вызовом гиперфункции. Например, первый сегмент программы (9) может быть замене на:

```
off:  Шаг_часов_без_будильника (t: #of: #set)
```

Более радикальной является декомпозиция автоматной программы, при которой ее части определяются независимыми процессами. Например, возможна следующая декомпозиция программы (9).

```
process Часы_без_будильника (Часы t: # set) {
 1:     receive H          { t.inc_h()    #1 }
        else receive M     { t.inc_m()    #1 }
        else receive A                     #set
        else  { tick()  #1 }
}
process Установка_будильника (Часы t: # on) {
 set:   receive H          { t.inc_alarm_h()    #set }
        else receive M     { t.inc_alarm_m()  #set }
        else receive A     { t.bell_on()          #on }
        else  { tick()  #set }
}
process Часы_с_установленным_будильником (Часы t: # off) {
 3:     receive H          { t.inc_h()      #3 }
        else receive M     { t.inc_m()      #3 }
        else receive A     { t.bell_off()   #off }
        else  { tick(); if (t.bell_limit()) { t.bell_off() #off } else #3 }
}
process Работа_часов_с_будильником {
 Часы t = Часы();
 off:  Часы_без_будильника (t: # set);
 set:  Установка_будильника (t: #on);
 on:   Часы_с_установленным_будильником (t: #off)
}
```

Возможен другой способ декомпозиции: любая пара сегментов программы (9) может быть объединена в единый независимый процесс. Например:

```
process Работа_часов_с_будильником {
 Часы t = Часы();
 off:  Часы_без_будильника_Установка_будильника (t: #on);
 on:   Часы_с_установленным_будильником (t: # off)
}
```

Процесс из одного сегмента кода можно было бы представить оператором цикла, например:

```
process Часы_без_будильника (Часы t: # set) {
  loop { receive H          t.inc_h()
         else receive M     t.inc_m()
         else receive A           #set
         else   tick()
}}
```

Тем не менее, мы запрещаем использование операторов цикла для описания процессов в автоматной программе. Они, безусловно, сокращают размер кода, однако не улучшают понимания программы. Использование операторов цикла неприемлемо для декомпозиции процессов в автоматных программах и в отдельных случаях увеличивает число управляющих состояний.

5. ПЕРЕДАТЧИК В ПРОТОКОЛЕ AAL-2

Технология автоматного программирования иллюстрируется на программе Передатчик в протоколе AAL-2, описанной на языке спецификаций SDL [17] в разделе 10.1 стандарта [1]. В нашей программе опущена детальная информация по протоколу, напрямую не связанная с алгоритмом передачи данных. Хотя представленная программа значительно отличается от оригинального алгоритма [1, разд. 10.1], большая часть обозначений сохранена. Использование аппарата гиперфункций, а также списков вместо массивов позволяет существенно упростить программу Передатчик. Однако она остается достаточно сложной. Поэтому применяется двухуровневый подход к ее построению на базе скелета, получаемого из предикатной программы передачи пакетов данных с плотной упаковкой в формате блоков постоянной длины.

Уровень адаптации 2 (AAL type 2) в асинхронном способе передачи данных ATM (Asynchronous Transfer Mode) применяется для эффективной передачи низкоскоростных, коротких пакетов переменной длины в приложениях, чувствительных к задержкам. Используются три уровня передачи данных: подуровень конвергенции SSCS (Service Specific Convergence Sublayer), общий подуровень CPS (Common Part Sublayer) и уровень ATM. Передатчик получает пакеты с уровня SSCS, плотно пакует их в один или несколько блоков данных и посылает на уровень ATM.

Передатчик получает очередной пакет через сообщение CPSpacket(pd, atrs), где pd – данные пакета переменной длины, atrs – атрибуты пакета. Передатчик строит *заголовок пакета* ph (packet header) длиной 3 октета[5]. Пакет, состоящий из ph и pd, пакуется в один или несколько блоков данных PDUs (protocol data units). Блок данных PDU состоит из *начального поля* STF (start field) длиной в 1 октет и *основной части* (payload) длиной 47 октетов. Пакет, не вместившийся в очередном блоке PDU, продолжается в следующем PDU. Начальное поле STF блока PDU содержит длину в октетах перенесенной части пакета из предыдущего блока PDU. Очередной построенный блок PDU передается на уровень ATM посылкой сообщения ATM_data(PDU).

Упаковка пакетов в блоке PDU плотная. Следующий пакет размещается в блоке PDU непосредственно после предыдущего. Блок PDU может содержать несколько пакетов. Заголовок ph также может быть перенесен из одного блока в другой; см. рис. 3a и 3б.

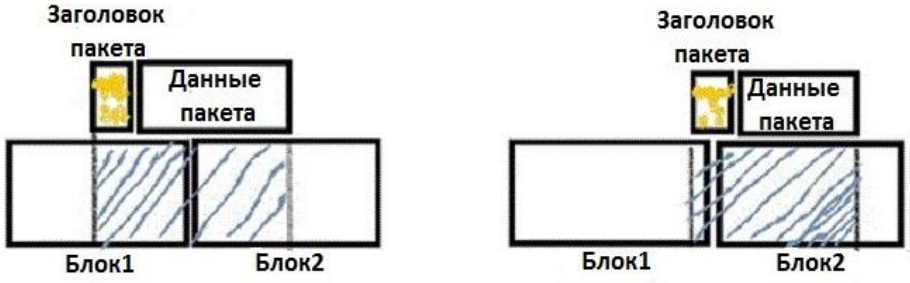

Рис. 3: Схема переноса пакета:
a – перенос данных; b – перенос заголовка

Остальную информацию по программе Передатчик представим позже.

[5] Октет состоит из 8 битов; термин «байт» не используется, т.к. имеет иной смысл.

Построим программу transfer, являющуюся скелетом программы Передатчик и реализующую передачу пакетов с плотной упаковкой в последовательности блоков PDU. Очередной блок PDU формируется в буфере buf длины 48 октетов. Для переписи на буфер buf произвольного пакета x произвольной длины применяется гиперфункция move. Предварительно дадим необходимые описания программы:

```
type OCTET = byte;
type DATA = list(OCTET);
type ATRS; // атрибуты пакета
```

Тип DATA определяет списки, элементами которого являются октеты.

```
pred split(DATA x, nat r: DATA y, x′)
     pre len(x) ≥ r post x = y + x′ & len(y) = r;
```

Здесь обозначение x′ используется для итогового значения x, модифицированного программой. Операция "+" означает конкатенацию двух списков. Программа split извлекает список y длины r из начала списка x; при этом x′ – оставшаяся часть списка.

Гиперфункция move добавляет список x к списку buf. Первая ветвь гиперфункции соответствует случаю, когда x по длине помещается в оставшуюся часть буфера buf.

```
hyper move(DATA x, buf : DATA x′, buf′ #1  : DATA x′, buf′ #2)
   pre len(buf) ≤ 48       pre 1: len(buf) + len(x) ≤ 48
{  if (len(buf) + len(x) > 48)  { split(x, 48 - len(buf): DATA y, x′); buf′ = buf + y #2 }
   else { buf′ = buf + x; x′ = nil #1 }
} post 1: len(buf′) ≤ 48  & buf′ = buf + x & x′ = nil
   post 2: len(buf′) = 48 & buf + x = buf′ + x′;
```

Если длина buf + x больше 48 (ветвь 2), список x расщепляется на y и x′, причем y дописывается в список buf, а x′ – оставшаяся часть от списка x, т.е. len(buf + y) = 48 и x = y + x′. Заметим, что присваивание x′ = nil на первой ветви необходимо, потому что переменная x′ включена в состояние процесса программы Передатчик; в противном случае x′ можно было бы изъять из результатов первой ветви, что упростило бы алгоритм.

Чтобы обеспечить эффективность программы в трансляторе с языка P реализуется трансформация замены списков массивами. Список x кодируется внутри массива X в виде вырезки массива X[jx: px-1], buf – кодируется вырезкой BUF[0: pBuf-1]. Проводятся следующие замены: len(x) → px – jx, len(buf) → pBuf. Ниже представлен код гиперфункции после трансформации.

```
hyper move(X, jx, px, BUF, pBuf : jx′, px′, pBuf′ #1 : jx′, px′, pBuf′ #2)
{  if (pBuf + px – jx > 48) { nat py = 48 + jx - pBuf; BUF[pBuf: 48] = X[jx: py-1];
                                pBuf = pBuf + py – jx; jx′=py #2 }
   else { BUF[pBuf: ] = X[jx: px-1];  pBuf = pBuf + px – jx; jx′ = px  #1 }
};
```

Определим типы данных программы transfer и предописания функций для создания заголовков STF и ph.

```
type BLOCK = DATA; // блок длиной 48 октетов
type INPUT = list(DATA); // тип последовательности пакетов
type OUT = list(BLOCK); // тип последовательности  блоков
OCTET ConstructSTF(nat len_of_packet_continuation);
DATA ConstructCPSPacketHeader(DATA pd, ATRS atrs);
```

Предикатная программа transfer преобразует потенциально бесконечную последовательность in пакетов типа INPUT в последовательность out блоков типа OUT с

плотной упаковкой. Поскольку список in никогда не завершается, можно упростить программу удалением ее ветвей для случая in = nil.

```
pred transfer(INPUT in: OUT out)
{  transfer1(in, ConstructSTF(0), nil: out) }
```

Программа transfer сводится к более общей программе transfer1, у которой второй параметр buf – текущий буфер, а третий параметр out0 – ранее сформированная последовательность блоков. В вызове transfer1 в качестве начального состояния буфера buf определяется октет STF с нулевой длиной перенесенной части, поскольку предыдущий пакет отсутствует.

```
pred transfer1(INPUT in, BLOCK buf, OUT out0: OUT out)
{ DATA pd = in.car, ph = ConstructCPS_PacketHeader(pd);
  transfer2(in, ph, pd, buf, out0: out)
}
```

В программе transfer1 из входного потока in извлекается очередной пакет pd, и конструируется заголовок пакета ph. Пакет pd и заголовок ph – дополнительные параметры более общей программы transfer2. Параметры pd и ph программы transfer2 определяют те части пакета и заголовка, которые еще не переписаны на буфер buf. В частности, если заголовок уже переписан, то ph = nil.

```
pred transfer2(INPUT in, DATA ph, pd, BLOCK buf, OUT out0: OUT out)
{ {    move(ph, buf: ph', buf' #G: ph', buf' #B);
   G:  move(pd, buf: pd', buf' #Z: pd', buf' #B);
   Z:  transfer1(in.cdr, buf', out0: out)
  } case B: { BLOCK buf1 =
                    ConstructSTF((len(ph) + len(pd) > 47)? 47 : len(ph) + len(pd) );
              transfer2(in, ph, pd, buf1, out0 + buf: out)
            }
}}
```

Используя вызовы гиперфункции move, программа transfer2 переписывает на буфер buf сначала заголовок ph, а затем пакет pd. Если оба вызова завершились по первой ветви, то передача текущего пакета завершена. Передача следующих пакетов реализуется вызовом программы transfer1. Если один из вызовов гиперфункции move завершился по второй ветви, срабатывает обработчик с меткой B. Конструируется новый буфер buf1 с начальным октетом STF, в котором фиксируется длина переносимой части текущего пакета, точнее оставшиеся (еще не переписанные) части заголовка ph и пакета pd. Оставшиеся части ph и pd переписываются в новый буфер buf1 вызовом программы transfer2, а заполненный буфер buf поступает в выходную последовательность out.

Проведем следующие трансформации предикатной программы transfer. Реализуем склеивание out0 → out в программе transfer1 и набор склеиваний в программе transfer2: out0 → out; ph' → ph; buf1, buf' → buf. Далее подставим тело программы transfer1 на место вызова в программе transfer2. Хвостовая рекурсия в transfer2 позволяет заменить рекурсию циклом. Тело программы transfer2 подставляется на место вызова в transfer1. Наконец, transfer1 подставляется в transfer. Операции с входными и выходными потоками in и out заменяются соответствующими операторами посылки и приема сообщений:

```
in = in.cdr; pd = in.car  →  receive CPSpacket(pd)
out = out + buf  →  send ATM_data(buf)
```

Замена списков ph, pd и buf массивами пока не проводится. Программа transfer после трансформаций принимает вид следующей автоматной программы:

```
process transfer()                                                    (11)
{       BLOCK buf = ConstructSTF(0);
        receive CPSpacket(pd);                                        (11.1)
        ph = ConstructCPS_PacketHeader(pd);                           (11.2)
  H:    move(ph, buf: ph, buf: ph, buf #B);
        move(pd, buf: pd, buf: pd, buf #B);
        receive CPSpacket(pd);                                        (11.3)
        ph = ConstructCPS_PacketHeader(pd);                           (11.4)
        #H                                                            (11.5)
  B:    send ATM_data(buf);
        buf1 = ConstructSTF((len(ph) + len(pd) > 47)? 47 : len(ph) + len(pd) );
        #H                                                            (11.6)
}
```

В первой ветви вызовов гиперфункции move опущены операторы перехода, поскольку переход реализуется на следующий оператор. Чтобы избежать совпадения операторов (11.1) и (11.2) с (11.3) и (11.4) в программе (12) для оператора (11.1) заведена метка IDLE. Вставка оператора (12.2) и замена (11.6) на (12.3)–(12.5) определяют оптимизации, присутствующие в исходной программе Передатчик [1].

```
process transfer()                                                    (12)
{       BLOCK buf = ConstructSTF(0);
 IDLE:  receive CPSpacket(pd);                                        (12.1)
        ph = ConstructCPS_PacketHeader(pd);
  H:    move(ph, buf: ph, buf: ph, buf #B);
  G:    move(pd, buf: pd, buf: pd, buf #B);
        if (len(buf) = 48) #B;                                        (12.2)
        #IDLE
  B:    send ATM_data(buf);
        buf = ConstructSTF((len(ph) + len(pd) > 47)? 47 : len(ph) + len(pd) );
        if (ph ≠ nil) #H                                              (12.3)
        else if (pd = nil) #IDLE                                      (12.4)
        else #G                                                       (12.5)
}
```

Программа Передатчик, представленная на рис. 4, построена модификацией программы (12) с учетом следующих дополнительных требований. Очередной заполненный блок PDU может быть отправлен на уровень ATM только после получения запроса на посылку – сообщения SEND_request(). Построение очередного блока PDU управляется таймером. При истечении установленного времени построение очередного блока PDU завершается заполнением нулями оставшейся части блока PDU вызовом программы fill со следующей спецификацией:

```
pred fill(DATA buf: DATA buf')
     pre len(buf) > 0 & len(buf) ≤ 48
     post len(buf') = 48 & ∃pad. buf' = buf + pad & null(pad);
```

Программа fill расширяет список buf до длины в 48 октетов с обнулением добавленных октетов. Дадим оставшиеся описания.

```
bool UnspecCondition;
```

Переменная UnspecCondition обозначает неопределенное условие, введенное для будущей стандартизации.

```
DATA pd = nil, ph = nil, buf; bool permit;
```

Переменные pd, ph, buf и permit определяют состояние программы Передатчика. Флаг permit равен **true** от момента получения сообщения message SEND_request() до посылки очередного блока PDU.

Определим примитивы работы с таймером. Оператор set Timer_CU включает таймер компьютера, устанавливая его в ноль. После истечения стандартного временного интервала посылается сообщение Timer_CU(). Условие active(Timer_CU) – истинно, если время еще не истекло. Всякое следующее исполнение оператора set Timer_CU отменяет установку предыдущего оператора.

Программа Передатчик имеет 11 управляющих состояний. Состояния H и G – новые. Остальные состояния – те же, что и в исходном алгоритме [1, разд. 10.1]. Состояния A и E исходного алгоритма заменены на G. Управляющие состояния снабжены инвариантом. В коде сегмента B оригинального алгоритма пересылка оставшейся части заголовка пакета на буфер buf для случая ph ≠ nil, заменена переходом #H на вызов гиперфункции move, делающий то же самое.

```
START: permit := false;
       buf = ConstructSTF(0);
IDLE: inv pd = nil & len(buf) = 1;
       receive CPSpacket(pd, ATRS atrs) {
         set Timer_CU;
C:       inv len(buf) < 48 & pd≠nil & ph=nil;
         ph=ConstructCPS_PacketHeader
                                (pd, atrs);
H:       inv len(buf) < 48 & pd≠nil & ph≠nil;
         move(ph, buf: ph', buf' #G:
                              ph', buf' #F);
G:       inv len(buf) < 48 & pd≠nil & ph=nil;
         move(pd, buf: pd', buf': pd', buf' #F);
         if (len(buf) = 48) #F
         else if (active(Timer_CU)) #PART
         else #SEND
       } else receive SEND_request() {
           if (UnspecCondition) #D
           else { permit = true; #IDLE }
       }
F:     inv len(buf) = 48;
       if (permit) #B else #FULL
D:     inv ¬active(Timer_CU) or
                        UnspecCondition;
       fill(buf: buf');
```

```
B:     inv len(buf) = 48;
       send ATM_data(buf);
       set_Timer_CU;
       permit = false;
       buf = ConstructSTF (
              if (len(ph) + len(pd) > 47)  47
              else len(ph) + len(pd) );
       if (ph ≠ nil) #H
       else if (pd = nil) #IDLE
       else #G

FULL:  inv len(buf) = 48 & ¬permit;
       receive SEND_request() #B
PART: inv len(buf) < 48 & pd = nil & ph = nil;
       receive CPSpacket(pd, ATRS atrs) #C
       else receive Timer_CU()
              { if (permit) #D else #SEND }
       else receive SEND_request()
              if (UnspecCondition)
                { set Timer_CU; #D }
              else { permit = true; #PART }
SEND: inv len(buf) < 48 & pd = nil & ph = nil
            & ¬active(Timer_CU) & ¬permit;
       receive CPSpacket(pd, ATRS atrs) #C
       else receive SEND_request() #D
```

Рис. 4 – Основная часть программы Передатчик

Реализация дополнительных требований существенно усложнила программу по сравнению с (12). Тем не менее, наша программа Передатчик на порядок проще по сравнению с исходной [1, разд. 10.1]. Состояние программы определяется 4-мя переменными против 7 переменных в оригинальном алгоритме (фактически 9, однако seq удалена в нашем Передатчик при упрощении алгоритма, а tmp – локальная). Это существенное улучшение, поскольку сложность программы экспоненциально зависит от числа переменных состояния программы.

Использование логической переменной permit в программе Передатчик – это живой пример нарушения золотого правила программирования. Возможно ли исключить эту переменную из программы? Конечно, да. Необходимо удалить прием сообщений SEND_request() в сегментах кода IDLE и PART. Модифицированная программа будет проще, однако, возможно, менее эффективной.

6. ОБЗОР РАБОТ

Концепция автоматного программирования [19, 14] разработана Анатолем Шалыто, в т.ч. в интеграции с объектно-ориентированным программированием. Автоматная программа определяется в виде классического конечного автомата. Используются графическое и текстовое представления программы. Управляющие состояния являются значениями управляющей переменной, которая фактически является частью состояния программы. При реализации автоматной программы применяется switch-технология [2]. Применение объектно-ориентированной технологии позволяет «уплотнить» состояние автоматной программы, сократить ее объем и локализовать часть связей программы внутри классов. Однако спорным являются предлагаемые решения по упрятыванию управляющих состояний и самой структуры автомата внутри классов, оставляя в интерфейсе лишь объекты взаимодействия с внешним окружением.

Термин «автоматное программирование» и его аналог «automata-based programming» используются только в России. Тем не менее, автоматные методы программирования заложены во многих языках, таких как UML, SDL [17], Дракон [10], Рефлекс [11], TLA+ [15], Event-B [16], а также LD, FBD и SFC, определяемых стандартом IEC 61131-3 для программируемых логических контроллеров. Все языки позиционируются как универсальные, однако их эффективная применимость ограничивается классом программ-процессов. Большинство этих языков являются графическими.

Иные подходы применяются в объектно-ориентированных средах программирования. В языках Java, C#, Scala [18] и др. средства конструирования программ-процессов представлены не конструкциями языка программирования, а библиотечными классами. Их эволюция определила появление промышленных платформ, таких как BEA Oracle, Apache Geronimo, jBoss и Microsoft Windows Workflow Foundation (WF) [22]. Платформа WF является частью Microsoft .NET, определяя метасреду для конструирования, исполнения, визуализации и отладки программ-процессов в виде потоков работ[6] (workflows). Поток работ есть иерархически конструируемое действие (activity), компонентами которого являются другие действия. Стандартные действия определяются библиотечными классами Assign, Sequence, If, While, TryCatch, StateMachine, Send, Receive и др. [22, 23]. Поток работ – не объектный код, а объект среды исполнения .NET. Потери эффективности при исполнении компенсируются высокой степенью интеграции с другими сервисами .NET.

В любом языке, ориентированном на разработку программ-процессов, наряду с конструкциями для определения процессов имеется базисная часть, определяющая построение обычных программ-функций. Если базисом такого языка является язык типа C++, то сложность программ-процессов усугубляется сложностью императивного программирования. Это одна из причин появления языков Erlang [20], Rust и др. с функциональным языком в качестве базиса, ориентированных на быструю разработку надежных программ для телекоммуникационных, клиент-серверных, игровых, мобильных, баз данных, облачных и других видов приложений. Язык Erlang [20] предлагает средства построения легковесных процессов, взаимодействующих лишь посредством сообщений и не требующих синхронизации, в соответствии с концепцией акторов [13]. Состояние легковесного процесса определяется параметрами сообщений, а не переменными, поскольку в функциональном языке нет оператора присваивания.

Программу (9) моделирования работы часов с будильником можно перестроить в стиле языка Erlang. Заменим программу (9) тремя параллельными процессами Off(), Set() и On(), переходящими в активное состояние с помощью сообщений of, set и on, соответственно.

[6] Используются также термины «бизнес-процесс» и «рабочий процесс».

```
process Off() {  receive off { Часы_без_будильника() } }
process Часы_без_будильника() {
        receive H        { send inc_h;   Часы_без_будильника() }
        else receive M  { send inc_m;  Часы_без_будильника() }
        else receive A   {  send set }
}
```

Процессы Set() и On() определяются аналогичным образом. Сообщения inc_h, inc_m, bell_on и другие обрабатываются другим параллельным процессом. Его параметрами являются значения времени часов и будильника. Несмотря на большое число сообщений, эквивалентная программа на языке Erlang будет работать достаточно эффективно.

В 2009г. архитектура легковесных процессов языка Erlang была реализована на более высоком уровне в виде библиотеки Akka [9] для языков Scala [18] и Java. Процесс в Akka может иметь состояние и его можно определить в виде конечного автомата. Пакет Akka отметился большим числом приложений; его популярность стремительно растет. Следует учитывать, что область применения Akka ограничена. Его нельзя использовать для протоколов, где недопустимы потери эффективности. Например, лучшая реализация программы Передатчика для стандарта AAL-2 через пакет Akka будет проигрывать программе на рис. 4, примерно как программа (10) проигрывает программе (9), поскольку управляющие состояния конечных автоматов кодируются в Akka значениями параметров сообщений.

ЗАКЛЮЧЕНИЕ

Понимание программ – ключевая проблема в программировании. Эргономические методы, применяемые в графическом языке Дракон [10], существенно улучшают восприятие программы, однако не могут уменьшить ее сложность. Применение методов предикатного программирования: аппарата гиперфункций, использования рекурсивных программ вместо циклов, алгебраических типов вместо указателей и массивов существенно снижает сложность автоматных программ. Предложенные методы иллюстрируются на программе сложного протокола AAL-2.

Большинство языков автоматного программирования являются графическими. Одна из причин этого в том, что они, преодолевая ограничения структурного программирования, обладают большей выразительностью. Тем не менее, гиперграфовая структура автоматной программы, являющаяся продолжением аппарата гиперфункций, имеет более высокую степень гибкости и возможностей адекватной декомпозиции программы по сравнению с графическим представлением языка SDL [17]. С учетом новых реалий текстовое представление программы, отвечающее эргономическим требованиям, может стать основным, а графическое – вспомогательным.

Дальнейшая задача разработки технологии автоматного программирования – специализация технологии для разных подклассов программ-процессов. Сопутствующей задачей является построение классификации программ внутри класса программ-процессов. Предстоит построить иерархию подклассов и определить точные границы между ними. Большую часть класса программ-процессов составляют реактивные системы. Необходимо идентифицировать оставшуюся часть класса программ-процессов, куда входят программы, определяемые недетерминированными и вероятностными автоматами. Для класса реактивных систем следует построить внутреннюю классификацию, в частности, определить гибридные и вероятностные реактивные системы, а также временные автоматы внутри гибридных систем. В рамках работ по формальной верификации программ разработано большое число разных моделей программ. При построении классификации их надо существенно трансформировать с ориентацией на технологию программирования.

Автор благодарен А.А. Шалыто за его работы по автоматному программированию и сайт в Википедии, стимулировавшие исследования автора.

Работа выполнена при поддержке РФФИ, грант № 12-01-00686.

Список литературы

1. ITU-T Recommendation I.363.2 (11/2000): '*B-ISDN ATM Adaptation Layer Specification*: *Type 2 AAL*'. – http://men.axenet.ru/itu/ORIGINAL/I/T-REC-I.363.2-200011-I!!PDF-E.pdf
2. Шалыто А. А. SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб: Наука, 1998. http://is.ifmo.ru/books/switch/1
3. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования Р. – Новосибирск, 2010. – 42с. – (Препр. / ИСИ СО РАН; N 153).
4. Шелехов В.И. Разработка программы построения дерева суффиксов в технологии предикатного программирования. — Новосибирск, 2004. — 52с. — (Препр. / ИСИ СО РАН; N 115).
5. Shelekhov V. I. 2011. Verification and Synthesis of Addition Programs under the Rules of Correctness of Statements. *Automatic Control and Computer Sciences*. Vol. 45, No. 7, 421–427.
6. Шелехов В.И. Верификация и синтез эффективных программ стандартных функций в технологии предикатного программирования // Программная инженерия, 2011, № 2. — С. 14-21.
7. В.А. Вшивков, Т.В. Маркелова, В.И. Шелехов. Об алгоритмах сортировки в методе частиц в ячейках // Научный вестник НГТУ, Т.4(33), с. 79-94, 2008.
8. Шелехов В.И. Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования. – Новосибирск, 2012. – 30с. – (Препр. / ИСИ СО РАН. N 164).
9. What is Akka? – http://doc.akka.io/docs/akka/snapshot/intro/what-is-akka.html
10. Пароннджанов В. Д. Язык ДРАКОН. Краткое описание. – М., 2009. – 124 с. – http://drakon.su/_media/biblioteka/drakondescription.pdf
11. Зюбин В. Е. Программирование информационно-управляющих систем на основе конечных автоматов: Учеб.-метод. пособие. – НГУ, Новосибирск, 2006. 96 с.
12. Cooke D. E., Rushton J. N. Taking Parnas's Principles to the Next Level: Declarative Language Design // Computer, Vol. 42, no. 9. – 2009. – P.56-63.
13. Hewitt C., Bisshop P. and Steiger R. A Universal Modular Actor Formalism for Artificial Intelligence // 3rd 3rd International Joint Conference on Artificial Intelligence, Stanford, CA, 1973, pp. 235-245.
14. Автоматное программирование. http://ru.wikipedia.org/wiki/Автоматное_программирование
15. Lamport L. Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers / Addison-Wesley. – 1999.
16. Abrial J.-R. Modelling in Event-B: System and Software Engineering / Cambridge Univ. Press, 2010.
17. Specification and description language (SDL). *ITU-T Recommendation Z.100* (03/93). – http://www.itu.int/ITU-T/studygroups/com17/languages/Z100.pdf
18. The Scala programming language. http://www.scala-lang.org/
19. Поликарпова Н.И., Шалыто А.А. Автоматное программирование / СПб.: Питер. 2009, 176с. http://is.ifmo.ru/books/_book.pdf
20. Larson J. Erlang for Concurrent Programming // ACM Queue. – 2008. – № 5. – P. 18-23.
21. Шелехов В.И. Предикатное программирование. Учебное пособие. – НГУ, Новосибирск, 2009. – 109с.
22. Shukla D., Schmidt B. Essential Windows Workflow Foundation / Addison Wesley Professional. – 2006.
23. Windows Workflow Foundation // Microsoft Developer Network. http://msdn.microsoft.com/en-us/library/dd489441.aspx

Vladimir I. Shelekhov

Automata-based software engineering: the language and development methods

The language and methods of automata-based software engineering integrated with object-oriented and predicate software engineering methods are developed. A program is constructed as a finite state automation presented in the form of control state hypergraph. Automata-based methods are illustrated on the programs of alarm clock modeling and ATM adaptation layer (type 2) protocol.

Keywords: *program comprehension, automata-based programming, predicate programming, ATM Adaptation Layer*