

# Оптимизация автоматных программ методом трансформации требований

Шелехов В.И.

Институт Систем Информатики им. А.П. Ершова  
г. Новосибирск  
vshel@iis.nsk.su

Технология автоматного программирования ориентирована на разработку простых, надежных и эффективных программ для класса реактивных систем. Автоматная программа реализует конечный автомат в виде гиперграфа управляющих состояний.

В качестве языка спецификаций автоматных программ предлагается язык продукций, применяемый для описания сценариев использования (use case) – одного из видов функциональных требований. Этот простой язык с высокой степенью декларативности. Спецификация программы компактна и легко транслируется в эффективную программу.

Построение эффективных автоматных программ реализуется применением набора оптимизирующих трансформаций программы, определяемой в виде набора правил (продукций). Данный метод иллюстрируется на примере сложного протокола передачи данных ATM Adaptation Layer уровня Type 2 AAL.

**Ключевые слова:** понимание программ; автоматное программирование; определение требований; продукционная система искусственного интеллекта; уровень адаптации ATM.

## I. ВВЕДЕНИЕ<sup>1</sup>

Существуют следующие классы программ: программы-функции, программы-процессы (реактивные системы), трансляторы, операционные системы и другие [3]. Первые два класса покрывают более 90% всех программ. Технология построения надежных и эффективных программ-функций разработана в рамках исследований по предикатному программированию [4-8]. Для класса программ-процессов предложена технология автоматного программирования [3].

Автоматная программа определяется в виде конечного автомата и состоит из нескольких сегментов кода. Вершина автомата – управляющее состояние программы. Ориентированная гипердуга автомата соответствует некоторому сегменту кода и связывает одну вершину с одной или несколькими другими вершинами. Сегмент является либо программой-функцией, либо

программой-процессом, декомпозиция которого представлена другим автоматом. Исполнение сегмента завершается оператором перехода на начало другого сегмента. Взаимодействие автоматной программы с внешним окружением реализуется через прием и посылку сообщений. Состояние автоматной программы определяется значениями набора переменных, модифицируемых в программе.

Определение требований – первый этап конструирования автоматной программы. Требования – совокупность утверждений относительно свойств разрабатываемой программы. Одним из видов требований являются функциональные требования, определяющие поведение программы. Их наиболее популярной формой являются сценарии использования (use case) [12]. В нашем подходе сценарии использования представлены в виде правил на языке продукций [19], обычно применяемом для систем искусственного интеллекта. Это простой язык с высокой степенью декларативности. Спецификация автоматной программы в виде набора правил компактна и легко транслируется в автоматную программу, что позволяет использовать язык спецификации требований как язык автоматного программирования.

Автоматное программирование универсально. Любая программа-функция может быть запрограммирована в виде автоматной программы, которая, однако, будет значительно сложнее аналогичной императивной (или предикатной) программы, построенной обычными средствами. Поскольку сегменты кода автоматной программы конструируются из программ-функций, необходима адекватная интеграция разных стилей программирования.

Сложность автоматной программы экспоненциально зависит от числа переменных состояния программы. Для упрощения программы применяется методы объектно-ориентированного программирования, позволяющие спрятать внутри классов часть переменных состояния и связей между ними. Другим эффективным средством является использование объектов алгебраических типов, списков и деревьев, вместо массивов и указателей. Полезен также механизм гиперграфовой декомпозиции программы.

<sup>1</sup> Работа выполнена при поддержке РФФИ, грант № 12-01-00686.

Типовые методы разработки автоматных программ представлены в работе [27] на нескольких примерах. Существуют автоматные программы, например, протоколы, для которых потери эффективности недопустимы. Такие программы обычно сложны. Их построение является трудной задачей. В настоящей работе предлагается метод построения сложных автоматных программ применением трансформаций, реализующих процесс последовательного улучшения программы, начиная с простой неэффективной программы, представленной в виде набора требований. Применяются трансформации: эквивалентные замены, специализация и др. Их описание дано в разд. III.E.

В разд. II настоящей статьи представлен обзор смежных работ. Базис автоматного программирования, включающий понятие автоматной программы, описание классов программ, парадигму предикатного программирования, язык требований и методы оптимизации автоматных программ, определяется в разд. III. В разд. IV методы построения и оптимизации автоматных программ иллюстрируется на примере сложного протокола передачи данных ATM Adaptation Layer уровня Type 2 AAL [1]. В заключении фиксируются некоторые особенности построения сложных автоматных программ на базе требований.

## II. ОБЗОР РАБОТ

В мировой практике технология определения требований разрабатывается и применяется в основном для больших интернетовских информационных систем и систем телекоммуникации, а также в системной инженерии (системотехнике). Технология спецификации требований отражена в стандарте [2]. В нашем подходе определение требований рассматривается для всего класса реактивных систем.

Инженерия требований имеет длительную почти сорокалетнюю историю, в т.ч. и в нашей стране, в основном на предприятиях аэрокосмического комплекса. К сожалению, исследования в этом направлении велись независимо и слабо интегрированы с мировым опытом, что обнаруживается и по используемой терминологии.

Языками спецификации требований являются: естественный язык, английский (80% случаев), формализованное подмножество естественного языка с использованием аппарата онтологий (15%) и формальный язык (5%) [10]. Популярной проблематикой является трансляция требований с естественного языка на формальные языки. Формальными языками требований являются: RSML [11], Statechart [15] SDL[17], UML, ALBERT [16] и др. Большинство из них являются графическими. В работе [18] выявлены существенные недостатки языка UML при его использовании для спецификации требований производственных систем. Формальными языками спецификации требований являются также общеизвестные универсальные языки спецификаций: VDM, Z, B и др. Семейство темпоральных языков спецификаций также используется для спецификации требований программных систем, в

частности, систем реального времени [28]; наиболее популярным для спецификации требований является язык LTL, см. например [9]. Язык определения требований, предложенный в настоящей работе, существенно отличается от перечисленных языков. Наиболее близок к нему разработанный в целях тестирования язык спецификаций реактивных систем [13].

Продукционные системы искусственного интеллекта [19] реализуются набором правил вида <условие> → <действие>. Правила именно такого вида используются в настоящей работе для определения требований. Ранее язык продукционных правил не применялся для определения требований программных систем. Исключением является работа [20], где язык продукций используется для разработки гибридных систем, однако без осознания того, что продукции являются требованиями к разрабатываемой системе. Другой разновидностью продукционных правил являются охраняемые действия (guarded actions). Языками охраняемых действий являются: язык универсального внутреннего представления для нескольких разнообразных языков спецификаций в целях верификации, моделирования и синтеза [30], а также язык описания аппаратуры BlueSpec [29].

Метод трансформации требований используется в процессе построения требований от наиболее общих к детальным [14]. Он применяется лишь в системной инженерии. Трансформации делятся на два класса: сверху вниз (добавление новых элементов) и снизу вверх (связывание существующих элементов). Определены следующие виды трансформаций: декомпозиция, специализация, уточнение целей (условий), агрегация. Других работ по трансформации требований на формальных языках не найдено. Трансформации неформальных требований на естественном языке и трансформации требований в дизайны программ не имеют ничего общего с трансформациями, используемыми в настоящей работе.

## III. БАЗИС АВТОМАТНОГО ПРОГРАММИРОВАНИЯ

### A. Понятие автоматной программы

Автоматная программа определяется в виде конечного автомата и состоит из нескольких сегментов кода. Вершина автомата соответствует некоторому управляющему состоянию. Ориентированная гипердуга автомата соответствует некоторому сегменту кода и связывает одну вершину с одной или несколькими вершинами. В качестве примера автоматной программы рассмотрим модуль операционной системы, реализующий следующий сценарий работы с пользователем, показанный на рис.1.

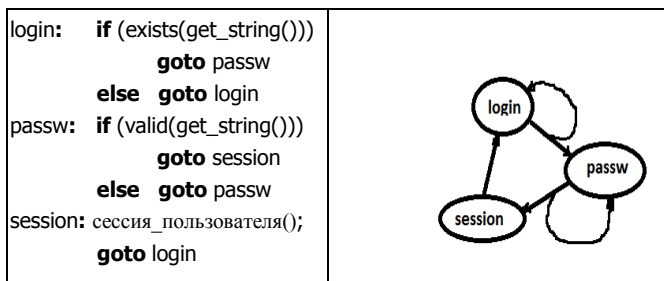


Рис. 1 – Схема работы ОС с пользователем: программа и ее автомат

В управляющем состоянии **login** операционная система запрашивает имя пользователя. Если полученное от пользователя имя существует в системе, она 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**.

Взаимодействие с *внешним окружением* автоматной программы реализуется через прием и посылку *сообщений*, а также через *разделяемые переменные*, доступные в данной программе и 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.

### В. Классы программ

**Класс не взаимодействующих 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 не взаимодействующих программ. Программа обязана всегда завершаться, поскольку бесконечно работающая и не взаимодействующая 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.

#### **Класс программ-процессов (автоматных программ).**

Любая программа-процесс является *реактивной системой*, реализующей взаимодействие с внешним окружением программы и реагирующей на определенный набор событий (сообщений) в окружении программы. Определим структуру класса программ-процессов, т.е. класса реактивных систем.

В общем случае программа-процесс определяется в виде композиции нескольких автоматных программ, исполняемых параллельно и взаимодействующих между собой через сообщения и разделяемые переменные. Каждая из параллельно исполняемых программ определяется независимым автоматом.

Подклассом реактивных систем являются *гибридные системы*, соединяющие дискретное и непрерывное поведение. Часть переменных состояния гибридной системы соответствует непрерывным параметрам (типа **real**), изменение которых реализуется независимо от программы гибридной системы по определенным законам, обычно формулируемым в виде дифференциальных уравнений. Важнейшими подклассами гибридных систем являются контроллеры систем управления и временные автоматы.

*Система управления* реализует взаимодействие с *объектом управления* для поддержания его функционирования в соответствии с поставленной целью. Системы управления используются в аэрокосмической отрасли, энергетике, медицине, массовом транспорте и др. отраслях. На каждом шаге вычислительного цикла *контроллер системы управления* получает входную информацию из окружения и обрабатывает ее. Результаты вычисления используются для передачи управляющего сигнала для воздействия на объект управления. Типовая структура контроллера системы управления определена в работе [21].

*Временной автомат* реализует функционирование процесса, используя показания времени. Пересчет времени проводится вне автоматной программы (временного автомата). Рассматриваются различные модели автоматов с дискретным и непрерывным временем [22]. Временной автомат является *системой реального времени*, если взаимодействие с окружением должно удовлетворять временным ограничениям. В системах с жестким реальным временем непредоставление результатов вычислений к определенному сроку является фатальной ошибкой. Большинство систем управления являются встроенными системами.

Автоматная программа является *детерминированной*, если из каждой вершины автомата (управляющего состояния) исходит не более одной гипердуги. Для *недетерминированного автомата* допускается несколько гипердуг (сегментов кода), исходящих из одного управляющего состояния. При исполнении программы из данного управляющего состояния недетерминировано выбирается один из сегментов кода. Недетерминированный автомат становится *вероятностным*, если для каждой гипердуги определена вероятность ее выбора. Отметим, что недетерминизм неявно реализуется для параллельной композиции автоматных программ. Параллельная композиция может быть представлена эквивалентной интерливинговой разверткой, являющейся недетерминированной последовательной программой [23].

### С. Предикатное программирование

Предикатная программа относится к классу программ-функций и является предикатом в форме вычислимого оператора, реализующим логику решения задачи. Язык предикатного программирования  $P$  [4] 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 предусловия и постусловия 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. Вместо циклов используются рекурсивные функции, а вместо массивов и указателей – объекты алгебраических типов (списки и деревья).

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 и деревьев) с помощью массивов и указателей.

Эффективность программы также 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 последующих 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 [24]) 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, совершаемых программистом вручную.

**Гиперфункции.** Рассмотрим определение предиката следующего вида:

```
pred A(x: y, z, c)
pre P(x)
{ ... }
post c = C(x) & (C(x)  $\Rightarrow$  S(x, y)) & ( $\neg$ C(x)  $\Rightarrow$  R(x, z));
```

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

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

В теле гиперфункции каждое присваивание  $c = \mathbf{true}$  заменено оператором перехода #1, а  $c = \mathbf{false}$  – на #2.

Гиперфункция  $A$  имеет две ветви результатов: первая ветвь включает набор переменных  $y$ , вторая ветвь –  $z$ . Метки 1 и 2 – дополнительные параметры, определяющие 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. Утверждения после “**post** 1” и “**post** 2” есть постусловия для первой и второй ветвей, соответственно.

Использование гиперфункций делает программу короче, быстрее и проще для понимания [8, 25, 26]. Отметим, что модель программ-процессов в форме гиперграфа является естественным продолжением аппарата гиперфункций.

### D. Язык требований

В качестве языка спецификаций программ-процессов предлагается язык продукций [19], применяемый для описания сценариев использования (use case) [12] – одного из видов функциональных требований. Это простой язык с высокой степенью декларативности, характерной для языков логического программирования. Он позволяет писать компактные спецификации программ-процессов. Спецификация автоматной

программы в виде набора правил легко транслируется в эффективную автоматную программу, что позволяет использовать язык спецификации требований как язык автоматного программирования.

*Требование* определяет один из вариантов функционирования автоматной программы и имеет следующую структуру:

$$\langle \text{условие}_1 \rangle, \langle \text{условие}_2 \rangle, \dots, \langle \text{условие}_n \rangle \rightarrow \langle \text{действие}_1 \rangle, \dots, \langle \text{действие}_m \rangle$$

*Условиями* являются: логические выражения, управляющие состояния, получаемые сообщения. *Действиями* являются: простые операторы, вызовы программ, посылаемые сообщения и итоговые управляющие состояния. Требование является спецификацией некоторого сегмента кода или его части. Семантика требования следующая: если в данный момент времени истинны все условия в левой части требования, то последовательно исполняется набор действий в правой части. Формальная семантика строится на базе темпоральной логики: условия определены для текущего управляющего состояния, а результаты действий – для следующего.

Управляющее состояние в качестве *<условия>* означает, что исполнение программы находится в данном управляющем состоянии. Оно должно быть первым в списке условий, и может быть опущено лишь в случае, когда автоматная программа имеет единственное управляющее состояние. Управляющее состояние в качестве *<действия>* – это следующее управляющее состояние, с которого продолжится исполнение программы после завершения исполнения данного требования. Оно должно быть последним в списке действий.

Неблокированный прием сообщения реализуется конструкцией *<имя сообщения>(<параметры>)*. Ее значение – **true**, если из окружения получено сообщение с указанным именем. Оператор

**receive** *<имя сообщения>(<параметры>)*

реализует прием сообщения с ожиданием появления сообщения из окружения. Отметим, что конструкция вида **M: if** (*<сообщение>*) *<оператор>* **else** *#M* эквивалентна

**receive** *<сообщение>; <оператор>*.

Оператор *#M* есть оператор **goto** *M*. Оператор **send** *m(e)* посылает сообщение *m* с параметрами – значениями набора выражений *e*.

В качестве примера рассмотрим требования для модуля, реализующего сценарий работы ОС с пользователем на рис.1.

Содержательное описание представлено в разд. III.A.

### Окружение.

**message** Get(**string** str);

// получение строки от пользователя

Управляющие состояния: login, passw, session;

### Требования.

login, Get(str) → User(str : #login : #passw);

passw, Get(str) → Password(str : #passw : #session);

session → UserSession(), login.

Здесь User и Password – гиперфункции с двумя ветвями без результирующих переменных.

Тип **time** используется для переменных и констант, значениями которых являются показания времени. Оператор **set t**, эквивалентный оператору **time t = 0**, реализует установку таймера, переменной *t*. Ее изменение производится непрерывно некоторым механизмом, не зависящим от автоматной программы. Оператор **delay T** реализует задержку исполнения программы на время *T*; по истечению этого времени программа продолжит работу со следующего оператора.

### *E. Методы оптимизации автоматных программ*

В типичном случае при построении автоматных программ оптимизация не требуется. Типовые методы разработки автоматных программ без оптимизации представлены в работе [27] на нескольких примерах. Существуют автоматные программы, например, протоколы, для которых потери эффективности недопустимы. **Для построения сложных автоматных программ применяются оптимизирующие трансформации, реализующие процесс последовательного улучшения программы, начиная с простой неэффективной программы.** Некоторые особенности оптимизации автоматных программ описываются в данном подразделе.

Оператор суперпозиции  $V(x; y); C(y; z)$ , условный оператор **if** ( $e(x)$ )  $V(x; y)$  **else**  $C(x; y)$  и параллельный оператор  $V(x; y) || C(x; z)$  определяют основные формы конструирования алгоритмов в предикатном и императивном программировании. Эти же формы конструирования применимы и в автоматном программировании с преломлением на структуру автоматной программы.

Аналогом условного оператора является фрагмент автоматной программы:

**H1: inv**  $e(x); V(x; y) \#H3$   
**H2: inv**  $\neg e(x); C(x; y) \#H3$   
**H3:**

Инвариант **inv**  $e(x)$  ассоциирован с управляющим состоянием *H1* и должен быть истинным, когда исполняемая программа приходит в состояние *H1*. Инвариант не вычисляется и должен быть истинным априори. Прежде всего, инвариант – средство конструирования программы, хотя он может быть использован и для верификации программы.

Следует особо подчеркнуть, что инварианты автоматной программы принципиально отличаются от инвариантов циклов и инвариантов классов императивной программы. В типичном случае, когда не требуется оптимизации автоматной программы, управляющее состояние не содержит инварианта; иначе говоря, инвариант тождественно истинен. Циклы в автоматной

программе имеют другую природу по сравнению с циклами императивной программы. Не рекомендуется смешивать разные стили программирования, в частности, использовать циклы типа **while** для конструирования автоматной программы.

Специализация сегмента кода на базе некоторого условия  $e(x)$  является популярной оптимизирующей трансформацией. Сегмент кода  $H: S$  заменяется на:

$H1: \text{inv } e(x); S$   
 $H2: \text{inv } \neg e(x); S$

Далее реализуются упрощающие преобразования двух разных вхождений  $S$  на базе условий  $e(x)$  и  $\neg e(x)$ .

Композиция в виде суперпозиции  $B(s: s'); C$ , где  $s$  – переменные состояния автоматной программы,  $B(s: s')$  – вызов программы-функции и  $C$  – программа-процесс, представляется следующим фрагментом автоматной программы:

$H: B(s: s') \# K$   
 $K: \text{inv } e(s); C$

Вызов  $B(s: s')$  можно было бы объединить в одном сегменте с процессом  $C$ . Его выделение в отдельный сегмент целесообразно в случае последующей специализации процесса  $C$ .

#### IV. ПЕРЕДАТЧИК В ПРОТОКОЛЕ AAL-2

Метод трансформации требований для построения автоматной программы иллюстрируется на программе Передатчик в протоколе AAL-2, описанной на языке спецификаций SDL [17] в разд. 10.1 стандарта [1]. Сначала строится простейшая формализация требований, которая далее последовательно трансформируется в целях получения эффективной программы.

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.

**Содержательное описание.** Передатчик получает пакеты данных с уровня SSCS и плотно упаковывает их в последовательности блоков данных, посылаемых на уровень ATM. Очередной пакет поступает через сообщение CPSpacket(pd), где pd – данные пакета переменной длины. Передатчик строит заголовок пакета rh (packet header) длиной 3 октета<sup>2</sup>. Пакет, состоящий из rh и pd, упаковывается в один или несколько блоков данных (protocol data units). Блок данных состоит из начального поля STF (start field) длиной в 1 октет и основной части (payload) длиной 47 октетов. Пакет, не вместившийся в очередном блоке, продолжается в следующем; см. рис. 2.

<sup>2</sup> Октет состоит из 8 битов; термин «байт» не используется, т.к. имеет иной смысл.

Начальное поле STF блока данных содержит длину в октетах перенесенной части пакета из предыдущего блока. Очередной построенный блок передается на уровень ATM посылкой сообщения ATM\_data("блок").

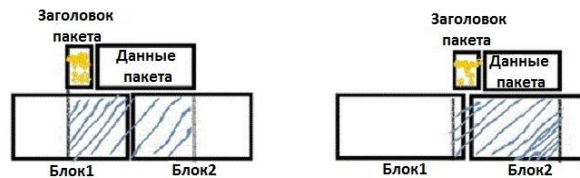


Рис. 2 – Схема переноса пакета:  
 а – перенос данных; б – перенос заголовка

Очередной заполненный блок может быть отправлен на уровень ATM только после получения запроса на посылку – сообщения SEND\_request(). Построение очередного блока управляется таймером: по истечении времени Twait построение очередного блока завершается заполнением нулями оставшейся части блока.

Определим непосредственную формализацию представленных требований. Введем буфер packs для хранения пакетов, полученных через сообщения CPSpacket и еще не отправленных на уровень ATM. Когда длина буфера packs достигает или превышает 47 (длины информационной части блока), очередной сформированный блок посылается на уровень ATM сообщением ATM\_data при дополнительном условии – получении сообщения SEND\_request.

#### Окружение:

message CPSpacket(pd), SEND\_request, ATM\_data(block).

Пакет и блок данных имеют тип DATA – список октетов:

type OCTET = byte;  
 type DATA = list(OCTET);

Использование списков вместо массивов упрощает программу. Константа nil обозначает пустой список, len(x) – длину списка x, x + y – конкатенацию списков x и y. При трансляции списки кодируются вырезками массивов [3, 4].

#### Состояние:

DATA packs = nil, OCTET stf = ConstructSTF(0), time t.

Локальные переменные: DATA pd, block.

#### Локальные программы.

pred extract\_block(DATA packs: DATA block, packs')  
 pre len(packs) ≥ 47  
 post packs = block + packs' & len(block) = 47;

Программа extract\_block выделяет очередной блок block из буфера packs при условии, что его длина не меньше 47. Результат программы packs' – оставшаяся часть буфера.

Программа ConstructCPS\_PacketHeader(pd) строит заголовок rh для пакета pd. Программа ConstructSTF(x)

строит начальное поле STF для следующего блока, где  $x$  – длина перенесенной части предыдущего пакета. Программа  $fill(packs)$  дополняет буфер  $packs$  нулями до длины в 47 октетов.

**Требования:** (1)

```
len(packs) < 47, CPSpacket(pd) →
  set t,
  packs' = packs + ConstructCPS_PacketHeader(pd) + pd;
len(packs) ≥ 47, SEND_request() →
  extract_block(packs: block, packs'),
  ATM_data(stf + block),
  if(packs'≠nil) set t,
  stf' = ConstructSTF(min(len(packs'), 47));
0 < len(packs) < 47, t > Twait, SEND_request() →
  ATM_data(stf + fill(packs)),
  packs' = nil,
  stf' = ConstructSTF(0).
```

Возможна эффективная реализация требований: буфер  $packs$  «плавает» внутри достаточно длинного массива, а при приближении к его концу переписывается в начало.

Наша цель – используя требования (1) построить эффективную программу, соответствующую релизам [1, 3], в которых вместо  $packs$  используется буфер  $buf$ , вмещающий не более 47 октетов. Буфер  $buf$ , заголовок очередного пакета  $ph$  и сам пакет  $pd$  в сумме составляют буфер  $packs$ .

**Трансформация 1.** Удалим оператор присваивания  $packs'$  в первом правиле. Заменяем  $packs$  на  $buf + ph + pd$  в требованиях (1). Соответствующая модификация требований представлена ниже.

**Состояние:**

DATA  $buf = nil, ph = nil, pd = nil, time t,$   
 ОСТЕТ  $stf = ConstructSTF(0)$

**Локальные программы.** Меняется программа  $extract\_block$ , реализующая перепись на буфер  $buf$  заголовок  $ph$  и пакета  $pd$ , полностью или частично.

```
pred extract_block(DATA buf, ph, pd: DATA buf', ph', pd')
  pre buf + ph + pd ≥ 47
  post buf + ph + pd = buf' + ph' + pd' & len(buf')=47;
```

В роли очередного блока выступает полностью заполненный буфер  $buf'$ . Другие результаты – модифицированные заголовок  $ph'$  и пакет  $pd'$ .

**Требования:** (2)

```
len(buf + ph + pd) < 47, CPSpacket(pd') →
  set t, buf' = buf + ph + pd,
  ph' = ConstructCPS_PacketHeader(pd');
0 < len(buf + ph + pd) < 47, t > Twait, SEND_request() →
  ATM_data(stf + fill(buf + ph + pd)),
  buf' = nil, ph' = nil, pd' = nil,
  stf' = ConstructSTF(0);
len(buf + ph + pd) ≥ 47, SEND_request() →
  extract_block(buf, ph, pd: buf', ph', pd'),
  ATM_data(stf + buf'),
  if (ph' + pd' ≠ nil) set t,
  buf'' = nil;
  stf' = ConstructSTF(min(len(ph' + pd'), 47)).
```

Для упрощения требований (2) предварительно следует провести перепись заголовка  $ph$  и пакета  $pd$  на буфер  $buf$ , как это реализуется программой  $extract\_block$ . В связи с этим новая версия автоматной программы конструируется в форме суперпозиции (см. разд. III.E).

Единственное управляющее состояние в требованиях (2) обозначим через  $H$ . Ведем новое управляющее состояние  $K$  с инвариантом:

$(len(buf+ph+pd) < 47 \Rightarrow ph = nil \ \& \ pd = nil) \ \&$   
 $(len(buf+ph+pd) \ge 47 \Rightarrow len(buf) = 47).$

Инвариант истинен после переписи  $ph$  и  $pd$  на буфер  $buf$ .

В управляющем состоянии  $H$  реализуется перепись  $ph$  и  $pd$  на буфер  $buf$  с переходом на управляющее состояние  $K$ , в котором реализуются аналоги правил требований (2).

**Локальные программы.**

```
pred move(DATA buf, x: DATA buf', x')
  pre len(buf) ≤ 47
  post buf' + x' = buf + x &
  (len(buf + x) ≥ 47 ? len(buf')=47 : x'=nil);
```

Программа  $move$  переписывает список  $x$  на буфер  $buf$  Результат  $x'$  – остаток от переписи, возможно пустой. Программа работает и при заполненном буфере с результатом  $x' = x$ .

**Требования:** (3)

$H \rightarrow move(buf, ph: buf', ph'), move(buf', pd: buf'', pd'), K;$  (3.1)

$K, len(buf) < 47, CPSpacket(pd) \rightarrow$   
 $set t, ph = ConstructCPS_PacketHeader(pd), H;$  (3.2)

$K, 0 < len(buf) < 47, t > Twait, SEND_request() \rightarrow$   
 $ATM\_data(stf + fill(buf)), buf' = nil, ph' = nil, pd' = nil,$   
 $stf' = ConstructSTF(0), H;$  (3.3)

$K, len(buf) = 47, SEND_request() \rightarrow$   
 $ATM\_data(stf + buf), if (ph + pd \neq nil) set t,$   
 $buf' = nil; stf' = ConstructSTF(min(len(ph + pd), 47)), H.$  (3.4)

Неэффективность версии (3) в том, что программа  $move$  запускается для пустых  $ph$  и  $pd$ , а также в случае  $len(buf) = 47$ . Для устранения этих недостатков определим программу  $move$  в виде гиперфункции с тремя ветвями:

**hyper** move(DATA buf, x: DATA buf' #1: DATA buf' #2:  
DATA buf', x' #3)

```
pre len(buf) ≤ 47 // общее предусловие
pre 1: len(buf+x) < 47
pre 2: len(buf+x) = 47
pre 3: len(buf+x) > 47
post 1: len(buf') < 47 & buf' = buf + x
post 2: len(buf') = 47 & buf' = buf + x
post 3: len(buf') = 47 & buf + x = buf' + x' & x'≠nil;
```

Для первой и второй ветви список x полностью переписан на буфер buf, причем для второй ветви буфер полностью заполнен. В третьей ветви после переписи остается непустой остаток x'.

**Трансформация 2.** В требовании (3.1) заменим вызовы программы move на эквивалентные вызовы гиперфункций. С этой целью определим выходы для всех трех ветвей в одну точку непосредственно после вызова гиперфункции. Например:

move(buf, ph: buf' #G : buf' #G : buf', ph' #G); G:

**Трансформация 3.** Введем новые управляющие состояния с инвариантами:

```
PART: inv len(buf) < 47 & ~ph & ~pd;
B1: inv len(buf) = 47 & ~ph & ~pd;
    // ph и pd полностью переписаны
B2: inv len(buf) = 47 & ~ph; // ph полностью переписан
B3: inv len(buf) = 47 & ph≠nil; // ph переписан частично
```

Здесь ~ph и ~pd обозначают условия ph=nil и pd=nil. Обнуление переменных ph и pd можно будет опустить, поскольку их значения далее использоваться не будут.

**Трансформация 4.** Проведем специализацию требования (3.4) для управляющих состояний B1, B2 и B3, а также требований (3.2) и (3.3) для управляющего состояния PART.

**Трансформация 5.** Проведем уточнение итоговых управляющих состояний во всех требованиях, в т.ч. определяемых выходами вызовов гиперфункций. С этой целью состояние в конце каждого требования сопоставляется с инвариантами управляющих состояний.

**Требования:** (4)

```
PART, CPSpacket(pd) →
    set t, ph = ConstructCPS_PacketHeader(pd), H;
H → move(buf, ph: buf' #G : buf' #B2: buf', ph' #B3);
G → move(buf, pd: buf' #PART : buf' #B1 : buf', pd' #B2);
PART, buf≠nil, t > Twait, SEND_request() →
    ATM_data(stf + fill(buf)),
    buf' = nil, stf' = ConstructSTF(0), PART;
B1, SEND_request() → ATM_data(stf + buf),
    buf' = nil; stf' = ConstructSTF(0), PART;
B2, SEND_request() → ATM_data(stf + buf); set t,
    buf' = nil; stf' = ConstructSTF(min(len(pd), 47)), G;
B3, SEND_request() → ATM_data(stf + buf), set t,
    buf' = ph; stf' = ConstructSTF(min(len(ph + pd), 47)), G.
```

Требования непосредственно переписываются в программу.

**Программа:**

```
process transfer() {
PART: inv len(buf) < 47 & ~ph & ~pd;
    if (CPSpacket(pd)) {
        set t;
        ph = ConstructCPS_PacketHeader(pd);
        move(buf, ph : buf' #G: buf' #B2 : buf', ph' #B3)
G:    inv len(buf) < 47 & ~ph;
        move(buf, pd : buf #PART: buf' #B1: buf', pd' #B2)
    } else if (buf≠nil & t > Twait & SEND_request()) {
        send ATM_data(stf + fill(buf));
        buf = nil; stf = ConstructSTF(0) #PART
    }
B1:    inv len(buf) = 47 & ~ph & ~pd;
        receive SEND_request();
        send ATM_data(stf + buf); stf' = ConstructSTF(0);
        buf = nil #PART
B2:    inv len(buf) = 47 & ~ph;
        receive SEND_request();
        send ATM_data(stf + buf); set t;
        buf = nil; stf' = ConstructSTF(min(len(pd), 47)) #G
B3:    inv len(buf) = 47 & ph≠nil;
        receive SEND_request();
        send ATM_data(stf + buf); set t;
        buf = ph; stf' = ConstructSTF(min(len(ph + pd), 47));
        #G
    }
}
```

## V. ЗАКЛЮЧЕНИЕ

В настоящей работе рассматривается проблема построения сложных автоматных программ на примере программы Передатчик в протоколе AAL-2. Ее описание в стандарте [1, разд. 10.1] – сложное и громоздкое с плохой эргономикой. Использование аппарата гиперфункций и списков вместо массивов существенно снижает сложность программы Передатчик в нашем релизе [3]. Однако и эта программа сложна, потому что она воссоздает неудачный дизайн стандарта [1], наследуя к тому же дефекты в эффективности.

Построение простой и эффективной программы Передатчик начинается с более простой задачи, представленной требованиями (1), допускающей эффективную реализацию при достаточно длинном буфере. Далее применяется последовательность трансформаций, приводящая к требованиям (4). Построенная по ним эффективная программа (см. в конце разд. IV) значительно проще аналогичных программ в [1, 3].

В принципе можно было бы начать разработку программы с требований (3) и, проведя их специализацию, получить требования (4). Преимущество метода трансформации требований от простых к более сложным заключается в дополнительной возможности нахождения ошибок в требованиях. Каждая новая версия



требований после очередной трансформации подвергается осмыслению и анализу, при этом достаточно часто ошибочные ситуации становятся более явными и заметными. В процессе трансформаций в исходных требованиях (1) было найдено 7 ошибок. Отметим, что при разработке информационных систем обычно допускается много ошибок в требованиях, они трудно ищутся в процессе моделирования и тестирования и дорого обходятся.

Представленный метод построения автоматных программ применением трансформаций может быть использован лишь для сложных программ. Для построения большинства автоматных программ, в частности, для программ в работе [27], трансформации не нужны. В данной работе трансформации лишь частично классифицированы. Формализация и детальная классификация – цель дальнейших исследований. Применение трансформаций в частности планируется при построении программ для серии протоколов взаимного исключения.

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] IEEE Recommended Practice for Software Requirements Specifications. Revision: 29/Dec/11.
- [3] Шелехов В.И. Язык и технология автоматного программирования // «Программная инженерия», №4, 2014. – С. 3-15. <http://persons.iis.nsk.su/files/persons/pages/automatProg.pdf>
- [4] Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования Р. – Новосибирск, 2010. – 42с. – (Препр. / ИСИ СО РАН; N 153). <http://persons.iis.nsk.su/files/persons/pages/plang12.pdf>
- [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] Arcaini P., Gargantini A., Riccobene E. Online Testing of LTL Properties for Java Code // *Hardware and Software: Verification and Testing*, LNCS 8244. – 2013. – P. 95-111.
- [10] Mich L, Franch M, Novi Inverardi P. Market research for requirements analysis using linguistic tools. *Requirements Engineering* 9(1). – 2004. – P. 40–56.
- [11] Leveson N., Heimdahl M., Hildreth H., Damon J. Requirements Specification for Process-Control Systems // *IEEE Transactions on Software Engineering*, 20 (4). – 1994. – P.684-707.
- [12] Cockburn A. *Writing Effective Use Cases* / Addison-Wesley. – 2001. – 270 P.
- [13] Sunha A., Sharad M. Modeling Firmware as Service Functions and Its Application to Test Generation // *Hardware and Software: Verification and Testing*, LNCS 8244. – 2013. – P. 61-77.
- [14] P. Brescani, A. Perini, P. Giorgini, F. Giunchiglia, J. Mylopoulos. Modelling Early Requirements in Tropos: A Transformation-Based Approach // *Agent-Oriented Software Engineering II*, LNCS 2222. – 2002. – P. 151-168.
- [15] Zhang W., Beaubouef T., and Ye H. “Statechart: A Visual Language for Software Requirement Specification // *International Journal of Machine Learning and Computing*. – 2012. – P. 52-61.
- [16] Aoumeur N., Saake G. Operational interpretation of requirements specification language ALBERT using timed rewriting logic // *5th International Workshop on Requirements Engineering: Foundation for Software Quality*. 1999.
- [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] Glinz M. Problems and Deficiencies of UML as a Requirements Specification Language // *10th International Workshop on Software Specification and Design*. – 2000. – P. 11-22.
- [19] Klahr D., Langley P., Neches R. *Production System Models of Learning and Development*. – Cambridge, Mass.: The MIT Press. 1987. – 467 P.
- [20] Шпаков В.М. Формализация и использование знаний о развитии процессов // Тр. 16-й междунар. конф. «Проблемы управления и моделирования в сложных системах». — Самара, Самарский научный центр РАН, 2014. – С. 290-295.
- [21] Тумуров Э.Г., Шелехов В.И. Определение требований к системе управления полетом квадрокоптера // Тр. 16-й междунар. конф. «Проблемы управления и моделирования в сложных системах». – Самара, Самарский научный центр РАН, 2014. – С. 627-633.
- [22] Alur R., Dill D.L. A theory of timed automata // *Theor. Comput. Sci.* – 1994. – P. 183-235.
- [23] Шелехов В.И. Тумуров Э.Г. Логика не взаимодействующих программ и реактивных систем // *Вестник Бурятского Государственного Университета. Секция: математика, информатика*, Вып. 9 / 2012. – Улан-Удэ, 2012. – С. 81-90.
- [24] 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.
- [25] Шелехов В.И. Разработка программы построения дерева суффиксов в технологии предикатного программирования. — Новосибирск, 2004. — 52с. — (Препр. / ИСИ СО РАН; N 115).
- [26] Шелехов В.И. Предикатное программирование. Учебное пособие. – НГУ, Новосибирск, 2009. – 109с.
- [27] Шелехов В.И. Разработка автоматных программ на базе определения требований. – 2014. [http://persons.iis.nsk.su/files/persons/pages/req\\_tech.pdf](http://persons.iis.nsk.su/files/persons/pages/req_tech.pdf)
- [28] Bellini P., Mattolini R., and Nesi P. Temporal logics for real-time system specification // *ACM Comp. Surveys*, 32(1). – 2000. – P.12–42.
- [29] Arvind H. Bluespec: a language for hardware design, simulation, synthesis and verification // *MEMOCODE*. – 2003. – P 249–254.
- [30] Brandt J., Gemünde M., Schneider K., Shukla S.K., Talpin J.-P. Representation of synchronous, asynchronous, and polychronous components by clocked guarded actions // *Design Automation for Embedded Systems*. – 2012. – P. 1 – 35.