Разработка автоматных программ на базе определения требований

В.И. Шелехов

Институт систем информатики им. А.П. Ершова СО РАН
630090, Новосибирск, пр. Лаврентьева, 6, Россия
vshel@iis.nsk.su
тел: +7 (383) 330-27-21, факс: +7 (383) 332-34-94

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. Технология представлена в виде свода золотых правил программирования, определяющих правильный баланс в интеграции автоматного, предикатного и объектно-ориентированного программирования. Технология иллюстрируется на наборе примеров.

Ключевые слова: понимание программ, автоматное программирование, определение требований

1. ВВЕДЕНИЕ

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 [3]. Сегменты кода конструируются из фрагментов, являющихся предикатными программами [4].

Автоматные программы принадлежат классу программ-процессов (реактивных систем), более сложному по сравнению с классом программ-функций. Технология построения надежных и эффективных программ-функций разработана в рамках исследований по предикатному программированию [4–8]. Технология автоматного программирования [3, 22, 23] интегрирована с технологиями предикатного и объектно-ориентированного программирования. Гиперграфовая структура автоматной программы обеспечивает высокую гибкость, выразительность и эффективность программ.

Наряду с операторным языком автоматных программ [3] используется язык спецификаций требований. *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 на этом языке компактна и легко транслируется в автоматную программу, что позволяет использовать его как язык автоматного программирования.

Базис автоматного программирования определяется в разд.3. Дается определение автоматной программы. Описывается структура класса реактивных систем. Дается общее описание парадигмы предикатного программирования. Определяется язык требований, используемый в качестве языка автоматного программирования. Технология автоматного программирования (разд. 4) представлена в виде свода золотых правил программирования, определяющих правильный баланс в интеграции автоматного, предикатного и объектно-ориентированного программирования. Технология иллюстрируется на наборе примеров построения автоматных программ в разд. 5–10.

2. 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. Это, в частности, обнаруживается и по используемой терминологии. В соответствии с тезисом «программирование без программистов» [30], именно специалисты по разработке космических аппаратов, знающие «физику» создаваемых аппаратов, должны разрабатывать управляющие программы космических аппаратов, а не программисты. При внимательном анализе обнаруживается, что здесь речь идет о той части программирования, которая относится к разработке требований. В зарубежных фирмах по разработке информационных систем разработку требований осуществляют специалисты – *инженеры требований*; иногда они составляют половину персонала [10].

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, FSP [40, 41] и др. Семейство темпоральных языков спецификаций также используется для спецификации требований программных систем, в частности, систем реального времени [35]; 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 [24] реализуются набором правил вида <условие> $\rightarrow$ <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 [21], где язык продукций используется в целях разработки гибридных систем, однако без осознания того, что продукционные правила 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 [36], а также язык описания аппаратуры BlueSpec [27].

В событийно-ориентированной парадигме [42, 43] программа представлена в виде цикла, в котором последовательно просматривается определенный набор событий (сообщений). Для всякого события запускается соответствующий обработчик события. Таким образом, программа составляется из набора независимых обработчиков. В частности, графический интерфейс пользователя (GUI), во всех инструментах его реализующих, определяется именно в такой архитектуре. Авторы данной парадигмы признают, что программирование в ней является сложным. Причина сложности в том, что в обработчике события неестественным образом перемешиваются части программ, которые в автоматной программе принадлежат сегментам кода для разных управляющих состояний. Событийно-ориентированный стиль программирования иногда применяется во избежание

параллелизма в программе, нередко используя при этом сопрограммную схему взаимодействия. Отметим, что замена параллельного исполнения сопрограммным является преобразованием, существенно усложняющим программу; см. например [44]. Событийно-ориентированное программирование должно быть ограничено первичной обработкой событий (сообщений) для последующей передачи их автоматной программе, исполняемой параллельно. В частности, графические интерфейсы пользователя должны быть ориентированы лишь для построения разнообразного гибкого внешнего окружения автоматной программы, но не ее самой.

Обзор работ по автоматному программированию представлен также в статье [3].

3. БАЗИС АВТОМАТНОГО ПРОГРАММИРОВАНИЯ

3.1 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 другими вершинами. В качестве примера автоматной программы рассмотрим модуль операционной системы, реализующий следующий сценарий работы с пользователем (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
```

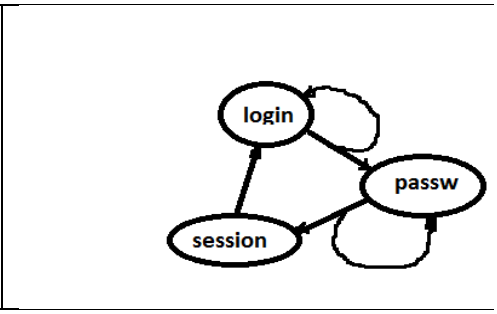

Рис. 1 – Схема работы ОС с пользователем: программа и ее автомат

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. Отметим, что реальная программа взаимодействия ОС с пользователем существенно сложнее; в частности, функция get_string должна поставлять текст в зашифрованном виде.

Состояние автоматной программы определяется значениями набора переменных, модифицируемых в программе, за исключением локальных переменных. 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.

3.2 Класс программ-процессов

Автоматное программирование ориентировано на класс программ-процессов. Его не следует применять для программ-функций, где предпочтительны традиционные средства, а также технология предикатного программирования [4-8].

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 [22].

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 [25]. Временной автомат является *системой реального времени*, если взаимодействие с окружением должно удовлетворять временным ограничениям, что характерно для встроенных систем. 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 [26].

Автоматная программа может быть составлена из частей, принадлежащим разным подклассам реактивных систем. Например, возможно сочетание вероятностных и недетерминированных автоматов в рамках одной программы. Системы реального времени в большинстве случаев являются системами управления. В дополнении к этому автоматная программа может быть частью *распределенной системы*. Отметим также возможность интеграции с объектно-ориентированной технологией, когда состояние автоматной программы реализовано как объект класса.

3.3 Предикатное программирование

Предикатная программа относится к классу программ-функций и является предикатом в форме вычислимого оператора, реализующим логику решения задачи. Язык предикатного программирования Р [4] обладает большей выразительностью в сравнение с языками функционального программирования и по стилю ближе к императивному программированию.

Полная предикатная программа состоит из набора рекурсивных *предикатных программ* на языке Р 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 [5, 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 [5–8]. Отметим, что в функциональном программировании (при общеизвестной ориентации на предельную компактность и декларативность [38]) 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 предикатную программу следующего вида:

pred A(x**:** y, z, c)
pre P(x)
{ … }
post c = C(x) & (C(x) ⇒ S(x, y)) & (¬C(x) ⇒ 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 = **true** и c = **false** – последние исполняемые операторы в теле предиката. Программа A может быть заменена следующей программой в виде *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 = **true** заменено оператором перехода #1, а c = **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 – 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, 39]. Отметим, что модель программ-процессов в форме гиперграфа является естественным продолжением аппарата гиперфункций.

3.4 Язык требований

В качестве языка спецификаций программ-процессов предлагается язык продукций [24], применяемый для описания сценариев использования (use case) [12] – одного из видов функциональных требований. Этот простой язык с высокой степенью декларативности, характерной для языков логического программирования. Он позволяет писать компактные и хорошо понимаемые 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:

<условие$_1$>, <условие$_2$>,…, <условие$_n$> → <действие$_1$>, …, <действие$_m$>;

Условиями являются: управляющие состояния, получаемые сообщения, логические выражения. *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. Далее ограничимся детерминированными программами: для исполнения выбирается первое правило с истинными условиями.

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. 3.1.

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.

Язык предикатного программирования P [4] 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 предикатной программы. При наличии единственного объекта класса доступ к элементу объекта возможен просто через <имя элемента класса>.

С управляющим состоянием может быть ассоциирован инвариант: **inv** <логическое выражение>. Логическое выражение должно быть истинным, когда исполняемая программа приходит в данное управляющее состояние. Инвариант не вычисляется при исполнении программы и должен быть истинным априори. Инвариант повышает понимание программы, его можно также использовать для верификации.

Для требований следующего вида:

s1, <условие$_1$>, … , <условие$_n$> → <действие$_1$>, … , <действие$_m$>, s2

s1 → <действие$_1$>, … , <действие$_m$>, s2

где s1 и s2 – управляющие состояния, иногда будем использовать, соответственно, другие формы записи требований:

s1**:** <условие$_1$>, … , <условие$_n$> → <действие$_1$>, … , <действие$_m$> #s2

s1**:** <действие$_1$>, … , <действие$_m$> #s2

4. ТЕХНОЛОГИЯ АВТОМАТНОГО ПРОГРАММИРОВАНИЯ

Конструирование автоматной программы реализуется следующей последовательностью этапов. Сначала формулируется *постановка задачи* в форме *содержательного описания* набора требований. Содержательное описание должно быть полным, определяя реакцию программы однозначно и непротиворечиво для любой ситуации, возникающей во *внешнем окружении* программы. Содержательное описание должно быть компактным и написано простым понятным языком. *Формализация задачи* начинается с описания элементов внешнего окружения. Специфицируются переменные *состояния* автоматной программы. Определяются связи между переменными. На следующем этапе фиксируются *управляющие состояния*. Некоторые из них снабжаются инвариантами. Построение автоматной программы реализуется в виде набора *требований*. Каждое из них обеспечивает адекватную реакцию на определенное сообщение или событие. Процесс построения программы сопровождается ее верификацией относительно содержательного описания.

Набор методов построения хороших (простых, надежных, эффективных и т.д.) программ представим в виде свода *золотых правил программирования*, определяющих правильный баланс в интеграции автоматного, предикатного и объектно-ориентированного программирования. Собирателем золотых правил в далеких 1960-ых годах был Г.И. Кожухин, один из разработчиков транслятора Альфа [31].

Правило №1 Геннадия Кожухина: *Ошибки в программе надо искать как грибы. Нашел одну, ищи рядом*. Сложность программы неравномерно распределена по программе. Имеются участки повышенной сложности; там вероятнее ошибиться. Один такой участок часто содержит более одной ошибки.

Автоматное программирование универсально. Любая программа-функция может быть запрограммирована в виде автоматной программы, которая, однако, будет значительно сложнее аналогичной предикатной или императивной программы, построенной обычными средствами. Отсюда следует **правило №2**: *не следует использовать автоматное программирование для программ-функций.*

Поскольку сегменты кода автоматной программы конструируются из частей, соответствующих программам-функциям, необходима адекватная интеграция разных стилей программирования. Не следует смешивать разные стили. **Правило №3**: *части сегментов кода, соответствующие программам-функциям, должны быть представлены вызовами программ за исключением случаев, когда часть сегмента сводится к одному простому оператору*. Значительный по размеру фрагмент кода загромождает программу. Вынесение его из программы и оформление независимой подпрограммой улучшает понимание исходной программы. Сопутствующее **правило №4**: *любая подпрограмма должна помещаться в пределах экрана монитора* – одновременная видимость всех ее информационно-управляющих связей упрощает ее анализ. Отметим, что запроцедуривание простых операторов в составе автоматной программы дает обратный эффект – избыточная структуризация введением дополнительного уровня иерархии усложняет программу.

Сложность автоматной программы экспоненциально зависит от числа переменных состояния программы, а также от числа и характера связей между переменными. Для упрощения программы применяется методы объектно-ориентированного программирования, позволяющие спрятать внутри классов часть переменных состояния и связей между ними. **Правило №5**: *замените состояние программы (или его часть) объектом класса, скрывающего часть переменных и связей между ними внутри класса*. Объектно-ориентированная декомпозиция существенно снижает сложность и размер автоматной программы. Однако не всегда. **Правило №6**: *не используйте классов, если нечего скрывать*. Автоматная программа проще не станет. **Правило №7**: *Не следует прятать внутри класса автомат программы, т.е. управляющие состояния и сегменты кода, оставляя в интерфейсе лишь объекты внешнего окружения*. Это худший способ реализации автоматной программы, выворачивающий ее наизнанку.

Модифицируем программу на рис.1 в стиле switch-технологии А. Шалыто [32]:

```
type STATE = enum(login, passw, session);
STATE state = login;
while (true)
switch (state) {
   case login:    if (exists(get_string()))
                      state =  passw
                  else   state = login
   case passw:   if (valid(get_string()))
                      state = session
                  else   state = passw
   case session:  сессия_пользователя();
                   state = login
}
```

В модифицированной программе имеется лишь одно управляющее состояние, соответствующее началу тела цикла **while**, а единственным сегментом кода является тело цикла **while**. Таким образом, управляющие состояния login, passw и session становятся значением переменной состояния state в модифицированной программе. В работе [3, разд. 4, рис.2] детальным сравнением исходной и модифицированной программ показано, что исходная программа проще. **Правило №8**: *не следует переводить управляющие состояния*

в состояние автоматной программы. Следствием является п**равило №9**: *управляющее состояние, используемое явно или неявно в содержательном описании программы должно быть воспроизведено в автоматной программе.*

Инварианты автоматной программы, ассоциированные с управляющими состояниями, 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. **Правило №10**: *не рекомендуется смешивать разные стили программирования, в частности, использовать циклы типа* ***while*** *для конструирования автоматной программы*.

В данной работе представлены типовые методы разработки автоматных программ. В случаях, когда оптимизация автоматной программы требует изменения ее автоматной структуры, следует использовать метод трансформации требований [23].

5. УПРАВЛЕНИЕ ОСВЕЩЕНИЕМ

Пример взят из руководства по системе верификации Uppal [20]. Описывается простой временной автомат для включения и выключения лампочки.

Содержательное описание. Имеется кнопка для включения и выключения лампочки. При нажатии на кнопку лампочка включается. Повторное нажатие на кнопку выключает лампочку. При двойном нажатии (быстрым нажатием на кнопку дважды) лампочка включается и загорается ярче. Нажатие считается *двойным*, если второе нажатие запаздывает по отношению к первому не более чем на 4 условные единицы измерения времени.

Окружение.

```
message press,      // при нажатии кнопки посылается сообщение press
        low_on,     // посылает сигнал для включения лампочки слабым светом
        bright_on, // посылает сигнал для включения лампочки ярким светом
        light_off;  // посылает сигнал для выключения лампочки
```

Управляющие состояния:

- off – лампочка выключена;
- low – лампочка включена и горит слабым светом;
- bright – лампочка включена и горит ярким светом.

Управляющее состояние bright реализуется после двойного нажатия кнопки, low – после одинарного.

Состояние.

```
time y; // хранит время в условных единицах
```

После установки в 0 значение переменной последовательно увеличивается, моделируя процесс изменения времени. Изменение значения y реализуется вне программы.

Представленные ниже требования являются непосредственной формализацией содержательного описания требований.

Требования.

```
off, press → set y, low_on, low
low, press, y<5 → bright_on, bright
low, press → light_off, off
bright, press → light_off, off
```

Требования непосредственно переписываются в программу.

Программа.

```
process Лампочка {
  off:    if (press)  { y=0; send low_on  #low }
          else #off
  low:    if  (press)  { if (y<5) {send bright_on #bright} else {send light_off #off }}
          else #low
  bright: if  (press)  {send light_off #off }
          else #bright
}
```

Поскольку конструкция off: **if** (press) <X> **else** #off эквивалентна off: **receive** press; <X>, программа преобразуется к следующему виду:

```
process Лампочка {
  off:    receive press; y=0; send low_on #low
  low:    receive press;  if (y<5) {send bright_on #bright} else {send light_off #off }
  bright: receive press; send light_off #off
}
```

6. ЭЛЕКТРОННЫЕ ЧАСЫ С БУДИЛЬНИКОМ

На примере автоматной программы «Электронные часы с будильником» [19] дадим иллюстрацию определения требований и построения автоматной программы.

Содержательное описание. 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.

На следующем шаге разработки программы из содержательного описания выделяется описание внешнего окружения программы. Действия с часами целесообразно представить в виде класса Часы.

Окружение.

```
message H, M, A; // нажатие кнопок H, M и A моделируется сообщениями
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
    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. Работа часов реализуется независимым параллельным процессом. Реализация методов inc_h, inc_m и других должна гарантировать отсутствие конфликтов по доступу к переменным класса.

Управляющие состояния:

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

Данные управляющие состояния явно обозначены в содержательном описании требований.

Представленные ниже функциональные требования являются непосредственной формализацией содержательного описания требований.

Требования.

off, H → inc_h
off, M → inc_m
off, A → set
set, H → inc_alarm_h
set, M → inc_alarm_m
set, A → bell_on, on
on, H → inc_h
on, M → inc_m
on, A → bell_off, off
on, bell_limit → bell_off, off

Состояние.

Часы t;

Программа. Программа очевидным образом строится по требованиям. Она отличается от представленной в работе [3].

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

7. СИСТЕМА УПРАВЛЕНИЯ БАНКОМАТОМ

Содержательное описание. *Банкомат* предназначен для автоматизированной выдачи и приёма наличных денежных средств с использованием платёжной *банковской карты*, ассоциированной со *счетом* в одном из *банков*. На магнитной полосе карты закодированы: номер карты, даты начала и окончания действия карты и др. информация. В банкомате есть устройство считывания банковских карт, устройство выдачи наличных, устройство приема наличных денег, устройство печати чеков, клавиатура и дисплей. Используя данные, закодированные на карте, банкомат через сервер, связанный с банкоматом, по системе межбанковской связи получает доступ к счету, ассоциированному с картой.

Клиент инициирует транзакцию, когда вставляет банковскую карту в устройство считывания банкомата. Если система опознает карту, то она проверяет, не истек ли срок действия, совпадает ли введенный клиентом ПИН-код с тем, что хранится в системе, не числится ли данная карта утерянной. Клиенту даются три попытки для ввода правильного ПИН-кода, после третьей ошибки карта блокируется.

Если ПИН-код введен правильно, система предлагает клиенту выбрать одну из трех операций: снятие денег, получение справки или пополнение счета, ассоциированного с картой. Прежде чем выдать наличные, система проверяет, что на указанном счете достаточно денег, не превышен суточный лимит и что в банкомате имеется требуемая сумма. Если транзакция одобрена, то выдается запрошенная сумма, печатается чек и карта возвращается клиенту. Для одобренной транзакции получения справки печатается чек. Клиент может в любой момент отменить транзакцию, при этом карта немедленно возвращается.

Окружение.

message card, cardTaken;

Сообщение card поступает в программу управления банкоматом, как только клиент вставит банковскую карту в устройство считывания банкомата. Сообщение cardTaken посылается программе управления банкоматом сразу после изъятия клиентом возвращенной банковской карты из устройства считывания банкомата.

Класс Card_ops определяет набор методов – операций с банковскими картами, применяемых в программе cashMachine управления банкоматом.

```
class Card_ops {
    validCard( : #yes : #no); // проверяется правильность банковской карты
    check_pin( : #yes: #no); // запрашивается ПИН-код и проверяется его правильность
    block_card(); // карта блокируется: дальнейшие операции с ней через банкомат невозможны
    process give_money();
    process put_money();
    print_info();    // печатает чек о состояние счета, ассоциированного с банковской картой
    return_card(); // карта возвращается клиенту в устройстве считывания карт банкомата
    hideCard();     // карта, возвращенная клиенту, поглощается банкоматом
    .... //  поля и методы скрытой части класса
}
```

Гиперфункция validCard проверяет, что вставленная карта является правильной банковской картой: карта с указанным на ней номером выдана в одном из банков, не просрочена, не блокирована и не числится среди утерянных; выход #yes соответствует правильной карте, #no – если карта не прошла контроль. Процесс give_money() реализует выдачу денег клиенту. Сумма выдаваемых денег задается клиентом. Выдача денег реализуется при условии, что указанная сумма имеется на счете и в банкомате. В любой момент процесс может быть прерван клиентом. Процесс put_money() реализует прием денег от клиента через устройство приема наличных денег с зачислением их на счет, ассоциированный с картой. Метод hideCard, прячущий карту внутрь банкомата, срабатывает через время Twait после того, как карта была возвращена клиенту в устройстве считывания карт. Атрибуты банковской карты, считанные методом validCard, хранятся в скрытой части класса Card_ops.

Локальные программы.

```
hyper select( : #take : #put : #info : #cancel);
```

Гиперфункция select в соответствии с выбором клиента реализует переключение на одно из действий: #take – транзакцию выдачи денег, #put – транзакцию пополнения счета, ассоциированного с картой, #info – транзакцию получения справки, #cancel – завершение операций с банкоматом для получения банковской карты.

Управляющие состояния:

idle – банкомат находится в состоянии ожидания следующего клиента;

transaction – состояние банкомата перед выбором одного из трех видов транзакций по банковской карте, прошедшей авторизацию;

take – банкомат находится в транзакции выдачи денег;

put – банкомат находится в транзакции пополнения счета по банковской карте;

info – банкомат находится в транзакции выдачи справки о состоянии счета, ассоциированного с банковской картой;

cancel – банкомат находится в состоянии возврата клиенту его банковской карты .

Состояние. Объект класса Card_ops.

Требования.

```
process cashMachine {
   idle, card, → checkCard( : #transaction : #cancel);
   transaction → select( : #take : #put : #info : #cancel);
   take → give_money(), cancel;
   put → put_money(), cancel;
   info → print_info(), transaction;
   cancel → returnCard( : #idle)
}
```

Первоначально банкомат находится в состоянии idle ожидания очередного клиента. Банковская карта, вставленная в устройство считывания карт, через сообщение card инициирует запуск процесса checkCard (см. ниже), реализующего чтение атрибутов карты на магнитной полосе и проверку карты. Если карта правильная и верен введенный ПИН-код, реализуется переход в управляющее состояние transaction, иначе – в состояние cancel, где карта возвращается клиенту. В управляющем состоянии transaction клиент выбирает одну из трех транзакций или отказ от действий (кнопку Cancel) на основе чего гиперфункция select переключает работу банкомата на требуемую транзакцию или управляющее состояние cancel. В управляющем состоянии take запускается процесс give_money для выдачи клиенту денег со счета, ассоциированного с банковской картой. В управляющем состоянии put запускается процесс put_money для пополнения счета по банковской карте. В управляющем состоянии cancel запускается процесс returnCard, описанный ниже.

```
process checkCard( : #transaction : #cancel) {
   c0 →  validCard( : #c1 : #cancel);
   c1 → check_pin( : #transaction: #c2);
   c2 → check_pin( : #transaction: #c3);
   c3 → check_pin( : #transaction: #c4);
   c4 → block_card(), cancel
}
```

Работа процесса checkCard начинается запуском гиперфункции validCard для считывания атрибутов банковской карты и проверки ее правильности. Если карта признана правильной, то вызывается гиперфункция check_pin, в которой клиенту предлагается ввести ПИН-код из четырех цифр; проверяется его правильность. Если ПИН-код верный, то процесс checkCard завершается выходом #transaction. В противном случае вызов check_pin реализуется еще два раза. После третьей ошибки карта блокируется вызовом метода block_card, после чего процесс checkCard завершается выходом #cancel.

```
process returnCard( : #idle) {
   r0 →  return_card(), set t, r1
   r1, cardTaken → idle;
   r1, t > Twait → hideCard(), idle;
}
```

Процесс returnCard начинается вызовом метода return_card, реализующего возврат банковской карты в устройстве считывания карт банкомата. Устанавливается таймер t. По истечении времени Twait возвращенная карта прячется внутри банкомата, если только ранее клиент не изъял ее из устройства считывания карт, о чем извещается сообщением cardTaken. В любом случае процесс returnCard завершается выходом #idle.

Программа процесса cashMachine строится по требованиям очевидным образом.

8. ПРОГРАММА УПРАВЛЕНИЯ ЛИФТОМ

Современный пассажирский лифт [29] является сложным техническим сооружением. Управление работой лифта реализуется нетривиальной программой. Ее нередко используют для демонстрации технологии программирования. Примеры программ управления лифтом можно найти в работах [1, 14, 27, 28].

Содержательное описание. *Лифт* установлен в здании с несколькими *этажами*. Этажи пронумерованы. Лифт либо стоит на одном из этажей с *открытой* или *закрытой* дверью, либо находится между этажами и движется *вверх* или *вниз*.

На каждом этаже есть две *кнопки* вызова лифта: одна – для движения вверх, другая – для движения вниз. На нижнем этаже нет кнопки для движения вниз, а на верхнем – для движения вверх.

Внутри *кабины лифта* есть кнопки с номерами этажей. Нажатие одной из этих кнопок определяет команду остановки по прибытии лифта на соответствующий этаж.

Нажатые кнопки на этажах и в кабине лифта определяют текущее множество *заявок* на обслуживание пассажиров лифта. В момент завершения выполнения заявки соответствующая кнопка отжимается. Лифт может находиться в одном из трех состояний: направлении движения вверх (up), направлении движения вниз (down) и нейтральном (neutral). Лифт движется в одном из направлений (up или down) до тех пор, пока существуют заявки, реализуемые в этом направлении; когда заявки заканчиваются, направление движения лифта меняется на противоположное. При отсутствии заявок в обоих направлениях лифт останавливается на текущем этаже (состояние neutral).

По прибытии на этаж лифт либо останавливается на этаже, либо проходит мимо без остановки. Остановка реализуется при наличии заявки по данному этажу, т.е. при нажатой кнопке в кабине лифта, либо при нажатой кнопке на этаже, но не в противоположном направлении движению лифта.

Решение об остановке на этаже принимается заранее вблизи этажа на определенном расстоянии по специальным датчикам. Если принято решение об остановке, включается торможение и лифт останавливается.

В случае остановки на этаже дверь лифта открывается. Закрытие двери лифта происходит через промежуток времени Tdoor, либо при нажатии кнопки «закрыть дверь» в кабине лифта. Если обнаружены помехи при закрытии дверей, они повторно открываются.

Окружение.

Класс Лифт определяет набор методов – примитивов, используемых в программе Lift управления лифтом.

```
class Лифт {
    decision1( : #idle  : #start : #open);
    decision2( : #idle  : #start );
    starting(); // лифт начинает движение из состояния покоя вверх или вниз
    stopping(); // вблизи этажа включается торможение для остановки на этаже
    check_floor( : #move : #stop) ; // вблизи этажа решается, остановиться или проехать мимо
    bool near_floor(); // = true, когда движущийся лифт оказывается вблизи очередного этажа
    openDoor(); // реализуется открытие дверей лифта
    closeDoor(); // запускается процесс закрытия дверей лифта
    bool closeButton(); // = true при нажатии кнопки «закрыть дверь»
    bool closedDoor(); // = true, если закрытие дверей лифта завершено
    bool blockedDoor(); // = true, если закрытие дверей лифта остановлено человеком на этаже
                //  поля и методы скрытой части класса:
    type DIR = enum (up, down, neutral); // тип состояния движения лифта
    DIR dir; // состояние движения лифта
    type FLOOR = first_floor .. last_floor; // тип номера этажа
    FLOOR floor;  // номер этажа, мимо которого лифт проехал или на котором остановился
    ....
}
```

Для лифта, стоящего с закрытой дверью на некотором этаже, гиперфункция decision1 в зависимости от состояния кнопок определяет один из трех вариантов дальнейших действий: idle – оставаться в состоянии покоя, start – начать движение, open – открыть дверь. После закрытия дверей лифта, остановившегося на некотором этаже, гиперфункция decision2 в зависимости от состояния кнопок выбирает один из выходов: idle – оставаться в состоянии покоя, start – начать движение.

Программа Lift управления лифтом использует только первые 11 методов класса Лифт. Остальная часть класса скрыта. Однако переменные скрытой части класса могут быть использованы в инвариантах. Отметим, что в скрытой части находится большая часть окружения программы, в частности, весь механизм работы с кнопками и их световой индикацией. Скрыто также много других возможных особенностей функционирования лифта, например, возможность блокировки закрывающейся двери лифта нажатием кнопки на этаже.

Состояние. Объект класса Лифт.

Управляющие состояния программы Lift:

```
idle:  inv dir = neutral;  // лифт стоит на некотором этаже, двери закрыты
start:  inv dir ≠ neutral; // лифт начинает движение в направлении dir
open;  // лифт подошел к некоторому этажу или стоит на этаже некоторое время
process Lift {
   idle → decision1( : #idle  : #start : #open);
   start → Movement( : #open);
   open → atFloor( : #idle  : #start)
}
```

В управляющем состоянии idle лифт стоит. В соответствии с гиперфункцией decision1 произойдет переход снова в idle, пока не будет нажата кнопка на одном из этажей. Разумеется, кнопку может нажать и пассажир, проснувшийся в кабине лифта. Если нажата кнопка на том же этаже, на котором стоит лифт, происходит переход в управляющее состояние open. Гиперфункция decision1 также формирует новое значение переменной dir. Вызовы Movement и atFloor соответствуют процессам, которые определены ниже.

Управляющие состояния программы Movement:

```
start:  inv dir ≠ neutral;   // лифт начинает движение в направлении dir
move:  inv dir ≠ neutral; // лифт движется в направлении dir
stop:  inv dir ≠ neutral;   // лифт движется, находясь вблизи некоторого этажа,
                            // и начинает торможение, чтобы остановиться.
```

```
process Movement( : #open) {
   start → starting(), move;
   move, near_floor() → check_floor( : #move : #stop);
   stop → stopping(), open;
}
```

В процессе Movement метод starting() инициирует начало движения лифта с переходом в управляющее состояние move, в котором метод near_floor() проверяет приближение движущегося лифта к очередному этажу. Когда лифт находится вблизи этажа, запускается гиперфункция check_floor, определяющая, нужно ли останавливаться на этаже. Процесс переходит в управляющее состояние stop, если остановка нужна. Метод stopping запускает торможение лифта, и процесс Movement завершается внешним выходом open.

Управляющие состояния программы atFloor:

```
open;      // лифт стоит на некотором этаже с закрытыми или полуоткрытыми дверями
opened;    // двери лифта открыты
close;     // двери лифта закрывается
```

Состояние программы atFloor:

```
time t;   // время, прошедшее после полного открытия дверей лифта на текущем этаже
process atFloor( : #idle  : #start) {
   open → openDoor(), set t, opened;
   opened, closeButton() or t ≥ Tdoor  → closeDoor(), close;
   close, closedDoor() → decision2( : #idle  : #start);
   close, blockedDoor() → open;
}
```

Процесс atFloor начинает работу в управляющем состоянии open. Вызов метода openDoor реализует открытие дверей лифта. Устанавливается таймер t, и происходит переход в управляющее состояние opened. При нажатии на кнопку «закрыть дверь» или через промежуток времени Tdoor метод closeDoor запускает процесс закрытия дверей с переходом в управляющее состояние close. В соответствии с третьим и четвертым правилами ожидается одно из двух событий: полное закрытие дверей при истинности closedDoor или блокировка дверей при истинности blockedDoor(). В случае блокировки дверей лифта процесс переходит в состояние open, в котором двери повторно открываются. В случае полного закрытия дверей гиперфунция decision2 в зависимости от состояния кнопок определяет вариант дальнейшего поведения лифта: оставаться ли в состоянии покоя или начать движение. В зависимости от выбранного варианта гиперфункция завершается по одному из выходов idle или start. Поскольку оба выхода – внешние для процесса atFloor, процесс atFloor завершает свою работу с возвратом в процесс Lift.

Программа. Сначала построим программы трех процессов по их требованиям.

```
process Lift {
   idle:   decision1( : #idle  : #start : #open);
   start:  Movement( : #open)
   open:  atFloor( : #idle  : #start)
}
process Movement( : #open) {
   start:   starting() #move;
   move:  if (near_floor()) check_floor( : #move : #stop);
          #move
   stop:    stopping()  #open;
}
```

```
process atFloor( : #idle  : #start) {
                time t;     // время, прошедшее после полного открытия дверей лифта
    open:       openDoor(); set t; #opened
    opened:     if (closeButton() or t ≥ Tdoor) { closeDoor() #close }
                #opened
    close:      if (closedDoor()) decision2( : #idle  : #start);
                if (blockedDoor()) #open;
                #close
}
```

Подставляя тела программ Movement и atFloor, после упрощений получаем окончательную программу:

```
time t;     // время, прошедшее после полного открытия дверей лифта
process Lift {
    idle:     decision1( : #idle  : #start : #open);
    start:    starting();
    move:     if (near_floor()) check_floor( : #move : #stop);
              #move
    stop:     stopping();
    open:     openDoor(); set t;
    opened:   if (closeButton() or t ≥ Tdoor) { closeDoor() #close }
              #opened
    close:    if (closedDoor()) decision2( : #idle  : #start);
              if (blockedDoor()) #open;
              #close
}
```

Приведенная программа управления лифтом на порядок проще аналогичных программ в работах [1, 14, 27, 28]. Простота программы обусловлена переносом ее информационных связей внутрь класса Лифт. Программа универсальна, поскольку вся специфика внешнего окружения (кнопок и их индикации, датчиков вблизи этажа, а также закрытия и блокировки дверей) находится в скрытой части класса Лифт. Ряд особенностей работы лифта, описанные в содержательном описании, также оказалась в скрытой части.

Реализация класса Лифт. Сначала определим три массива кнопок: Up – для движения вверх на этажах, Down – для движения вниз и Cab – в кабине лифта.

```
type BUTTONS = array (bool, FLOOR); // тип массива кнопок
BUTTONS  Up, Down, Cab;
```

Факт нажатия кнопки идентифицируется значением **true** соответствующего элемента массива; значение **false** соответствует отжатой кнопке. С каждой кнопкой связана независимая программа, реагирующая на нажатие (press) и отжатие (release) кнопки пассажиром:

```
process Button(BUTTONS ButAr, int j)
    { Cycle: if (press)  ButAr[j] = true elsif (release)  ButAr[j] = false; #Cycle }
```

Программа работает параллельно с программой Lift, и поэтому возможен конфликт по доступу к переменной ButAr[j]. При этом не произойдет нарушения работы лифта, и нет необходимости применять средства синхронизации.

При открытии двери лифта срабатывает следующая программа:

```
Release() { Up[floor] = false; Down[floor] = false; Cab[floor] = false; }
```

Вызов программы Release вставляется в программу примитива openDoor. Нажатие и отжатие кнопок должно сопровождаться соответствующим изменением их световой индикации. Здесь этих действий нет, но они должны быть вставлены в реальную программу.

Ниже представлена программа гиперфункции decision1, которая в зависимости от состояния кнопок реализует выбор одного из трех управляющих состояний: idle, start или open. Аргументами гиперфункции decision1 являются переменные floor, dir, Up, Down и Cab;

результатом – dir. Отметим, что эти переменные не указываются при описании decision1 в интерфейсе класса Лифт, так как принадлежат скрытой части класса.

```
decision1( : #idle  : DIR dir′ #start : #open)
    pre dir = neutral
    pre idle: ∀ FLOOR j. ¬Up[j] & ¬Down[j] & ¬Cab[j]
    pre open: Up[floor] ∨ Down[floor] ∨ Cab[floor]
{  decision1_from( first_floor : #idle  : DIR dir #start :  #open) }
    post start: ∃ FLOOR j. (Up[j] ∨ Down[j] ∨ Cab[j]) &
                                ( j > floor & dir′ = up ∨ j < floor & dir′ = down)
```

Здесь dir = neutral – общее предусловие. Предусловие по ветви idle реализуется при всех отжатых кнопках, а по ветви open – если нажата одна из кнопок на этаже. Предусловие для ветви start определяется как дополнение по отношению к предусловиям для ветвей idle и open в рамках общего предусловия. Постусловие для ветви start определяет условие на значение итоговой переменной dir. Постусловия для ветвей idle и open отсутствуют, поскольку по этим ветвям нет результирующих переменных.

Гиперфункция decision1_from является более общей программой. Она работает начиная с этажа j в предположении, что все кнопки для этажей меньших j отжаты.

```
decision1_from(FLOOR j : #idle  : DIR dir #start : #open)
    pre dir = neutral & ∀ FLOOR p<j. ¬Up[p] & ¬Down[p] & ¬Cab[p]
{   if (Up[j] or Down[j] or Cab[j]) {
        if (j = floor)  #open ;
        if (j < floor) { dir = down; #start };
        dir = up; #start
    }
    if (j = last_floor) #idle ;
    decision1_from( j+1 : #idle  : DIR dir #start : #open)
}
```

Предусловия и постусловия по ветвям те же, что и у гиперфункции decision1.

Программа гиперфункции decision2 использует подпрограмму inFloors для проверки наличия нажатых кнопок для этажей с номерами от j до k.

```
formula InFloors(int j, k) = ∃ p = j..k. (Up[p] ∨ Down[p] ∨ Cab[p]);
inFloors(int j, k: bool b)
    pre  j > k ∨ j, k ∈ FLOOR
{   if (j > k) b = false
    else if (Up[j] or Down[j] or Cab[j]) b = true
    else inFloors(j + 1, k: b)
} post b = InFloors(j, k);
```

Подпрограммы below и above проверяют наличие нажатых кнопок, соответственно, ниже и выше этажа floor.

```
formula Below( ) = InFloors(first_floor, floor-1);
formula Above( ) = InFloors(floor+1, last_floor);
below( : bool b) { inFloors(first_floor, floor-1: b) } post b = Below( );
above( : bool b) { inFloors(floor+1, last_floor: b) } post b = Above( );
```

Гиперфункция decision2 в зависимости от состояния кнопок определяет выбор между управляющими состояниями idle и start. Кроме того, определяется направление движения dir. Действует правило: лифт не может изменить направления движения, пока по ходу движения лифта имеются нажатые кнопки.

```
decision2( DIR dir : DIR dir′ #idle  : DIR dir′ #start )
    pre dir ≠ neutral
    pre idle: ∀ FLOOR j≠ floor. ¬Up[j] & ¬Down[j] & ¬Cab[j]
{  if (dir = down & below() or dir = up & above()) #start
   else if (below()) { dir′ = down; #start }
   else if (above()) { dir′ = up; #start }
   else { dir′ = neutral; #idle }
}
    post idle: dir′ = neutral
    post start: ( Below() & dir = down ⇒ dir′ = down ) &
                ( Above() & dir = up ⇒ dir′ = up ) &
                ( Below() & dir′ = down ∨ Above() & dir′ = up );
```

Поскольку проверяются все этажи, кроме текущего, нажатие одной из кнопок для текущего этажа floor не препятствует переходу в состояние idle. Требуемое открывание дверей лифта реализуется после срабатывания decision1 в состоянии idle.

Примитив starting запускает движение лифта из состояния покоя вверх при dir = up или вниз при dir = down. В дополнении к этому в качестве текущего этажа соответственно назначается следующий по ходу движения.

```
starting(FLOOR floor: FLOOR floor′)
    pre  dir = up  ∨ dir = down
{   if (dir = down) { floor′ = floor – 1; startingDown() }
    else { floor′ = floor + 1; startingUp() }
};
```

Примитивы startingDown и startingUp реализуют запуск механизма движения лифта.

Гиперфункция check_floor, запускаемая при приближении к очередному этажу, решает, остановиться или проехать мимо. Остановка реализуется при нажатой кнопке в кабине лифта, либо при нажатой кнопке на этаже, но не в противоположном направлении движению лифта. Лифт останавливается также при отсутствии нажатых кнопок в направлении движения лифта. Если лифт проходит мимо, необходимо будет изменить значение текущего этажа floor на следующий по ходу движения.

```
check_floor( : FLOOR floor #move :  #stop)
    pre  dir = up ∨ dir = down
    pre stop: Cab[floor] ∨ dir = down & (Down[floor] ∨ ¬Below()) ∨
                            dir = up & (Up[floor] ∨  ¬Above())
{   if (Cab[floor]) #stop ;
    if (dir = down) { if (Down[floor] or not below()) #stop }
    else { if (Up[floor] or not above()) #stop };
    if (dir = down) floor′ = floor – 1 else floor′ = floor + 1;
    #move
};
```

Реализация остальных примитивов класса Лифт использует датчики и механизмы, обеспечивающие функционирование лифта. Эти датчики и механизмы предварительно должны быть определены в классе Лифт. Реализация оставшихся примитивов не представляет особенных проблем. Отметим лишь, что в реализации примитива closeDoor() дополнительно требуется запустить отжатие кнопки «закрыть дверь».

Преобразование автоматной программы управления лифтом на язык C++ и особенности компоновки итоговой программы представлены в Приложении 1.

Анализ программы. Разработка программы обычно сопровождается анализом ее свойств. Свойства программы управления лифтом исследуются исходя из основного назначения лифта – перевозить пассажиров лифта с одного этажа на другой. Рассмотрим следующее свойство: при нажатии кнопки на этаже гарантируется приход лифта на этот этаж в пределах фиксированного времени. Например, нажата кнопка на 7-ом этаже для движения вниз, причем лифт находится на 4-ом этаже и движется вниз на 1-ый этаж. В

соответствии со стратегией работы лифта он сначала должен доехать до 1-го этажа, остановиться и затем поехать вверх. При этом, если обнаруживается заявка на 9-ом этаже, то лифт должен будет пропустить 7-ой этаж при движении вверх и лишь после остановки на 9-ом этаже поехать вниз и остановится на 7-ом этаже. Таким образом, движение лифта делится на три отрезка (4-1, 1-9, 9-7), и мы должны будем проверить, что каждый из этих отрезков движения лифта является реализуемым.

Анализ интересующего нас свойства прихода лифта по нажатию кнопки реализуется разбором случаев. При этом будем исходить из того, что нажатая кнопка не может быть отжата пассажиром лифта. Предположим сначала, что лифт стоит на том же этаже. Это возможно лишь в состоянии idle. Легко проверить, что срабатывание decision1 приводит к открытию дверей. Если лифт стоял на другом этаже, то срабатывание decision1 приводит либо к движению, либо к открытию дверей лифта в случае, когда одновременно нажата кнопка на этаже, где стоит лифт. Если произошло открытие дверей лифта, то в соответствии с программой atFloor далее возможна потенциально бесконечная серия открываний и закрываний дверей из-за их блокировки. Здесь необходимо потребовать, что эта серия не может быть бесконечной, а для оценки времени необходимо вводить временное ограничение. Если двери лифта успешно закрылись, то легко проверить, что срабатывание decision2 запускает движение лифта.

Таким образом, мы приходим ко второму случаю, когда кнопка нажата и лифт находится в движении. Возможны следующие подслучаи:

2а – лифт движется в противоположном направлении по отношению к этажу с нажатой кнопкой;

2б – лифт движется к этажу с нажатой кнопкой, однако кнопка нажата в противоположном направлении;

2в – лифт движется к этажу с нажатой кнопкой, причем кнопка нажата в том же направлении.

Для состояния 2а необходимо доказать, что лифт через определенное время закончит движение в текущем направлении и повернет в обратном направлении, в результате чего попадаем в ситуацию 2б или 2в. Доказательство проводится индукцией по числу этажей до конечного этажа, на котором лифт должен гарантированно повернуть в обратную сторону. В соответствии с программой Movement лифт, подходя к очередному этажу, либо проходит мимо, либо останавливается. Если лифт проходит мимо, срабатывает индукционное предположение. Если лифт останавливается, то после закрытия дверей реализуется вызов программы decision2. Здесь достаточно доказать, что при нажатой кнопке результатом decision2 не может быть управляющее состояние idle.

Аналогичным образом для состояния 2б доказывается, что лифт либо останавливается на этаже с нажатой кнопкой, либо, пройдя мимо, продолжит движение, пока не выполнит все заявки в текущем направлении, после чего гарантированно повернет в обратном направлении, т.е. перейдет в состояние 2в. Для состояния 2в индукцией по числу этажей до этажа с нажатой кнопкой доказывается, что лифт дойдет до нужного этажа и откроет дверь лифта.

Замечания. Анализ программы является составной частью процесса разработки программы, в частности, ее тестирования и моделирования. Эффективность и качество проведения анализа программы существенно зависит от возможностей ее понимания программистом. Здесь мы акцентируем внимание не на формальном, а на содержательном анализе программы. Предлагаемая технология автоматного программирования нацелена на упрощение программы для того, чтобы улучшить ее понимание. Для целей анализа автоматной программы управления лифтом наиболее удобным является ее представление в виде трех программ Lift, Movement и atFloor на языке спецификации требований.

Чтобы программа была корректной, необходимо гарантировать, что значения переменных dir, floor, Up, Down и Cab правильно отражают реальное состояние лифта. Например, в случае отказа одного из датчиков вблизи этажа не будет запущена программа

check_floor, вследствие чего значение текущего этажа floor не будет соответствующим образом изменено; дальнейшая работа программы Lift при неправильном значении floor будет ошибочной. Чтобы повысить отказоустойчивость программы Lift, в качестве результата примитива near_floor() следует выдавать не логическое значение, а номер текущего этажа.

9. ПРОТОКОЛ ЧЕРЕДОВАНИЯ БИТОВ

Содержательное описание. Протокол чередования битов (Alternating Bit Protocol, ABP) реализует передачу данных через ненадёжные каналы связи. Процесс *передатчик* S вводит из внешнего окружения потенциально бесконечную последовательность блоков данных с помощью сообщения in(d), где d – блок данных. Передатчик S пересылает очередной блок d сообщением a(n, d), где n – номер блока, другому параллельно функционирующему процессу – *приемнику* R. Получив сообщение a(n, d), приемник R выводит блок d во внешнее окружение с помощью сообщения out(d).

Сообщения между процессами S и R передаются через *ненадёжные каналы* связи: возможны повторы, потери и искажения сообщений, но не меняется их порядок. Надёжная передача данных осуществляется при помощи передаваемого синхронизующего бита и повторов сообщений. Здесь мы определим более простую версию, протокол n-ABP, в котором синхронизация реализуется с помощью номера блока, а затем покажем, как этот протокол трансформируется в классический протокол ABP.

Для обеспечения надежности протокол n-ABP реализует следующие дополнительные действия. Приемник S содержит независимый счетчик m номеров блока. Получив очередной блок сообщением a(n, d) приемник проверяет, что его номер блока n совпадает с m, и лишь в случае n = m пересылает его сообщением out(d). Получение любого сообщения a(n, d) подтверждается посылкой ответного сообщения b(n) приемнику S. Получив сообщение b(j), приемник сверяет j с номером текущего посланного блока n. Условие j = n гарантирует доставку очередного блока с номером n. В этом случае передатчик переходит к запросу следующего блока данных; в противном случае, через промежуток времени Twait повторяет посылку сообщения a(n, d).

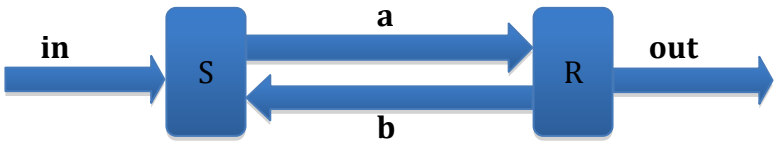

Рис.1. Схема протокола ABP

Моделирование ненадежной передачи сообщений a и b реализуется через логические переменные окружения sa и sb, значения которых непредсказуемо меняются во времени. Истинность переменной sa (или sb) определяет надежность передачи сообщения a (или b). Контроль порчи блока реализуется контрольным суммированием очередного блока и передачей этой суммы в сообщении a с последующей проверкой этой суммы в приемнике. Здесь мы опускаем действия по контролю порчи блоков, поскольку они не меняют принципиально схему протокола.

Окружение.

```
type Data;
message a(nat, Data), b(nat);
bool sa, sb;
```

Состояние: **nat** n = 0, m =0; **time** t; Data d

Формально следовало бы включить в состояние значение блока d в процессе R, однако фактически это локал.

Управляющие состояния: s0, s1, s2, s3, s4 , r0, r1, r3, r4:

Требования.

```
process ABP() { S || R }
```

Оператор S || R реализует параллельное исполнение процессов S и R. При запуске протокола ABP счетчики n и m совпадают. Если один из процессов, S или R, будет остановлен, его перезапуск нельзя проводить автономно от другого процесса – это может привести к ошибке, поскольку счетчики n и m должны быть синхронизированы. Необходимо будет заново перезапускать протокол ABP (т.е. оба процесса S и R), либо использовать протокол рукопожатия для обеспечения синхронизации.

```
process S() {
 s0:   nat n = 0 #s1
 s1:   in(d) → #s2
 s2:   set t #s3
 s3:   sa → a(n, d) #s4
 s3:   #s4
 s4:   b(j), j = n  → n = n+1 #s1
 s4:   t>Twait   → #s2
}
process R() {
 r0:   nat m = 0 #r1
 r1:   a(i, d) → #r3
 r3:   i = m  → out(d), m = m+1 #r4
 r3:   #r4
 r4:   sb  → b(i) #r1
 r4:   #r1
}
```

Программа.

```
process ABP()  { S || R }
process S() {
        nat n = 0;
 s1:    receive in(DATA d);
 s2:    set t;
        if (sa) send a(n, d);
 s4:    if (b(nat j) & j = n) { n = n+1 #s1}
        if (t>Twait) #s2 else #s4
}
process R() {
        nat m = 0;
 r1:    receive a(nat i, DATA d);
        if (i = m) { send out(d); m = m+1};
        if (sb)  send b(i);
        #r1
}
```

Анализируя программу протокола, можно определить, что номера блоков в сравнениях j = n и i = m отличаются не более, чем на единицу. Тогда переменные n и m можно заменить логическими, n = n+1 заменить на n = ¬n, а m = m+1 – на m = ¬m. В результате получим классический протокол ABP.

10. БЫТОВОЙ АЛГОРИТМ: РЫБНАЯ ЛОВЛЯ

Приведенная ниже автоматная программа используется в работах [33, глава 4, стр. 47] и [34, глава 1, стр. 10] для иллюстрации основной алгоритмической структуры «силуэт» графического языка программирования Дракон. Программа проста и не требует предварительного содержательного описания. Интерес к этой программе определяется тем, что ее структура подобна соответствующей Дракон-схеме. Начнем с определения требований.

Требования.

```
process Рыбная_ловля( : #Конец) {
  Подкотовка_к_ловле:     Накопай_червей, Возьми_удочку,
                          Доберись_до_места_ловли  #Начало_ловли
  Начало_ловли:           Насади_червяка  #Ожидание_клева
  Ожидание_клева:         Забрось_удочку  #Процесс_клева
  Процесс_клева:          Клюнула → Подсекай  #Рыбацкая_работа
  Процесс_клева:          Пора_домой →  #Обратная_дорога
  Рыбацкая_работа:        Рыбка_попалась →
                                  Сними_добычу_с_крючка_и_кинь_в_садок
                                  #Продолжение_ловли
  Рыбацкая_работа:        Пора_домой →  #Обратная_дорога
  Рыбацкая_работа:        Червяк_цел →  #Ожидание_клева
  Рыбацкая_работа:        #Начало_ловли
  Продолжение_ловли:      Пора_домой →  #Обратная_дорога
  Продолжение_ловли:      #Начало_ловли
  Обратная_дорога:        Собери_вещи  #Дорога_домой
  Дорога_домой:           Удалось_что-нибудь_поймать →
                              С_хорошим_настроением_направляйся_домой #Конец
  Дорога_домой:           Зайди_в_магазин_купи_рыбы_чтобы_дома_не_краснеть
                          #Конец
}
```

Программа.

```
process Рыбная_ловля( : #Конец) {
                          Накопай_червей;
                          Возьми_удочку;
                          Доберись_до_места_ловли;
  Начало_ловли:           Насади_червяка;
  Ожидание_клева:         Забрось_удочку;
  Процесс_клева:          if (Клюнула) { Подсекай  #Рыбацкая_работа }
                          else if (Пора_домой)#Обратная_дорога
                          else #Процесс_клева
  Рыбацкая_работа:        if (Рыбка_попалась)
                              {  Сними_добычу_с_крючка_и_кинь_в_садок
                                 if (Пора_домой) #Обратная_дорога
                                 else #Начало_ловли
                              }
                          else if (Пора_домой) #Обратная_дорога
                          else if (Червяк_цел) #Ожидание_клева
                          else #Начало_ловли
  Обратная_дорога:        Собери_вещи;
                          if (Удалось_что-нибудь_поймать)
                             С_хорошим_настроением_направляйся_домой
                          else
                             Зайди_в_магазин_купи_рыбы_чтобы_дома_не_краснеть
                          #Конец
}
```

Данная программа изоморфна соответствующей Дракон-схеме, приведенной в работах [33, 34]. Каждому сегменту кода соответствует ветвь Дракон-схемы. Однко есть отличия. Для управляющих состояний Начало_ловли и Процесс_клева нет соответствующих ветвей в Дракон-схеме. Дополнительную ветвь Начало_ловли можно было бы внести в Дракон-схему, однако в этом случае она не поместилась бы на одной странице. Дополнительное

управляющее состояние Процесс_клева введено из-за того, что внутри сегмента кода запрещены внутренние циклы, которые разрешены в Дракон-схеме.

Отметим, что требования и Дракон-схема не изоморфны. Набор управляющих состояний, используемых в требованиях, покрывает все ветви Дракон-схемы, но не наоборот.

ЗАКЛЮЧЕНИЕ

Автоматные программы по своей структуре существенно сложнее программ-функций. Технология автоматного программирования предлагает комплекс методов для упрощения автоматных программ в целях улучшения их понимания программистом.

Язык спецификации требований позволяет компактно формализовать логику процессов в виде набора правил. Каждое правило независимо определяет реакцию автоматной программы на внешнее событие.

Гиперграфовая структура автоматной программы, являющаяся продолжением аппарата гиперфункций, обладает более высокой степенью гибкости в декомпозиции программы, не реализуемой классическими автоматами. Гиперграфовая декомпозиция позволяет заменить информационные связи управляющими, упрощая тем самым состояние автоматной программы.

Автоматная программа строится из частей, относящихся к классу программ-функций. Технология автоматного программирования должна быть адекватно интегрирована с технологиями объектно-ориентированного и предикатного программирования. Набор рекомендаций по интеграции разных стилей программирования представлен в виде свода золотых правил программирования. Отметим, что описанная технология автоматного программирования может быть адаптирована для автоматного программирования на базе императивного языка вместо предикатного.

Для упрощения программы применяется методы объектно-ориентированного программирования, позволяющие спрятать внутри классов часть переменных состояния программы и связей между ними. Для программ, представленных в разд. 6–8, состояние автоматной программы полностью спрятано внутри соответствующего класса. Как следствие, в автоматной программе нет переменных и нет информационных связей между ее частями, что существенно ее упрощает.

В настоящей работе исследуется базис технологии автоматного программирования. Задачами следующего уровня являются:

- – разработка методов верификации автоматных программ;
- – специализация технологии автоматного программирования для разных подклассов реактивных систем;
- – разработка редактора для дуального (текстового и графического) представления автоматной программы; в качестве графического языка допустим только язык Дракон [33, 34].

Автор благодарен А.А. Шалыто за его работы по автоматному программированию, стимулировавшие исследования автора.

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

Список литературы

1. Гома Х. UML. Проектирование систем реального времени, параллельных и распределенных приложений. М.: ДМК Пресс, 2002. 684 с.
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).
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, P. 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. Вшивков В.А., Маркелова Т.В., Шелехов В.И. Об алгоритмах сортировки в методе частиц в ячейках // Научный вестник НГТУ, Т. 4 (33) , 2008. С. 79-94.
8. Шелехов В.И. Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования. – Новосибирск, 2012. – 30с. – (Препр. / ИСИ СО РАН. № 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, LCNS 8244. – 2013. – P. 61-77.
14. Наумов А.С., Шалыто А.А. Система управления лифтом. Санкт-Петербург, 2003. 51с. http://is.ifmo.ru/download/elevator_a.pdf
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. Поликарпова Н.И., Шалыто А.А. Автоматное программирование / СПб.: Питер. 2009, 176с. http://is.ifmo.ru/books/_book.pdf
20. A Tutorial on Uppaal 4.0. Revised and extended version. – 2006. – http://www.uppaal.org/
21. Шпаков В.М. Формализация и использование знаний о развитии процессов // Тр. 16-й межд. конф. «Проблемы управления и моделирования в сложных системах». — Самара, Самарский научный центр РАН, 2014. – С. 290-295.
22. Тумуров Э.Г., Шелехов В.И. Определение требований к системе управления полетом квадрокоптера // Тр. 16-й межд. конф. «Проблемы управления и моделирования в сложных системах». – Самара, Самарский научный центр РАН, 2014. – С. 627-633.
23. Шелехов В.И. Оптимизация автоматных программ методом трансформации требований // Тр. 2-й межд. конф. «Инструменты и методы анализа программ». – Кострома, Костромской государственный технологический университет. – 2014. – С. 175-183. http://persons.iis.nsk.su/files/persons/pages/req_k.pdf
24. Klahr D., Langley P., Neches R. Production System Models of Learning and Development. – Cambridge, Mass.: The MIT Press. 1987. – 467 P.
25. Alur R., Dill D.L. A theory of timed automata // Theor. Comput. Sci. – 1994. – P. 183-235.
26. Шелехов В.И. Тумуров Э.Г. Логика невзаимодействующих программ и реактивных систем // Вестник Бурятского Государственного Университета. Секция: математика, информатика, Вып. 9 / 2012. – Улан-Удэ, 2012. – С. 81-90.
27. Решетников Е.О., Смачных М.В. Система управления пассажирским лифтом / Санкт-Петербургский государственный университет информационных технологий, механики и оптики. 2006. http://is.ifmo.ru/download/umlift.pdf
28. Кнут Д.Э. Искусство Программирования. – Том 1. Основные Алгоритмы. – 2006. разд. 2.2.5.
29. Манухин С.Б., Нелидов И.К. Устройство, техническое обслуживание *и* ремонт лифтов. М. «Академия», 2004. – 336 с.
30. Пароджанов В. Д. Как улучшить работу ума. Алгоритмы без программистов – это очень просто! – М.: Дело, 2001. – 360 с.
31. Бабецкий Г.И. и др. Система автоматизации программирования АЛЬФА // ЖВМиМФ, т.5, №2, М., 1965.
32. Шалыто А. А. SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб: Наука, 1998. http://is.ifmo.ru/books/switch/1

33. Пароннджанов В. Д. Учись писать, читать и понимать алгоритмы. Алгоритмы для правильного мышления. Основы алгоритмизации. — М.: ДМК Пресс, 2012. — 520 с.
34. Пароннджанов В. Д. Язык ДРАКОН. Краткое описание. – М., 2009. – 124 с. – http://drakon.su/_media/biblioteka/drakondescription.pdf
35. Bellini P., Mattolini R., and Nesi P. Temporal logics for real-time system specification // ACM Comp. Surveys, 32(1). – 2000. – P.12–42.
36. Arvind H. Bluespec: a language for hardware design, simulation, synthesis and verification // MEMOCODE. – 2003. – P 249–254.
37. 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.
38. Cooke D. E., Rushton J. N. Taking Parnas's Principles to the Next Level: Declarative Language Design. *Computer*, 2009, vol. 42, no. 9, P. 56-63.
39. Шелехов В.И. Предикатное программирование. Учебное пособие. Новосибирск: НГУ, 2009. 109с.
40. FSP Language Specification. http://www.doc.ic.ac.uk/ltsa/eclipse/help/index.html?appendix_b___fsp_language_spec.htm
41. Chaudhary S., Li L., Berki E., Helenius M., Kela J., Turunen M. Applying Finite State Process Algebra to Formally Specify a Computational Model of Security Requirements in the Key2phone-Mobile Access Solution // FMICS'15. LCNS 9128, 2015. P. 128-145.
42. Vlissides J., Johnson R., Helm R., Gamma E. Design Patterns: Elements of Reusable Object-Oriented Software / Addison-Wesley. – 1994. – 431 P.
43. Ferg S. Event-Driven Programming: Introduction, Tutorial, History / SourceForge. – 2006. – 59 P.
44. Шелехов В.И., Демаков И.В. Спецификация и реализация радиус-сервера интернет-телефонии. // Методы предикатного программирования. Вып.2 / ИСИ СО РАН. — Новосибирск, 2006. — С. 40-63.

Приложение 1

Особенности реализации программы управления лифтом

В Приложении 1 представлено преобразование автоматной программы управления лифтом (см. разд. 8) на язык C++ и особенности компоновки итоговой программы

Сначала проведем оптимизирующую трансформацию предикатных программ, являющихся примитивами класса Лифт, на императивное расширение языка P, с которого программа легко кодируется на язык C++. Далее определим порядок сборки полной программы.

В программах decision1 и decision1_from реализуются склеивания dir ← dir', Up← Up', Down← Down', Cab←Cab' и замена хвостовой рекурсии циклом:

```
decision1( : #idle  : DIR dir #start : #open)
{  decision1_from( first_floor : #idle  : DIR dir #start : #open) }

decision1_from(FLOOR j : #idle  : DIR dir #start : #open)
{ loop {
     if (Up[j] or Down[j] or Cab[j]) {
        if (j = floor) #open;
        if (j < floor) { dir = down; #start };
        dir = up; #start
    }
    if (j = last_floor) #idle ;
    j = j+1;
  }
}
```

Далее проводится подстановка вызова decision1_from и оформление цикла

```
decision1( : #idle  : DIR dir #start : #open)
{ for (j = first_floor; ; j = j+1 ) {
    if (Up[j] or Down[j] or Cab[j]) {
        if (j = floor) #open;
        if (j < floor) { dir = down; #start };
        dir = up; #start
    }
    if (j = last_floor) #idle ;
  }
}
```

В программе inFloors проводится замена хвостовой рекурсии циклом.

```
inFloors(int j, k: bool b)
{ loop {
    if (j > k) { b = false; break}
    else if (Up[j] or Down[j] or Cab[j]) { b = true; break}
    else j = j + 1
  }
};
```

Далее реализуется оформление цикла и замена предиката функцией.

```
bool inFloors(int j, k)
{ for(; ; j = j + 1) {
    if (j > k) return false
    else if (Up[j] or Down[j] or Cab[j]) return true
  }
};
bool below( ) { return inFloors(first_floor, floor-1)};
bool above() { return inFloors(floor+1, last_floor)};
```

В остальных программах проводятся склеивания: dir ← dir′, floor ← floor′.

```
decision2( DIR dir : DIR dir #idle  : DIR dir #start )
{  if (dir = down & below() or dir = up & above()) #start
   else if (below()) { dir = down; #start }
   else if (above()) { dir = up; #start }
   else { dir = neutral; #idle }
}
starting(FLOOR floor: FLOOR floor)
{   if (dir = down) { floor = floor – 1; startingDown() }
    else { floor = floor + 1; startingUp() }
};

check_floor( : FLOOR floor #move :  #stop)
{   if (Cab[floor]) #stop ;
    if (dir = down) { if (Down[floor] or not below()) #stop }
    else { if (Up[floor] or not above()) #stop };
    if (dir = down) floor = floor – 1 else floor = floor + 1;
    #move
};
```

Поскольку для гиперфункций нет аналогов в языке C++ и у программ-гиперфункций decision1, decision2 и check_floor по одному вызову, то логичным выглядит решение открыто подставить программы гиперфункций на место вызовов. Однако такое решение является плохим, поскольку будут нарушены **Правила** 3, 4 и 10, сформулированные в разд. 4. По этой причине в итоговой программе коды программ-гиперфункций вынесены из кода программы Lift, а в позиции вызова соответствующей гиперфункции стоит оператор перехода на начало программы-гиперфункции.