Технология автоматного программирования на примере программы управления лифтом

В.И. Шелехов, Э.Г. Тумуров

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. В качестве языка автоматного программирования используется язык спецификации функциональных требований. Технология представлена в виде свода правил, определяющих правильный баланс в интеграции автоматного, предикатного и объектно-ориентированного программирования. Технология иллюстрируется на примере программы управления лифтом.

Ключевые слова: понимание программ, автоматное программирование, реактивная система, инженерия требований, спецификация требований.

ВВЕДЕНИЕ

Автоматное программирование [1, 11, 23] ориентировано на класс *реактивных систем*, реализующих взаимодействие с внешним окружением системы и реагирующих на определенный набор событий (сообщений).

Определение требований – **первый этап** в разработке реактивной системы. *Требования* — совокупность утверждений о свойствах разрабатываемой программы. *Функциональные требования* определяют поведение программы. Их наиболее популярной формой являются *сценарии использования* (*use case*) [3]. Определение требований должно проводиться методами *инженерии требований* в соответствии со стандартом [4]. Для сложных производственных систем построение требований требует высокой квалификации специалистов в области инженерии требований.

Второй этап – формализация функциональных требований, содержательно сформулированных на первом этапе, в виде автоматной программы. При этом реализуется верификация формальных требований относительно содержательных, что обычно позволяет обнаружить значительное число ошибок.

Спецификация функциональных требований на формальном языке спецификаций не отличается принципиально от автоматной программы на некотором императивном языке. Отметим, что в мировой практике формальные языки спецификации требований используются лишь в 5% случаев [28]. Фактически, построение автоматной программы является процессом формализации содержательных требований. Эта особенность является причиной многочисленных спекуляций по поводу программирования без программистов. В действительности, здесь нужны программисты, владеющие инженерией требований и навыками формализации требований.

Ввиду трудности формализации требований является существенным, в какой степени используемый язык автоматного программирования способствует хорошему пониманию автоматных программ. Другой важный аспект – применяемая технология автоматного программирования.

Язык автоматного программирования [1, 23] – компактный язык с синтаксисом в стиле C и C++, который строится как расширение *базисного языка* предикатного или императивного программирования. Транслятор с языка автоматного программирования преобразует автоматную структуру программы в конструкции базисного языка. В дополнении к операторному языку автоматного программирования имеется эквивалентный

формальный *язык требований* с более компактной формой их записи, которая по структуре ближе к содержательным функциональным требованиям.

В настоящей статье описывается развиваемая авторами технология автоматного программирования, эффективность использования которой иллюстрируется на примере программы управления лифтом. В разд. 1 дается общее описание базисного языка, которым является язык предикатного программирования P [2]. Базис автоматного программирования определяется в разд. 2. Язык требований описывается в разд. 3. Технология автоматного программирования формулируется в разд. 4; определены особенности интеграции автоматного, предикатного и объектно-ориентированного программирования. Процесс построения программы управления лифтом описывается в разд. 5. Обзор работ представлен в разд. 6. Особенности верификации автоматных программ излагаются в рад. 7.

1. ПРЕДИКАТНОЕ ПРОГРАММИРОВАНИЕ

Предикатная программа относится к классу программ-функций [29] и является предикатом в форме вычислимого оператора. Язык предикатного программирования P [2] 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 P следующего вида:

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

Необязательные конструкции предусловия и постусловия являются формулами на языке исчисления предикатов; они используются для улучшения понимания программ и для дедуктивной верификации [5-7]. Ниже представлены основные конструкции языка P: оператор присваивания, блок (оператор суперпозиции), условный оператор, вызов программы и описание переменных, используемое для аргументов, результатов и локальных переменных.

<переменная> = <выражение>
{<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, значительно усложняющие программу. Вместо циклов используются рекурсивные программы, 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.

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.

Аппарат гиперфункций является более общим и гибким по сравнению с известным механизмом обработки исключений, например, в таких языках, как Java и C++. Использование гиперфункций делает программу короче, быстрее и проще для понимания [7, 10]. Отметим, что гиперграфовая структура автоматной программы является естественным продолжением аппарата гиперфункций.

2. ЯЗЫК АВТОМАТНОГО ПРОГРАММИРОВАНИЯ

Язык автоматного программирования строится расширением языка P [2]. *Автоматная программа* определяется следующей конструкцией:

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

Автоматная программа определяет конечный автомат. Вершина автомата – *управляющее состояние* программы. Ориентированная гипердуга автомата соответствует некоторому *сегменту кода* и связывает одну вершину с одной или несколькими другими вершинами.

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 в других программах из окружения данной программы.

Произвольный <сегмент кода> представляется конструкцией:

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

Исполнение <оператора> завершается либо оператором перехода вида #M, либо нормально; в последнем случае исполнение продолжится с начала следующего сегмента. Оператор #M, где M – имя управляющего состояния, реализует переход на начало сегмента, ассоциированного с управляющим состоянием M.

<Инвариант сегмента> должен быть истинным перед выполнением <оператора>, когда исполняемая программа приходит в данное управляющее состояние. Инвариант повышает понимание программы, его можно также использовать для верификации.

В качестве примера автоматной программы рассмотрим модуль операционной системы, реализующий следующий сценарий работы с пользователем (рис.1).

```
login:   if (exists(get_string()))
                #passw
         else   #login
passw:   if (valid(get_string()))
                #session
         else   #passw
session: сессия_пользователя();
         #login
```

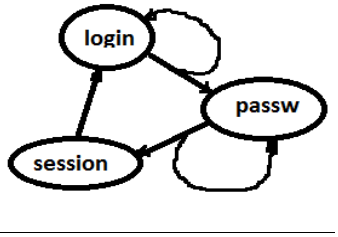

Рис. 1 – Схема работы ОС с пользователем: программа и ее автомат

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.

В общем случае реактивная система определяется в виде композиции нескольких независимых автоматных программ, исполняемых параллельно и взаимодействующих между собой через сообщения и разделяемые переменные.

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

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.

3. ЯЗЫК ТРЕБОВАНИЙ

Требования — совокупность утверждений относительно свойств разрабатываемой программы. *Функциональные требования* определяют поведение программы. Их наиболее популярной формой являются *сценарии использования* (*use case*) [3]. В нашем подходе они формализованы в виде правил на языке продукций [24], который обычно применяется 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.

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

$$<\text{условие}_1>, <\text{условие}_2>, \ldots, <\text{условие}_n> \rightarrow <\text{действие}_1>, \ldots, <\text{действие}_m>;$$

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 утверждение того, 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.

4. ТЕХНОЛОГИЯ АВТОМАТНОГО ПРОГРАММИРОВАНИЯ

Разработка автоматной программы состоит из следующих этапов:

- определение требований в содержательной форме;
- формализация требований в виде набора правил;
- трансляция набора правил на язык автоматного программирования.

Фрагменты, соответствующие программам-функциям, разрабатываются в технологии предикатного программирования. Наконец, полная программа транслируется на язык C++.

На первом этапе определение требований фиксируется в виде *содержательного описания*. Рекомендуемая технология представлена в стандарте [4]. Формализация требований реализуется в следующей последовательности:

- специфицируются объекты внешнего окружения автоматной программы;
- описываются переменные *состояния* автоматной программы, определяются связи между переменными;
- фиксируются *управляющие состояния*; некоторые из них снабжаются инвариантами;
- определяется набор *требований* в виде системы правил. Каждое правило обеспечивает адекватную реакцию на определенное сообщение или событие.

Процесс формализации требований сопровождается их верификацией относительно содержательного описания.

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

Содержательное описание представлено в разд. 2.

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.

Иногда (менее, чем в 10% случаев) непосредственная формализация требований дает неэффективную программу. В такой ситуации применяется метод трансформации требований [11], изменяющий автоматную структуру программы.

Интеграция с технологиями предикатного и императивного программирования. 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. Поэтому *не следует использовать автоматное программирование для программ-функций*, являющихся фрагментами сегментов кода автоматных программ.

Необходима адекватная интеграция разных стилей программирования. *Не следует смешивать разные стили: 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. Отметим, что замена простых операторов вызовами в составе автоматной программы дает обратный эффект – избыточная структуризация введением дополнительного уровня иерархии усложняет программу.

Циклы в автоматной программе имеют другую природу по сравнению с циклами императивной программы. *Не рекомендуется использовать циклы типа* ***while*** *для конструирования автоматной программы*.

Интеграция с объектно-ориентированной технологией. Объектно-ориентированная декомпозиция позволяет существенно снизить сложность автоматной программы. *Внешнее окружение и состояние автоматной программы определяется как интерфейс класса в более простом и абстрактном виде*. При этом часть переменных состояния и связей между ними, а также детали внешнего окружения скрываются внутри класса.

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.

Баланс информационных и управляющих связей. Управляющие состояния можно заменить значениями дополнительной переменной в состоянии автоматной программы. И наоборот, переменную с ограниченным числом значений можно удалить из состояния программы с введением дополнительных управляющих состояний, эквивалентным образом заменяя информационные связи управляющими.

Модифицируем программу на рис.1 в стиле switch-технологии А. Шалыто [30]:

```
type STATE = enum(login, passw, session);
STATE state = login;
M:    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
      }
#M
```

В модифицированной программе имеется лишь одно управляющее состояние M и единственный сегмент кода – оператор **switch**. Управляющие состояния login, passw и session становятся значениями переменной состояния state в модифицированной программе.

Исходная программа проще модифицированной. Детальное сравнение проводилось в работе [1, разд. 4, рис.2]. Поэтому *не следует переводить управляющие состояния в состояние автоматной программы*. Однако замена информационных связей управляющими, приводящая к существенному увеличению программы, вряд ли оправдана.

Использование гиперфункций уменьшает число переменных состояния, тем самым упрощая автоматную программу. Отметим, что сложность автоматной программы зависит от числа переменных состояния программы, иногда экспоненциально.

Эргономическое правило. *Любая подпрограмма должна помещаться в пределах экрана монитора* – одновременная видимость всех ее информационно-управляющих связей упрощает ее восприятие и анализ. Применение гиперграфовой декомпозиции дает возможность удобно и гибко разделить программу на части произвольного размера.

Инварианты автоматной программы, ассоциированные с управляющими состояниями, принципиально отличаются от инвариантов циклов и инвариантов классов императивной программы. В большинстве случаев управляющее состояние не содержит инварианта. Иначе говоря, инвариант тождественно истинен.

5. ПРОГРАММА УПРАВЛЕНИЯ ЛИФТОМ

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.

R1: Лифт движется в одном из направлений до тех пор, пока существуют заявки, реализуемые в этом направлении; когда заявки заканчиваются, направление движения лифта может быть изменено. Здесь R1 обозначает имя, введенное для идентификации приведенного требования. При отсутствии заявок в обоих направлениях лифт *останавливается* на текущем этаже.

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 {
    decisionIdle( : #idle  : #start : #open);
    decisionClosed( : #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 блокировано пассажиром
                //  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, на котором стоит лифт или к которому подъезжает
    ....
}
```

Состояние dir = neutral устанавливается при отсутствии заявок после остановки лифта.

Для лифта, стоящего с закрытой дверью на некотором этаже, гиперфункция decisionIdle 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 decisionClosed в зависимости от состояния кнопок выбирает один из выходов: idle – перейти 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.

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;  // перед открытием дверей лифт встал или уже стоит на некотором этаже
```

```
process Lift {
   idle → decisionIdle( : #idle  : #start : #open);
   start → Movement( : #open);
   open → atFloor( : #idle  : #start)
}
```

В управляющем состоянии idle лифт стоит. В соответствии с гиперфункцией decisionIdle 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 decisionIdle 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;   // перед торможением лифт движется вблизи некоторого этажа,
```

```
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 закрываются
```

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() → decisionClosed( : #idle  : #start);
   close, blockedDoor() → open;
}
```

Процесс atFloor начинает работу в управляющем состоянии open. Вызов метода openDoor реализует открытие дверей лифта. Далее 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 процесса atFloor 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 гиперфункция decisionClosed в зависимости от состояния кнопок определяет вариант дальнейшего поведения лифта: перейти 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:   decisionIdle( : #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()) decisionClosed( : #idle  : #start);
            if (blockedDoor()) #open;
            #close
}
```

Подставляя тела программ Movement и atFloor на место их вызовов, после упрощений получаем окончательную автоматную программу:

```
time t;     // время, прошедшее после полного открытия дверей лифта
process Lift {
   idle:    decisionIdle( : #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()) decisionClosed( : #idle  : #start);
            if (blockedDoor()) #open;
            #close
}
```

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

Программа работает параллельно с программой Lift, и поэтому возможен конфликт по доступу к переменной Buttons[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 decisionIdle, которая в зависимости от состояния кнопок реализует выбор одного из трех управляющих состояний: idle, start или open. Аргументами гиперфункции decisionIdle являются переменные floor, dir, Up, Down и Cab; результатом – dir. Отметим, что эти переменные не указываются при описании decisionIdle в интерфейсе класса Лифт, так как принадлежат скрытой части класса.

```
decisionIdle( : #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]
    post start: ∃ FLOOR j. (Up[j] ∨ Down[j] ∨ Cab[j]) &
                                ( j > floor & dir' = up ∨ j < floor & dir' = down)
{  decisionIdle_from( first_floor : #idle  : DIR dir' #start :  #open) }
```

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

```
decisionIdle_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 ;
   decisionIdle_from( j+1 : #idle  : DIR dir #start : #open)
}
```

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

Программа гиперфункции decisionClosed 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
   post b = InFloors(j, k)
{   if (j > k) b = false
   else if (Up[j] or Down[j] or Cab[j]) b = true
   else inFloors(j + 1, k: b)
};
```

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

Гиперфункция decisionClosed 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.

```
decisionClosed( DIR dir : DIR dir' #idle  : DIR dir' #start )
   pre dir ≠ neutral
   pre idle: ∀ FLOOR j≠ floor. ¬Up[j] & ¬Down[j] & ¬Cab[j]
   post idle: dir' = neutral
   post start: ( Below() & dir = down ⇒ dir' = down ) &
               ( Above() & dir = up ⇒ dir' = up ) &
               ( Below() & dir' = down ∨ Above() & dir' = up );
{  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 }
}
```

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 decisionIdle 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, значение текущего этажа floor заменяется 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».

Моделированиее управления лифтом. Для предикатных программ, являющихся примитивами класса Лифт, применяется оптимизирующая трансформация на императивное расширение языка P, с которого программа легко кодируется на язык C++. Итоговые программы примитивов приведены в Приложении 1. Лифт моделируется в виде графического интерфейса пользователя, созданного с помощью инструмента Qt. Модель управления лифта работает под ОС Windows и доступна по адресу: http://persons.iis.nsk.su/files/persons/pages/lift-win32.zip

Другие варианты. Возможна другая версия программы без переменной dir с увеличением числа управляющих состояний и раздваиванием фрагментов программы, реализующих движение. Вместо процесса Movement появятся процессы Movement_up и Movement_down. Аналогичным образом расклеиваются decisionClosed, check_floor, starting и другие примитивы. Программа увеличивается по размеру почти вдвое, но при этом упрощается. Заметим, что упрощаются лишь примитивы, принадлежащие классу программ-функций. Удвоение числа управляющих состояний существенно усложняет автоматную программу. Поскольку критичной является сложность именно автоматной части программы, новая версия программы хуже исходной. Можно пойти дальше и устранить переменную floor. При этом следует зафиксировать число этажей, например, пять, и реализовать примитивы перемещения лифта между соседними этажами. Полезность этой версии сомнительна.

Замечания. Представленная выше программа управления лифтом на порядок проще аналогичных программ в работах [25, 12-15]. 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.

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.

6. ОБЗОР РАБОТ

Современный пассажирский лифт [16] является сложным техническим сооружением. Программа управления работой лифта нетривиальна. Ее нередко используют для демонстрации технологии программирования. Примеры программ управления лифтом можно найти в работах [25, 12-15]. В книге Д. Кнута представлена нетривиальная реализация в форме сопрограмм [14]. В книге Х. Гома иллюстрируются различные виды диаграмм UML [25]. Автоматные методы программирования в сочетании с технологией объектно-ориентированного программирования представлены в работах [12, 13]. В руководстве [15] определяется серия из пяти последовательно уточняемых моделей лифта в технологии Event-B. Предложенная авторами программа управления лифтом существенно проще и короче программ в упомянутых выше работах. Есть три составляющие, обеспечивающие простоту нашей программы Lift: гиперграфовая композиция в сочетании с механизмом гиперфункций, объектная ориентированность и реализация примитивов вне автоматной программы на языке P.

Формальный метод моделирования и анализа реактивных систем Event-B [17] определяет процесс построения серии моделей исходя из содержательного описания требований к реактивной системе. Каждая очередная модель является детализацией (refinement) предыдущей модели. Корректность моделей и согласованность моделей различных уровней обеспечивается проведением математических доказательств истинности инвариантов. Следует отметить неадекватность терминологии. *Событие* (*event*) в Event-B включает действия, реализующие реакцию на событие, т.е. сопоставимо с требованием в предлагаемом нами подходе. По нашему мнению нет никаких оснований называть аксиомами ограничения на значения входных переменных модели.

Система управления лифтом рассматривается в руководстве [15] в качестве примера использования технологии Event-B. Определяется пять уровней:

- лифт без дверей и кнопок;
- добавление дверей лифта;
- добавление дверей на этажах;
- добавление кнопок в лифте;
- добавление кнопок на этажах, по одной кнопке на этаж.

На пятом уровне используется 17 событий и 8 переменных: номер текущего этажа, статус лифта (moving, stopped, idle), направление движения (up, down), статус дверей лифта (closed, opening, open, closing), статус дверей на этажах, массив кнопок лифта, подмножество нажатых кнопок, массив кнопок на этажах. В нашей программе используется 10 требований, две переменных (dir и floor) и три массива кнопок. Определяя более сложную функциональность, наша программа существенно проще. Переменные, определяющие статусы лифта и дверей, у нас не нужны, поскольку все действия с дверями локализованы в подпрограмме atFloor, работающей при остановленном лифте и завершающей работу с гарантией закрытия дверей. Подпрограмма atFloor появилась в результате гиперграфовой декомпозиции программы Lift, что обеспечивает ее простоту.

Уже модель первого уровня (без дверей и кнопок) в [15] оказалась значительно сложнее нашей реализации. Полная коллекция из пяти уровней является громоздкой. Технология Event-B [17] декларирует возможность экстракции исполняемой программы из коллекции моделей. Даже если это возможно, польза от такой программы сомнительна. Отметим, что в нашем подходе автоматная программа может быть оттранслирована в эффективную программу на язык C++.

Язык интервальной темпоральной логики, разработанный Джоном Рушби для верификации систем реального времени, демонстрировался для спецификации системы управления лифтом [27]. Спецификация в виде 31 темпоральной формулы достаточно

сложна; она намного сложнее представленной в данной работе. Работа [27] по интервальной логике не имела продолжений [1], поскольку появились более предпочтительные подходы к спецификации, например, диаграммы use case.

Разрабатываемая технология предикатного программирования до недавнего времени не имела в мире близких аналогов. Впервые появился язык, похожий на язык предикатного программирования P. Функциональный язык доказательного программирования Smart разработан французской компанией Prove&Run [2]. Корректность программ, а также отсутствие уязвимостей обеспечиваются дедуктивной верификацией. Язык Smart содержит конструкции, аналогичные гиперфункциям. Программа может быть оттранслирована на языки C и Java. Полное описание языка недоступно. Некоторые особенности языка Smart изложены в работах [18, 19].

В событийно-ориентированной парадигме [20, 21] 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 [22]. 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 статьях [1, 11, 23].

7. ВЕРИФИКАЦИЯ АВТОМАТНЫХ ПРОГРАММ

Валидация требований, по которым строится автоматная программа, заключается в их проверке на соответствие потребностям пассажиров лифта. Обычно здесь применяется моделирование. Результаты валидации оцениваются совместно разработчиком и заказчиком. Например, может оцениваться правильность решения о двух кнопках на каждом этаже вместо одной. Требование R1 также может быть предметом оценки и сопоставления с другими возможными стратегиями выполнения заявок.

Верификация автоматной программы реализуется относительно ее спецификации, в роли которой выступают инварианты управляющих состояний. Спецификацией являются также и требования. Однако, поскольку программа транслируется из требований, то здесь нет предмета для верификации.

Формальная верификация автоматных программ может быть реализована следующим образом. Автоматная программа преобразуется в эквивалентный набор логических формул в соответствии с формальной операционной семантикой предикатных программ [26]. Формула для процесса, например, Movement, строится в виде конъюнкции формул для требований, составляющих процесс. Вызов предикатной программы заменяется ее спецификацией, т.е. конъюнкцией предусловия и постусловия. Для каждого сегмента кода

[1] Одно исключение: работа [27] использовалась А.А. Калентьевым и А.А. Тюгашевым при построении собственного языка интервальной логики для спецификации программ реального времени.

[2] www.provenrun.com

(или требования) в качестве предусловия выступает инвариант исходного управляющего состояния для данного сегмента, а в качестве постусловия – инвариант следующего управляющего состояния. Если инварианты отсутствуют, верификация ограничивается проверкой истинности предусловий для всех используемых вызовов и операций. Например, для вызова примитива starting проверяется истинность предусловия dir = up ∨ dir = down. Дедуктивная верификация используемых предикатных программ реализуется методами, описанными в работах [5-7].

Инварианты управляющих состояний, если присутствуют, являются довольно слабыми, как, например, инвариант dir ≠ neutral. По этой причине верификация автоматной программы оказывается принципиально неполной. Проведение верификации не гарантирует отсутствия ошибок в автоматной программе. Не помогут также и дополнительные инварианты внутри сегментов кода. Факт неполноты верификации реактивных систем серьезно недооценивается во многих работах.

Модель в виде предусловия и постусловия для сегментов кода неадекватна, поскольку сегмент кода состоит из нескольких разнородных кусков и не представляет цельной программы [23].

Отмечается, что формальная верификация программ реактивных систем, построенных по функциональным требованиям, весьма трудоемка, а число обнаруживаемых ошибок, в дополнении к найденным при моделировании, ничтожно.

Объектами верификации могут быть также *свойства* автоматной программы, обычно формулируемые на языке темпоральной логики. Свойства программы управления лифтом определяются исходя из основного назначения лифта – перевозить пассажиров лифта с одного этажа на другой. Главное свойство: любая заявка, инициируемая нажатием кнопки, должна быть обслужена, разумеется, при условии, что кнопка не будет отжата. Точнее, это свойство формулируется следующим образом. Если для некоторого этажа j нажата одна из кнопок на этаже или соответствующая кнопка в кабине лифта, то через определенное время лифт гарантированно подойдет к этажу j и откроет двери. Формально данное условие записывается формулой:

$$\Box \forall \text{ FLOOR } j.\ (Up[j] \vee Down[j] \vee Cab[j]) \ \&\ (\Box \neg release) \Rightarrow \Diamond\ floor = j\ \&\ open$$

Здесь «□» и «◊» - темпоральные кванторы. Формула вида □A определяет следующее утверждение: в любой момент работы лифта будет истинной формула A. Формула вида ◊A есть утверждение: через конечный промежуток времени работы лифта в будущем будет истинна формула A.

Свойства программы на языке темпоральной логике могут быть верифицированы с помощью инструментов проверки на модели (model checking). При этом автоматная программа должна быть оттранслирована в эквивалентный набор логических формул в соответствии с формальной операционной семантикой [26].

ЗАКЛЮЧЕНИЕ

Программа управления работой лифта нетривиальна. Являясь автоматной программой, она содержит фрагменты, относящиеся к классу программ-функций, для построения которых нужно применять другие методы, отличные от методов автоматного программирования. Другая особенность – сложность внешнего окружения. Для упрощения программы применяется объектно-ориентированный подход. Взаимодействие с окружением реализуется через интерфейс класса, а детали окружения спрятаны внутри класса.

Для понимания программы Lift наиболее подходящим является ее описание на языке требований в виде трех процессов Lift, Movement и atFloor. Построение программ примитивов, таких как decisionIdle и check_floor, проводится в технологии предикатного программирования; автоматные программы таких примитивов оказались бы существенно сложнее соответствующих предикатных программ.

Представленная в настоящей статье программа управления лифтом существенно проще и короче программ в других работах [25, 12-15], демонстрирующим различные технологии. Есть три составляющие, обеспечивающие простоту нашей программы Lift: гиперграфовая автоматная композиция в сочетании с механизмом гиперфункций, объектная ориентированность и реализация примитивов вне автоматной программы на языке P. Отметим, что в оценке сложности программы критичной является сложность автоматной части программы.

Работа выполнена при поддержке РФФИ, грант № 16-01-00498.

Список литературы

1. 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
2. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования Р. – Новосибирск, 2010. – 42с. – (Препр. / ИСИ СО РАН; N 153).
3. Cockburn A. Writing Effective Use Cases / Addison-Wesley. – 2001. – 270 P.
4. Systems and software engineering — Life cycle processes — Requirements engineering. ISO/IEC/ IEEE 29148. URL: http://datateca.unad.edu.co/contenidos/204019/EConocimiento/M11-IEEE_29148-2011.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, 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. Шелехов В.И. Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования. – Новосибирск, 2012. – 30с. – (Препр. / ИСИ СО РАН. № 164).
8. --------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.
9. 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.
10. Шелехов В.И. Предикатное программирование. Учебное пособие. Новосибирск: НГУ, 2009. 109с.
11. Шелехов В.И. Оптимизация автоматных программ методом трансформации требований // «Программная инженерия», №11, 2015. – С. 3-13. http://persons.iis.nsk.su/files/persons/pages/req_k.pdf
12. Наумов А.С., Шалыто А.А. Система управления лифтом. Санкт-Петербург, 2003. 51с. http://is.ifmo.ru/download/elevator_a.pdf
13. 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
14. Кнут Д.Э. Искусство Программирования. – Том 1. Основные Алгоритмы. – 2006. разд. 2.2.5.
15. Robinson K. System Modelling & Design. Using Event-B. Draft book. – 2012. – 142 P. http://www.cse.unsw.edu.au/~cs2111/PDF/SMD-KAR.pdf
16. Манухин С.Б., Нелидов И.К. Устройство, техническое обслуживание *и* ремонт лифтов. М. «Академия», 2004. – 336 с.
17. Abrial J.-R. Modeling in Event-B: System and Software Engineering. Cambridge University Press. – 2010. – 586 P.
18. Andreescu O.F., Jensen T., Lescuyer S. Dependency Analysis of Functional Specifications with Algebraic Data Structures // ICFEM'15. LNCS 9407, 2015. P. 116-133.
19. Lescuyer S. Towards a verified isolation micro-kernel // MILS: Architecture and Assurance for Secure Systems, Amsterdam, 2015. 8 P.
20. Vlissides J., Johnson R., Helm R., Gamma E. Design Patterns: Elements of Reusable Object-Oriented Software / Addison-Wesley. – 1994. – 431 P.
21. Ferg S. Event-Driven Programming: Introduction, Tutorial, History / SourceForge. – 2006. – 59 P.
22. 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.

23. Шелехов В.И. Разработка автоматных программ на базе определения требований // Системная информатика, №4, 2014. — ИСИ СО РАН, Новосибирск. — С. 1-29. http://persons.iis.nsk.su/files/persons/pages/req_tech.pdf
24. Klahr D., Langley P., Neches R. Production System Models of Learning and Development. – Cambridge, Mass.: The MIT Press. 1987. – 467 P.
25. Гома Х. UML. Проектирование систем реального времени, параллельных и распределенных приложений. М.: ДМК Пресс, 2002. 684 с.
26. Шелехов В.И. Семантика языка предикатного программирования // ЗОНТ-15. — Новосибирск, 2015. — 13с. http://persons.iis.nsk.su/files/persons/pages/semZont1.pdf
27. Rushby J. Specifying real-time systems with interval logic (SuDoc NAS 1.26:181804). SRI International. Menlo Park, CA, USA, 1988. – 81 P.
28. Mich L, Franch M, Novi Inverardi P. Market research for requirements analysis using linguistic tools. Requirements Engineering 9(1). – 2004. – P. 40–56.
29. Шелехов В.И. Классификация программ, ориентированная на технологию программирования // «Программная инженерия», Том 7, № 12, 2016. — С. 531–538.
30. Шалыто А. А. SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб: Наука, 1998. http://is.ifmo.ru/books/switch/1

Приложение 1
Особенности реализации программы управления лифтом

В Приложении 1 приводятся программы примитивов класса Лифт, полученные оптимизирующей трансформацией предикатных программ на императивное расширение языка P, с которого программа легко кодируется на язык C++.

В конце Приложения 1 определены особенности компоновки итоговой программы.

```
decisionIdle( : #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 ;
  }
}

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)};

decisionClosed( 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 decisionIdle, decisionClosed 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 правило нежелательности смешения стилей и эргономическое правило, 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.