Автоматное программирование

Автоматное программирование (прод.)

Примеры: Гадание на кофейных зернах.

Программа управления лифтом.

Верификация автоматных программ

Коммуникационные протоколы

Протокол чередования битов

Модель программы управления полетом спутника qXz

Автоматные трансформации. Протокол AAL-2

Колония автономных роботов

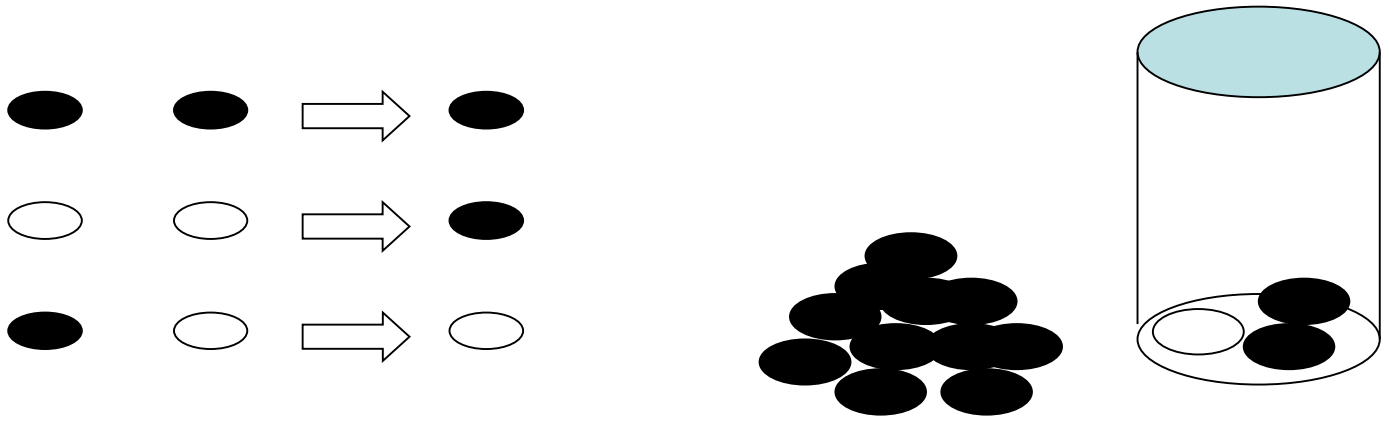
Автоматное программирование

1.4. Примеры

http://persons.iis.nsk.su/files/persons/pages/lift1.pdf

http://persons.iis.nsk.su/files/persons/pages/ctrlspacecraft.pdf

Пример 4. Гадание на кофейных зернах



Содержательное описание. Зерна: черные и белые в стакане, который вначале не пуст. Запасная кучка черных зерен. Из стакана выбирается два зерна, одно возвращается. В конце останется одно зерно. Какое?

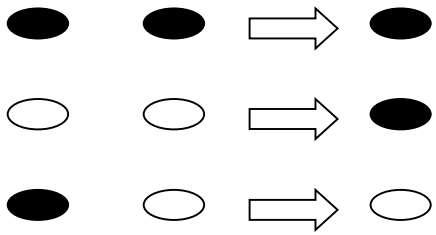
Состояние. nat b, w; //b — число черных зерен, w — число белых зерен

Окружение. Операция выбора пары зерен:

bb_ww_bw(nat b, w : #bb : #ww : #bw)

pre b+w ≠ 0 pre bb: b>1 pre ww: w>1 pre bw: b ≠ 0 & w ≠ 0

недетерминированность



Функциональные требования

Ф1: (b, w) = (1, 0) → Результат black **Выход**

Ф2: (b, w) = (0, 1) → Результат white **Выход**

Ф3: Взятие двух черных → Вернуть один черный

Ф4: Взятие двух белых → Вернуть один черный

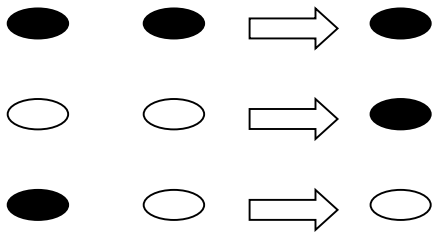
Ф5: Взятие белого и черного → Вернуть один белый

Уточнение требований

Ф3: b > 1, Выбор двух черных → b’ = b - 1

Ф4: w > 1, Выбор двух белых → w’ = w – 2, b’ = b + 1

Ф5: b > 0, w > 0, Выбор белого и черного → b’ = b - 1



Сегменты и их инварианты. step: inv b + w ≠ 0;

bb: inv b>1; ww: inv w>1; bw: inv b ≠ 0 & w ≠ 0;

Локальная программа.

choice(nat b, w: #black : #white : #bb : #ww : #bw)

pre b+w ≠ 0 pre black: b=1 & w=0 pre white: b=0 & w=1

pre bb: b>1 pre ww: w>1 pre bw: b ≠ 0 & w ≠ 0

process Гадание(: #black : #white) pre b+w ≠ 0 {

step: choice(b, w: #black : #white : #bb : #ww : #bw)

bb: b’ = b - 1 #step

ww: w’ = w – 2, b’ = b + 1 #step

bw: b’ = b - 1 #step

}

```
hyper  choice(nat b, w:  #black : #white : #bb : #ww : #bw )
 pre b+w ≠ 0 pre black: b=1 & w=0 pre white: b=0 & w=1
 pre bb: b>1 pre ww: w>1 pre bw: b ≠ 0 & w ≠ 0
{ if (b=1 & w=0) #black
  else if (b=0 & w=1) #white
  else bb_ww_bw(b, w : #bb : #ww : #bw)
}
```

Дополнительное свойство на переменных состояния процесса:

$(b' = b - 1 \text{ or } b' = b + 1) \ \& \ (w' = w \text{ or } w' = w - 2)$

Процесс всегда завершается.

Мера на состоянии процесса: nat m(nat b, w) = b + w;

```
Гадание(nat b, w: #black : #white) pre b + w ≠ 0
    pre black:  ?????;
    pre white:  ?????;
```

```
process Гадание(nat b, w: #black : #white) ≡
    pre b + w ≠ 0
    pre black:  even(w);
    pre white: ¬even(w);
```

Пример 5. Управление 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 две *кнопки* вызова лифта для движения *вверх* и *вниз*. На *нижнем* этаже нет кнопки движения вниз, а на *верхнем* – для движения вверх.

Внутри *кабины лифта* кнопки с номерами этажей.

Нажатие кнопки определяет остановку 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 пока существуют заявки в этом направлении.
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. Лифт останавливается при наличии заявки по данному этажу, но не в противоположном направлении движению лифта. *Решение об остановке* на этаже принимается заранее вблизи этажа по специальным датчикам.

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

Объекты. Лифт, этажи, кнопки на этажах и в лифте

Операции лифта: движение от текущего этажа, остановка на этаже, открытие дверей стоящего на этаже лифта, закрытие дверей лифта.

Состояние. Текущий этаж, направление движения, нажатые кнопки на этажах и в лифте.

Функциональные требования.

Ф1: Если лифт остановлен и на текущем этаже нажимается кнопка (в лифте или на этаже), то двери лифта открываются.

Ф2: Если лифт остановлен и нажаты кнопки (в лифте или на этажах) и нет нажатых кнопок на текущем этаже, то лифт начинает движение в одном из направлений, где есть нажатые кнопки.

Ф3: Если двери лифта открыты, то через промежуток времени **Tdoor** или при нажатии кнопки «закрыть дверь» двери лифта закрываются.

Функциональные требования.

Ф4: Если двери лифта закрываются и обнаружены помехи при закрытии дверей или при нажатии кнопки «открыть дверь» то двери лифта снова открываются.

Ф5: Если двери лифта закрыты (только что) и нет нажатых кнопок (в лифте и на этажах), то лифт стоит.

Ф6: Если двери лифта закрыты (только что) и есть нажатые кнопки (в лифте или на этажах) на этажах кроме текущего, то лифт начинает движение в одном из направлений, где есть нажатые кнопки.

Ф7: Если лифт движется вблизи очередного этажа и нет нажатых кнопок (в лифте и на этажах) в направлении движения лифта от текущего этажа, то на очередном этаже лифт останавливается.

Функциональные требования.

Ф8: Если лифт движется вблизи очередного этажа и есть нажатые кнопки (в лифте или на этажах) в направлении движения лифта после очередного этажа и для очередного этажа отжаты кнопка в лифте и кнопка на этаже в направлении по ходу лифта, то лифт пропускает очередной этаж.

Ф9: Если лифт движется вблизи очередного этажа и есть нажатые кнопки на очередном этаже, то лифт останавливается на очередном этаже. Исключение: для кнопки в противоположном направлении при наличии других нажатых кнопок.

Стратегии.

Минимизировать среднее время ожидания

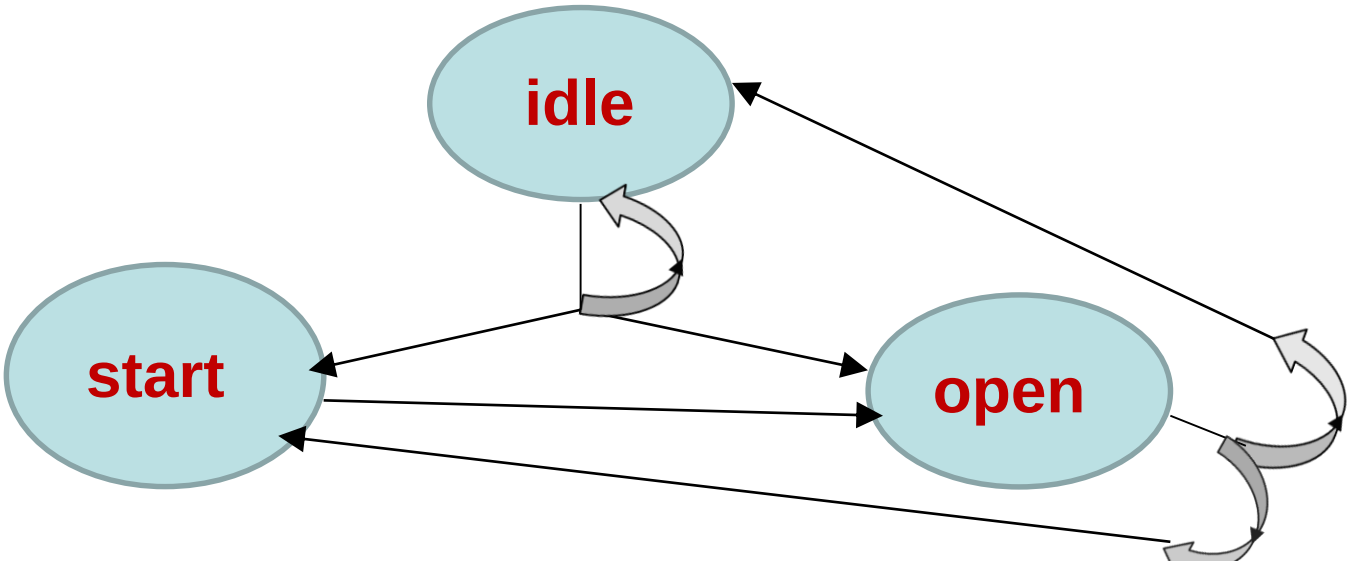
Окружение. Класс Лифт

```
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);  //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 «закрыть дверь»
bool closedDoor(); // = true, если закрытие дверей лифта завершено
bool blockedDoor(); // = true, закрытие дверей лифта остановлено
          //  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 ....
}
```

Состояние. Объект класса Лифт.
Сегменты и инварианты программы Lift:

idle: **inv** dir = neutral; // лифт стоит на этаже, двери закрыты
start: **inv** dir ≠ neutral; // лифт начинает движение в напр. dir
open; // лифт подошел к этажу или уже стоит на этаже

```
process Lift {
    idle → decision1( : #idle  : #start : #open);
    start → Movement( : #open);
    open → atFloor( : #idle  : #start)
}
```



Сегменты и инварианты программы Movement:

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

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

Сегменты программы atFloor:

```
open;      // лифт стоит (или остановился)
              //на некотором этаже с закрытыми дверями
opened;    // двери лифта находится в полуоткрытом состоянии;
             //ожидается повторное открытие дверей
close;     // двери лифта закрывается
```

```
process atFloor( : #idle  : #start) {
 open → openDoor(), set t, opened;
 opened, closeButton() or t ≥ Tdoor → closeDoor(), close;
 close, closedDoor() → decision2( : #idle  : #start);
 close, blockedDoor() → open;
}
```

Автоматная программа управления лифтом

```
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 (closedButton() or t ≥ Tdoor) {closeDoor(); #close}
           #opened
 close:    if (closedDoor()) decision2( : #idle  : #start);
           if (blockedDoor()) #open;
           #close
}
```

Реализация класса лифт

type BUTTONS = **array** (**bool**, FLOOR);
BUTTONS Up, Down, Cab;

Нажатие (press) и отжатие (release) кнопки:

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

При открытии двери лифта:

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

Верификация и проверка свойств

Если на этаже j нажата одна из кнопок, то через определенное время лифт гарантированно подойдет к этажу j и откроет двери.

□ ∀ FLOOR j. (Up[j] ∨ Down[j] ∨ Cab[j]) & (□ ¬release)
⇒ ◊ floor = j & open

Верификация автоматных программ

Верификация программы-функции П:

Тройка Хоара { P(x) } П { Q(x, y) }. Логика Хоара-Флойда.

Спецификация реактивной системы:

инварианты сегментов и инварианты секций.

Инвариант секции: инвариант-требование, программный инвариант

Инварианты на языке темпоральной логики

Инвариант сегмента истинен в начале своего сегмента

Инвариант локальной секции истинен в начале каждого сегмента автоматной программы

Инвариант глобальной секции истинен в начале каждого сегмента реактивной системы

П – сегмент одной из автоматных программ.

x – набор переменных состояния автоматной программы

I(x) – конъюнкция всех инвариантов локальной и глобальной секций.

wdП(x) – предусловие сегмента П

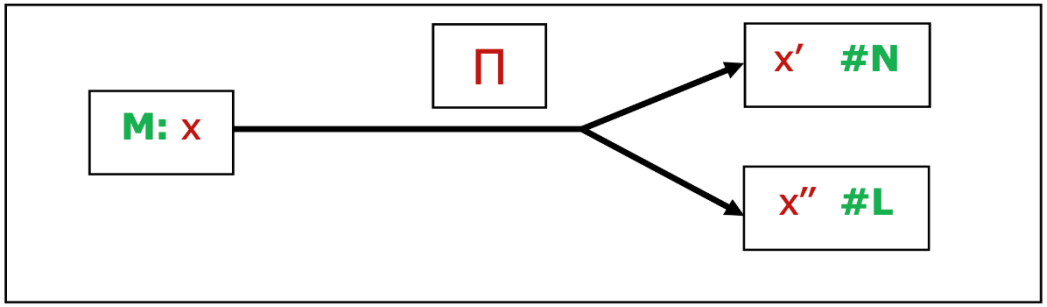
Формулы корректности:

1. Правильность (well-definedness): $I(x) \Rightarrow wdП(x)$

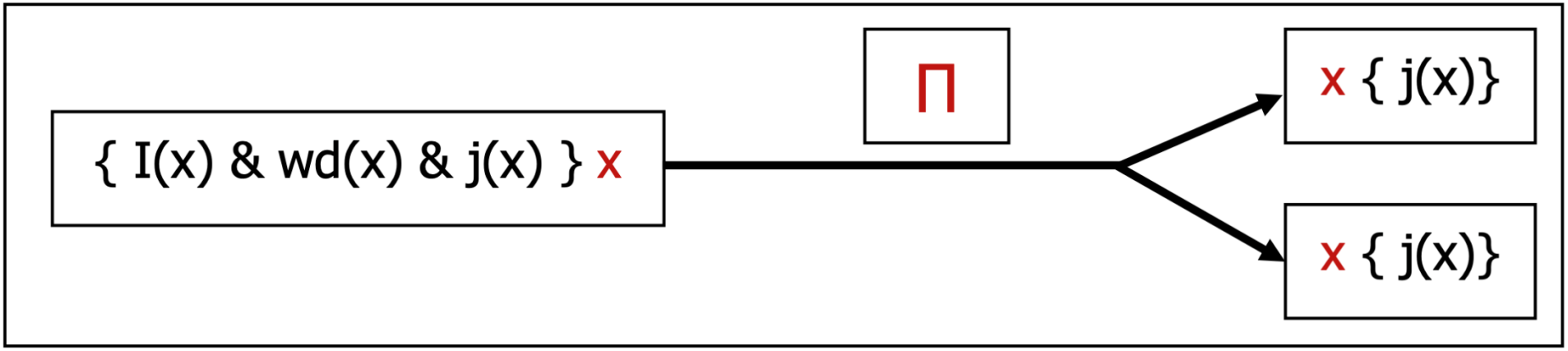
j(x) – один из инвариантов локальной или глобальной секции

2. В начале работы реактивной системы: $\vdash j(x_0)$

3. Сегмент П сохраняет инвариант j(x)



Гипертройка Хоара:



4. Завершимость конечных процессов.

5. Отсутствие дедлоков livelock

Метод верификации на базе тотального инварианта TI(x)

Автоматное программирование

Протоколы. Автоматные трансформации

persons.iis.nsk.su/files/persons/pages/req_k.pdf

Коммуникационные протоколы

Протокол — стандарт, определяющий взаимодействие двух или более объектов (***узлов***) — соединение, аутентификацию, кодирование и передачу данных, …

Коммуникационный протокол – стандарт, определяющий правила передачи информации между двумя удаленными узлами

Пример 6. Протокол чередования битов

(Alternating Bit Protocol, ABP)

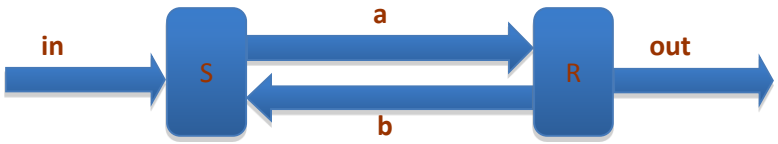
Содержательное описание. Передача данных через *ненадёжные* каналы связи. Процесс *передатчик* S вводит из внешнего окружения потенциально бесконечную последовательность блоков данных сообщением in(d), где d – блок данных. Передатчик S пересылает очередной блок d сообщением a(n, d), где n – номер блока, другому параллельно функционирующему процессу – *приемнику* R. Получив сообщение a(n, d), приемник R выводит блок d во внешнее окружение с помощью сообщения out(d). Каналы ненадежные: сообщения м.б. потеряны, но не испорчены. Передатчик ожидает подтверждения получения приемником очередного блока: получив сообщение b(n), передатчик посылает следующий блок. Если сообщение b(n) не получено по истечении времени Twait, очередной блок d посылается повторно.

Протокол n-ABP. m – номер блока. Синхронизирующий бит.

```
section { type Data;
          message a(nat, Data), b(nat);
}
process ABP  { S || R }

section { message in(Data);
          bool sa; // непредсказуемо меняются во времени
          nat n = 0; time t; Data d
}
process S {… }

section { message out(Data);
          bool sb; // непредсказуемо меняются во времени
          nat m = 0; Data d
}
process R {… }
```




```
section { type Data;  message a(nat, Data), b(nat); }
process ABP()  { S || R }

process S() {
  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() {
    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() {
  section { nat n = 0; bool sa; DATA d; time t}
  s1:    receive  in(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() {
     section {nat m = 0; bool sb }
  r1:    receive a(nat i, DATA d);
         if (i = m) { send out(d); m = m+1};
         if (sb)  send b(i);
         #r1
}
```

n = n+1 заменить на n = ¬n, а m = m+1 – на m = ¬m

Нужны ли детальные поименованные требования?
Нужны более емкие требования.

Ф1: Всякий блок, введенный в Передатчике, должен быть через определенное конечное время доставлен и выведен Приемником. Проверяемость?

Требование надежности Ф2: В любой момент времени последовательность блоков, введенная в Передатчике должна совпадать с последовательностью блоков, выведенной в Приемнике, за возможным исключением последнего введенного блока.

Требование достижимости Ф1': Если в Передатчике введен блок с номером n, то через определенное конечное время блок с номером n должен быть доставлен и выведен Приемником.

Протокол скользящего окна

Блоки посылаются друг за другом без ожидания подтверждения с записью в окно. Если требуется повторить передачу, этот блок берется из окна. Число блоков, посылаемых в текущий момент, регулируется размером окна. Заводится таймер на каждый блок

окно - круговой буфер блоков

Модель программы управления полетом спутника qXz

Модельно-ориентированный подход
(model-driven engineering)

Слой – независимая часть модели

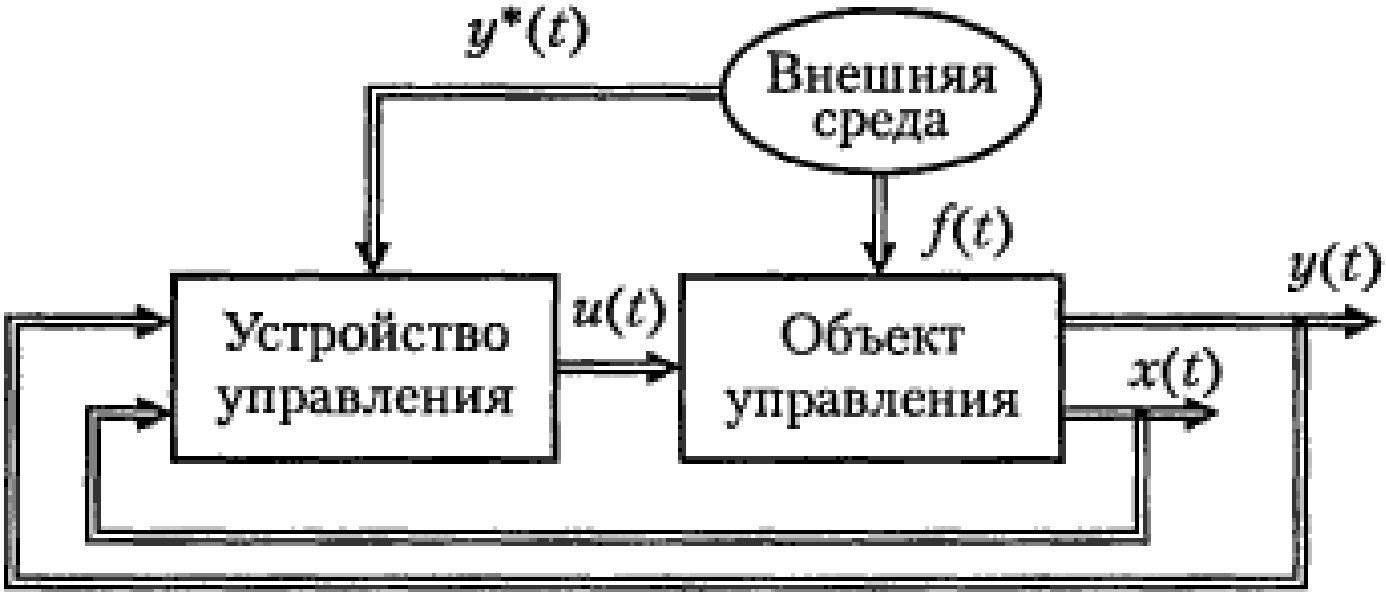
Модель строится последовательным добавлением новых слоев

Пошаговое построение модели ПО

Модель бортовой программы управления спутником в виде набора слоев (аспектов).

- Модель простейшей системы управления
- Модель отказоустойчивой системы управления
- Модель управления летательным аппаратом (+ слои интеграции автоматического и ручного управления, мониторинга и защиты от несанкционированного доступа)
- Модель ПО космического аппарата (КА) + слои управления движением КА, поддержки работы служебных систем (энергообеспечение, терморегулирование и др.) и разнообразных сервисов, мониторинг всех процессов и подсистем КА, взаимодействие с наземным комплексом управления

Модель системы управления



next(in) → Step(in, s: s’, com), control(com)

s – состояние системы управления

Сообщение next(in) – запускает работу контроллера

Step – очередной шаг работы контроллера

control – управляющее воздействие на объект управления

wait T, Step(in, s: s’, com), control(com)

Отказоустойчивая модель системы управления

run, next(in) → Step(in, s: s', com: errC #alarm),
control(com)

alarm → Восстановление(s, errC : #run : err #exit)

Сегменты:

run; // основной (штатный) режим работы контроллера

alarm; // аварийный режим – восстановление

exit; // аварийный выход из контроллера

errC – код ошибки

s – состояние системы управления

Сообщение next(in) – запускает работу контроллера

Step – очередной шаг работы контроллера

control – управляющее воздействие на объект управления

==========

next(in) → Step(in, s: s', com), control(com)

Структура бортовой программы

Сочетание жесткого и мягкого реального времени под управлением ОС реального времени. *Циклограмма*

Штатный режим:

```
process _qXz {
    ОсновнаяМиссия ||
    УправлениеПолетом ||
    Функционирование ||
    Мониторинг ||
    УправлениесЗемли
}
```

Интерфейс от каждой подсистемы Функционирования.

Жизнеобеспечение аппаратуры процесса ОсновнаяМиссия реализуется через интерфейсные объекты подсистем Функционирования.

Реализация мониторинга требует нетривиальной модификации всех других подсистем.

Модель процесса управления полетом

В модели определяется *состояние*, локальные программы, *сегменты* и описание процесса на языке правил.

Состояние процесса УправлениеПолетом.

S s; // состояние спутника: время, координата центра масс спутника в инерционной системе координат, направление полета, скорость и угловая скорость спутника на момент последнего сеанса навигации; S – тип переменной s

time Tw; // время ожидания следующего сеанса навигации

Модель процесса управления полетом

Локальные программы.

Ориентация(: s #out : s #run : errC #alarm) – определяет координаты спутника в инерциальной системе координат, направление полета, скорость и угловую скорость. Их значения формируются параметром-результатом s. При незначительном отклонении реализуется выход run, иначе out.

Маневр(s : s', Tw : errC #alarm) – реализация маневра для выхода на целевую орбиту. Новое значение состояния спутника s' учитывает изменения высоты и наклонения орбиты.

Коррекция(s : s', Tw : errC #alarm) – реализуется коррекция незначительного отклонения от орбиты.

Восстановление(s, errC : #start : err #exit) – на основе анализа текущего состояния спутника s и кода ошибки errC реализуется процесс восстановления штатного режима полета спутника. В ситуации, когда восстановление невозможно, реализуется выход exit с кодом ошибки err

Сегменты:

start – начало очередного цикла ориентации спутника;
out – спутник пока вне орбиты;
run – штатный режим движения спутника по орбите;
alarm – режим восстановления после аварийной ситуации;
exit – внешний аварийный выход.

Описание процесса на языке правил.

```
process УправлениеПолетом( : err #exit) {
    start →  Ориентация(: s #out: s #run: errC #alarm);
    out   →  Маневр(s : s', Tw : errC #alarm),
             wait Tw, start;
    run   →  Коррекция(s : s', Tw : errC #alarm),
             wait Tw, start;
    alarm → Восстановление(s, errC : #start : err #exit)
}
```

Автоматные трансформации

1. Эквивалентные замены

move(buf, x : buf’, x’) →
move(buf, x: buf’ #G: buf’ #G: buf’, x’ #G); G:

2. Специализация

H: S → H1: **inv** e(x); S
H2: **inv** ¬e(x); S

3. Редукция суперпозиции

H: B(s: s’); C → H: B(s: s’) #K
K: **inv** e(s); C

Пример 7. Протокол AAL-2

https://persons.iis.nsk.su/files/persons/pages/req_k.pdf

Уровень адаптации 2 (*AAL type 2*) в асинхронном способе передачи данных ATM (*Asynchronous Transfer Mode*) для эффективной передачи низкоскоростных, коротких пакетов переменной длины в приложениях, чувствительных к задержкам.

Уровни передачи данных:

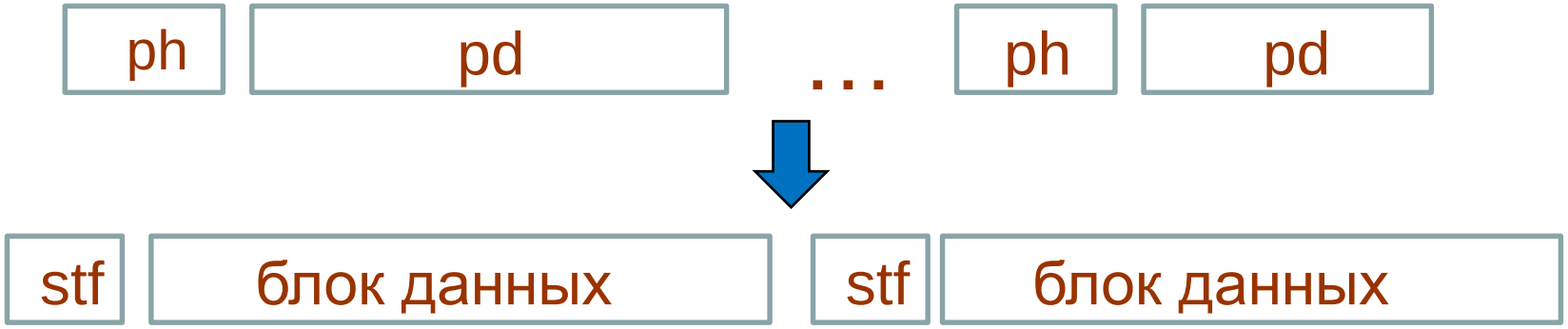
- уровень конвергенции SSCS (*Service Specific Convergence Sublayer*)
- общий подуровень CPS (*Common Part Sublayer*)
- уровень ATM

SSCS: пакеты переменной длины

CPS: блоки длиной 47 октетов, начальное поле **STF**

Схема: получить пакет **pd** → создать заголовок **ph** → **ph + pd** дописать в последовательность блоков → очередной блок с полем **STF** передать на уровень ATM

Особенности: плотная упаковка, механизм сообщений, сброс незаполненного блока по таймеру, передача блока по запросу



Протокол AAL-2

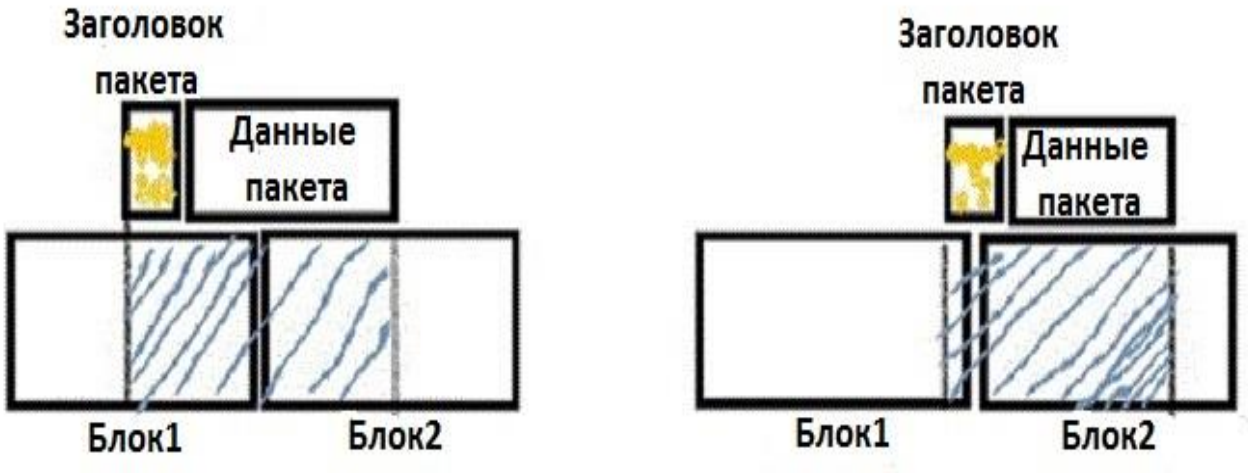


Схема переноса пакета

a – перенос данных; b – перенос заголовка

Пакеты переменной длины, блоки

Требования надежности и достижимости.

Ф1: В любой момент времени доставленная последовательность блоков должна совпадать информационно (за вычетом заголовков) с введенной последовательностью пакетов, за возможным исключением нескольких последних пакетов.

Ф2: Всякий введенный пакет должен быть быстро (менее чем за оговоренное время) доставлен при наличии запроса на доставку. Если ввод пакетов прерван, очередной блок дополняется нулями чтобы быть вовремя доставленным.

Ф3: Очередной блок выводится только при поступлении запроса на доставку.

Недопустимы потери быстродействия в протоколах

Окружение: CPSpacket(pd), SEND_request, ATM_data(block).

Пакет и блок данных имеют тип DATA – список октетов:

type OCTET = **byte**;

type DATA = list(OCTET);

Состояние: DATA packs = nil, OCTET stf = ConstructSTF(0).

Локалы: DATA pd, block.

Локальные программы.

pred extract_block(DATA packs**:** DATA block, packs')

pre packs ≥ 47 **post** packs = block + packs' & len(block) = 47;

Правила: (1)

len(packs) < 47, CPSpacket(pd) →

set_Timer_CU,

packs’ = packs + ConstructCPS_PacketHeader(pd)+ pd;

len(packs) ≥ 47, SEND_request() →

extract_block(packs**:** block, packs’),

ATM_data(stf + block),

if (packs’≠nil) set_Timer_CU,

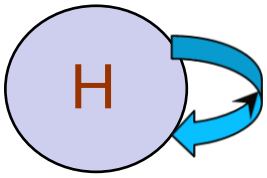
stf’ = ConstructSTF(min(len(packs’), 47));

0 < len(packs) < 47, timer(), SEND_request() →

ATM_data(stf + fill(packs)),

packs’ = nil,

stf’ = ConstructSTF(0).



Трансформация 1. Заменим packs на buf + ph + pd в требованиях (1). Корректность трансформации следует из того, что условие packs = buf + ph + pd всегда истинно.

Состояние: DATA buf = nil, ph = nil, pd = nil,
OCTET stf = ConstructSTF(0)

Локальные программы. Меняется программа выделения очередного блока:

pred extract_block(DATA buf, ph, pd**:** DATA buf', ph', pd')

pre buf + ph + pd ≥ 47

post buf + ph + pd = buf' + ph' + pd' & len(buf')=47;

Правила: (2)

len(buf + ph + pd) < 47, CPSpacket(pd') →

set_Timer_CU, buf' = buf + ph + pd,

ph' = ConstructCPS_PacketHeader(pd');

0 < len(buf + ph + pd) < 47, timer(), SEND_request() →

ATM_data(stf + fill(buf + ph + pd)),

buf' = nil, ph' = nil, pd' = nil, stf' = ConstructSTF(0);

len(buf + ph + pd) ≥ 47, SEND_request() →

extract_block(buf, ph, pd: buf', ph', pd'),

ATM_data(stf + buf'),

if (ph' + pd' ≠ nil) set_Timer_CU,

buf'' = nil;

stf' = ConstructSTF(min(len(ph' + pd'), 47)).

Применение трансформации редукции суперпозиции.

Локальные программы.

pred move(DATA buf, x**:** DATA buf’, x’)

pre len(buf) ≤ 47

post buf’ + x’ = buf + x &

(len(buf + x) ≥ 47 ? len(buf’)=47 **:** x’=nil);

Правила: (3)

H → move(buf, ph**:** buf’, ph’), move(buf’, pd**:** buf’’, pd’), K; (3.1)

K, len(buf) < 47, CPSpacket(pd) → (3.2)

set_Timer_CU, ph = ConstructCPS_PacketHeader(pd), H;

K, 0 < len(buf) < 47, timer(), SEND_request() → (3.3)

ATM_data(stf + fill(buf)),

buf’ = nil, ph’ = nil, pd’ = nil, stf’ = ConstructSTF(0), H;

K, len(buf) = 47, SEND_request() → (3.4)

ATM_data(stf + buf),

if (ph + pd ≠ nil) set_Timer_CU,

buf’ = nil; stf’ = ConstructSTF(min(len(ph + pd), 47)), H.

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

pre len(buf) ≤ 47 // общее предусловие

pre 1**:** len(buf+x) < 47

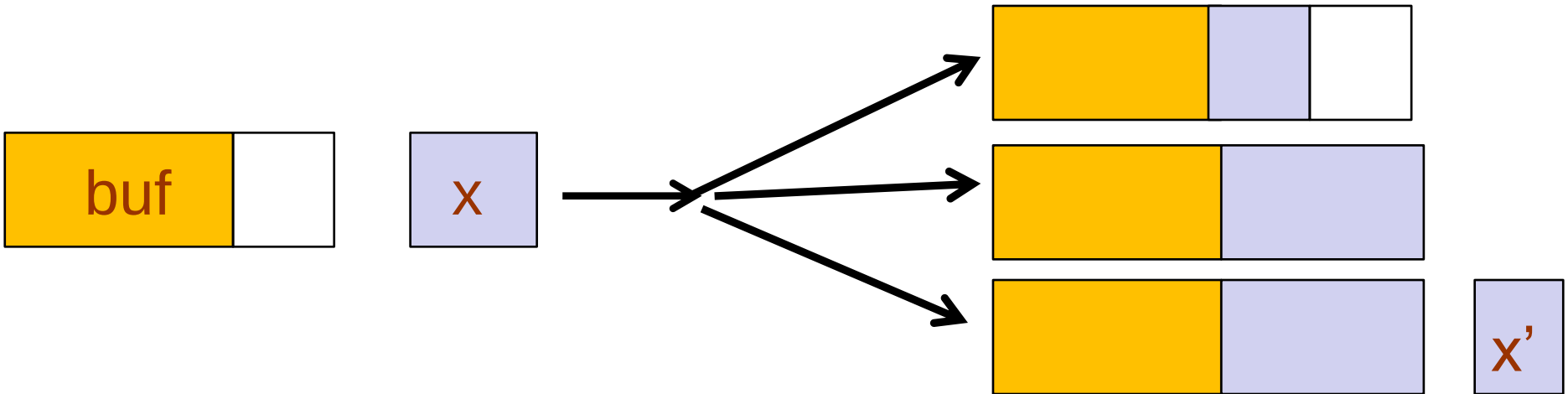
pre 2**:** len(buf+x) = 47

pre 3**:** len(buf+x) > 47

post 1**:** len(buf’) < 47 & buf’ = buf + x

post 2**:** len(buf’) = 47 & buf’ = buf + x

post 3**:** len(buf’) = 47 & buf + x = buf’ + x’ & x’≠nil;



Трансформация 2. В требовании (3.1) заменим вызовы программы move на эквивалентные вызовы гиперфункций.

Трансформация 3. Введем новые сегменты с инвариантами:

PART: **inv** len(buf) < 47 & ~ph & ~pd;

B1: **inv** len(buf) = 47 & ~ph & ~pd; // ph и pd полностью переписаны

B2: **inv** len(buf) = 47 & ~ph; // ph полностью переписан

B3: **inv** len(buf) = 47 & ph'≠nil; // ph переписан частично

Трансформация 4. Проведем специализацию требования (3.4) для сегментов B1, B2 и B3, а также требований (3.2) и (3.3) для сегмента PART.

Трансформация 5. Проведем уточнение итоговых сегментов во всех требованиях

Правила: (4)

PART, CPSpacket(pd) →

set_Timer_CU, ph = ConstructCPS_PacketHeader(pd), H;

H → move(buf, ph: buf’ #G : buf’ #B2: buf’, ph’ #B3);

G → move(buf, pd: buf’ #PART : buf’ #B1 : buf’, pd’ #B2);

PART, buf≠nil, timer(), SEND_request() →
ATM_data(stf + fill(buf)),
buf’ = nil, stf’ = ConstructSTF(0), PART;

B1, SEND_request() → ATM_data(stf + buf),
buf’ = nil; stf’ = ConstructSTF(0), PART;

B2, SEND_request() → ATM_data(stf + buf); set_Timer_CU,
buf’ = nil; stf’ = ConstructSTF(min(len(pd), 47)), G;

B3, SEND_request() → ATM_data(stf + buf), set_Timer_CU,
buf’ = ph; stf’ = ConstructSTF(min(len(ph + pd), 47)), G.

```
process transfer() {
 PART:      inv len(buf) < 47 & ~ph & ~pd;
            if (CPSpacket(pd)) {
                set_Timer_CU();
                ph = ConstructCPS_PacketHeader(pd);
                move(buf, ph : buf' #G: buf' #B2 : buf', ph' #B3)
 G:             inv len(buf) < 47 & ~ph;
                move(buf, pd : buf #PART: buf' #B1: buf', pd' #B2)
            } else if (buf≠nil & timer() & SEND_request()) {
                send ATM_data(stf + fill(buf));
                buf = nil; stf = ConstructSTF(0)    #PART
            }
 B1:        inv len(buf) = 47 &~ph & ~pd;
            receive SEND_request();
            send ATM_data(stf + buf); stf' = ConstructSTF(0);
            buf =nil  #PART
 B2:        inv len(buf) = 47 &~ph;
            receive SEND_request();
            send ATM_data(stf + buf); set_Timer_CU();
            buf =nil; stf' = ConstructSTF(min(len(pd), 47))  #G
 B3:        inv len(buf) = 47 & ph≠nil;
            receive SEND_request();
            send ATM_data(stf + buf); set_Timer_CU();
            buf = ph; stf' = ConstructSTF(min(len(ph + pd), 47));  #G
}
```

8. Колония автономных роботов

Площадка размера NxN с непреодолимыми стенками.
E1: *Объект A* – параллелепипед с квадратной базой
E2: *Объект-цель B* – цилиндр с диаметром Rb, высотой Hb.
E3: *Роботы C1, C2 , … Ck* цилиндрической формы
E4: Все объекты расположены на плоскости своим основанием. Роботы расположены произвольно.

Цель – переместить объект A до касания цели B

https://www.sheffield.ac.uk/naturalrobotics/supp/2012-004