

# Математические методы верификации схем и программ

Лекторы:

Захаров Владимир Анатольевич

Подымов Владислав Васильевич

е-mail рассказчика:

**valdus@yandex.ru**

Осень 2018

# Семинар 7

Spin  
(обзор средства)

# Рассматриваемая ЗАДАЧА

*Даны*

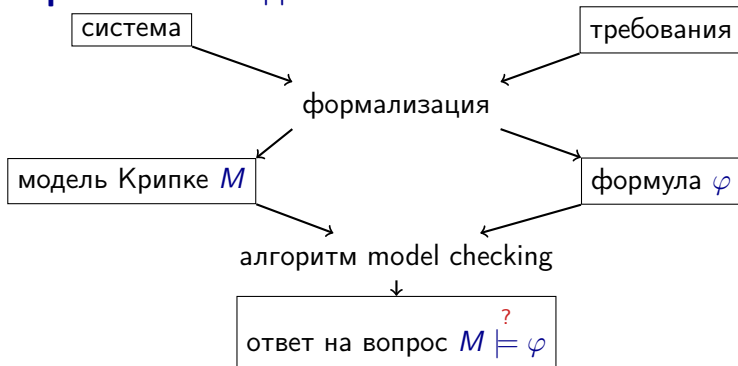
- ▶ неформальное описание системы
- ▶ содержательное описание требований к системе

*Требуется проверить,*

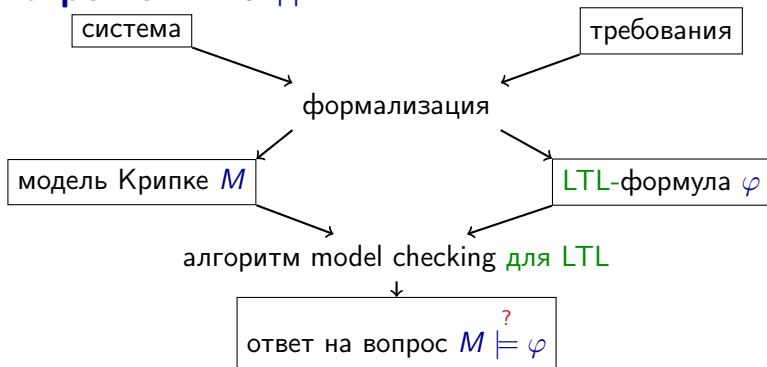
удовлетворяет ли система набору требований



# Схема решения ЗАДАЧИ

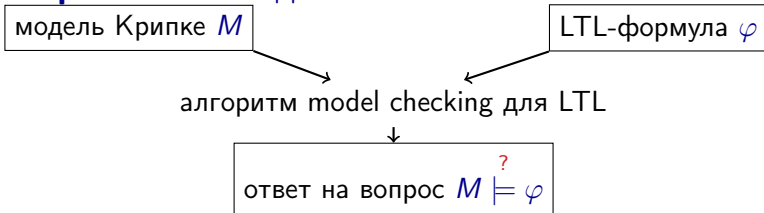


# Схема решения ЗАДАЧИ



На ближайших семинарах будет рассматриваться логика линейного времени

# Схема решения ЗАДАЧИ



Вам известно (как минимум) два алгоритма проверки LTL-формул:

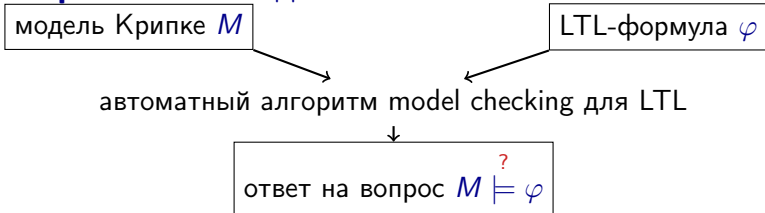
- ▶ **табличный алгоритм**

- ▶ наглядный, хотя и не такой понятный, как для CTL
- ▶ лежит в основе всех других алгоритмов
- ▶ крайне неэффективный

- ▶ **автоматный алгоритм**

- ▶ более сложно устроенный
- ▶ намного эффективнее табличного (?)

# Схема решения ЗАДАЧИ

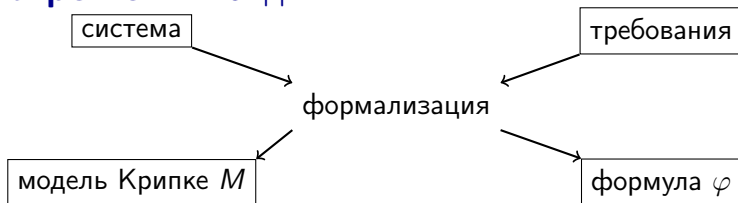


Эффективность автоматного алгоритма model checking для LTL обеспечивается двумя составляющими:

1. эффективно работающее представление автоматов Бюхи
2. эффективный алгоритм поиска компонент сильной связности, достижимых из входов автомата

В программном средстве LTL-верификации, как правило, эффективно реализована вариация автоматного алгоритма

## Схема решения ЗАДАЧИ



Этап формализации модели и требований по-прежнему остаётся и мало чем отличается от того, что происходило на семинарах, посвящённых CTL



# Схема решения ЗАДАЧИ

Вот список программных средств, способных проверять выполнимость LTL-формул в *каких-то* моделях:

*(на случай если захотите их использовать)*

BANDERA   CADENCE SMV   LTSA   LTSmin

NuSMV   PAT   ProB   SAL

SATMC   SPIN   Spot   ...

*Disclaimer: список неполный, и я не знаю большинства этих средств;  
информация взята из соответствующей страницы в википедии*

Некоторые из этих средств работают и с CTL

В курсе сосредоточим внимание на средстве Spin:

- ▶ оно открытое и бесплатное
- ▶ оно довольно популярно
- ▶ его язык (**P**romela: **P**rocess **m**eta **l**anguage) достаточно прост для понимания

*(и намного более приятен, чем язык NuSMV)*

## (S) Hello, World!

```
1 bool b;  
2  
3 active proctype P() {  
4     do  
5         :: b = !b;  
6     od  
7 }  
8  
9 ltl f1 {[<>b}  
10 ltl f2 {<>[b]}
```

На этом примере можно разобрать основы того,

- ▶ как описывать модели Крипке на языке Promela
- ▶ как применять средство средство Spin для верификации LTL-формул

Многие конструкции языка Promela схожи с

*аналогичными конструкциями C/C++ и/или NuSMV*

и синтаксисом, и семантикой

*(можно даже особнным образом вставлять в модель C/C++-код, но эти возможности в обзоре не обсуждаются)*

## (S) Модель Крипке: состояния

```
1 bool b;
2
3 active proctype P() {
4     do
5         :: b = !b;
6     od
7 }
8
9 ltl f1 {[<>b}
10 ltl f2 {<>[b]}
```

В примере *объявлена* одна *глобальная переменная* `b` типа `bool`

Эта переменная

- ▶ может принимать два значения:  
0 и 1, они же `false` и `true` соответственно
- ▶ инициализируется значением 0

## (S) Модель Крипке: состояния

```
1 bool b;  
2  
3 active proctype P() {  
4     do  
5         :: b = !b;  
6     od  
7 }  
8  
9 ltl f1 {[<>b}  
10 ltl f2 {<>[b]}
```

В примере описан один **тип процесса** P  
( ~класс/функция в C/C++, ~модуль в NuSMV)

Описание типа процесса имеет следующий вид:

**proctype** *тип\_процесса* (*параметры*) {*тело\_процесса*}

## (S) Модель Крипке: состояния

```
1 bool b;  
2  
3 active proctype P() {  
4     do  
5         :: b = !b;  
6     od  
7 }  
8  
9 ltl f1 {[<>b}  
10 ltl f2 {<>[b]}
```

На каждом шаге выполнения Promela-системы:

- ▶ запущено произвольное число процессов описанных типов
- ▶ в каждом процессе определена **текущая команда** тела — эта команда может выполняться во время перехода
- ▶ совокупное состояние системы (*состояние модели Крипке*) = совокупность значений переменных (*состояние данных*) + совокупность текущих команд запущенных процессов (*состояние управления*)

## (S) Модель Крипке: состояния

```
1 bool b;  
2  
3 active proctype P() {  
4     do  
5         :: b = !b;  
6     od  
7 }  
8  
9 ltl f1 {[<>b}  
10 ltl f2 {<>[b]}
```

Ключевое слово **active** перед типом процесса означает, что в начале работы запускается один процесс этого типа

Состояние управления процесса после запуска =  
первая команда тела его типа

**В этом примере** на каждом шаге выполнения системы  
текущая команда = весь цикл  
(о том, почему это так, рассказывается позже)

## (S) Модель Крипке: состояния

```
1 bool b;  
2  
3 active proctype P() {  
4     do  
5         :: b = !b;  
6     od  
7 }  
8  
9 ltl f1 {[<>b}  
10 ltl f2 {<>[b]}
```

Модель Крипке для этого примера имеет два состояния, среди которых ровно одно начальное:

b/0

b/1

*(состояния управления процессов в иллюстрациях опускаются)*

## (S) Модель Крипке: переходы

```
1 bool b;  
2  
3 active proctype P() {  
4     do  
5         :: b = !b;  
6     od  
7 }  
8  
9 ltl f1 {[<>b}  
10 ltl f2 {<>[b]}
```

Если в системе запущен один процесс, то он выполняется последовательно *обычным образом* (примерно как в C)

Единственный процесс, запущенный в примере,

- ▶ выполняет безусловный бесконечный цикл do-od
- ▶ на каждом витке цикла выполняет *присваивание* значения !b в переменную b
- ▶ выполняет ровно один виток цикла на каждом переходе



## (S) Модель Крипке: переходы

```
1 bool b;  
2  
3 active proctype P() {  
4     do  
5         :: b = !b;  
6     od  
7 }  
8  
9 ltl f1 {[<>b}  
10 ltl f2 {<>[]b}
```

Модель Крипке для этого примера выглядит так:



## (S) Язык LTL-формул

```
1 bool b;  
2  
3 active proctype P() {  
4     do  
5         :: b = !b;  
6     od  
7 }  
8  
9 ltl f1 {[ ]<>b}  
10 ltl f2 {<>[ ]b}
```

Описание системы включает в себя список проверяемых LTL-свойств

Описание свойства располагается вне тел всех процессов и выглядит так:

- ▶ `ltl имя_свойства { тело_свойства }` — для именованных свойств
- ▶ `ltl { тело_свойства }` — для безымянных свойств
  - ▶ безымянное свойство рекомендуется использовать только в том случае, если оно одно во всей системе

## (S) Язык LTL-формул

```
1 bool b;  
2  
3 active proctype P() {  
4     do  
5         :: b = !b;  
6     od  
7 }  
8  
9 ltl f1 {[<>b}  
10 ltl f2 {<>[b]}
```

БНФ, описывающая синтаксис тел LTL-свойств ( $\varphi$ ):

$\varphi ::= \text{булево\_выражение} \mid \varphi \ \&\& \ \varphi \mid \text{“}\varphi \mid\mid \varphi\text{”} \mid !\varphi \mid$   
 $\varphi \rightarrow \varphi \mid \varphi \text{ implies } \varphi \mid \varphi \leftrightarrow \varphi \mid \varphi \text{ equivalent } \varphi$   
 $[] \varphi \mid \text{always } \varphi \mid \langle \rangle \varphi \mid \text{eventually } \varphi \mid$   
 $\varphi \text{ U } \varphi \mid \varphi \text{ until } \varphi$

Операция	В БНФ	Операция	В БНФ
импликация	$\rightarrow$ , eventually	равносильность	$\leftrightarrow$ , equivalent
<b>G</b>	$[]$ , always	<b>F</b>	$\langle \rangle$ , eventually
<b>U</b>	$\text{U}$ , until	<b>X</b>	отсутствует

# Использование средства Spin

## Способ 1: консоль Linux

1. По исходному тексту модели получить исполняемый файл верификатора

```
> ls
helloworld.pml
> spin -a helloworld.pml
ltl f1: [] (<=> (b))
ltl f2: <=> ([] (b))
the model contains 2 never claims: f2, f1
only one claim is used in a verification run
choose which one with ./pan -a -N name (defaults to -N f1)
or use e.g.: spin -search -ltl f1 helloworld.pml
> ls
helloworld.pml pan.b pan.c pan.h pan.m pan.p pan.t _spin_nvr.tmp
> gcc -o pan pan.c
> ls
helloworld.pml pan pan.b pan.c pan.h pan.m pan.p pan.t _spin_nvr.tmp
```

# Использование средства Spin

## Способ 1: консоль Linux

- 2а. Запустить верификатор с флагом “проверь это свойство” и убедиться, что свойство выполнено

```
> ./pan -a -N f1
pan: ltl formula f1

(Spin Version 6.4.7 -- 19 August 2017)
  + Partial Order Reduction

Full statespace search for:
  never claim           + (f1)
  assertion violations  + (if within scope of claim)
  acceptance cycles    + (fairness disabled)
  invalid end states    - (disabled by never claim)

State-vector 28 byte, depth reached 3, errors: 0
   3 states, stored
   1 states, matched
   4 transitions (= stored+matched)
   0 atomic steps
hash conflicts:      0 (resolved)

Stats on memory usage (in Megabytes):
  0.000    equivalent memory usage for states (stored*(State-vector + overhead))
  0.290    actual memory usage for states
 128.000    memory used for hash table (-w24)
  0.534    memory used for DFS stack (-m10000)
 128.730    total actual memory usage

unreached in proctype P
  helloworld.pml:7, state 5, "-end-"
  (1 of 5 states)
unreached in claim f1
  spin nvr.tmp:10, state 13, "-end-"
  (1 of 13 states)

pan: elapsed time 0 seconds
```

# Использование средства Spin

## Способ 1: консоль Linux

26. Запустить верификатор с флагом “проверь это свойство” и убедиться, что свойство не выполнено

```
> ./pan -a -N f2
pan: ltl formula f2
pan:1: acceptance cycle (at depth 0)
pan: wrote helloworld.pml.trail
(Spin Version 6.4.7 -- 19 August 2017)
Warning: Search not completed
+ Partial Order Reduction

Full statespace search for:
  never claim           + (f2)
  assertion violations   + (if within scope of claim)
  acceptance cycles      + (fairness disabled)
  invalid end states     - (disabled by never claim)

State-vector 28 byte, depth reached 3, errors: 1
  2 states, stored (3 visited)
  1 states, matched
  4 transitions (= visited+matched)
  0 atomic steps
hash conflicts:          0 (resolved)

Stats on memory usage (in Megabytes):
  0.000    equivalent memory usage for states (stored*(State-vector + overhead))
  0.290    actual memory usage for states
 128.000   memory used for hash table (-w24)
  0.534    memory used for DFS stack (-m10000)
 128.730   total actual memory usage

pan: elapsed time 0 seconds
```

"есть бесконечный цикл, опровергающий свойство"

трасса, для которой свойство не выполнено, записана в этот файл

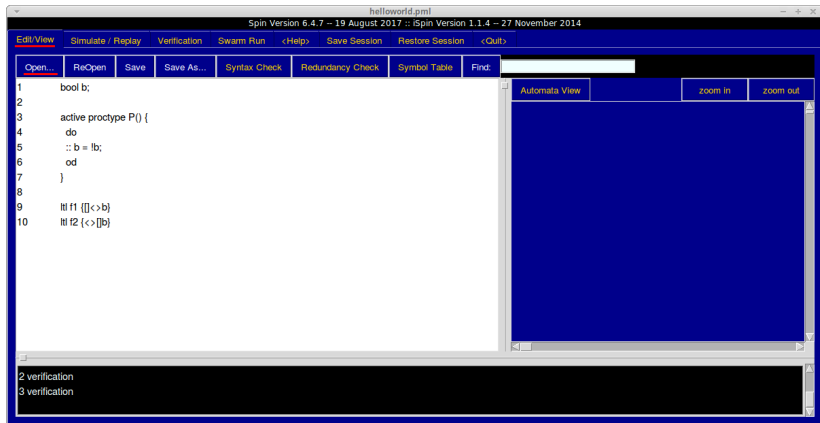
что-то не выполнено

# Использование средства Spin

## Способ 2: GUI ispin

ispin — это графическая оболочка от разработчиков Spin, написанная на tcl/tk

```
> ./iSpin/ispin.tcl
```

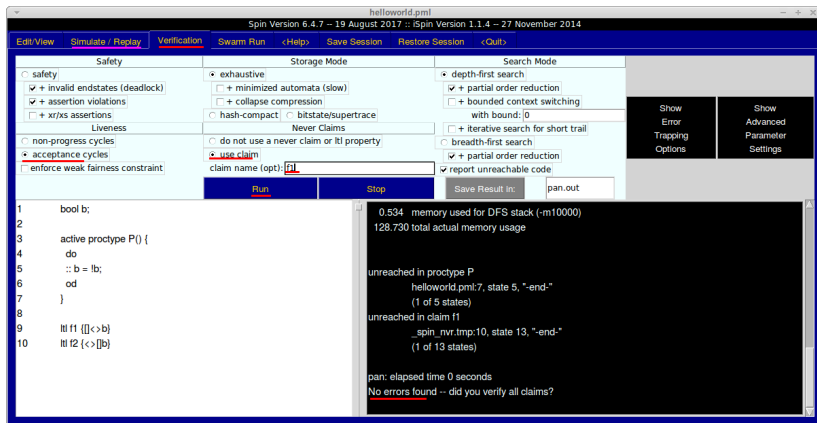


# Использование средства Spin

## Способ 2: GUI ispin

ispin — это графическая оболочка от разработчиков Spin, написанная на tcl/tk

```
> ./iSpin/ispin.tcl
```





# Использование средства Spin

## Способ 3: GUI jspin

jspin — это сторонняя графическая оболочка, написанная на Java

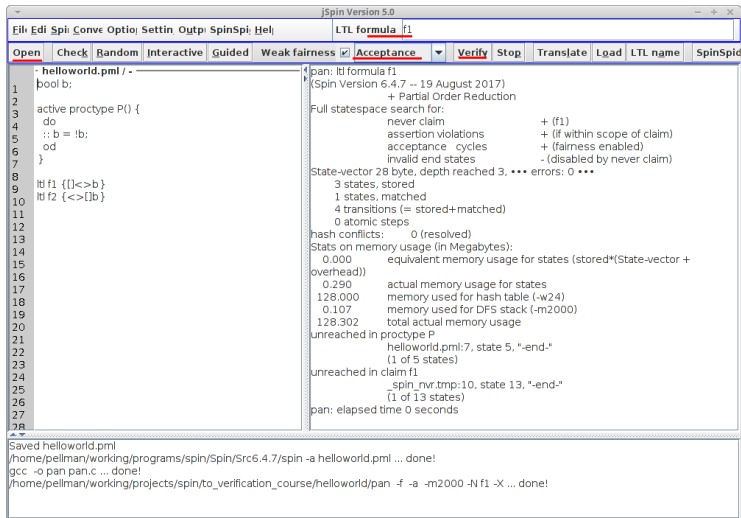
```
config.cfg
1 #jSpin configuration file
2 #Wed Dec 15 09:27:07 IST 2010
3 VERIFY_OPTIONS=-a
4 FONT_SIZE=14
5 PAN_OPTIONS=-X
6 WIDTH=1000
7 INTERACTIVE_OPTIONS=-i -X
8 SELECT_MENU=5
9 WRAP=true
10 SELECT_BUTTON=220
11 SPIN=/path/to/spin/binary/spin
12 LR_DIVIDER=400
13 CHECK_OPTIONS=-a
14 VERIFY_MODE=Safety
15 VARIABLE_WIDTH=10
16 MSC=false
17 SOURCE_DIRECTORY=jspin-examples
18 TAB_SIZE=4
19 RAW=false
20 C_COMPILER_OPTIONS=-o pan pan.c
21 VERSION=6
22 COMMON_OPTIONS=-g -l -p -r -s
23 TRAIL_OPTIONS=-t -X
24 MAX_DEPTH=2000
25 TRANSLATE_OPTIONS=-f
26 C_COMPILER=gcc
27 STATEMENT_TITLE=Statement
28 DOT=dot
29 TRAIL_FILE_NAME=txt\copyright
```

```
> java -jar ./jspin-5-0/jspin.jar
```

# Использование средства Spin

## Способ 3: GUI jspin

jspin — это сторонняя графическая оболочка, написанная на Java



## (S) Простые типы данных

Многие типы данных Promela *похожи на типы данных C*:

- ▶ `bool`: значения 0 и 1, они же `false` и `true`
- ▶ `bit`: синоним `bool`
- ▶ `byte`: значения — целые числа от 0 до 255
- ▶ `short`: значения — целые числа от  $-2^{15} - 1$  до  $2^{15} - 1$
- ▶ `int`: значения — целые числа от  $-2^{31} - 1$  до  $2^{31} - 1$
- ▶ `unsigned`: беззнаковые числа, хранящиеся в заданном числе бит, явно указываемом при объявлении
  - ▶ `unsigned x : 5;` — объявление беззнакового пятибитового числа `x`

Переменные всех этих типов по умолчанию инициализируются значением 0

Инициализация не по умолчанию выглядит *так же, как в C*:

*тип переменная = значение ;*

## (S) Непростые типы данных

Одномерные массивы определяются *так же, как в C*, с поправкой на инициализацию:

- ▶ `byte x[4];` — одномерный массив `x` из 4-х элементов типа `byte`, каждый из которых инициализируется значением 0
- ▶ `byte x[4] = 1;` — то же самое, но каждый элемент инициализируется значением 1

Структуры определяются при помощи ключевого слова `typedef`, *аналогичного слову `struct` языка C*:

- ▶ `typedef T {bool a; int b};` — тип `T` с полями `a`, `b`
- ▶ `typedef onedim {bool a[4];};` — “костыль”, позволяющий определять многомерные массивы

Доступ к элементам массивов и полям структур осуществляется *так же, как в C*:

```
onedim x[3];  
...  
x[0].a[2] = true;
```

## (S) Непростые типы данных

`mtype` — особый тип, *аналогичный перечислению `enum` в C* с некоторыми поправками:

- ▶ `mtype` — это имя типа, и такой тип всегда один
- ▶ объявление: `mtype = {имя1, ..., имяN};`
- ▶ объявлений может быть несколько, и все имена “сливаются” в один тип `mtype`
- ▶ начальное значение переменной этого типа — 0, и это значение отличается от всех имён

**Ремарка:** “`mtype`” = “message type”; изначальное назначение типа `mtype` — перечисление типов сообщений в каналах связи между процессами; но `mtype` можно использовать и в других целях, а о каналах будет рассказано дальше

**Пример:**

```
mtype = {A, B, C};  
mtype = {D, E, F};  
mtype x = B;  
...  
x = D;
```

## (S) Композиция процессов

На каждом шаге выполнения системы текущая команда каждого запущенного процесса может быть:

- ▶ **активной**: процесс **может** выполнить команду во время перехода (**процесс активен**)
- ▶ **неактивной**, или **заблокированной**: процесс **не может** выполнить команду во время перехода (**процесс неактивен, заблокирован**)

Выполнению активной команды соответствует один или несколько переходов в модели Крипке (команды выполняются недетерминированно)

## (S) Композиция процессов

Система выполняется пошагово согласно *семантике чередующихся вычислений*:

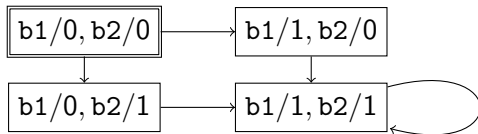
- ▶ недетерминированно выбирается активный процесс, и недетерминированно выполняется его текущая команда
- ▶ в модели Крипке для системы содержатся переходы согласно **всем** возможностям выбора процесса и способа выполнения его команды
- ▶ если все процессы неактивны, то *по умолчанию* в модель Крипке добавляется петля: переход, не изменяющий состояние системы

В особо оговорённых случаях при выполнении перехода могут выбираться несколько процессов, и их команды выполняются одновременно (*синхронно*)

## (S) Композиция процессов

```
1 bool b1;  
2 bool b2;  
3  
4 active proctype P() {b1 = !b1;}  
5 active proctype Q() {b2 = !b2;}
```

Модель Крипке для этого примера выглядит так:



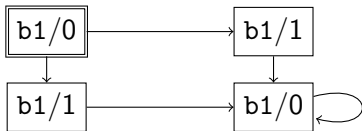


## (S) Запуск нескольких одинаковых процессов

```
1 bool b1;  
2  
3 active [2] proctype P() {b1 = !b1;}
```

Чтобы запустить несколько процессов одного типа в начале работы системы, достаточно написать “**[N]**” справа от слова `active`, где `N` — то, сколько процессов должно быть запущено

Модель Крипке для этого примера выглядит так:



## (S) Тело процесса

Тело процесса — это последовательность команд, разделённых символом “;”

*Как и в C*, каждая команда может быть предварена меткой:

*метка : команда*

Основные команды, использующиеся в теле процесса:

- ▶ *присваивание*
- ▶ условная команда      (*аналогов в C нет*)
- ▶ *ветвление*
- ▶ *цикл*
- ▶ *goto*

## (S) Тело процесса: присваивание

Присваивания выглядят *примерно так же*, как и в C с *сильно ограниченным синтаксисом*:

```
переменная++  
переменная--  
переменная = выражение
```

Присваивание всегда **активно**

**Выполнение** присваивания выглядит так:

- ▶ *переменная* изменяется заданным образом (увеличивается на 1; уменьшается на 1; в неё записывается значение *выражения*)
- ▶ управление передаётся следующей команде последовательности

## (S) Выражения

В выражениях используются переменные, константы (целые числа, true, false, имена перечисления) и многие операции, *аналогичные операциям языка C*, например:

- ▶ арифметические операции: +, -, \*, /
- ▶ побитовые операции: <<, >>, ~, &, ^, |
- ▶ арифметические отношения: <, >, <=, >=, ==, !=
- ▶ логические операции: !, &&, ||
- ▶ тернарный оператор: ->: (“->” вместо “?”)
- ▶ операция индексирования массива: []
- ▶ оператор доступа к полям структур: .

## (S) Тело процесса: условная команда

Условной командой является любое булево выражение

Условная команда **активна**  $\Leftrightarrow$  значение выражения равно true

**Выполнение** условной команды выглядит так:

- ▶ переменные не изменяются
- ▶ управление передаётся следующей команде последовательности

## (S) Тело процесса: ветвление

```
if  
  :: непустая_последовательность_команд  
  ...  
  :: непустая_последовательность_команд  
fi
```

**Альтернатива** — это *непустая последовательность команд*, располагающаяся после “::”

**Голова** альтернативы — это первая команда *последовательности*

Альтернатива **активна**  $\Leftrightarrow$  активна её голова

Ветвление **активно**  $\Leftrightarrow$  активна хотя бы одна его альтернатива

**Выполнение** ветвления выглядит так:

- ▶ недетерминированно выбирается одна из активных альтернатив
- ▶ ветвление заменяется на выбранную альтернативу
- ▶ выполняется голова выбранной альтернативы

## (S) Тело процесса: ветвление

```
if  
  :: непустая_последовательность_команд  
  ...  
  :: непустая_последовательность_команд  
fi
```

**else** — специальная **условная команда**:

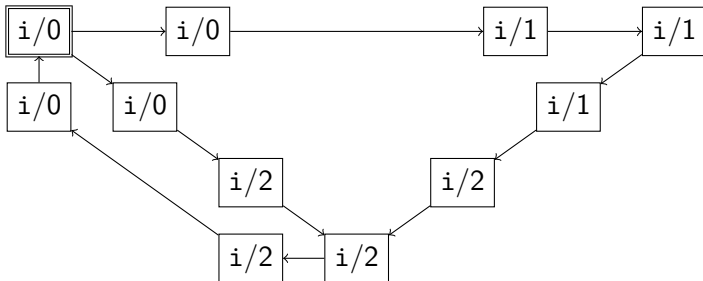
- ▶ её можно написать только в голове не более чем одной альтернативы ветвления
- ▶ команда **else** **активна**  $\Leftrightarrow$   
все остальные альтернативы неактивны

Вместо разделителя команд “;” можно писать синоним “->”,  
повышающий наглядность записи альтернатив

## (S) Тело процесса: ветвление

```
1 byte i;  
2  
3 active proctype P() {  
4     L1: if  
5         :: i < 1 -> i = i + 2;  
6         :: i < 2 -> i++;  
7         :: else -> i = 0;  
8     fi;  
9     goto L1  
10 }
```

Модель Крипке для этого примера выглядит так:





## (S) Тело процесса: цикл

do

:: *непустая\_последовательность\_команд*

...

:: *непустая\_последовательность\_команд*

od

Команда цикла работает *почти так же*, как и команда ветвления

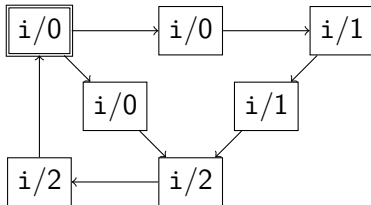
Единственное отличие: после выполнения альтернативы управление не передаётся следующей команде, а возвращается в начало цикла

Ключевое слово **break** — это всегда активная команда, не изменяющая данных и принудительно передающая управление команде, следующей за циклом, ближайшим по вложенности

## (S) Тело процесса: цикл

```
1 byte i;  
2  
3 active proctype P() {  
4   do  
5     :: i < 1 -> i = i + 2;  
6     :: i < 2 -> i++;  
7     :: else -> i = 0;  
8   od  
9 }
```

Модель Крипке для этого примера выглядит так:



## (S) Каналы связи

`chan канал = [ёмкость] of {тип};`

Там же, где объявляются глобальные переменные, можно объявлять и **каналы связи** двух видов:

- ▶ **синхронные** (*ёмкость* == 0)
- ▶ **асинхронные** (*ёмкость* > 0)

Через каналы пересылаются **сообщения**: значения заданного *типа*

В канале содержится **очередь**, вмещающая столько сообщений, какова *ёмкость* канала

Специальными командами можно:

- ▶ **послать** сообщение в канал: добавить в очередь
- ▶ **принять** сообщение из канала: извлечь из очереди сообщение, посланное раньше остальных, и по необходимости сохранить это сообщение в переменную

## (S) Каналы связи (асинхронные)

`chan канал = [ёмкость] of {тип};`  
(ёмкость > 0)

Команда отправки сообщения,  
равного значению *выражения*, в канал:  
*канал ! выражение*

Команда **активна**  $\Leftrightarrow$   
размер очереди *канала* меньше его *ёмкости*

**Выполнение** команды выглядит так:

- ▶ значение *выражения* добавляется в очередь *канала*
- ▶ управление передаётся следующей команде

## (S) Каналы связи (асинхронные)

`chan канал = [ёмкость] of {тип};`  
(ёмкость > 0)

Команда приёма сообщения,  
равного значению *выражения*, из канала:

*канал ? выражение*

Команда **активна**  $\Leftrightarrow$  очередь *канала* не пуста и сообщение,  
посланное в *канал* раньше остальных,  
равно значению *выражения*

**Выполнение** команды выглядит так:

- ▶ значения переменных не изменяются
- ▶ управление передаётся следующей команде

## (S) Каналы связи (асинхронные)

```
chan канал = [ёмкость] of {тип};  
                (ёмкость > 0)
```

Команда приёма произвольного сообщения из *канала* с сохранением его в *переменную*:

*канал*?*переменная*

Команда **активна**  $\Leftrightarrow$  очередь *канала* не пуста

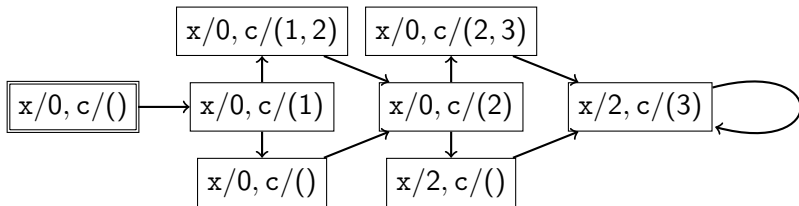
**Выполнение** команды выглядит так:

- ▶ сообщение, посланное в *канал* раньше остальных, удаляется из канала и присваивается *переменной*
- ▶ управление передаётся следующей команде

## (S) Каналы связи (асинхронные)

```
1 chan c = [2] of {byte};  
2 byte x;  
3  
4 active proctype P() {c!1; c!2; c!3;}  
5 active proctype Q() {c?1; c?x; c?2;}
```

Модель Крипке для этого примера выглядит так:



## (S) Каналы связи (синхронные)

`chan канал = [0] of {тип};`

Команда отправки сообщения,  
равного значению *выражения*, в канал:  
*канал ! выражение*

Команда **активна**  $\Leftrightarrow$  среди текущих команд других процессов  
содержится одна из команд приёма

- ▶ *канал ? переменная*
- ▶ *канал ? другое\_выражение*,  
причём значения *выражения* и *другого\_выражения* равны

Обозначенные команды приёма (**соответствующие отсылке**)  
также считаются **активными**



## (S) Каналы связи (синхронные)

`chan канал = [0] of {тип};`

Команда отправки сообщения,  
равного значению *выражения*, в канал:  
*канал ! выражение*

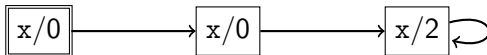
**Выполнение** команды отправки выглядит так:

- ▶ недетерминированно выбирается процесс с приёмом, соответствующим отправке
- ▶ в процессе с отправкой и **выбранном** процессе с **приёмом** управление передаётся следующим командам
- ▶ если выбран процесс с приёмом в *переменную*, то в *переменную* записывается значение *выражения*

## (S) Каналы связи (синхронные)

```
1 chan c = [0] of {byte};  
2 byte x;  
3  
4 active proctype P() {c!1; c!2; c!3;}  
5 active proctype Q() {c?1; c?x; c?2;}
```

Модель Крипке для этого примера выглядит так:



## (S) Команда запуска процесса

Специальной командой можно запускать новые процессы при выполнении системы:

```
run тип_процесса (аргументы)
```

Эта команда всегда **активна**

**Выполнение** команды выглядит так:

- ▶ в системе запускается процесс указанного типа
- ▶ управление передаётся следующей команде

В описании типа процесса могут содержаться *параметры*:

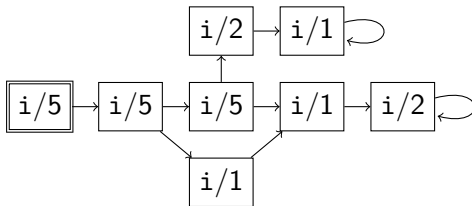
- ▶ *параметры* — это список *объявлений*, разделённых “;”
- ▶ *объявление* ::= *тип список\_имён\_через\_запятую*

Параметры запускаемого процесса и аргументы в описании процесса работают так же, как и *передача аргументов по значению при вызове функции в C/C++*

## (S) Команда запуска процесса

```
1 byte i = 5;  
2  
3 proctype Q(byte a) {i = a;}  
4  
5 active proctype P() {  
6     run Q(1);  
7     run Q(2);  
8 }
```

Модель Крипке для этого примера выглядит так:



## (S) Неделимые последовательности команд

Иногда при описании системы параллельно работающих процессов возникает необходимость указать, что несколько команд процесса выполняются **неделимо**: так, чтобы между выполнением этих команд ни один другой запущенный процесс не мог выполнить свою команду

### Например:

- ▶ если назначение процесса Р в последнем примере — *инициализация* системы с двумя немного отличающимися процессами типа Q, то хотелось бы, чтобы первый запущенный процесс типа Q до выполнения своего первого действия подождал, пока запустится второй процесс типа Q
- ▶ в *протоколе доступа в критическую секцию*, основанном на семафорах, хотелось бы, чтобы между проверкой значения семафора и его изменением одним процессом другие процессы не могли начать работать с тем же семафором

## (S) Неделимые последовательности команд

`atomic {непустая_последовательность_команд}`

Любую последовательность команд можно “обернуть” скобками с ключевым словом `atomic`, чтобы сделать её *неделимой*

**Правила выполнения** неделимой последовательности команд:

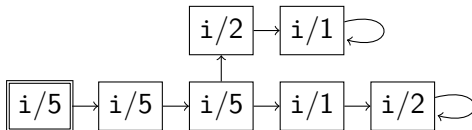
- ▶ если
  1. на предыдущем шаге работы системы выполнена команда процесса  $p$ , входящая в неделимую последовательность  $s$ ,  
и
  2. текущая команда процесса  $p$  **активна** и входит в  $s$ ,  
то при переходе **обязательно** выбирается процесс  $p$
- ▶ в остальных случаях  
система выполняется по обычным правилам

**Ограничение по использованию:** средство `Spin` может работать некорректно, если в какой-либо неделимой последовательности системы содержится цикл

## (S) Неделимые последовательности команд

```
1 byte i = 5;  
2  
3 proctype Q(byte a) {i = a;}  
4  
5 active proctype P() {  
6     atomic{  
7         run Q(1);  
8         run Q(2);  
9     }  
10 }
```

Модель Крипке для этого примера выглядит так:



## (S) Локальные переменные

В начале тела процесса можно объявлять *локальные переменные* так же, как вне тел всех процессов объявляются глобальные

Локальные переменные создаются и инициализируются для каждого процесса в момент его запуска, и исчезают из системы, когда исчезает процесс

Объявление локальной переменной не является командой

**Если в системе запущен ровно один процесс заданного типа P, то доступ извне к его локальным переменной x и метке L осуществляется так: P:x, P@L**

В частности, такие выражения можно использовать в LTL-свойствах (“P@L” = булево выражение “управление процесса P находится у метки L”)

*(доступ к локальным переменным и меткам процесса, если запущено много процессов одного типа, менее тривиален — читайте документацию)*



## (S) Пример для размышлений

```
bool near, dead, hunted;
mtype = {ping};
chan c = [0] of {mtype};

active proctype mosquito() {
    do
        :: !near && !dead          -> near = true; c!ping;
        :: near && !hunted && !dead -> near = false;
    od
}

active proctype bird() {
    do
        ::          c?ping          -> hunted = true;
        :: atomic{hunted && near  -> dead = true; hunted = false;}
        ::          hunted && !near -> hunted = false;
    od
}

ltl f1 {<>(near && <> dead)}
ltl f2 {[ ](near -> <> dead)}
ltl f3 {[ ](hunted -> <> dead)}
```

Что здесь происходит, и какие свойства выполнены?