

# Математические методы верификации схем и программ

[mk.cs.msu.ru](http://mk.cs.msu.ru) → Лекционные курсы  
→ Математические методы верификации схем и программ

## Блок 01

Обзор средства Spin

Лектор:

Подымов Владислав Васильевич

E-mail:

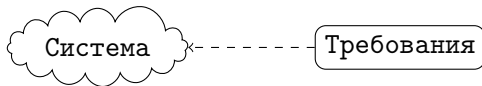
**[valdus@yandex.ru](mailto:valdus@yandex.ru)**

# Рассматриваемая ЗАДАЧА

**Дано:** неформальное описание

- ▶ системы и
- ▶ требований к ней

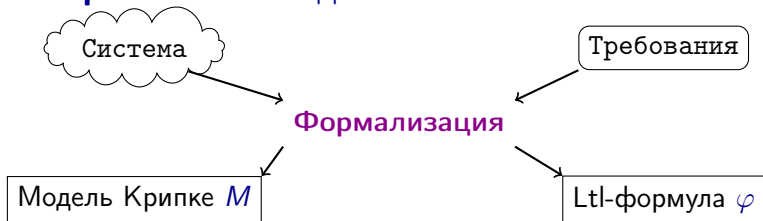
**Требуется** проверить, удовлетворяет ли система требованиям



Этой **ЗАДАЧЕ** посвящены обязательные домашние задания, в каждом из которых выбираются

- ▶ программное средство model checking, и в соответствии с ним —
- ▶ вид моделей для формализации систем,
- ▶ язык формальных спецификаций моделей и
- ▶ конкретные представления моделей и спецификаций на языке средства

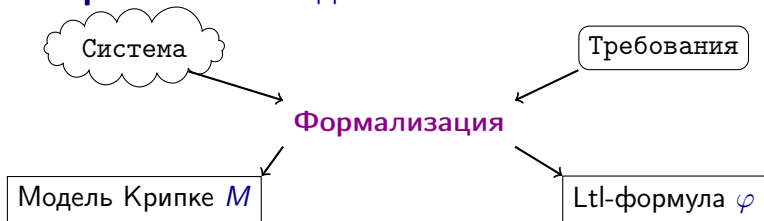
# Рассматриваемая ЗАДАЧА



Средство **Spin** (для краткости —  $\xi$ ) будем обсуждать относительно

- ▶ **моделей Крипке** как моделей рассматриваемых систем и
- ▶ **LTL** как языка формальных спецификаций

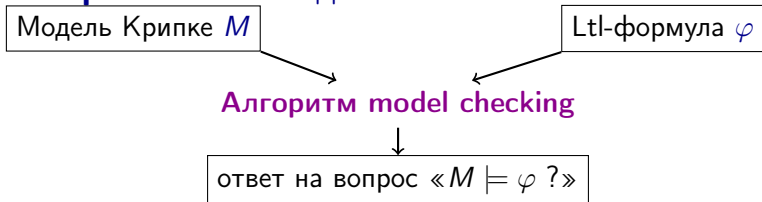
# Рассматриваемая ЗАДАЧА



Основная трудность всех обязательных домашних заданий — это этап **формализации**, приближенный к «боевым условиям» (хотя и для «игрушечных» систем):

- ▶ Даются неформальные описания системы и требований
- ▶ Требуется придумать и реализовать модель и формальную спецификацию и убедить в их правильности сначала себя, и затем «заказчика»
  - ▶ *Здесь «заказчик» — это я, и если всё решено разумно и верно, то убедить меня будет нетрудно*

# Рассматриваемая ЗАДАЧА



Алгоритмы верификации, используемые на практике в соответствующих программных средствах, как правило, так или иначе основаны на **автоматном алгоритме**

В  $\S$  используются

- ▶ автоматный алгоритм
- ▶ с построением автомата Бюхи, исследуемого на пустоту, «на лету» при помощи комбинации обходов в глубину,
- ▶ и эвристиками для оптимизации автомата и его обхода

# Рассматриваемая ЗАДАЧА

Существует немало программных средств для проверки выполнимости ltl-формул на моделях Крипке или родственных видах моделей:

BANDERA	CADENCE	SMV	LTSA	LTSmin
NuSMV	PAT		ProB	SAL
SATMC	Spin		Spot	...

*Дисклеймер: это просто несколько средств, выбранных как целенаправленно, так и наугад с соответствующей страницы в Википедии несколько лет назад*

Подробно остановимся только на  $\zeta$ , так как:

- ▶ у этого средства открытый исходный код, и его можно свободно использовать для академических целей
- ▶ это средство, *хотя и старое, но* достаточно популярно, особенно в качестве отправной точки для обучения model checking на практике
- ▶ его язык (**P**romela: **P**rocess **m**eta **l**anguage; для краткости — p) достаточно прост для понимания

## §: 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]}
```

Начнём с простого примера, чтобы на нём быстро и просто обсудить,

- ▶ как связаны  $\rho$  и модели Крипке с ltl-формулами и
- ▶ как использовать § для model checking

Некоторые конструкции  $\rho$  похожи на аналогичные конструкции в C/C++ или даже полностью совпадают (*вплоть до возможности встраивания кода на C в модель на  $\rho$* )

## μ и модели 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 строке 1 объявлена глобальная булева переменная  $b$

В этой переменной могут храниться два значения: 0 (синоним `false`) и 1 (синоним `true`)

Эта переменная инициализируется значением 0



## µ и модели 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]}
```

µ в строке 3 — это **тип процесса** (что-то вроде **класса/функции** в C/C++)

Объявление типа процесса устроено так:

```
proctype <тип_процесса> (<параметры>) {<тело>}
```

## μ и модели 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 в системе содержится некоторое количество процессов

Процесс — это, по сути, **императивная программа**: имеет своё **состояние вычисления**, складывающееся из **состояния управления** (того, какая команда процесса должна выполняться следующей) и **состояния данных** (значений локальных переменных процесса, если они есть)

## р и модели 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]}
```

Глобальные переменные модели — это **общие переменные**, к которым имеют доступ все процессы

**Состояние модели** р — это совокупность значений глобальных переменных и состояний вычисления процессов

## μ и модели 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 словом `proctype` означает, что в начале выполнения системы (в начальном состоянии соответствующей модели Крипке) запущен один процесс этого типа

Состояние управления процесса при запуске — это первая команда тела

## р и модели 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}
```

Процесс типа P в примере содержит ровно одно состояние управления  
(это объяснится позже)

С учётом этого в модели Крипке, соответствующей системе выше,  
содержится ровно два состояния, и ровно одно из них начальное:

b/0

b/1

(состояния управления здесь и далее опущены для экономии места)

## μ и модели 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 содержится ровно один процесс, то он выполняется естественным образом как **императивная программа**

**Например**, процесс в примере

- ▶ содержит бесконечный безусловный цикл (do-od)
- ▶ на каждом витке цикла переключает значение переменной *b*
- ▶ выполняет один виток цикла за один переход (*это объяснится позже*)

## μ и модели 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}
```

Состояния и переходы модели Крипке, соответствующей системе выше:



## р и ltl-формулы

```
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}
```

Формальная спецификация системы в р записывается в том же файле (тексте), что и сама система

Ltl-спецификации в р пишутся так:

- ▶ Для **именованных** формул:

```
ltl <имя_формулы> { <ltl-формула> }
```

- ▶ Для **безымянных** формул:

```
ltl { <ltl-формула> }
```

- ▶ Безымянную формулу рекомендуется использовать в том и только том случае, если это единственная формула в тексте



## ρ и ltl-формулы

```
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 ltl-формулы ( $\varphi$ ) в  $\rho$ :

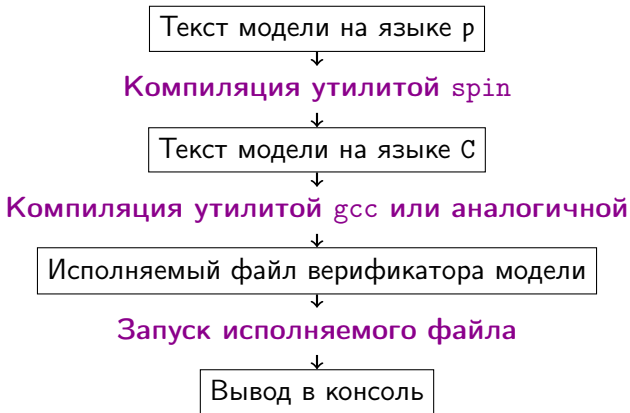
$$((\varphi \leftrightarrow \psi) = ((\varphi \rightarrow \psi) \& (\psi \rightarrow \varphi)))$$

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

В лекциях	В $\rho$	В лекциях	В $\rho$
$\rightarrow$	$\rightarrow$ , implies	$\leftrightarrow$	$\leftrightarrow$ , equivalent
<b>G</b>	$[]$ , always	<b>F</b>	$\langle \rangle$ , eventually
<b>U</b>	$U$ , until	<b>X</b>	отсутствует(!)

## §: использование

Общая «низкоуровневая» схема верификации при помощи §:



Чтобы не тратить много времени на набор нужных команд с нужными флагами в консоли, рекомендуется использовать оболочку, в которой эти команды выполняются в нужном порядке по нажатию кнопки

## §: ИСПОЛЬЗОВАНИЕ

Оболочка `jspin` (рекомендуется)

Это оболочка от сторонних разработчиков, написанная на `java`

Для запуска оболочки следует запустить основной архив `java jspin.jar` согласно возможностям ОС

Например, в консоли Linux:

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

## §: ИСПОЛЬЗОВАНИЕ

### Оболочка jspin (рекомендуется)

После запуска в папке jar-архива появится файл настроек config.cfg

Для не-Windows следует его отредактировать (*как минимум подчёркнутые строки ниже*) и перезапустить оболочку

```
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 EXIT_FILE_NAME=txt\copyright
```

## §: использование

Оболочка `ispin` (тоже можно)

Это оболочка от разработчиков §, написанная на `tcl/tk`

Эта оболочка такая же старая, как и сам §, но в целом выполняет свою задачу

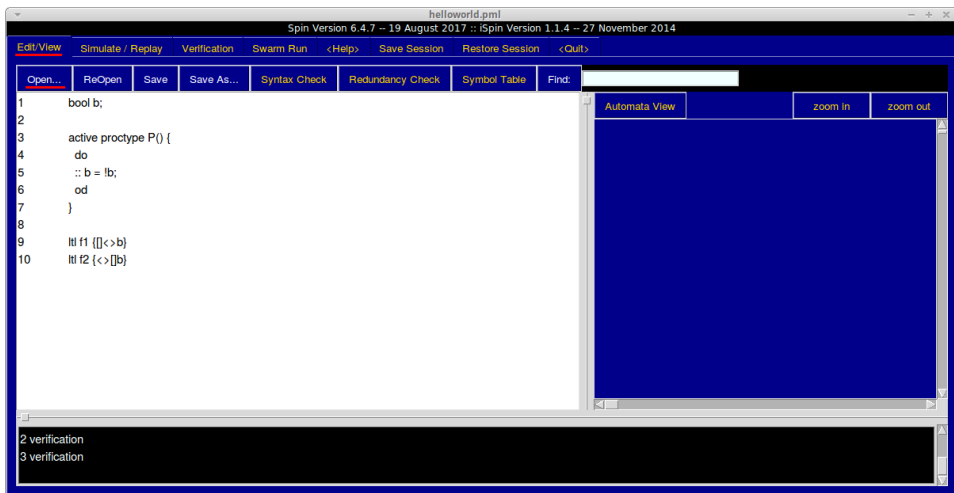
Для запуска оболочки требуется сделать так, чтобы в консоли была команда «`spin`», и запустить главный `tcl`-файл:

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

# §: ИСПОЛЬЗОВАНИЕ

Оболочка *ispin* (тоже можно)

Вкладка обзора кода (Edit/View):



# §: ИСПОЛЬЗОВАНИЕ

Оболочка *ispin* (тоже можно)

Вкладка верификации (Verification):

The screenshot shows the Spin verification tool interface. The title bar indicates the file is `helloworld.pml` and the version is `Spin Version 6.4.7 -- 19 August 2017 :: ispin Version 1.1.4 -- 27 November 2014`. The interface is divided into several sections:

- Verification Tab:** Contains options for Safety, Liveness, Storage Mode, and Search Mode. Under Safety, `+ invalid endstates (deadlock)` and `+ assertion violations` are checked. Under Liveness, `acceptance cycles` is selected. Under Storage Mode, `exhaustive` is selected. Under Search Mode, `depth-first search` is selected with `+ partial order reduction` checked. A `Run` button is visible.
- Code Editor:** Displays the following PML code:

```
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}
```
- Console Output:** A window showing the results of the verification:

```
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  
No errors found -- did you verify all claims?
```

## §: ИСПОЛЬЗОВАНИЕ

**Консоль** (не рекомендуется, но и не запрещается)

Далее приводятся команды и результаты для консоли Linux

*Компиляция утилитами spin и gcc:*

```
> 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

Консоль (не рекомендуется, но и не запрещается)

*Запуск верификатора и вывод, отвечающий выполнению формулы:*

```
> ./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

Консоль (не рекомендуется, но и не запрещается)

Запуск верификатора и вывод, отвечающий невыполнению формулы:

```
> ./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

## p: «простые» типы данных

Некоторые типы данных p похожи на **типы данных C**:

- ▶ **bool**: 0 (false), 1 (true)
- ▶ **bit**: синоним типа bool
- ▶ **byte**: целые числа от 0 до 255
- ▶ **short**: целые числа от  $-2^{15} - 1$  до  $2^{15} - 1$
- ▶ **int**: целые числа от  $-2^{31} - 1$  до  $2^{31} - 1$
- ▶ **unsigned**: неотрицательные целые числа в двоичной записи с заданным числом битов:
  - ▶ **unsigned x : N**; — объявление переменной x, хранящей N-битовое целое неотрицательное число

Значение по умолчанию для всех этих типов — 0

Инициализация значением не по умолчанию устроена **так же, как и в C**:

*<тип> <переменная> = <значение>;*

## р: «непростые» типы данных

Одномерные массивы устроены так же, как и в C, если не считать особенностей инициализации:

- ▶ `byte x[4];` — объявление массива `x` длины 4 с элементами типа `byte`, и все элементы инициализируется значением 0
- ▶ `byte x[4] = 1;` — то же самое, но все элементы инициализируются значением 1

Структуры объявляются примерно как в C, но с ключевым словом `typedef` вместо `struct`:

- ▶ `typedef T {bool a; int b};` — объявление структуры `T` с булевым полем `a` и полем `b` типа `int`
- ▶ `typedef onedim {bool a[4];};` — многомерный массив можно объявить как массив структур, содержащих массивы меньшей размерности

Доступ к элементам массивов и структур устроен так же, как и в C:

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

## р: «непростые» типы данных

`mtype` — тип, похожий на `enum` в C, но с непривычными особенностями:

- ▶ `mtype` — это имя типа (как `bool` и `int`)
- ▶ определение перечисляемых элементов этого типа выглядит так:  
`mtype = {name_1, ..., name_N};`
- ▶ все такие определения «сливаются» в одно содержащее совокупность всех определённых элементов
- ▶ значение переменной типа `mtype` по умолчанию — 0 и не совпадает ни с одним из перечисляемых значений

**Пояснение:** «`mtype`» = «`message type`»; по задумке, это конечный набор типов сообщений, которыми могут обмениваться процессы (*об этом будет говориться позже*); но можно его использовать и как обычное перечисление

### Пример:

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

## p: композиция процессов

В каждом состоянии системы каждая команда находится в одном из двух режимов:

- ▶ **активна**: может быть выполнена
- ▶ **неактивна (заблокирована)**: не может быть выполнена

Все процессы тоже делятся на активные и неактивные (заблокированные) относительно заданного состояния системы: **процесс активен**  $\Leftrightarrow$  активна команда, которая должна быть выполнена в нём следующей

Выполнение следующей команды активного процесса (**один шаг согласно операционной семантике**) отвечает одному переходу в модели Крипке

В языке есть команды, которые могут выполняться несколькими способами, и в этом случае каждому способу выполнения отвечает свой переход (а выполнению команды в целом — несколько переходов)

## р: композиция процессов

Композиция процессов в р устроена согласно **семантике чередования**

Один шаг выполнения системы устроен так:

- ▶ Недетерминированно выбирается активный процесс
- ▶ В выбранном процессе недетерминированно выбирается способ выполнения следующей команды
- ▶ Выполняется переход, отвечающий выбранному выполнению команды выбранного процесса

Таким образом, переходы системы из заданного состояния отвечают **всем** возможностям выбора активного процесса и выполнения его команды

Если в состоянии нет ни одного активного процесса, то считается, что существует переход из этого состояния в него же<sup>1</sup>

Иногда, *когда об этом говорится явно*, могут одновременно (**синхронно**) выполняться команды нескольких процессов, то есть в р содержатся некоторые средства синхронизации процессов

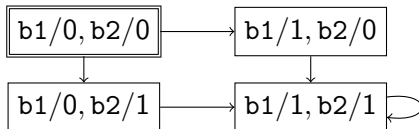
---

<sup>1</sup> Дела обстоят более нетривиально, но это лучше обсудить на семинаре

## р: композиция процессов

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

Модель Крипке, отвечающая этой системе:



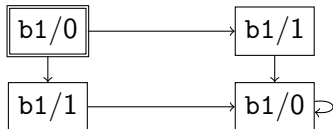


## **P:** активация нескольких процессов одного типа

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

Чтобы добавить  $N$  процессов одного типа в начальное состояние, достаточно дописать « $[N]$ » после ключевого слова «active»

Модель Крипке, отвечающая системе выше:



## p: тело процесса

Тело процесса — это последовательность команд с разделителем «;» (или синонимом «->»)

Как и в C, перед каждой командой может быть дописана **метка**:

*<метка> : <команда>*

Основной набор команд:

- ▶ присваивания
- ▶ условные команды (*их аналогов в C нет*)
- ▶ ветвления
- ▶ циклы
- ▶ goto

## p: тело процесса, присваивания

Присваивание выглядит так же, как и в C с ограниченным синтаксисом:

*<переменная> ++*

*<переменная> --*

*<переменная> = <выражение>*

Присваивание всегда **активно**

**Шаг выполнения** присваивания  $S$  единственно и определяется естественно:

- ▶ Значение *<переменной>* изменяется как написано в  $S$  (увеличивается или уменьшается на 1; перезаписывается значением *<выражения>*)
- ▶ **Управление** передаётся команде, следующей за  $S$

## р: выражения

Выражение составляется из переменных, констант (целые числа, true, false, перечисляемые имена) и операций, **аналогичных операциям в C:**

- ▶ арифметические: +, -, \*, /
- ▶ побитовые: <<, >>, ~, &, ^, |
- ▶ сравнения: <, >, <=, >=, ==, !=
- ▶ логические: !, &&, ||
- ▶ тернарный оператор: ->: ( «->» вместо «?» )
- ▶ индексирование: []
- ▶ доступ к полю: .

## p: тело процесса, условные команды

**Условная команда** — это команда, представляющая собой булево выражение

Условная команда **активна**  $\Leftrightarrow$  значение этого выражения есть true

**Шаг выполнения** активной условной команды  $S$  единственно и устроено так:

- ▶ Значения всех переменных не изменяются
- ▶ **Управление** передаётся команде, следующей за  $S$

## p: тело процесса, ветвление

```
if  
  :: <альтернатива>  
  ...  
  :: <альтернатива>  
fi
```

**Альтернатива** ветвления — это непустая последовательность команд, записанная после «::»

**Голова** альтернативы — это первая команда последовательности, а **хвост** — всё остальное

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

Ветвление **активно**  $\Leftrightarrow$  активна хотя бы одна из его альтернатив

**Шаг выполнения** активного ветвления:

- ▶ недетерминированно выбирается одна из его активных альтернатив
- ▶ ветвление заменяется на выбранную альтернативу
- ▶ выполняется один шаг головы альтернативы

## p: тело процесса, ветвление

```
if  
  :: <альтернатива>  
  ...  
  :: <альтернатива>  
fi
```

**else** — это особая **условная команда**:

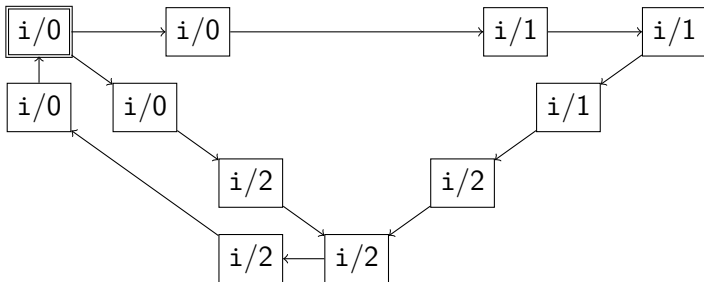
- ▶ её можно использовать только в голове альтернативы, и не более чем одной для ветвления
- ▶ эта команда **активна** ⇔ остальные альтернативы ветвления заблокированы

Чтобы повысить читаемость кода, иногда может быть удобно разделять голову и хвост альтернативы записью «-» вместо «;»

## p: 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, отвечающая этой системе:





## p: тело процесса, цикл

```
do
  :: <альтернатива>
  ...
  :: <альтернатива>
od
```

Цикл в p во многом похож на ветвление

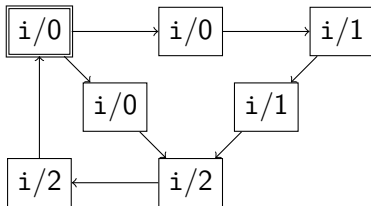
Единственное отличие: когда выполнены все команды выбранной альтернативы, **управление** передаётся не следующей команде, а обратно циклу

Ключевое слово **break** — это всегда **активная** команда, **шаг выполнения** которой не изменяет значения переменных и передаёт управление команде, следующей за ближайшим объемлющим циклом

## p: 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, отвечающая этой системе:



## р: тело процесса, команда запуска процесса

**Команда запуска** предназначена для добавления процессов в систему по ходу выполнения:

```
run <тип_процесса> (<аргументы>)
```

Эта команда всегда **активна**

**Шаг выполнения** команды запуска  $S$ :

- ▶ **Запускается** и добавляется в систему процесс указанного типа
- ▶ **Управление** передаётся команде, следующей за  $S$

В общем случае в объявлении типа процесса могут содержаться **<параметры>**:

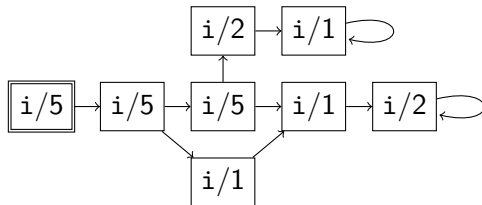
- ▶ **<Параметры>** — это список **<объявлений>** с разделителем «;»
- ▶ **<объявление>** ::= **<тип>** **<список\_имён\_через\_,>**

**<Параметры>** и **<аргументы>** связаны так же, как и в C/C++ при **передаче по значению**

## p: тело процесса, 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, отвечающая этой системе:



## p: тело процесса, атомарные наборы команд

Иногда фрагмент тела процесса, выполняющийся за несколько шагов, требуется сделать **атомарным**: таким, чтобы его выполнение не могло быть прервано выполнением команд других процессов

### Например:

- ▶ Если процесс типа P в последнем примере предназначен для *инициализации* системы с процессами Q(1) и Q(2), то выполнение P должно быть атомарным (не прерываться выполнением процессов Q)
- ▶ В протоколах доступа в критическую секцию, основанных на семафорах, проверка условия и блокировка семафора должны быть атомарной парой действий

## **p**: тело процесса, атомарные наборы команд

Объявить `<непустую_последовательность_команд>` атомарной можно так:

```
atomic {<непустая_последовательность_команд> }
```

Особенности выполнения атомарной последовательности:

### ▶ Если

1. последняя выполненная команда входит в атомарную последовательность  $s$  процесса  $p$  и
2. следующая команда процесса  $p$  **активна** и тоже входит в  $s$ , то все процессы, кроме  $p$ , блокируются при выборе следующего перехода

### ▶ Иначе система выполняется как обычно

**Будьте осторожны при использовании `atomic`**: в § содержится ряд документированных и undocumented особенностей трактовки атомарных последовательностей, в связи с которыми, в числе прочего, лучше (а) **никогда** не писать атомарные циклы и (б) иметь в виду, что последовательность переходов атомарной последовательности может быть «схлопнута» в один переход с потерей промежуточных состояний



## **p:** тело процесса, локальные переменные

Локальная переменная процесса объявляется в начале тела, как и в C (но не в современном C++)

Локальные переменные существуют тогда же, когда существует содержащий их процесс, и инициализируются при запуске процесса

Объявление локальной переменной — это не команда

Если в системе создаётся и выполняется **ровно один** процесс типа P, то обратиться к его локальной переменной x и метке L можно так: P:x, P@L

Эти выражения могут быть использованы, в частности, и в ltl-формуле («значение переменной x» и булево выражение «управление процесса находится у метки L»)

*(Если в системе создаётся несколько процессов заданного типа, то **внимательно** прочитайте документацию, если хотите обратиться к его локальной переменной или метке)*



## р: каналы связи

**Каналы** связи объявляются там же, где и глобальные переменные, и делается это так:

```
chan <канал> = [<ёмкость>] of {<тип>};
```

**Сообщение** — это значение заданного <типа>, передающееся через канал

Каналы в р работают по принципу **очереди** заданной <ёмкости>:

- ▶ Можно **отправлять** (добавлять) сообщения в канал и **принимать** (удалять) их из канала
- ▶ Если сообщение  $m1$  отправлено раньше  $m2$ , то и принято  $m1$  будет раньше, чем  $m2$
- ▶ Если в канале уже содержится столько сообщений, какова его <ёмкость>, то в канал нельзя отправить ещё одно сообщение

## р: каналы связи

Каналы ёмкости 0 будем называть **синхронными**, а остальные — **асинхронными**

Канал **пуст**, если не содержит ни одного сообщения, и **непуст** иначе

Канал **полон**, если содержит столько сообщений, какова его *ёмкость*, и **неполон** иначе

**Головой** канала будем называть сообщение, хранящееся в нём и отправленное раньше всех остальных хранящихся в нём сообщений

## p: каналы связи, асинхронные

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

Команда отправки сообщения:

```
<канал> ! <выражение>
```

Команда активна  $\Leftrightarrow$

<канал> неполон

Шаг выполнения команды отправки S:

- ▶ Вычисляется значение <выражения>
- ▶ Вычисленное значение отправляется в канал
- ▶ Управление передаётся команде, следующей за S

## р: каналы связи, асинхронные

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

Команда чтения сообщения:

```
<канал> ? <переменная>
```

Команда активна  $\Leftrightarrow$  <канал> непуст

Шаг выполнения команды чтения S:

- ▶ Голова <канала> присваивается в <переменную> и удаляется из <канала>
- ▶ Управление передаётся команде, следующей за S

## p: каналы связи, асинхронные

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

Команда приёма сообщения: *(выражение* отлично от переменной)  
*<канал>*? *<выражение>*

Команда активна  $\Leftrightarrow$  *<канал>* непуст, и значение *<выражения>* равно  
голове канала

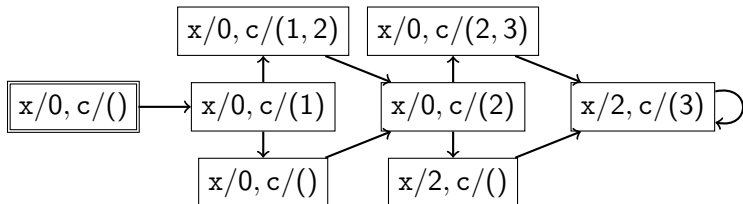
Шаг выполнения команды приёма *S*:

- ▶ Значения переменных не изменяются
- ▶ Голова удаляется из канала
- ▶ Управление передаётся команде, следующей за *S*

## p: 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, отвечающая это системе:



## p: каналы связи, асинхронные

```
chan <канал> = [0] of {<тип>};
```

Команда отправки сообщения:

```
<канал> ! <выражение>
```

Команда **активна**  $\Leftrightarrow$  хотя бы в одном процессе следующей должна выполняться команда хотя бы одного из следующих видов:

- ▶ **Команда чтения:** `<канал> ? <переменная>`
- ▶ **Команда приёма:** `<канал> ? <другое_выражение>`,  
где `<другое_выражение>` отлично от переменной, и его значение равно значению `<выражения>`

Упомянутые команды чтения и приёма также считаются **активными**

## p: каналы связи, асинхронные

```
chan <канал> = [0] of {<тип>};
```

Команда отправки сообщения:

```
<канал> ! <выражение>
```

Шаг выполнения команды отправки:

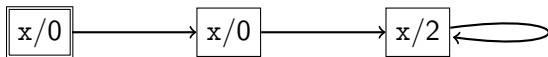
- ▶ Недетерминированно выбирается одна из соответствующих активных команд чтения или приёма
- ▶ Если выбрана команда чтения, то в записанную в неё переменную присваивается значение *<выражения>*
- ▶ Если выбрана команда приёма, то значения переменных не изменяются
- ▶ **Управление** передаётся командам, следующим за выбранными командами отправки и приёма/чтения



## р: каналы связи, асинхронные

```
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, отвечающая этой системе:



## §: заключительный пример

```
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)}
```

Как устроена соответствующая модель Крипке, и какие из записанных формул выполняются на ней?