Математические методы верификации схем и программ

Лекторы:

Захаров Владимир Анатольевич

Подымов Владислав Васильевич

e-mail рассказчика:

valdus@yandex.ru

Осень 2017

Семинар 7

`Spin`

(обзор средства)

Рассматриваемая ЗАДАЧА

Даны

- неформальное описание системы
- содержательное описание требований к системе

Требуется проверить,

удовлетворяет ли система набору требований

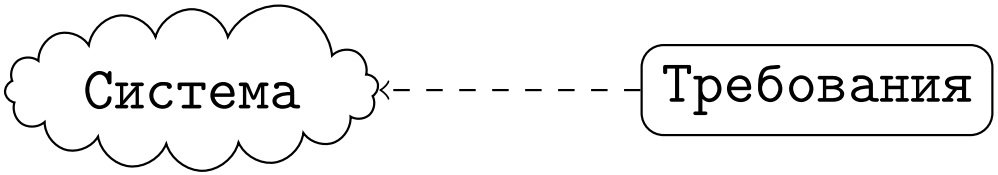


Схема решения ЗАДАЧИ

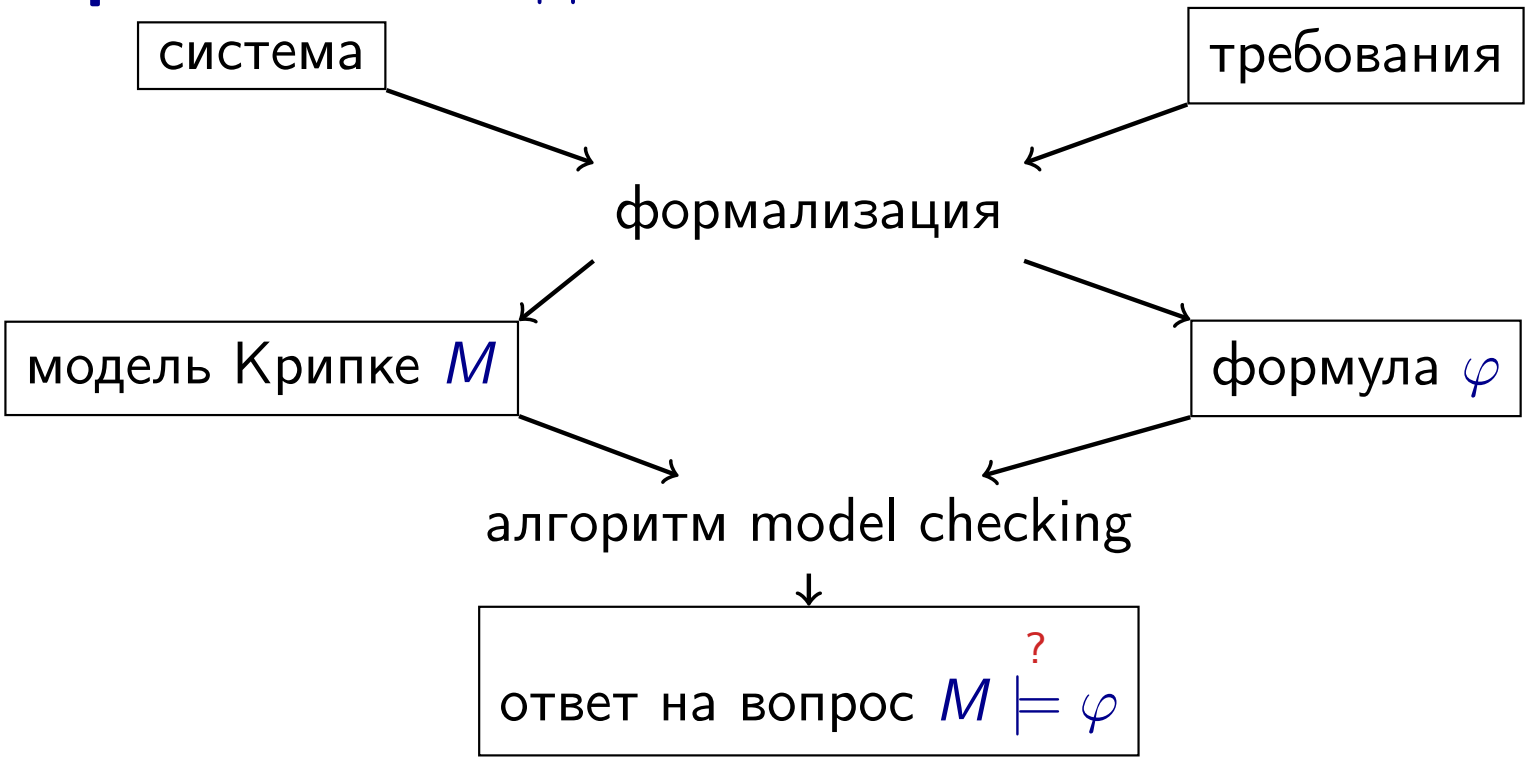
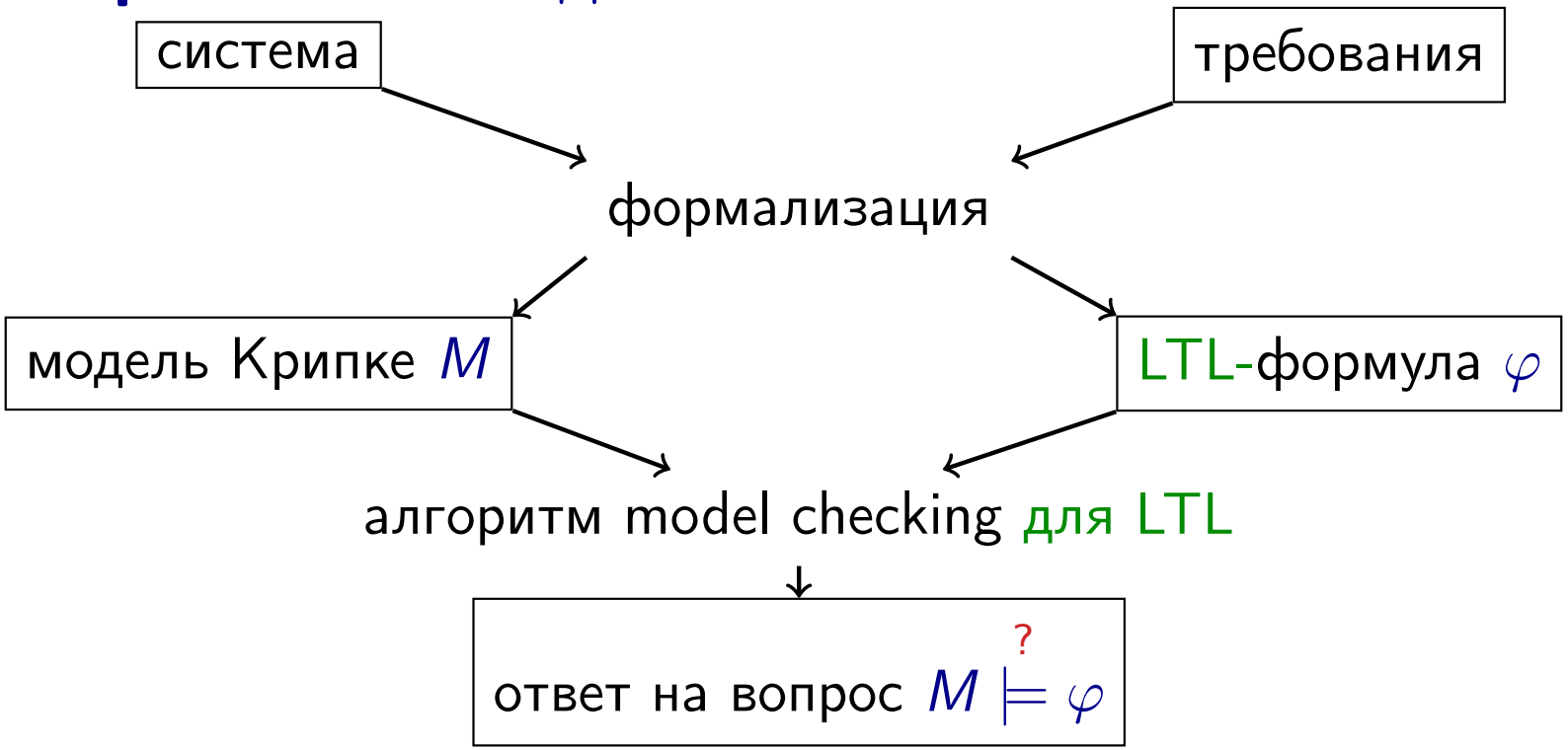
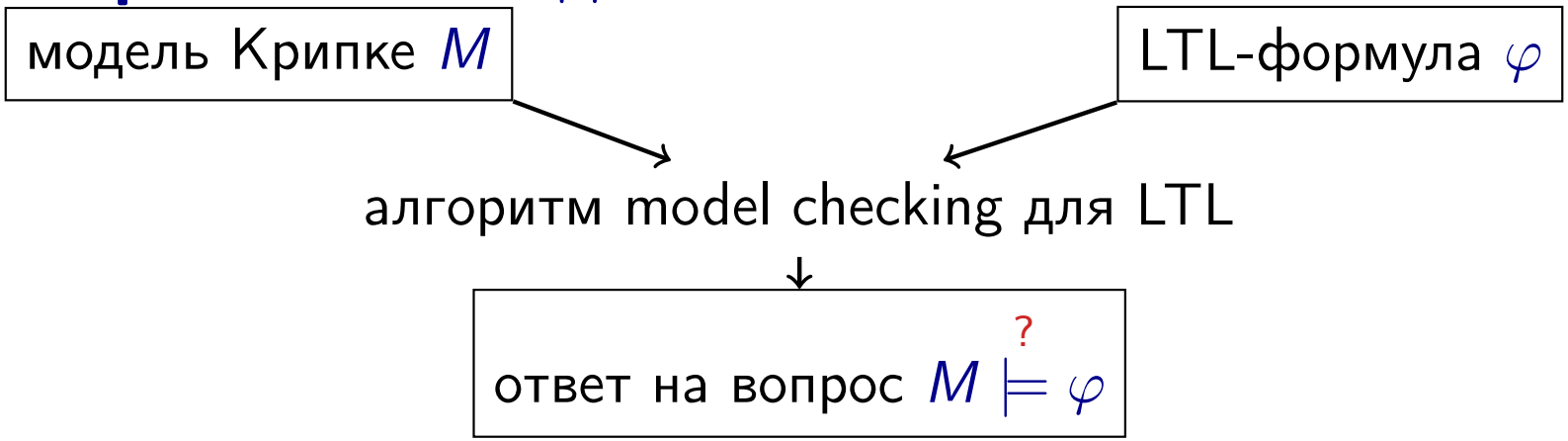


Схема решения ЗАДАЧИ



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

модель Крипке M → автоматный алгоритм model checking для LTL ← LTL-формула φ

↓

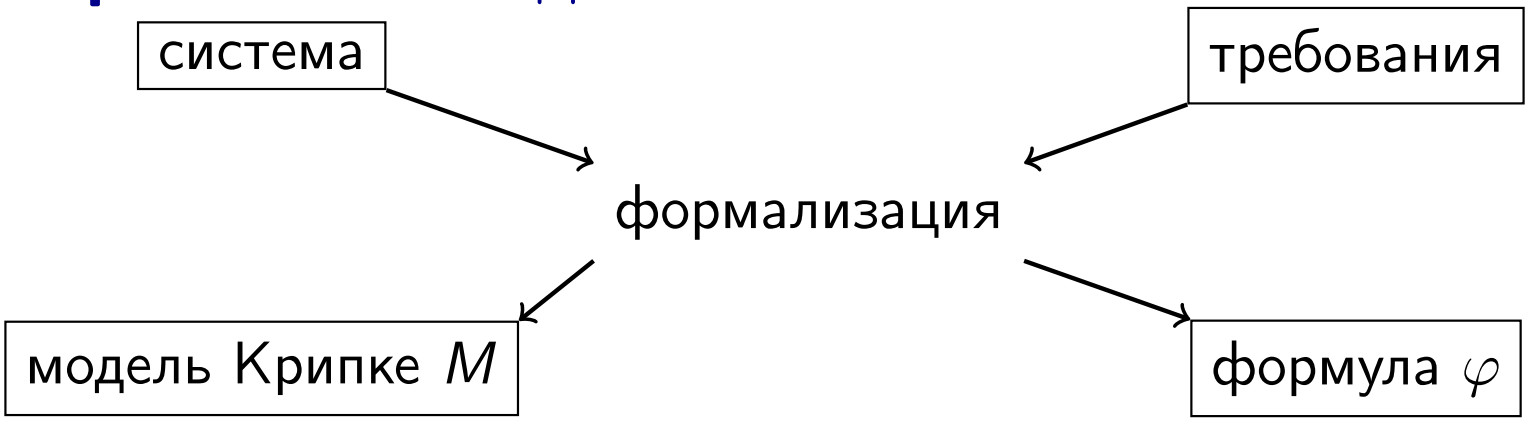
ответ на вопрос $M \overset{?}{\models} \varphi$

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

А может, никто так не делает,
и вообще верификация LTL никому не нужна?

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 (Promela: Process meta language) 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 желаемые модели Крипке на языке `Promela`
- как применять средство средство `Spin` для верификации LTL-формул

Многие конструкции языка `Promela` схожи с *такими же конструкциями C/C++ и/или NuSMV* как по синтаксису, так и по семантике (*вплоть до того, что некоторые части C/C++-кода можно особенным образом вставлять в модель Promela, 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 (*~объект/функция в C/C++, ~модуль в NuSMV*)

Описание типа процесса имеет следующий вид:

`proctype` *тип_процесса*`(`*аргументы*`)` `{`*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}
```

Система на каждом шаге работы содержит некоторое число запущенных процессов

Каждый процесс

- заданным образом **последовательно** изменяет переменные системы согласно описанию в теле типа процесса
- на каждом шаге работы системы имеет текущее состояние управления: значение счётчика команд, указывающее на команду, которая должна выполниться следующей

(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 запускаемого процесса — это первая команда в теле процесса

Состояние модели Крипке, соответствующей системе на языке `Promela`, в числе прочего содержит

- текущие значения всех глобальных переменных
- текущие состояния управления всех запущенных процессов

(S) Модель Крипке: состояния

```
bool b;

active proctype P() {
  do
  :: b = !b;
  od
}

ltl f1 {[]<>b}
ltl f2 {<>[]b}
```

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

Состояния управления процессов будут опускаться в иллюстрациях моделей для краткости и наглядности

(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}
```

Переменные изменяются совокупностью параллельно работающих процессов согласно семантике чередующихся вычислений

В частности, если в системе запущен один процесс, то он

- в рамках перехода выполняет следующую команду
- если выполнил все команды, то *может* завершиться (исчезнуть из системы)

(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}
```

Единственный процесс, запущенный в примере,

- выполняет безусловный бесконечный цикл `do-od`
- на каждом витке цикла выполняет *присваивание* значения `!b` в переменную `b`

Семантика языка устроена так, что **в данном примере** один виток цикла выполняется за один переход в модели Крипке, то есть после каждого перехода счётчик команд указывает на начало цикла

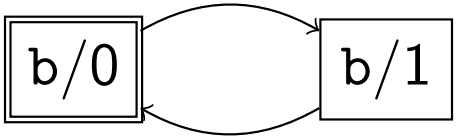
(S) Модель Крипке: переходы

```
bool b;

active proctype P() {
  do
  :: b = !b;
  od
}

ltl f1 {[]<>b}
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 LTL-свойств (φ):

φ::=*булево_выражение* | φ && φ | “φ || φ” | !φ |
φ -> φ | φ implies φ | φ <-> φ | φ equivalent φ
[] φ | always φ | <> φ | eventually φ |
φ U φ | φ until φ

Операция	В БНФ	Операция	В БНФ
импликация	->, eventually	равносильность	<->, equivalent
G	[], always	**F**	<>, eventually
U	U, until	**X**	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

2a. 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

2б. 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

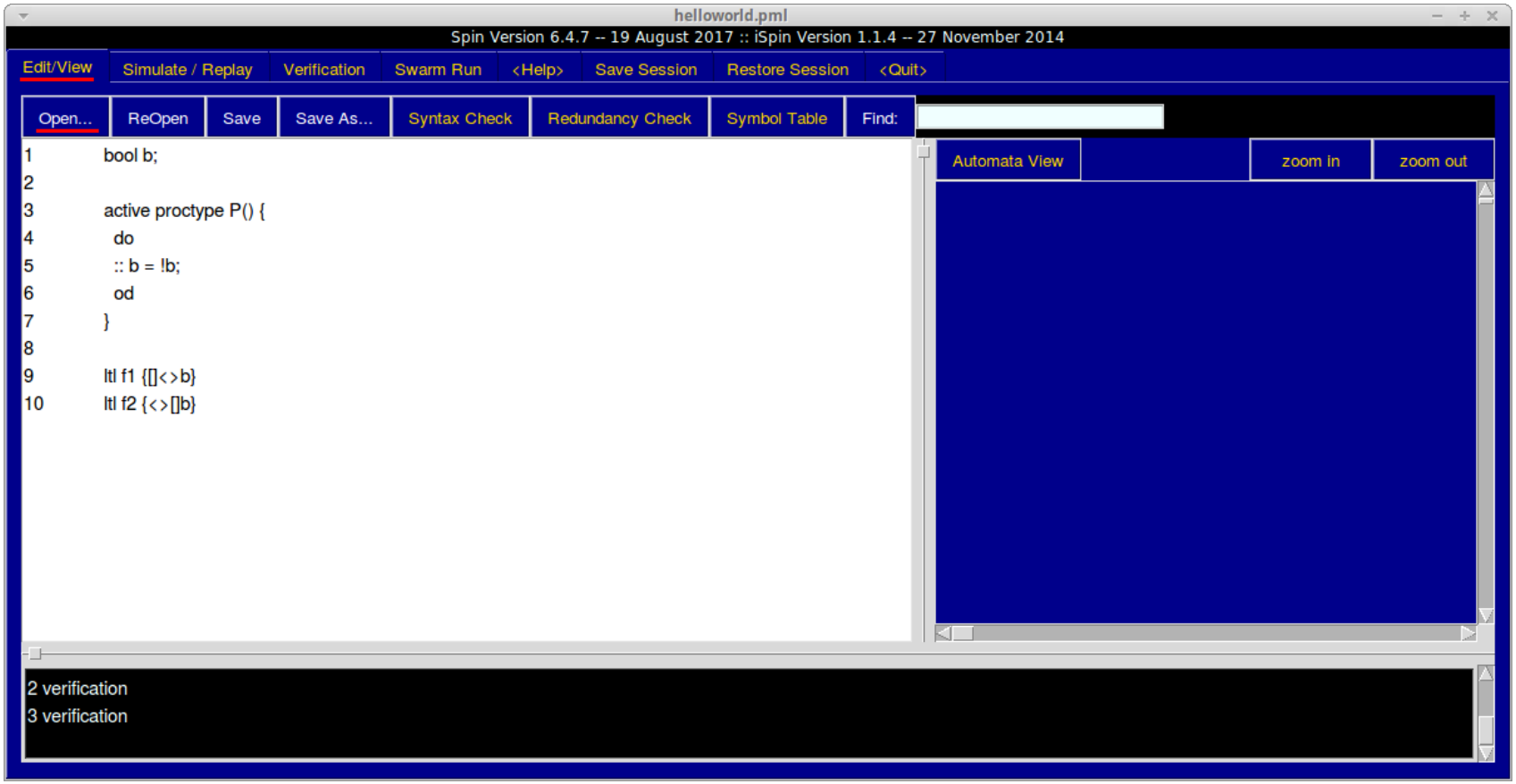
что-то не выполнено

Использование средства Spin

Способ 2: GUI ispin

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

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

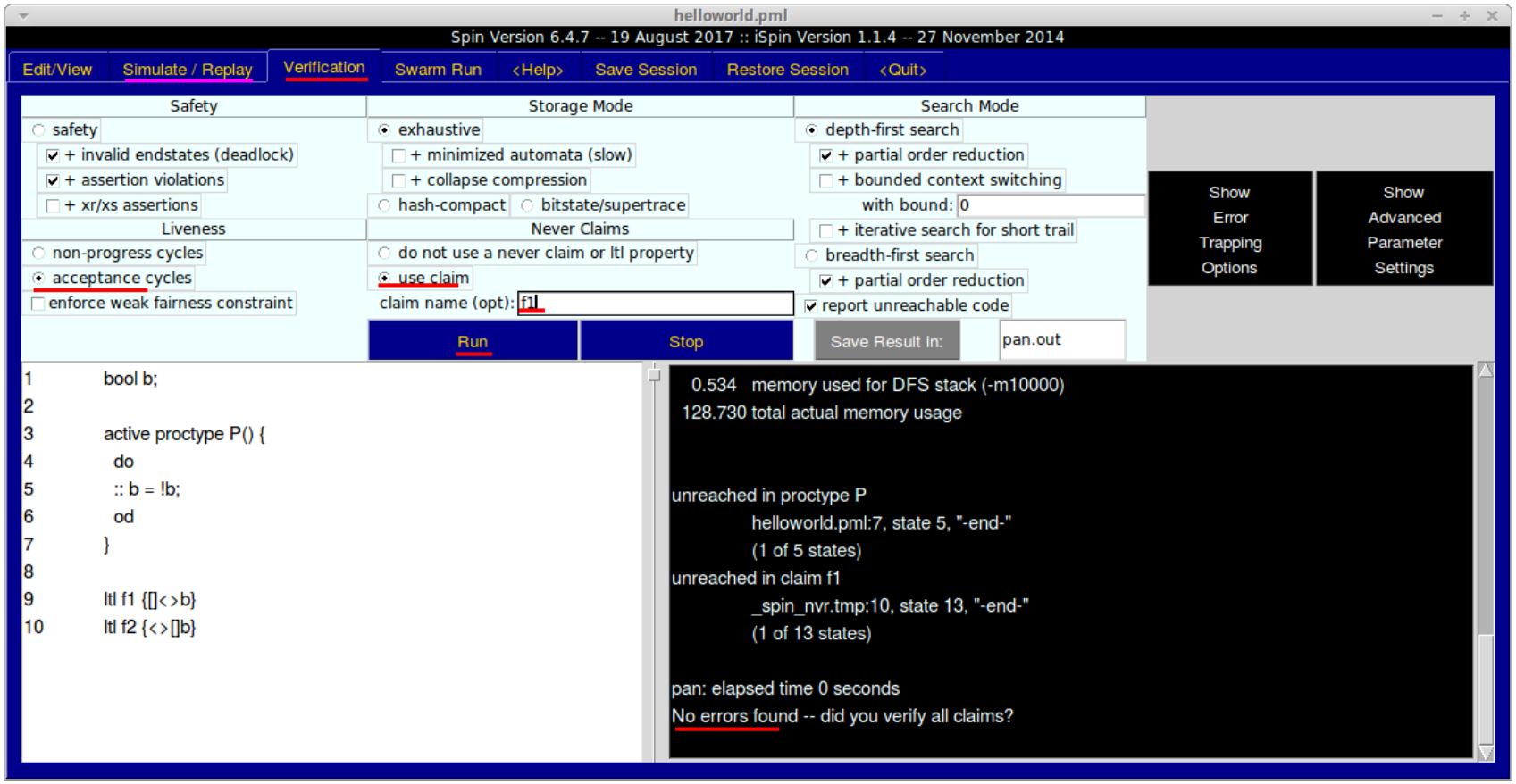


Использование средства Spin

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

```
#jSpin configuration file
#Wed Dec 15 09:27:07 IST 2010
VERIFY_OPTIONS=-a
FONT_SIZE=14
PAN_OPTIONS=-X
WIDTH=1000
INTERACTIVE_OPTIONS=-i -X
SELECT_MENU=5
WRAP=true
SELECT_BUTTON=220
SPIN=/path/to/spin/binary/spin
LR_DIVIDER=400
CHECK_OPTIONS=-a
VERIFY_MODE=Safety
VARIABLE_WIDTH=10
MSC=false
SOURCE_DIRECTORY=jspin-examples
TAB_SIZE=4
RAW=false
C_COMPILER_OPTIONS=-o pan pan.c
VERSION=6
COMMON_OPTIONS=-g -l -p -r -s
TRAIL_OPTIONS=-t -X
MAX_DEPTH=2000
TRANSLATE_OPTIONS=-f
C_COMPILER=gcc
STATEMENT_TITLE=Statement
DOT=dot
ABOUT_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

jSpin Version 5.0

File Edit Spin Convert Options Settings Output SpinSpin Help | LTL formula f1

Open | Check | Random | Interactive | Guided | Weak fairness | Acceptance | Verify | Stop | Translate | Load | LTL name | SpinSpid

helloworld.pml / -

```
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}
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
```

```
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 enabled)
                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)
   0.107        memory used for DFS stack (-m2000)
 128.302        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
```

```
Saved helloworld.pml
/home/pellman/working/programs/spin/Spin/Src6.4.7/spin -a helloworld.pml ... done!
gcc  -o pan pan.c ... done!
/home/pellman/working/projects/spin/to_verification_course/helloworld/pan  -f  -a  -m2000 -N f1 -X ... done!
```

(S) Простые типы данных

Многие встроенные типы данных, имеющиеся в `Promela`, *аналогичны типам с теми же названиями в 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 = 2;` — объявление беззнакового пятибитового числа `x` с начальным значением 2

(S) Непростые типы данных

- разрешено объявление одномерных массивов, *и выглядит это так же, как и в C*, с поправкой на инициализацию:
  - `byte state[4] = 1;` — объявление массива `state` из 4-х элементов типа `byte`, **каждый** из которых инициализируется числом 1
- `typedef` — это ключевое слово, *аналогичное `struct` в C*:
  - `typedef T {bool a; int b;};` — тип `T` с двумя полями
  - `typedef onedim {bool a[4];};` — “костыль”, позволяющий работать с многомерными массивами

(S) Непростые типы данных

- перечисление: `mtype = {`*имена_через_запятую*`}`
  - *аналогично enum в C* с некоторыми поправками
  - тип перечисления всегда один, и это тип `mtype`
    - `mtype x;` — объявление перечисляемой переменной `x`
  - несколько перечислений "сливаются" в одно записью перечисляемых имён в порядке следования в тексте
  - нумерация перечисляемых объектов начинается с 1, то есть перечисляемая переменная инициализируется значением 0, отличающимся от всех числовых эквивалентов перечисляемых 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

(S) Непростые типы данных

Пример

```
mtype = {A, B, C};
typedef T {
  bool a[5];
};
mtype = {D, E, F};
T twodim[3];
mtype symvar;
```

Здесь объявлены

- переменная `symvar`, которой можно присаивать значения `A`, `B`, `C`, `D`, `E`, `F`
- массив `twodim`, содержащий три элемента типа `T`:
  - каждый элемент содержит массив `a` из пяти булевых значений

(S) Композиция процессов

Каждая команда в теле процесса описывает **недетерминированный** способ изменения состояния системы: выполнение команды соответствует одному, нескольким или ни одному переходу в модели Крипке

В последнем случае команда и процесс, содержащий эту команду, считаются заблокированными в текущем состоянии, а иначе — активными

Шаг выполнения системы (*переход в модели Крипке*) выглядит так:

- недетерминированно выбирается активный процесс
- недетерминированно выполняется следующая команда процесса

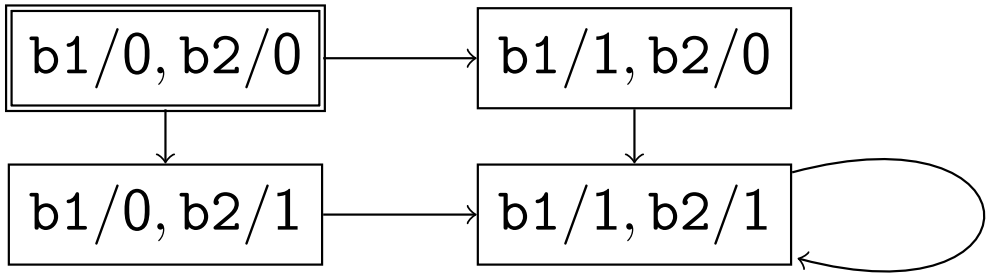
Если `Spin` применяется согласно инструкции, то в случае, когда все процессы заблокированы, в систему добавляется переход, при выполнении которого состояние системы не изменяется

(S) Композиция процессов

```
bool b1;
bool b2;

active proctype P() {b1 = !b1;}
active proctype Q() {b2 = !b2;}
```

Модель Крипке в примере выглядит так:



(S) Тело процесса

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

имя_метки : *команда*

Тело процесса содержит последовательность команд, разделённых “;”, включающую

- присваивания
- условия
- ветвления
- циклы
- *команду goto*

(S) Тело процесса: присваивание

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

имя = выражение
имя++
имя--

(S) Тело процесса: присваивание

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

- `+`, `-`, `*`, `/`
- `<<`, `>>`, `~`, `&`, `^`, `|`
- `<`, `>`, `<=`, `>=`, `==`, `!=`
- `!`, `&&`, `||`
- `->`: (*аналог `?:`*), `[]` (индексирование массивов), `.` (обращения к полям структур)

Присваивание всегда активно и выполняется за один переход в модели Крипке:

- переменная изменяется так же, *как в C*
- управление передаётся следующей команде последовательности

(S) Тело процесса: условие

Любое булево выражение может быть записано как условная команда

Условная команда активна $\Leftrightarrow$ значение выражения — `true`

При выполнении условной команды

- значения всех переменных остаются неизменными
- управление передаётся следующей команде последовательности

(S) Тело процесса: ветвление

```
if
:: последовательность_команд
...
:: последовательность_команд
fi
```

Альтернатива — это последовательность команд после “::”

Голова альтернативы — это первая команда

Альтернатива активна, если активна её голова

(S) Тело процесса: ветвление

```
if
:: последовательность_команд
...
:: последовательность_команд
fi
```

Команда ветвления активна $\Leftrightarrow$ активна хотя бы одна альтернатива

При выполнении активной команды ветвления

- недетерминированно выбирается одна из активных альтернатив
- команда ветвления заменяется на выбранную альтернативу, и переходы совершаются согласно этой альтернативе
  - первый переход для команды ветвления — это первый переход для выбранной альтернативы
- когда все команды альтернативы выполнены, управление передаётся следующей команде

(S) Тело процесса: ветвление

```
if
:: последовательность_команд
...
:: последовательность_команд
fi
```

Ключевое слово `else` — это специальная команда, работающая так:

- её можно писать непосредственно после "::", и не более чем в одной из альтернатив
- команда `else` активна $\Leftrightarrow$ все остальные альтернативы неактивны
- при выполнении команды `else` значения переменных не изменяются

Запись "`->`" — это синоним "`;`", повышающий наглядность записи альтернатив

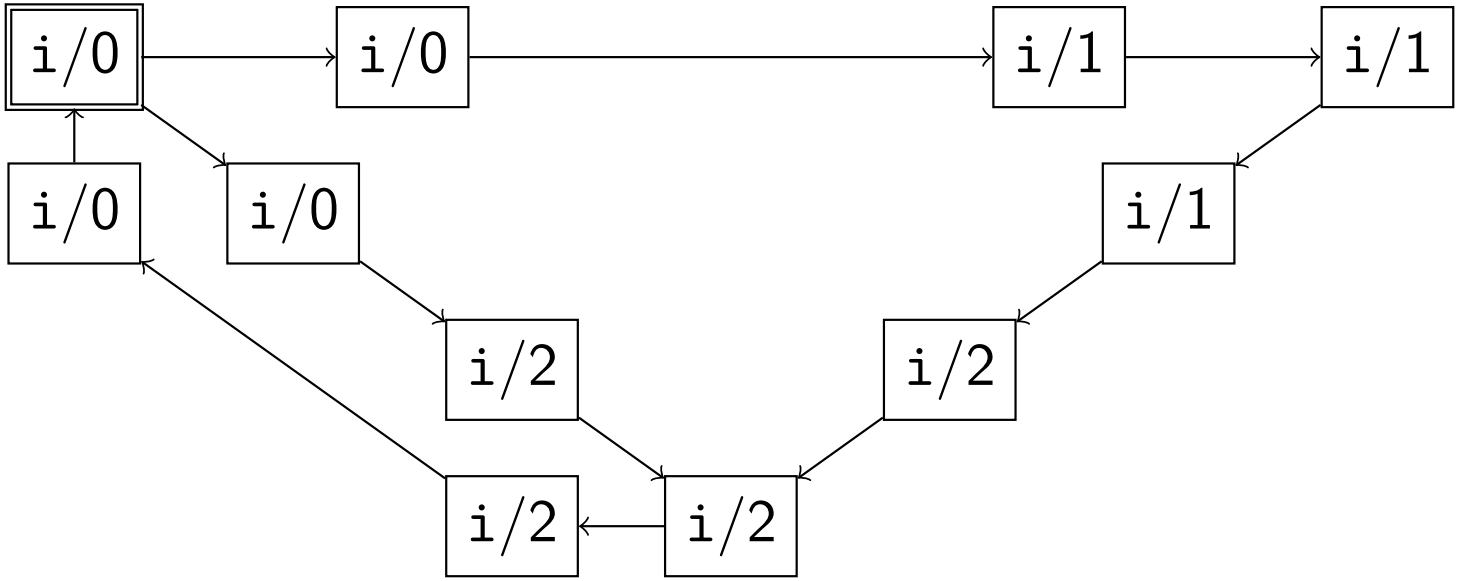
(S) Тело процесса: ветвление

Пример

```
byte i;

active proctype P() {
  L1: if
        :: i < 1 -> i++;
        :: i < 2 -> i = i + 2;
        :: else -> i = 0;
        fi;
  goto L1
}
```

Модель Крипке для этого примера выглядит так:



(S) Тело процесса: цикл

```
do
:: последовательность_команд
...
:: 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 инструкции, следующей за циклом

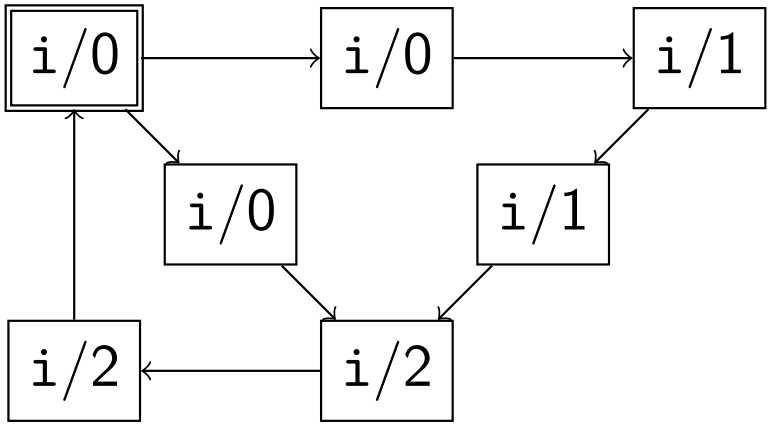
(S) Тело процесса: цикл

Пример

```
byte i;

active proctype P() {
  do
  :: i < 1 -> i++;
  :: i < 2 -> i = i + 2;
  :: else -> i = 0;
  od
}
```

Модель Крипке для этого примера выглядит так:



(S) Запуск и аргументы процессов

Специальной командой можно запускать новые процессы в системе:

`run` *тип_процесса*`(`*параметры*`)`

Эта команда всегда активна

Команда выполняется за один переход, выполнение передаётся следующей команде, и в системе запускается процесс указанного типа

В общем случае тип процесса имеет аргументы:

- аргументы — это список *объявлений*, разделённых “`;`”
- *объявление* `::=` *тип* *список_имён_через_запятую*

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

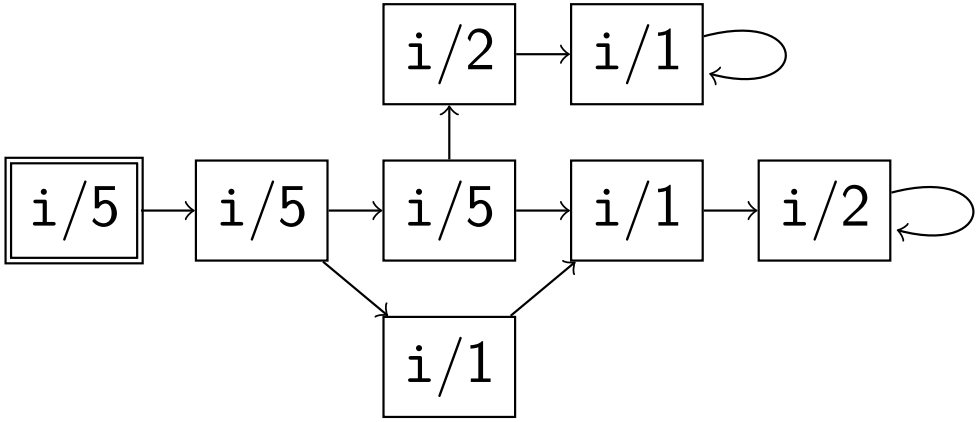
(S) Запуск и аргументы процессов

```
byte i = 5;

proctype Q(byte a) {i = a;}

active proctype P() {
  run Q(1);
  run Q(2);
}
```

Модель Крипке для этого примера выглядит так:



(S) Локальные переменные

Помимо глобальных переменных `Promela` позволяет объявлять и локальные переменные в начале текста тела процесса

Объявления локальных переменных выглядят так же, как и объявления глобальных

Локальные переменные создаются для каждого процесса в момент его запуска

Инициализация локальных переменных работает так же, как и для глобальных переменных

Объявление локальной переменной не является командой

(S) Каналы связи

Язык `Promela` позволяет объявлять каналы связи двух видов:

- синхронные
- асинхронные

Асинхронный канал способен хранить количество сообщений, не превосходящее его ёмкость (положительное целое число) и объявляется так:

```
chan имя_канала = [ёмкость] of {тип_сообщений}
```

Синхронный канал связи — это канал ёмкости 0:

```
chan имя_канала = [0] of {тип_сообщений}
```

(S) Каналы связи

Команда посылки сообщения в канал имеет вид

<имя канала>!<выражение>

Команда приёма сообщения из канала имеет вид

<имя канала>?<имя переменной>

или

<имя канала>?<выражение>

В зависимости от ёмкости канала эти команды работают по-разному

Асинхронный канал:

- команда посылки сообщения активна $\Leftrightarrow$ канал хранит сообщений меньше, чем его ёмкость
- команда приёма сообщения активна $\Leftrightarrow$ канал хранит хотя бы одно сообщение и (*в случае инструкции второго вида*) значение сообщения совпадает со значением выражения

(S) Каналы связи

Команда посылки сообщения в канал имеет вид

<имя канала>!<выражение>

Команда приёма сообщения из канала имеет вид

<имя канала>?<имя переменной>

или

<имя канала>?<выражение>

В зависимости от ёмкости канала эти команды работают по-разному

Асинхронный канал:

- по выполнении команды посылки сообщения в канал добавляется сообщение с тем же значением, что и выражение в инструкции
- по выполнении команды приёма первого вида в переменную записывается значение сообщения, посланного раньше остальных
- по выполнении команды приёма обоих видов прочитанное сообщение удаляется из канала

(S) Каналы связи

Команда посылки сообщения в канал имеет вид

<имя канала>!<выражение>

Команда приёма сообщения из канала имеет вид

<имя канала>?<имя переменной>

или

<имя канала>?<выражение>

В зависимости от ёмкости канала эти команды работают по-разному

Синхронный канал:

- команды посылки и приёма сообщения выполняются **одновременно**, каждая ровно в одном процессе
- посылаемое значение напрямую передаётся принимающему процессу
- в остальном передача сообщения происходит так же, как для асинхронных каналов

(S) Каналы связи

```
1  chan c = [0] of {byte};
2  byte i, j;
3
4  active proctype P() {
5    do
6    :: c! j+1;
7    :: c? j;
8    od
9  }
10
11 active proctype Q() {
12   do
13   :: c! i+1;
14   :: c? i;
15   od
16 }
```

В этой системе два процесса постепенно увеличивают значения переменных `i`, `j` (*с переполнением*) пересылая увеличенные значения через синхронный канал связи

Команда “**!**” одного процесса и команда “?” другого процесса выполняются одновременно за один переход

(S) Неделимые последовательности инструкций

А что делать, если хочется выполнить последовательность команд за один шаг выполнения системы?

Например, в *протоколах доступа к критической секции* часто бывает важным уметь неделимо проверять и изменять значение переменной (*семафора*)

Неделимость последовательности команд моделируется в языке PROMELA так:

`atomic{<`*команды*`>}`

Как это работает: (*в простом случае*)

1. если сейчас выполнилась команда из блока `atomic` и на следующем шаге есть возможность выполнить команду из этого же блока, то эта команда обязательно выполняется

(S) Неделимые последовательности инструкций

А что делать, если хочется выполнить последовательность команд за один шаг выполнения системы?

Например, в *протоколах доступа к критической секции* часто бывает важным уметь неделимо проверять и изменять значение переменной (*семафора*)

Неделимость последовательности команд моделируется в языке PROMELA так:

`atomic{<`*команды*`>}`

Как это работает: (*в простом случае*)

2. если сейчас выполнилась команда из блока `atomic`, но ни одна команда из этого блока больше не может выполниться, то выполнение системы происходит обычным образом

(S) Неделимые последовательности инструкций

А что делать, если хочется выполнить последовательность команд за один шаг выполнения системы?

Например, в *протоколах доступа к критической секции* часто бывает важным уметь неделимо проверять и изменять значение переменной (*семафора*)

Неделимость последовательности команд моделируется в языке PROMELA так:

`atomic{<`*команды*`>}`

Как это работает: (*в простом случае*)

3. если при выполнении обычным образом *вдруг* выполнилась ещё одна команда из блока `atomic`, то дальнейшее выполнение опять происходит согласно пунктам 1, 2

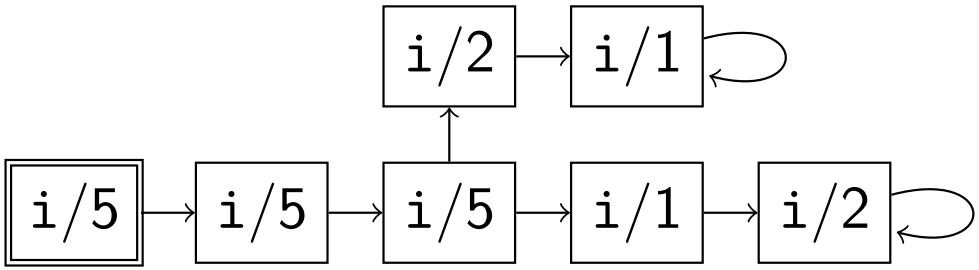
(S) Неделимые последовательности инструкций

```
byte i = 5;

proctype Q(byte a) {i = a;}

active proctype P() {
  atomic{
    run Q(1);
    run Q(2);
  }
}
```

Модель Крипке для этого примера выглядит так:



(S) Пример для размышлений

```
bool near, dead, hunted;
mtype = {ping}; chan c = [0] of {ping};
active proctype mosquito() {   do
  :: !near -> atomic{near = true; c!ping;}
  :: near && !hunted && !dead -> near = false;
  od }
active proctype bird() {   do
  :: atomic{c?ping -> hunted = true;}
  :: hunted && near -> dead = true; hunted = false;
  :: hunted && !near -> hunted = false;
  od }
ltl inevitable_death {<>(near && <> dead)}
ltl cannot_fly_away {[](near -> <> dead)}
ltl reliable_hunt {[](hunting -> <> dead)}
```

Как это работает, и какие свойства выполнены?

Конец семинара 7