Formal techniques for software and hardware verification

Lecturers:

Vladimir Zakharov

Vladislav Podymov

e-mail:

valdus@yandex.ru

2020, fall semester

Seminar 7

Spin

tool overview

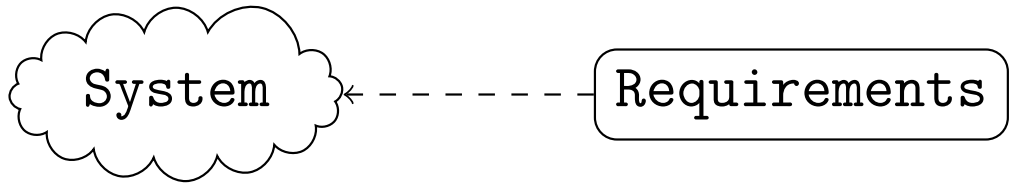
Considered PROBLEM

Given: informal descriptions of

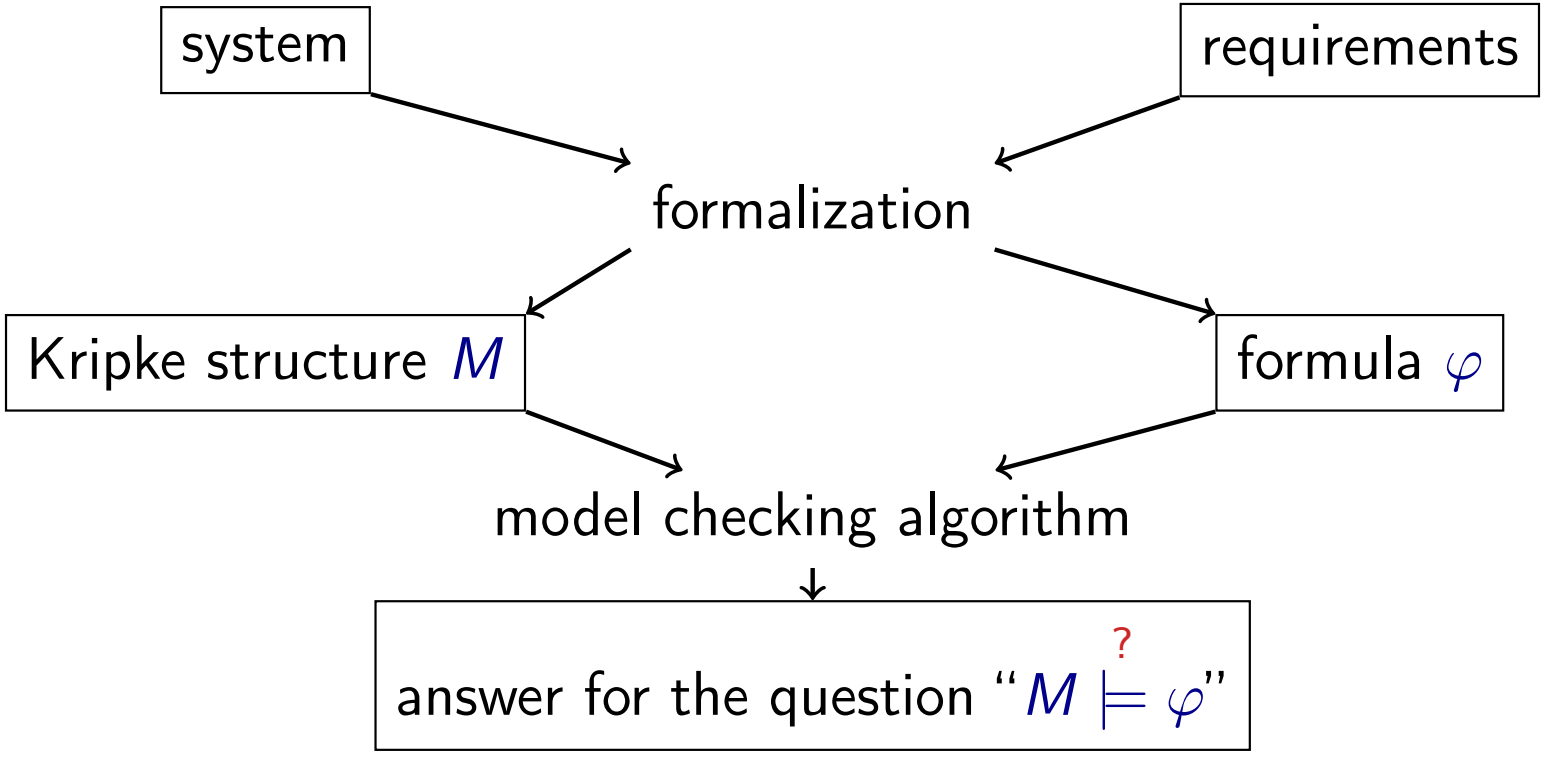
- a system, and
- requirements

Check whether

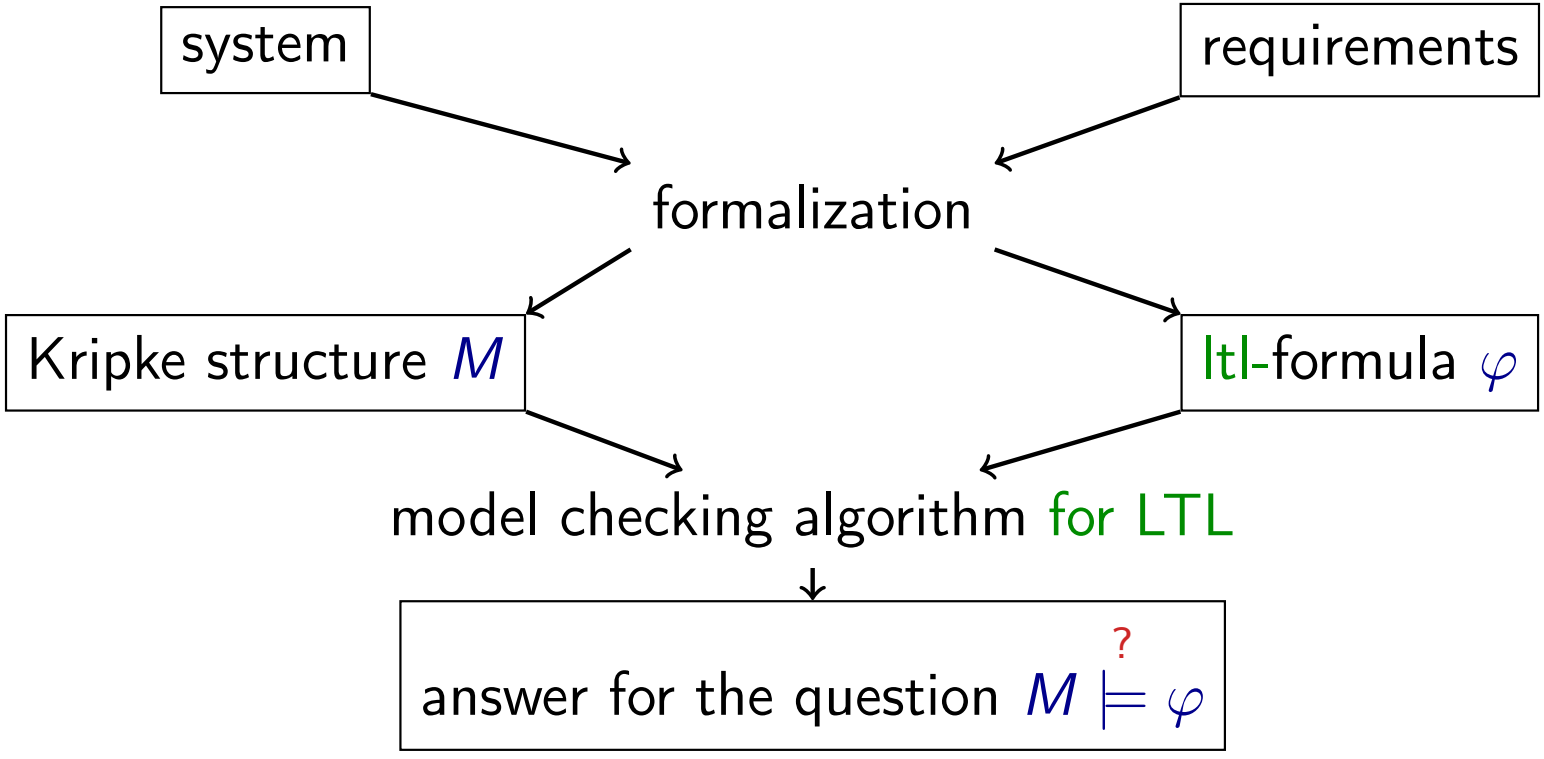
the system satisfies all the requirements



General solution scheme for the PROBLEM

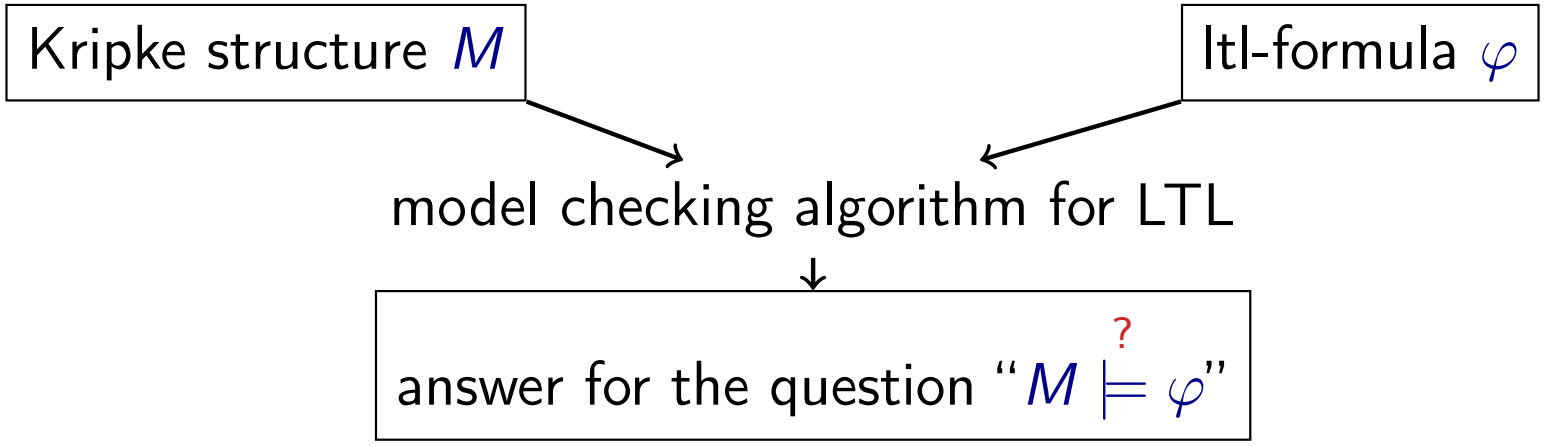


General solution scheme for the PROBLEM



Spin-related seminars are all about linear temporal logic

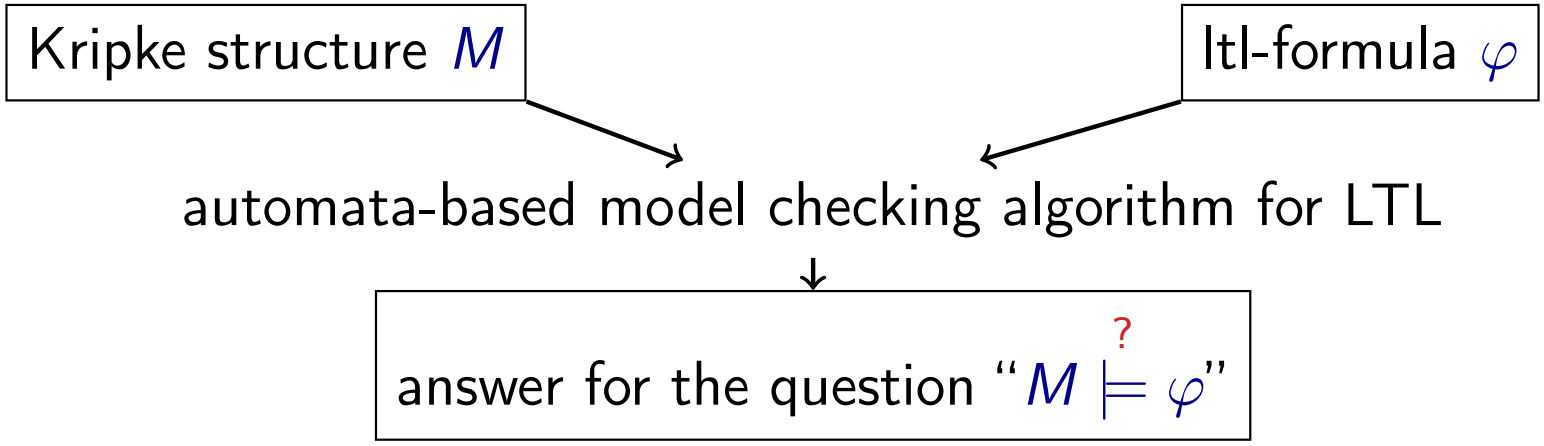
General solution scheme for the PROBLEM



You already know *at least* two model-checking algoirthms for LTL:

- a **tableau-based** algorithm
  - clear and *relatively* simple
  - lies in the base of other algorithms
  - inefficient
- an **automata-based** algorithm
  - harder to understand, but not much
  - much more efficient (**?**)

General solution scheme for the PROBLEM

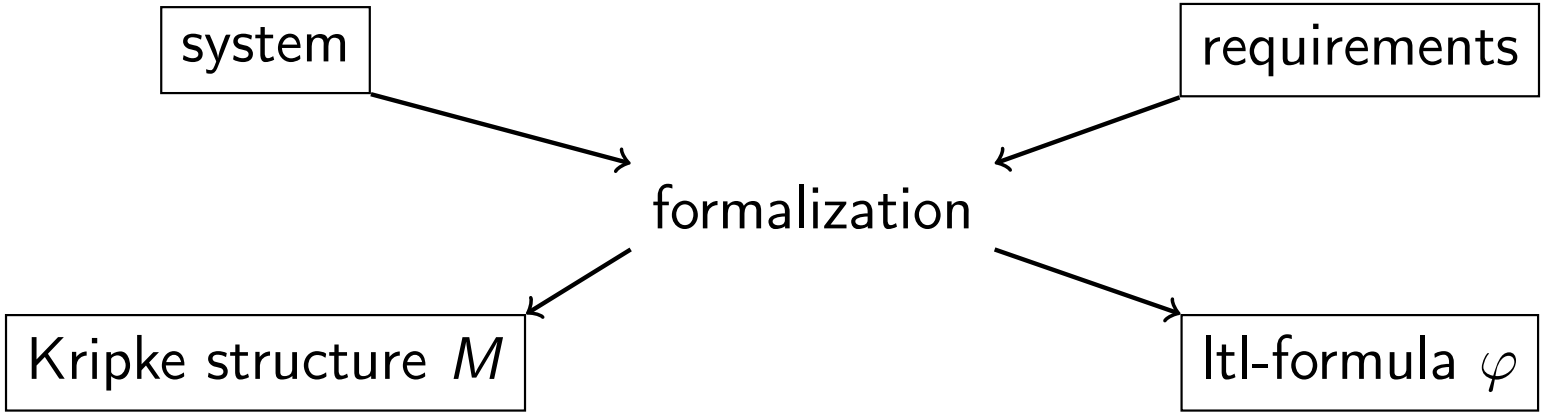


Efficiency of automata-based algorithms depends on efficiency of

1. Büchi automata construction, and
2. search algorithms for strongly connected components

An LTL model checking tool usually has an implementation of an automata-based algorithm in its core

General solution scheme for the PROBLEM



The formalization stage remains the same as it was discussed for CTL

General solution scheme for the PROBLEM

For those interested, here are several tools capable of LTL model checking for *some* models:

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

Disclaimer: *these are just several tools randomly taken from a related Wikipedia page some time ago*

Some of the tools are capable of CTL model checking as well

We will focus on the tool Spin:

- it is open-source and free
- it is quite popular (*often considered as a starting point for practical model checking*)
- its language (Promela: Process meta language) is easy to understand

(*easier than the NuSMV language*)

(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}
```

Let us start with a simple example which shows general ideas of

- how `Promela` models are related to Kripke structures, and
- how to use `Spin` tool for model checking against LTL

Some `Promela` language features are similar (or even identical) to those of *C/C++* and *NuSMV* languages
(*up to inline C code allowed in some places of `Promela` models*)

(S) Corresponding Kripke structure: states

```
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}
```

Line 1 is a *declaration* of a *global* `boolean` *variable* `b`

This variable

- stores two possible values:
  `0` (alias `false`) and `1` (alias `true`)
- is initialized to `0` by default

(S) Corresponding Kripke structure: states

```
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}
```

P is a process type

(similar to a *class/function in C/C++*, or a *module in NuSMV*)

A process type declaration is structured as follows:

`proctype` *<process_type>* `(` *<parameters>* `)` `{` *<body>* `}`

(S) Corresponding Kripke structure: states

```
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}
```

At each execution state of `Promela` system

- some (arbitrary) number of processes of declared types are being executed, and
- a current statement is defined for each process

A state of a `Promela` system
= values of all variables (a *data state*)
+ processes together with their current statements (a *control state*)

(S) Corresponding Kripke structure: states

```
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}
```

A keyword `active` prior to a process type means that one process of this type is included in the (single) initial state of the system

A current statement of each process in the initial state is the first statement of its body

(S) Corresponding Kripke structure: states

```
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}
```

Let us assume that processes of type P have exactly one control state
(*transition specifics is to be explaned later*)

Then the corresponding Kripke structure has exactly two states, and exactly one of them is initial:

(*control states are omitted in pictures*)

(S) Corresponding Kripke structure: transitions

```
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}
```

If a state contains exactly one process, then the process executes sequentially in a natural way (*similar to C*)

For instance, the process above

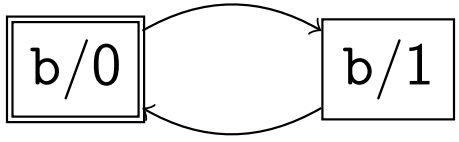
- contains an unconditional infinite loop (`do-od`)
- negates the variable `b` at each loop iteration
- executes one iteration during a transition

(*to be explained later*)

(S) Corresponding Kripke structure: transitions

```
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}
```

States and transitions of the Kripke structure corresponding to the system above:



(S) LTL properties

```
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}
```

Like in NuSMV, in `Promela` a property is included into a system description

LTL property is defined as follows:

- `ltl` *<property_name>* `{` *<property_body>* `}` for named properties
- `ltl {` *<property_body>* `}` for unnamed properties
  - an unnamed propety is used if it is the only LTL property of the system

(S) LTL properties

```
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}
```

BNF for Promela LTL formulae (φ):

φ ::= *<boolean_expression>* | 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

In math	In Promela	In math	In Promela
$\rightarrow$	->, eventually	$\equiv$	<->, equivalent
G	[], always	**F**	<>, eventually
U	U, until	**X**	absent

Spin tool usage

Option 1: Linux terminal

1. Compile sources into a verifier executable

```
> 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
```

Spin tool usage

Option 1: Linux terminal

2a. Execute the verifier with a flag "check this property", and make sure that the answer is "yes"

```
> ./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
```

Spin tool usage

Option 1: Linux terminal

2b. Execute the verifier with a flag “check this property”, and make sure that the answer is “no”

```
> ./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

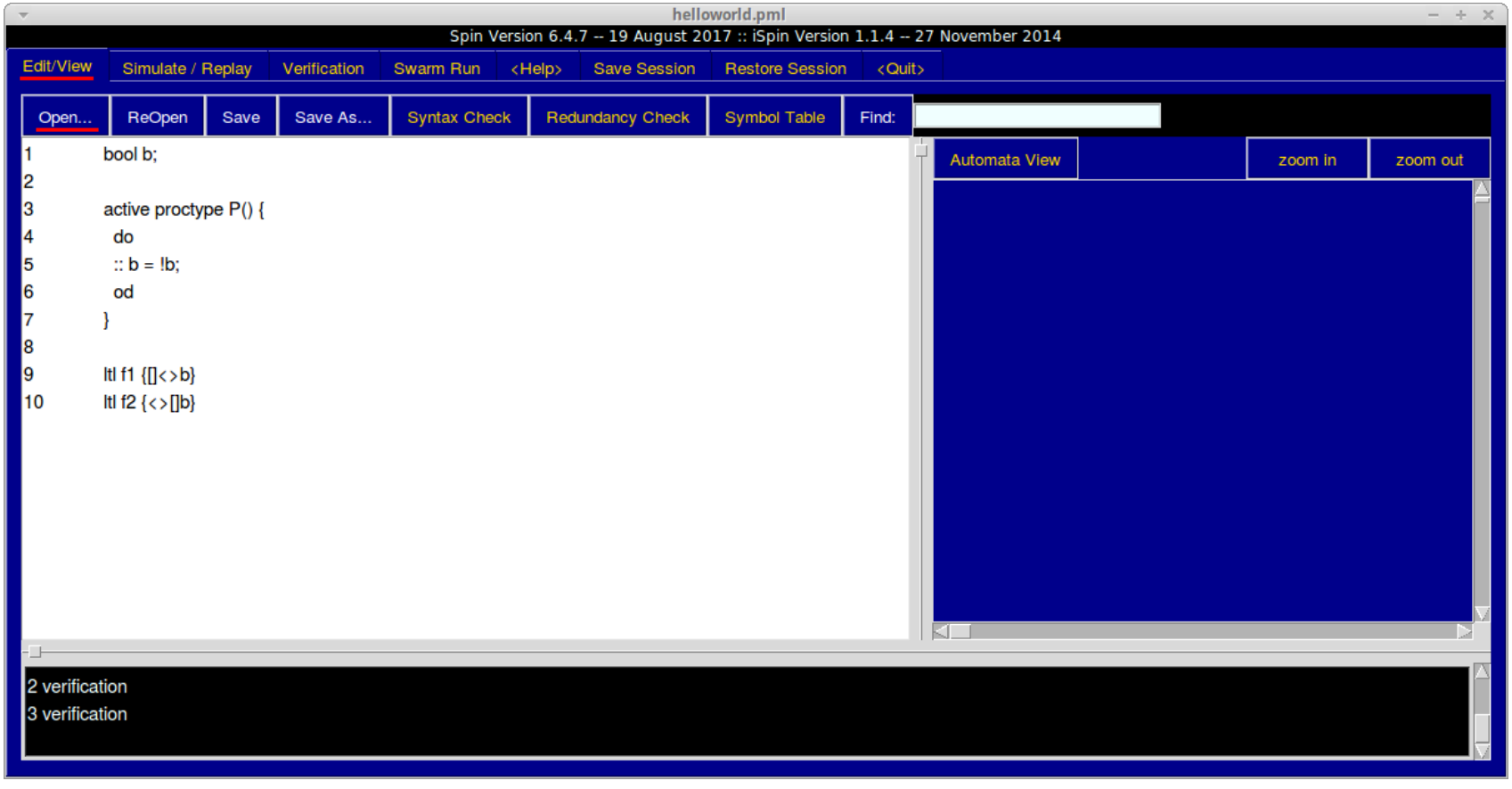
что-то не выполнено

Spin tool usage

Option 2: GUI ispin

ispin is an official tcl/tk-based graphical interface for spin

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

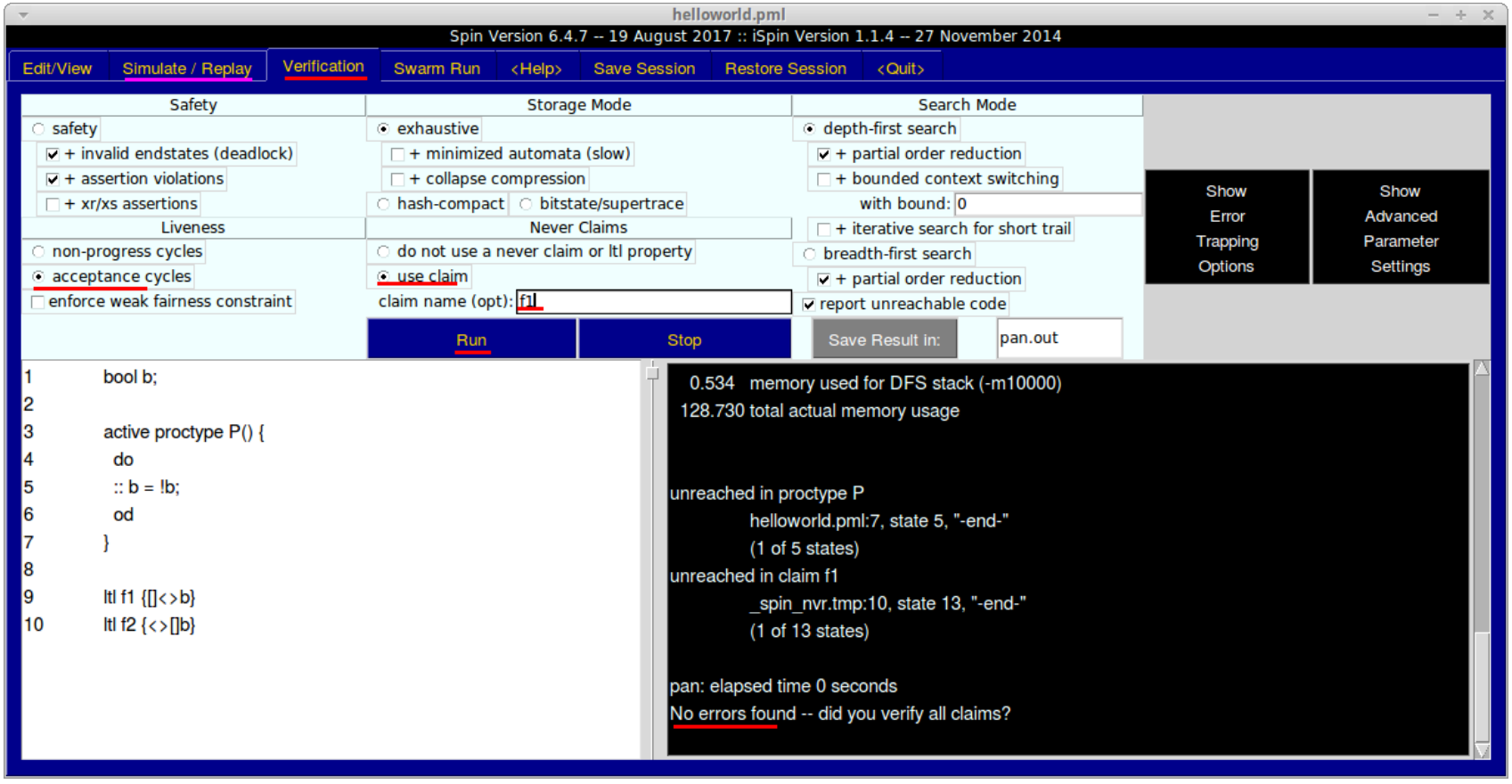


Spin tool usage

Option 2: GUI ispin

ispin is an official tcl/tk-based graphical interface for spin

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



Spin tool usage

Option 3: GUI jspin

jspin is a third-party java-based graphical interface for spin

config.cfg

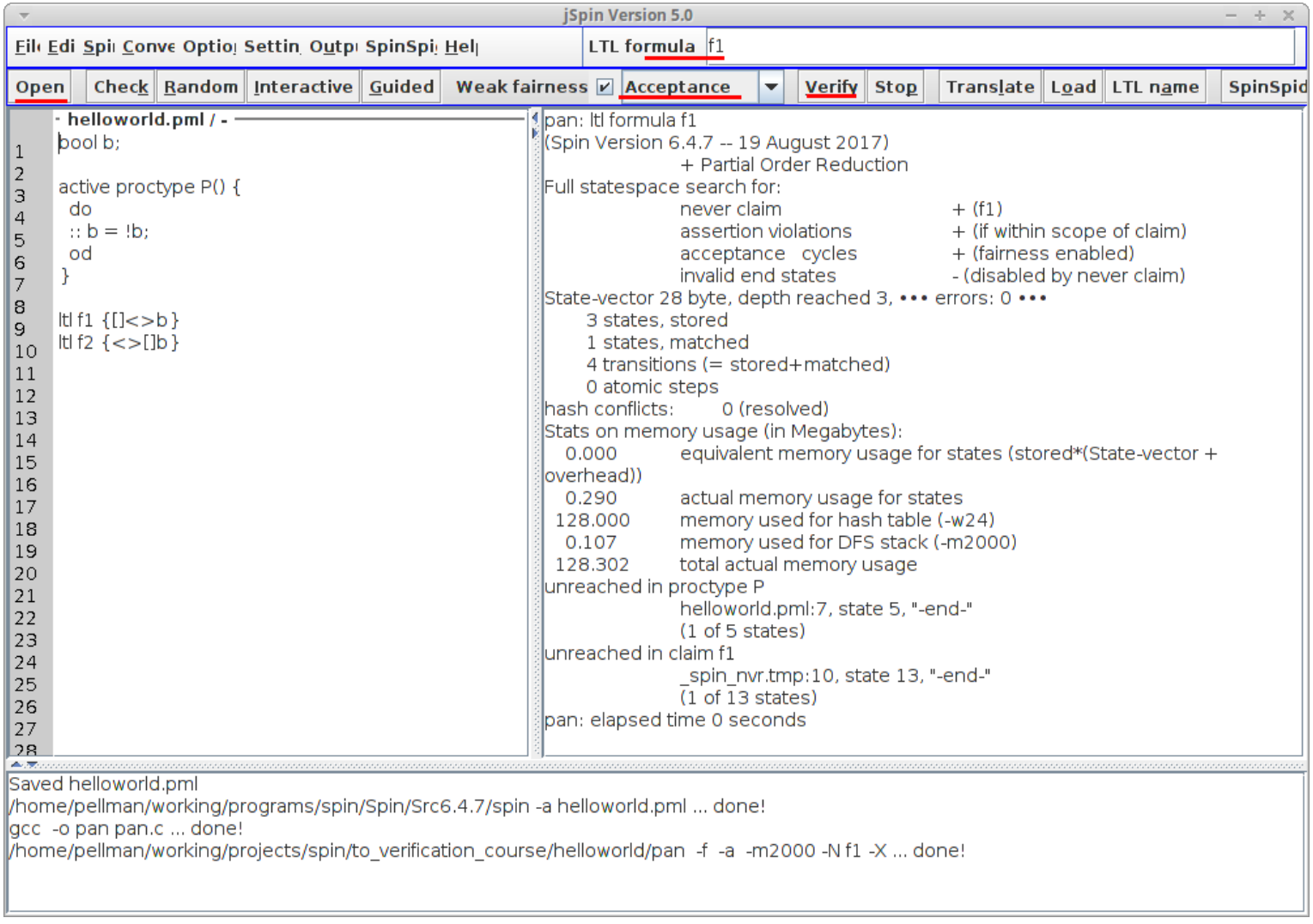
```
#jSpin configuration file
#Wed Dec 15 09:27:07 IST 2010
VERIFY_OPTIONS=-a
FONT_SIZE=14
PAN_OPTIONS=-X
WIDTH=1000
INTERACTIVE_OPTIONS=-i -X
SELECT_MENU=5
WRAP=true
SELECT_BUTTON=220
SPIN=/path/to/spin/binary/spin
LR_DIVIDER=400
CHECK_OPTIONS=-a
VERIFY_MODE=Safety
VARIABLE_WIDTH=10
MSC=false
SOURCE_DIRECTORY=jspin-examples
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
```

Spin tool usage

Option 3: GUI jspin

jspin is a third-party java-based graphical interface for spin



(S) "Simple" data types

Some `Promela` types are similar to *C types*:

- `bool`: 0 (`false`), 1 (`true`)
- `bit`: alias for `bool`
- `byte`: all integers between 0 and 255
- `short`: all integers between $-2^{15} - 1$ and $2^{15} - 1$
- `int`: all integers between $-2^{31} - 1$ and $2^{31} - 1$
- `unsigned`: unsigned integers stored in a fixed number of bits explicitly written in a declaration:
  - "`unsigned x : N;`" declares an unsigned `N`-bit integer `x`

Default values for all these types are 0

Non-default initialization is similar to *C non-default initialization*:

<type> <variable> = <value> ;

(S) "Hard" data types

One-dimensional arrays look *just like in C*, excluding initialization details:

- `byte x[4];`: `x` is a one-dimensional array of 4 elements of type `byte`, and each element is initialized to 0
- `byte x[4] = 1;`: the same, but each element is initialized to 1

Structures are defined with a keyword `typedef`, *instead of "struct" in C*:

- `typedef T {bool a; int b;};`: `T` is a structure with a boolean field `a` and an integer field `b`
- `typedef onedim {bool a[4];};`: this is how multidimensional arrays are declared

Array elements and structure fields are accessed *just like in C*:

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

(S) "Hard" data types

`mtype` is a special type similar to *enum in C*, but much more tricky:

- `mtype` is a type name (like `bool` and `int`)
- declarations have the following form: `mtype = {name_1, ..., name_N};`
- all declarations are merged into one containing all the declared names
- default value is 0, and this value differs from all names

Remark: "`mtype`" = "`message type`"; this type is intended to represent message types, but are to be explained later, and technically it **is** an enumeration

Example:

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

(S) Process composition

In each state each current statement is

- either active: it is possible to execute the statement
  (and the corresponding process is active) —
- or inactive (blocked): the statement cannot be executed
  (and the corresponding process is inactive/blocked)

An execution of the active statement corresponds to one or more transitions in the corresponding Kripke structure
(more than one transition means a nondeterminism)

(S) Process composition

Execution steps of a `Promela` system are based on the *interleaving semantics*:

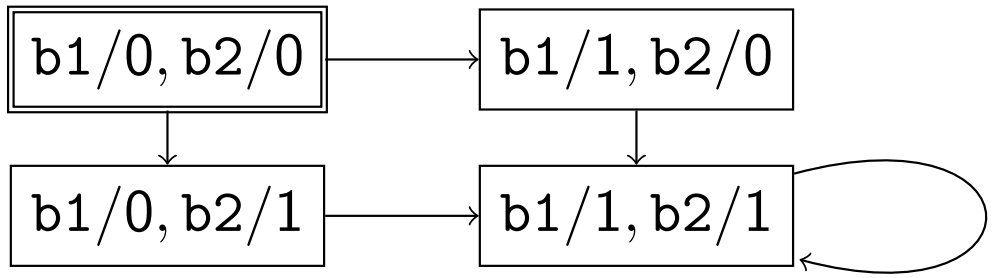
- an active process is nondeterministically picked, and its current statement is nondeterministically executed
- outgoing Kripke structure transitins cover **all** nondeterministic choices
- if all processes are blocked, then *by default* a loop transition is added (from the state to itself)

In special (explicitly discussed) cases several processes can be picked, and their statements are executed simultaneously (*synchronously*)

(S) Process composition

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

The system above corresponds to the following Kripke structure:

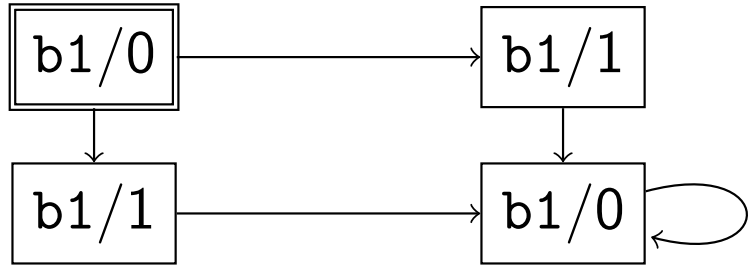


(S) Activation of several processes

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

To put `N` processes of the same type in the initial state, it is sufficient to write "`[N]`" after "`active`"

The system above corresponds to the following Kripke structure:



(S) Proctype body

Proctype body is a sequence of statements separated by ";"

Like in C, each statement can be preceded by a label:

`<label> : <statement>`

The most important statements are:

- *assignments*
- conditional statements (*with no analogues in C/C++*)
- *if-statement*
- *unconditional loop* (do-statement)
- *goto*

(S) Proctype body: assignments

An assignment looks *just like in C with restricted syntax*:

```
<variable>++
<variable>--
<variable> = <expression>
```

The assignment is always active

An execution of the assignment is defined naturally:

- the *<variable>* changes as written
  (increases or decreases by 1;
  stores the value of the *<expression>*)
- the next statement becomes current

(S) Expressions

An expression is made up of variables, constants (integers, `true`, `false`, enumeration names) and a number of operations *analogous to the C operations*:

- arithmetic: `+`, `-`, `*`, `/`
- bitwise: `<<`, `>>`, `~`, `&`, `^`, `|`
- relations: `<`, `>`, `<=`, `>=`, `==`, `!=`
- logic: `!`, `&&`, `||`
- ternary: `->:` ("->" instead of "?")
- indexing: `[]`
- field access: `.`

(S) Proctype body: conditional statement

Conditional statement is any boolean expression

The conditional statement is active $\Leftrightarrow$ the expression evaluates to `true`

An execution of the conditional statement:

- the variables remain unchanged
- the next statement becomes current

(S) Proctype body: if-statement

```
if
:: <alternative>
...
:: <alternative>
fi
```

An alternative is a nonempty sequence of statements placed after "::"

A head of the alternative is the first statement of the sequence

The alternative is active $\Leftrightarrow$ its head is active

The if-statement is active $\Leftrightarrow$ at least one its alternative is active

An execution of the if-statement:

- one of the active alternatives is picked (nondeterministically)
- the statement is replaced with the picked alternative
- the head of the picked alternative is executed

(S) Proctype body: if-statement

```
if
:: <alternative>
...
:: <alternative>
fi
```

`else` is a special *conditional statement*:

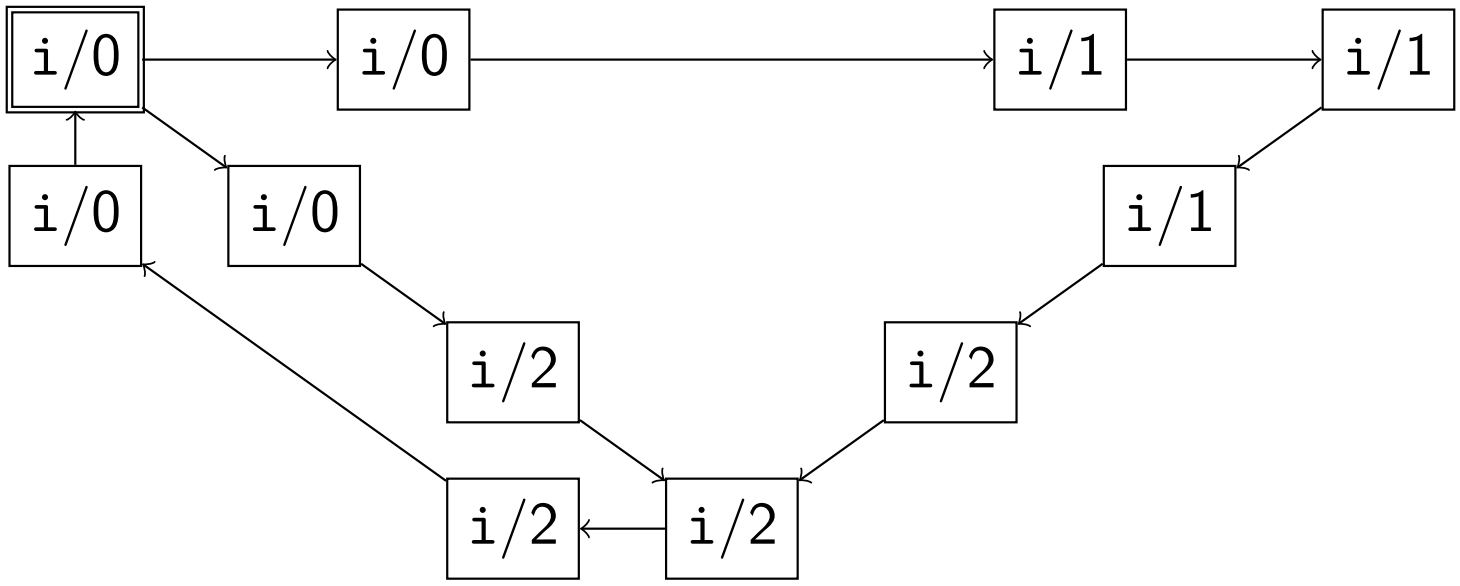
- it is allowed as a head of at most one of the alternatives
- the statement is active $\Leftrightarrow$
  all other alternatives are blocked

To increase readability, sometimes "`;`" is replaced with an alias "`->`"

(S) Proctype body: if-statement

```
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 }
```

The system above corresponds to the following Kripke structure:



(S) Proctype body: loop

```
do
:: <alternative>
...
:: <alternative>
od
```

A loop statement is similar to an if-statement (in which "if-fi" is replaced with "do-od")

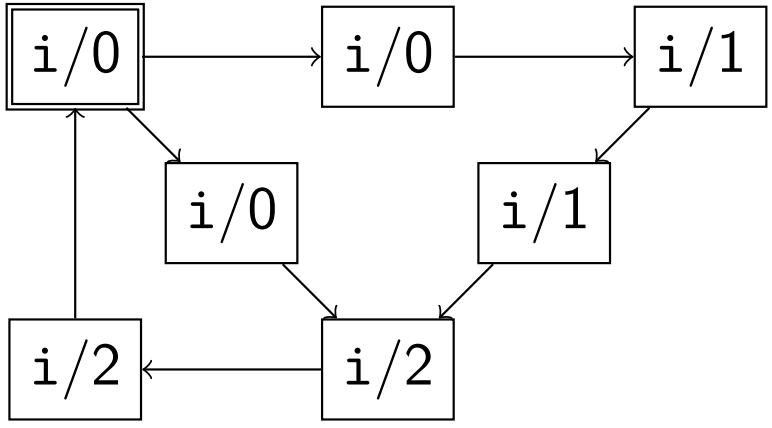
The only difference: when all the statements of the picked alternative are executed, the loop becomes current again

A keyword `break` is an always-active statement which do not change variables and forces the statement next to the loop to become current

(S) Proctype body: loop

```
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 }
```

The system above corresponds to the following Kripke structure:



(S) Channels

```
chan <channel> = [<capacity>] of {<type>};
```

Communication channels are declared in the same way and place as global variables

If `<capacity> == 0`, then the channel is synchronous, otherwise asynchronous (and `<capacity> > 0`)

Channels are made to transfer messages: values of the given `<type>`

The channel contains a message *queue* of the given `<capacity>`

Special statements are used to

- send a message to the channel (*enqueue*)
- receieve a message from the channel, storing it to a variable if necessary (and *dequeue*)

(S) Channels (asynchronous)

```
chan <channel> = [<capacity>] of {<type>};
                                 (<capacity> > 0)
```

Send statement: (*<expression>* is not a variable)

```
<channel>!<expression>
```

The statement is active $\Leftrightarrow$
the size of the *<channel>* queue is less than the *<capacity>*

An execution of the statement:

- the evaluated value of the *<expression>* is enqueued
- the next statement becomes current

(S) Channels (asynchronous)

```
chan <channel> = [<capacity>] of {<type>};
                                 (<capacity> > 0)
```

Receive statement:

```
<channel>?<expression>
```

The statement is active $\Leftrightarrow$ the `<channel>` queue is not empty and the next dequeued value equals to the value of the `<expression>`

An execution of the statement:

- variables remain unchanged
- a message is dequeued from the channel
- the next statement becomes current

(S) Channels (asynchronous)

```
chan <channel> = [<capacity>] of {<type>};
                                 (<capacity> > 0)
```

Receive-and-store statement:

```
<channel>?<variable>
```

The statement is active $\Leftrightarrow$ the `<channel>` queue is not empty

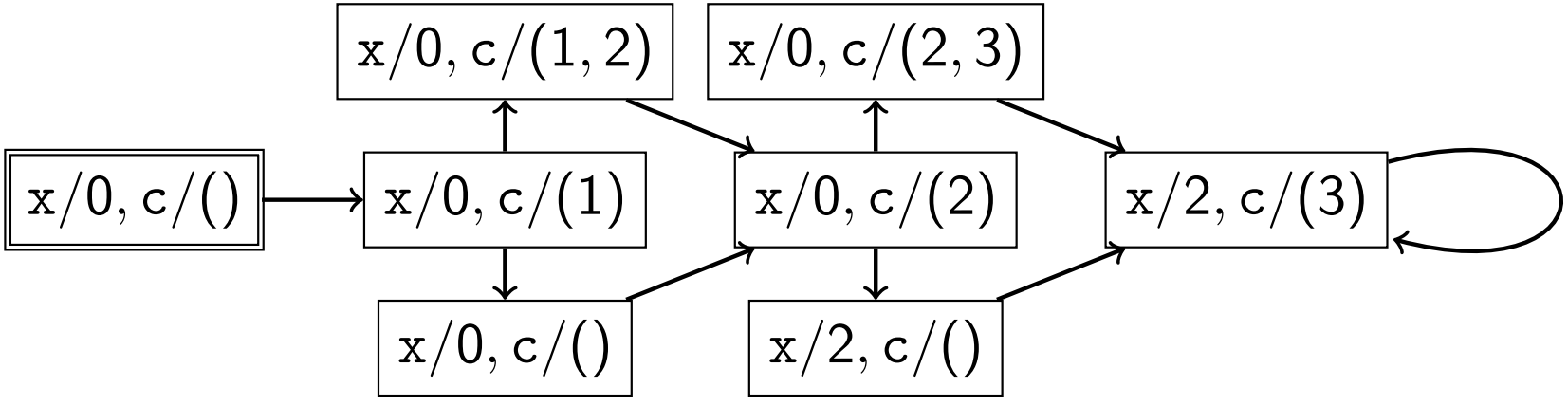
An execution of the statement:

- a message is dequeued from the `<channel>` and stored into the `<variable>`
- the next statement becomes current

(S) Channels (asynchronous)

```
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;}
```

The system above corresponds to the following Kripke structure:



(S) Channels (synchronous)

```
chan <channel> = [0] of {<type>};
```

Send statement:

```
<channel>!<expression>
```

The statement is active $\Leftrightarrow$ at least one of the current statements has one of the following forms (receive statement):

- `<channel>?<variable>`
- `<channel>?<another_expression>`, and values of the `<expression>` and the `<another_expression>` are equal

All mentioned receive statements are considered to be active as well

(S) Channels (synchronous)

```
chan <channel> = [0] of {<type>};
```

Send statement:

```
<channel>!<expression>
```

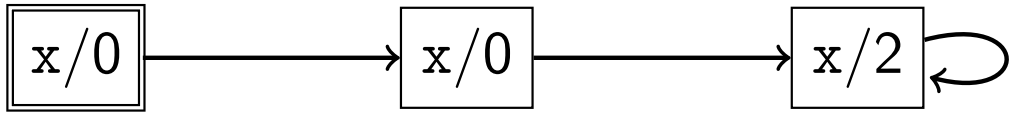
An execution of the statement:

- one of corresponding active receive statements is picked nondeterministically
- the next statements become current in both sending and receiving processes
- if "`<channel>?<variable>`" is picked, then the value of the `<expression>` is stored into the *variable*

(S) Channels (synchronous)

```
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;}
```

The system above corresponds to the following Kripke structure:



(S) Run statement

Processes can be added to the system in runtime with the following run statement:

`run <process_type>(<arguments>)`

The statement is always active

An execution of the statement:

- a new process of the given `<process_type>` is added to the state (with the first body statement as current)
- the next statement becomes current

Process type declaration has a number of `<parameters>` in general case

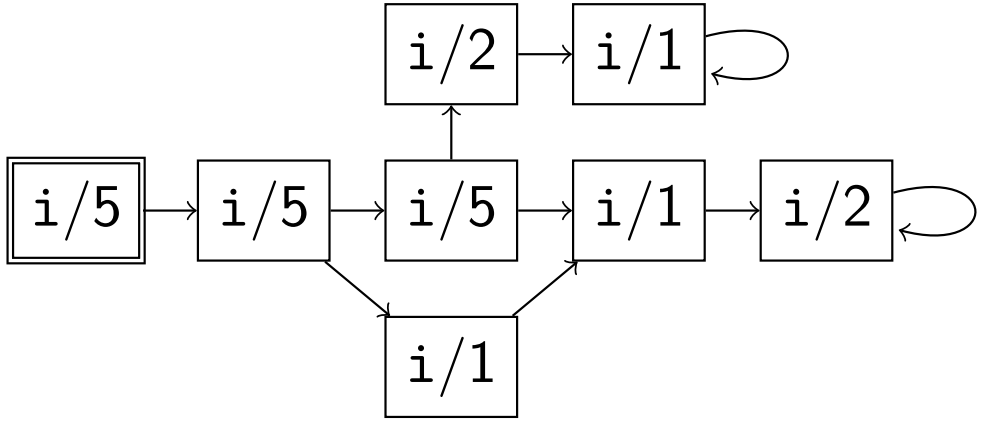
- `<parameters>` is a list of `<declaration>`s separated by ";"
- `<declaration> ::= <type> <list_of_names_separated_by_comma>`

`<Parameters>` and `<arguments>` are related in the same way as in *C/C++ for function calls with* ***passing by value***

(S) Run statement

```
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 }
```

The system above corresponds to the following Kripke structure:



(S) Atomic statement sequences

Sometimes several consequent statements of a system are required to be atomic, which means that other processes are not allowed to interfere with an execution of the sequence

For instance:

- if the process `P` in the last example is intended to *initialize* a system with two different `Q` processes, then the execution of `P` should be atomic (and not mixed with executions of `Q`-s)
- in semaphore-based critical section management, a semaphore test and the following value change should be an atomic sequence

(S) Atomic statement sequences

`atomic {<nonempty_sequence_of_statements>}`

To be declared atomic, a nonempty sequenced of statements should be wrapped as shown above

Execution restrictions for an atomic sequence:

- if
  1. the last executed statement is an element of an atomic sequence s of a process p and
  2. the next statement of p is **active** and belongs to s,

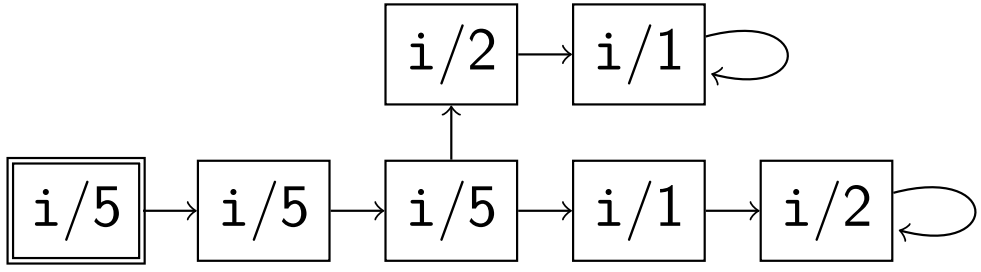
  then p becomes the only active process of the system
- otherwise, the system executes as usual

Be careful: `Spin` has several important undocumented `atomic`-related details, because (a) atomic loops are forbidden and (b) sometimes atomic sequences are "merged" into a single transition, and all intermediate states are lost

(S) Atomic statement sequences

```
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 }
```

The system above corresponds to the following Kripke model:



(S) Local variables

Local variables of a process can be declared at the beginning of its body, *just like local variables of C functions*

Local variables exist exactly when the coreesponding process exists, and initialized when the process is created

A local variable declaration is **not** a statement

If a system creates a process of type `P` **exactly once**, then its local variable `x` and label `L` can be accessed as follows: P:x, P@L

These expressions can be used in ltl formulae: "the variable `x` of `P`", "the current statement `P` is the one labeled with `L`"

(*if you want to write the same for many processes of a common type, then read manuals carefully*)

(S) Concluding example

```
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)}
```

What does the corresponding Kripke structure look like, and which formulae are satisfied?