Session 6

Promela: Process MetaLanguage

Promela

- "PROcess MEtaLAnguage" developed for
- SPIN Model checker (Simple Promela INterpreter) `spinroot.com`
- in 2002 was awarded the ACM System Software Award
- Promela is a specification language and
- SPIN a simulator and verifier
- A program $\bar{P}$ is a set of processes

  $$P_1, \ldots P_n$$

  that execute concurrently (asynchonously), with
  - shared variables
  - channels FIFO or synchronous
- to describe a process is used a language of guarded commands: statements, communication actions and atomic regions.
- guarded commands: a guard and an action
- see also [Hol03, BA08]

Statements Subset

proctype *name*(*list of formal parameters*) {stmt}

$$
\begin{array}{rcl}
\mathsf{stmt} & ::= & \mathtt{skip} \mid x := \mathsf{expr} \mid c?x \mid c!\mathsf{expr} \mid \\
 & & \mathsf{stmt}_1 \,;\, \mathsf{stmt}_2 \mid \mathtt{atomic}\{\mathsf{assignments}\} \mid \\
 & & \textbf{if} \quad :: g_1 \Rightarrow \mathsf{stmt}_1 \quad \ldots \quad :: g_n \Rightarrow \mathsf{stmt}_n \quad \textbf{fi} \quad \mid \\
 & & \textbf{do} \quad :: g_1 \Rightarrow \mathsf{stmt}_1 \quad \ldots \quad :: g_n \Rightarrow \mathsf{stmt}_n \quad \textbf{do}
\end{array}
$$

- global variables are declared outside processes
- local variables are declared inside processes
- formal parameters are local to processes and can be instantiated by the caller

- each process state includes the program counter and the values of local variables
- g_i are conditions over the variables
- *skip* ends in a step and does not changes the state

Statements Subset

proctype *name*(*list of formal parameters*) {stmt}

$$
\begin{array}{rcl}
\mathsf{stmt} & ::= & \mathtt{skip} \mid x := \mathsf{expr} \mid c?x \mid c!\mathsf{expr} \mid \\
 & & \mathsf{stmt}_1 \,;\, \mathsf{stmt}_2 \mid \mathtt{atomic}\{\mathsf{assignments}\} \mid \\
 & & \textbf{if} \quad :: g_1 \Rightarrow \mathsf{stmt}_1 \quad \ldots \quad :: g_n \Rightarrow \mathsf{stmt}_n \quad \textbf{fi} \quad \mid \\
 & & \textbf{do} \quad :: g_1 \Rightarrow \mathsf{stmt}_1 \quad \ldots \quad :: g_n \Rightarrow \mathsf{stmt}_n \quad \textbf{do}
\end{array}
$$

- ; or $\Rightarrow$ sequential execution
- any statement in any state can be *executable* or *blocked*
- an expression if evaluates to false *blocks*
- `atomic` is a statement that cannot be interleaved
- see the operational semantics in [BKL08] (Cap. 2.2.5).
- other basic statments: **printf** and **assert**.

Data Types

Type	Size(bits)	Example
bit, bool	1	bit turn=1;
byte	8	byte counter;
short	16	
int	32	
unsigned	≤ 32	
pid		process id, *_pid*
mtype		list of symbolic values
chan		channel
Arrays		byte colour[5];
typedef	Records	typedef Point byte x; byte y;
		Point p;
		p.x =...

Selection if-fi

$$\textbf{if} \quad :: g_1 \Rightarrow \mathsf{stmt}_1 \quad \ldots \quad :: g_n \Rightarrow \mathsf{stmt}_n \quad \textbf{fi}$$

- nondeterministic choice of $stmt_i$ for which the guard g_i holds in the current state.
- execution is atomic
- if none of the guards $g_1, \ldots, g_n$ holds in the current state the process *blocks*.
- in this case the execution of other process can unblock this process
- imperative **if-then-else** corresponds to
- any statement can be a guard

$$\textbf{if} \ :: g \Rightarrow \mathsf{stmt}_1 \ :: \neg g \Rightarrow \mathsf{stmt}_2 \ \textbf{fi},$$

- One can use **::else** that avoids blocking if all other guards are false.

```
active proctype P() {
int a  = 7, b = 5, max;
if
:: a >= b -> max = a
:: b >= a  -> max = b
fi;
printf("max %d\n", max);
assert (a>=b -> max ==a : max ==b);
}
```

A model is a LTS where transitions correspond to action of basic commands: assingnmets, channel receive and send, print and assert. (See exampel in ispin)

Loop do-od

$$\textbf{do} \quad :: g_1 \Rightarrow \mathsf{stmt}_1 \quad \ldots \quad :: g_n \Rightarrow \mathsf{stmt}_n \quad \textbf{do}$$

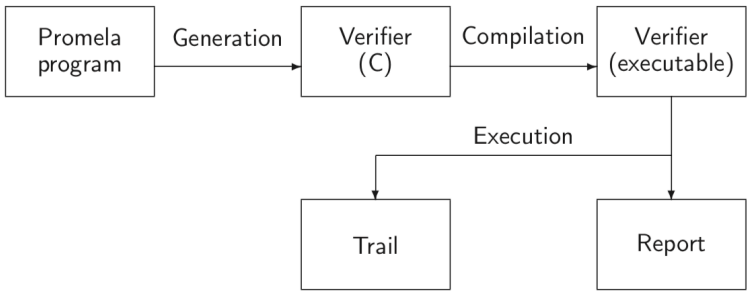
- Repeated execution of the nondeterministic choice of guarded commands which guards are true
- if all guards are false the process blocks
- the loop ends with the statement **break**
- or if there is a statement **::else**
- one can also use the goto statement **goto** *label*, where *label:* should label the instruction to be executed next.

Verification of sequential programs

- Assertions allow to verify sequential programs
- **assert**(*cond*)
- a model checker verifies all possible nondeterministic executions
- and if *cond* does not hold for one execution a counter-example is given.

Verification with SPIN

- For efficiency reasons SPIN generates a verifier in C and after compilation allows to execute the verification
- `spin -a max.pml; gcc -o pan pan.c; ./pan`
- Error analysis: `spin -t max.pml`
- and options `-gplv` give more details



Examples

- max.pml
- mdc.pml
- vending.pml

Verification of Concurrent Programs

When there is more than one process

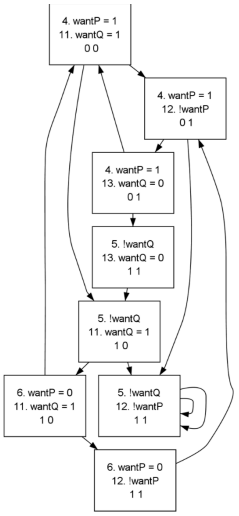
- Promela:
  - Execution of several copies of P:**active** [n] P()

  - **_pid** indicates the number of the current process
  - **_nr_pr** indicates the number of active processes
  - instead of **active**
  - first process: **init** { ... }
  - use **run** to execute a specific process inside a process Ex: **run**(P(1,5))
  - **run** returns the pid of the process; it is not a statement
- SPIN:
  - Random execution (of a certain number of steps):`spin pq.pml`
  - Interactive execution: `spin -i pq.pml`
  - Construction of a verifier: `spin -a pq.pml`
  - ... see other options
  - npr.pml , nprc.pml

Mutual Excusion

- naive version with a counter me.pml
- with sincronization busy-waiting: me2.pml
- with deadlock: me1.pml
- with a semaphore: mes.pml
- implement Peterson algorithm (Labs)

LTS for me1.pml



LTL specification

We have the following notation

$$\begin{array}{lll} \mathrm{F} & \rightarrow & <> \\ \mathrm{G} & \rightarrow & [] \\ \mathrm{X} & \rightarrow & X \\ \mathrm{U} & \rightarrow & U \\ \wedge & \rightarrow & \&\& \\ \vee & \rightarrow & || \\ \neg & \rightarrow & ! \\ \rightarrow & \rightarrow & -> \end{array}$$

For instance

GF`p`

corresponds to [] <> `p`.

LTL usage in SPIN

- An LTL formula can be specified with the model with **ltl**
- or in the command line but it this case it must be *negated*
- If the formula is GFp then
- `spin -a -f '![]<>p' name.pml` or in general

```
% spin -a -f <formula_negated> <name>.pml
% gcc -o pan pan.c
%./pan -a -f
```

- or in a file (also negated)

```
% spin -a -F forltl.prp > forltl.pml
% spin -a -N forltl.pml  <nome>.pml
% gcc -o pan pan.c
%./pan -a -f
```

  In this case, the formula was transformed in a Promela program `forltl.pml` with the statement `never` (never claim).

SPIN Correctness Claims – Safety

- Assertions (`assert`)
- Absence of deadlock (invalid end states):In every state of every computation, if no statements are executable, the location counter of each process must be at the end of the process or at a statement labeled end (end-state labels) Option -E disables reporting of end-state error
- absence of unreachable code

SPIN Correctness Claims – Liveness

- Progress-state labels: check that every potentially infinite execution cycle permitted by a model passes through at least one of the progress labels in that model (`progress`)
- accept-state labels : when looking for acceptance cycles (i.e infinite executions that pass through a state labeled `accept` (pan -a).
- Never claims: negation of ltl formulas transformed in a Promela program
- Just use LTL formulas

SPIN Correctness Claims

- Weak fairness (stricter version) : if a process reaches a point where it has an executable statement and executability of that statement never changes, it will eventually proceed by executing the statement
- Strong fairness (general version) : if the process reaches point where it has a statement that becomes executable infinitely often it will eventually proceed by executing the statement

Spin trail-trace with a detected error

```
1:0:14
2:0:15
3:0:16
4:2:7
5:2:8
6:1:0
7:2:9
```

In each step $S : P : T$ where

S step number on the execution trail

P process identifier

T transition identifier of the current step

Use `spin -t name.pml` Options -p, -l -v give more information about the trail

References

[BA08] Mordechai Ben-Ari. *Principles of the Spin Model Checker*. Springer, 2008.

[BKL08] Christel Baier, Joost-Pieter Katoen, and Kim Guldstrand Larsen. *Principles of Model Checking*. MIT Press, 2008.

[Hol03] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley, 2003.

Loop **do-od**

$$\textbf{do} \quad :: g_1 \Rightarrow \mathsf{stmt}_1 \quad \ldots \quad :: g_n \Rightarrow \mathsf{stmt}_n \quad \textbf{do}$$

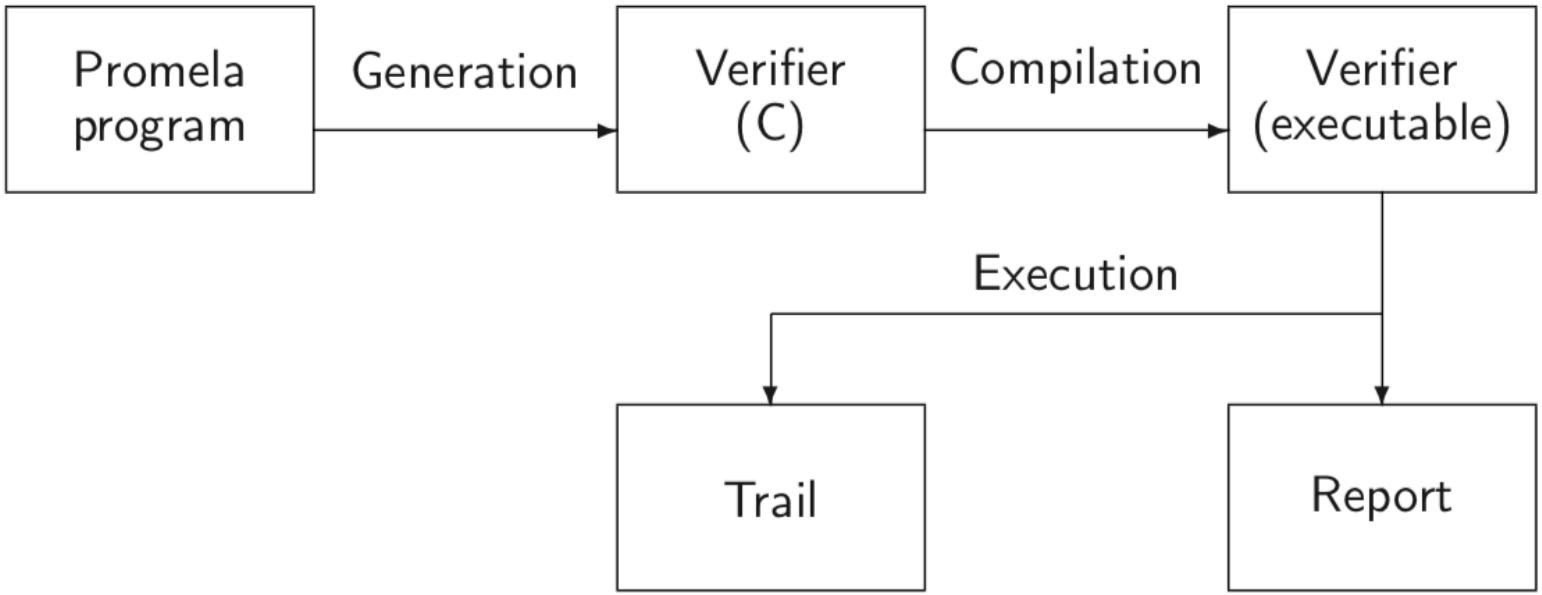
- Repeated execution of the nondeterministic choice of guarded commands which guards are true
- if all guards are false the process blocks
- the loop ends with the statement **break**
- or if there is a statement **::else**
- one can also use the goto statement **goto** *label*, where *label:* should label the instruction to be executed next.

Verification of sequential programs

- Assertions allow to verify sequential programs
- **assert**(*cond*)
- a model checker verifies all possible nondeterministic executions
- and if *cond* does not hold for one execution a counter-example is given.

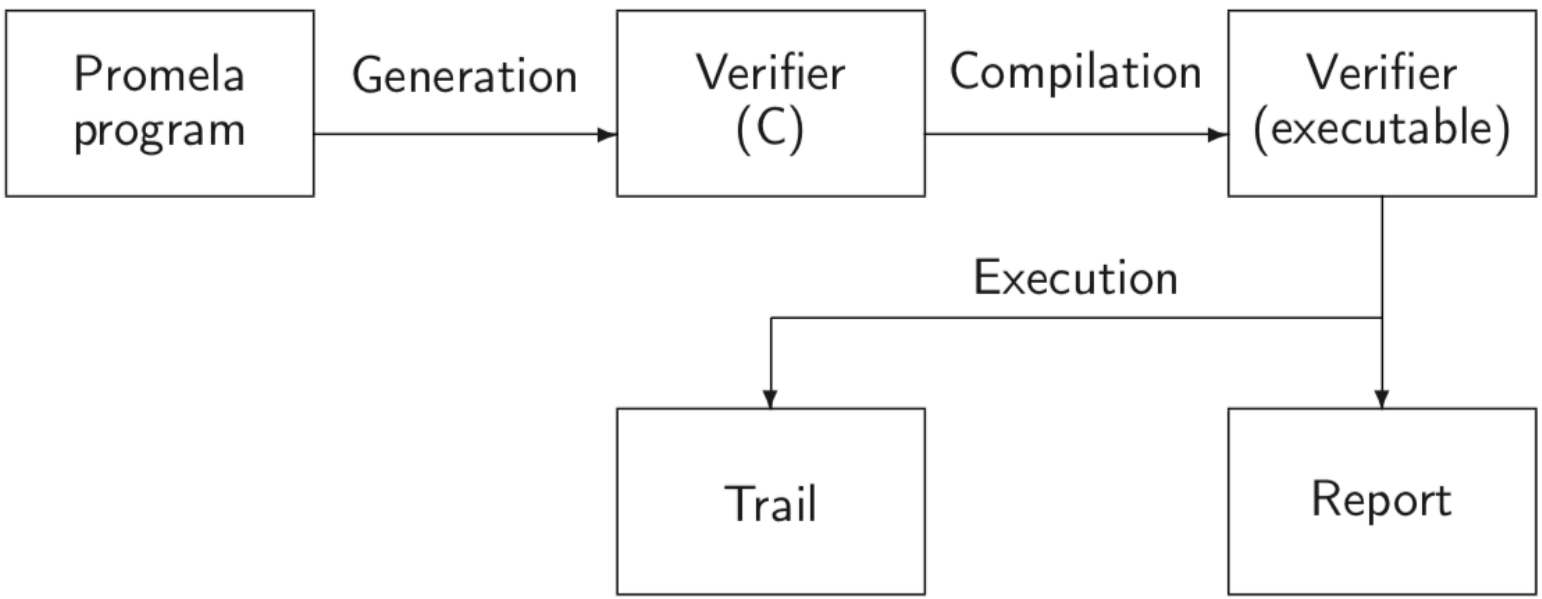
Verification with SPIN

- For efficiency reasons SPIN generates a verifier in C and after compilation allows to execute the verification



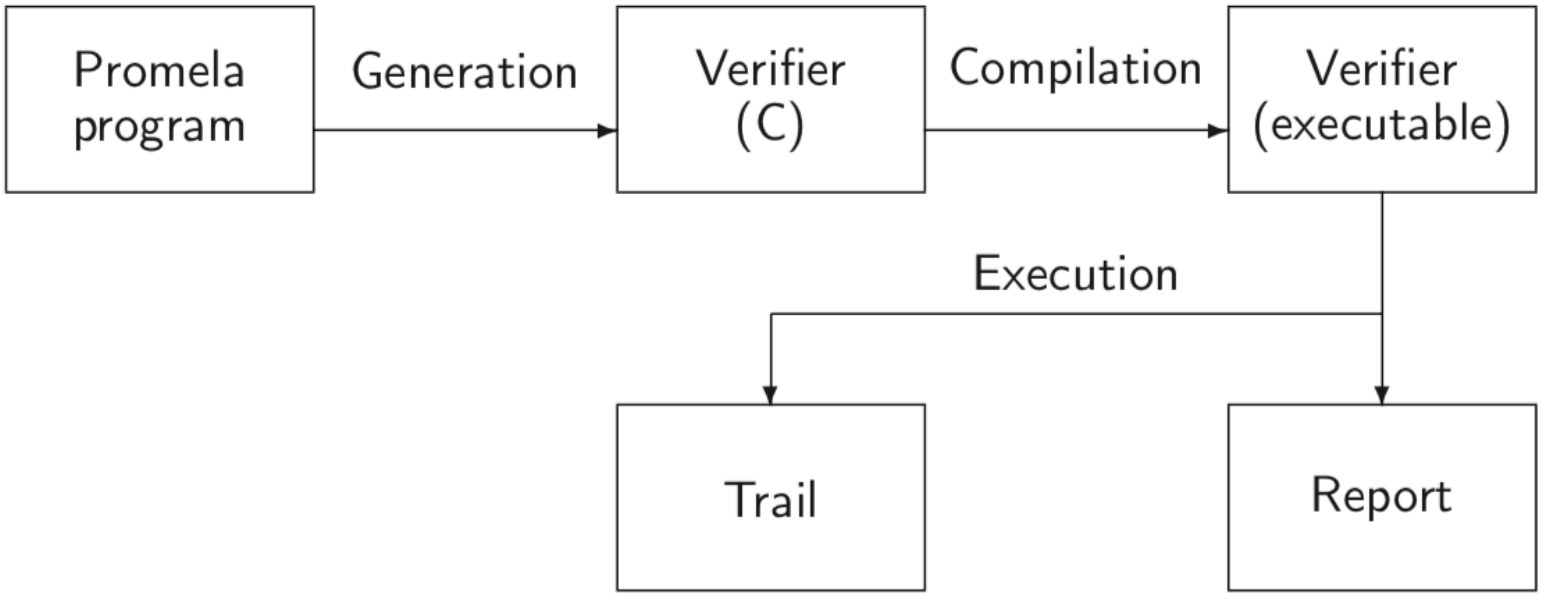
Verification with SPIN

- For efficiency reasons SPIN generates a verifier in C and after compilation allows to execute the verification
- `spin -a max.pml; gcc -o pan pan.c; ./pan`



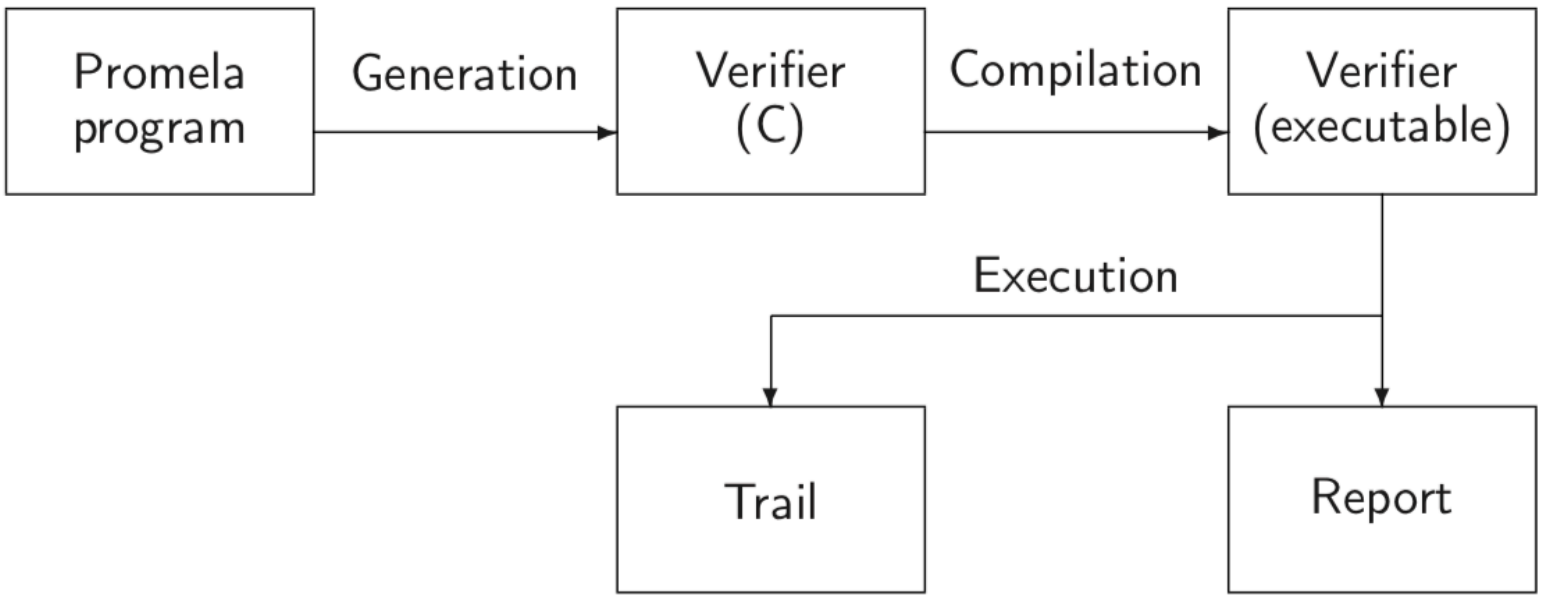
Verification with SPIN

- For efficiency reasons SPIN generates a verifier in C and after compilation allows to execute the verification
- `spin -a max.pml; gcc -o pan pan.c; ./pan`
- Error analysis: `spin -t max.pml`



Verification with SPIN

- For efficiency reasons SPIN generates a verifier in C and after compilation allows to execute the verification
- `spin -a max.pml; gcc -o pan pan.c; ./pan`
- Error analysis: `spin -t max.pml`
- and options `-gplv` give more details



Examples

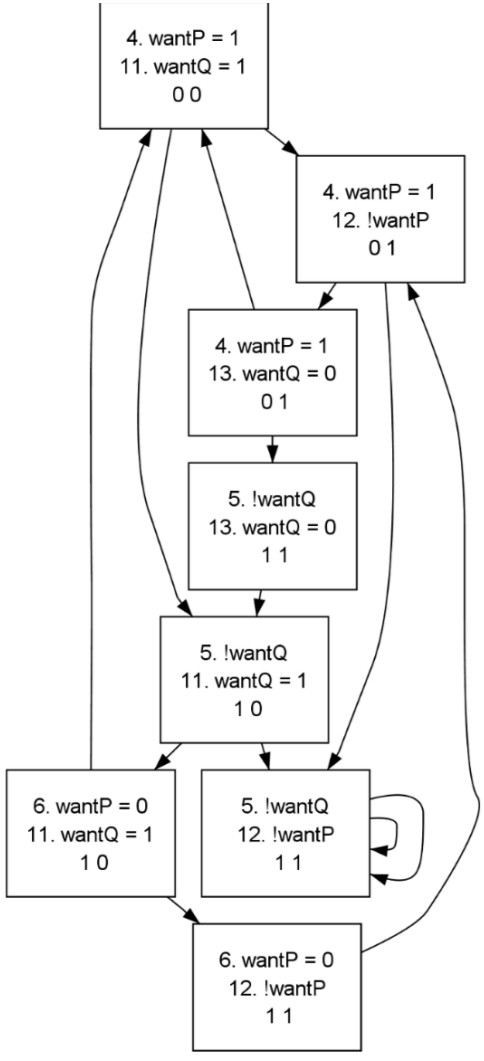
- max.pml
- mdc.pml
- vending.pml

When there is more than one process

- Promela:
  - Execution of several copies of P:**active** [n] P()
  - **_pid** indicates the number of the current process
  - **_nr_pr** indicates the number of active processes
  - instead of **active**
  - first process: **init** { . . . }
  - use **run** to execute a specific process inside a process Ex: **run**(P(1,5))
  - **run** returns the pid of the process; it is not a statement
- SPIN:
  - Random execution (of a certain number of steps):`spin pq.pml`
  - Interactive execution: `spin -i pq.pml`
  - Construction of a verifier: `spin -a pq.pml`
  - ... see other options
  - npr.pml , nprc.pml

- naive version with a counter me.pml
- with sincronization busy-waiting: me2.pml
- with deadlock: me1.pml
- with a semaphore: mes.pml
- implement Peterson algorithm (Labs)

LTS for me1.pml



LTL specification

We have the following notation

F	$\rightarrow$	$<>$
G	$\rightarrow$	[]
X	$\rightarrow$	X
U	$\rightarrow$	U
$\wedge$	$\rightarrow$	&&
$\vee$	$\rightarrow$	$\|\|$
$\neg$	$\rightarrow$	!
$\rightarrow$	$\rightarrow$	$->$

For instance

GFp

corresponds to [] <> p.

LTL usage in SPIN

- An LTL formula can be specified with the model with **ltl**
- or in the command line but it this case it must be negated
- If the formula is GFp then
- `spin -a -f '![]<>p' name.pml` or in general

```
% spin -a -f <formula_negated> <name>.pml
% gcc -o pan pan.c
%./pan -a -f
```

- or in a file (also negated)

```
% spin -a -F forltl.prp > forltl.pml
% spin -a -N forltl.pml  <nome>.pml
% gcc -o pan pan.c
%./pan -a -f
```

In this case, the formula was transformed in a Promela program `forltl.pml` with the statement `never` (never claim).

SPIN Correctness Claims – Safety

- Assertions (`assert`)
- Absence of deadlock (invalid end states):In every state of every computation, if no statements are executable, the location counter of each process must be at the end of the process or at a statement labeled end (end-state labels) Option -E disables reporting of end-state error
- absence of unreachable code

SPIN Correctness Claims – Liveness

- Progress-state labels: check that every potentially infinite execution cycle permitted by a model passes through at least one of the progress labels in that model (`progress`)
- accept-state labels : when looking for acceptance cycles (i.e infinite executions that pass through a state labeled `accept` (pan -a).
- Never claims: negation of ltl formulas transformed in a Promela program
- Just use LTL formulas

SPIN Correctness Claims

- Weak fairness (stricter version) : if a process reaches a point where it has an executable statement and executability of that statement never changes, it will eventually proceed by executing the statement
- Strong fairness (general version) : if the process reaches point where it has a statement that becomes executable infinitely often it will eventually proceed by executing the statement

Spin trail-trace with a detected error

```
1:0:14
2:0:15
3:0:16
4:2:7
5:2:8
6:1:0
7:2:9
```

In each step $S : P : T$ where

- S step number on the execution trail
- P process identifier
- T transition identifier of the current step

Use `spin -t name.pml` Options -p, -l -v give more information about the trail