Tutorial on SPIN and PROMELA

(as part of the lecture "model checking", WS 2012/13)

Dr.-Ing. Sascha Klüppelholz

Institut für Theoretische Informatik
Lehrstuhl für Algebraische und logische Grundlagen der Informatik

November 15, 2012

Outline

Outline

1. Overview
2. Repetition
   - Channel systems
   - Guarded command languages
3. ProMeLa
4. SPIN

SPIN overview

- open-source software tool (http://spinroot.com) (freely available since 1991)

SPIN overview

- open-source software tool (http://spinroot.com) (freely available since 1991)
- one of the most prominent tools for formal verification of distributed software systems

SPIN overview

- open-source software tool (http://spinroot.com) (freely available since 1991)
- one of the most prominent tools for formal verification of distributed software systems
- developed by Gerald Holzmann at Bell Labs (beginning in 1980)

SPIN overview

- open-source software tool (http://spinroot.com) (freely available since 1991)
- one of the most prominent tools for formal verification of distributed software systems
- developed by Gerald Holzmann at Bell Labs (beginning in 1980)
- awarded the prestigious System Software Award 2001 by the ACM

SPIN overview

- open-source software tool (http://spinroot.com) (freely available since 1991)
- one of the most prominent tools for formal verification of distributed software systems
- developed by Gerald Holzmann at Bell Labs (beginning in 1980)
- awarded the prestigious System Software Award 2001 by the ACM
- Primer and Reference Manual [Hol03]

SPIN main features

- modeling language of SPIN is called ProMeLa (**Pro**cess **Me**ta **La**nguage) $\rightarrow$ the name SPIN stands for **S**imple **P**roMeLa **In**terpreter

SPIN main features

- modeling language of SPIN is called ProMeLa (**Pro**cess **Me**ta **La**nguage) $\rightarrow$ the name SPIN stands for **S**imple **P**roMeLa **In**terpreter
- main features of SPIN: 1) on-the-fly verifier for safety and liveness properties and 2) on-the-fly LTL model checking

SPIN main features

- modeling language of SPIN is called ProMeLa (**Pro**cess **Me**ta **La**nguage) $\rightarrow$ the name SPIN stands for **S**imple **P**roMeLa **In**terpreter
- main features of SPIN: 1) on-the-fly verifier for safety and liveness properties and 2) on-the-fly LTL model checking
- properties can be specified as 1) invariants (using assertions), 2) LTL formula, 3) Büchi automaton, or 4) never claims (omega regular)

ProMeLa main characteristics

- modeling language for channel systems with
  1) a (finite) number of processes,
  2) synchronous and asynchronous channel-based communication, and 3) shared variables

ProMeLa main characteristics

- modeling language for channel systems with
  1) a (finite) number of processes,
  2) synchronous and asynchronous channel-based communication, and 3) shared variables
- guarded-command language [Dij76] (plus embedded C-code)

The model checker SPIN

ProMeLa main characteristics

- modeling language for channel systems with
  1) a (finite) number of processes,
  2) synchronous and asynchronous channel-based communication, and 3) shared variables
- guarded-command language [Dij76] (plus embedded C-code)
- featuring nondeterminism (language features and interleaving processes)

ProMeLa main characteristics

- modeling language for channel systems with 1) a (finite) number of processes, 2) synchronous and asynchronous channel-based communication, and 3) shared variables
- guarded-command language [Dij76] (plus embedded C-code)
- featuring nondeterminism (language features and interleaving processes)
- semantics based on program graphs (and hence transition systems)

Outline

1. Overview
2. Repetition
   - Channel systems
   - Guarded command languages
3. ProMeLa
4. SPIN

Channel systems

Channel systems

representation of data-dependent parallel systems with

- communication over shared variables
- synchronous message passing
- asynchronous message passing

Channel systems

representation of data-dependent parallel systems with

- communication over shared variables
- synchronous message passing
- asynchronous message passing

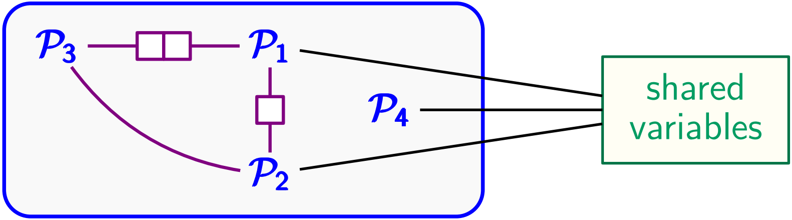
communication over channels

Channel systems

representation of data-dependent parallel systems with

- communication over shared variables
- synchronous message passing
- asynchronous message passing

communication over channels

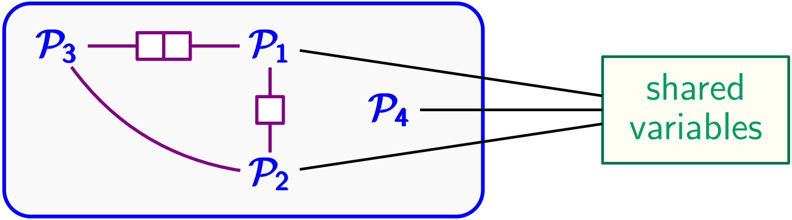


Channel systems

representation of data-dependent parallel systems with

- communication over shared variables
- synchronous message passing
- asynchronous message passing

communication over channels

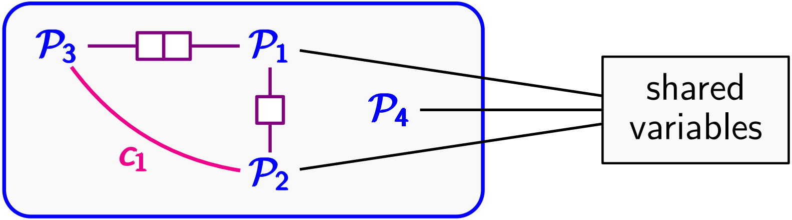


channel types: synchronous or FIFO

Channel systems

representation of data-dependent parallel systems with

- communication over shared variables
- synchronous message passing $\longleftarrow$ capacity **0**
- asynchronous message passing



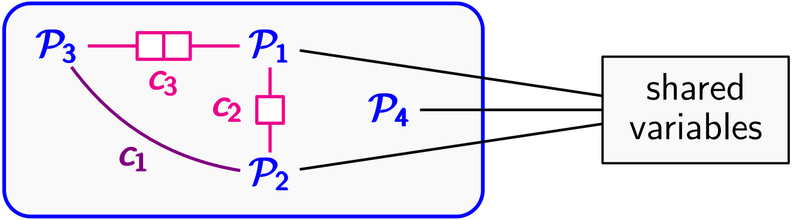
channel types: synchronous or FIFO

$\uparrow$

no buffer (read/write simultaneously)

representation of data-dependent parallel systems with

- communication over shared variables
- synchronous message passing $\longleftarrow$ capacity **0**
- asynchronous message passing $\longleftarrow$ capacity $\geqslant$ **1**



channel types: synchronous or FIFO

capacity = number of buffer cells

Channel systems

representation of data-dependent parallel systems with

- communication over shared variables
- synchronous message passing $\longleftarrow$ capacity **0**
- asynchronous message passing $\longleftarrow$ capacity $\geqslant \mathbf{1}$

formalization through program graphs for $\mathcal{P}_1, \ldots, \mathcal{P}_n$

Channel systems

representation of data-dependent parallel systems with

- communication over shared variables
- synchronous message passing $\longleftarrow$ capacity $\mathbf{0}$
- asynchronous message passing $\longleftarrow$ capacity $\geqslant \mathbf{1}$

formalization through program graphs for $\mathcal{P}_1, \ldots, \mathcal{P}_n$

* with conditional transitions $\ell_i \xhookrightarrow{g:\alpha} \ell_i'$ (as before)

Channel systems

representation of data-dependent parallel systems with

- communication over shared variables
- synchronous message passing $\longleftarrow$ capacity 0
- asynchronous message passing $\longleftarrow$ capacity $\geqslant 1$

formalization through program graphs for $\mathcal{P}_1, \ldots, \mathcal{P}_n$

- with conditional transitions $\ell_i \xhookrightarrow{g:\alpha} \ell_i'$ (as before)
- and communication actions

$\ell_i \xhookrightarrow{c!v} \ell_i'$ sending value v via channel c

$\ell_i \xhookrightarrow{c?x} \ell_i'$ receiving a value for variable x via c

Interleaving for program graphs

... modeling parallel systems with processes communicating via shared variables

Interleaving for program graphs

program graph $\mathcal{P}_1$
$(Loc_1, \ldots, \hookrightarrow_1, \ldots)$

program graph $\mathcal{P}_2$
$(Loc_2, \ldots, \hookrightarrow_2, \ldots)$

Interleaving for program graphs

program graph $\mathcal{P}_1$
$(Loc_1, \ldots, \hookrightarrow_1, \ldots)$

program graph $\mathcal{P}_2$
$(Loc_2, \ldots, \hookrightarrow_2, \ldots)$

interleaving operator

$$\mathcal{P}_1 \,|||\, \mathcal{P}_2 \;=\; (Loc_1 \times Loc_2, \ldots, \hookrightarrow, \ldots)$$

Interleaving for program graphs

program graph $\mathcal{P}_1$
$(Loc_1, \ldots, \hookrightarrow_1, \ldots)$

program graph $\mathcal{P}_2$
$(Loc_2, \ldots, \hookrightarrow_2, \ldots)$

interleaving operator

$$\mathcal{P}_1 \,|||\, \mathcal{P}_2 \;=\; (Loc_1 \times Loc_2, \ldots, \hookrightarrow, \ldots)$$

$$\frac{\ell_1 \overset{g:\alpha}{\hookrightarrow}_1 \ell_1'}{\langle \ell_1, \ell_2 \rangle \overset{g:\alpha}{\hookrightarrow} \langle \ell_1', \ell_2 \rangle}$$

Interleaving for program graphs

program graph $\mathcal{P}_1$
$(Loc_1, \ldots, \hookrightarrow_1, \ldots)$

program graph $\mathcal{P}_2$
$(Loc_2, \ldots, \hookrightarrow_2, \ldots)$

interleaving operator

$$\mathcal{P}_1 \,|||\, \mathcal{P}_2 \;=\; (Loc_1 \times Loc_2, \ldots, \hookrightarrow, \ldots)$$

$$\frac{\ell_1 \xhookrightarrow{g:\alpha}_1 \ell_1'}{\langle \ell_1, \ell_2 \rangle \xhookrightarrow{g:\alpha} \langle \ell_1', \ell_2 \rangle} \qquad \frac{\ell_2 \xhookrightarrow{g:\alpha}_2 \ell_2'}{\langle \ell_1, \ell_2 \rangle \xhookrightarrow{g:\alpha} \langle \ell_1, \ell_2' \rangle}$$

Example: interleaving for PG

Example: interleaving for PG

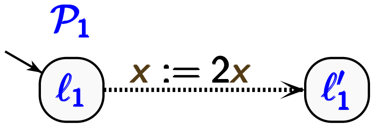
$\mathcal{P}_1$

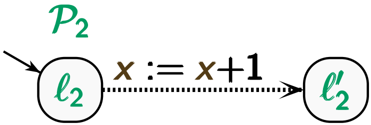
$$\to \ell_1 \xrightarrow{x := 2x} \ell_1'$$

$\mathcal{P}_2$

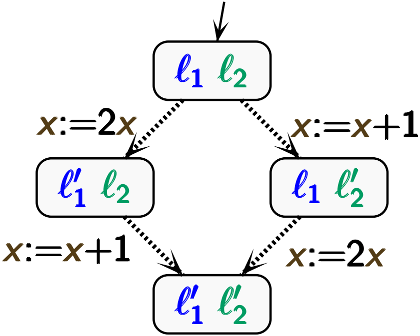
$$\to \ell_2 \xrightarrow{x := x+1} \ell_2'$$

Example: interleaving for PG

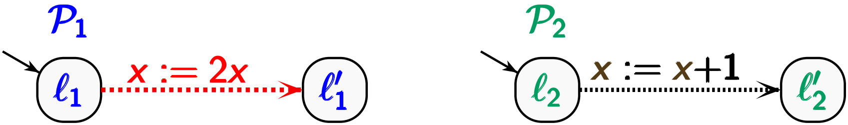




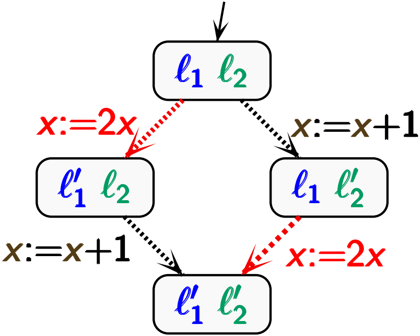
PG $\mathcal{P}_1 \,|||\, \mathcal{P}_2$



Example: interleaving for PG

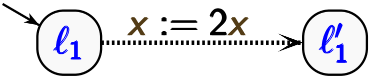


PG $\mathcal{P}_1 \,|||\, \mathcal{P}_2$

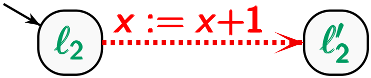


Example: interleaving for PG

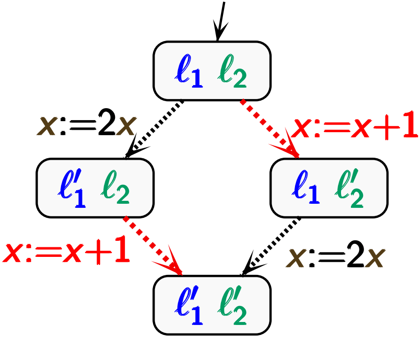
$\mathcal{P}_1$



$\mathcal{P}_2$

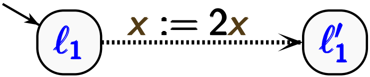


PG $\mathcal{P}_1 \,|||\, \mathcal{P}_2$

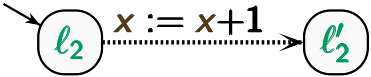


Example: interleaving for PG

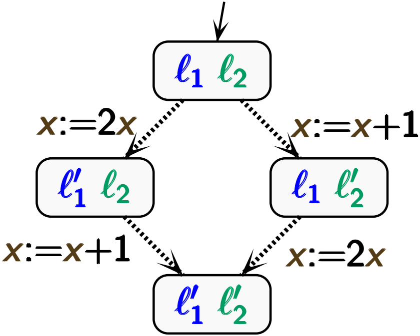
$\mathcal{P}_1$



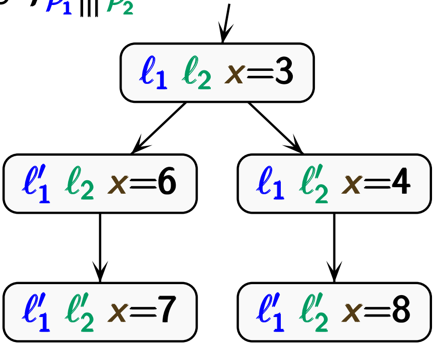
$\mathcal{P}_2$



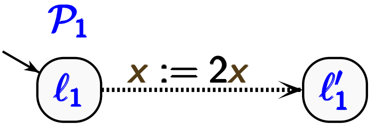
PG $\mathcal{P}_1 \,|||\, \mathcal{P}_2$

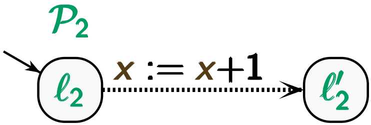


TS $\mathcal{T}_{\mathcal{P}_1 \,|||\, \mathcal{P}_2}$

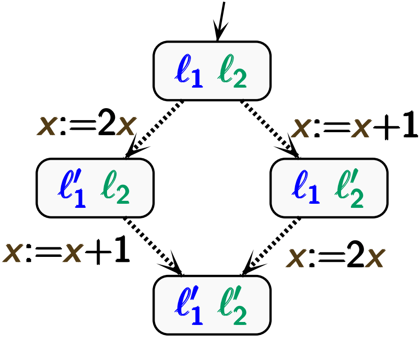


Example: interleaving for PG

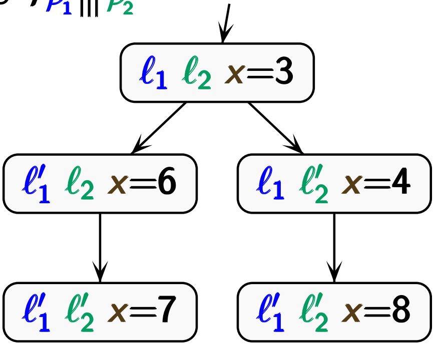




PG $\mathcal{P}_1 \,|||\, \mathcal{P}_2$



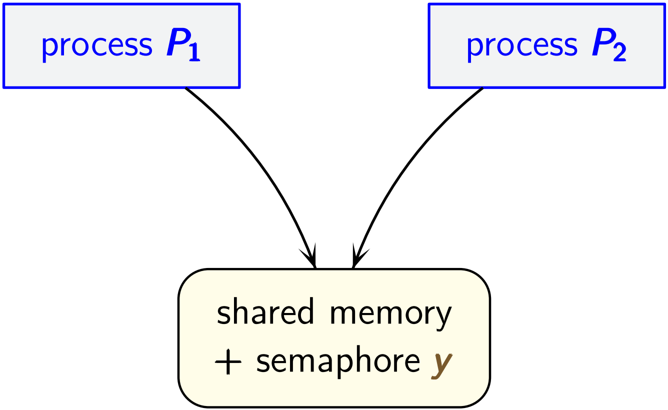
TS $\mathcal{T}_{\mathcal{P}_1 \,|||\, \mathcal{P}_2}$



note: $\mathcal{T}_{\mathcal{P}_1} \,|||\, \mathcal{T}_{\mathcal{P}_2} \neq \mathcal{T}_{\mathcal{P}_1 \,|||\, \mathcal{P}_2}$

Mutual exclusion with semaphore

Mutual exclusion with semaphore



Mutual exclusion with semaphore

protocol for process P_i

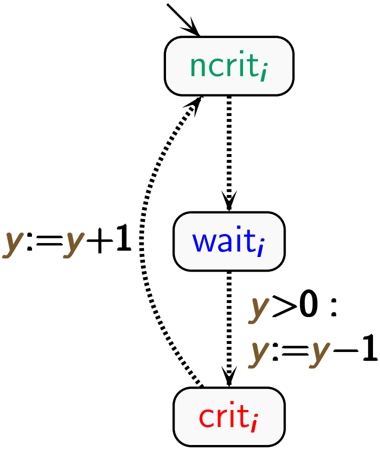
```
LOOP FOREVER
    noncritical actions;
    AWAIT  y > 0 DO
           y:=y−1
    OD
    critical actions;
    y:=y+1
    END LOOP
```

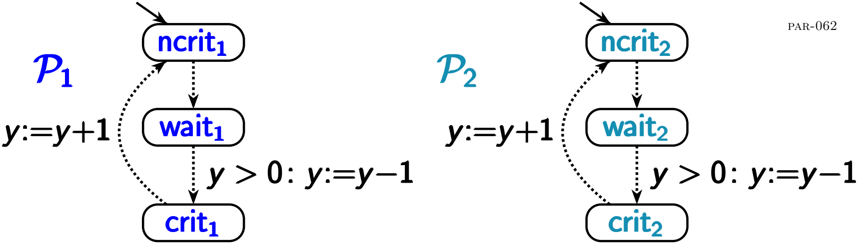
Mutual exclusion with semaphore

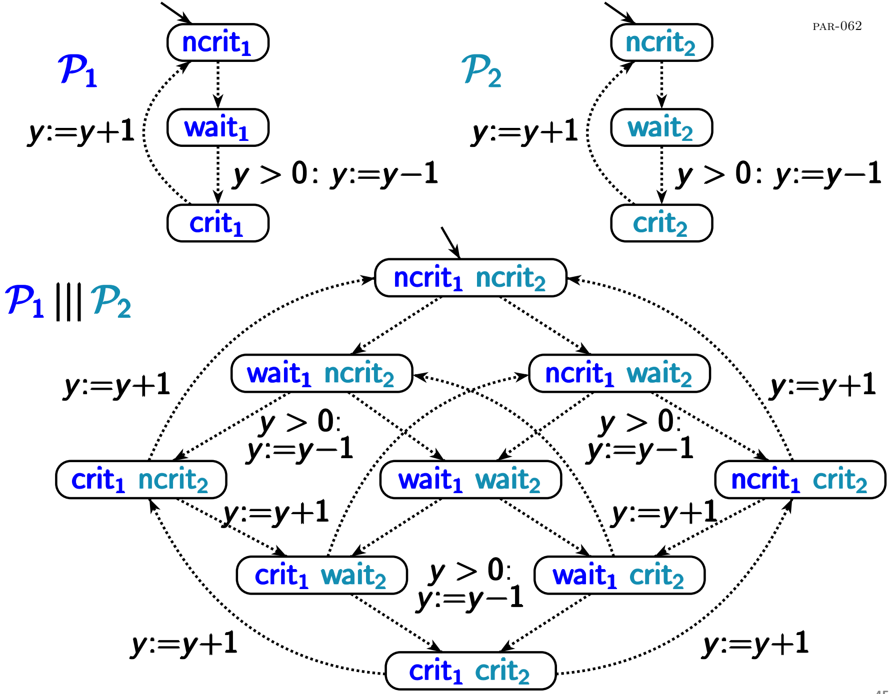
protocol for process P_i

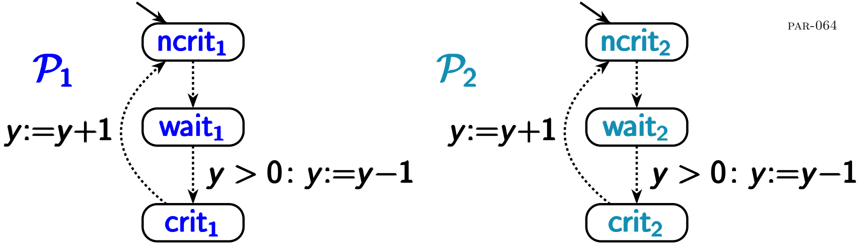
```
LOOP FOREVER
    noncritical actions;
    AWAIT  y > 0 DO
              y:=y−1
    OD
    critical actions;
    y:=y+1
    END LOOP
```

program graph $\mathcal{P}_i$

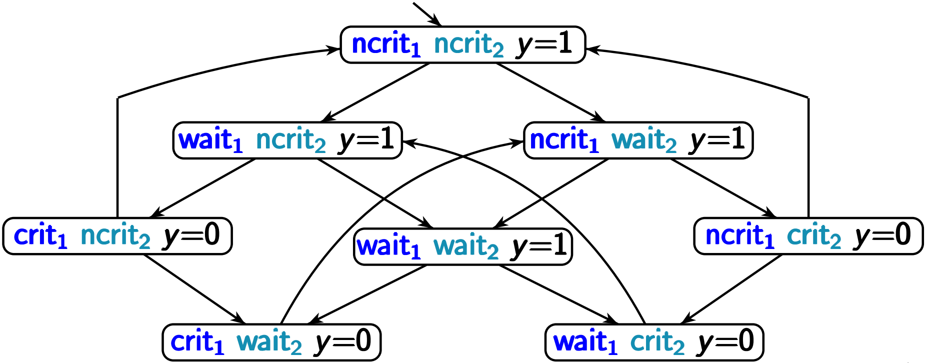


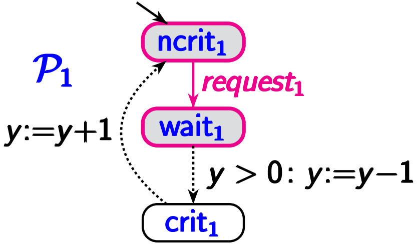

$\mathcal{P}_1$
ncrit$_1$
wait$_1$
crit$_1$
$y{:=}y{+}1$
$y > 0$: $y{:=}y{-}1$
$\mathcal{P}_2$
ncrit$_2$
wait$_2$
crit$_2$
$y{:=}y{+}1$
$y > 0$: $y{:=}y{-}1$

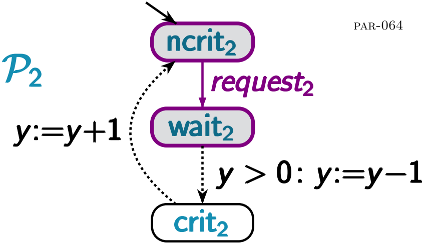

$\mathcal{P}_1$
ncrit$_1$
wait$_1$
crit$_1$
y:=y+1
$y > 0$: y:=$y-1$
$\mathcal{P}_2$
ncrit$_2$
wait$_2$
crit$_2$
y:=y+1
$y > 0$: y:=$y-1$
$\mathcal{P}_1 \,|||\, \mathcal{P}_2$
ncrit$_1$ ncrit$_2$
wait$_1$ ncrit$_2$
ncrit$_1$ wait$_2$
crit$_1$ ncrit$_2$
wait$_1$ wait$_2$
ncrit$_1$ crit$_2$
crit$_1$ wait$_2$
wait$_1$ crit$_2$
crit$_1$ crit$_2$
y:=y+1
$y > 0$: y:=$y-1$
y:=y+1
$y > 0$: y:=$y-1$
y:=y+1
y:=y+1
$y > 0$: y:=$y-1$
y:=y+1
y:=y+1



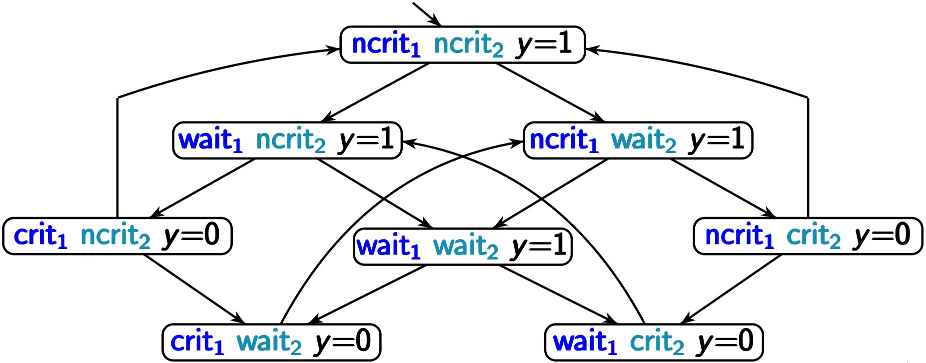
reachable fragment of the transition system $\mathcal{T}_{\mathcal{P}_1 ||| \mathcal{P}_2}$

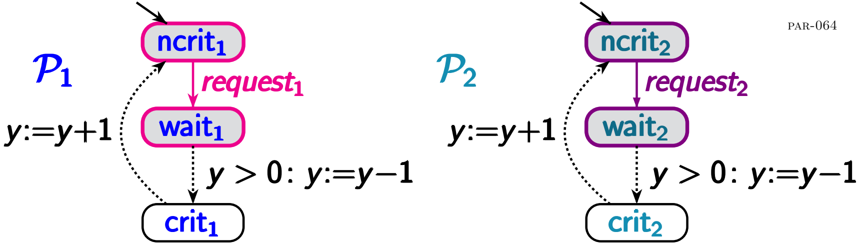




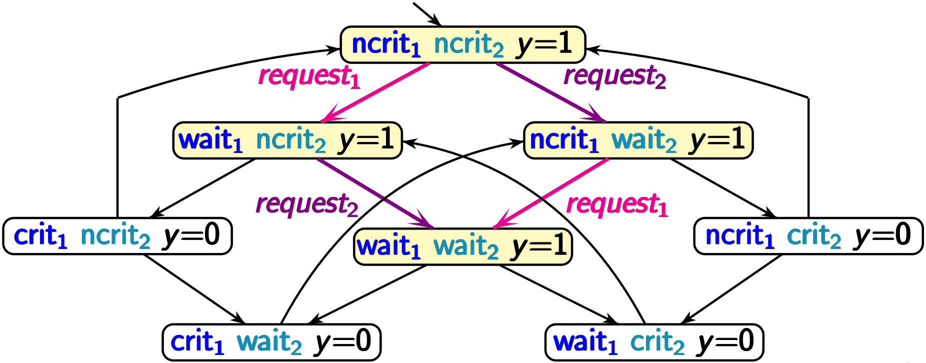


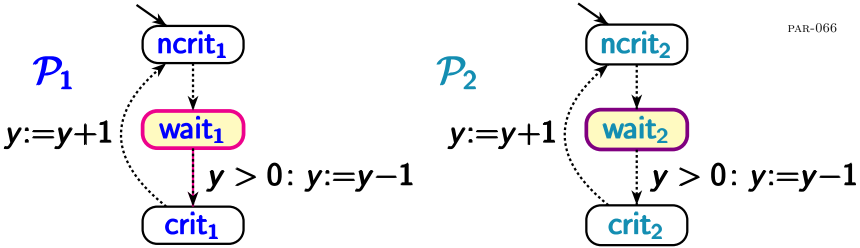
reachable fragment of the transition system $\mathcal{T}_{\mathcal{P}_1 \,|||\, \mathcal{P}_2}$



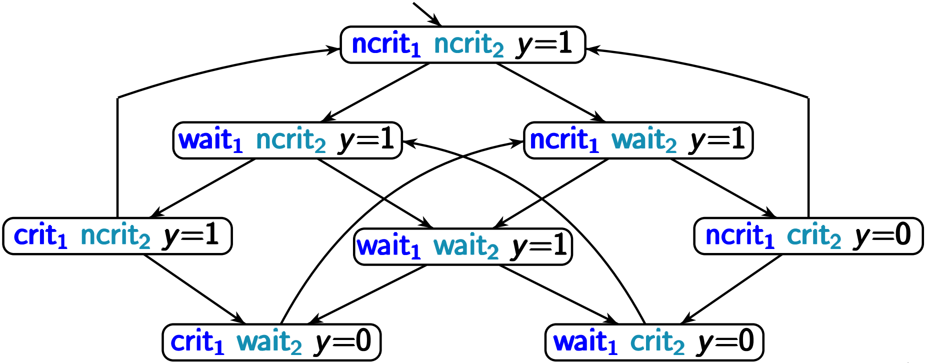


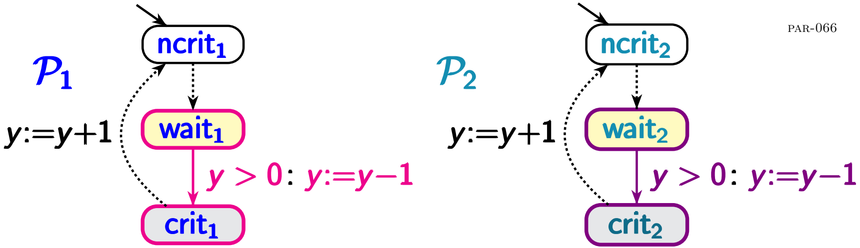
interleaving of the independent request actions



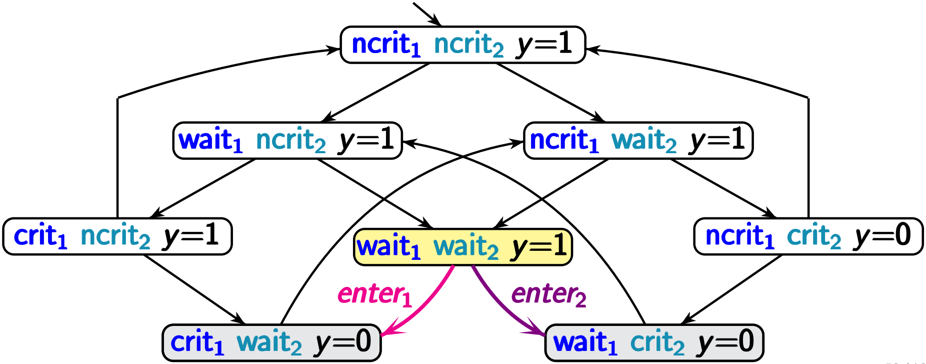


competition between the waiting processes



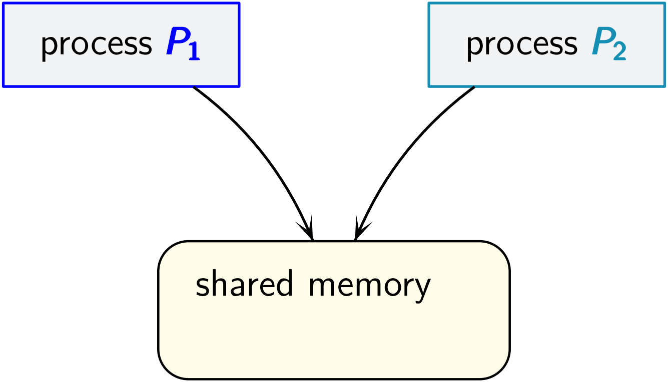


competition between the waiting processes

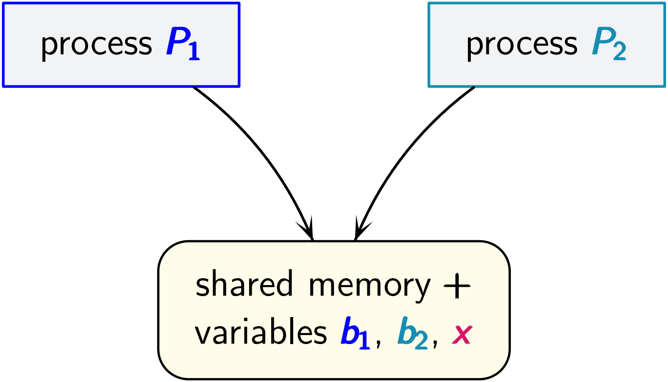


Peterson algorithm for mutual exclusion

Peterson algorithm for mutual exclusion



Peterson algorithm for mutual exclusion



where b_1, $b_2 \in \{0, 1\}$ and $x \in \{1, 2\}$

Peterson algorithm for mutual exclusion

for competing processes P_1, P_2

with additional shared variables:

- $b_1, b_2 \in \{0, 1\}$
- $x \in \{1, 2\}$

Peterson algorithm for mutual exclusion

for competing processes P_1, P_2

with additional shared variables:

- $b_1, b_2 \in \{0, 1\}$
- $x \in \{1, 2\}$

protocol for P_1:

LOOP FOREVER

noncritical actions;

$b_1 := 1$; $x := 2$;

AWAIT $x{=}1 \vee \neg b_2$ DO critical section OD

$b_1 := 0$

END LOOP

Peterson algorithm for mutual exclusion

for competing processes P_1, P_2

with additional shared variables:

- $b_1, b_2 \in \{0, 1\}$
- $x \in \{1, 2\}$

protocol for P_1:

LOOP FOREVER

noncritical actions;

atomic$\{b_1 := 1;\ \ x := 2\}$ $\longleftarrow$ atomic region

AWAIT $x{=}1 \vee \neg b_2$ DO critical section OD

$b_1 := 0$

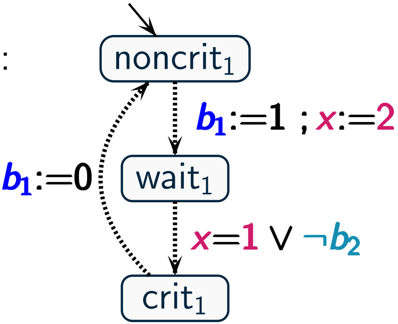
END LOOP

Peterson algorithm for mutual exclusion

for competing processes P_1, P_2

with additional shared variables:

- $b_1, b_2 \in \{0, 1\}$
- $x \in \{1, 2\}$



protocol for P_1:

LOOP FOREVER

noncritical actions;

atomic$\{b_1 := 1;\ \ x := 2\}$ ⟵ atomic region

AWAIT $x{=}1 \lor \neg b_2$ DO critical section OD

$b_1 := 0$

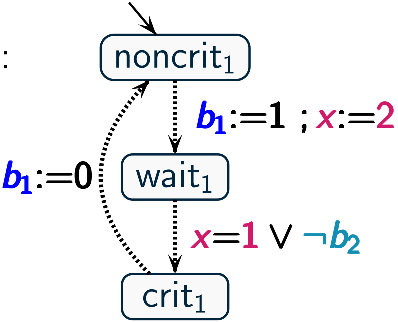
END LOOP

Peterson algorithm for mutual exclusion

for competing processes P_1, P_2

with additional shared variables:

- $b_1, b_2 \in \{0, 1\}$
- $x \in \{1, 2\}$



protocol for P_1:

LOOP FOREVER

noncritical actions;

atomic$\{b_1 := 1;\ x := 2\}$ ⟵ atomic region

AWAIT x=1 ∨ ¬b_2 DO critical section OD

$b_1 := 0$

END LOOP

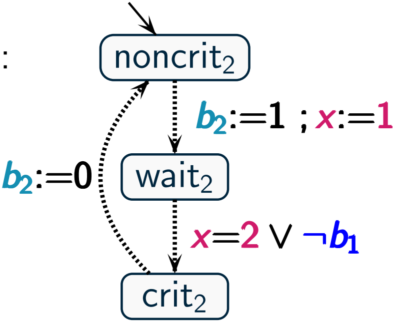
symmetric protocol for P_2

Peterson algorithm for mutual exclusion

for competing processes P_1, P_2

with additional shared variables:

- $b_1, b_2 \in \{0, 1\}$
- $x \in \{1, 2\}$



protocol for P_2:

LOOP FOREVER

noncritical actions;

atomic$\{b_2 := 1;\ \ x := 1\}$ $\longleftarrow$ atomic region

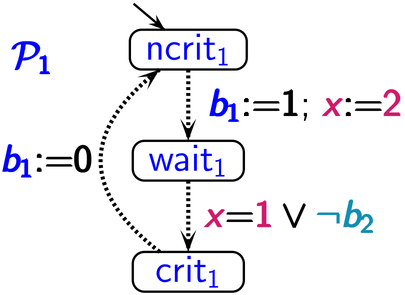
AWAIT $x{=}2 \vee \neg b_1$ DO critical section OD

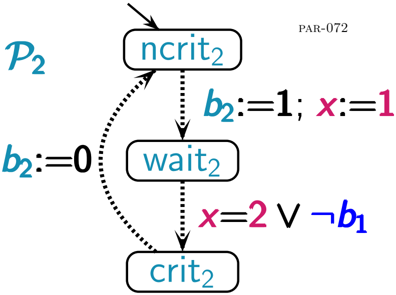
$b_2 := 0$

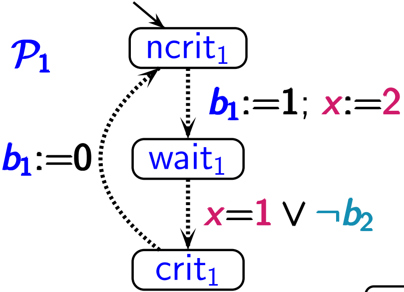
END LOOP

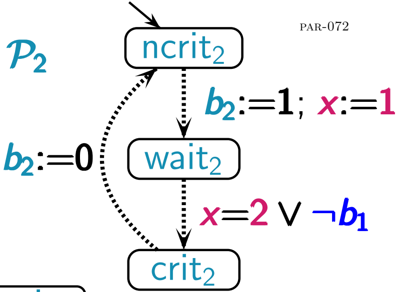
Program graph for Peterson algorithm

program graph for Peterson algorithm results from the interleaving of the program graphs for P_1, P_2

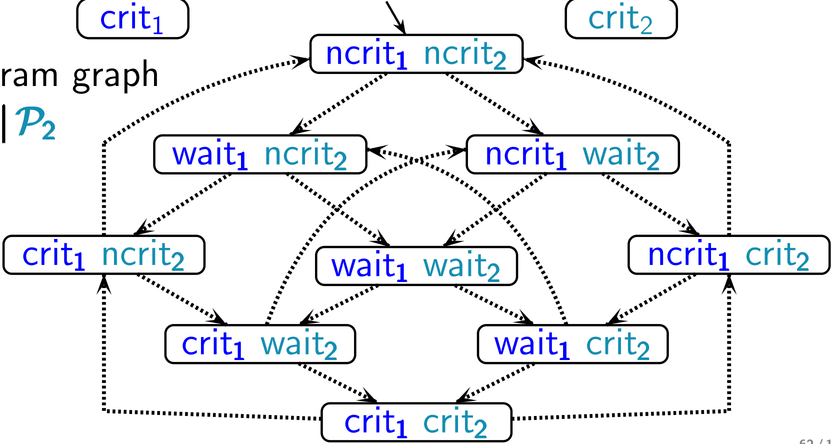

$\mathcal{P}_1$
ncrit_1
$b_1{:=}1;\ x{:=}2$
$b_1{:=}0$
wait_1
$x{=}1 \vee \neg b_2$
crit_1

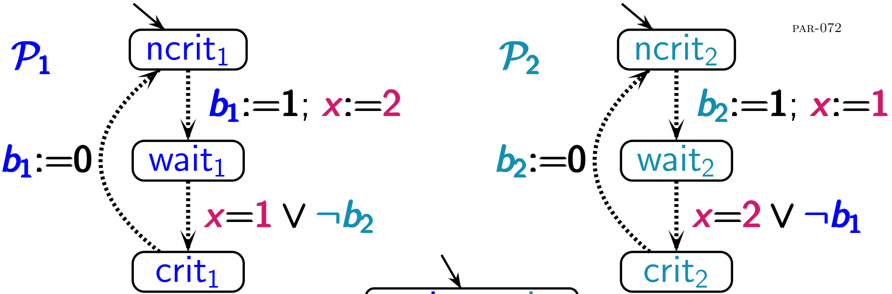

$\mathcal{P}_2$
ncrit_2
$b_2{:=}1;\ x{:=}1$
$b_2{:=}0$
wait_2
$x{=}2 \vee \neg b_1$
crit_2



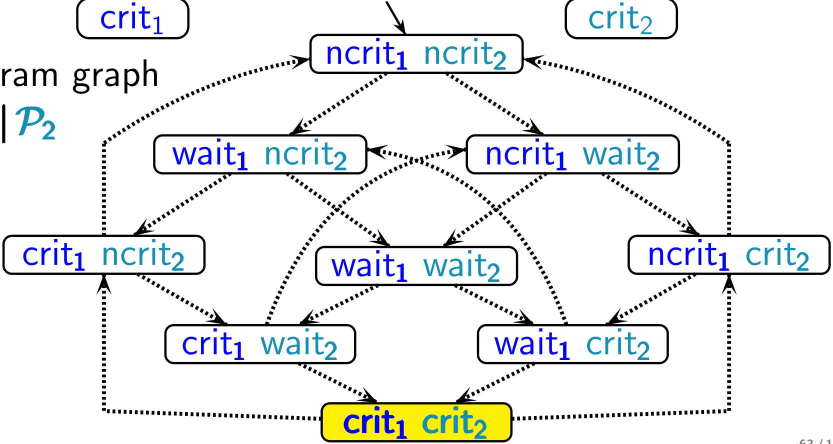


program graph
$\mathcal{P}_1 \,|||\, \mathcal{P}_2$

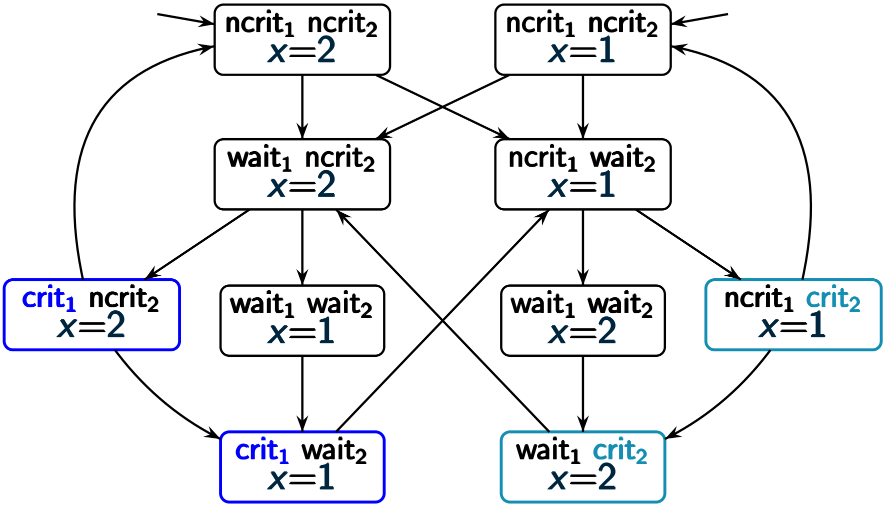




program graph
$\mathcal{P}_1 \,|||\, \mathcal{P}_2$

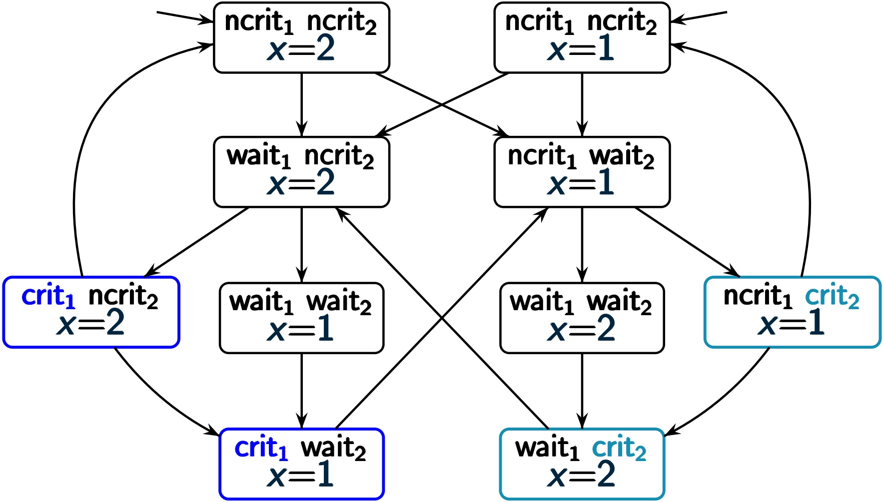


TS for the Peterson algorithm



value of b_i is given by $\mathbf{wait}_1 \vee \mathbf{crit}_i$

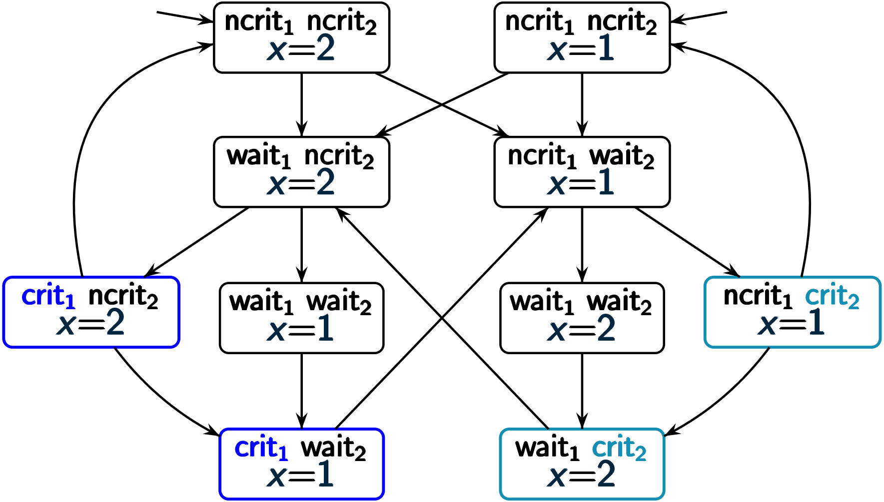
TS for the Peterson algorithm



value of b_i is given by **wait**$_1$ $\vee$ **crit**$_i$

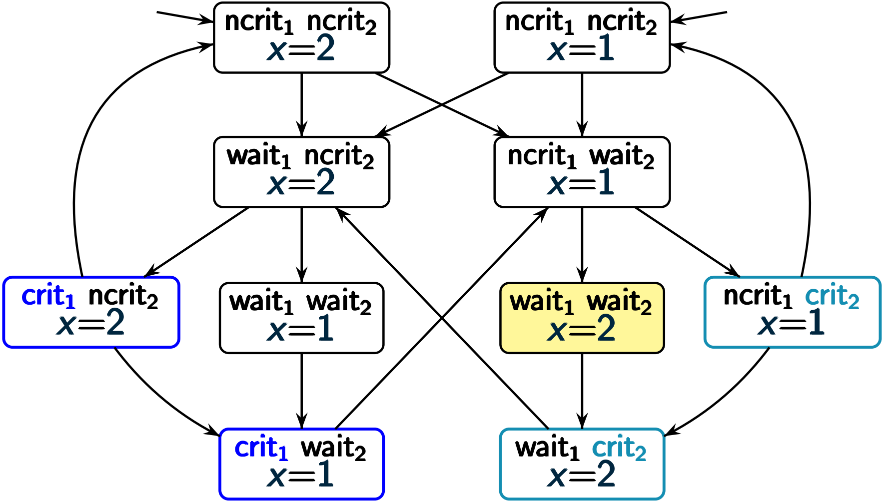
62 states are unreachable

TS for the Peterson algorithm



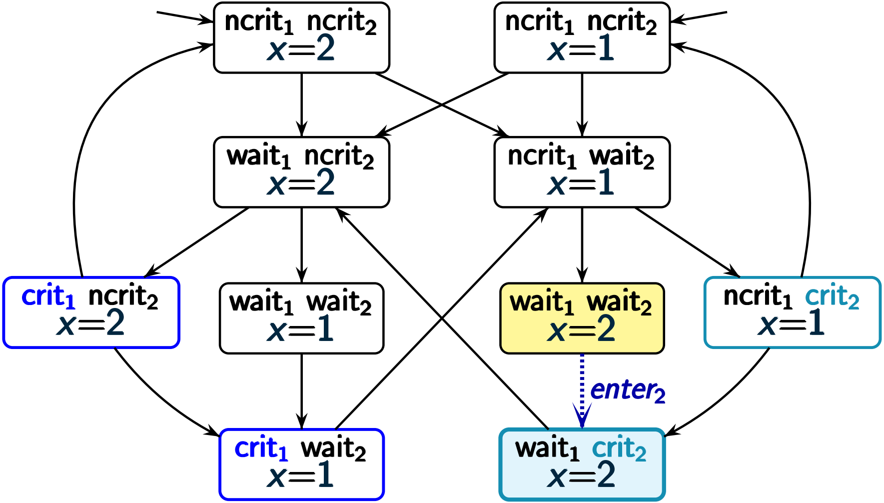
mutual exclusion property is satisfied as
no state $\langle\, crit_1, crit_2, x = \ldots \,\rangle$ is reachable

TS for the Peterson algorithm



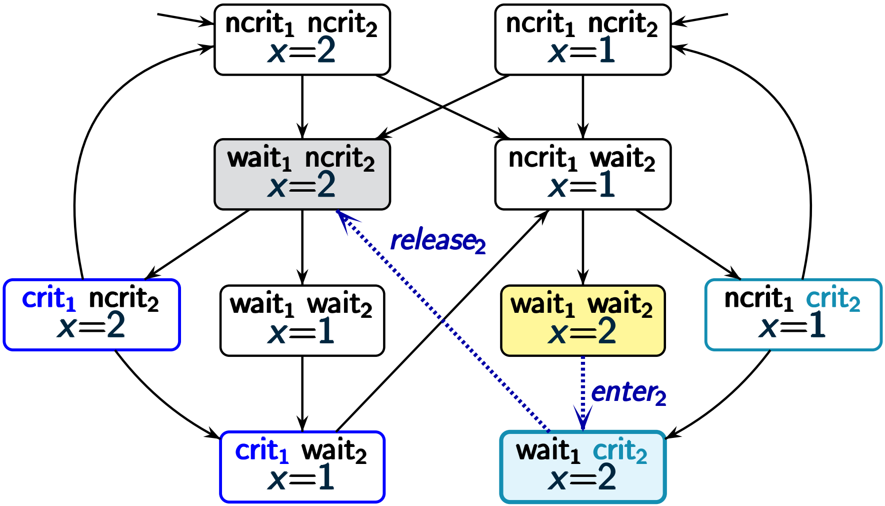
if both processes are waiting . . .

TS for the Peterson algorithm



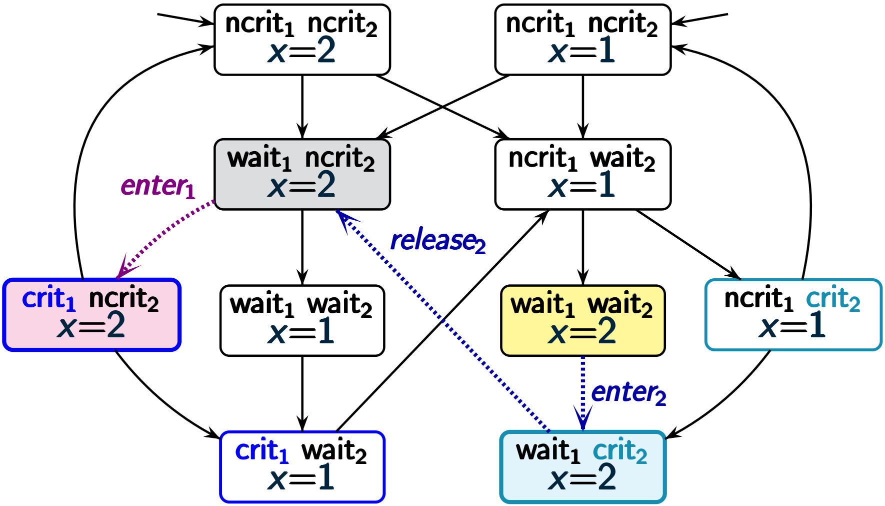
if both processes are waiting . . .

TS for the Peterson algorithm



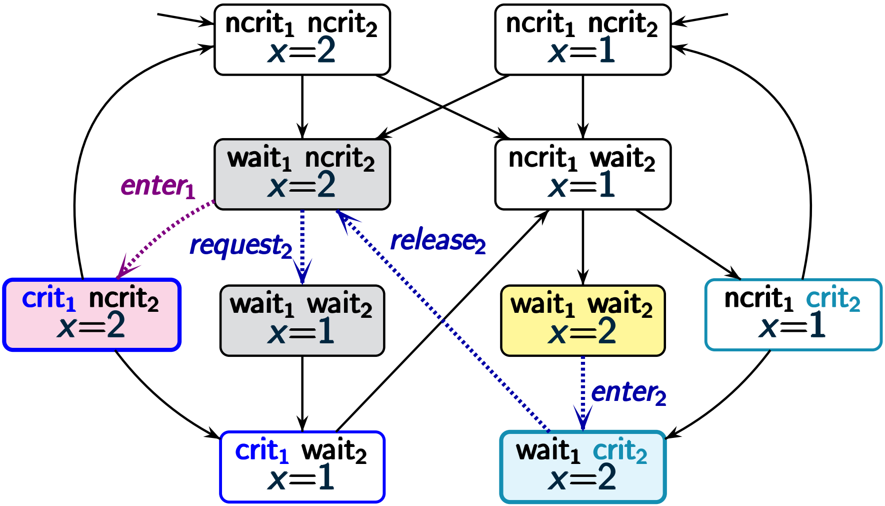
if both processes are waiting . . .

TS for the Peterson algorithm



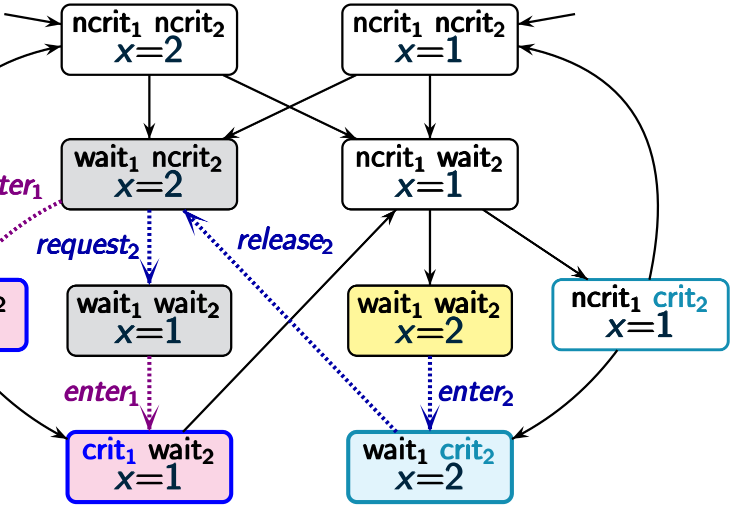
if both processes are waiting ...

TS for the Peterson algorithm



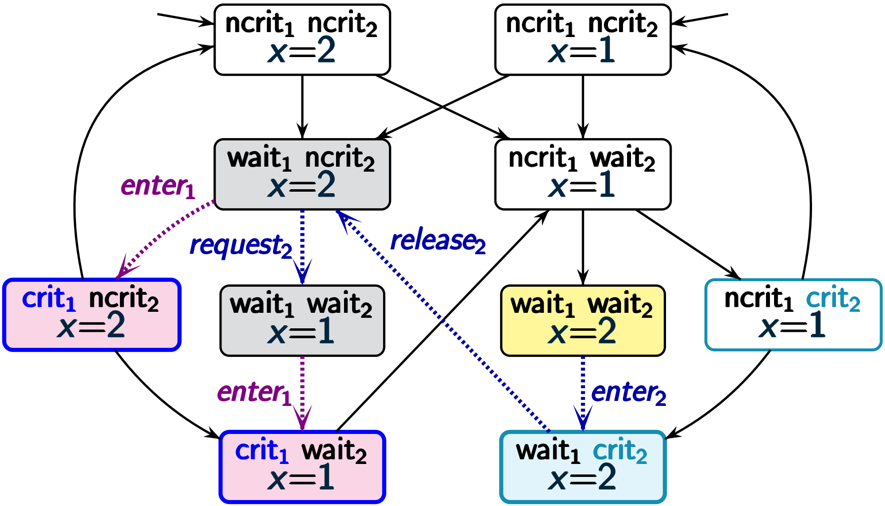
if both processes are waiting . . .

TS for the Peterson algorithm



if both processes are waiting ...

TS for the Peterson algorithm



liveness: the process that waits longer will enter its critical section first

Outline

1. Overview
2. Repetition
   - Channel systems
   - Guarded command languages
3. ProMeLa
4. SPIN

Guarded Command Language (GCL)

Guarded Command Language (GCL)

- high-level modeling language that contains features of imperative languages and nondeterministic choice

by Dijkstra, ca. 1975

Guarded Command Language (GCL)

- high-level modeling language that contains features of imperative languages and nondeterministic choice
- provides the basis for many modeling languages, e.g., input language of model checker SPIN

by Dijkstra, ca. 1975

Guarded Command Language (GCL)

- high-level modeling language that contains features of imperative languages and nondeterministic choice
- provides the basis for many modeling languages, e.g., input language of model checker SPIN
- operational semantics via program graphs

by Dijkstra, ca. 1975

Guarded Command Language (GCL)

- high-level modeling language that contains features of imperative languages and nondeterministic choice
- provides the basis for many modeling languages, e.g., input language of model checker SPIN
- operational semantics via program graphs

$$\boxed{\text{GCL-program}}$$
$$\Downarrow$$
$$\boxed{\text{program graph}}$$
$$\Downarrow$$
$$\boxed{\text{transition system}}$$

by Dijkstra, ca. 1975

Guarded Command Language (GCL)

guarded command $g \Rightarrow \textit{stmt}$

g	:	guard, i.e., Boolean condition on the program variables
$\textit{stmt}$	:	statement

Guarded Command Language (GCL)

guarded command $g \Rightarrow \textit{stmt}$ $\longleftarrow$ enabled if g holds

g	:	guard, i.e., Boolean condition on the program variables
$\textit{stmt}$	:	statement

Guarded Command Language (GCL)

guarded command $g \Rightarrow \mathit{stmt}$ $\longleftarrow$ $\boxed{\text{enabled if } g \text{ holds}}$

repetitive command/loop:

`DO` $::\ g \Rightarrow \mathit{stmt}$ `OD`

g : guard, i.e., Boolean condition on the program variables

stmt : statement

Guarded Command Language (GCL)

guarded command $g \Rightarrow \mathit{stmt}$ $\longleftarrow$ $\boxed{\text{enabled if } g \text{ holds}}$

repetitive command/loop:

$\texttt{DO} \; :: g \Rightarrow \mathit{stmt} \; \texttt{OD}$ $\longleftarrow$ $\boxed{\texttt{WHILE } g \texttt{ DO } \mathit{stmt} \texttt{ OD}}$

g : guard, i.e., Boolean condition on the program variables

stmt : statement

Guarded Command Language (GCL)

guarded command $g \Rightarrow \mathit{stmt}$ $\longleftarrow$ enabled if g holds

repetitive command/loop:

$\texttt{DO} :: g \Rightarrow \mathit{stmt}\ \texttt{OD}$ $\longleftarrow$ $\texttt{WHILE}\ g\ \texttt{DO}\ \mathit{stmt}\ \texttt{OD}$

conditional command:

$$
\begin{array}{lll}
\texttt{IF} & :: & g \Rightarrow \mathit{stmt}_1 \\
 & :: & \neg g \Rightarrow \mathit{stmt}_2 \\
\texttt{FI} & &
\end{array}
$$

Guarded Command Language (GCL)

guarded command $g \Rightarrow stmt$ $\longleftarrow$ enabled if g holds

repetitive command/loop:

$\mathtt{DO}\ :: g \Rightarrow stmt\ \mathtt{OD}$ $\longleftarrow$ $\mathtt{WHILE}\ g\ \mathtt{DO}\ stmt\ \mathtt{OD}$

conditional command:

$$\begin{array}{lll} \mathtt{IF} & :: & g \Rightarrow stmt_1 \\ & :: & \neg g \Rightarrow stmt_2 \\ \mathtt{FI} & & \end{array} \longleftarrow \begin{array}{l} \mathtt{IF}\ g \\ \quad \mathtt{THEN}\ stmt_1 \\ \quad \mathtt{ELSE}\ stmt_2 \\ \mathtt{FI} \end{array}$$

Guarded Command Language (GCL)

guarded command $g \Rightarrow stmt$ $\longleftarrow$ enabled if g holds

repetitive command/loop:

DO :: $g \Rightarrow stmt$ OD $\longleftarrow$ WHILE g DO $stmt$ OD

conditional command:

```
IF ::  g ⇒ stmt1
   :: ¬g ⇒ stmt2
FI
```

$\longleftarrow$

```
IF g
   THEN stmt1
   ELSE stmt2
FI
```

symbol :: stands for the nondeterministic choice between enabled guarded commands

Guarded Command Language (GCL)

guarded command $g \Rightarrow \mathit{stmt}$ $\longleftarrow$ $\boxed{\text{enabled if } g \text{ holds}}$

repetitive command/loop:

$\texttt{DO} :: g \Rightarrow \mathit{stmt}\ \texttt{OD}$ $\longleftarrow$ $\boxed{\texttt{WHILE}\ g\ \texttt{DO}\ \mathit{stmt}\ \texttt{OD}}$

conditional command:

$$\begin{array}{lll} \texttt{IF} & :: & g \Rightarrow \mathit{stmt}_1 \\ & :: & \neg g \Rightarrow \mathit{stmt}_2 \\ \texttt{FI} & & \end{array} \longleftarrow \boxed{\begin{array}{ll} \texttt{IF}\ g & \\ \quad \texttt{THEN} & \mathit{stmt}_1 \\ \quad \texttt{ELSE} & \mathit{stmt}_2 \\ \texttt{FI} & \end{array}}$$

GCL: nondeterministic choices between arbitrary guarded commands in DO … OD and IF … FI

Guarded Command Language (GCL)

modeling language with nondeterministic choice

$$
\begin{aligned}
\mathit{stmt} \stackrel{\text{def}}{=}\ & x := \mathit{expr} \quad \Big| \quad \mathit{stmt}_1 ; \mathit{stmt}_2 \quad \Big| \\
& \texttt{DO} :: g_1 \Rightarrow \mathit{stmt}_1 \ \ \ldots \ \ :: g_n \Rightarrow \mathit{stmt}_n \ \texttt{OD} \\
& \texttt{IF} :: g_1 \Rightarrow \mathit{stmt}_1 \ \ \ldots \ \ :: g_n \Rightarrow \mathit{stmt}_n \ \texttt{FI} \\
& \quad \vdots
\end{aligned}
$$

where x is a typed variable and expr an expression of the same type

Guarded Command Language (GCL)

modeling language with nondeterministic choice

$$\mathit{stmt} \stackrel{\text{def}}{=} x := \mathit{expr} \;\; \Big| \;\; \mathit{stmt}_1 ; \mathit{stmt}_2 \;\; \Big|$$

$$\texttt{DO} :: g_1 \Rightarrow \mathit{stmt}_1 \;\; \ldots \;\; :: g_n \Rightarrow \mathit{stmt}_n \; \texttt{OD}$$

$$\texttt{IF} :: g_1 \Rightarrow \mathit{stmt}_1 \;\; \ldots \;\; :: g_n \Rightarrow \mathit{stmt}_n \; \texttt{FI}$$

$$\vdots$$

semantics of a GCL-program: program graph

Guarded Command Language (GCL)

modeling language with nondeterministic choice

$$stmt \overset{\text{def}}{=} x := expr \;\mid\; stmt_1 ; stmt_2 \;\mid$$
$$\texttt{DO} :: g_1 \Rightarrow stmt_1 \;\ldots\; :: g_n \Rightarrow stmt_n \;\texttt{OD}$$
$$\texttt{IF} :: g_1 \Rightarrow stmt_1 \;\ldots\; :: g_n \Rightarrow stmt_n \;\texttt{FI}$$
$$\vdots$$

semantics of a GCL-program: program graph

- locations are statements
- plus auxiliary location for termination

Guarded Command Language (GCL)

modeling language with nondeterministic choice

$$stmt \stackrel{\text{def}}{=} x := expr \;\Big|\; stmt_1 ; stmt_2 \;\Big|$$
$$\mathtt{DO} :: g_1 \Rightarrow stmt_1 \;\ldots\; :: g_n \Rightarrow stmt_n \;\mathtt{OD}$$
$$\vdots$$

conditional transition for assignment:

$$x := expr \xrightarrow{true:\alpha} exit$$

where α has the effect of "$x := expr$"

Guarded Command Language (GCL)

modeling language with nondeterministic choice

$$stmt \stackrel{\text{def}}{=} x := expr \;\Big|\; stmt_1 ; stmt_2 \;\Big|$$
$$\mathtt{DO} :: g_1 \Rightarrow stmt_1 \;\ldots\; :: g_n \Rightarrow stmt_n \;\mathtt{OD}$$
$$\vdots$$

conditional transition for assignment:

$$x := expr \xhookrightarrow{true:\alpha} exit$$

analogously: multiple assignments in an atomic step

Guarded Command Language (GCL)

modeling language with nondeterministic choice

$$\mathit{stmt} \stackrel{\text{def}}{=} x := \mathit{expr} \quad \big| \quad \mathit{stmt}_1 ; \mathit{stmt}_2 \quad \big|$$
$$\texttt{DO} :: g_1 \Rightarrow \mathit{stmt}_1 \quad \ldots \quad :: g_n \Rightarrow \mathit{stmt}_n \ \texttt{OD}$$
$$\vdots$$

two SOS-rules for the PG-semantics of
sequential composition

Guarded Command Language (GCL)

modeling language with nondeterministic choice

$$stmt \stackrel{\text{def}}{=} x := expr \quad | \quad stmt_1 ; stmt_2 \quad |$$

$$\text{DO} :: g_1 \Rightarrow stmt_1 \quad \ldots \quad :: g_n \Rightarrow stmt_n \ \text{OD}$$

$$\vdots$$

$$\frac{stmt_1 \xhookrightarrow{g:\alpha} stmt_1'}{stmt_1 ; stmt_2 \xhookrightarrow{g:\alpha} stmt_1' ; stmt_2}$$

$$exit \neq stmt_1'$$

Guarded Command Language (GCL)

modeling language with nondeterministic choice

$$stmt \stackrel{\text{def}}{=} x := expr \;\Big|\; stmt_1; stmt_2 \;\Big|$$

$$\texttt{DO} :: g_1 \Rightarrow stmt_1 \;\; \ldots \;\; :: g_n \Rightarrow stmt_n \;\texttt{OD}$$

$$\vdots$$

$$\frac{stmt_1 \xrightarrow{g:\alpha} stmt_1'}{stmt_1; stmt_2 \xrightarrow{g:\alpha} stmt_1'; stmt_2}$$

$$\frac{stmt_1 \xrightarrow{g:\alpha} exit}{stmt_1; stmt_2 \xrightarrow{g:\alpha} stmt_2}$$

Guarded Command Language (GCL)

modeling language with nondeterministic choice

$$stmt \stackrel{\text{def}}{=} x := expr \quad | \quad stmt_1 ; stmt_2 \quad |$$

$$\mathtt{IF} :: g_1 \Rightarrow stmt_1 \quad \ldots \quad :: g_n \Rightarrow stmt_n \ \mathtt{FI}$$

$$\vdots$$

single SOS-rule for the PG-semantics of
conditional statements

Guarded Command Language (GCL)

modeling language with nondeterministic choice

$$stmt \stackrel{\text{def}}{=} x := expr \;\mid\; stmt_1; stmt_2 \;\mid$$
$$\texttt{IF} :: g_1 \Rightarrow stmt_1 \;\ldots\; :: g_n \Rightarrow stmt_n \;\texttt{FI}$$
$$\vdots$$

Let $cstmt$ be an `IF` –`FI` -statement as above.

$$\frac{stmt_i \xhookrightarrow{h:\alpha} stmt'}{cstmt \xhookrightarrow{g_i \wedge h:\alpha} stmt'}$$

Guarded Command Language (GCL)

modeling language with nondeterministic choice

$$
\begin{array}{lll}
stmt & \overset{\text{def}}{=} & x := expr \;\;\big|\;\; stmt_1 ; stmt_2 \;\;\big| \\
 & & \texttt{DO} :: g_1 \Rightarrow stmt_1 \;\; \ldots \;\; :: g_n \Rightarrow stmt_n \;\texttt{OD} \\
 & & \;\;\vdots
\end{array}
$$

SOS-rules for the PG-semantics of loops

Guarded Command Language (GCL)

modeling language with nondeterministic choice

$$\mathit{stmt} \stackrel{\text{def}}{=} x := \mathit{expr} \;\Big|\; \mathit{stmt}_1; \mathit{stmt}_2 \;\Big|$$
$$\texttt{DO} :: g_1 \Rightarrow \mathit{stmt}_1 \;\ldots\; :: g_n \Rightarrow \mathit{stmt}_n \;\texttt{OD}$$
$$\vdots$$

Let *loop* be a DO –OD -statement as above.

$$\frac{\mathit{stmt}_i \xhookrightarrow{h:\alpha} \mathit{stmt}'}{\mathit{loop} \xhookrightarrow{g_i \wedge h:\alpha} \mathit{stmt}'; \mathit{loop}}$$

$\mathit{exit}; \mathit{loop} \mathrel{\widehat{=}} \mathit{loop}$

Guarded Command Language (GCL)

modeling language with nondeterministic choice

$$stmt \stackrel{\text{def}}{=} x := expr \;\;|\;\; stmt_1; stmt_2 \;\;|$$
$$\texttt{DO} :: g_1 \Rightarrow stmt_1 \;\; \ldots \;\; :: g_n \Rightarrow stmt_n \; \texttt{OD}$$
$$\vdots$$

Let *loop* be a DO –OD -statement as above.

$$\frac{stmt_i \xrightarrow{h:\alpha} stmt'}{loop \xrightarrow{g_i \wedge h:\alpha} stmt'; loop} \qquad \frac{f = \neg g_1 \wedge \ldots \wedge \neg g_n}{loop \xrightarrow{f:skip} exit}$$

Outline

1. Overview
2. Repetition
   - Channel systems
   - Guarded command languages
3. **ProMeLa**
4. SPIN

ProMeLa: general structure of a ProMeLa model

ProMeLa model consists of...

ProMeLa: general structure of a ProMeLa model

ProMeLa model consists of. . .

- type declarations,
  e.g., $\textit{\textbf{mtype}} = \{\textit{MSG}, \textit{ACK}\};$

ProMeLa: general structure of a ProMeLa model

ProMeLa model consists of...

- type declarations,
  e.g., $\boldsymbol{mtype} = \{\boldsymbol{MSG}, \boldsymbol{ACK}\};$
- global variables declarations,
  e.g., `bool` $\boldsymbol{flag} \ = \ \boldsymbol{true};$

ProMeLa model consists of...

- type declarations,
  e.g., $\textit{mtype} = \{\textit{MSG}, \textit{ACK}\};$
- global variables declarations,
  e.g., **bool** *flag* $=$ *true*;
- channel declarations,
  e.g., **chan** *ch* $=$ [5] **of** $\{T_1, T_2, \ldots, T_k\};$

ProMeLa: general structure of a ProMeLa model

ProMeLa model consists of...

- type declarations,
  e.g., $\textit{mtype} = \{\textit{MSG}, \textit{ACK}\};$
- global variables declarations,
  e.g., $\texttt{bool}\ \textit{flag}\ =\ \textit{true};$
- channel declarations,
  e.g., $\texttt{chan}\ \textit{ch}\ =\ [5]\ \texttt{of}\ \{T_1, T_2, \ldots, T_k\};$
- process declarations,
  e.g., $\texttt{proctype}\ \textit{Sender}(\textit{args})\{\ldots\};$

ProMeLa: general structure of a ProMeLa model

ProMeLa model consists of...

- type declarations,
  e.g., $\textit{mtype} = \{\textit{MSG}, \textit{ACK}\}$;
- global variables declarations,
  e.g., **bool** $\textit{flag} = \textit{true}$;
- channel declarations,
  e.g., **chan** $\textit{ch} = [5]$ **of** $\{T_1, T_2, \ldots, T_k\}$;
- process declarations,
  e.g., **proctype** $\textit{Sender}(\textit{args})\{\ldots\}$;
- the init process (optional),e.g., **init** $(\textit{args})\{\ldots\}$;

ProMeLa: general structure of a ProMeLa model

ProMeLa model consists of...

- type declarations,
  e.g., $\textbf{\textit{mtype}} = \{\mathit{MSG}, \mathit{ACK}\};$
- global variables declarations,
  e.g., $\texttt{bool}\ \mathit{flag} = \mathit{true};$

Data types (for global and local variables)

Data types (for global and local variables)

- basic data types:

boolean	1 Bit	`bool` *flag* = *true*;
bytes	8 Bit	`byte` *answer* = 42;
shorts	16 Bit	`short` *value* = 7;
integer	32 Bit	`int` *i* = 99;

Data types (for global and local variables)

- basic data types:

boolean	1 Bit	**bool** *flag* = *true*;
bytes	8 Bit	**byte** *answer* = 42;
shorts	16 Bit	**short** *value* = 7;
integer	32 Bit	**int** *i* = 99;

- structured data types:

arrays	**int** *myarray*[12];
records	**typedef** *myrec*{**int** r_1; **int** r_2; }

Things to know about variables and data types

- variables **must be declared**

Things to know about variables and data types

- variables **must be declared**
- variables are **strictly typed** (conflicts are found at on-the-fly state-space generation)

Things to know about variables and data types

- variables **must be declared**
- variables are **strictly typed** (conflicts are found at on-the-fly state-space generation)
- variables are assigned by **initialization**, **assignment**, **argument passing** or **message passing via channels**

Things to know about variables and data types

- variables **must be declared**
- variables are **strictly typed** (conflicts are found at on-the-fly state-space generation)
- variables are assigned by **initialization**, **assignment**, **argument passing** or **message passing via channels**
- default **initial value is 0**

Things to know about variables and data types

- variables **must be declared**
- variables are **strictly typed** (conflicts are found at on-the-fly state-space generation)
- variables are assigned by **initialization**, **assignment**, **argument passing** or **message passing via channels**
- default **initial value is 0**
- expressions include most **arithmetic**, **relational** and **logical operators** of C

ProMeLa: general structure of a ProMeLa model

ProMeLa model consists of...

- type declarations,
  e.g., $\textit{mtype} = \{MSG, ACK\};$
- global variables declarations,
  e.g., $\mathtt{bool}\ \textit{flag} = \textit{true};$
- channel declarations,
  e.g., $\mathtt{chan}\ \textit{ch} = [5]\ \mathtt{of}\ \{T_1, T_2, \ldots, T_k\};$
- process declarations,
  e.g., $\mathtt{proctype}\ \textit{Sender}(\textit{args})\{\ldots\};$
- the init process (optional),e.g., $\mathtt{init}\ (\textit{args})\{\ldots\};$

ProMeLa model consists of...

- channel declarations,
  e.g., **chan** ch = [5] **of** $\{T_1, T_2, \ldots, T_k\}$;

The modeling language ProMeLa

Channel declarations:

$\texttt{chan}$ *name* $=$ [*capacity*] $\texttt{of}$ $\{T_1, T_2, \ldots, T_k\}$;

name	:	name of the channel
capacity	:	capacity of the FIFO channel
T_i, $1 \leq i \leq k$	:	type of transmittable data (tuples)

The modeling language ProMeLa

Channel declarations:

$$\texttt{chan}\ \mathit{name} = [\mathit{capacity}]\ \texttt{of}\ \{T_1, T_2, \ldots, T_k\};$$

name	:	name of the channel
capacity	:	capacity of the FIFO channel
T_i, $1 \leq i \leq k$	:	type of transmittable data (tuples)

Communication:

- synchronous message passing $\longleftarrow$ *capacity* **0**
- asynchronous message passing $\longleftarrow$ *capacity* $\geqslant 1$

The modeling language ProMeLa

Channel declarations:

$$\texttt{chan}\ \mathit{name} = [\mathit{capacity}]\ \texttt{of}\ \{T_1, T_2, \ldots, T_k\};$$

name	:	name of the channel
capacity	:	capacity of the FIFO channel
T_i, $1 \leq i \leq k$	:	type of transmittable data (tuples)

Communication actions:

- sending: $\mathit{name}!\mathit{expr}_1, \mathit{expr}_2, \ldots, \mathit{expr}_k;$
- receiving: $\mathit{name}?x_1, x_2, \ldots, x_k;$

Things to know about channels

The modeling language ProMeLa

Things to know about channels

- synchronous communication: **sending can happen** iff there is a **corresponding receive** that can be executed **simultaneously**.

Things to know about channels

- synchronous communication: **sending can happen** iff there is a **corresponding receive** that can be executed **simultaneously**.
- asynchronous communication: **reading blocks** when the **channel is empty**, **writing** is **either blocking or losing** (SPIN option) when the **channel is full**.

Things to know about channels

- synchronous communication: **sending can happen** iff there is a **corresponding receive** that can be executed **simultaneously**.
- asynchronous communication: **reading blocks** when the **channel is empty**, **writing** is **either blocking or losing** (SPIN option) when the **channel is full**.

rule for a **lossy write**:
$$\frac{}{c!expr \xrightarrow{full(c):skip} exit}$$

ProMeLa: general structure of a ProMeLa model

ProMeLa: general structure of a ProMeLa model

ProMeLa model consists of...

- type declarations,
  e.g., $\textit{\textbf{mtype}} = \{\textit{MSG}, \textit{ACK}\};$
- global variables declarations,
  e.g., $\texttt{\textbf{bool}}\ \textit{flag} = \textit{true};$
- channel declarations,
  e.g., $\texttt{\textbf{chan}}\ \textit{ch} = [5]\ \texttt{\textbf{of}}\ \{T_1, T_2, \ldots, T_k\};$
- process declarations,
  e.g., $\texttt{\textbf{proctype}}\ \textit{Sender}(\textit{args})\{\ldots\};$
- the init process (optional),e.g., $\texttt{\textbf{init}}\ (\textit{args})\{\ldots\};$

ProMeLa: general structure of a ProMeLa model

ProMeLa model consists of...

- process declarations,
  e.g., **proctype** *Sender*(*args*){...};
- the init process (optional),e.g., **init** (*args*){...};

Processes

The modeling language ProMeLa

Processes

$$\mathit{proc} ::= [\texttt{active}]\ \texttt{proctype}\ \mathit{name}(\mathit{args})\{\mathit{stmt}\}$$

$$\mathit{stmt} ::= x := \mathit{expr} \mid \mathit{stmt}_1 ; \mathit{stmt}_2 \mid \dots \mid$$

$$\texttt{DO} :: g_1 \Rightarrow \mathit{stmt}_1 \dots :: g_n \Rightarrow \mathit{stmt}_n\ \texttt{OD} \mid$$

$$\texttt{IF} :: g_1 \Rightarrow \mathit{stmt}_1 \dots :: g_n \Rightarrow \mathit{stmt}_n\ \texttt{FI}$$

active	:	marks the process active
name	:	name of the process
args	:	process parameter variables
stmt	:	statement

The modeling language ProMeLa

Things to know about processes

- there can be **more than one** process

Things to know about processes

- there can be **more than one** process
- initially: `init` **process** and **all `active` processes** (together at least one)

Things to know about processes

- there can be **more than one** process
- initially: `init` **process** and **all `active` processes** (together at least one)
- creation **at any point inside the model** using the **`run`** statement (max. 255 processes)
- processes **execute concurrently**

The modeling language ProMeLa

Things to know about processes

- there can be **more than one** process
- initially: `init` **process** and **all `active` processes** (together at least one)
- creation **at any point inside the model** using the **`run`** statement (max. 255 processes)
- processes **execute concurrently**
- process termination: control flow **reaches `exit` location**
- program termination: **all processes** reached their **`exit` location**

The modeling language ProMeLa

Statements

$$\begin{array}{rl} stmt ::= & x := expr \;\Big|\; stmt_1 ; stmt_2 \;\Big|\; \ldots \;\Big| \\ & \texttt{DO} :: g_1 \Rightarrow stmt_1 \ldots :: g_n \Rightarrow stmt_n \; \texttt{OD} \;\Big| \\ & \texttt{IF} :: g_1 \Rightarrow stmt_1 \ldots :: g_n \Rightarrow stmt_n \; \texttt{FI} \end{array}$$

The modeling language ProMeLa

Statements

$$\mathit{stmt} ::= x := \mathit{expr} \;\Big|\; \mathit{stmt}_1 ; \mathit{stmt}_2 \;\Big|\; \ldots \;\Big|$$
$$\texttt{DO} :: g_1 \Rightarrow \mathit{stmt}_1 \ldots :: g_n \Rightarrow \mathit{stmt}_n \;\texttt{OD} \;\Big|$$
$$\texttt{IF} :: g_1 \Rightarrow \mathit{stmt}_1 \ldots :: g_n \Rightarrow \mathit{stmt}_n \;\texttt{FI}$$

where g_i are guards, i.e., boolean expressions:

$$\mathit{bexpr} ::= x \odot \mathit{expr} \;\Big|\; c!\mathit{expr} \;\Big|\; c?x \;\Big|\; \ldots \;\Big|$$
$$\neg \mathit{bexpr} \;\Big|\; \mathit{bexpr}_1 \sim \mathit{bexpr}_2$$

with $\odot \in \{<, \leq, =, \geq, >\}$ and $\sim \in \{\wedge, \vee, \rightarrow, \ldots\}$

Simple statements

The modeling language ProMeLa

Simple statements

skip statement: `skip` ***true***

Simple statements

skip statement:	**`skip`**	***true***
goto statement:	**`goto`** ℓ	***true***

Simple statements

skip statement:	**`skip`**	***true***
goto statement:	**`goto`** ℓ	***true***
assignments:	$x := expr$	***true***

The modeling language ProMeLa

Simple statements

skip statement:	**skip**	***true***
goto statement:	**goto** ℓ	***true***
assignments:	$x := expr$	***true***
expressions:	$expr$	$expr \neq 0$

The modeling language ProMeLa

Simple statements

skip statement:	**skip**	***true***
goto statement:	**goto** ℓ	***true***
assignments:	$x := expr$	***true***
expressions:	$expr$	$expr \neq 0$
run statement:	**run** $P()$	if possible

The modeling language ProMeLa

Simple statements

skip statement:	`skip`	***true***
goto statement:	**goto** ℓ	***true***
assignments:	$x := expr$	***true***
expressions:	$expr$	$expr \neq 0$
run statement:	**run** $P()$	if possible
sending/ receiving:	$ch!expr$, $ch?x$	if enabled

The modeling language ProMeLa

Simple statements

skip statement:	**skip**	***true***
goto statement:	**goto** ℓ	***true***
assignments:	$x := \mathit{expr}$	***true***
expressions:	expr	$\mathit{expr} \neq 0$
run statement:	**run** $P()$	if possible
sending/ receiving:	$\mathit{ch}!\mathit{expr}$, $\mathit{ch}?x$	if enabled
atomic statements:	**atomic**$\{\mathit{stmt}\}$	first statement

Simple statements

skip statement:	`skip`	*true*
goto statement:	`goto` ℓ	*true*
assignments:	$x := \mathit{expr}$	*true*
expressions:	*expr*	$\mathit{expr} \neq 0$
run statement:	`run` $P()$	if possible
sending/ receiving:	$\mathit{ch}!\mathit{expr}$, $\mathit{ch}?x$	if enabled
atomic statements:	`atomic`$\{\mathit{stmt}\}$	first statement
$\vdots$	$\vdots$	$\vdots$
printf statement:	`printf ...`	*true*
assert statement:	`assert`(bexpr)	*true*

Conditional commands

$$
\begin{array}{lcrcl}
\texttt{IF} & :: & g_1 & \Rightarrow & \mathit{stmt}_1 \\
 & & & \vdots & \\
 & :: & g_n & \Rightarrow & \mathit{stmt}_n \\
 & :: & \texttt{else} & \Rightarrow & \mathit{stmt}_0 \\
\texttt{FI} & & & &
\end{array}
$$

Conditional commands

$$
\begin{array}{lll}
\texttt{IF} & :: & g_1 \Rightarrow \mathit{stmt}_1 \\
 & & \vdots \\
 & :: & g_n \Rightarrow \mathit{stmt}_n \\
 & :: & \texttt{else} \Rightarrow \mathit{stmt}_0 \\
\texttt{FI} & &
\end{array}
$$

two-step semantics!

possible interleaving between evaluation of guard g_i and the execution of statement stmt_i

The modeling language ProMeLa

Conditional commands

$$
\begin{array}{lrrcl}
\texttt{IF} & :: & g_1 & \Rightarrow & \mathit{stmt}_1 \\
 & & & \vdots & \\
 & :: & g_n & \Rightarrow & \mathit{stmt}_n \\
 & :: & \textbf{else} & \Rightarrow & \mathit{stmt}_0 \\
\texttt{FI} & & & &
\end{array}
$$

two-step semantics!
possible interleaving between evaluation of guard g_i and the execution of statement stmt_i

Things to know about if-statements

- **nondeterministic choice** between enabled guards
- **else** case if **none of the guards** is enabled
- if-statement is **executable** if there is **at least one** enabled guard and **blocked otherwise**

Loop commands

$$
\begin{array}{lll}
\texttt{DO} & :: \quad g_1 & \Rightarrow \textit{stmt}_1 \\
 & & \vdots \\
 & :: \quad g_n & \Rightarrow \textit{stmt}_n \\
 & :: \ \texttt{else} & \Rightarrow \textit{stmt}_0 \\
\texttt{OD} & &
\end{array}
$$

Loop commands

$$
\begin{array}{lrl}
\texttt{DO} & :: \quad g_1 & \Rightarrow \mathit{stmt}_1 \\
 & & \vdots \\
 & :: \quad g_n & \Rightarrow \mathit{stmt}_n \\
 & :: \ \mathbf{\texttt{else}} & \Rightarrow \mathit{stmt}_0 \\
\texttt{OD} & &
\end{array}
$$

again two-step semantics and loop statement has **blocking semantics**, meaning that the loop exits on **`break`** only!

Loop commands

$$\begin{array}{lllll} \texttt{DO} & :: & g_1 & \Rightarrow & stmt_1 \\ & & & \vdots & \\ & :: & g_n & \Rightarrow & stmt_n \\ & :: & \texttt{else} & \Rightarrow & stmt_0 \\ \texttt{OD} & & & & \end{array}$$

again two-step semantics and loop statement has **blocking semantics**, meaning that the loop exits on **`break`** only!

standard GCL semantics:

$$\frac{f = \neg g_1 \wedge \ldots \wedge \neg g_n}{loop \xhookrightarrow{f:skip} exit}$$

ProMeLa semantics:

$$\frac{break \xhookrightarrow{true:skip} exit}{loop \xhookrightarrow{g_i:skip} exit}$$

$(stmt_i = break)$

Loop commands

$$
\begin{array}{lllcl}
\texttt{DO} & :: & g_1 & \Rightarrow & \mathit{stmt}_1 \\
 & & & \vdots & \\
 & :: & g_n & \Rightarrow & \mathit{stmt}_n \\
 & :: & \texttt{else} & \Rightarrow & \mathit{stmt}_0 \\
\texttt{OD} & & & &
\end{array}
$$

again two-step semantics and loop statement has **blocking semantics**, meaning that the loop exits on **break** only!

Things to know about do-statements

- **nondeterministic choice** between enabled guards
- **else** case if **none of the guards** is enabled
- do-statement is **executable** if there is **at least one** enabled guard and **blocked otherwise**

Nondeterminism occurs because of

Nondeterminism occurs because of

- the choice of different possible actions within each process (**`IF`** ... **`FI`** and **`DO`** ... **`OD`**)

Nondeterminism occurs because of

- the choice of different possible actions within each process (**IF** ... **FI** and **DO** ... **OD**)
- non-deterministic scheduling of processes (interleaving). Use **atomic**{*stmt*$_1$; ...; *stmt*$_n$} statements to avoid interleaving.

The modeling language ProMeLa

Nondeterminism occurs because of

- the choice of different possible actions within each process (`IF` ... `FI` and `DO` ... `OD`)
- non-deterministic scheduling of processes (interleaving). Use `atomic`{$stmt_1$; ...; $stmt_n$} statements to avoid interleaving.

Things to know about atomic regions

- execution in a **single step** instead of interleaved
- executable if the **first statement** is executable
- atomicity is **broken** if **any** of the statements is **blocking**

Example: ProMeLa model of Petersons algorithm

Example: ProMeLa model of Petersons algorithm

bool $b1, b2$;
int x;
bool $incrit1, incrit2$;

Example: ProMeLa model of Petersons algorithm

$\texttt{bool}\ b1, b2;$
$\texttt{int}\ x;$
$\texttt{bool}\ incrit1, incrit2;$

$\texttt{active proctype}\ p1()\{$
$\quad \texttt{DO}$
$\qquad :: \texttt{atomic}\{b1 = true; x = 2\};$
$\qquad \texttt{IF}$
$\qquad\quad :: \texttt{atomic}\{(b2 == false \,||\, x == 1) \Rightarrow incrit1 = true\};$
$\qquad\qquad \texttt{atomic}\{incrit1 = false; b1 = false\};$
$\qquad \texttt{FI}$
$\quad \texttt{OD}$
$\}$

Example: ProMeLa model of Petersons algorithm

```
bool b1, b2;
int x;
bool incrit1, incrit2;

active proctype p1(){
  DO
    :: atomic{b1 = true; x = 2};
    IF
      :: atomic{(b2 == false || x == 1) ⇒ incrit1 = true};
        atomic{incrit1 = false; b1 = false};
    FI
  OD
}

active proctype p1(){ ... symmetric ... }
```

Outline

The model checker SPIN

will be part of the lecture in January 2013

E.W. Dijkstra.
A Discipline of Programming.
Prentice-Hall, 1976.

G. Holzmann.
The SPIN Model Checker, Primer and Reference Manual.
Addison Wesley, 2003.