The SPIN Model Checker

Metodi di Verifica del Software

Andrea Corradini – GianLuigi Ferrari

Lezione 3

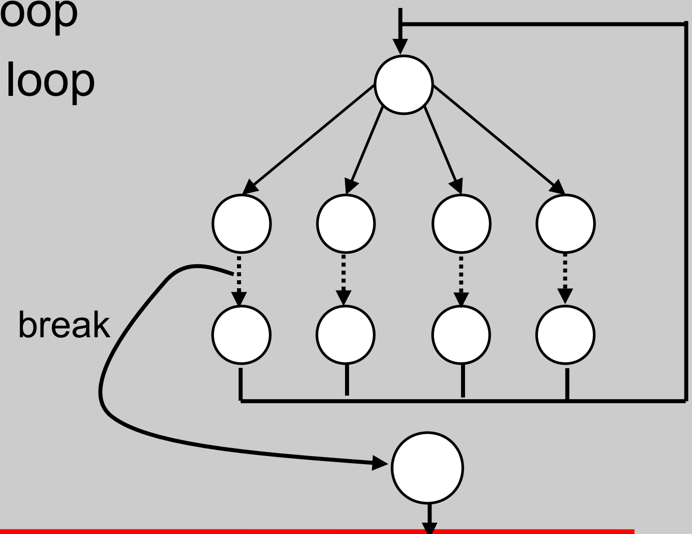
2011

Slides per gentile concessione di Gerard J. Holzmann

the do-statement

```
do
:: guard1 -> stmnt1.1; stmnt1.2; stmnt1.3; …
:: guard2 -> stmnt2.1; stmnt2.2; stmnt2.3; …
:: …
:: guardn -> stmntn.1; stmntn.2; stmntn.3; …
od
```

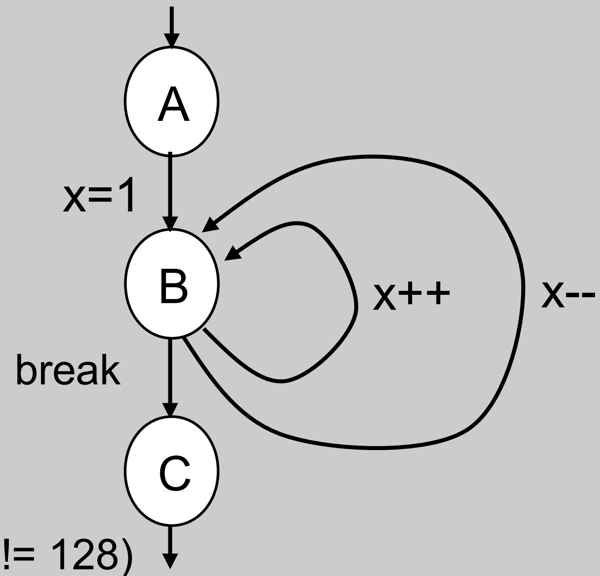
- any type of basic or compound statement can be used as a *guard*
- a do-statement is an **if** statement caught in a cycle
- only a *break* or a *goto* can exit from a do-loop
- a *break* transfers control to the end of the loop



do-statement
underlying automaton

```
byte x;

A:   x = 1;
B:   do
       :: x++
       :: x--
       :: break
       od;
C:   assert(x != 128)
```

Q1: how many process states do you think this model defines?



Q2: can the assertion be violated?

Q3: is the x-- statement needed?

the guards (and statements in general) define *state transitions (state transfomers)* and not *states*

exploiting executability rules

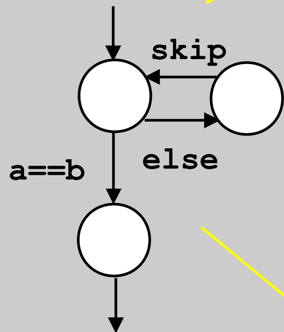
wait for (a==b) to hold

```
do
:: (a == b) -> break
:: else -> skip
od
```

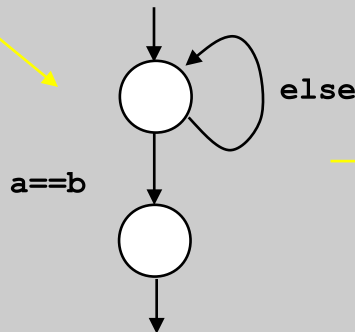
```
L:        if
          :: (a==b) -> skip
          :: else -> goto L
          fi
```

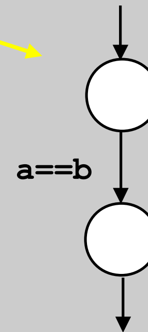
these two constructs are equivalent to a single expression statement

the skip is not needed here and can introduce an unnecessary control state



(a == b)

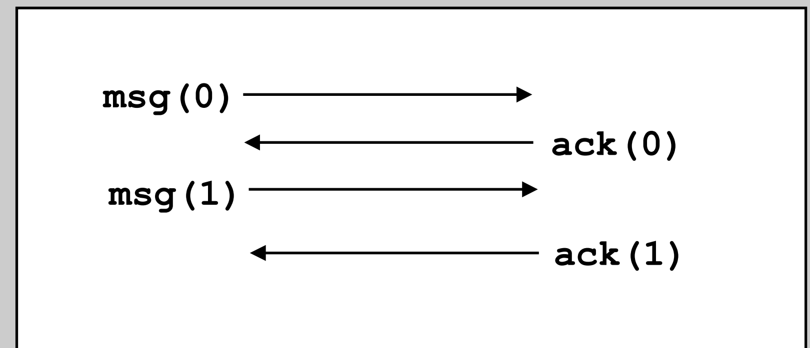




note that 'break', like 'goto', is not a basic statement but a control-flow specifier

example: the alternating bit protocol
(Bartlett et al, 1969)

- two processes, a sender and a receiver
- to every message, the sender adds a sequence number *bit*
- the receiver acknowledges each message by returning the received bit
- if the sender is sure that the receiver has correctly received the previous message, it sends a new message and it alternates the accompanying bit
- if the bit value doesn’t change, the receiver concludes that a message is being repeated

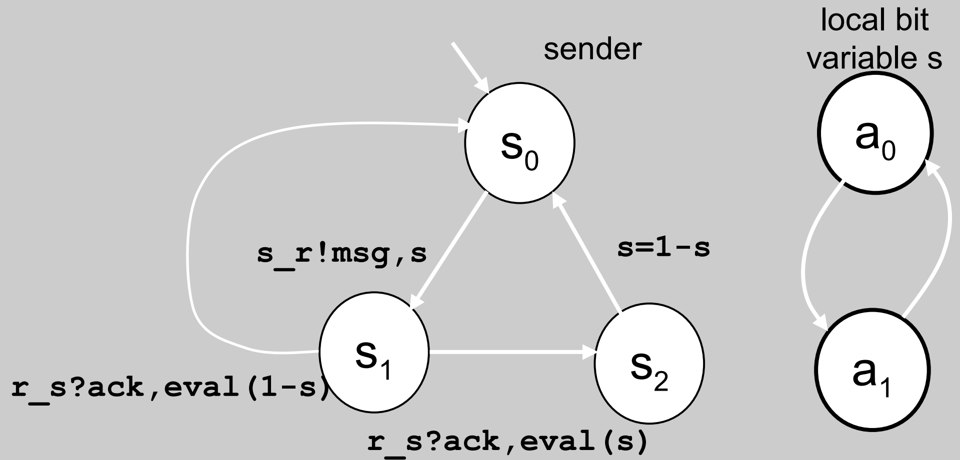


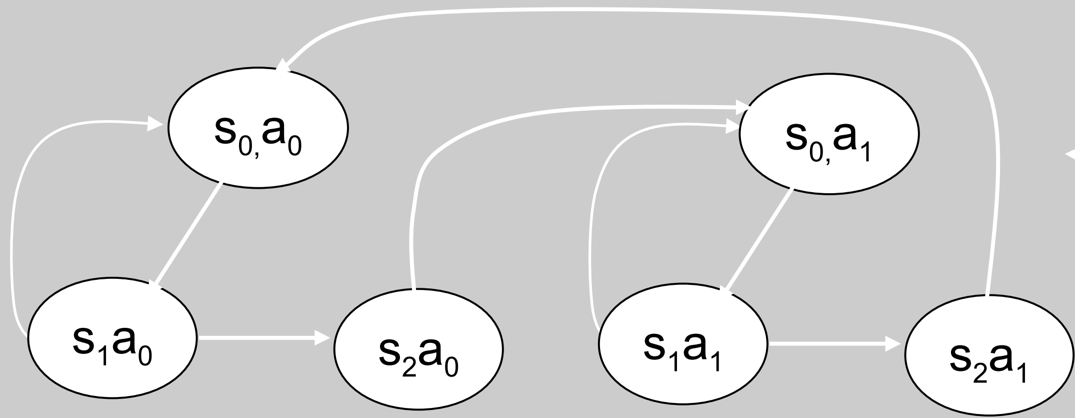
basic Promela model

```
mtype = { msg, ack };
chan s_r = [2] of { mtype, bit };
chan r_s = [2] of { mtype, bit };

active proctype sender()
{    bit seqno;
     do
     :: s_r!msg,seqno ->
         if
         :: r_s?ack,eval(seqno) ->
              seqno = 1 – seqno /* fetch new msg */
         :: r_s?ack,eval(1-seqno)
         fi
     od
}

active proctype receiver()
{    bit expect, seqno;
     do
     :: s_r?msg,seqno ->
         r_s!ack,seqno;
         if
         :: seqno == expect   /* store msg */
         :: else           /* ignore    */
         fi
     od
}
```

the automata view

```
active proctype sender()
{    bit s;
     do
     :: s_r!msg,s ->
          if
          :: r_s?ack,eval(s) ->
               s = 1 - s
          :: r_s?ack,eval(1-s)
          fi
     od
}
```





sender and s together...
(a pure state automaton)

a simulation run

```
$ spin -u20 -c abp        # first 20 steps only
proc 0 = sender
proc 1 = receiver
q\p   0   1
  1   s_r!msg,0
  1   .   s_r?msg,0
  2   .   r_s!ack,0
  2   r_s?ack,0
  1   s_r!msg,1
  1   .   s_r?msg,1
  2   .   r_s!ack,1
  2   r_s?ack,1
-------------
depth-limit (-u20 steps) reached
-------------
final state:
-------------
#processes: 2
                queue 1 (s_r):
                queue 2 (r_s):
 20:    proc  1 (receiver) line  18 "abp" (state 7)
 20:    proc  0 (sender) line   6 "abp" (state 7)
2 processes created
```

the default verification

```
$ spin -a abp.pml
$ gcc –o pan pan.c
$ ./pan
(Spin Version 4.1.0 -- 19 November 2003)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +

State-vector 60 byte, depth reached 11, errors: 0
      12 states, stored
       2 states, matched
      14 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.573   memory usage (Mbyte)

unreached in proctype sender
        line 11, state 5, "-end-"
        (1 of 5 states)
unreached in proctype receiver
        line 19, state 5, "-end-"
        (1 of 5 states)
```

how was it checked?

which properties?

no errors…

amount of work done
(computation of a p.o. reduction of the global state space)

mem. resources used

unreachable code detected
(the processes do no terminate)

the function eval()

ch!msg(12)
ch?msg(eval (x))

maps the current value of x to a constant
to serve as a constraint on the receive statement

receive statement is executable
if the variable x equals 12

```
chan q = [1] of { byte, byte };

x = 12;
q!5(12);        # same as writing: q!5,12
q?x(eval(x))    # same as writing: q?x,eval(x)
```

Q: is this receive statement executable?

modelling message loss

```
mtype = { msg, ack };
chan s_c = [2] of { mtype, bit };
chan c_r = [2] of { mtype, bit };
chan c_s = [2] of { mtype, bit };
chan r_c = [2] of { mtype, bit };

active proctype sender()
{	bit seqno;
	do
	:: s_c!msg,seqno ->
		if
		:: c_s?ack,eval(seqno) ->
			seqno = 1 - seqno /* fetch new msg */
		:: c_s?ack,eval(1-seqno)
		fi
	od
}
active proctype channel()
{	mtype m; bit s;
	do
	:: s_c?m,s -> c_r!m,s	/* faithful transmission */
	:: s_c?m,s		/* to model message loss */
	:: r_c?m,s -> c_s!m,s	/* return channel error-free */
	od
}
active proctype receiver()
{	bit expect, seqno;
	do
	:: c_r?msg,seqno ->
		r_c!ack,seqno;
		if
		:: seqno == expect	/* store msg */
		:: else		/* ignore	*/
		fi
	od
}
```

viewing the automata with xspin

SPIN CONTROL 4.1.1 -- 2 January 2004 -- File: abp2.pml

File.. Edit.. Run.. Help SPIN DESIGN VERIFICATION Line#: 38 Find:

```
mtype = { msg, ack };
chan to_sndr = [1] of { mtype, bit };
chan to_rcvr = [1] of { mtype, bit };
chan from_sndr = [1] of { mtype, bit };
chan from_rcvr = [1] of { mtype, bit };

active proctype sender()
{       bit a;
        do
        :: from_sndr!msg,a;
                if
                :: to_sndr?ack,eval(a);
                        a = 1 - a
                :: timeout /* retransmission */
                fi
        od
}

active proctype channel()
{       mtype m; bit s;
        do
        :: from_sndr?m,s -> to_rcvr!m,s
        :: from_sndr?m,s     /* message loss */
        :: from_rcvr?m,s ->
```

```
gcc -w -o pan -D_POSIX_SOURCE pan.c
<compilation complete>
pan -d      # compute fsm
<dot graph layout...>
<done>
```

Run..

- Run Syntax Check
- Run Slicing Algorithm
- Set Simulation Parameters..
- (Re)Run Simulation
- Set Verification Parameters..
- (Re)Run Verification
- LTL Property manager..
- View Spin Automaton for each Proctype..

FSM channel

line 25
to_sndr!m,s
[(1,2)]
from_rcvr?m,s
line 21
from_sndr?m,s
from_sndr?m,s
[(3,3)]
to_rcvr!m,s
[(2,2)]
line 22

No Labels | Smaller | Larger | Save in: channel.ps | Close

atomic sequences

suppressing process interleavings

```
atomic { guard -> stmnt1; stmnt2; ... stmntn }
```

- executable if the guard statement is executable
- any statement can serve as the guard statement
- executes all statements in the sequence *without* interleaving with statements in other processes
- if any statement other than the guard blocks, atomicity is lost atomicity can be regained when the statement becomes executable
- example: mutual exclusion with an indivisible test&set:

```
active [10] proctype P()
{ atomic { (busy == false) -> busy = true };
  mutex++;

  assert(mutex==1);

  mutex--;
  busy = false;
}
```

d_step sequences

more restrictive and more efficient than atomic sequences

```
d_step { guard -> stmnt1; stmnt2; ... stmntn }
```

- like an atomic, but *must be deterministic* and *may not block* anywhere inside the sequence
- especially useful to perform intermediate computations with a deterministic result, in a single indivisible step

```
d_step { /* reset array elements to 0 */
    i = 0;
    do
    :: i < N -> x[i] = 0; i++
    :: else -> break
    od;
    i = 0
   }
```

- atomic and d_step sequences are often used as a model reduction method, to lower complexity of large models (improving tractability)

d_steps and gotos

- goto-jumps into and out of atomic sequences are allowed
  - atomicity is preserved only if the jump starts inside on atomic sequence and ends inside another atomic sequence, and the target statement is executable
- goto-jumps into and out of d_step sequences are forbidden

```
d_step {
      i = 0;
      do
      :: i < N -> x[i] = 0; i++
      :: else -> break
      od
};
x[0] = x[1] + x[2];
```

```
this is a jump out
of the d_step sequence
and it will trigger an
error from Spin
```

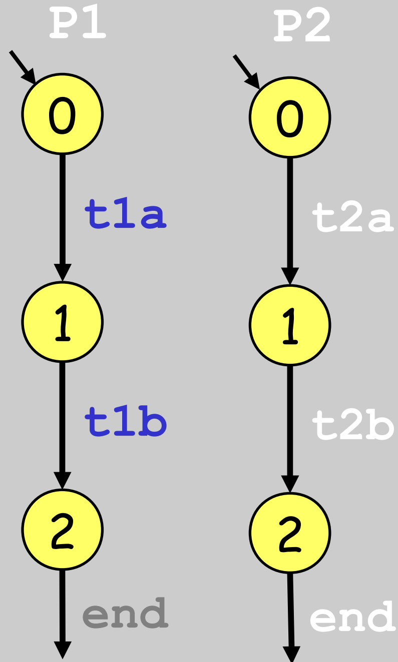
```
the problem is prevented in this
case by adding a "; skip" after the
od keyword - there's no runtime penalty for
this, since it's inside the d_step
```

atomic and d_step

- both sequences are executable only when the *first* (guard) statement is executable
  - **atomic**: if any other statement blocks, atomicity is lost at that point; it can be regained once the statement becomes executable later
  - **d_step**: it is an *error* if any statement other than the guard statement blocks
- other differences:
  - **d_step**: the entire sequence is executed as *one* single transition
  - **atomic**: the sequence is executed step-by-step, but without interleaving; non-deterministic choices inside an atomic sequence are allowed
- caution:
  - infinite loops inside atomic or d_step sequences *are not* detected
  - the execution of this type of sequence models an indivisible step, which means that it cannot be infinite

```
active proctype P1() { t1a; t1b }
active proctype P2() { t2a; t2b }
```

execution
without atomics or d_steps



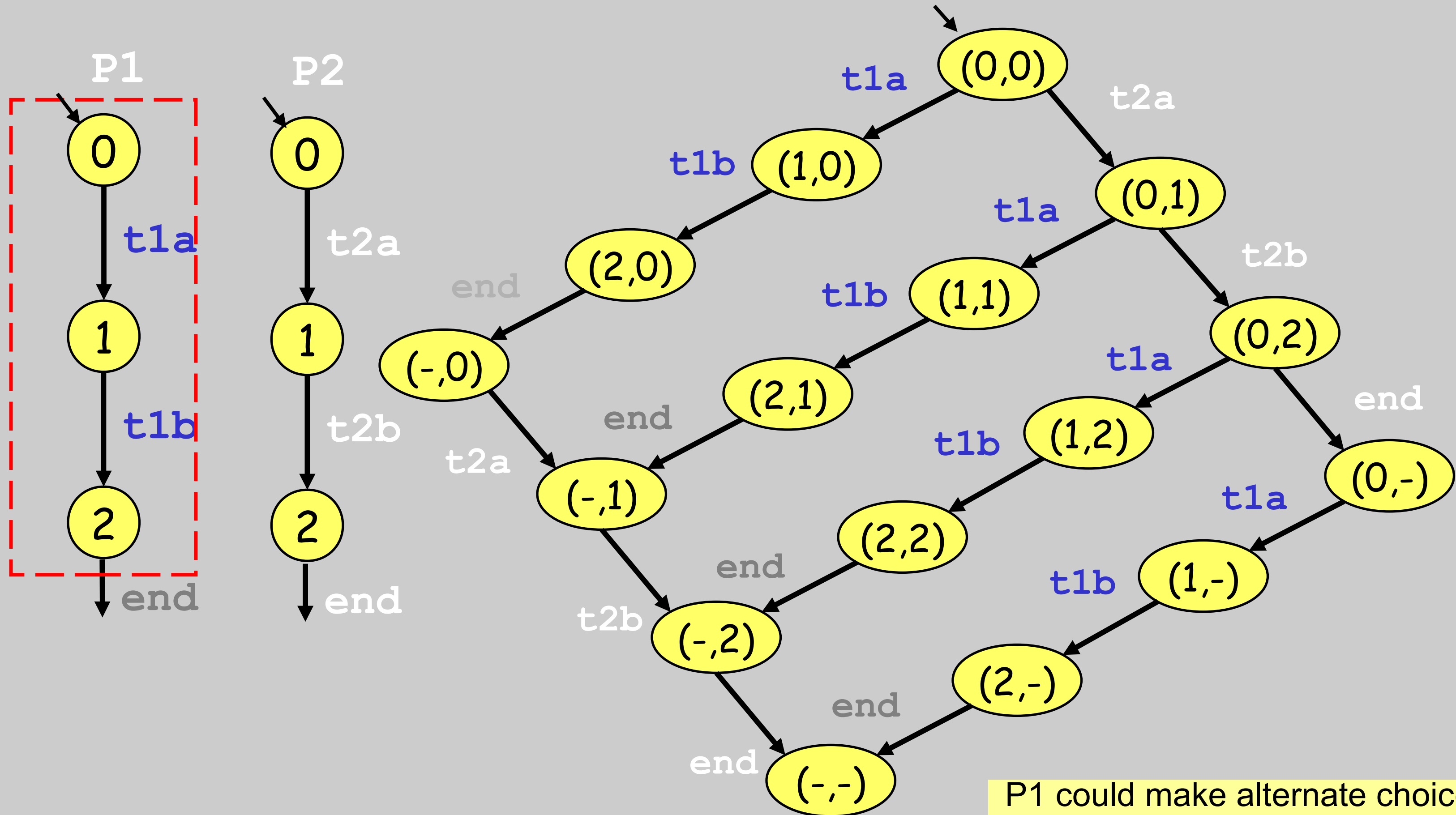
(0,0)
t1a t2a
(1,0) (0,1)
t1b t2a t1a t2b
(2,0) (1,1) (0,2)
end t2a t1b t2b t1a end
(-,0) (2,1) (1,2) (0,-)
t2a end t2b t1b end t1a
(-,1) (2,2) (1,-)
t2b end end t1b
(-,2) (2,-)
end end
(-,-)

execution with
full interleaving

```
active proctype P1() { atomic { t1a; t1b } }
active proctype P2() { t2a; t2b }
```

execution with one atomic sequence

P2 can be interrupted, but not P1

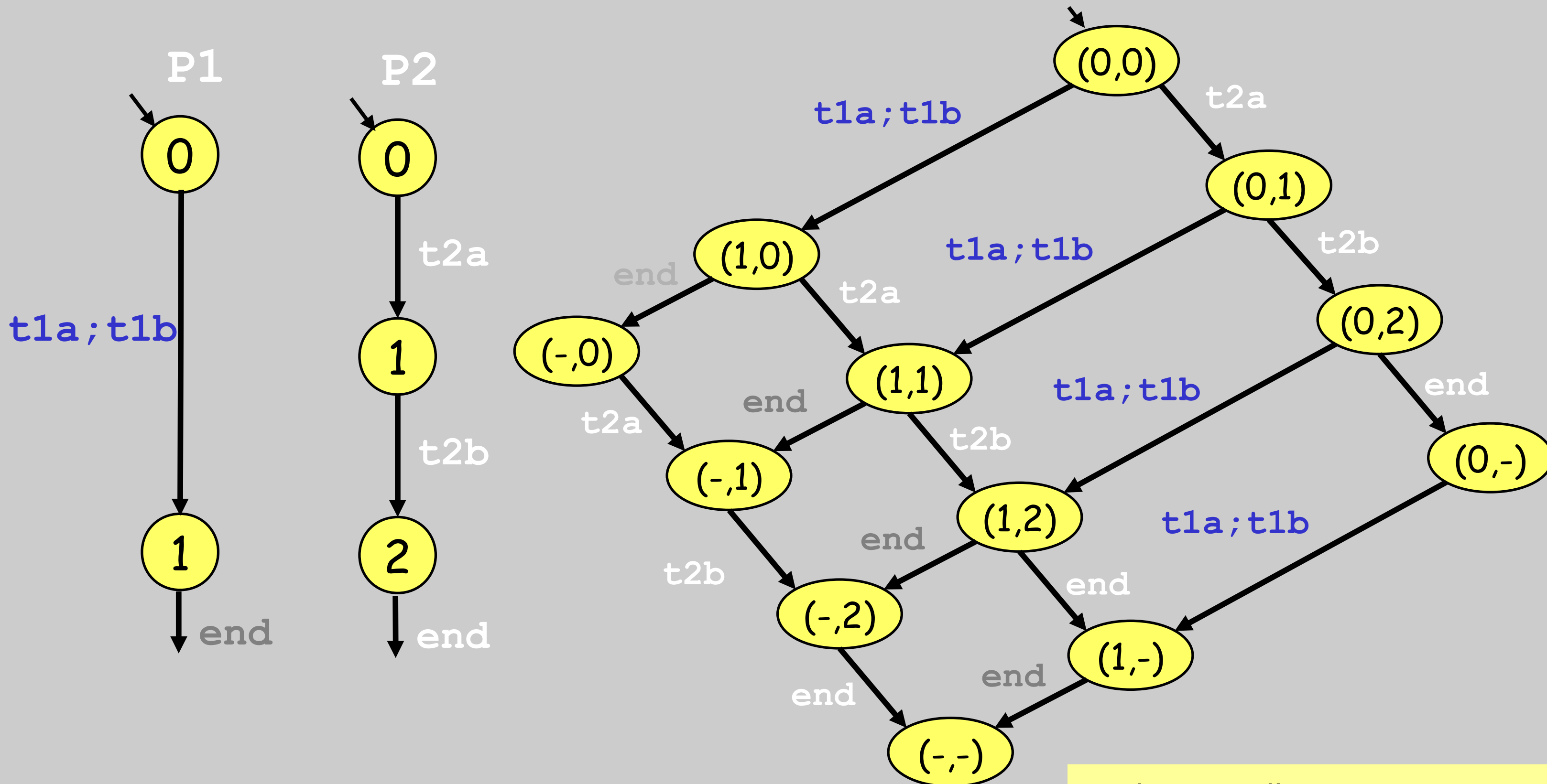


P1 could make alternate choices at the intermediate states (e.g., in if or do-statements)

```
active proctype P1() { d_step {t1a; t1b} }
active proctype P2() { t2a; t2b }
```

execution with a d_step sequence

P1 now has only one transition…

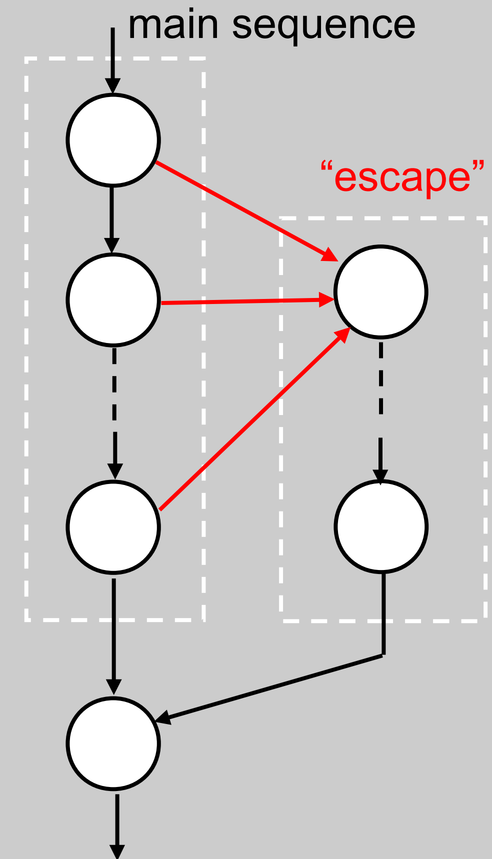


no intermediate states are created: faster, smaller graph, but no non-determinism possible inside d_step sequence itself

the last control construct: unless sequences

(cf. book, fig. 3.1, p. 63)

```
active proctype pots()
{       chan who;
idle:   line?offhook,who ->
        {       who!dialtone;
                who?number;
                if
                :: who!busy
                :: who!ringing;
                        who!connected;
                        who!hungup;
                fi;
                goto wait
        } unless {
                if
                :: who?hangup -> goto idle
                :: timeout -> goto wait
                fi
        }
wait:   who?hangup;
        goto idle
}
```



unless sequences

{ guard1; <*stmnts1*> } unless { *guard2*;<*stmnts2*> }

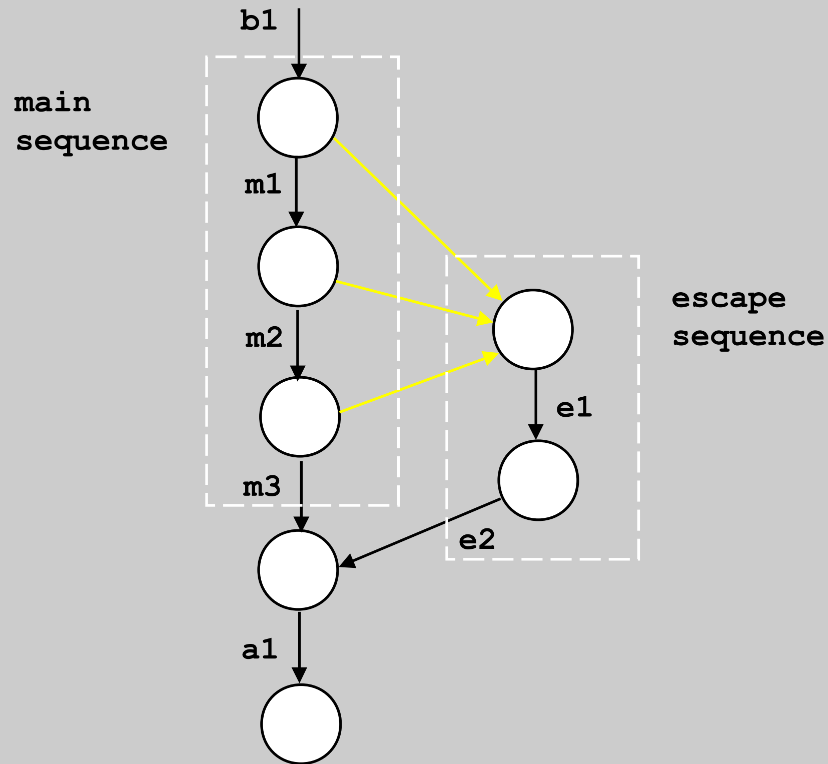
- the unless statement as a whole is executable if either the guard statement of the main sequence is executable (guard1), or the guard statement of the escape sequence is executable (guard2)
- statements in the main sequence continue to be executed until the guard statement of the escape sequence becomes executable, *if so*
- if and only if this happens, execution of the main sequence stops and execution proceeds with the escape sequence, which is then executed to completion (there is *no* return to the main sequence)
- resembles exception handling in languages like Java

```
proctype cpu()
{
  { ... /* normal flow */
    ...
  } unless { port?INTERRUPT ->
    ... /* interrupt handling */
  }
}
```

nesting

- unless structures may be nested arbitrarily deeply
- escape clauses can be used to define levels of priority of execution in this way
- the order of evaluation of escape clauses by default is *inside out*, but can be reversed with Spin option –J (to match the evaluation order for nested exception handling in Java)

automaton view




```
b1;
{ m1 -> m2; m3 }
   unless
{ e1 -> e2 };
a1;
...
```

the predefined variable _

- the *write-only* scratch variable _
- e.g., flushing the contents of a buffer with two message fields:

```
d_step {
	do
	:: atomic { nempty(q) -> q?_,_ }
	:: else -> break
	od;
	skip
}
```

- note that normally *all* data objects store 'state' information
  - if two global states differ only in the value of a single local variable in one of the active processes, then it's still a different global state
  - the write-only scratch variable _ can be useful to avoid storing redundant data that may affect the state space size
  - (you can achieve the same effect on other variables by prefixing their declaration with the keyword **hidden**)

other Promela language features

- conditional expressions
  - (i -> t : e) works precisely like the expression (i?t:e) in C

  if i is true then the result of the conditional expression is the value of t, if false the result of the expression is the value of e

  can be used to define *conditional* rendezvous:

```
chan q[3] = [0] of { mtype };
sender:    q[(P->1:2)]!msg
receiver:  q[(Q->1:0]?msg
```

```
rendezvous is now only possible
when P is true at the sender
and  Q is true at the receiver
```

- the declaration prefixes hidden and show

  hidden byte x;   /* x declared not to hold state information */

  show byte x;   /* x can be tracked in the Xspin GUI */

- embedded c_code primitives

  we'll return to this when discussing advanced model checking techniques

defining correctness properties

- the basic building blocks of a Spin model
  - asynchronous process behavior
  - variables, data types
  - message channels
  - logical correctness properties
    - assertions
    - end-state, progress-state, and acceptance state labels
    - never claims
    - temporal logic formulae
    - default properties:
      - absence of system deadlock
      - absence of dead code (unreachable code)

the properties define the real objective of a verification