Model Checking with SPIN

A Bit More about SPIN

Wishnu Prasetya

wishnu@cs.uu.nl

www.cs.uu.nl/docs/vakken/pv

Content

- SPIN internal data structures
- SPIN result report
- Writing LTL formulas
- Containing state explosion
- Example

Acknowledgement: some slides are taken and adapted from Theo Ruys's SPIN Tutorials.

Data structures involved in SPIN DFS

- Representation of a state.

- Stack for the DFS
  - To remember where to backtrack in DFS
  - It corresponds to the current “execution prefix” that is being inspected → used for reporting.

- Something to hold the set of visited states = “state space”.

State

- Each (global) state of a system is a “product” of the states of its processes.
- E.g. Suppose we have:
  - One global var byte *x*
  - Process *P* with byte *y*
  - Process *Q* with byte z
- Each system state should describe:
  - all these variables
  - Program counter of each process
  - Other SPIN predefined vars

- Represent each global state as a tuple … this tuple can be quite big.

The stack, optimization

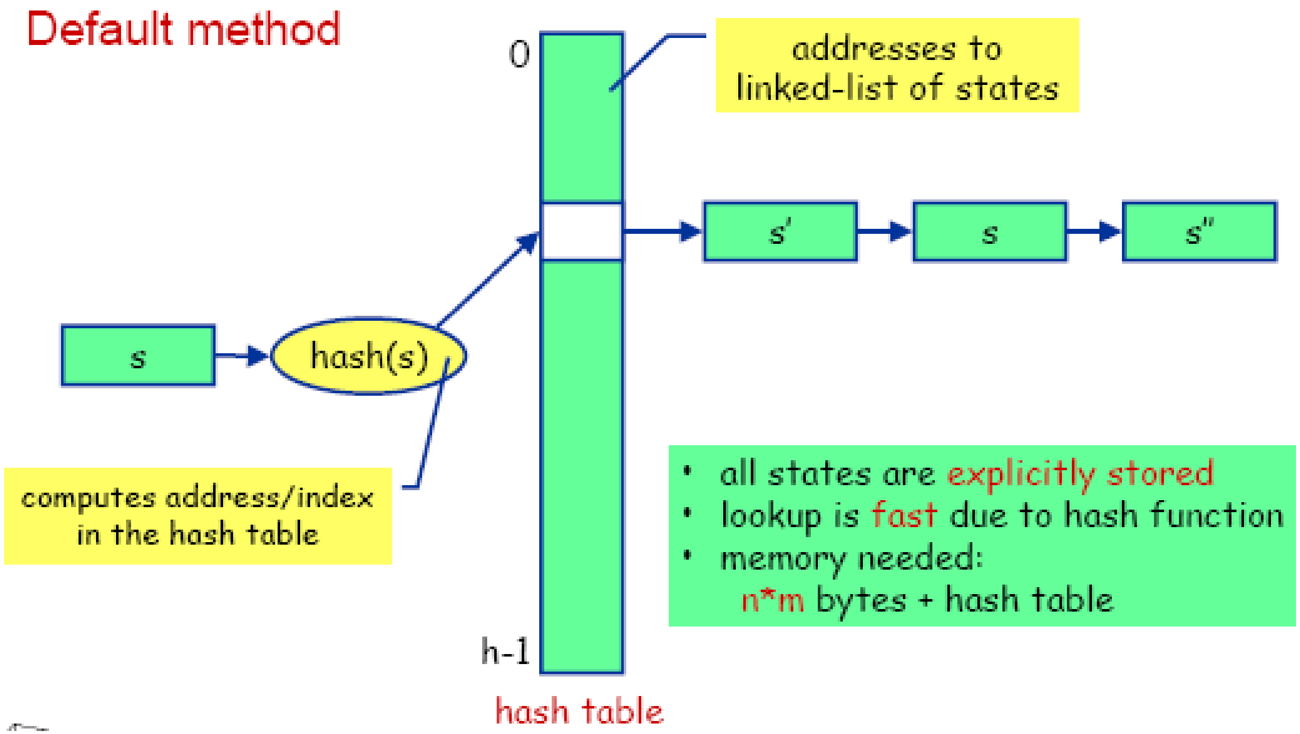
- To save space SPIN does not literally keep a stack of states (large) → most of the time, states can be replaced by the ID of the actions that produce them (smaller)

- But, when you backtrack you need to know the state!

  SPIN calculated the reverse/undo of every action. So, knowing the current state is sufficient.

State-space is stored with a hash table

The list of "visited states" is maintained by a Hash-table. So matching if a state occurring in the table is fast!



Optimization: bit state hashing ☺

Verifier's output

```
assertion violated  !((crit[0]&&crit[1])) (at depth 5)     // computation depth
...
Warning: Search not completed

Full statespace search for:
            ...
             never-claim            - (not selected)
             assertion violations   +
             invalid endstates      +

State-vector 20 byte, depth reached 7, errors: 1          // max. stack depth
     24 states, stored                                    // states stored in hash table
     17 states, matched                                   // states found re-revisited
     41 transitions (= stored+matched)

hash conflicts: 0 (resolved)
(max size 2^19 states)

2.542       memory usage (Mbyte)
```

Watch out for state explosion!

```
int x,y,z ;

P  { do :: x++ od }
Q  { do :: y++ od }
R  { do ::  x/=y → z++ od }
```

- Size of each state: > 12 bytes
- Number of possible states $\approx (2^{32})^3 = 2^{96}$
- Using byte (instead of int) brings this down to 50 MB
- Focus on the critical aspect of your model; abstract from data when possible.

Another source of explosion : concurrency

imposing a coarser grain atomicity

atomic { guard → stmt_1; ... ; stmt_n }

active proctype P { x++ ; (y>0) ; y-- ; x=y }

put in *atomic* ?

- more abstract, less error prone, but less parallelism
- executable if the guard statement is executable
- none of stmt-i should be blocking; or rather : if any of then blocks, atomicity is lost

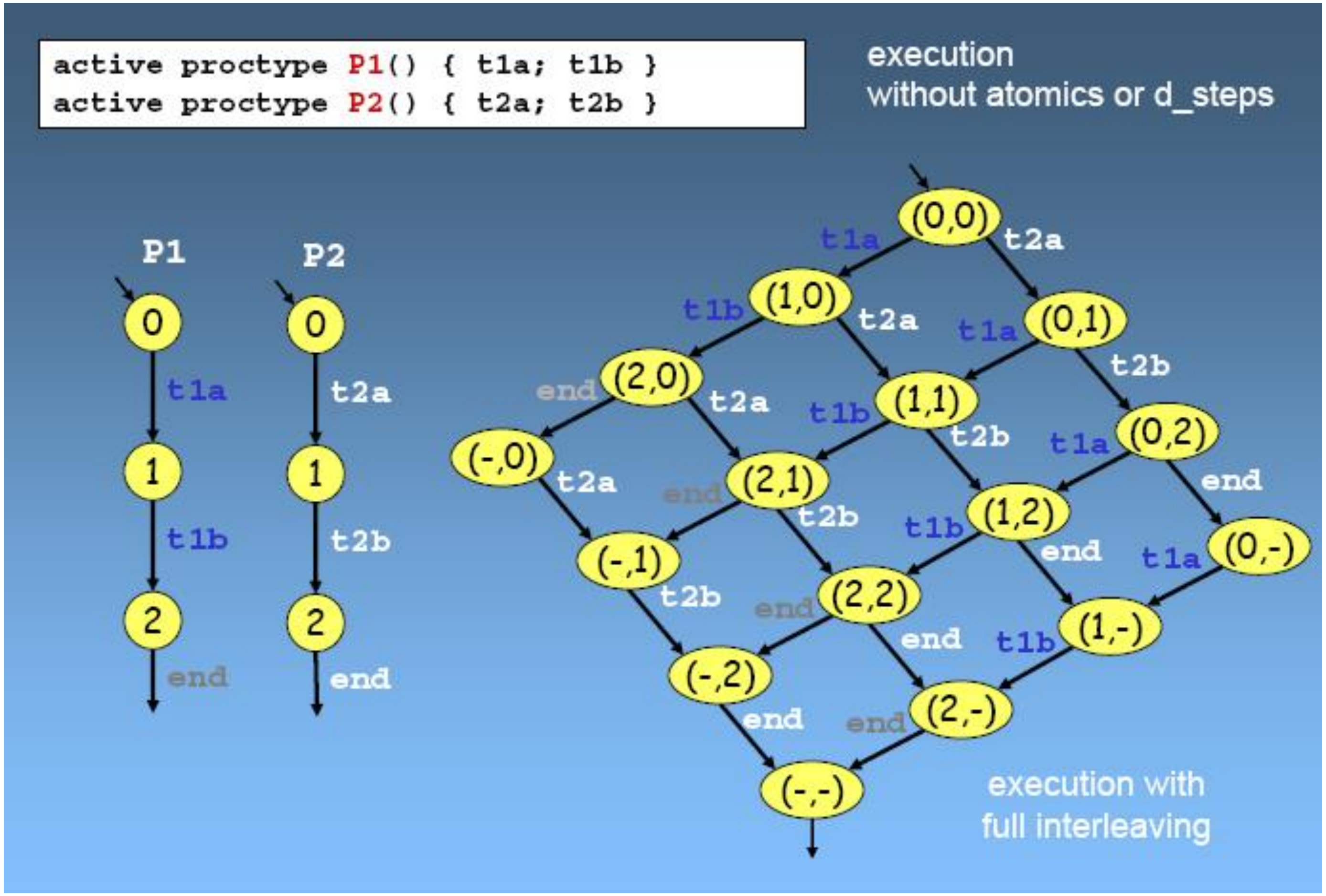
d_step sequences

d_step { guard → stmt_1; ... ; stmt_n }

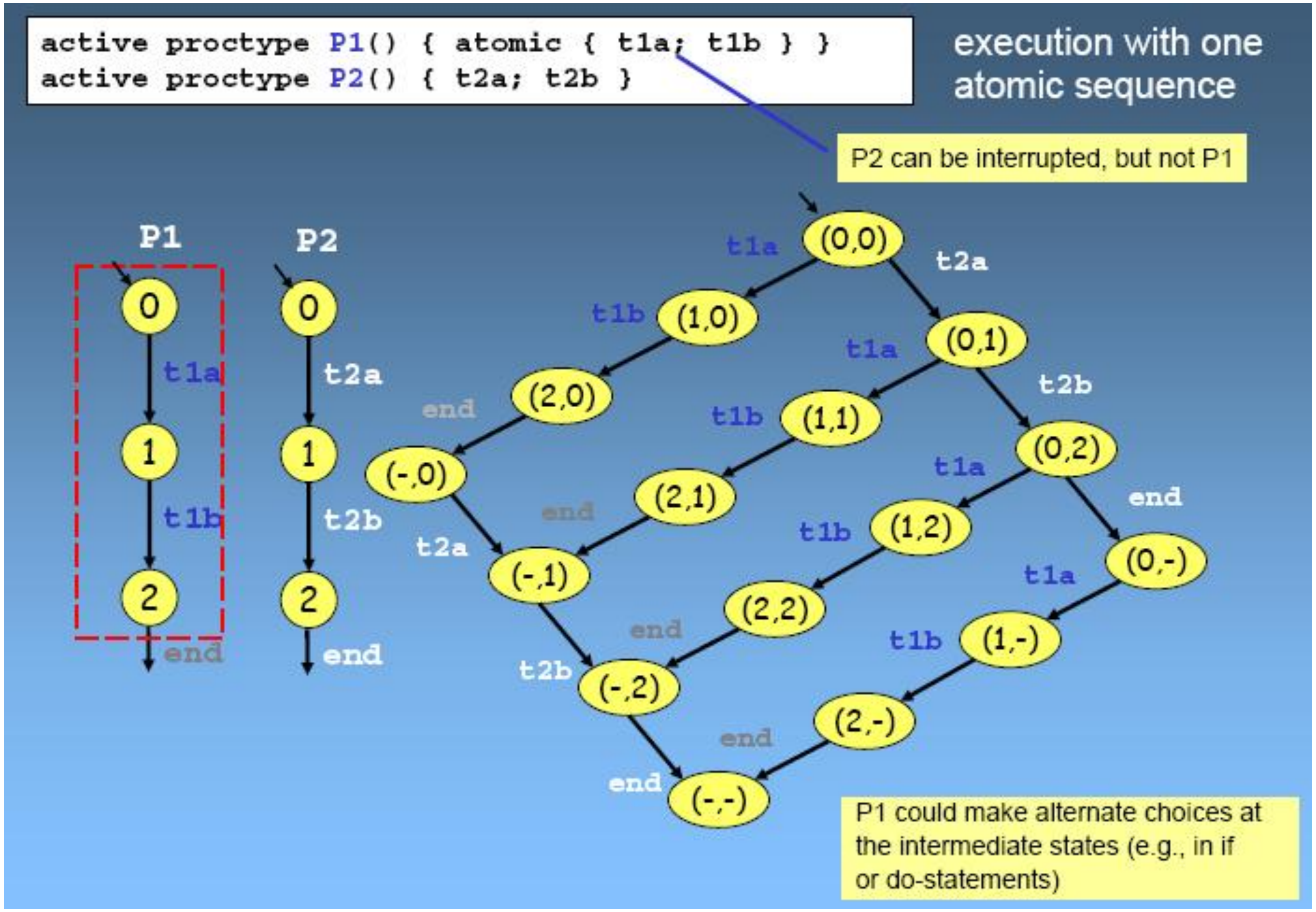
```
d_step {   /* reset array elements to 0 */
    i = 0;
    do
    :: i < N -> x[i] = 0; i++
    :: else -> break
    od;
    i = 0
}
```

- like an atomic, but *must be deterministic* and *may not block* anywhere
- atomic and d_step sequences are often used as a model reduction method, to lower complexity of large models (improving tractability)
- No jump into the middle of a d_step

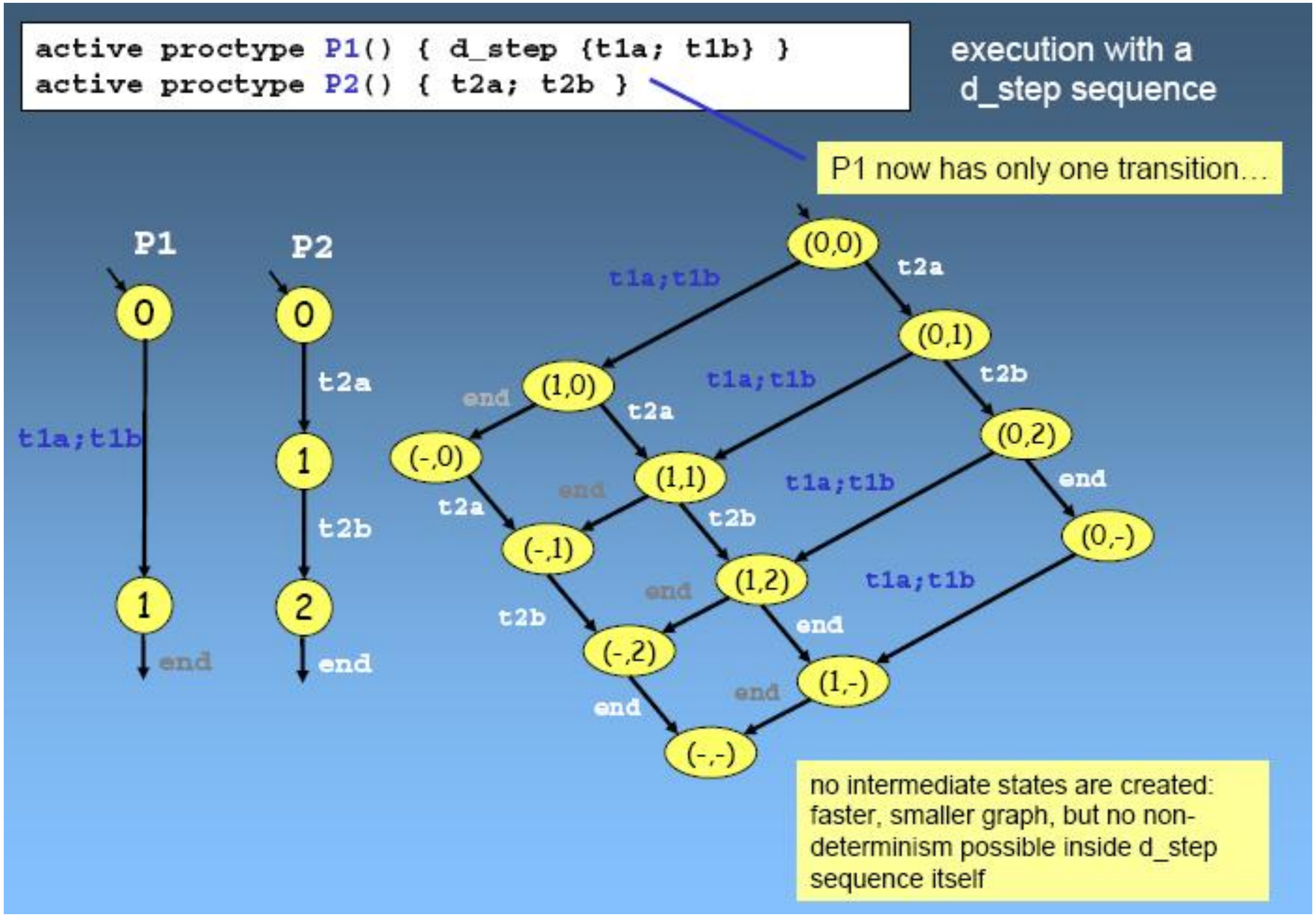
execution without atomics or d_steps



execution with one atomic sequence



execution with a d_step sequence



atomic vs d_step

- d_step:
  - executed as *one* block
  - deterministic
  - blocking or non-termination would hang you ☺
  - the verifier does not peek into intermediate states

- atomic:
  - translated to a series of actions
  - executed step-by-step, but without interleaving
  - it can make non-deterministic choices
  - verifies sees intermediate states

Partial Order Reduction

- The validity of a property ϕ is often insensitive to the order in which 'independent' actions are interleaved.

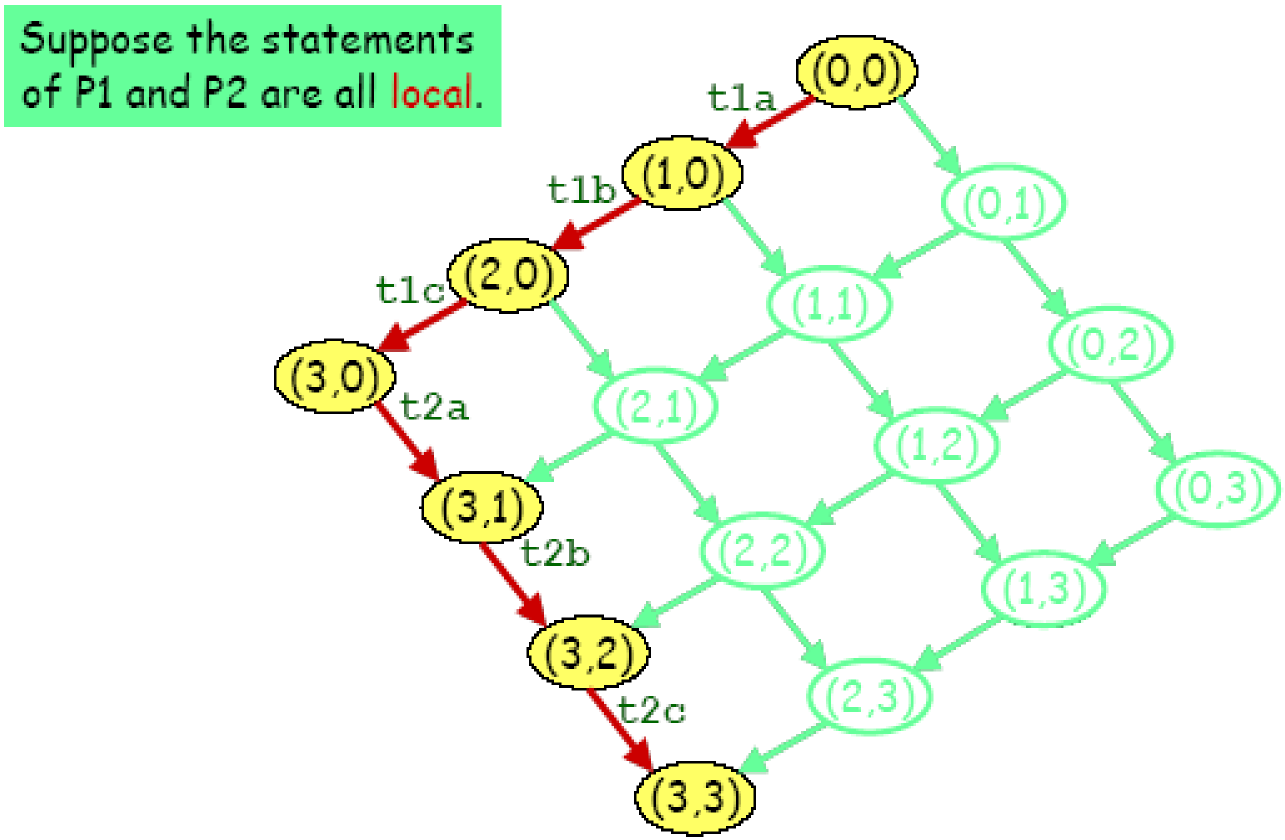
  e.g. stutter invariant φ (does not contain X) that only refers to global variables, is insensitive to the relative order of actions that only access local variables.

- Idea: if in some global state, a process P can execute only actions updating local variables, always do these actions first (so they will not be interleaved!)

- We can also do the same with actions that :
  - receive from a queue, from which no other process receives
  - send to a queue, to which no other process sends

Reduction Algorithms

- Partial Order Reduction



Results on Partial Order Reduction

Protocol	Algorithm	States	Transitions	Time(sec.)	Memory (Mb)
Best-Case	Non-Reduced	100,001	450,002	13.2	4.3
	Static Reduction	47	47	(<0.1)	1.0
	Dynamic Reduction	47	47	0.1	1.4
Worst-Case	Non-Reduced	100,001	450,002	14.5	5.0
	Static Reduction	100,001	450,002	16.7	5.1
	Dynamic Reduction	100,001	450,002	84.5	5.3
Tpc	Non-Reduced	3,918,286	11,762,426	630.6	268.4
	Static Reduction	391,534	466,753	30.6	26.2
	Dynamic Reduction	267,204	295,395	131.4	18.9
Snoopy	Non-Reduced	91,920	305,460	14.4	11.5
	Static Reduction	16,279	23,532	1.7	3.2
	Dynamic Reduction	7,158	8,459	6.8	2.6
Pftp	Non-Reduced	417,321	1,244,865	73.2	62.3
	Static Reduction	53,244	67,901	6.8	9.3
	Dynamic Reduction	125,718	163,459	105.5	20.6
Leader	Non-Reduced	45,885	185,032	8.1	9.6
	Static Reduction	79	79	0.1	1.1
	Dynamic Reduction	79	79	0.2	1.4

This result is from Holzmann & Peled in 1994, on a Sparc-10 workstation with 128Mbyte of RAM. (about 40 mhz; so 1 mips??)

Specifying LTL properties

- (Check out the Manual)

```
#define PinCritical    crit[1]
#define QinCritical    crit[2]

[]!(PinCritical &&  QinCritical)
```

- SPIN then generates the Buchi automaton for this LTL formula; called “Never Claim” in SPIN.

Example of a Never Claim

To verify: <>[] *p*

SPIN generates this never-claim / Buchi of **[]<>¬*p***

```
never {
   init:
      if
      :: ¬p   → goto   accept
      :: else → goto   init
      fi;
   accept:
      skip; goto  init ;
}
```

Neverclaim

- From SPIN perspective, a neverclaim is just another process, but it is executed in "*lock-step*" :
  - innitially, it is executed first, before the system does it first step
  - then, after each step of the system, we execute a step from the neverclaim.
- Is used to express properties
  - E.g. by writing assertions inside a neverclaim
  - Or by using acceptance states
  - If an NC reaches its final state (its final "}") → violation → used to match against finite executions.

You can also manually write your custom NC …

```
never {
  do
  :: assert (b) ; assert (!b)
  od
}
```

Expressing the value of b should be alternating.

Note: in LTL this can be expressed as:

$$[]((b \rightarrow \mathbf{X}\neg b) \wedge (\neg b \rightarrow \mathbf{X} b))$$

```
never {
   accept : do :: (x==0) ; (x==1) od
}
```

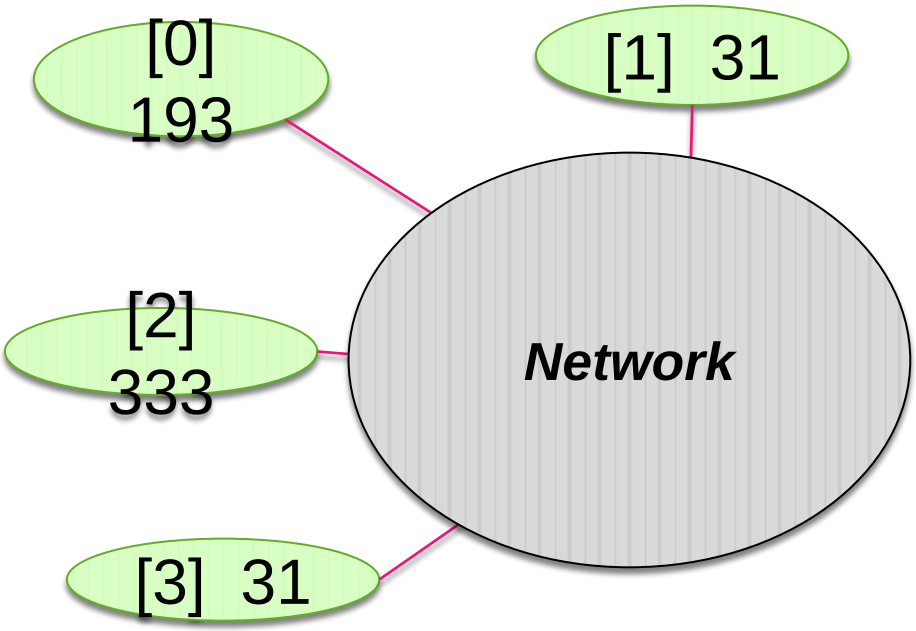
recognize an execution where (x==0)(x==1) holds alternatingly, which would then be considered as error.

You can also manually write your custom NC …

```
never {
   do
   :: (x>0) → skip
   :: else → break
}
```

If x ever becomes 0, then this would be a violation (because the NC then reaches its end-state).

Example: distributed sorting



- Idea:

Let P(i) swap values with P(i+1), if they are in the wrong order.

- Spec:

Eventually the values will be sorted.

SPIN model

```
#define N 5

byte a[N] ;

proctype P(byte i) {
  byte tmp = 0 ;
  do
  :: d_step{ a[i]>a[i+1] ->
           tmp=a[i] ;
           a[i]=a[i+1] ;
           a[i+1]=tmp  ;
           tmp=0 }
  od ;
}
```

(let's just assume locking a[i] and a[i+1] atomically is reasonable.)

```
init {
  byte i ;
  do
  :: i<N ->
       if
       :: a[i]=0
       :: a[i]=1
       ...
       fi      ;  i++
  :: else -> break ;
  od ;
  i=0 ;
  do
  :: i< N - 1 -> run P(i)     ; i++
  :: else     -> run detect() ; break
  od
}
```

Swap values, set tmp back to 0 to save state.

Expressing the spec

Eventually the values will be sorted.

With LTL: $<>[] (\forall i : 0 \leq i < N\text{-}1 : a[i] \leq a[i+1])$

But SPIN does not support quantification in its Expr!

Introduce a global shadow var i, non-deterministically initialized to : $0 \leq i < N\text{-}1$. *Then verify this instead :*

$<>[]\ a[i] \leq a[i+1]$

Detecting "termination"

New spec: we want the processes themselves to know that the goal (to sort values) has been acomplished.

```
proctype detect() {
  byte i ;
  timeout ->
    do
    :: i<N-1 -> assert (a[i]<=a[i+1])
    :: else  -> break
    od
}
```

done = true

Extend P(i), such that when it sees "done" is true, it will terminate.

Unfortunately, not good enough. The above solution uses "timeout" which in SPIN is actually implemented as a mechanism to detect non-progress; in the actual system we now assume not to have this mechanism in the first place, and hence have to implement it ourselves.

Detecting "termination"

Idea: let "detect" keep scanning the array to check if it is sorted.

```
proctype detect() {
  byte i ;
  i=0 ;
  do
  :: i<N-1 -> if
              :: a[i]>a[i+1] ->  i=0
              :: else          ->  i++
              fi
  :: else -> done=true ; break
  od
}
```

Unfortunately, this doesn't work perfectly. Consider this sequence of steps:

[4, 5, 1]

detect 0,1 → ok

swap 1,2

[4, 1, 5]

detect 1,2 → ok

now "detect" concludes termination!

Can you find a solution for this??