Introduction &
LTL Model Checking

Avinash Malik

2022

Introduction

- Who am I?
- Senior Lecturer at the University of Auckland, ECE
- Office: 405.775
- Prefer email contact over visits to office.
- Email: avinash.malik@auckland.ac.nz

Important information for 2022

- Week-9 is systems week, so no lectures.

- Test (50%)
  - Date: October 19/2022
  - Venue: **Online via Canvas**
  - Time: 16:10 – 18:10

Introduction – second part of 705

- Learn two important things:

1. Linear temporal model-checking – i.e., an automated technique to check if software implementation is correct.

2. Constraint programming and proving – an efficient technique to check if software is correct.

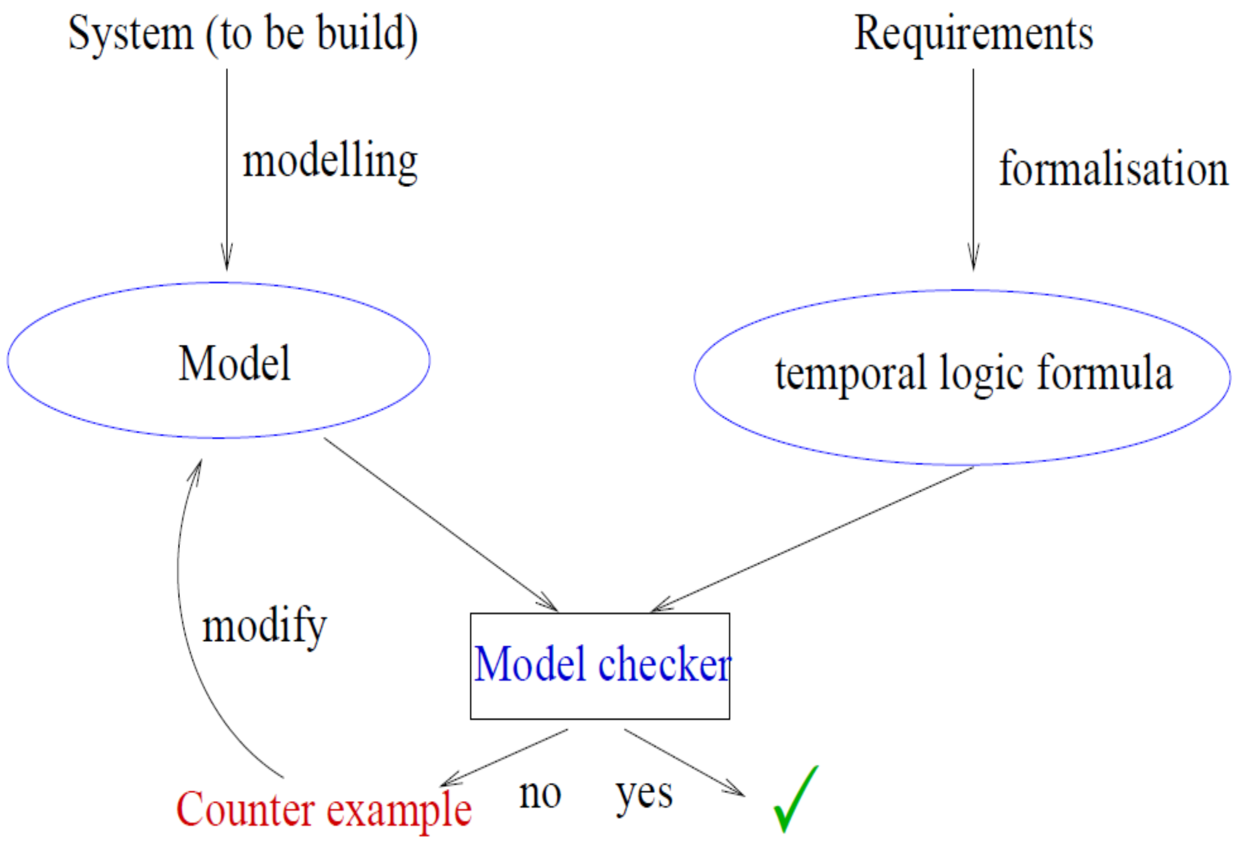
Need for reliable software and hardware

- Suppose you work (or run) a software/hardware company
- Suppose you have sunk 30+ years into a product
  - BMAX 737 – failure
  - Araine 5 – rocket failure
  - Intel floating point bug failure
- It is essential to get software/hardware implementation correct for safety critical systems.

Learning outcomes – Part 1

- Understand what is model-checking
- Understand what is Linear Temporal Logic (LTL)
- Understand the semantics of linear temporal logic
- Understand the Promela programming language for describing concurrent processes.
- Understand SPIN – the LTL model-checker.

The big picture



LTL Model Checking

- **LTL**
  - Subset of CTL* of the form:

    A f

    where f is a path formula
- LTL model checking
  - Model checking of a property expressed as an LTL formula:
  - Given a model M and an initial state s_0:

    $M,s_0 \models A\ f$

LTL Formulas

- Subset of CTL*
  - Distinct from CTL
    - AFG p $\in$ LTL
- Contains a single universal quantifier
  - The path formula f holds for every path
- Commonly:
  - A is omitted
  - G is replaced by □ (box or always)
  - F is replaced by ◊ (diamond or eventually)

Examples of LTL formulas

- Always eventually p:
    - $\Box \Diamond p$
    - AGF p or AG AF p
- Always after p eventually q
    - $\Box (p \rightarrow \Diamond q)$
    - AG (p -> F q) or AG (p -> AF q)
- Fairness
    - $(\Box \Diamond p) \rightarrow \varphi$
    - $A ((GF\ p) \rightarrow \varphi)$

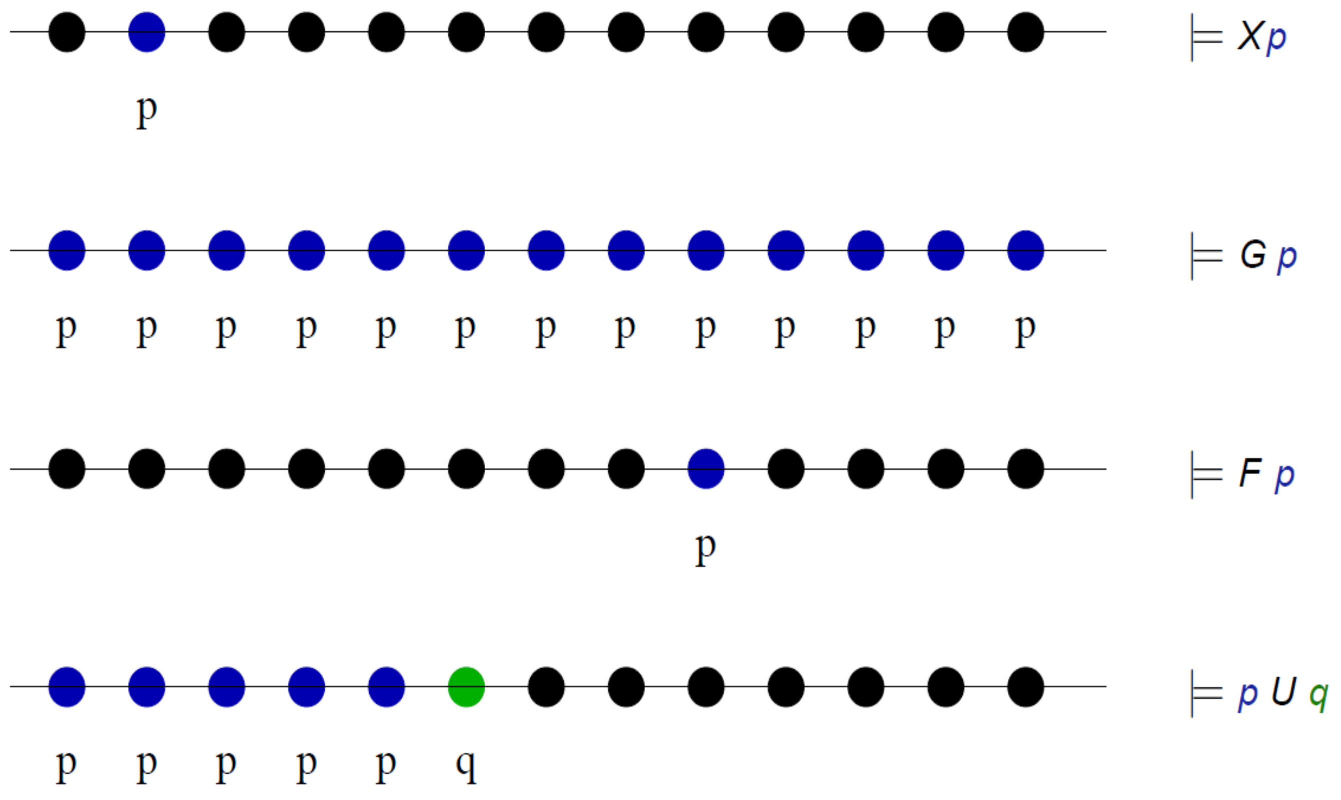
Not a CTL formula

LTL Semantics

Def.. *Let* $\pi = s_0 s_1 s_2 \ldots$ *a path,* φ *an LTL formula.* $\pi \models \varphi$ *is inductively defined as follows.*

- $\pi \models p, p \in AP$ *iff* p *holds in* s_0 *(i.e.* $p \in L(s_0)$*)*
- $\pi \models \neg\varphi$ *iff not* $\pi \models \varphi$
- $\pi \models \varphi \vee \psi$ *iff* $\pi \models \varphi$ *oder* $\pi \models \psi$
- $\pi \models X\, \varphi$ *iff* $\pi^1 \models \varphi$
- $\pi \models G\, \varphi$ *iff* $\forall i \geq 0 : \pi^i \models \varphi$
- $\pi \models F\, \varphi$ *iff* $\exists j \geq 0 : \pi^j \models \varphi$
- $\pi \models \varphi\, U\, \psi$ *iff* $\exists k \geq 0 : \pi^k \models \psi$ *and*
  $\forall j, 0 \leq j < k, \pi^j \models \varphi$.

LTL semantics



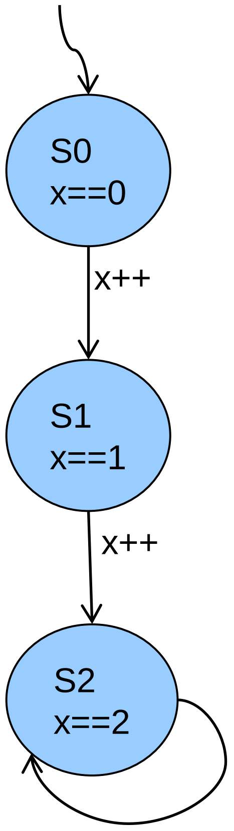
LTL Model Checking

- Given a model M and an LTL formula φ
  - All traces of M must satisfy φ
    - Σ_M is the set of traces of M
    - Σ_φ is the set of traces that satisfy φ
    - $\Sigma_M \subseteq \Sigma_\varphi$
  - If a trace of M does not satisfy φ
    - Counterexample
- Equivalently $\Sigma_M \cap \Sigma_{\neg\varphi} = \varnothing$

An Example

```
int x = 0;
x++;
x++;
```

C-code



Model
'M'

S0,S1,S2,....

Trace

M |= (x==0)?

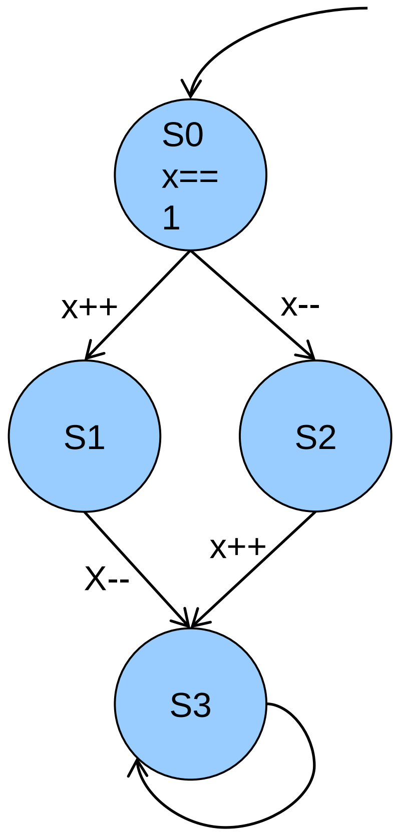
M |= (x==1)?

M |= ◊(x==2)?

M |= □(x==2)?

M |= ◊□(x==2)?

Another Example



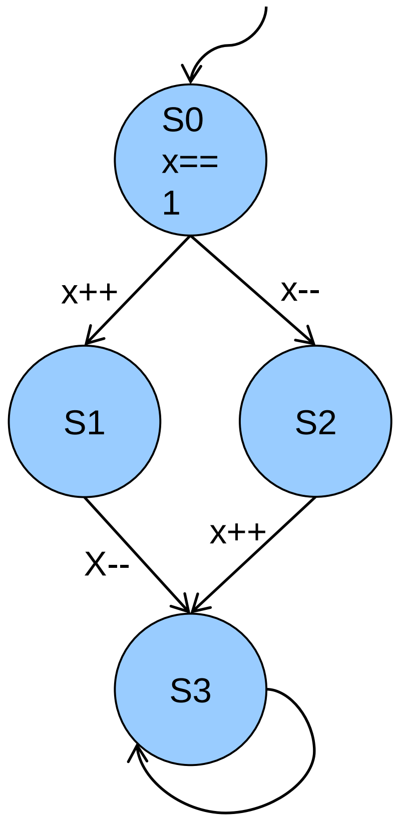
M |= F(x==2)?

Property specification

- AP = {coffee_chosen, tea_chosen, money_inserted, coffee_delivered, tea_delivered}
  - Once in a while someone chooses tea or coffee?
    - GF(tea_chosen \/ coffee_chosen)
  - If coffee is chosen and *next* money is inserted coffee will be delivered
    - G((coffee_chosen /\ X money_inserted) => F coffee_delivered)
  - When coffee has been chosen tea will not be delivered until tea is chosen

•G(coffee_chosen => (¬tea_delivered U tea_chosen))

Is LTL = CTL?



$\boxed{M \models F(x{=}{=}2)?}$ NO

LTL

$\boxed{M \models EF(x{=}{=}2)?}$ YES

CTL

Application of model-checking to software programs

- What kind of software programs need model-checking?
  - Concurrent programs
  - Concurrent programs with shared variables.
  - When two (or more) processes write/read from a shared variable it needs synchronization, e.g., mutexes, spin locks, etc.
  - We will use model-checkers to make sure that mutual exclusion is working.

Types of errors we can avoid using model-checking

- Deadlocks
- Livelocks
- Underspecification
- Over specification
- Buffer over-runs
- Array bound violations
- ..... many others

WE WILL USE THE SPIN MODEL-CHECKER

What is SPIN?

- SPIN = Simple Promela Interpreter

(see: spinroot.com)

- SPIN model-checks LTL properties on a model of the concurrent software program(s).

- The *model* of the concurrent software program is specified in a language called Promela, and hence the name SPIN!

Promela parts

- Promela model consist of:
  - type declarations
  - channel declarations
  - variable declarations
  - process declarations
  - [`init` process]
- A Promela model corresponds with a (usually very large, but) finite transition system, so
  - no unbounded data
  - no unbounded channels
  - no unbounded processes
  - no unbounded process creation

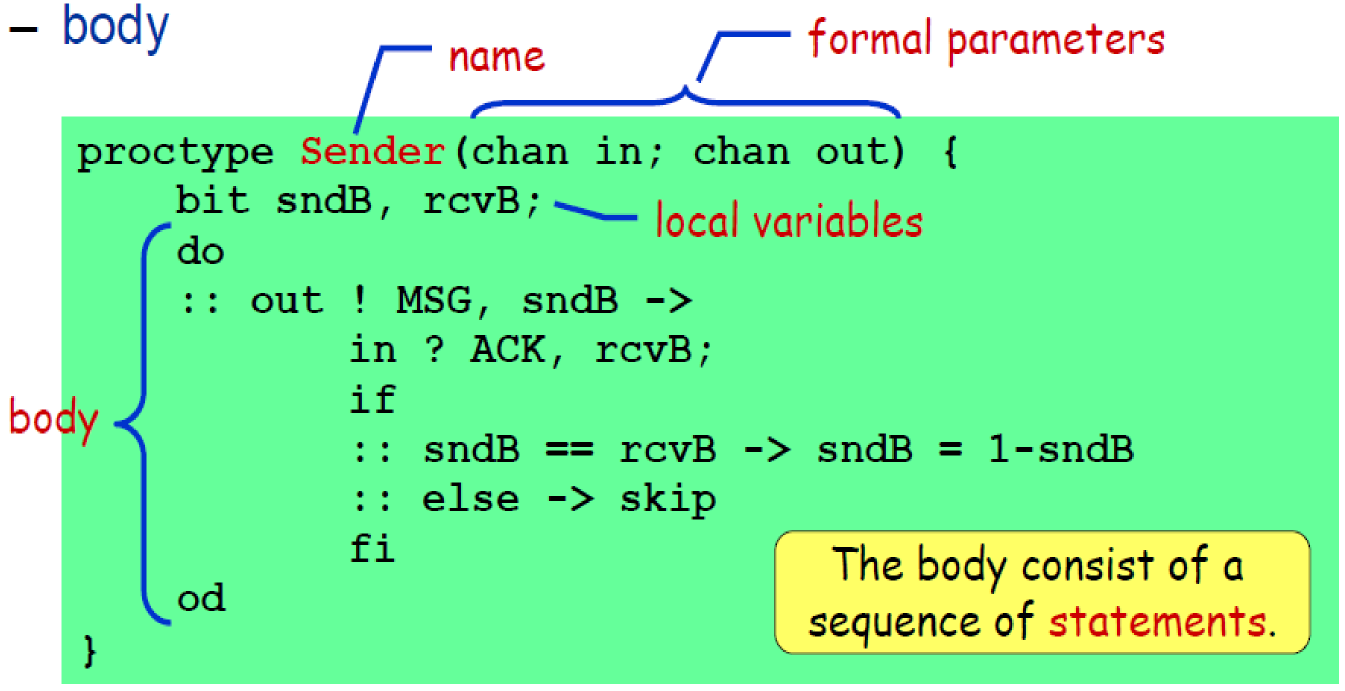
```
mtype = {MSG, ACK};
chan toS = ...
chan toR = ...
bool flag;

proctype Sender() {
   ...
}

proctype Receiver() {
   ...
}

init {
   ...
}
```

process body

creates processes

Promela process

- A process type (`proctype`) consist of
  - a name
  - a list of formal parameters
  - local variable declarations
  - body



```
proctype Sender(chan in; chan out) {
    bit sndB, rcvB;
    do
    :: out ! MSG, sndB ->
            in ? ACK, rcvB;
            if
            :: sndB == rcvB -> sndB = 1-sndB
            :: else -> skip
            fi
    od
}
```

Promela process

- A process
  - is defined by a `proctype` definition
  - executes concurrently with all other processes, independent of speed of behaviour
  - communicate with other processes
    - using global (shared) variables
    - using channels
- There may be several processes of the same type.
- Each process has its own local state:
  - process counter (location within the `proctype`)
  - contents of the local variables

Promela process

- Process are created using the **run** statement (which returns the process id).
- Processes can be created at any point in the execution (within any process).
- Processes start executing after the **run** statement.
- Processes can also be created by adding `active` in front of the `proctype` declaration.

```
proctype Foo(byte x) {
  ...
}


init {
  int pid2 = run Foo(2);
  run Foo(27);
}


active[3] proctype Bar() {
  ...
}
```

number of procs. (opt.)

parameters will be initialised to 0

Promela process

```
/* A "Hello World" Promela model for SPIN. */
active proctype Hello() {
    printf("Hello process, my pid is: %d\n", _pid);
}
init {
    int lastpid;
    printf("init process, my pid is: %d\n", _pid);
    lastpid = run Hello();
    printf("last pid was: %d\n", lastpid);
}
```

random seed

```
$ spin -n2 hello.pr
init process, my pid is: 1
        last pid was: 2
Hello process, my pid is: 0
                Hello process, my pid is: 2
3 processes created
```

running SPIN in random simulation mode

Variables and Types

- Five different (integer) basic types.
- Arrays
- Records (structs)
- Type conflicts are detected at runtime.
- Default initial value of basic variables (local and global) is 0.

Basic types

`bit turn=1;`	[0..1]
`bool flag;`	[0..1]
`byte counter;`	[0..255]
`short s;`	$[-2^{16}-1 .. 2^{16}-1]$
`int msg;`	$[-2^{32}-1 .. 2^{32}-1]$

Arrays

array indicing start at 0

```
byte a[27];
bit  flags[4];
```

Typedef (records)

```
typedef Record {
  short f1;
  byte  f2;
}
Record rr;
rr.f1 = ..
```

variable declaration

Variables and Types

- Variables should be declared.
- Variables can be given a value by:
  - assignment
  - argument passing
  - message passing (see communication)
- Variables can be used in expressions.

Most arithmetic, relational, and logical operators of C/Java are supported, including bitshift operators.

```
int ii;
bit bb;

bb=1;
ii=2;

short s=-1;

typedef Foo {
  bit bb;
  int ii;
};
Foo f;
f.bb = 0;
f.ii = -2;

ii*s+27 == 23;
printf(“value: %d”, s*s);
```

assignment =

declaration + initialisation

equal test ==

Promela statements

- The body of a process consists of a sequence of statements. A statement is either
  - executable: the statement can be executed immediately.
  - blocked: the statement cannot be executed.

executable/blocked depends on the global state of the system.

- An assignment is always executable.
- An expression is also a statement; it is executable if it evaluates to non-zero.

`2 < 3`	always executable
`x < 27`	only executable if value of `x` is smaller 27
`3 + x`	executable if `x` is not equal to –3

Promela statements

Statements are separated by a semi-colon: ";".

- The **skip** statement is always executable.
  - "does nothing", only changes process' process counter
- A **run** statement is only executable if a new process can be created (remember: the number of processes is bounded).
- A **printf** statement is always executable (but is not evaluated during verification, of course).

```
int x;
proctype Aap()
{
  int y=1;
  skip;
  run Noot();
  x=2;
  x>2 && y==1;
  skip;
}
```

Executable if Noot can be created…

Can only become executable if a some other process makes x greater than 2.

Promela statements

- `assert(<expr>);`
  - The `assert`-statement is always executable.
  - If `<expr>` evaluates to zero, SPIN will exit with an error, as the `<expr>` "has been violated".
  - The `assert`-statement is often used within Promela models, to check whether certain properties are valid in a state.

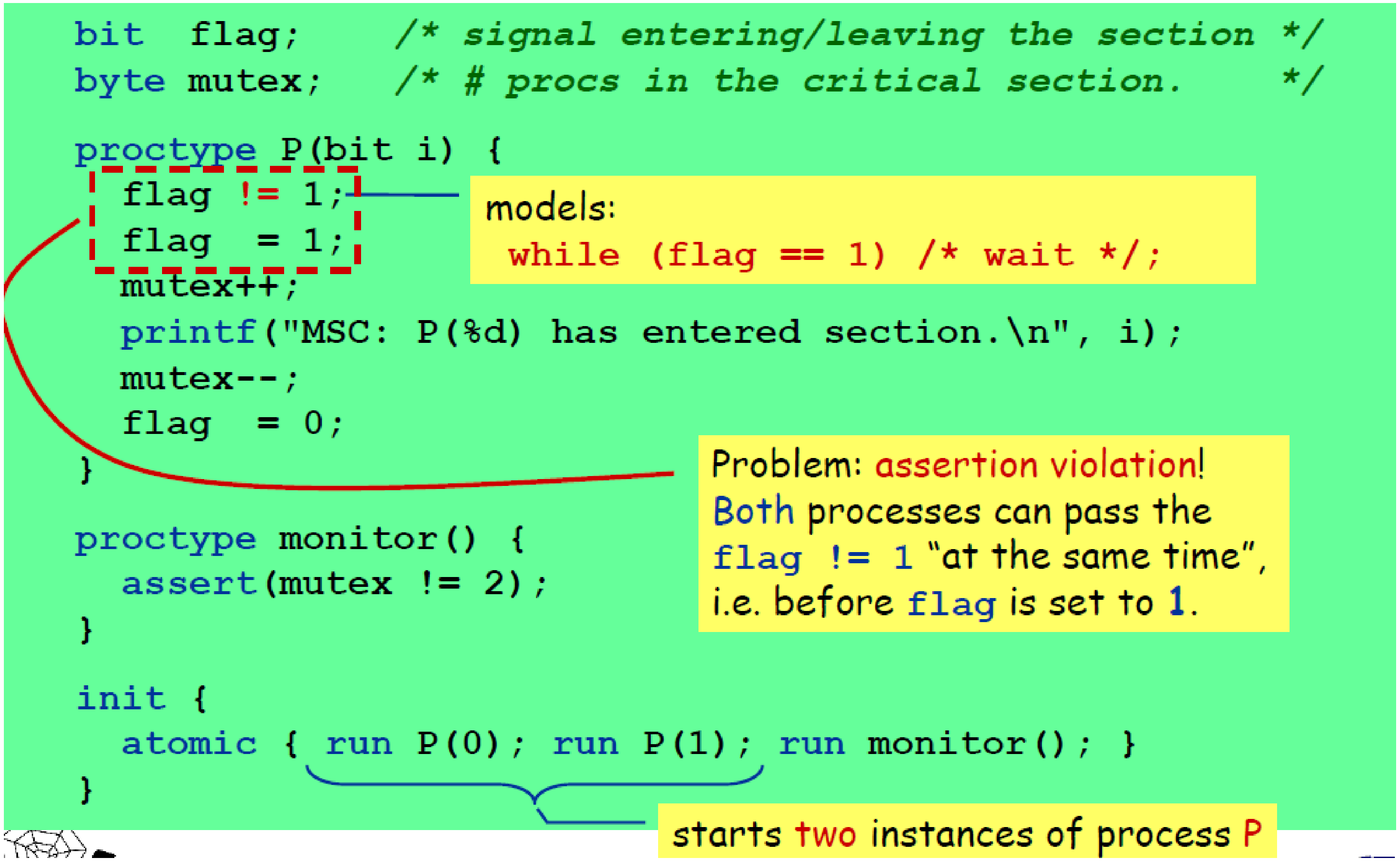
```
proctype monitor() {
  assert(n <= 3);
}

proctype receiver() {
  ...
  toReceiver ? msg;
  assert(msg != ERROR);
  ...
}
```

Promela semantics

- Promela processes execute concurrently.
- Non-deterministic scheduling of the processes.
- Processes are interleaved (statements of different processes do not occur at the same time).
  - exception: rendez-vous communication.
- All statements are atomic; each statement is executed without interleaving with other processes.
- Each process may have several different possible actions enabled at each point of execution.
  - only one choice is made, non-deterministically.

= randomly

Promela example – mutual exclusion



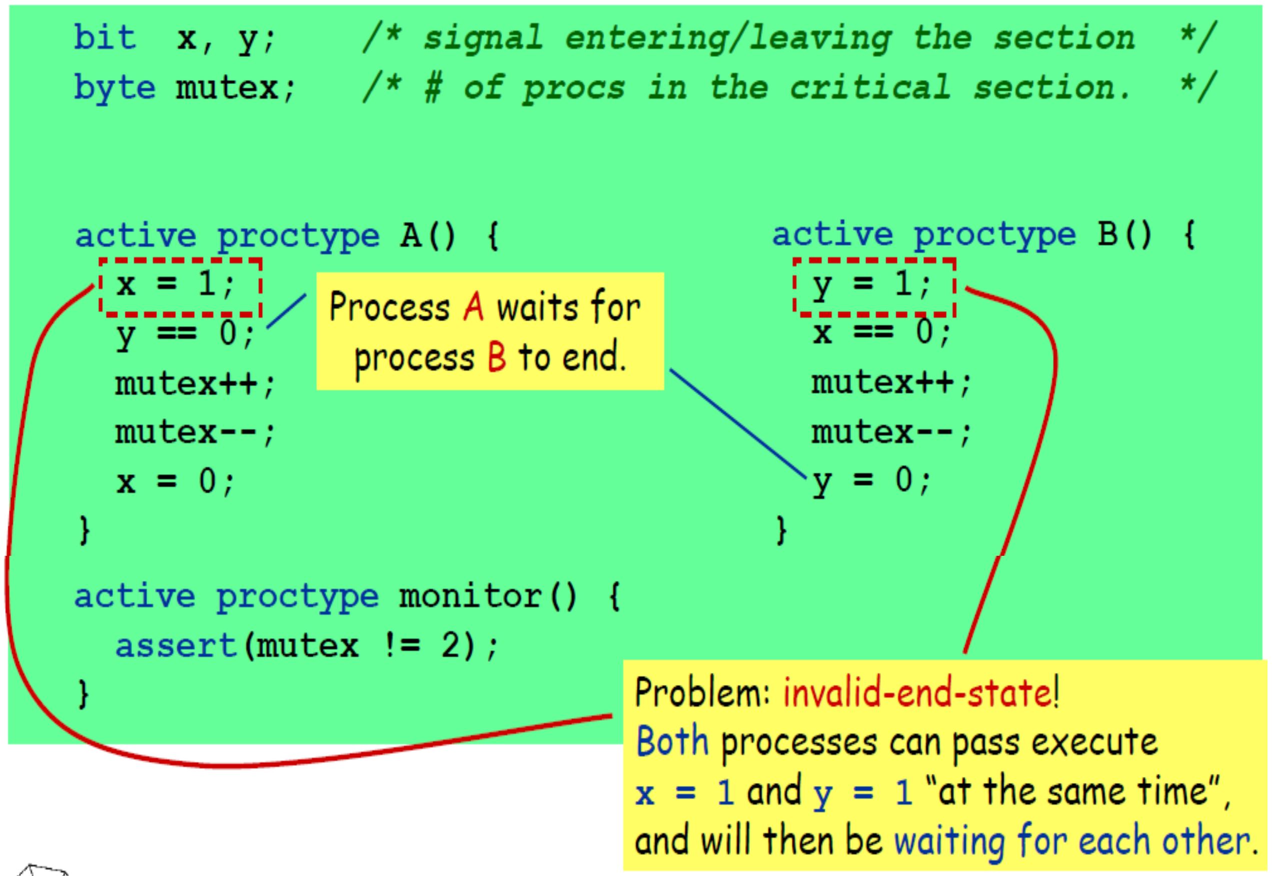
```
bit  flag;     /* signal entering/leaving the section */
byte mutex;    /* # procs in the critical section.    */

proctype P(bit i) {
   flag != 1;
   flag  = 1;
   mutex++;
   printf("MSC: P(%d) has entered section.\n", i);
   mutex--;
   flag  = 0;
}

proctype monitor() {
   assert(mutex != 2);
}

init {
   atomic { run P(0); run P(1); run monitor(); }
}
```

models:
`while (flag == 1) /* wait */;`

Problem: assertion violation!
Both processes can pass the `flag != 1` “at the same time”, i.e. before `flag` is set to 1.

starts two instances of process P

Promela example – mutual exclusion

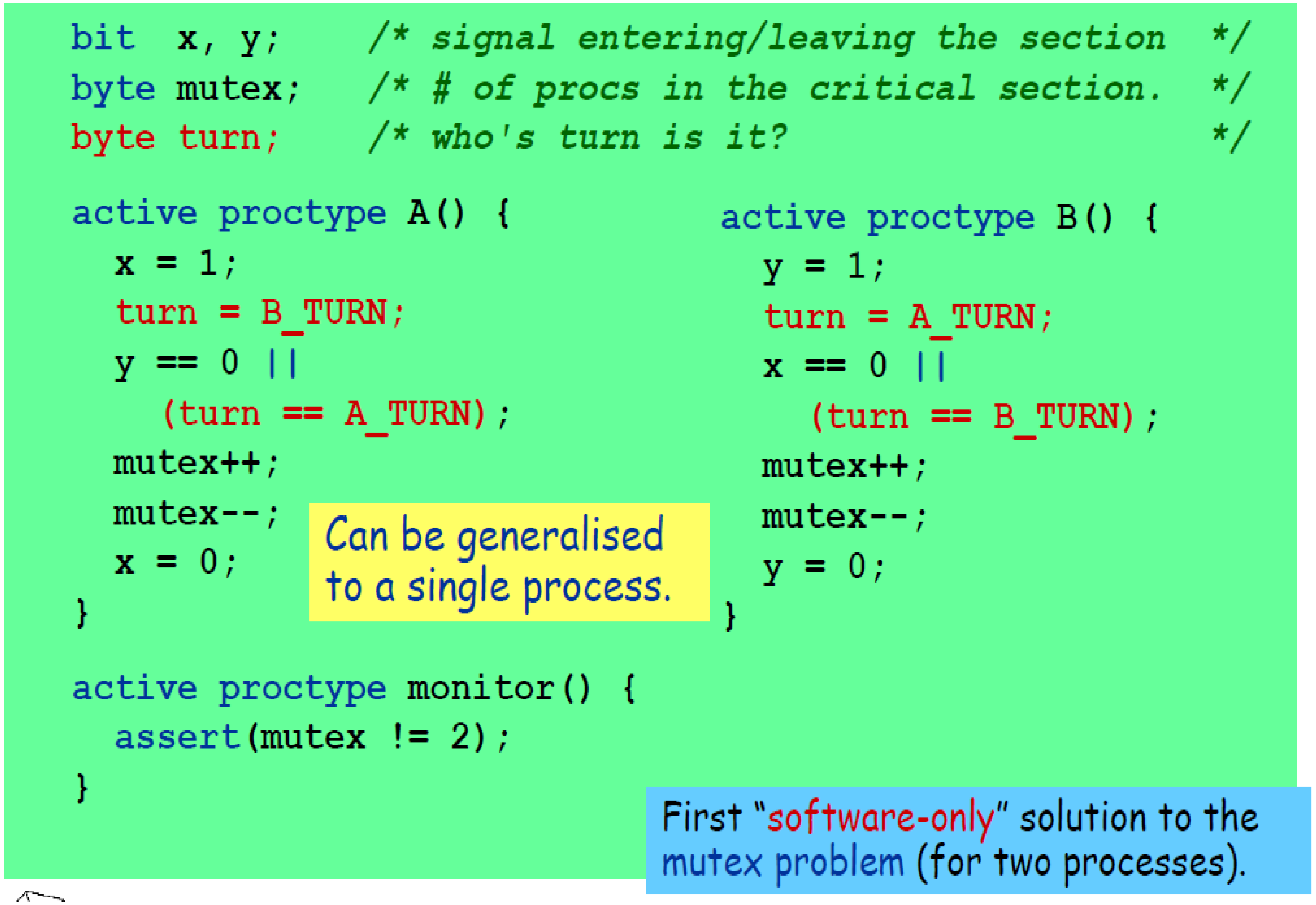


```
bit  x, y;      /* signal entering/leaving the section  */
byte mutex;     /* # of procs in the critical section.  */


active proctype A() {                 active proctype B() {
  x = 1;                                y = 1;
  y == 0;                               x == 0;
  mutex++;                              mutex++;
  mutex--;                              mutex--;
  x = 0;                                y = 0;
}                                     }

active proctype monitor() {
  assert(mutex != 2);
}
```

Process A waits for process B to end.

Problem: invalid-end-state!
Both processes can pass execute
x = 1 and y = 1 "at the same time",
and will then be waiting for each other.

Promela example – mutual exclusion



```
bit  x, y;      /* signal entering/leaving the section  */
byte mutex;     /* # of procs in the critical section.  */
byte turn;      /* who's turn is it?                    */

active proctype A() {              active proctype B() {
  x = 1;                             y = 1;
  turn = B_TURN;                     turn = A_TURN;
  y == 0 ||                          x == 0 ||
    (turn == A_TURN);                  (turn == B_TURN);
  mutex++;                           mutex++;
  mutex--;                           mutex--;
  x = 0;                             y = 0;
}                                  }

active proctype monitor() {
  assert(mutex != 2);
}
```

Can be generalised to a single process.

First "software-only" solution to the mutex problem (for two processes).

Promela If-statements

```
if
:: choice₁ -> stat₁.₁; stat₁.₂; stat₁.₃; …
:: choice₂ -> stat₂.₁; stat₂.₂; stat₂.₃; …
:: …
:: choiceₙ -> statₙ.₁; statₙ.₂; statₙ.₃; …
fi;
```

- If there is at least one `choice`$_i$ (guard) executable, the `if`-statement is executable and SPIN non-deterministically chooses one of the executable choices.
- If no `choice`$_i$ is executable, the `if`-statement is blocked.
- The operator "`->`" is equivalent to "`;`". By convention, it is used within `if`-statements to separate the guards from the statements that follow the guards.

Promela If-statements

```
if
:: (n % 2 != 0) -> n=1
:: (n >= 0)      -> n=n-2
:: (n % 3 == 0) -> n=3
:: else          -> skip
fi
```

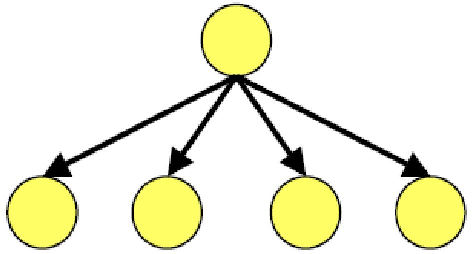
- The `else` guard becomes executable if none of the other guards is executable.

give n a random value

```
if
:: skip -> n=0
:: skip -> n=1
:: skip -> n=2
:: skip -> n=3
fi
```

skips are redundant, because assignments are themselves always executable...

non-deterministic branching



Promela do-statement (loop)

```
do
:: choice1 -> stat1.1; stat1.2; stat1.3; …
:: choice2 -> stat2.1; stat2.2; stat2.3; …
:: …
:: choicen -> statn.1; statn.2; statn.3; …
od;
```

- With respect to the choices, a `do`-statement behaves in the same way as an `if`-statement.
- However, instead of ending the statement at the end of the choosen list of statements, a `do`-statement repeats the choice selection.
- The (always executable) `break` statement exits a `do`-loop statement and transfers control to the end of the loop.

Promela do-statement (loop)

- Example – modelling a traffic light

if- and do-statements are ordinary Promela statements; so they can be nested.

```
mtype = { RED, YELLOW, GREEN } ;
```

mtype (message type) models enumerations in Promela

```
active proctype TrafficLight() {
    byte state = GREEN;
    do
    ::  (state == GREEN)  -> state = YELLOW;
    ::  (state == YELLOW) -> state = RED;
    ::  (state == RED)    -> state = GREEN;
    od;
}
```

Note: this do-loop does not contain any non-deterministic choice.

Promela atomic-statement

`atomic { stat`$_1$`; stat`$_2$`; ... stat`$_n$` }`

- can be used to group statements into an atomic sequence; all statements are executed in a single step (no interleaving with statements of other processes)
- is executable if `stat`$_1$ is executable / no pure atomicity
- if a `stat`$_i$ (with `i>1`) is blocked, the "atomicity token" is (temporarily) lost and other processes may do a step

(Hardware) solution to the mutual exclusion problem:

```
proctype P(bit i) {
  atomic {flag != 1; flag = 1; }
  mutex++;
  mutex--;
  flag  = 0;
}
```

Promela d_step-statement

`d_step { stat₁; stat₂; ... statₙ }`

- more efficient version of `atomic`: no intermediate states are generated and stored
- may only contain deterministic steps
- it is a run-time error if $\texttt{stat}_i$ `(i>1)` blocks.

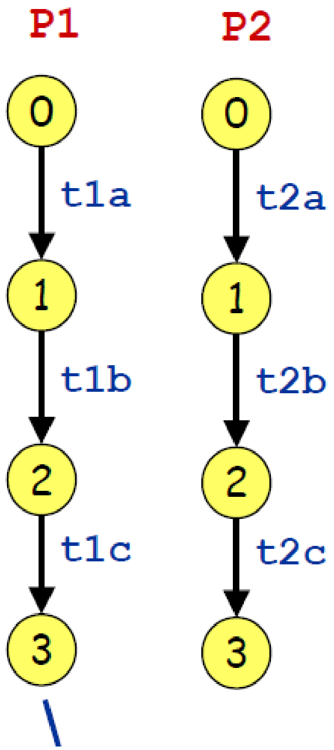
- `d_step` is especially useful to perform intermediate computations in a single transition

```
::  Rout?i(v) -> d_step {
        k++;
        e[k].ind = i;
        e[k].val = v;
        i=0; v=0 ;
      }
```

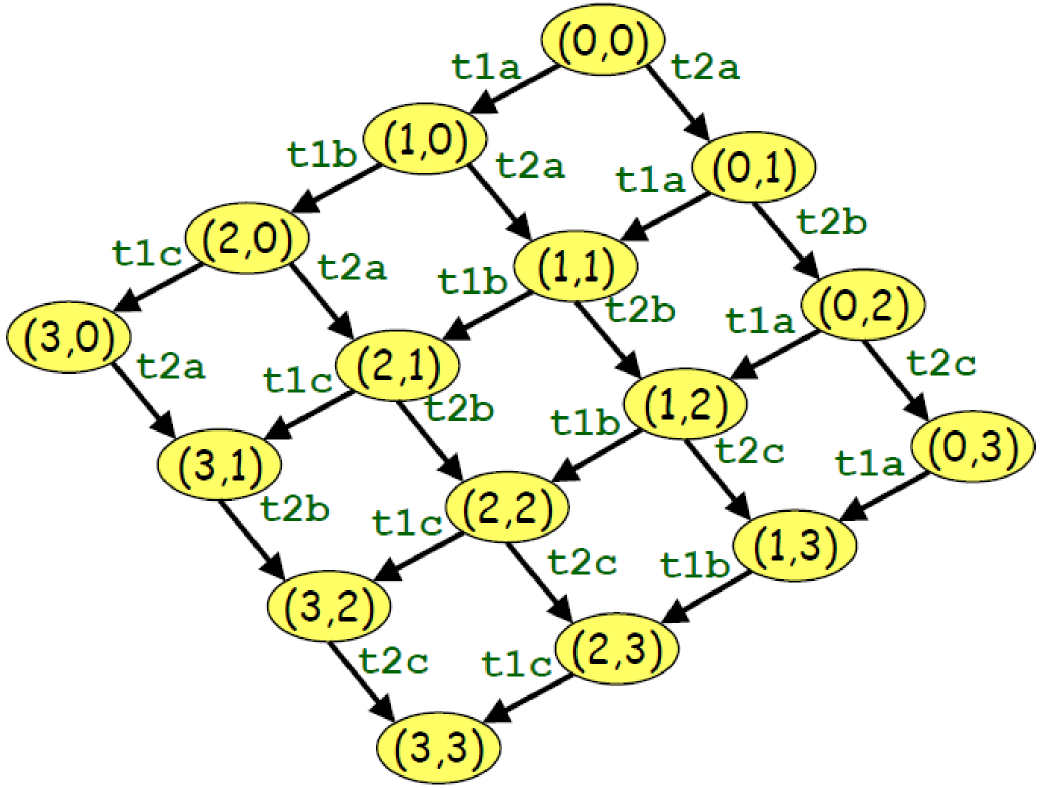
`atomic` and `d_step` can be used to lower the number of states of the model

Promela atomic/d_step-semantics

```
proctype P1() { t1a; t1b; t1c }
proctype P2() { t2a; t2b; t2c }
init { run P1(); run P2() }
```



No atomicity



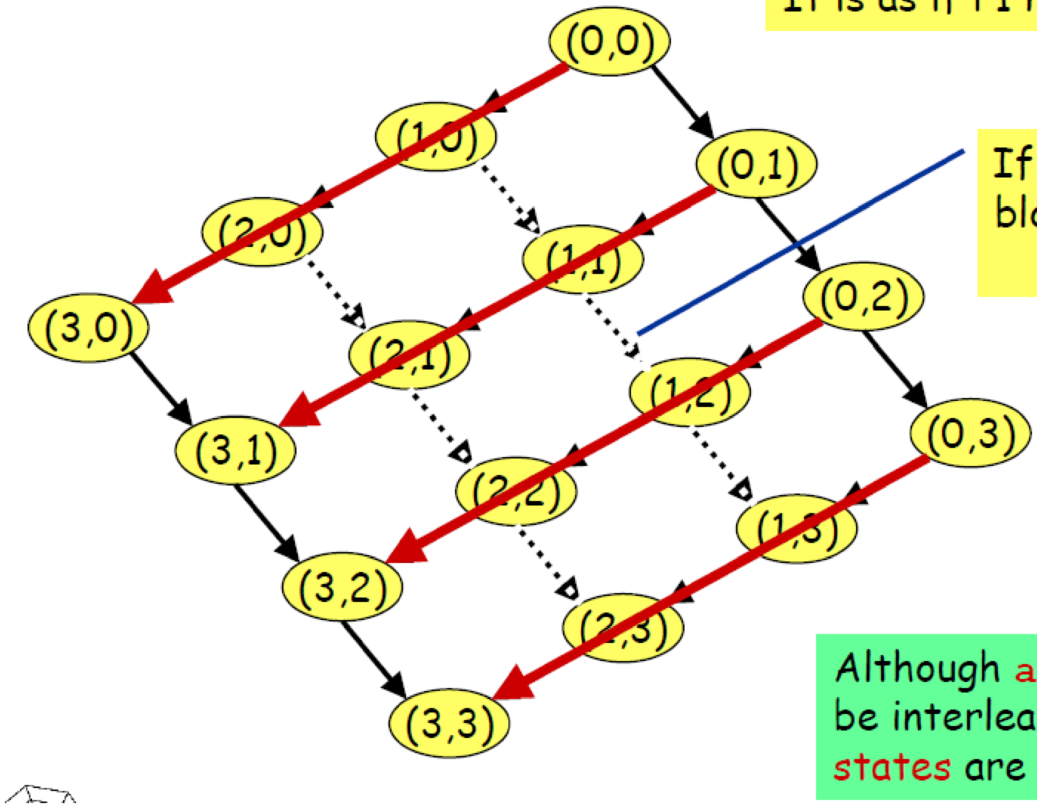
Promela atomic-semantics

```
proctype P1() { atomic {t1a; t1b; t1c} }
proctype P2() { t2a; t2b; t2c }
init { run P1(); run P2() }
```

atomic

It is as if P1 has only one transition…

If one of P1's transitions blocks, these transitions may get executed



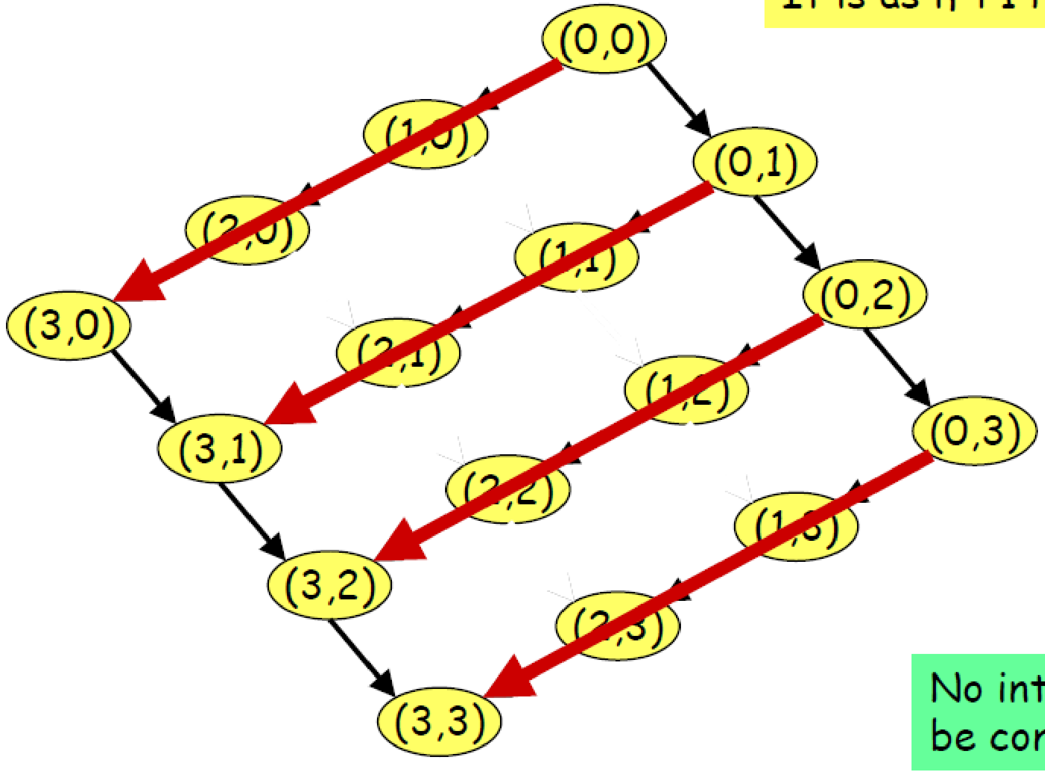
Although atomic clauses cannot be interleaved, the intermediate states are still constructed.

Promela d_step-semantics

```
proctype P1() { d_step {t1a; t1b; t1c} }
proctype P2() { t2a; t2b; t2c }
init { run P1(); run P2() }
```

d_step

It is as if P1 has only one transition…



No intermediate states will be constructed.

Promela LTL property specification

```
bit flag;
byte mutex;
ltl {[](mutex != 2)}
proctype P(bit i) {
  flag != 1;
  flag = 1;
  mutex++;
  printf("MSC: P(%d) has
            entered section.\n", i);
  mutex--;
  flag = 0;
}

init {
  atomic { run P(0); run P(1);}
}
```

```
bit flag;
byte mutex;
proctype P(bit i) {
  flag != 1;
  flag = 1;
  mutex++;
  printf("MSC: P(%d) has
            entered section.\n", i);
  mutex--;
  flag = 0;
}
proctype monitor() {
  assert(mutex != 2);
}

init {
  atomic { run P(0); run P(1); run monitor(); }
}
```

Using interactive SPIN – spin

In the class!

Some things to remember

- Atomicity
  - Enclose statements that do not have to be interleaved within an `atomic` / `d_step` clause
    - Beware: the behaviour of the processes may change!
    - Beware of infinite loops.
- Computations
  - use `d_step` clauses to make the computation a single transition
  - reset temporary variables to **0** at the end of a `d_step`
- Processes
  - sometimes the behaviour of two processes can be combined into one; this is usually more effective.

Assignment problem update

- Implement in Promela a simple mutual exclusion algorithm and verify properties on it.
  - Safety property (we have seen this) – 2 processes do not enter the critical section together.
  - Liveness property – if a process is waiting it will eventually get access to the critical section.
  - etc...