CS477 Formal Software Development Methods

Elsa L Gunter
2112 SC, UIUC
egunter@illinois.edu
http://courses.engr.illinois.edu/cs477

Slides mostly a reproduction of Theo C. Ruys – SPIN Beginners' Tutorial

April 10, 2013

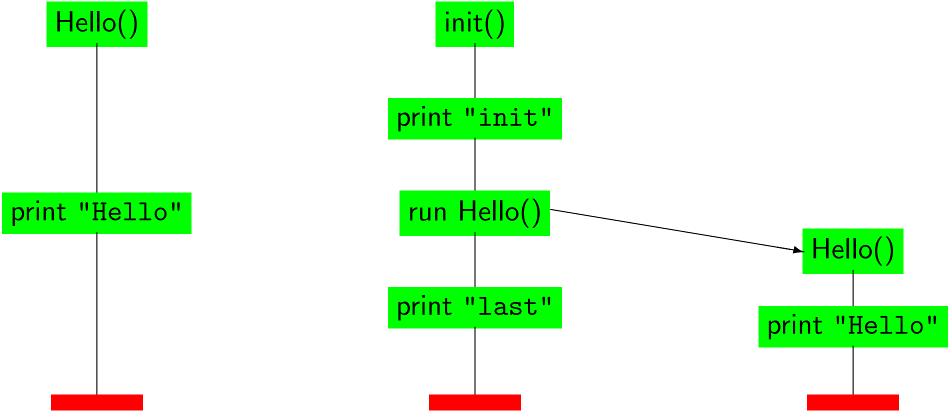
Hello World

```
/* 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);
}
```

Hello World, Sample Execution

```
bash-3.2$ spin hello.pml
          init process, my pid is: 1
      Hello process, my pid is: 0
              Hello process, my pid is: 2
          last pid was: 2
3 processes created
bash-3.2$ spin hello.pml
      Hello process, my pid is: 0
          init process, my pid is: 1
          last pid was: 2
              Hello process, my pid is: 2
3 processes created
```

Hello Processes



Hello()

print "Hello"

init()

print "init"

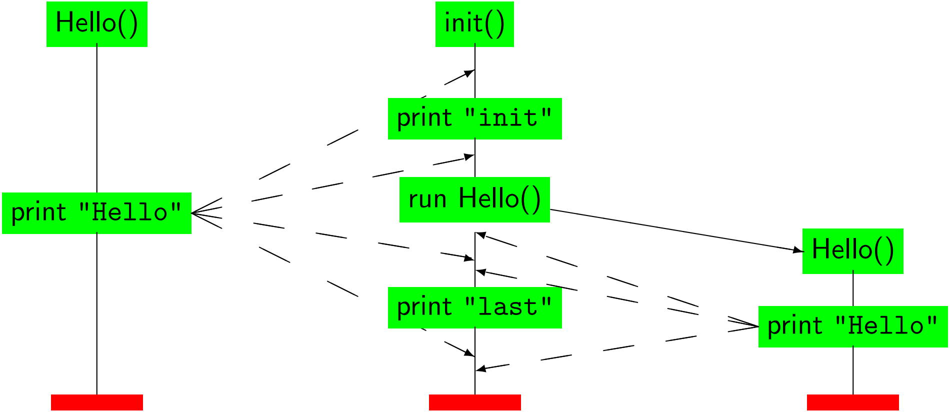
run Hello()

print "last"

Hello()

print "Hello"

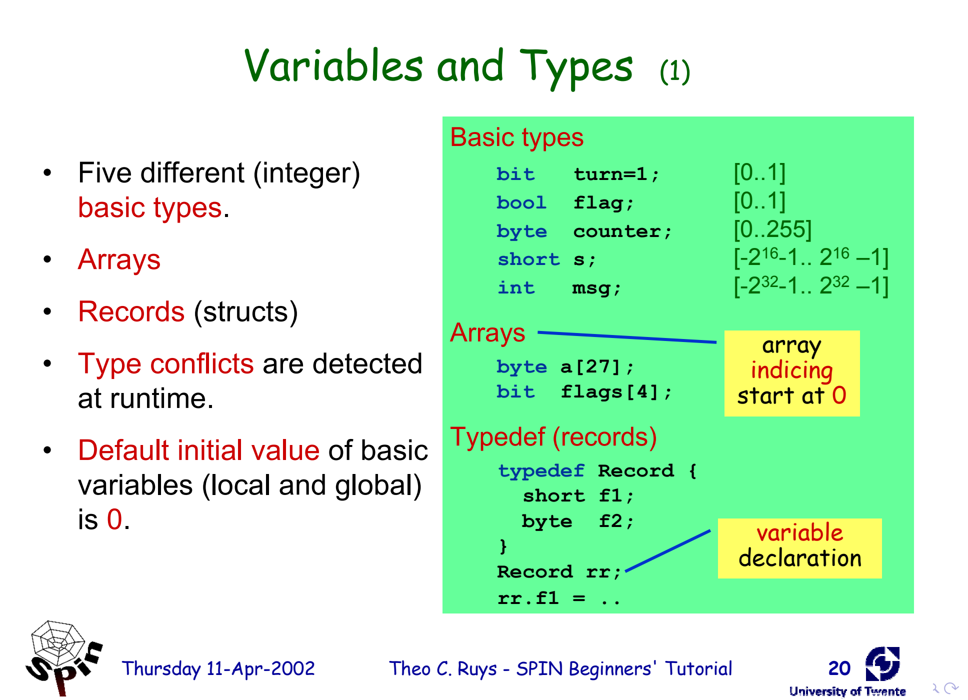
Hello Processes Interleavings



Interleaving 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



Variables and Types (1)

- 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

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

array indicing start at 0

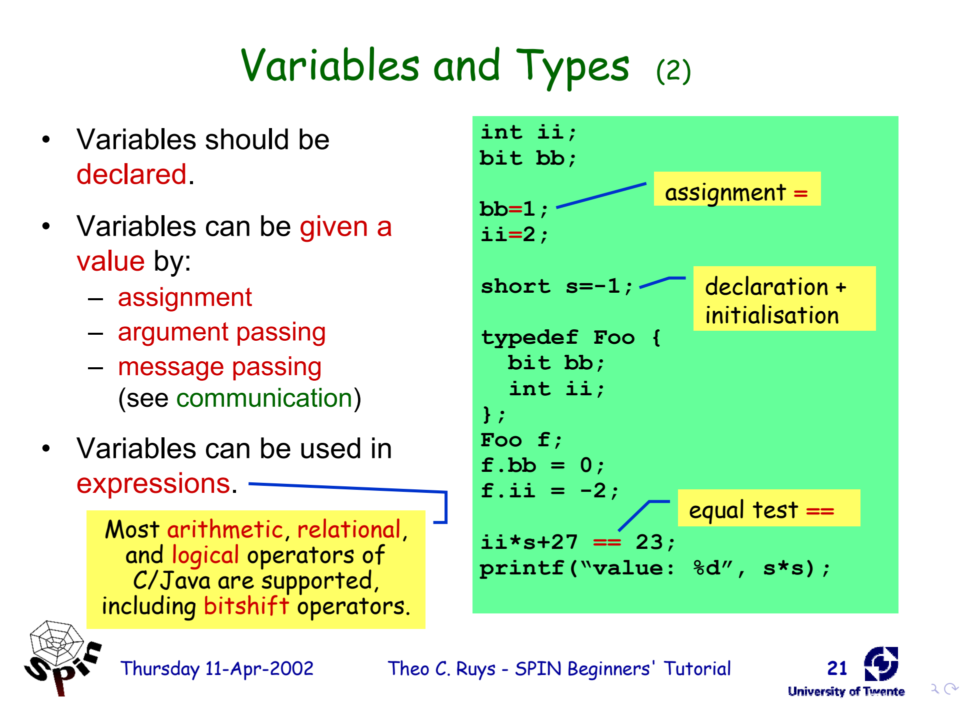
Typedef (records)

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

variable declaration

Thursday 11-Apr-2002 Theo C. Ruys - SPIN Beginners' Tutorial 20

University of Twente



Variables and Types (2)

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

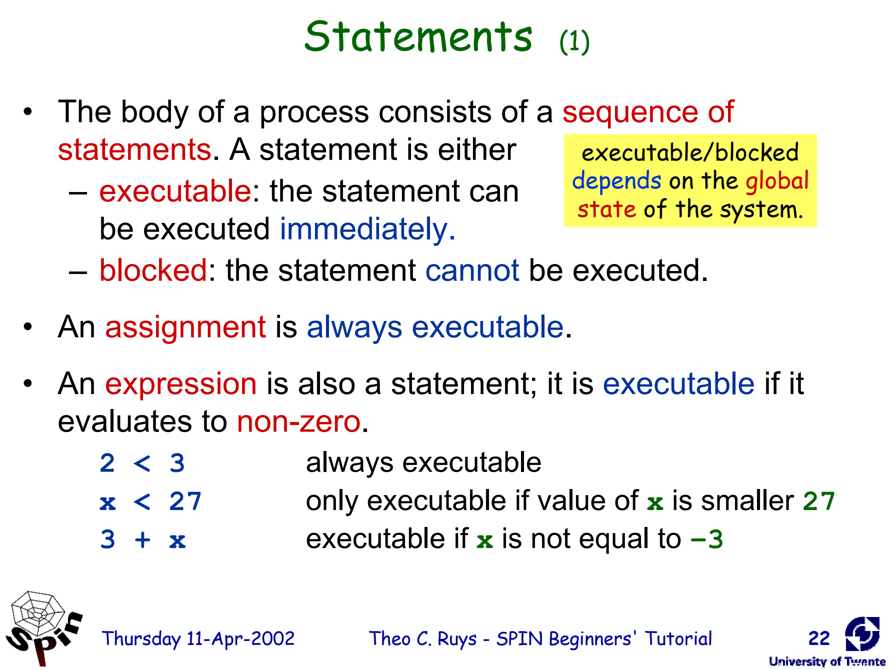
Most arithmetic, relational, and logical operators of C/Java are supported, including bitshift operators.

```
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 ==



Statements (1)

- 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.

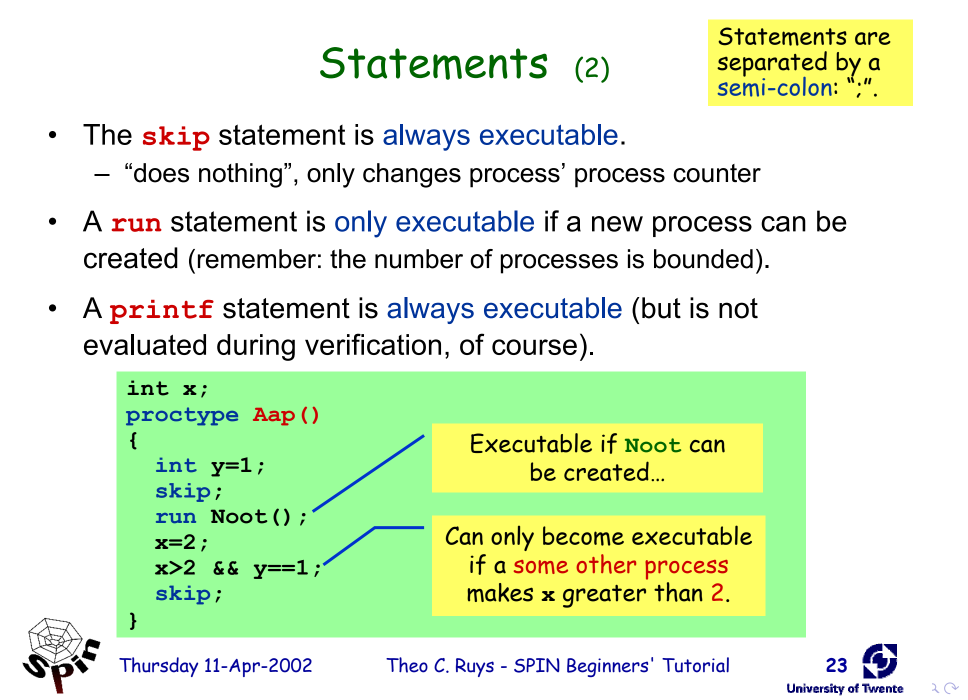
executable/blocked depends on the global state of the system.

- 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`

Thursday 11-Apr-2002 Theo C. Ruys - SPIN Beginners' Tutorial 22

University of Twente



Statements (2)

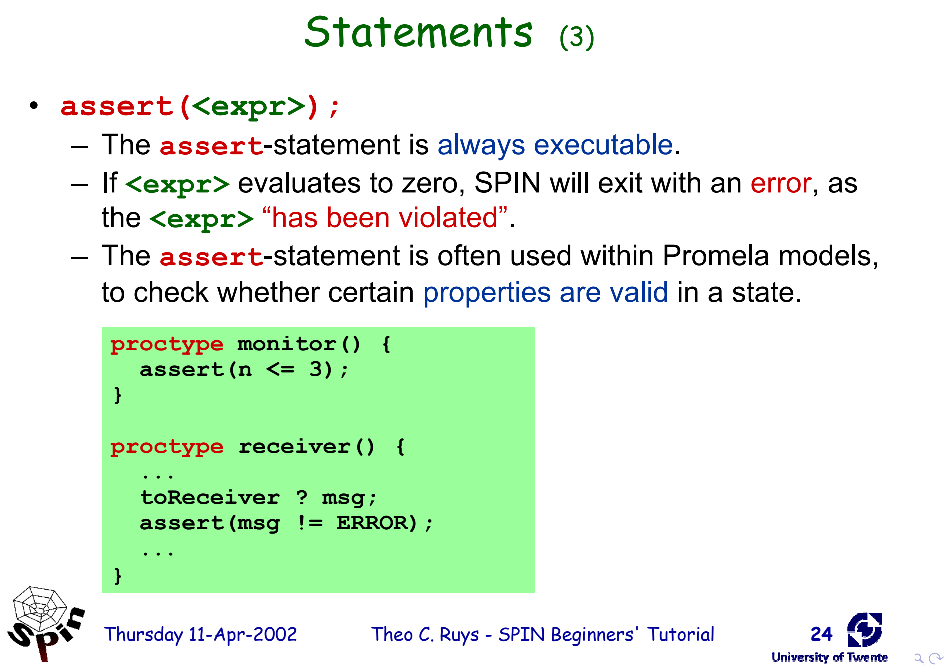
Statements are separated by a semi-colon: ";".

- 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…

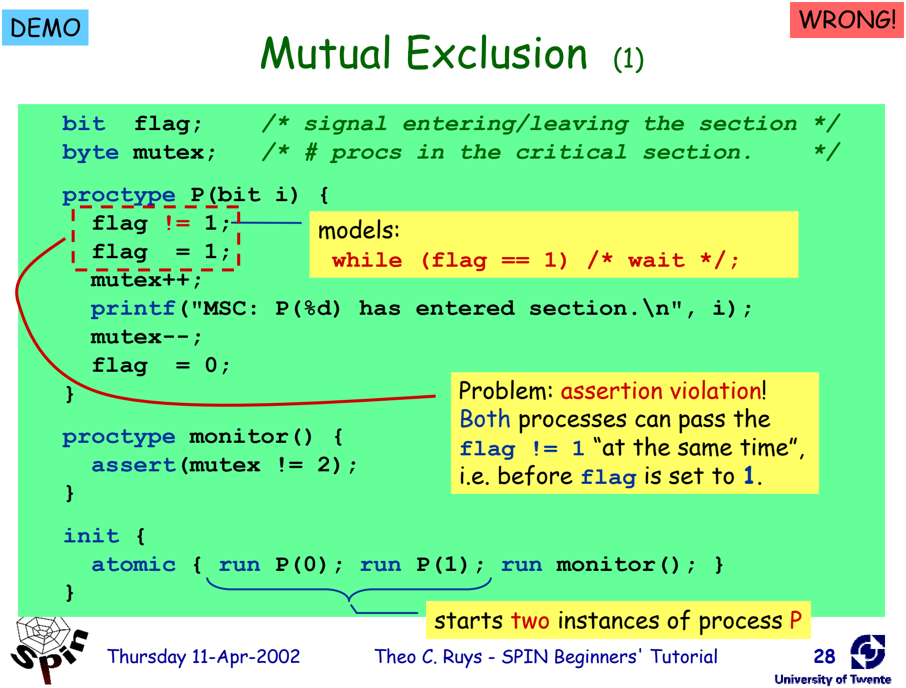
Can only become executable if a some other process makes **x** greater than 2.



Statements (3)

- **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);
  ...
}
```



DEMO

WRONG!

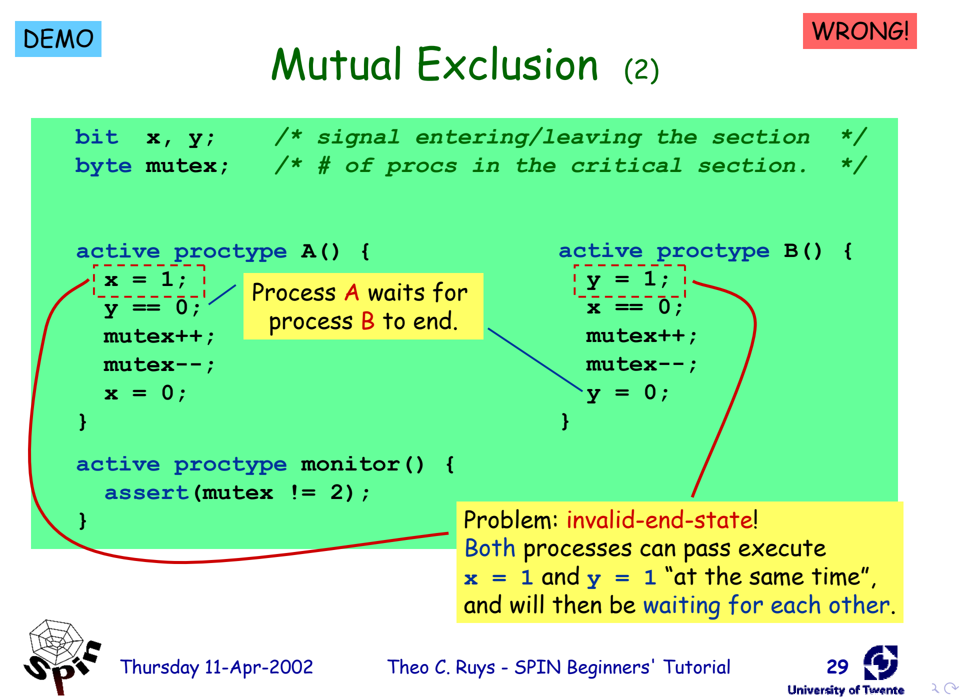
Mutual Exclusion (1)

```
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



DEMO

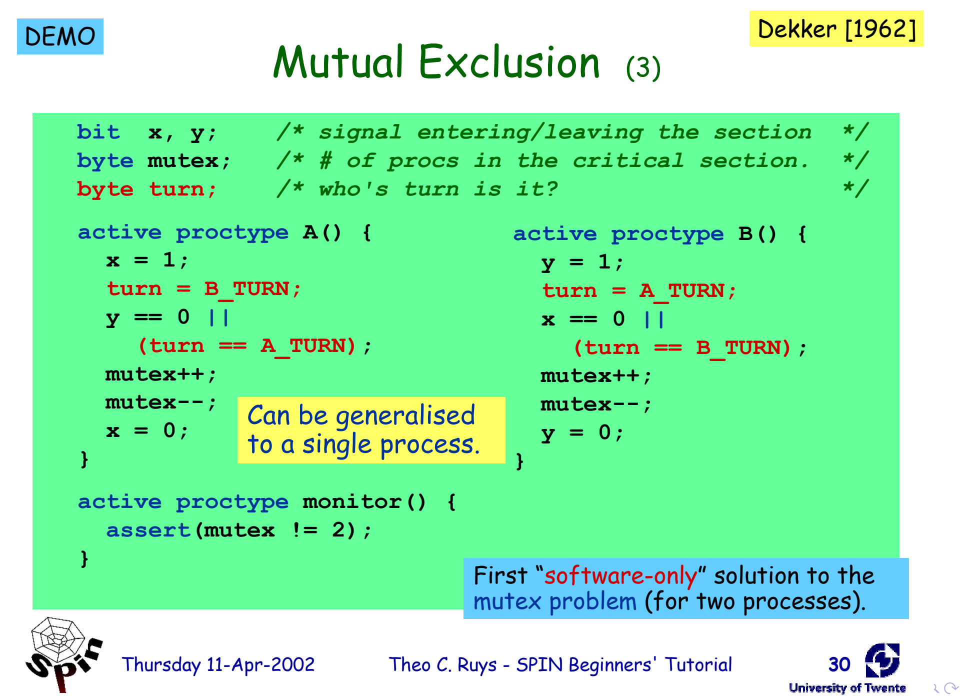
WRONG!

Mutual Exclusion (2)

```
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.



DEMO

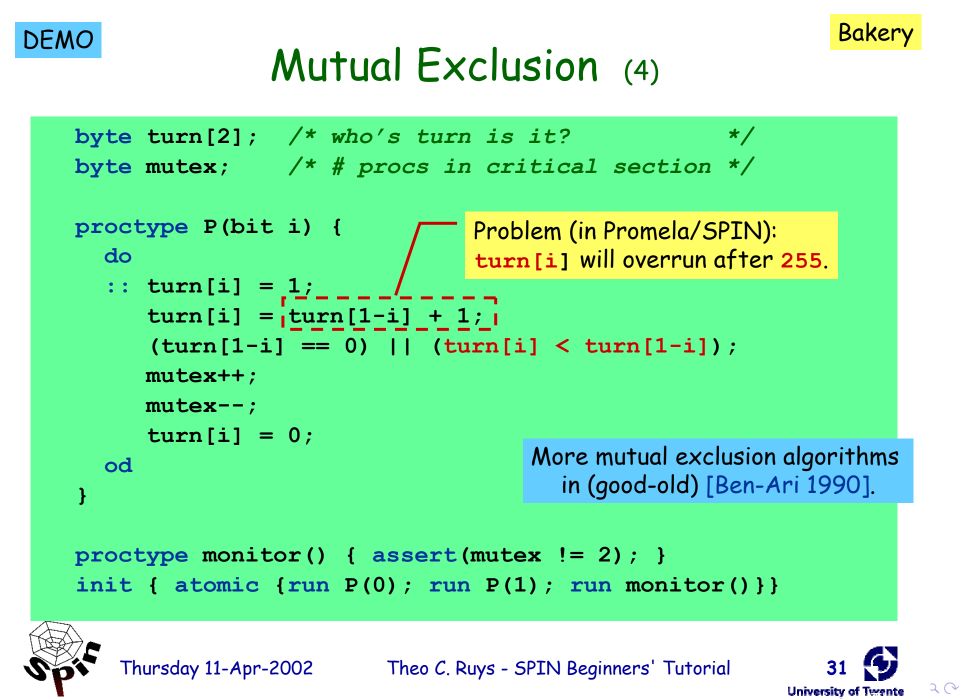
Dekker [1962]

Mutual Exclusion (3)

```
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).



DEMO

Bakery

Mutual Exclusion (4)

```
byte turn[2];   /* who's turn is it?          */
byte mutex;     /* # procs in critical section */

proctype P(bit i) {
  do
  :: turn[i] = 1;
     turn[i] = turn[1-i] + 1;
     (turn[1-i] == 0) || (turn[i] < turn[1-i]);
     mutex++;
     mutex--;
     turn[i] = 0;
  od
}

proctype monitor() { assert(mutex != 2); }
init { atomic {run P(0); run P(1); run monitor()}}
```

Problem (in Promela/SPIN):
turn[i] will overrun after 255.

More mutual exclusion algorithms
in (good-old) [Ben-Ari 1990].

Thursday 11-Apr-2002 Theo C. Ruys - SPIN Beginners' Tutorial 31

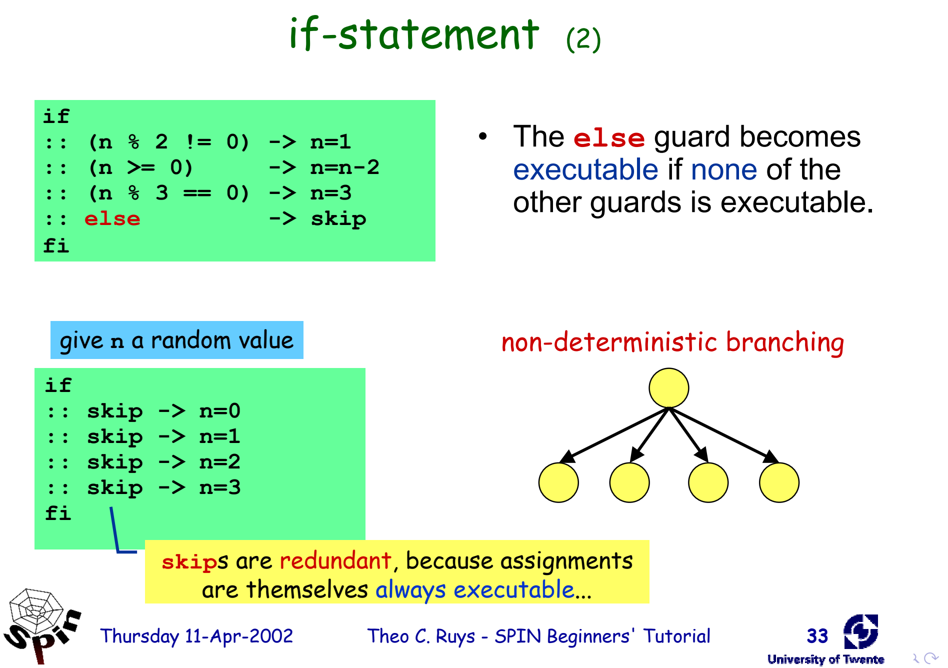
University of Twente

if-statement (1)

inspired by:
Dijkstra's guarded
command language

```
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.



if-statement (2)

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

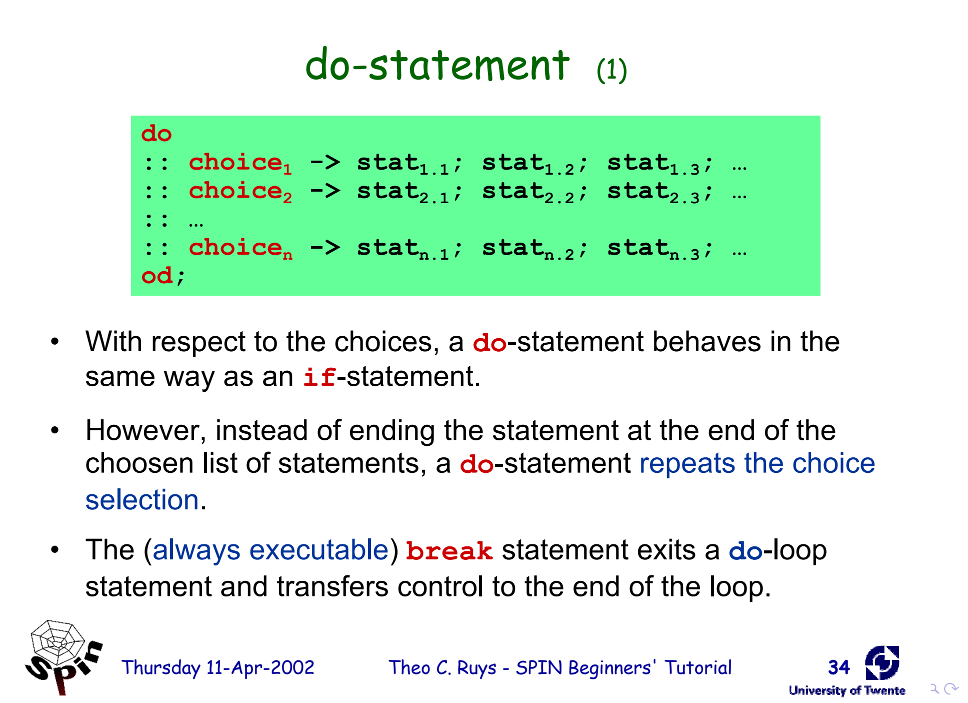
- The `else` guard becomes executable if none of the other guards is executable.

give `n` a random value

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

non-deterministic branching

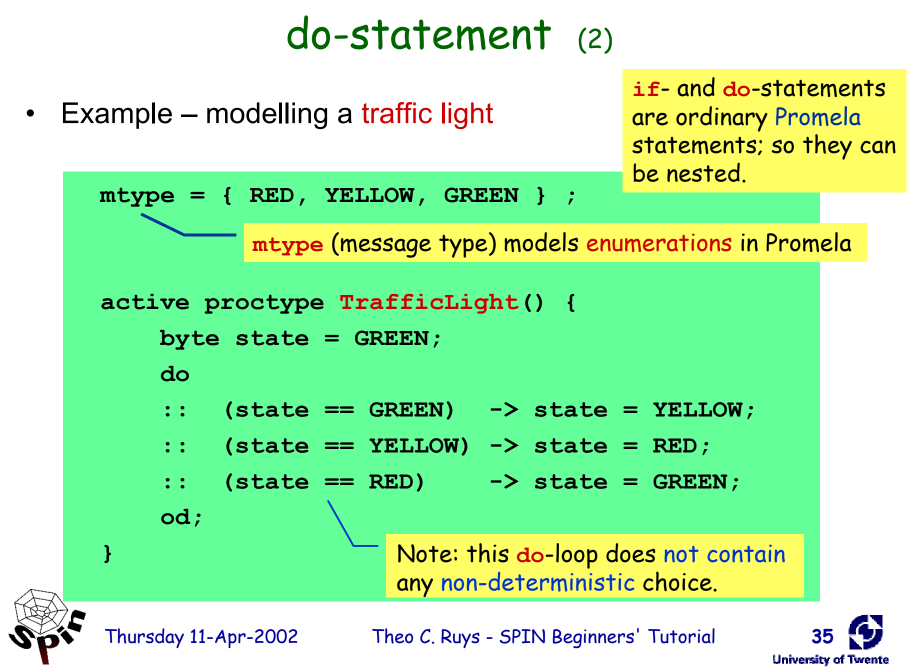
`skip`s are redundant, because assignments are themselves always executable...



do-statement (1)

```
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.



do-statement (2)

- Example – modelling a traffic light

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

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

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

mtype (message type) models enumerations in Promela

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