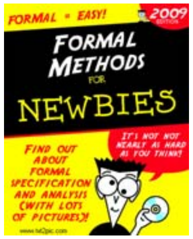
CISC422/853: Formal Methods in Software Engineering: Computer-Aided Verification


FORMAL = EASY!
2009 EDITION
FORMAL METHODS FOR NEWBIES
FIND OUT ABOUT FORMAL SPECIFICATION AND ANALYSIS (WITH LOTS OF PICTURES)!
IT'S NOT NOT NEARLY AS HARD AS YOU THINK!

Topic 6: Intro to Promela and Spin

Juergen Dingel
Feb, 2009

Readings:
Spin book, Chapters 3, 7, 11, 12

Modeling Behaviour of Systems

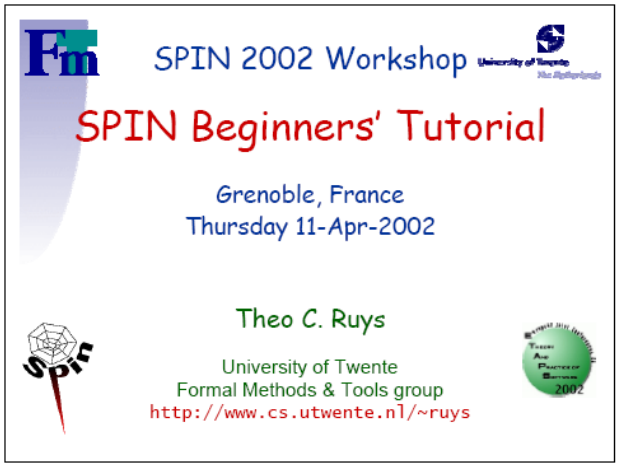
- Where are we?
  - We've decided to use FSAs to model the behaviour of software systems
  - Have seen:
    - Definition
    - Two types of parallel composition
    - Various extensions
- What's next?
  - But, to be able to feed FSAs into a model checker, we need to be able to express FSAs textually in some language
  - Also, it would be nice if that language was as high-level (user-friendly) as possible.
  - 2 examples for modeling languages based on FSAs:
    - BIR (used by Bogor model checker)
    - Promela (used by Spin model checker)

Promela and Spin

- **Promela** (PROcess MEta LAnguage):
  - modeling language used to describe concurrent systems, e.g.,
    - network protocols, telephone systems
    - multi-threaded programs that communicate via
      - shared variables, or
      - synchronous/asynchronous message passing
  - used by…
- **SPIN** (Simple Promela INterpreter):
  - analyzes Promela programs to detect errors such as
    - deadlocks, race conditions,
    - violations of assertions, invariants, safety and liveness properties
  - developed since late 1970s by Gerard Holzmann at Bell Labs (now at NASA's Jet Propulsion Lab)
  - received ACM Software System award in 2001

Intro to Promela

- `http://spinroot.com/spin/Doc/SpinTutorial.pdf`:



SPIN 2002 Workshop

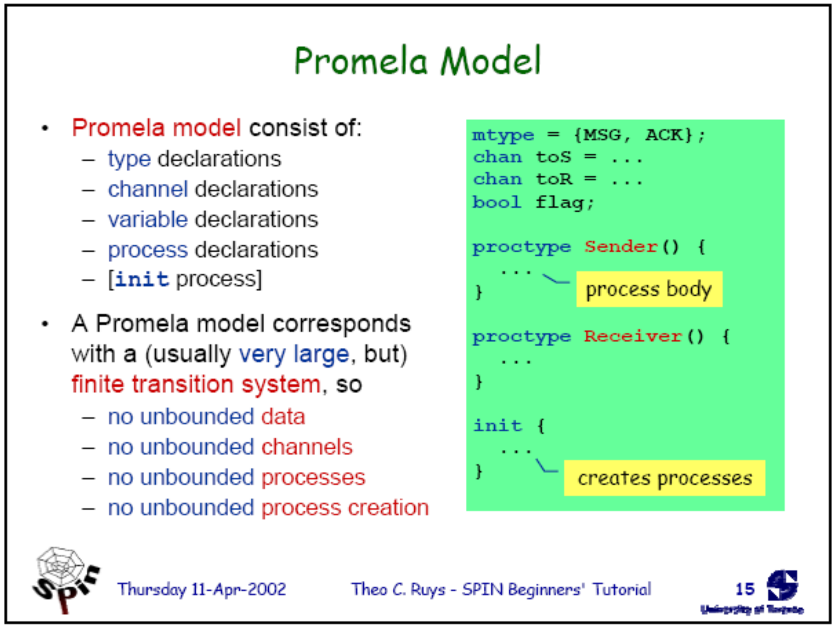
SPIN Beginners' Tutorial

Grenoble, France
Thursday 11-Apr-2002

Theo C. Ruys

University of Twente
Formal Methods & Tools group
http://www.cs.utwente.nl/~ruys

Fm
University of Twente
The Netherlands
Spin
2002



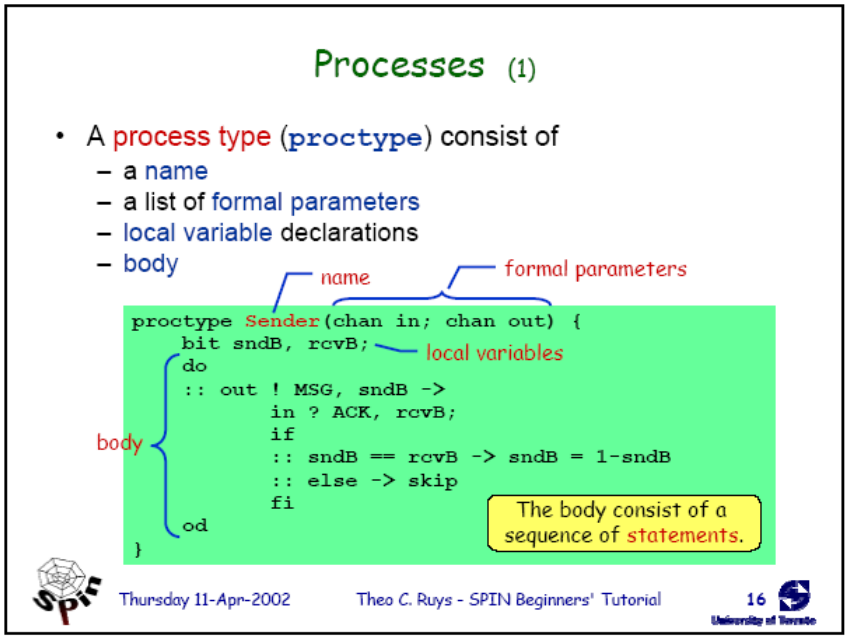
Promela Model

- 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



Processes (1)

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

name

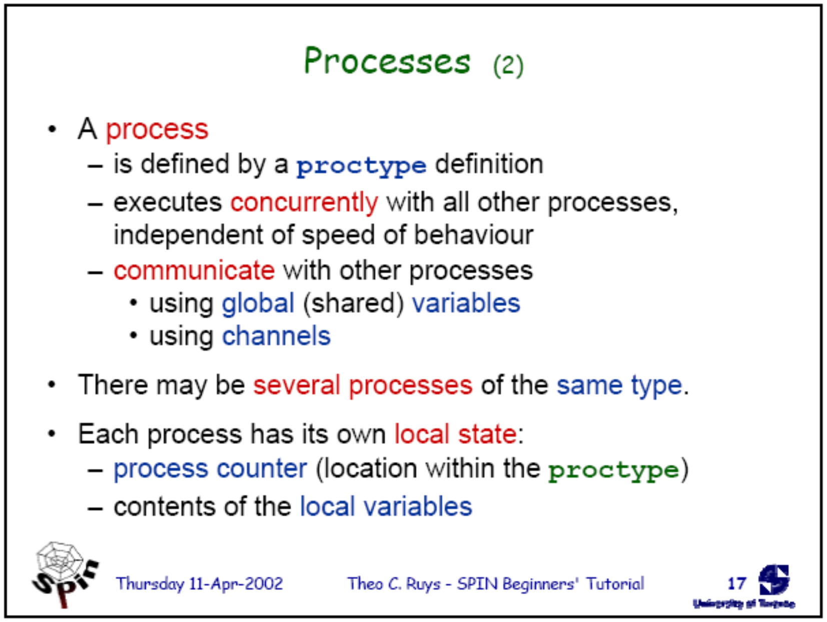
formal parameters

```
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
}
```

local variables

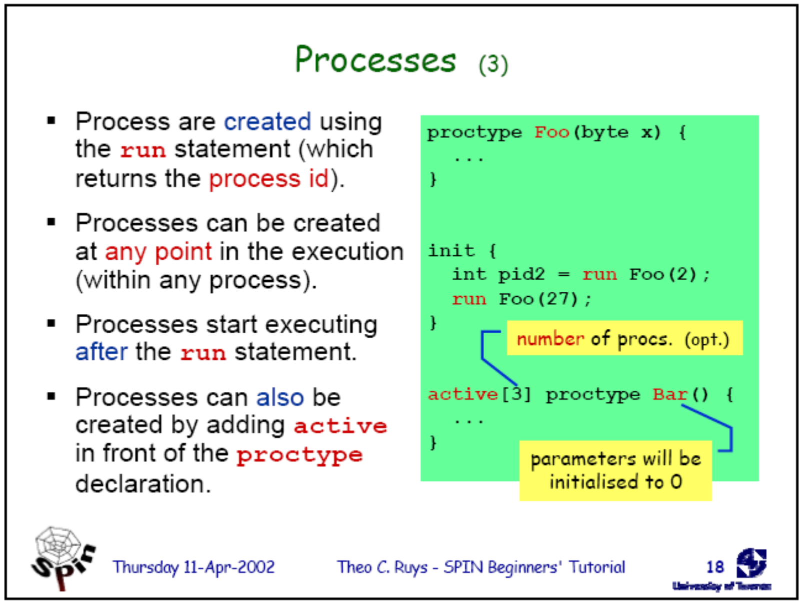
body

The body consist of a sequence of statements.



Processes (2)

- 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



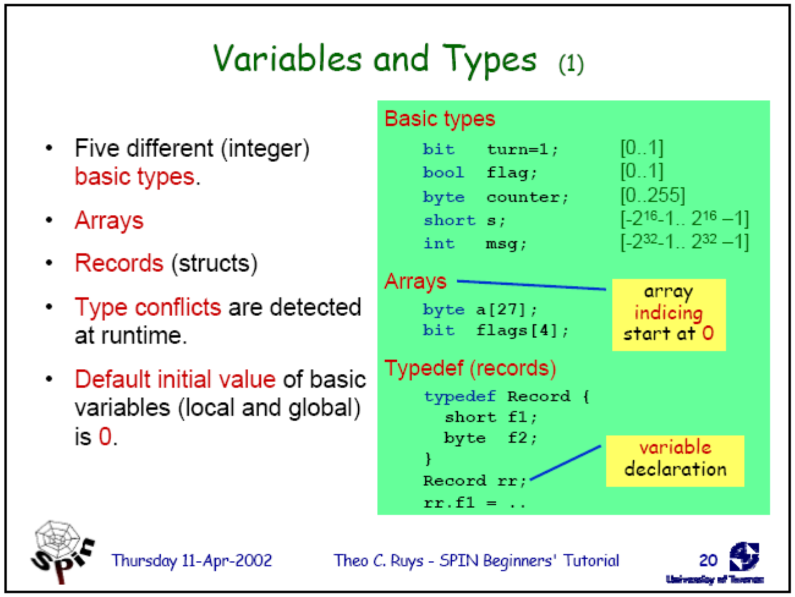
Processes (3)

- 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



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];
```

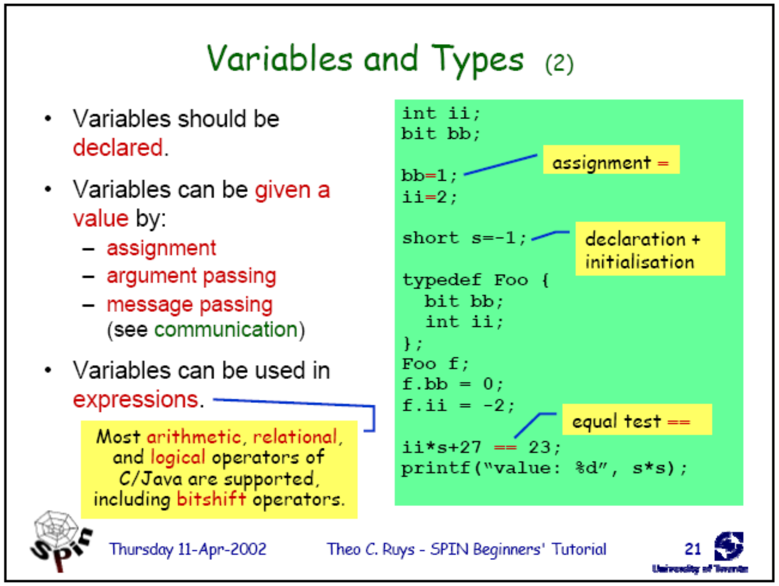
array indicing start at 0

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



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.

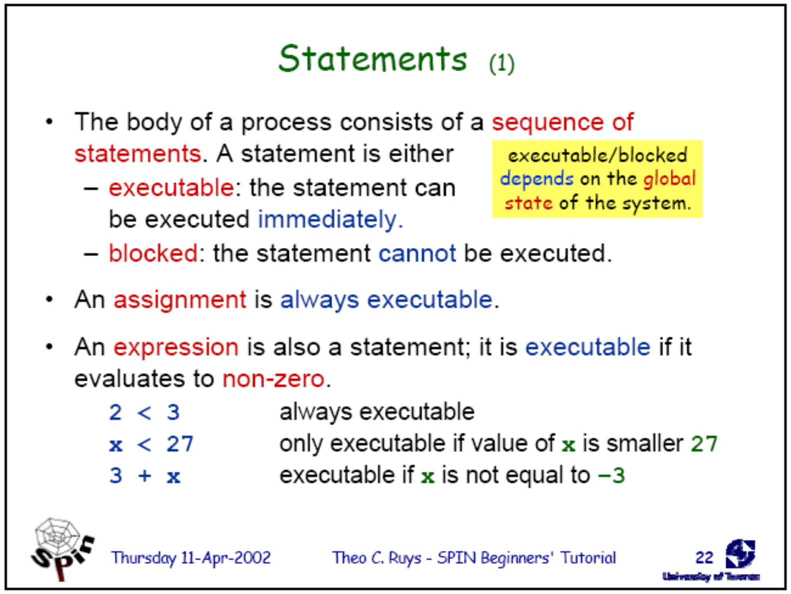
```
int ii;
bit bb;

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 ==

Thursday 11-Apr-2002 Theo C. Ruys - SPIN Beginners' Tutorial 21



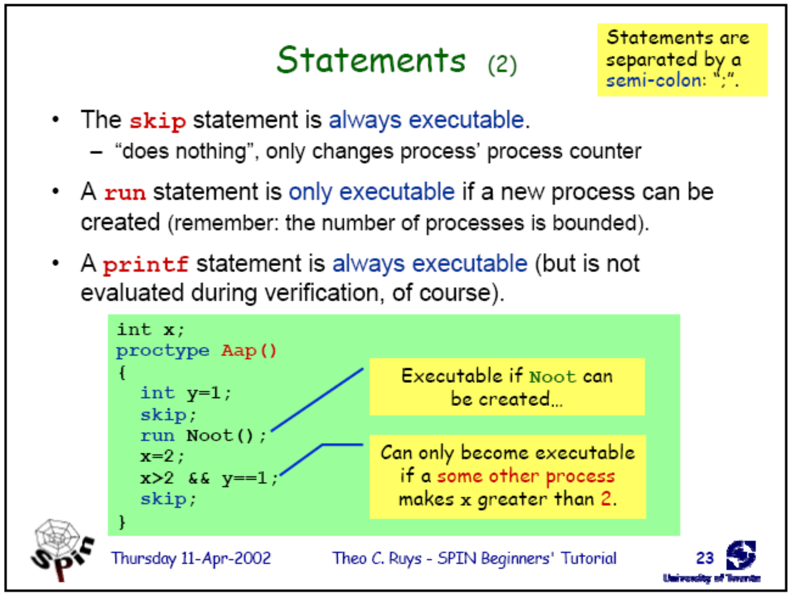
Statements (1)

- 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.
- 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

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

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



Statements (2)

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

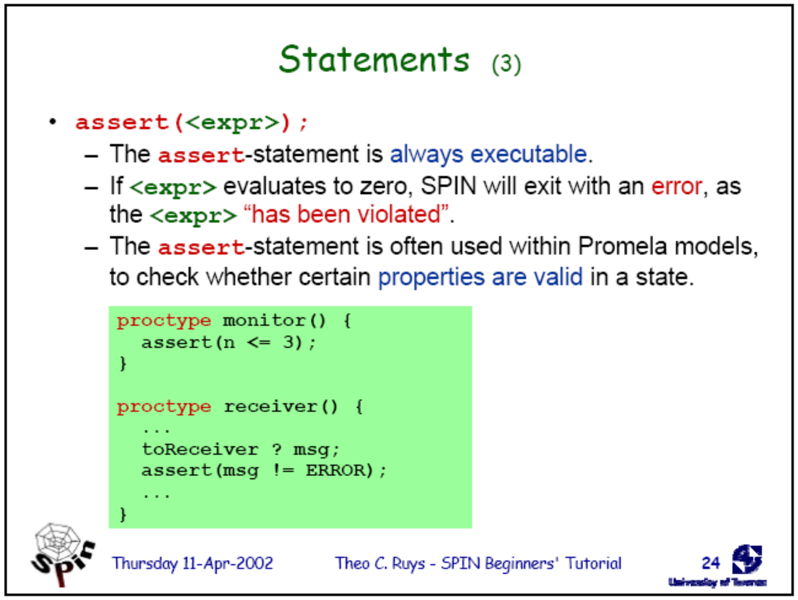
- The `skip` statement is always executable.
  - "does nothing", only changes process' process counter
- A `run` statement is only executable if a new process can be created (remember: the number of processes is bounded).
- A `printf` statement is always executable (but is not evaluated during verification,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

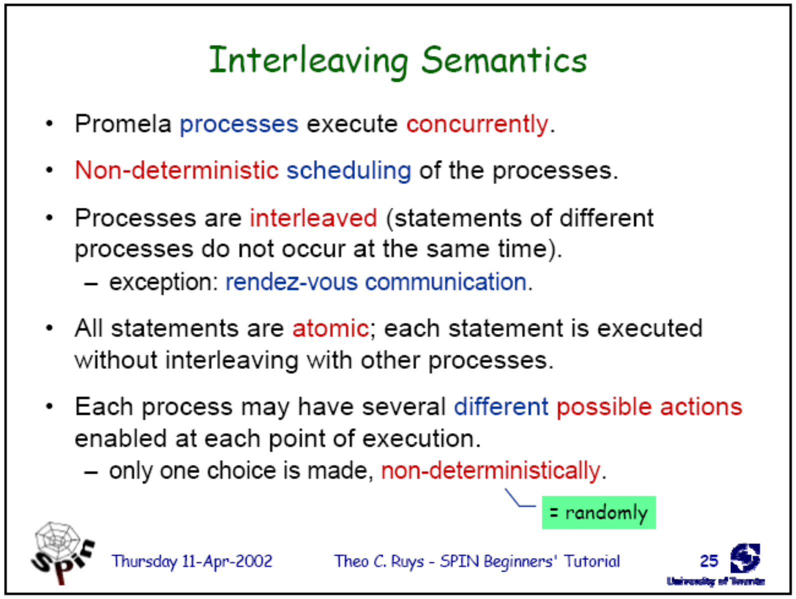
Thursday 11-Apr-2002 Theo C. Ruys - SPIN Beginners' Tutorial 23



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.

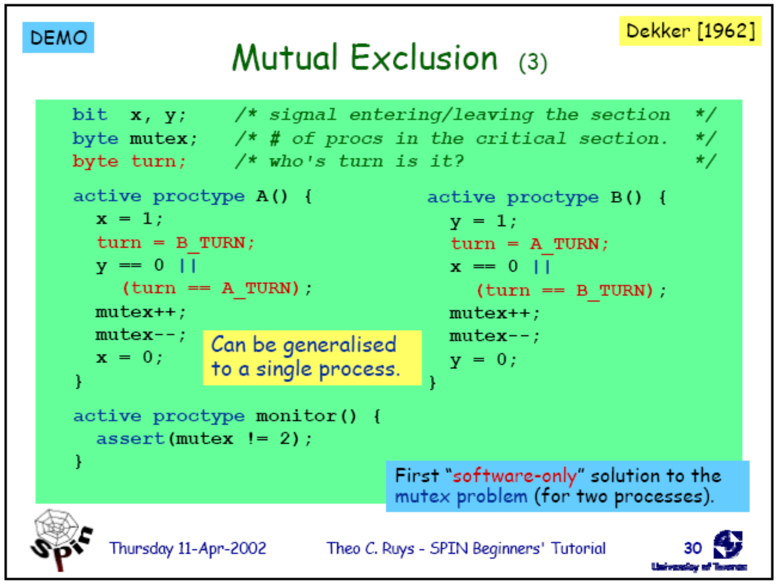
```
proctype monitor() {
  assert(n <= 3);
}

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



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



DEMO

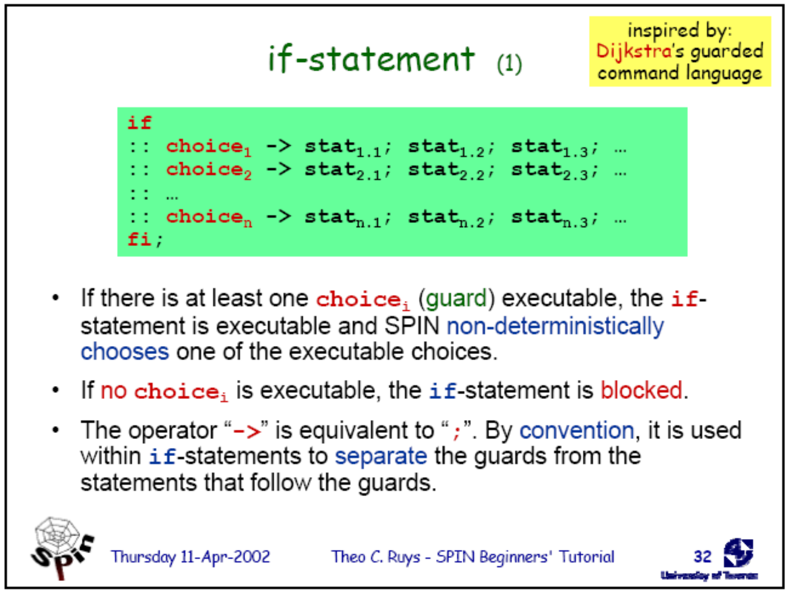
Mutual Exclusion (3)

Dekker [1962]

```
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).

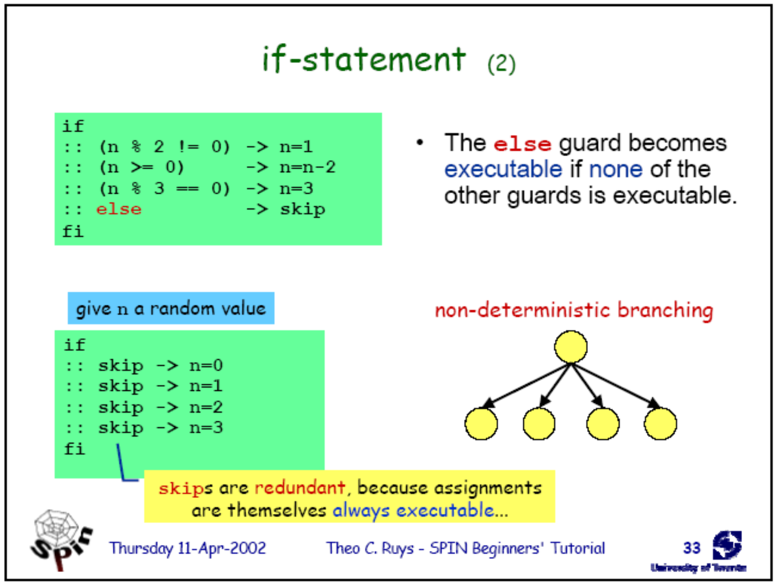


if-statement (1)

inspired by: Dijkstra's guarded command language

```
if
:: choice_1 -> stat_1.1; stat_1.2; stat_1.3; …
:: choice_2 -> stat_2.1; stat_2.2; stat_2.3; …
:: …
:: choice_n -> stat_n.1; stat_n.2; stat_n.3; …
fi;
```

- If there is at least one $\texttt{choice}_i$ (guard) executable, the `if`-statement is executable and SPIN non-deterministically chooses one of the executable choices.
- If no $\texttt{choice}_i$ 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
```

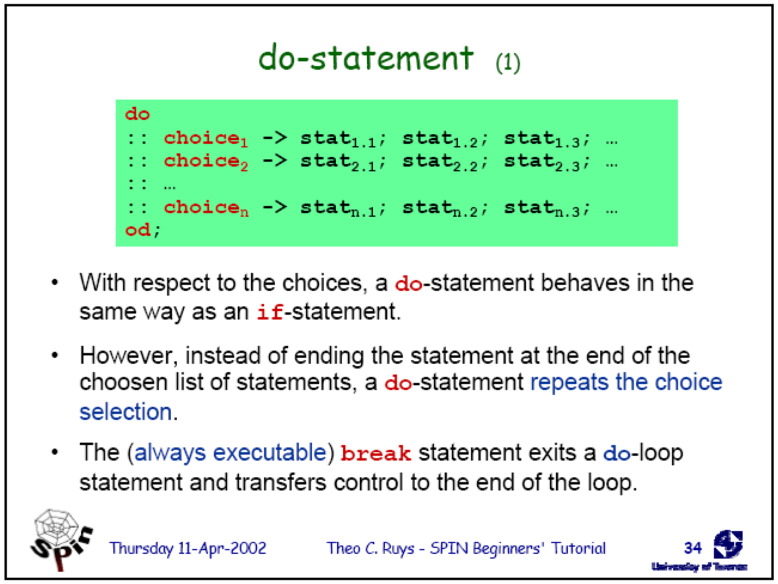
- The else guard becomes executable if none of the other guards is executable.

give n a random value

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

non-deterministic branching

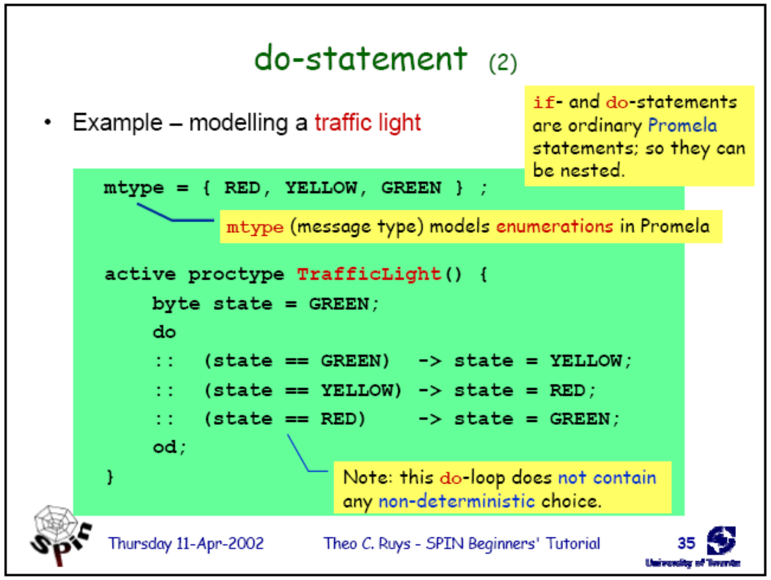
skips are redundant, because assignments are themselves always executable...



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)

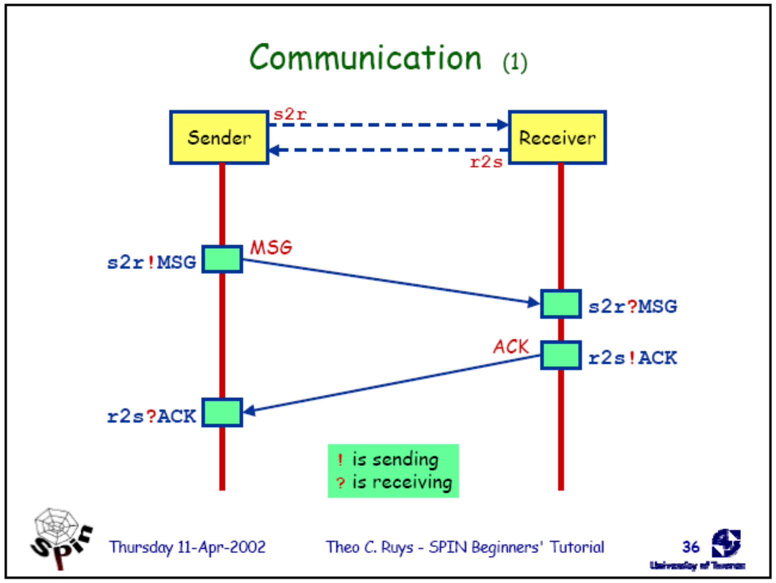
- Example – modelling a traffic light

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.



Communication (1)

Sender

Receiver

s2r

r2s

s2r!MSG

MSG

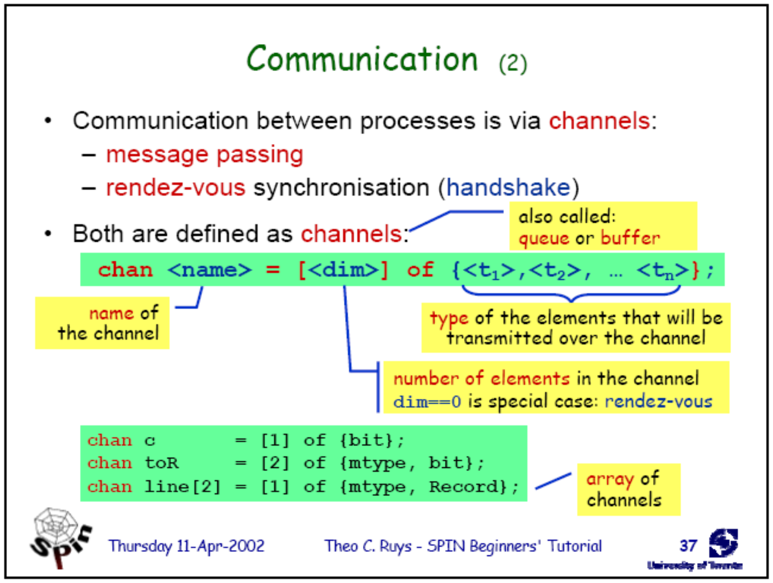
s2r?MSG

ACK

r2s!ACK

r2s?ACK

! is sending
? is receiving



Communication (2)

- Communication between processes is via channels:
  - message passing
  - rendez-vous synchronisation (handshake)
- Both are defined as channels:

also called: queue or buffer

```
chan <name> = [<dim>] of {<t1>,<t2>, … <tn>};
```

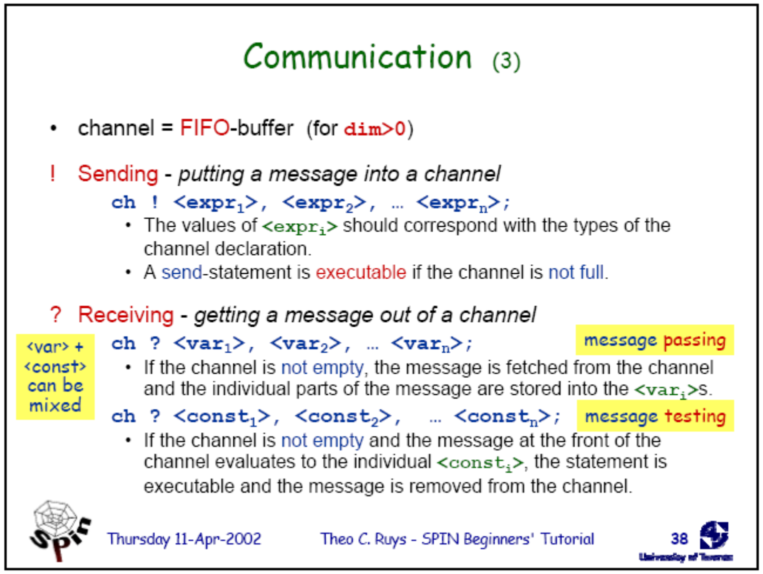
name of the channel

type of the elements that will be transmitted over the channel

number of elements in the channel
dim==0 is special case: rendez-vous

```
chan c       = [1] of {bit};
chan toR     = [2] of {mtype, bit};
chan line[2] = [1] of {mtype, Record};
```

array of channels



Communication (3)

- channel = FIFO-buffer (for **dim>0**)

! Sending - *putting a message into a channel*

ch ! <expr$_1$>, <expr$_2$>, … <expr$_n$>;

- The values of <expr$_i$> should correspond with the types of the channel declaration.
- A send-statement is executable if the channel is not full.

? Receiving - *getting a message out of a channel*

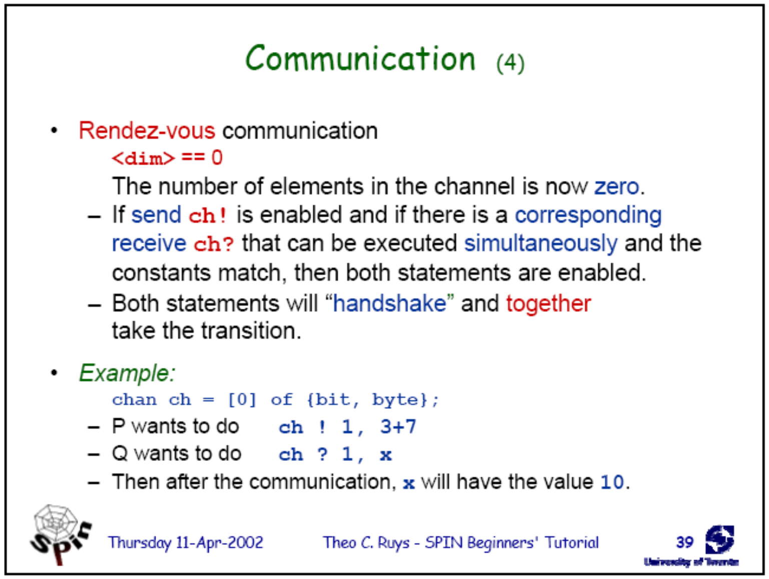
<var> + <const> can be mixed

ch ? <var$_1$>, <var$_2$>, … <var$_n$>; message passing

- If the channel is not empty, the message is fetched from the channel and the individual parts of the message are stored into the <var$_i$>s.

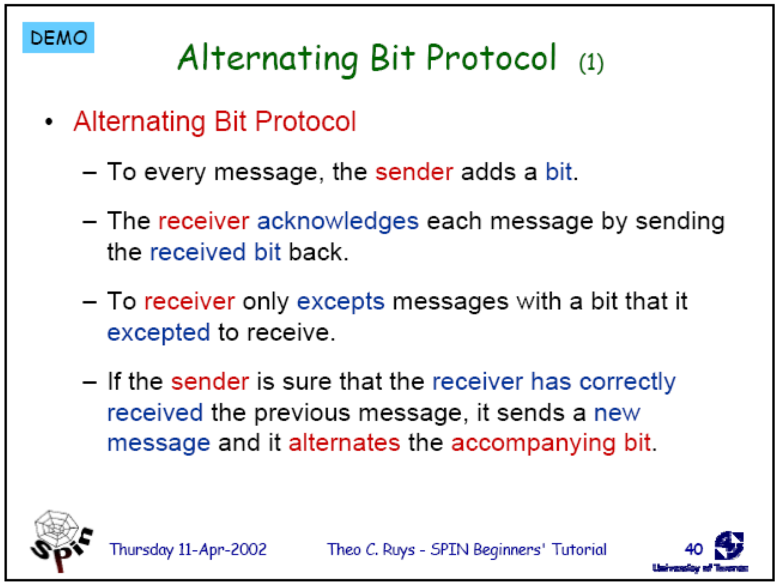
ch ? <const$_1$>, <const$_2$>, … <const$_n$>; message testing

- If the channel is not empty and the message at the front of the channel evaluates to the individual <const$_i$>, the statement is executable and the message is removed from the channel.



Communication (4)

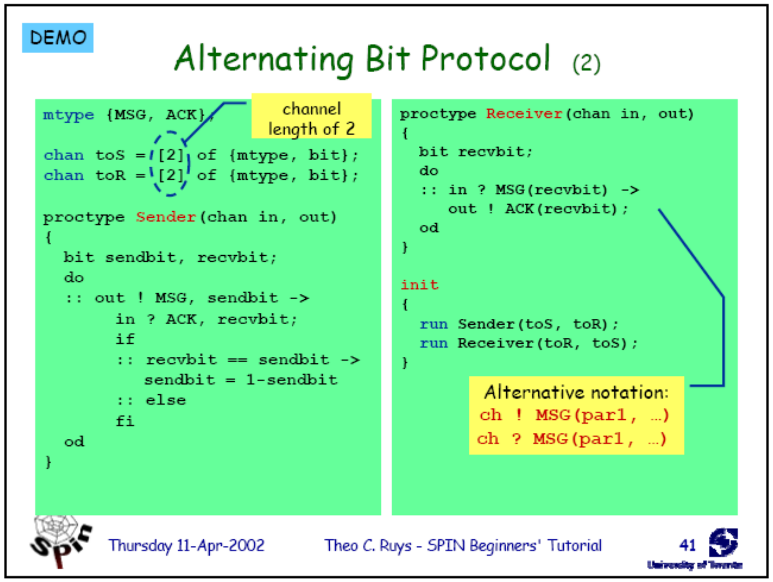
- Rendez-vous communication
  <dim> == 0
  The number of elements in the channel is now zero.
  - If send **ch!** is enabled and if there is a corresponding receive **ch?** that can be executed simultaneously and the constants match, then both statements are enabled.
  - Both statements will “handshake” and together take the transition.
- *Example:*
  `chan ch = [0] of {bit, byte};`
  - P wants to do `ch ! 1, 3+7`
  - Q wants to do `ch ? 1, x`
  - Then after the communication, **x** will have the value **10**.



DEMO

Alternating Bit Protocol (1)

- Alternating Bit Protocol
  - To every message, the sender adds a bit.
  - The receiver acknowledges each message by sending the received bit back.
  - To receiver only excepts messages with a bit that it excepted to receive.
  - 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.



DEMO

Alternating Bit Protocol (2)

```
mtype {MSG, ACK};

chan toS = [2] of {mtype, bit};
chan toR = [2] of {mtype, bit};

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

channel length of 2

```
proctype Receiver(chan in, out)
{
  bit recvbit;
  do
  :: in ? MSG(recvbit) ->
       out ! ACK(recvbit);
  od
}

init
{
  run Sender(toS, toR);
  run Receiver(toR, toS);
}
```

Alternative notation:

```
ch ! MSG(par1, …)
ch ? MSG(par1, …)
```

Thursday 11-Apr-2002 Theo C. Ruys - SPIN Beginners' Tutorial 41

More Promela

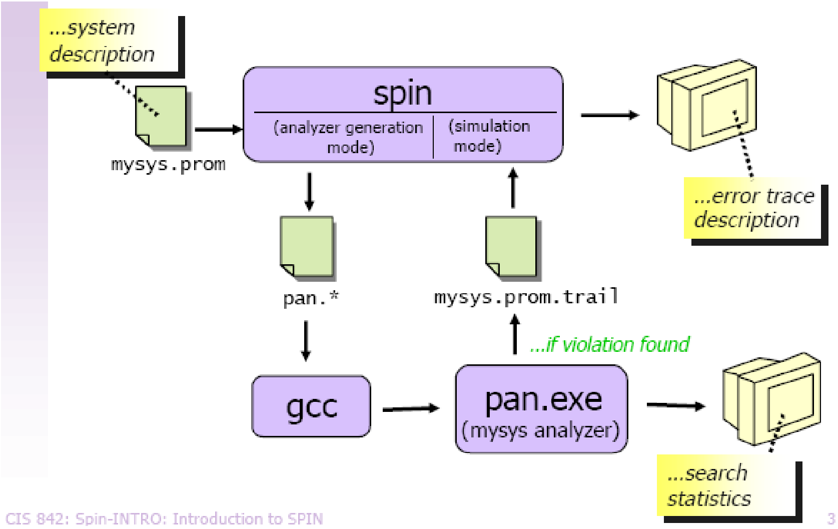
- atomic
  - force sequence of statements to be executed atomically
  - should use as little as possible (why?)
- timeout
  - becomes executable when no other statement is executable
  - note that there’s no time argument
  - should use as little as possible (why?)
- labels
  - for gotos
  - for identifying
    - accepting states: E.g.: accept0: do :: true od
    - end states
    - progress states: E.g.: progress: sendbit = 1-sendbit

} used to express properties (more later)

More Promela (Cont’d)

- macros (cpp preprocessor)
  - #define DEBUG 1
  - #ifdef DEBUG
- All described in
  - G. Holzmann, The Spin Model Checker: Primer and Reference Manual. Addison Wesley. 2003.
  - `www.spinroot.com`

Using Spin

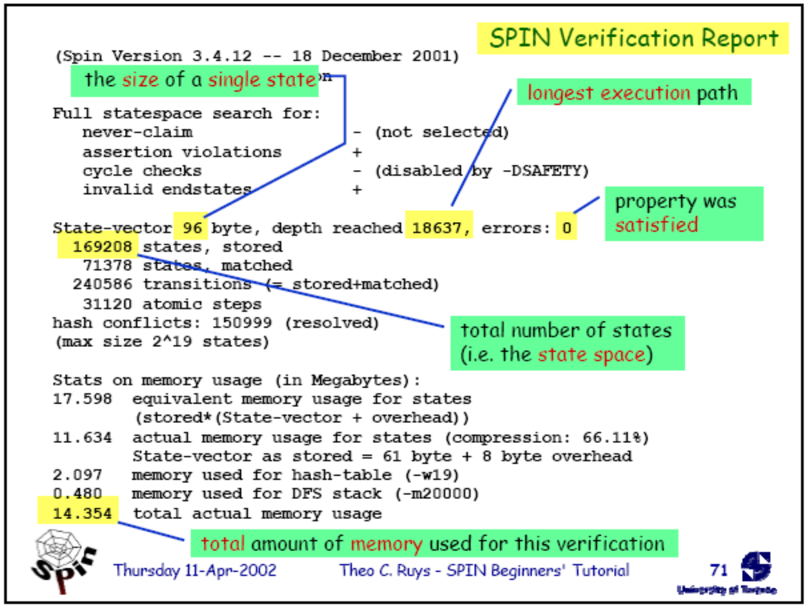


Using Spin (Cont'd)

- `>spin -a mysys.prom`
  - creates dedicated PROMELA analyzer C program (**pan**.*) that implements an exhaustive search on the system described in **mysys.prom**
- `>gcc pan.c -o pan.exe`
  - compiles the analyzer source (**pan.c**) to yield an executable (**pan.exe**)
  - lots of compiler flags
- `>pan.exe`
  - runs the analyzer
  - lots of command-line flags
  - produces **mysys.prom.trail** containing violating trace
- `>spin -t mysys.prom`
  - runs SPIN in simulation mode along the trace in **mysys.prom.trail**
  - prints out diagnostic information

Using Spin (Cont'd)

- Use Spin/XSPIN to
  - check syntax of model: **spin -A model.prom**
  - simulate the model
    - interactively: **spin -p model.prom**
    - randomly: **spin -i -p model.prom**
  - generate verifier: **spin -a model.prom**
  - inspect/display error traces: **spin -t -p model**
- Use verifier to check model for
  - assertion violations
  - deadlock (invalid endstates) (default)
  - non-progress and acceptance cycles
  - complex temporal properties expressed as
    - Never claims
    - Linear Temporal Logic formula



SPIN Verification Report

```
(Spin Version 3.4.12 -- 18 December 2001)

Full statespace search for:
   never-claim              - (not selected)
   assertion violations     +
   cycle checks             - (disabled by -DSAFETY)
   invalid endstates        +

State-vector 96 byte, depth reached 18637, errors: 0
  169208 states, stored
   71378 states, matched
  240586 transitions (= stored+matched)
   31120 atomic steps
hash conflicts: 150999 (resolved)
(max size 2^19 states)

Stats on memory usage (in Megabytes):
17.598  equivalent memory usage for states
        (stored*(State-vector + overhead))
11.634  actual memory usage for states (compression: 66.11%)
        State-vector as stored = 61 byte + 8 byte overhead
2.097   memory used for hash-table (-w19)
0.480   memory used for DFS stack (-m20000)
14.354  total actual memory usage
```

the size of a single state

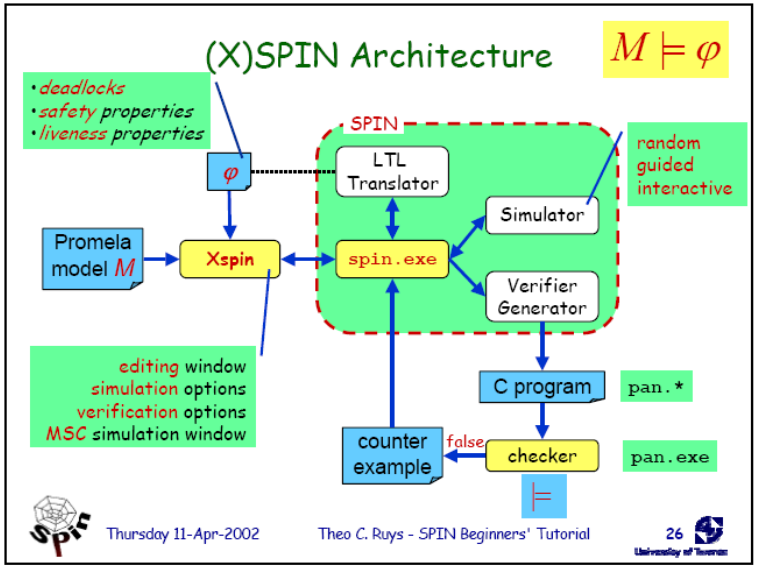
longest execution path

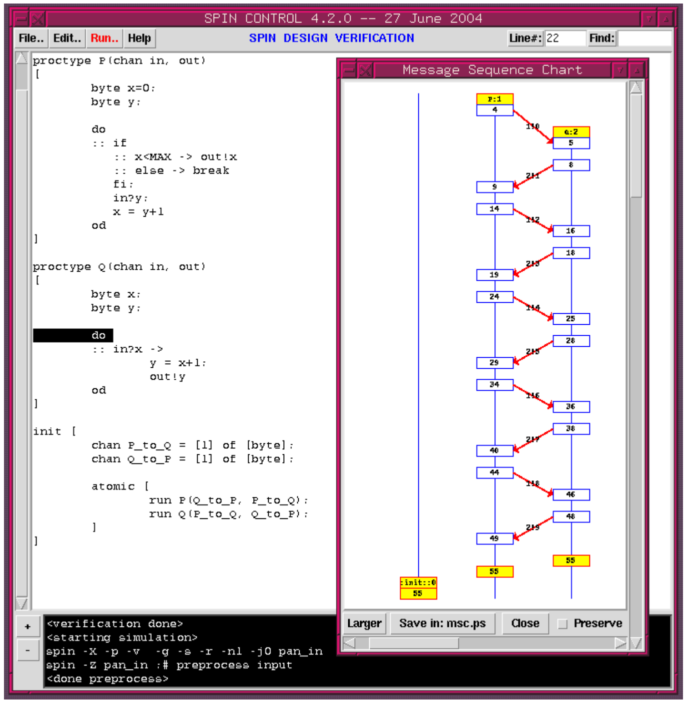
property was satisfied

total number of states (i.e. the state space)

total amount of memory used for this verification

Thursday 11-Apr-2002 Theo C. Ruys - SPIN Beginners' Tutorial 71





```
proctype P(chan in, out)
[
        byte x=0;
        byte y;

        do
        :: if
           :: x<MAX -> out!x
           :: else -> break
           fi;
           in?y;
           x = y+1
        od
]

proctype Q(chan in, out)
[
        byte x;
        byte y;

        do
        :: in?x ->
                y = x+1;
                out!y
        od
]

init [
        chan P_to_Q = [1] of [byte];
        chan Q_to_P = [1] of [byte];

        atomic [
                run P(Q_to_P, P_to_Q);
                run Q(P_to_Q, Q_to_P);
        ]
]
```

```
<verification done>
<starting simulation>
spin -X -p -v  -g -s -r -nl -j0 pan_in
spin -Z pan_in :# preprocess input
<done preprocess>
```

Using XSPIN

XSPIN also generates graphical representation of FSA corresponing to PROMELA model

PROMELA Semantics

Each PROMELA proctype (process) p describes an FSA (S, S_0, L, δ, F) with

- states S: control locations in p
- initial states S_0: {first control location in p}
- labels L: basic statements in p
  - assignments: `x=e`
  - assertions: `assert(b)`
  - print statements: `printf("%d\n", x)`
  - send or receive statements: `c!3` or `c?x`
  - expression statements: `(x==3)`

PROMELA Semantics (Cont'd)

Each PROMELA proctype (process) p describes an FSA (S, S_0, L, δ, F) with

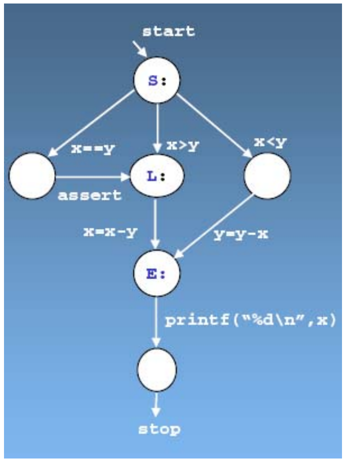
- transition relation δ: Control flow graph of p
- final states F: combination of
  - end states: last location of p and locations labeled with "end"
  - progress states: locations in p labeled with "progress"
  - accepting states: locations in p labeled with "accept"

depending on what we check for (more on this later)

PROMELA Semantics (Cont'd)

For example:

```
active proctype not_euclid()
{
S: if
   :: x == y ->    assert(x != y); goto L
   :: x > y  -> L: x = x - y
   :: x < y  ->    y = y - x
   fi;
E: printf("%d\n", x)
}
```



Note:

- Basic statements change variables
- if, goto, ;, ->, do, break, unless, atomic are not basic statements and are not used as labels

PROMELA Semantic Engine

Semantic engine stores information about

- global variables (e.g., current values)
- message channels (e.g., current contents)
- processes
  - names, types, initial, and current values of local variables
  - current state (i.e., control location)
  - transition relation
    - source and target location of transition
    - enabledness condition and effect of transition

PROMELA Semantic Engine (Cont’d)

- Semantic engine of SPIN constructs PROMELA model (i.e., the iFSA corresponding to the FSA representing the PROMELA program) in step by step manner
- Construction of model and error checking happens at the same time (“on-the-fly” model checking)
- Two basic modes
  - simulation (random, guided, interactive)
  - verification

Random Simulation Algorithm of SPIN’s Semantic Engine

```
while (!error & !allBlocked) {
        ActionList menu = getCurrentExecutableActions();
        allBlocked = (menu.size() == 0);
        if (! allBlocked) {
                Action act = menu.chooseRandom();
                error = act.execute();
        }
}
```

Visit all processes and collect all executable actions

Execute act and make system enter the new state

For interactive simulation: act is chosen by the user

Simplified Verification Algorithm of SPIN’s Semantic Engine

- By default, SPIN uses a depth first search algorithm (DFS) to generate and explore the complete state space
- Can also ask for BFS

```
procedure dfs(s: state) {
  if error(s)  reportError(CurrentPath);
   foreach (successor t of s) {
     if (t not in AlreadySeen) {
        add t to AlreadySeen;
        push(t, CurrentPath);
        dfs(t);
        pop(CurrentPath);
     }
}
```

requires “state matching”

implemented as hash table

stack containing path from initial to current state

More later!

More Info on PROMELA and SPIN

- Gerard Holzmann. The Spin Model Checker: Primer and Reference Manual. Addison Wesley. 2003
  - Chapter 3 (Promela)
  - Chapter 7 (Semantics)
  - Chapter 11 (Using Spin)
  - Chapter 12 (Using Xspin)
- `spinroot.com`
  - `spinroot.com/spin/Man/index.html`
    - Manual pages
    - Basic Spin Manual
    - Guidelines for using Spin and XSPIN
    - Tutorials