



SPIN 2002 Workshop  University of Twente
The Netherlands

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>




Credits should go to ...

- **Gerard Holzmann** (Bell Laboratories)
Developer of SPIN, Basic SPIN Manual. 
- **Radu Iosif** (Kansas State University, USA)
Course: Specification and Verification of Reactive Systems (2001)
- **Mads Dam** (Royal Institute of Technology, Sweden)
Course: Theory of Distributed Systems (2001).
- **Bengt Jonsson** (Uppsala University, Sweden)
Course: Reactive Systems (2001).
- **Joost-Pieter Katoen** (University of Twente)
Course: Protocol/System Validation (2000).





Thursday 11-Apr-2002 Theo C. Ruys - SPIN Beginners' Tutorial 2  University of Twente

Audience & Contents

- **Basic SPIN**
intended audience:
 people **totally new** to (model checking and) SPIN
- **Advanced SPIN**
intended audience:
 people at least at the level of "Basic SPIN"
- **Contents**
 Emphasis is on "using SPIN" not on technical details.
 In fact, we almost regard SPIN as a **black box**.

We just want to "press-the-button".


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


3 
 University of Twente

Common Design Flaws


- **Deadlock**
- **Livelock**, starvation
- **Underspecification**
 - unexpected reception of messages
- **Overspecification**
 - Dead code
- **Violations of constraints**
 - Buffer overruns
 - Array bounds violations
- **Assumptions about speed**
 - Logical correctness vs. real-time performance


In designing **distributed systems**:
 network applications,
 data communication protocols,
 multithreaded code,
 client-server applications.

Designing **concurrent (software) systems** is so hard, that these flaws are mostly overlooked...



Fortunately, most of these design errors can be **detected** using **model checking** techniques!


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

4 
 University of Twente

What is Model Checking?

- [Clarke & Emerson 1981]:
 “Model checking is an **automated** technique that, given a **finite-state model** of a system and a **logical property**, systematically **checks** whether this property holds for (a given initial state in) that model.”
- Model checking tools **automatically** verify whether $M \models \phi$ holds, where M is a (finite-state) **model** of a system and **property** ϕ is stated in some formal notation.
- Problem: **state space explosion!**
- **SPIN** [Holzmann 1991] is one of the most **powerful** model checkers.

Although **finite-state**, the model of a system typically grows **exponentially**.

Based on [Vardi & Wolper 1986].

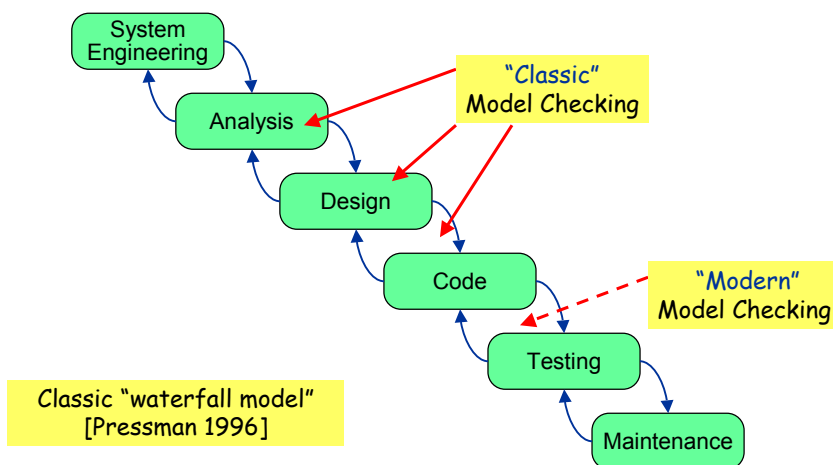


Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



System Development



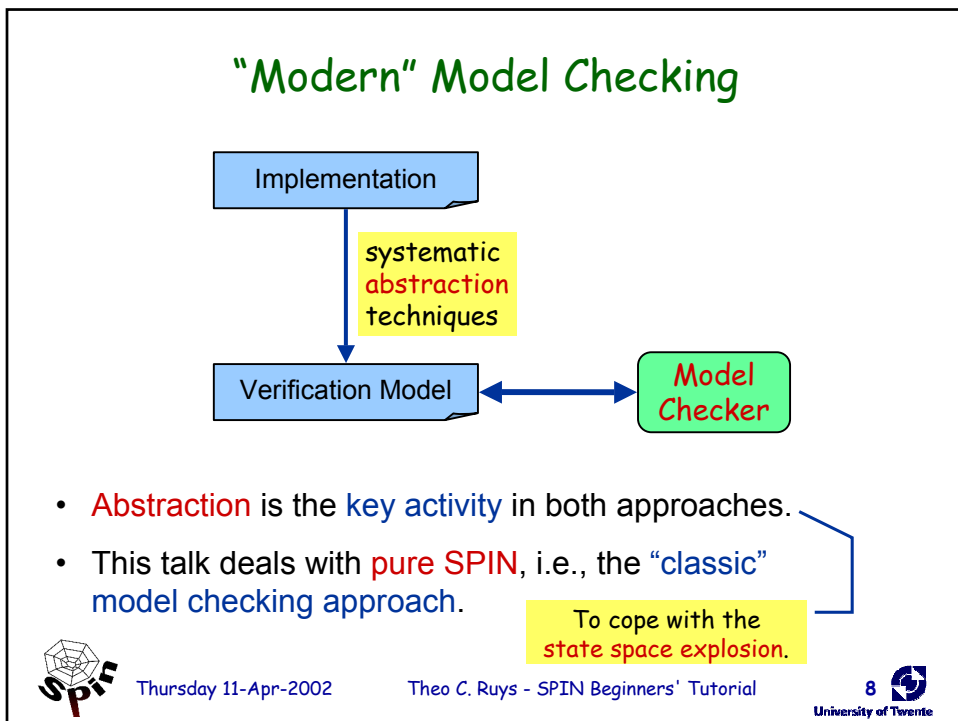
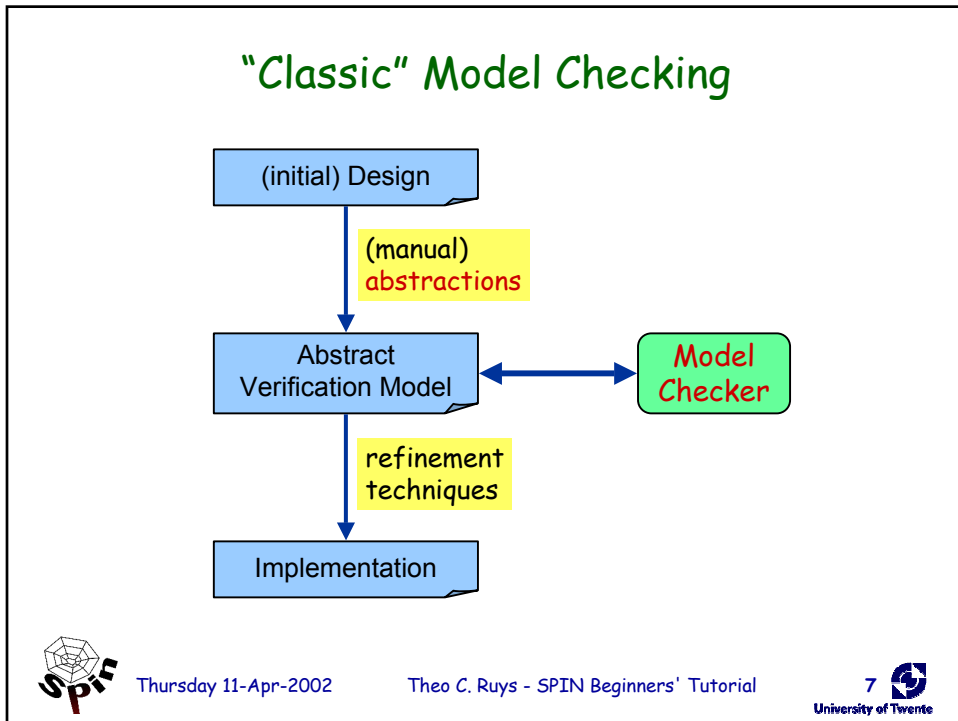
Classic "waterflood model" [Pressman 1996]



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial





Verification vs. Debugging

- Two (extreme) **approaches** with respect to the application of model checkers.
 - **verification** approach: tries to ascertain the correctness of a detailed model M of the system under validation.
 - **debugging** approach: tries to find errors in a model M .
- Model checking is **most effective** in combination with the **debugging** approach.

Automatic verification is *not* about proving correctness, but about **finding bugs** much earlier in the development of a system.



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



9

Program suggestions

- Some presentations at **ETAPS/SPIN 2002** somehow **related** to this tutorial:
 - **Dennis Dams**
Abstraction in Software Model Checking
 - Friday April 12th 10.45-13.00
 - **John Hatcliff, Matthew Dwyer and Willem Visser**
Using the Bandera Tool Set and JPF (Tutorial 10)
 - Saturday April 13th (full day)
 - **SPIN Applications**
 - Saturday April 13th 11.00-12.30

"Modern" model checking approach.



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial




10

Basic SPIN


- **Gentle introduction** to **SPIN** and **Promela**
 - SPIN Background
 - Promela **processes**
 - Promela **statements**
 - Promela **communication** primitives
 - **Architecture** of (X)Spin
 - Some **demo's**: **SPIN** and **Xspin**
 - **hello world**
 - **mutual exclusion**
 - **alternating bit protocol**
 - Cookie for the break

Windows 2000: OK, but
SPIN runs more smoothly
under **Unix/Linux**.



Thursday 11-Apr-2002


Theo C. Ruys - SPIN Beginners' Tutorial

11 
University of Twente

SPIN - Introduction (1)


- **SPIN** (= Simple Promela Interpreter)
 - = is a tool for analysing the logical consistency of **concurrent systems**, specifically of **data communication protocols**.
 - = **state-of-the-art** model checker, used by **>2000 users**
 - Concurrent systems are described in the **modelling language** called **Promela**.
- **Promela** (= Protocol/Process Meta Language)
 - allows for the **dynamic creation** of **concurrent processes**.
 - communication via **message channels** can be defined to be
 - **synchronous** (i.e. rendezvous), or
 - **asynchronous** (i.e. buffered).
 - resembles the programming language **C**
 - specification language to model **finite-state systems**

+ features from CSP



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial

12 
University of Twente

SPIN - Introduction (2)

- Major versions:

1.0	Jan 1991	initial version [Holzmann 1991]
2.0	Jan 1995	partial order reduction
3.0	Apr 1997	minimised automaton representation
4.0	late 2002	Ax: automata extraction from C code

- Some **success factors** of SPIN (subjective!):
 - “press on the button” verification (model checker)
 - very **efficient implementation** (using C)
 - nice **graphical user interface** (Xspin)
 - not just a research tool, but **well supported**
 - contains more than two decades research on **advanced computer aided verification** (many **optimization algorithms**)



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



13

Documentation on SPIN

- SPIN's starting page:

<http://netlib.bell-labs.com/netlib/spin/whatispin.html>

- **Basic SPIN manual**
- Getting started with **Xspin**
- Getting started with **SPIN**
- Examples and **Exercises**
- Concise Promela Reference (by **Rob Gerth**)
- Proceedings of all **SPIN Workshops**

Also part of SPIN's documentation distribution (file: **html.tar.gz**)

- Gerard Holzmann's** website for papers on SPIN:

<http://cm.bell-labs.com/cm/cs/who/gerard/>

- SPIN version 1.0 is described in [Holzmann 1991].



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



14

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



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



15

Processes (1)

- 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
}
    
```

name

formal parameters

local variables

body

The body consist of a sequence of statements.



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



16

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**



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



17

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



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



18

DEMO

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);
}
    
```

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

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

19

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;            [-216-1.. 216-1]
int    msg;          [-232-1.. 232-1]
            
```

Arrays

```

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

Typedef (records)

```

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

array indexing start at 0

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);
    
```



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial


21 

University of Twente

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



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**.


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

23 
 University of Twente


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);
    ...
}
            
```


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

24 
 University of Twente

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**.

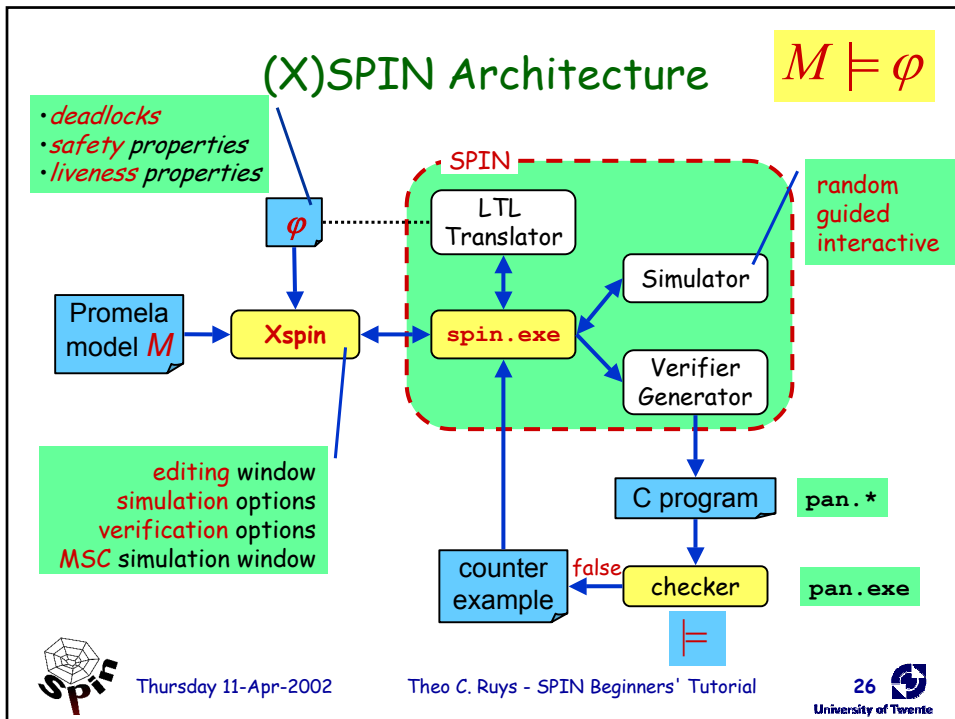
= randomly



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial

25 University of Twente



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial

26 University of Twente

Xspin in a nutshell

- **Xspin** allows the user to
 - **edit** Promela models (+ syntax check)
 - **simulate** Promela models
 - random
 - interactive
 - guided
 - **verify** Promela models
 - exhaustive
 - bitstate hashing mode
 - additional **features**
 - Xspin suggest **abstractions** to a Promela model (**slicing**)
 - Xspin can **draw automata** for each process
 - **LTl property manager**
 - **Help system** (with verification/simulation **guidelines**)

with dialog boxes to set various **options** and **directives** to **tune** the verification process



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



27

University of Twente

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



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



28

University of Twente

DEMO
WRONG!

Mutual Exclusion (2)

```

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



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

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


Thursday 11-Apr-2002
Theo C. Ruys - SPIN Beginners' Tutorial
29 
University of Twente

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

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


Thursday 11-Apr-2002
Theo C. Ruys - SPIN Beginners' Tutorial
30 
University of Twente

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



inspired by:
Dijkstra's guarded
command language


if-statement (1)

```

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.


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

32


if-statement (2)

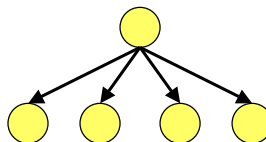
```
if
:: (n % 2 != 0) -> n=1
:: (n >= 0) -> n=n-2
:: (n % 3 == 0) -> n=3
:: else -> skip
fi
```

- 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**...



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial

33 University of Twente

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 chosen 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.



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial

34 University of Twente

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 } ;
```

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**.

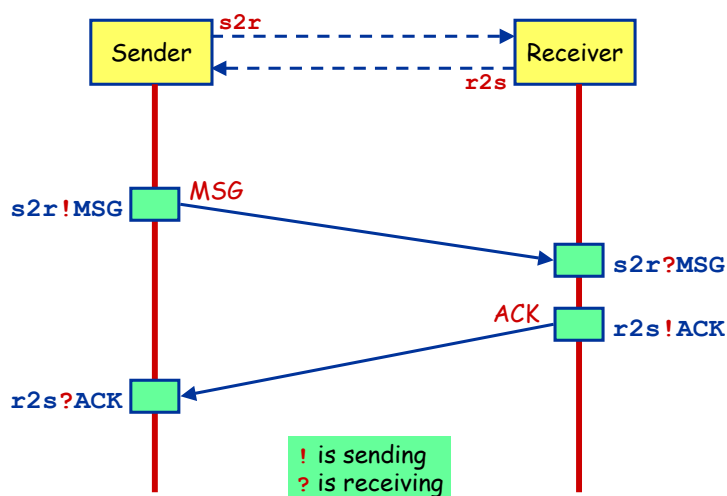


Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial

35 University of Twente

Communication (1)



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial

36 University of Twente

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>};
```



Annotations for the channel declaration:

- name of the channel**: points to `<name>`
- number of elements in the channel**: points to `<dim>`
`dim==0` is special case: **rendez-vous**
- type of the elements that will be transmitted over the channel**: points to `{<t1>, <t2>, ... <tn>}`

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

Annotation for the examples:

- array of channels**: points to `line[2]`


Thursday 11-Apr-2002
Theo C. Ruys - SPIN Beginners' Tutorial
37 
University of Twente

Communication (3)

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

! Sending - putting a message into a channel

```
ch ! <expr1>, <expr2>, ... <exprn>;
```

- The values of `<expri>` 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> + ch ? <var1>, <var2>, ... <varn>;
```



Annotation: **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 `<vari>`s.

```
<const> can be mixed ch ? <const1>, <const2>, ... <constn>;
```

Annotation: **message testing**

- If the channel is **not empty** and the message at the front of the channel evaluates to the individual `<consti>`, the statement is executable and the message is removed from the channel.


Thursday 11-Apr-2002
Theo C. Ruys - SPIN Beginners' Tutorial
38 
University of Twente

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**.



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



39

University of Twente

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**.



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



40

University of Twente

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, rcvbit;
  do
  :: out ! MSG, sendbit ->
    in ? ACK, rcvbit;
    if
  :: rcvbit == sendbit ->
      sendbit = 1-sendbit
  :: else
    fi
  od
}
        
```


```

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

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


channel length of 2

Alternative notation:
 ch ! MSG(par1, ...)
 ch ? MSG(par1, ...)



Thursday 11-Apr-2002

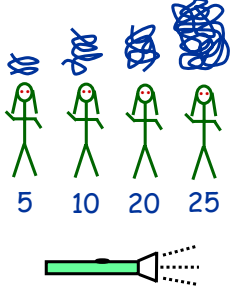
Theo C. Ruys - SPIN Beginners' Tutorial

41 
University of Twente

[Ruys & Brinksma 1998]

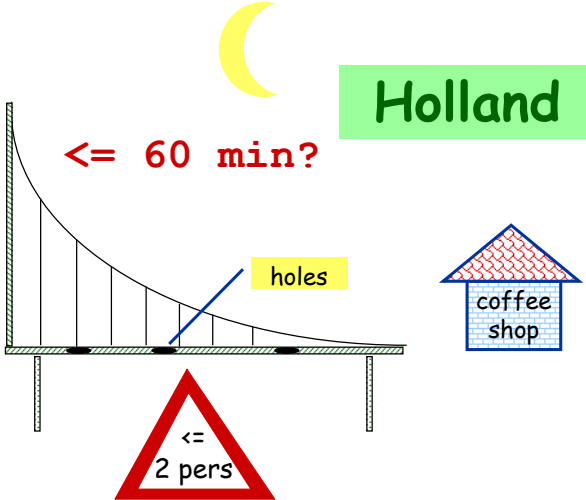
Cookie: "hippies" problem

Germany




5 10 20 25


<= 60 min?



Holland




coffee shop



Thursday 11-Apr-2002

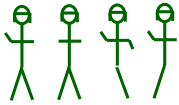
Theo C. Ruys - SPIN Beginners' Tutorial

42 
University of Twente

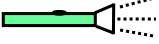
[Ruys & Brinksma 1998]

Cookie: soldiers problem

unsafe

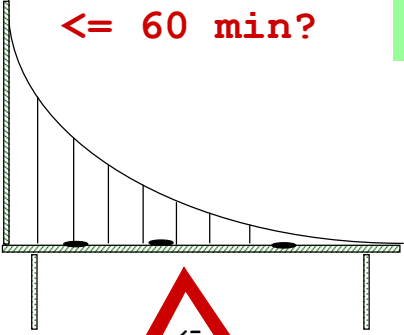



5 10 20 25



<= 60 min?


safe





Thursday 11-Apr-2002


Theo C. Ruys - SPIN Beginners' Tutorial

43 

University of Twente


Advanced SPIN

- Towards **effective modelling** in Promela
 - Some **left-over Promela statements**
 - **Properties** that can be verified with SPIN
 - Introduction to SPIN **validation algorithms**
 - SPIN's **reduction algorithms**
 - **Extreme modelling**: the “art of modelling”
 - **Beyond Xspin**: managing the verification trajectory
 - Concluding remarks
 - Summary



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial


44 


University of Twente

Basic SPIN

Promela Model

- A **Promela** model consist of:
 - **type** declarations mtype, typedefs, constants
 - **channel** declarations chan ch = [dim] of {type, ...}
asynchronous: dim > 0
rendez-vous: dim == 0
 - **global variable** declarations can be accessed by all processes
 - **process** declarations behaviour of the processes: local variables + statements
 - [**init** process] initialises variables and starts processes


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


45

 University of Twente


Basic SPIN

Promela statements

are either **executable** or **blocked**

skip	always executable
assert (<expr>)	always executable
expression	executable if not zero
assignment	always executable
if	executable if at least one guard is executable
do	executable if at least one guard is executable
break	always executable (exits do -statement)
send (ch!)	executable if channel ch is not full
receive (ch?)	executable if channel ch is not empty


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

46

 University of Twente

atomic

atomic { stat₁; stat₂; ... 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 **stat₁** 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;
}
```



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



47

d_step

d_step { stat₁; stat₂; ... stat_n }

- more **efficient** version of **atomic**: **no intermediate states** are generated and stored
 - may only contain **deterministic** steps
 - it is a **run-time error** if **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



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



48


```

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

No atomicity

P1

P2

Not completely correct as each process has an implicit end-transition...

Thursday 11-Apr-2002
Theo C. Ruys - SPIN Beginners' Tutorial
49

```

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.

Thursday 11-Apr-2002
Theo C. Ruys - SPIN Beginners' Tutorial
50

```

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

d_step

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

51

Checking for pure atomicity

- Suppose we want to check that **none** of the atomic clauses in our model are **ever blocked** (i.e. **pure atomicity**).

1. Add a global bit variable:

```
bit aflag;
```

3. Check that aflag is always 0.

```
[!]aflag
```

e.g.

```
active process monitor {
  assert(!aflag);
}
```

2. Change all atomic clauses to:

```
atomic {
  stat1;
  aflag=1;
  stat2
  ...
  statn
  aflag=0;
}
```

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

52

timeout (1)

- Promela does **not** have **real-time** features.
 - In Promela we can only specify **functional behaviour**.
 - Most protocols, however, use **timers** or a **timeout** mechanism to **resend** messages or acknowledgements.
- timeout**
 - SPIN's **timeout** becomes **executable** if there is **no other process** in the system which is executable
 - so, **timeout** models a **global timeout**
 - timeout** provides an **escape** from **deadlock states**
 - beware** of **statements** that are always executable...



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



53

University of Twente

timeout (2)

- Example to **recover** from **message loss**:

```
active proctype Receiver()
{
    bit rcvbit;
    do
        :: toR ? MSG, rcvbit -> toS ! ACK, rcvbit;
        :: timeout          -> toS ! ACK, rcvbit;
    od
}
```

- Premature timeouts** can be modelled by replacing the **timeout** by **skip** (which is always executable).

One might want to limit the number of premature timeouts (see [Ruys & Langerak 1997]).



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial




54

University of Twente

DEMO


Alternating Bit Protocol (3)

- **abp-1.pr**
 - perfect lines
- **abp-2.pr**
 - **stealing daemon** (models **lossy channels**)
 - how do we know that the protocol works correctly?
- **abp-3.pr**
 - model different messages by a **sequence number**
 - **assert** that the protocol works correctly
 - how can we be sure that **different messages** are being transmitted?



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial

55 
University of Twente

How large should **MAX** be such that we are sure that the ABP works correctly?

only three!

goto label

- **transfers** execution to **label**
- each Promela statement might be labelled
- quite useful in modelling **communication protocols**


```

wait_ack:
  if
  :: B?ACK -> ab=1-ab ; goto success
  :: ChunkTimeout?SHAKE ->
    if
    :: (rc < MAX) -> rc++; F!(i==1), (i==n), ab,d[i];
                    goto wait_ack
    :: (rc >= MAX) -> goto error
    fi
  fi ;
    
```

Timeout modelled by a channel.


Part of model of BRP

goto



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial

56 
University of Twente

unless

```
{ <stats> } unless { guard; <stats> }
```

- Statements in *<stats>* are executed **until** the first statement (*guard*) in the escape sequence becomes **executable**.
- resembles **exception handling** in languages like Java
- *Example:*

```
proctype MicroProcessor() {
  {
    ...
    /* execute normal instructions */
  }
  unless { port ? INTERRUPT; ... }
}
```



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



57

macros - **cpp** preprocessor

- Promela uses **cpp**, the **C preprocessor** to preprocess Promela models. This is useful to define:

- **constants**

```
#define MAX 4
```

All **cpp** commands start with a **hash**:
#define, #ifdef, #include, etc.

- **macros**

```
#define RESET_ARRAY(a) \
  d_step { a[0]=0; a[1]=0; a[2]=0; a[3]=0; }
```

- **conditional** Promela model fragments

```
#define LOSSY 1
...
#ifdef LOSSY
  active proctype Daemon() { /* steal messages */ }
#endif
```



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



58

inline - poor man's procedures

- Promela also has its own **macro-expansion** feature using the **inline**-construct.

```

inline init_array(a) {
  d_step {
    i=0;
    do
      :: i<N -> a[i] = 0; i++
      :: else -> break
    od;
    i=0;
  }
}
    
```

Should be declared somewhere else (probably as a local variable).

Be sure to **reset** temporary variables.

- error messages are more **useful** than when using **#define**
- **cannot** be used as **expression**
- all **variables** should be **declared somewhere else**



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



59

Properties (1)

- Model checking tools **automatically** verify whether $M \models \phi$ holds, where M is a (finite-state) **model** of a system and **property** ϕ is stated in some formal notation.
- With SPIN one may **check** the following type of properties:
 - **deadlocks** (invalid endstates)
 - **assertions**
 - **unreachable code**
 - **LTL formulae**
 - **liveness** properties
 - non-progress cycles (livelocks)
 - acceptance cycles



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial





60

Historical Classification

Properties (2)

<p>safety property</p> <ul style="list-style-type: none"> - "nothing bad ever happens" - invariant <i>x is always less than 5</i> - deadlock freedom <i>the system never reaches a state where no actions are possible</i> - SPIN: find a trace leading to the "bad" thing. If there is not such a trace, the property is satisfied. 	<p>liveness property</p> <ul style="list-style-type: none"> - "something good will eventually happen" - termination <i>the system will eventually terminate</i> - response <i>if action X occurs then eventually action Y will occur</i> - SPIN: find a (infinite) loop in which the "good" thing does not happen. If there is not such a loop, the property is satisfied.
--	--


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

61



University of Twente


Properties (3)

- LTL formulae are used to specify liveness properties.
LTL \equiv **propositional logic** + **temporal operators**
 - **[] P** always P
 - **<> P** eventually P
 - **P U Q** P is true until Q becomes true
- Some LTL patterns

Xspin contains a special "LTL Manager" to edit, save and load LTL properties.

 - invariance [] (p)
 - response [] ((p) -> (<> (q)))
 - precedence [] ((p) -> ((q) U (r)))
 - objective [] ((p) -> <>((q) || (r)))


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

62


University of Twente

Properties (4)

- Suggested **further reading** (on temporal properties):

[Bérard et. al. 2001]

- Textbook on **model checking**.
- One part of the book (six chapters) is devoted to “**Specifying with Temporal Logic**”.
- Also available in **French**.

[Dwyer et. al. 1999]

- **classification** of temporal logic properties
- **pattern-based** approach to the **presentation, codification** and **reuse** of property specifications for finite-state verification.

Note: although this tutorial focuses on how to construct an **effective Promela model M**, the definition of the **set of properties** which are to be verified is **equally important!**



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



63

DEMO

Solution to the Hippies problem (1)

```
chan germany_to_holland = [0] of {hippie, hippie} ;
chan holland_to_germany = [0] of {hippie} ;
chan stopwatch         = [0] of {hippie} ;
byte time ;
...
```

A hippie is a byte.

```
proctype Germany()
{
  bit      here[N] ;
  hippie   h1, h2 ;
  here[0]=1; here[1]=1; here[2]=1; here[3]=1;
  do
  :: select_hippie(h1) ;
     select_hippie(h2) ;
     germany_to_holland ! h1, h2 ;
     IF all_gone -> break FI ;
     holland_to_germany ? h1 ;
     here[h1] = 1 ;
     stopwatch ! h1 ;
  od
}
```

Process "Holland" is the dual of "Germany".

It can be modelled more effectively See [Ruys 2001] for directions.



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



64

Solution to the Hippies problem (2)

```

proctype Timer()
{
end:
do
:: stopwatch ? 0 -> atomic { time=time+5 ; MSCTIME }
:: stopwatch ? 1 -> atomic { time=time+10; MSCTIME }
:: stopwatch ? 2 -> atomic { time=time+20; MSCTIME }
:: stopwatch ? 3 -> atomic { time=time+25; MSCTIME }
od
}
init {
atomic { run Germany(); run Holland(); run Timer(); }
}
    
```

Now we should check:
 <> (time>60)



(random) Simulation Algorithm

```

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

deadlock \equiv allBlocked

interactive simulation:
 act is chosen by the user

act is executed and the
 system enters a new state

Visit all processes and collect
 all executable actions .



Verification Algorithm (1)

- SPIN uses a **depth first search** algorithm (DFS) to generate and explore the **complete state space**.

```

procedure dfs(s: state) {
  if error(s)
    reportError();
  foreach (successor t of s) {
    if (t not in Statespace)
      dfs(t);
  }
}
```

Only works for state properties.

states are stored in a hash table

requires state matching

the old states s are stored on a stack, which corresponds with a complete execution path

- Note that the **construction** and **error checking** happens at the same time: SPIN is an **on-the-fly** model checker.

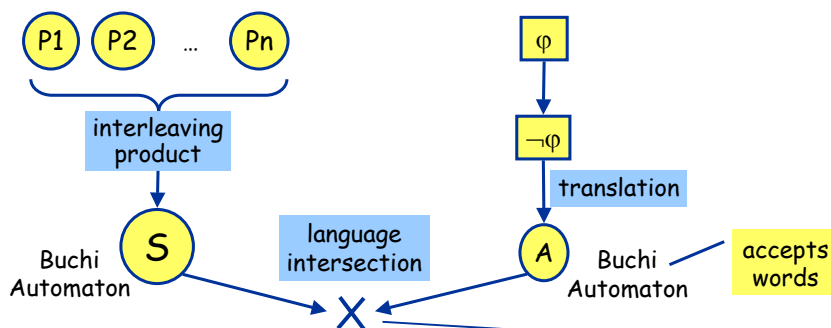


Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



Verification Algorithm (2)



X should be **empty**.
 Search for an **accepting state** in the intersection, which is reachable from itself. In SPIN this is implemented by **two basic DFS** procedures. See [Holzmann 1996 et. al. - DFS] for details.

Based on [Vardi & Wolper 1986].



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



State vector

- A **state vector** is the information to uniquely identify a **system state**; it contains:
 - **global variables**
 - contents of the **channels**
 - for each **process** in the system:
 - **local variables**
 - **process counter** of the process
- It is important to **minimise** the **size** of the **state vector**.

state vector = m bytes
 state space = n states → storing the state space may require $n * m$ bytes

SPIN provides several algorithms to **compress** the state vector.

[Holzmann 1997 - State Compression]



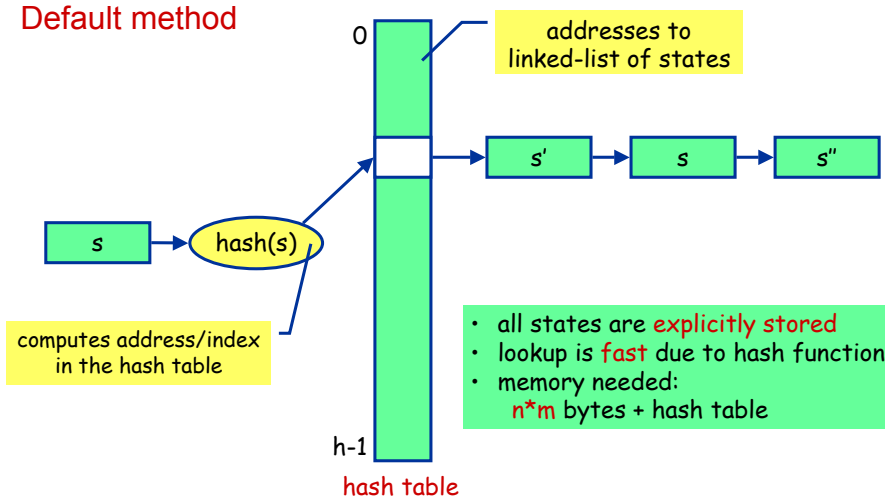
Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial

69 University of Twente

Storing States in SPIN

Default method



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial

70 University of Twente

SPIN Verification Report

```

(Spin Version 3.4.12 -- 18 December 2001)
the size of a single state
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
  
```

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 

Reduction Algorithms (1)

- SPIN has several **optimisation algorithms** to make verification runs more **effective**:
 - partial order reduction
 - bitstate hashing
 - minimised automaton encoding of states (not in a hashtable)
 - state vector compression
 - dataflow analysis
 - slicing algorithm

SPIN's **power** (and **popularity**) is based on these (default) optimisation/reduction algorithms.

SPIN supports several **command-line options** to select and further tune these **optimisation algorithms**.

See for instance: Xspin → Run → Set Verification Parameters → Set Advanced options → Extra Compile-Time Directives




Reduction Algorithms (2)

enabled by default


- **Partial Order Reduction** [Holzmann & Peled 1995 - PO]
 - **observation**: the validity of a property φ is often **insensitive** to the **order** in which concurrent and **independently** executed events are interleaved
 - **idea**: if in some global state, a process P can execute only “**local**” statements, then all other processes may be deferred until later
 - **local statements**, e.g.:
 - statement accessing only **local variables**
 - **receiving** from a queue, from which **no** other process **receives**
 - **sending** to a queue, to which **no** other process **sends**

It is hard to determine exclusive access to channels:
 let user **annotate** exclusive channels with **xs** or **xs**.



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial

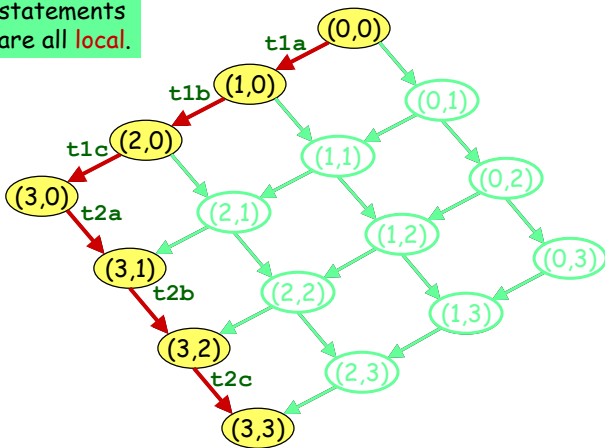
73 


University of Twente

Reduction Algorithms (3)

- **Partial Order Reduction** (cont.)


Suppose the statements of P1 and P2 are all **local**.





Thursday 11-Apr-2002


Theo C. Ruys - SPIN Beginners' Tutorial

74 

University of Twente


Reduction Algorithms (3) approximation

- **Bit-state hashing** [Holzmann 1998 - Bitstate hashing]
 - instead of storing each state explicitly, only **one bit** of memory are used to store a **reachable state**
 - given a state, a **hash function** is used to compute the **address** of the bit in the **hash table**
 - **no collision detection**
 - **hash factor** = # available bits / # reached states
 - aim for hash factor > 100
- **Hash-compaction** [Holzmann 1998 - Bitstate hashing]
 - **large** hash table: 2^{64}
 - store **address** in regular (smaller) hash table
 - **with collision detection**



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial


75 

University of Twente

Reduction Algorithms (4)

- **Bit-state hashing** (cont.)


s → **hash(s)**



The hash table only holds bits: **0** or **1**.


If $hash(s) = n$ and $h[n] == 1$, SPIN concludes that **s** has already been visited.

- states are **not stored** explicitly
- lookup is **fast** due to hash function
- memory needed: **hash table** (only)



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial

76 

University of Twente

Reduction Algorithms (5)

- **State compression** [Holzmann 1997 - State Compression]
 - instead of storing a state explicitly, a **compressed version** of the state is stored in the state space
- **Minimised automaton** [Holzmann & Puri 1999 - MA]
 - states are stored in a dynamically changing, **minimised** deterministic finite automaton (DFA)
 - **inserting/deleting** a state **changes** the DFA
 - close relationship with OBDDs
- **Static analysis algorithms**
 - **slicing algorithm**: to get hints for possible reductions
 - **data-flow** optimisations, **dead variable** elimination, **merging** of safe and atomic **statements**

very memory effective, ... but slow.



Thursday 11-Apr-2002

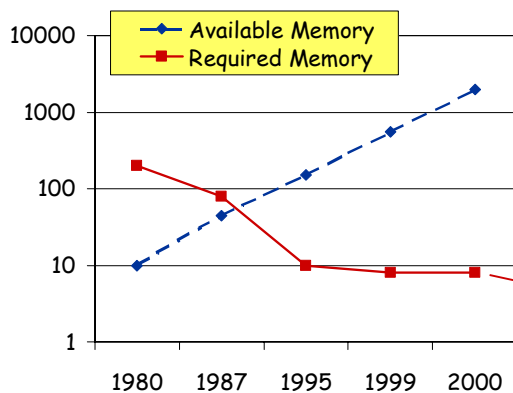
Theo C. Ruys - SPIN Beginners' Tutorial



Moore's Law & Advanced Algorithms

[Holzmann 2000 M'dorf]

- Verification results of **Tpc** (The phone company)



1980: pan
1987: bitstate hashing
1995: partial order reduction
1999: minimised automaton

memory requirements to (fully) verify Tpc



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



BRP - Effective Modelling

- **BRP** = Bounded Retransmission Protocol
 - alternating bit protocol with **timers**
 - 1997: exhaustive verification with **SPIN** and **UPPAAL**
 - 2001: **optimised SPIN** version
 - shows the **effectiveness** of a **tuned model**

	BRP 1997	BRP 2002
state vector	104 bytes	96 bytes
# states	1,799,340	169,208
Memory (Mb)	116.399	14.354

Both verified with SPIN 3.4.x

took upto an hour in 1997



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



79

Recipes in [Ruys 2001]

- Tool Support
- Lossy channels
- First Things First
- Multicast Protocols
- Macros
- Reordering a Promela model
- Atomicity
- **Invariance**
- Randomness
- Still in the pipeline...
- Bitvectors
- Modelling Time in Promela
- Subranges
- Scheduling algorithms
- Abstract Data Types: Deque



Thursday 11-Apr-2002


Theo C. Ruys - SPIN Beginners' Tutorial



80


Invariance [] P

- [] P where P is a **state property**
 - safety property
 - **invariance** ≡ global **universality** or global **absence** [Dwyer et. al. 1999]:
 - 25% of the properties that are being checked with model checkers are **invariance properties**
 - BTW, 48% of the properties are **response properties**
 - examples:
 - [] !aflag
 - [] mutex != 2
- SPIN supports (at least) **7 ways** to **check** for invariance.



Thursday 11-Apr-2002

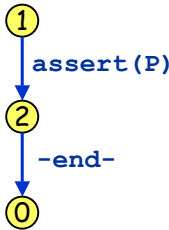
Theo C. Ruys - SPIN Beginners' Tutorial

81 
University of Twente

variant 1+2 - monitor process (single assert) [] P


- **proposed** in SPIN's documentation
- **add** the following **monitor** process to the Promela model:

```
active proctype monitor()
{
  assert(P);
}
```




- Two variations:
 - 1. monitor process is created **first**
 - 2. monitor process is created **last**

If the monitor process is created **last**, the **-end-** transition will be executable after executing **assert(P)**.



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial

82 
University of Twente

variant 3 - guarded monitor process []P

- Drawback of solution “1+2 monitor process” is that the **assert** statement is enabled in **every** state.


```
active proctype monitor()
{
  assert(P) ;
}
```

➔

```
active proctype monitor()
{
  atomic {
    !P -> assert(P) ;
  }
}
```


- The **atomic** statement **only** becomes **executable** when **P** itself is **not** true.

We are searching for a state where **P** is not true. If it does not exist, **[]P** is true.



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial


83 

University of Twente


variant 4 - monitor process (do assert) []P

- From an operational viewpoint, the following monitor process **seems less effective**:

```
active proctype monitor()
{
  do
  :: assert(P)
  od
}
```




- But the number of **states** is clearly advantageous.



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial

84 

University of Twente

[] P

variant 5 - never claim (do assert)

- also proposed in SPIN's documentation

```


never {
  do
  :: assert(P)
  od
}
```


SPIN will synchronise the **never claim** automaton with the automaton of the system. SPIN uses never claims to verify **LTL formulae**.

... but SPIN will issue the following unnerving **warning**:

warning: for p.o. reduction to be valid the never claim must be stutter-closed (never claims generated from LTL formulae are stutter-closed)

... and this never claim has not been generated...


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

85


University of Twente


[] P


variant 6 - LTL property

- The **logical** way...
- SPIN translates the **LTL formula** to an accepting **never claim**.

```

never { ![ ]P
  TO_init:
  if
  :: (!P) -> goto accept_all
  :: (1) -> goto TO_init
  fi;
  accept_all:
  skip
}
```


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

86


University of Twente



[] P

variant 7 - unless { !P -> ... }

- Enclose the **body** of (at least) one of the processes into the following **unless** clause:


```
{ body } unless { atomic { !P -> assert(P) ; } }
```
- Discussion
 - + **no extra process** is needed: saves 4 bytes in state vector
 - + **local variables** can be used in the property P
 - definition of the process has to be **changed**
 - the **unless** construct can **reach** inside **atomic** clauses
 - **partial order reduction** may be **invalid** if rendez-vous communication is used within **body**
 - the **body** is **not allowed** to end This is quite restrictive

Note: **disabling partial reduction (-DNOREDUCE)** may have severe **negative** consequences on the **effectiveness** of the verification run.


Thursday 11-Apr-2002
Theo C. Ruys - SPIN Beginners' Tutorial
87 

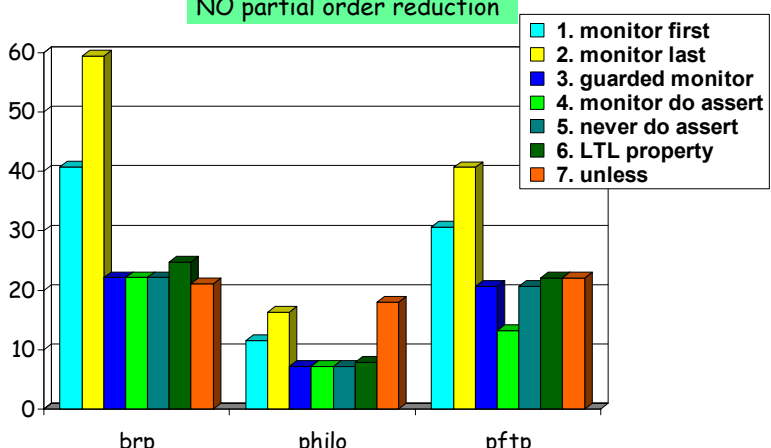
[] P

PII 300Mhz
 128 Mb
 SPIN 3.3.10
 Linux 2.2.12



Invariance experiments

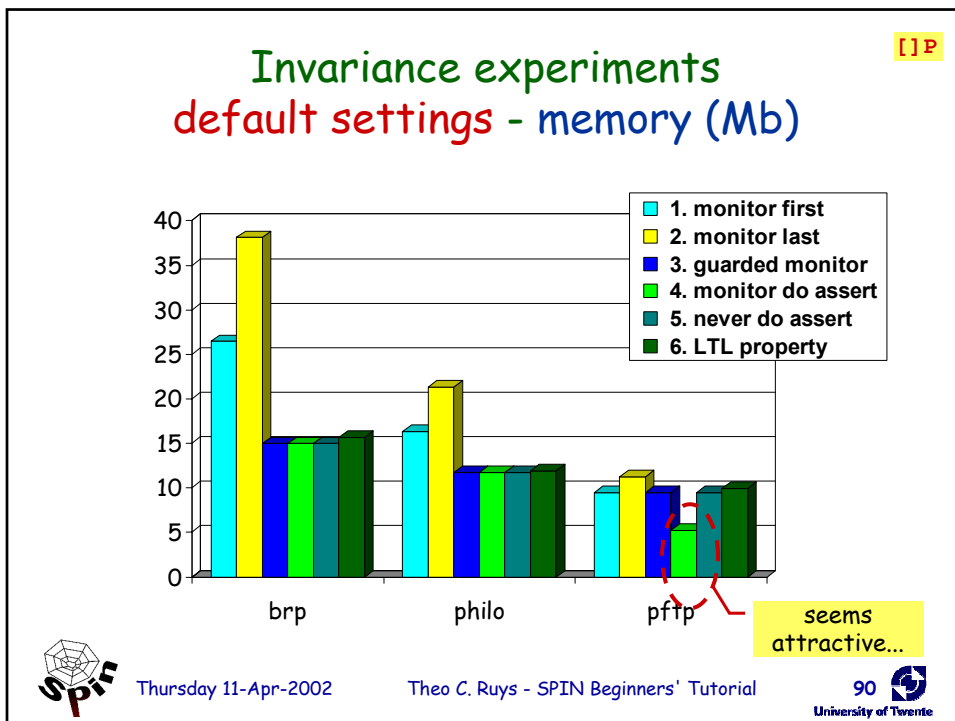
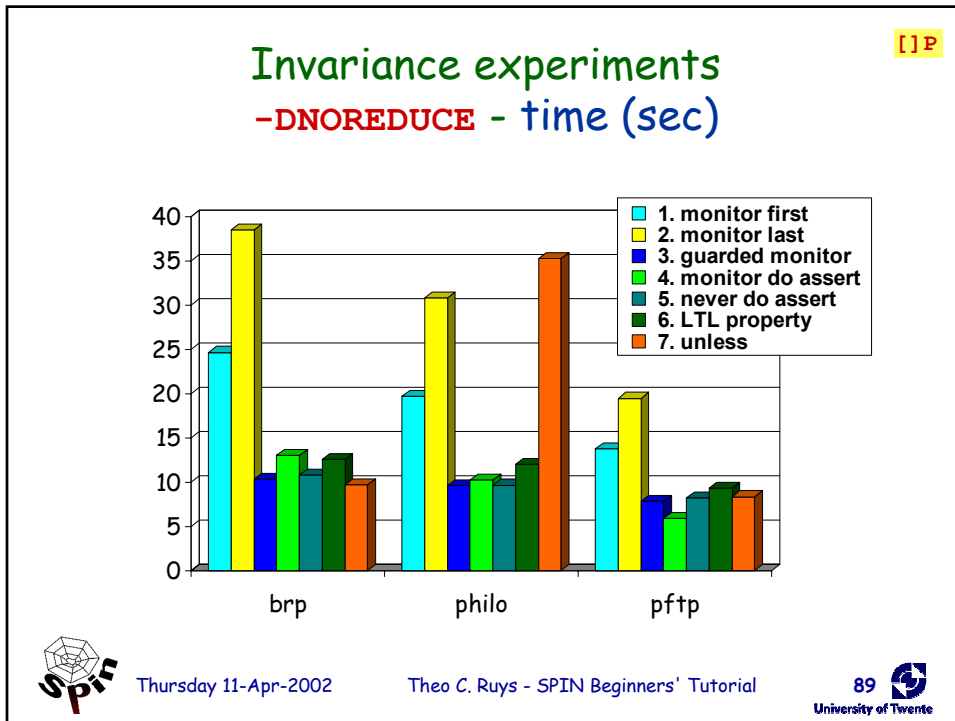
-DNOREDUCE - memory (Mb)

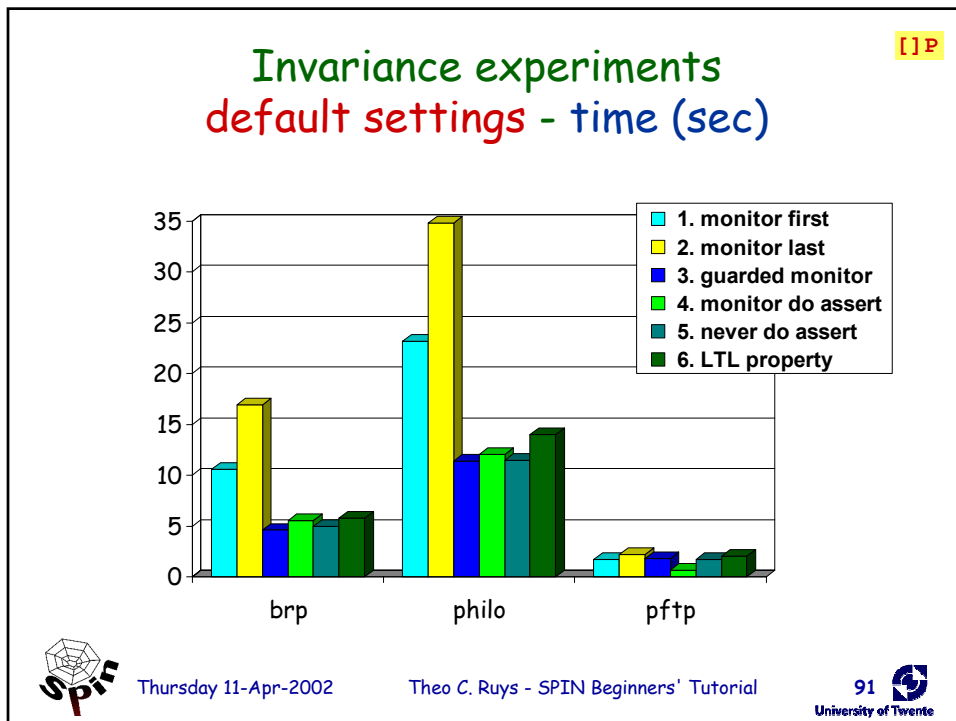
NO partial order reduction



Benchmark	1. monitor first	2. monitor last	3. guarded monitor	4. monitor do assert	5. never do assert	6. LTL property	7. unless
brp	41	60	23	23	23	25	22
philo	12	17	8	8	8	8	18
pftp	31	41	21	14	21	23	23


Thursday 11-Apr-2002
Theo C. Ruys - SPIN Beginners' Tutorial
88 






- ### Invariance - Conclusions
- The methods 1 and 2 “monitor process with single assert” performed worst on all experiments.
 - When checking invariance, these methods should be avoided.
 - Variant 4 “monitor do assert” seems attractive, after verifying the pftp model.
 - unfortunately, this method modifies the original pftp model!
 - the pftp model contains a timeout statement
 - because the do-assert loop is always executable, the timeout will never become executable
 - ⇒ never use variant 4 in the presence of timeouts
 - Variant 3 “guarded monitor process” is the most effective and reliable method for checking invariance.
- Thursday 11-Apr-2002 Theo C. Ruys - SPIN Beginners' Tutorial 92 University of Twente

Basic recipe to check $M \models \varphi$

Properties:


1. deadlock
2. assertions
3. invariance
4. liveness (LTL)

1. **Sanity check**
Interactive and random simulations
2. **Partial check**
Use SPIN's **bitstate hashing** mode to quickly sweep over the state space. states are *not stored*; fast method
3. **Exhaustive check**
If this fails, SPIN supports several options to proceed:
 1. **Compression** (of state vector)
 2. **Optimisations** (SPIN-options or manually)
 3. **Abstractions** (manually, guided by SPIN's slicing algorithm)
 4. **Bitstate hashing**



Thursday 11-Apr-2002


Theo C. Ruys - SPIN Beginners' Tutorial

93 

University of Twente


Optimising a Promela Model

- Use SPIN's "**Slicing Algorithm**" to guide abstractions
 - SPIN will propose reductions to the model on basis of the property to be checked.
- **Modelling priorities** (space over time):
 1. minimise the **number of states**
 2. minimise the **state vector**
 3. minimise the **maximum search depth**
 4. minimise the **verification time**
- Often **more than one validation model**
 - Worst case: **one model** for each property.
 - This **differs** from **programming** where one usually develops only a **single program**.

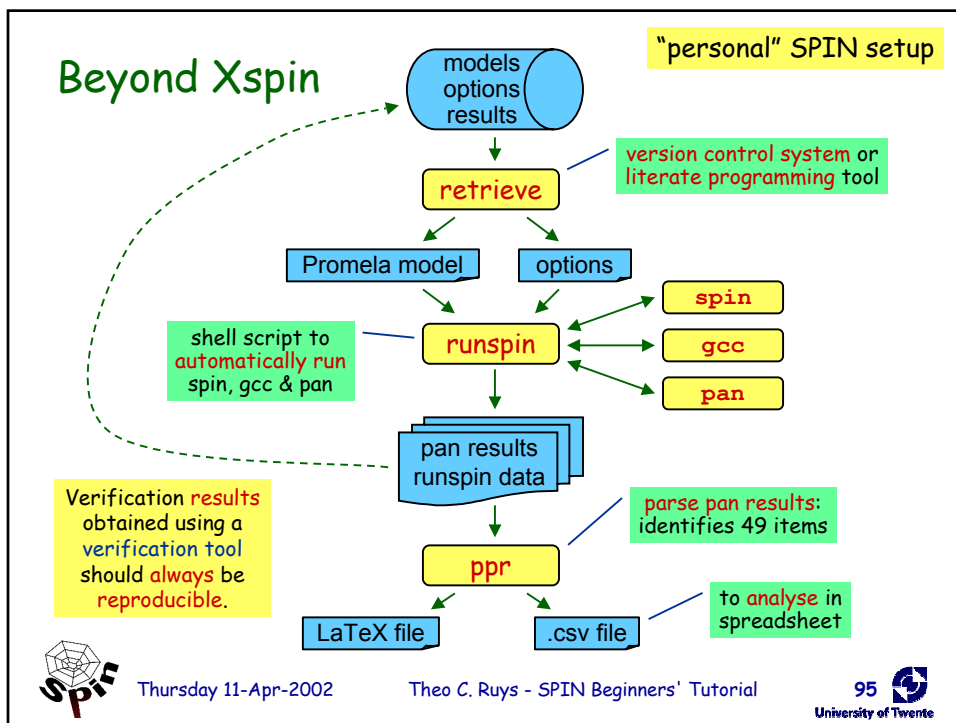


Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial

94 

University of Twente



runspin & ppr

- **runspin**
 - automates the complete verification of Promela model
 - shell script (270 loc)
 - adds extra information to SPIN's verification report, e.g.
 - options passed to SPIN, the C compiler and pan
 - system resources (time and memory) used by the verification
 - name of the Promela source file
 - date and time of the verification run
- **ppr**
 - parse pan results: recognises 49 items in verification report
 - Perl script (600 loc)
 - output to LaTeX or CSV (general spreadsheet format)



Becoming a "SPIN doctor"

- **Experiment** freely with SPIN
 - Only by **practicing** with the Promela language and the SPIN tool, one get a feeling of what it takes to construct **effective validation models** and **properties**.
- Read **SPIN** (html) **documentation** thoroughly.
- Consult "**Proceedings** of the **SPIN Workshops**":
 - papers on successful **applications** with SPIN
 - papers on the **inner workings** of SPIN
 - papers on **extensions** to SPIN
- Further reading
 - [Holzmann 2000 M'dorf] ✓ Nice overview of SPIN machinery & "modern" model checking approach.



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial

97 University of Twente

Some rules of thumb (1)

- See "**Extended Abstract**" of this tutorial in the **SPIN 2002 Proceedings** for:
 - Techniques to **reduce the complexity** of a Promela model (borrowed from Xspin's Help).
 - Tips (one-liners) on **effective Promela patterns**.
 - See [Ruys 2001] for details.
- Be careful with **data** and **variables**
 - all data ends up in the **state vector**
 - the more **different values** a variable can be assigned, the more **different states** will be generated
 - limit the number of **places** of a **channel** (i.e. the dimension)
 - prefer **local variables** over global variables



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial

98 University of Twente

Some rules of thumb (2)

- **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.



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



99

University of Twente

Summary

- **Basic SPIN**
 - Promela basics
 - Overview of Xspin
 - Several Xspin demo's
- **Advanced SPIN**
 - Some more Promela statements
 - SPIN's **reduction algorithms**
 - **Beyond Xspin**: verification management
 - **Art of modelling**

Final word of advice: get your own copy of **SPIN** and **start playing** around!



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial



100

University of Twente