Credits should go to ...

- **Gerard Holzmann** (Bell Laboratories)  
  Developer of SPIN, Basic SPIN Manual.

- **Radu Iosif** (Kansas State University, USA)  

- **Mads Dam** (Royal Institute of Technology, Sweden)  

- **Bengt Jonsson** (Uppsala University, Sweden)  

- **Joost-Pieter Katoen** (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.

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!
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 $\phi$ is stated in some formal notation.

- Problem: state space explosion!
- SPIN [Holzmann 1991] is one of the most powerful model checkers.

System Development

Modern Model Checking

<table>
<thead>
<tr>
<th>System Engineering</th>
<th>Analysis</th>
<th>&quot;Classic&quot; Model Checking</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Design</td>
<td>&quot;Modern&quot; Model Checking</td>
</tr>
<tr>
<td></td>
<td>Code</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Testing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Maintenance</td>
<td></td>
</tr>
</tbody>
</table>

Based on [Vardi & Wolper 1986].
"Classic" Model Checking

(Initial) Design

(manual) abstractions

Abstract Verification Model

refinement techniques

Implementation

Model Checker

"Modern" Model Checking

Implementation

systematic abstraction techniques

Verification Model

Model Checker

- Abstraction is the key activity in both approaches.
- This talk deals with pure SPIN, i.e., the "classic" model checking approach.
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.

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.
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.

SPIN - Introduction (1)

- SPIN (= Simple Promela Interpreter)
  = is a tool for analysing the logical consistencies 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
**SPIN - Introduction (2)**

- **Major versions:**
  
<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>Jan 1991</td>
<td>initial version [Holzmann 1991]</td>
</tr>
<tr>
<td>2.0</td>
<td>Jan 1995</td>
<td>partial order reduction</td>
</tr>
<tr>
<td>3.0</td>
<td>Apr 1997</td>
<td>minimised automaton representation</td>
</tr>
<tr>
<td>4.0</td>
<td>late 2002</td>
<td>Ax: automata extraction from C code</td>
</tr>
</tbody>
</table>

- **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)

**Documentation on SPIN**

- **SPIN’s starting page:**
  
    - 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

- **Gerard Holzmann’s website for papers on SPIN:**
  
  - [http://cm.bell-labs.com/cm/cs/who/gerard/](http://cm.bell-labs.com/cm/cs/who/gerard/)

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

Processes (1)

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

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

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

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

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

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.

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

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

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

Number of procs. (opt.)
Parameters will be initialised to 0
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);
}  
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

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.
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.

```c
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);
```

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

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
Statements (2)

- 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).

```plaintext
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 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.

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

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

- The `assert`-statement is often used within Promela models, to check whether certain properties are valid in a state.
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.

(X)SPIN Architecture
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)

---

**Mutual Exclusion** (1)

```plaintext
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(); }
}
```

---

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

---

**Wrong!**
Mutual Exclusion (2)

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

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

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.
```

Mutual Exclusion (3)

Dekker [1962]

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

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

Can be generalised
to a single process.

First "software-only" solution to the
mutex problem (for two processes).
```
Mutual Exclusion (4)

```promela
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].

if-statement (1)

```promela
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 `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.

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

- Skips are redundant, because assignments are themselves always executable...

non-deterministic branching

do-statement (1)

```plaintext
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

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

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.

Communication (1)

Sender: s2r

Receiver: r2s

s2r!MSG → MSG → s2r?MSG

r2s?ACK → ACK → r2s!ACK

! is sending
? is receiving

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

mtype (message type) models enumerations in Promela

mtype = { RED, YELLOW, GREEN };
Communication (2)

- Communication between processes is via channels:
  - message passing
  - rendez-vous synchronisation (handshake)
- Both are defined as channels:
  $$\text{chan}\ <\text{name}>\ =\ [\text{dim}]\ of\ \{<t_1>,<t_2>,\ldots,<t_n>\};$$
  
  - name of the channel
  - type of the elements that will be transmitted over the channel
  - number of elements in the channel
  - \text{dim}=0\ is\ special\ case:\ \text{rendez-vous}

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

Communication (3)

- channel = FIFO-buffer (for \text{dim}>0)

! Sending - putting a message into a channel
```
ch ! <expr_1>, <expr_2>, … <expr_n>;
```
  - The values of \text{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
```
ch ? <var_1>, <var_2>, … <var_n>;
```
  - If the channel is not empty, the message is fetched from the channel and the individual parts of the message are stored into the \text{var}_i\s.

```
ch ? <const_1>, <const_2>, … <const_n>;
```
  - If the channel is not empty and the message at the front of the channel evaluates to the individual \text{const}_i\s, the statement is executable and the message is removed from the channel.
**Communication (4)**

- **Rendez-vous communication**
  
  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.
  ```

---

**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 **alters the accompanying bit**.
Alternating Bit Protocol (2)

```plaintext
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
}

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, …)

Cookie: “hippies” problem

Germany

Holland

<= 60 min?

holes

<= 2 pers

[Ruys & Brinksma 1998]
**Cookie: soldiers problem**

![Diagram showing the cookie problem with safe and unsafe states and time limits.]

[Ruys & Brinksma 1998]

---

**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
Promela Model

- A Promela model consist of:
  - type declarations
  - channel declarations
  - global variable declarations
  - process declarations
  - \([\text{init}]\) process

Promela statements

- skip: always executable
- assert\((<\text{expr}>)\): 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 \((\text{ch}!)\): executable if channel \(\text{ch}\) is not full
- receive \((\text{ch}?)\): executable if channel \(\text{ch}\) is not empty
atomic

\begin{Verbatim}
\textbf{atomic} \{ stat_1; stat_2; ... stat_n \}
\end{Verbatim}

- 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 \texttt{stat}_1 is executable \; no pure atomicity
- if a \texttt{stat}_1 (with \texttt{i>1}) is blocked, the "atomicity token" is (temporarily) lost and other processes may do a step

• (Hardware) solution to the mutual exclusion problem:

\begin{Verbatim}
\texttt{proctype P(bit i) \{ \n  atomic \{ flag != 1; flag = 1; \} \n  mutex++; \n  mutex--; \n  flag = 0; \n \}}
\end{Verbatim}

\begin{Verbatim}
d_step \{ stat_1; stat_2; ... stat_n \}
\end{Verbatim}

- more efficient version of \textbf{atomic}: no intermediate states are generated and stored
- may only contain deterministic steps
- it is a run-time error if \texttt{stat}_1 (i>1) blocks.

- \texttt{d_step} is especially useful to perform intermediate computations in a single transition

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

• \textbf{atomic} and \textbf{d_step} can be used to lower the number of states of the model
Theo C. Ruys - SPIN Beginners' Tutorial

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

No atomicity

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

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.
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;
   ```

2. Change all atomic clauses to:
   
   ```
   atomic {
       stat_1;
       aflag=1;
       stat_2
       ...
       stat_n
       aflag=0;
   }
   ```

3. Check that `aflag` is always 0.
   
   ```
   []!aflag
   ```

   e.g. active process monitor {
       assert(!aflag);
   }
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...

timeout (2)

- Example to recover from message loss:

```promela
active proctype Receiver()
{
  bit recvbit;
do::
    toR ? MSG, recvbit -> toS ! ACK, recvbit;
  ::
    timeout -> toS ! ACK, recvbit;
  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]).
**Alternating Bit Protocol (3)**

- **abp-1.pr**
  - perfect lines
  - How large should MAX be such that we are sure that the ABP works correctly?

- **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?

**goto**

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

```promela
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
**unless**

```plaintext
{ <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:**

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

**macros - **cpp** preprocessor**

- Promela uses **cpp**, the C preprocessor to preprocess Promela models. This is useful to define:
  - **constants**
    ```cpp
    #define MAX 4
    ```
  - **macros**
    ```cpp
    #define RESET_ARRAY(a) \ 
    d_step { a[0]=0; a[1]=0; a[2]=0; a[3]=0; }
    ```
  - **conditional** Promela model fragments
    ```cpp
    #define LOSSY 1
    ...
    ifdef LOSSY
    active proctype Daemon() { /* steal messages */ } endif
    ```
**inline - poor man’s procedures**

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

```c
inline init_array(a) {
    d_step { Should be declared somewhere else (probably as a local variable).
        i=0;
        do :: i<N -> a[i] = 0; i++
            :: else -> break
        od;
        i=0;
    }

    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

---

**Properties (1)**

- Model checking tools automatically verify whether $M \models \phi$ holds, where $M$ is a (finite-state) model of a system and property $\phi$ 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
### Properties (2)

#### Safety property
- “nothing bad ever happens”
- invariant: \( x \) is always less than 5
- deadlock freedom: the system never reaches a state where no actions are possible
- SPIN: find a trace leading to the “bad” thing. If there is not such a trace, the property is satisfied.

#### Liveness property
- “something good will eventually happen”
- termination: the system will eventually terminate
- response: if action \( X \) occurs then eventually action \( Y \) will occur
- SPIN: find a (infinite) loop in which the “good” thing does not happen. If there is not such a loop, the property is satisfied.

### Properties (3)

- LTL formulae are used to specify liveness properties.

**LTL \( \equiv \) propositional logic + temporal operators**
- \([P] \) always \( P \)
- \(<\!<>P\> \) eventually \( P \)
- \(P \mathbin{U} Q\) \( P \) is true until \( Q \) becomes true

- Some LTL patterns
  - invariance \( [] (p) \)
  - response \( [] ((p) \rightarrow (<> (q))) \)
  - precedence \( [] ((p) \rightarrow ((q) \mathbin{U} (r))) \)
  - objective \( [] ((p) \rightarrow (<> (q) || (r))) \)
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!

Solution to the Hippies problem (1)

```plaintext
chan germany_to_holland = [0] of {hippie, hippie} ;
chan holland_to_germany = [0] of {hippie} ;
chan stopwatch = [0] of {hippie} ;
byte time ;
...
proctype Germany () {
  bit     here[N] ;
  hippie h1, h2 ;
  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”.

A hippie is a byte.

It can be modelled more effectively.

See [Ruys 2001] for directions.
Solution to the Hippies problem (2)

```plaintext
proc type 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: $\langle\diamond \neg (\text{time}>60)\rangle$

(random) Simulation Algorithm

```plaintext
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.

act is executed and the system enters a new state
Verification Algorithm (1)

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

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

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

Verification Algorithm (2)

1. Interleaving product of P₁, P₂, ..., Pₙ
2. Translation
3. Buchi Automaton
4. Buchi Automaton

Language intersection

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.
**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**.

\[
\text{state vector} = m \text{ bytes} \\
\text{state space} = n \text{ states} \\
\text{storing the state space may require } n \times m \text{ bytes}
\]

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

---

**Storing States in SPIN**

**Default method**

- all states are explicitly stored
- lookup is fast due to hash function
- memory needed: \( n \times m \) bytes + hash table

- **computes address/index in the hash table**

- **addresses to linked-list of states**
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.
Reduction Algorithms (2)

- **Partial Order Reduction** [Holzmann & Peled 1995 – PO]
  - observation: the validity of a property $\phi$ 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 $xr$ or $xs$.

Reduction Algorithms (3)

- **Partial Order Reduction** (cont.)
  
  Suppose the statements of $P_1$ and $P_2$ are all local.
Reduction Algorithms (3)

- **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

Reduction Algorithms (4)

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

Moore’s Law & Advanced Algorithms

[Holzmann 2000 M’dorf]
- Verification results of Tpc (The phone company)

![Graph showing memory requirements over time](image)

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

Memory requirements to (fully) verify Tpc
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

<table>
<thead>
<tr>
<th></th>
<th>BRP 1997</th>
<th>BRP 2002</th>
</tr>
</thead>
<tbody>
<tr>
<td>state vector</td>
<td>104 bytes</td>
<td>96 bytes</td>
</tr>
<tr>
<td># states</td>
<td>1,799,340</td>
<td>169,208</td>
</tr>
<tr>
<td>Memory (Mb)</td>
<td>116.399</td>
<td>14.354</td>
</tr>
</tbody>
</table>

Both verified with SPIN 3.4.x

took up to an hour in 1997

Recipes in [Ruys 2001]

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

Still in the pipeline…
Invariance

- \([P]\) where \(P\) is a state property
  - safety property
  - invariance \(\equiv\) 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:
    - \([\neg aflag]\)
    - \([\neg \text{mutex } != 2]\)

- SPIN supports (at least) 7 ways to check for invariance.

variant 1+2 - monitor process (single assert)

- 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 \(\text{assert}(P)\).
variant 3 - guarded monitor process

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

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

```plaintext
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.

variant 4 - monitor process (do assert)

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

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

- But the number of **states** is clearly advantageous.
variant 5 - never claim (do assert)

- also proposed in SPIN's documentation

\[
\text{never} \{ \\
\quad \text{do} \\
\quad \quad \text{assert(P)} \\
\quad \text{od} \\
\}\]

... 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...

variant 6 - LTL property

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

\[
\text{never} \{ ![]P \\
\text{TO_init:} \\
\quad \text{if} \\
\quad \quad :: (!P) -> \text{goto accept_all} \\
\quad \quad :: (1) -> \text{goto TO_init} \\
\quad \text{fi}; \\
\text{accept_all:} \\
\quad \text{skip} \\
\}\]
variant 7 - unless \{ !P \rightarrow \ldots \}

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

\[
\{ \text{body} \} \text{ unless } \{ \text{atomic} \{ !P \rightarrow \text{assert}(P) ; \} \}
\]

- 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

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

- DNOREDUCE - time (sec)

![Graph showing time (sec) for different scenarios]

- brp
- philo
- pftp

1. monitor first
2. monitor last
3. guarded monitor
4. monitor do assert
5. never do assert
6. LTL property
7. unless

Invariance experiments

default settings - memory (Mb)

![Graph showing memory (Mb) for different scenarios]

- brp
- philo
- pftp

1. monitor first
2. monitor last
3. guarded monitor
4. monitor do assert
5. never do assert
6. LTL property

seems attractive...

Thursday 11-Apr-2002  
Theo C. Ruys - SPIN Beginners' Tutorial  
Version: Friday, 13 September 2002
Invariance experiments

default settings - time (sec)

- 1. monitor first
- 2. monitor last
- 3. guarded monitor
- 4. monitor do assert
- 5. never do assert
- 6. LTL property

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.
Basic recipe to check $M \models \varphi$

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**

---

**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.
Beyond Xspin

/models 
/ options 
/ results 

"personal" SPIN setup

/retrieve

Promela model
/options

/runspin

/pan results 
/runspin data

/pan results 
/parse pan results: identifies 49 items 
/latex file
/csv file

to analyse in spreadsheet

shell script to automatically run spin, gcc & pan

Verification results obtained using a verification tool should always be reproducible.

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.

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
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.

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!