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

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.
"Classic" Model Checking

(initial) Design → (manual) abstractions → Abstract Verification Model → Implementation

"Modern" Model Checking

Implementation → systematic abstraction techniques → Verification Model → Model Checker

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

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

Basic SPIN

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

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

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 model extraction from C code

- Some success factors of SPIN (subjectively):
  - "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
  - result of 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

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

```promela
type msg = {MSG, ACK};
chan toS = ...
chan toR = ...
bool flag;
proctype Sender() {
}
proctype Receiver() {
}
init {
}
```
- Creates processes

Promela

- Hello World!

```java
/* A "Hello World" Promela model for SPIN. */
active proctype Hello() {
    printf("Hello process, my pid is: \%d", _pid);
}
init {
    int lastpid;
    printf("init process, my pid is: \%d", _pid);
    lastpid = run Hello();
    printf("last pid was: \%d", lastpid);
}
```
- Running SPIN in random simulation mode
- Number of processes (opt.)

Processes (1)

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

```promela
proctype Sender(chan in; chan out) {
    bit sndB, rcvB;
    do
        out ! MSG, sndB ->
    in ? ACK, rcvB;
    if
        sndB == rcvB ->
    else ->
    fi
    od
}
```
- Name
- Local variables
- Body
- Formal Parameters

Processes (2)

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

Processes (3)

- Process are created using the run statement (which returns the process id).
- Processes can be created at any point in the execution (within any process).
- Processes start executing after the run statement.
- Processes can also be created by adding active in front of the proctype declaration.
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)
```c
typedef Record {
  short f1;
  byte f2;
} Record;
```
```c
Record rr;
rr.f1 = ...
```

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.

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.

```c
2 < 3  // always executable
x < 27 only executable if value of x is smaller than 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).

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

Statements (3)

- 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.

```
assert(x <= 3);
```

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.
- 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.
XSPIN Architecture

(X)SPIN Architecture

Promela model M

SPIN

LTL Translator

Simulator

Verifier Generator

C program

Xspin

spin.exe

ϕ = | M ϕ |

LTL Translator

Simulator

Verifier Generator

C program

Xspin

spin.exe

ϕ = | M ϕ |

LTL property manager

Help system (with verification/simulation guidelines)

• SPIN 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)

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)

• 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)

• 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)

• 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)
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 choicei (guard) executable, the if-statement is executable and SPIN non-deterministically chooses one of the executable choices.
- If no choicei 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 % 3 == 0) -> n=3
:: else         -> skip
fi

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

do-statement (1)

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

- With respect to the choices, a do-statement behaves in the same way as an if-statement.
- However, instead of ending the statement at the end of the 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.

do-statement (2)

• Example – modelling a traffic light

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

Communication (1)

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

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

channel = FIFO-buffer (for dim>0)

! Sending - putting a message into a channel
  ch ! <expr1>, <expr2>, ..., <exprn>;

? Receiving - getting a message out of a channel
  ch ? <var1>, <var2>, ..., <varn>;

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** (3)

- **Rendez-vous communication**
  - \texttt{dim} \rightarrow 0
  - The number of elements in the channel is now zero.
  - If \texttt{send ch!} is enabled and if there is a corresponding \texttt{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:**
  - \texttt{chan ch = [0] of {bit, byte};}
  - P wants to do \texttt{ch ! 1, 3+7}
  - Q wants to do \texttt{ch ? 1, x}
  - Then after the communication, \texttt{x} will have the value 10.

**Alternating Bit Protocol** (1)

- \texttt{章程 [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 -> skip
      fi
    od
  }
  proctype Receiver(chan in, out)
  {
    bit recvbit;
    do
      :: in ? MSG(recvbit) ->
        out ! ACK(recvbit);
      :: timeout ->
        out ! ACK(recvbit);
    od
  }
  init
  {
    run Sender(toS, toR);
    run Receiver(toR, toS);
  }

**DEMO**

**Alternating Bit Protocol** (2)

- **Alternating Bit Protocol**
  - To every message, the \texttt{sender} adds a \texttt{bit}.
  - The \texttt{receiver} acknowledges each message by sending the received bit back.
  - To \texttt{receiver} only \texttt{excepts} messages with a bit that it expected to receive.
  - If the \texttt{sender} is sure that the \texttt{receiver} has correctly received the previous message, it sends a new message and it alternates the accompanying bit.

**DEMO**

**Alternating Bit Protocol** (3)

- \texttt{abp-1.pr}
  - perfect lines
- \texttt{abp-2.pr}
  - stealing daemon (models lossy channels)
  - how do we know that the protocol works correctly?
- \texttt{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?

**DEMO**

**Cookie: soldiers problem**

- \texttt{unsafe} <= 60 min? \texttt{safe}

<table>
<thead>
<tr>
<th>5</th>
<th>10</th>
<th>20</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsafe</td>
<td>safe</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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

- **Promela model** consist of:
  - type declarations
  - channel declarations
  - global variable declarations
  - process declarations
  - [init process]

Promela statements

- `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

atomic

```promela
atomic { stat1; stat2; ... statn }
```

- can be used to group statements into an atomic sequence;
  - all statements are executed in a single step
  - is executable if `stat1` is executable
  - if a `stat1` (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:

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

**d_step**

```promela
d_step { stat1; stat2; ... statn }
```

- more efficient version of `atomic`: no intermediate states are generated and stored
- may only contain deterministic steps
- it is a run-time error if `stat1 (i>1)` blocks.
- `d_step` is especially useful to perform intermediate computations in a single transition

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

No atomicity

![Diagram of no atomicity](image_url)

It is as if P1 has only one transition.

![Diagram of atomic](image_url)

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:
   ```plaintext
   bit aflag;
   ```

2. Change all atomic clauses to:
   ```plaintext
   atomic {
   stat1;
   stat2
   ...
   statn
   }
   ```

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

   Example:
   ```plaintext
   process monitor {
   assert([!]aflag);
   }
   ```

   It is as if P1 has only one transition.

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
  - 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:
  ```plaintext
  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).

goto

- Transfers execution to `label`
- Each Promela statement might be labelled
- Quite useful in modelling communication protocols

unless

- 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
  unless { port ? INTERRUPT; ... } {
  proctype MicroProcessor() {
  ...
  /* execute normal instructions */
  }
  }
  ```
macros - \texttt{cpp} preprocessor

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

inline - poor man's procedures

- Promela also has its own macro-expansion feature using the \texttt{inline}-construct.

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

Properties (1)

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

Properties (2)

| safety property | "nothing bad ever happens" |
| liveness property | "something good will eventually happen" |
| 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. |

Historical Classification

| safety property | "nothing bad ever happens" |
| 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.
  \[ LTL = \text{propositional logic} + \text{temporal operators} \]
  \begin{align*}
  &- [\phi]\ P \\
  &- (\phi) \ P \\
  &- P \ U \ Q \ P \text{ is true until } Q \text{ becomes true}
  \end{align*}

- Some LTL patterns
  - invariance \( [\phi] \ (p) \)
  - response \( [(\phi) \ (q)] \)
  - precedence \( [(\phi) \ (q) \ U \ (r)] \)
  - objective \( [(\phi) \ (q) \ || \ (r)] \)

Properties (4)

- Suggested further reading:
  \[ \text{Bérard et. al. 2001} \]
  - Textbook on model checking.
  - One part of the book (six chapters) is devoted to “Specifying with Temporal Logic”.
  - Also available in French.
  \[ \text{Dwyer et. al. 1999} \]
  - classification of temporal logic properties
  - pattern-based approach to the presentation, codification and reuse of property specifications for finite-state verification.
(random) Simulation Algorithm

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

act is executed and the system enters a new state.

interactive simulation: act is chosen by the user

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.

```java
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)

P1  P2  ...  Pn

interleaving product

(Spin Version 3.4.12 -- 18 December 2001)

+ Partial Order Reduction

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

State-vector 96 byte, depth reached 18187, 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

SPIN Verification Report

- the size of a single state
- longest execution path
- total number of states (i.e. the state space)
- total amount of memory used for this verification

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.

Based on [Holzmann 1997 - State Compression]

SPIN has several optimisation algorithms to make verification runs more effective:
- partial order reduction
- bitwise 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 (1)

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)

- **Partial Order Reduction** [Holzmann & Peled 1995 - PO]
  - Observation: the validity of a property $\varphi$ 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

\[\text{Holzmann & Peled 1995 – PO}\]

It is hard to determine exclusive access to channels: let user annotate exclusive channels with $x_r$ or $x_s$.

Reduction Algorithms (3)

- **Bit-state hashing** [Holzmann 1998 – Bitstate hashing]
  - Instead of storing each state explicitly, only one bit of memory is 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

\[\text{Holzmann 1998 – Bitstate hashing}\]

- **Hash-compaction** [Holzmann 1998 – Bitstate hashing]
  - Large hash table: $2^{64}$
  - Store address in regular (smaller) hash table
  - With collision detection

\[\text{Holzmann 1998 – Bitstate hashing}\]

Reduction Algorithms (4)

- **State compression** [Holzmann 1997 – State Compression]
  - Instead of storing a state explicitly, a compressed version of the state is stored in the statespace
- **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

\[\text{Holzmann & Puri 1999 - MA}\]

Moore's Law & Advanced Algorithms

[Holzmann 2000 M'dorf]

- Verification results of Tpc (The phone company)

<table>
<thead>
<tr>
<th>Year</th>
<th>Available Memory</th>
<th>Required Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1980</td>
<td>10,000</td>
<td>1,000</td>
</tr>
<tr>
<td>1987</td>
<td>1,000</td>
<td>100</td>
</tr>
<tr>
<td>1995</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>1999</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>2000</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

14.3541 16.399

Memory (Mb)

<table>
<thead>
<tr>
<th>Year</th>
<th>Memory (Mb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1980</td>
<td>116.399</td>
</tr>
<tr>
<td>1997</td>
<td>14.354</td>
</tr>
</tbody>
</table>

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

Moore's Law is exponential.

BRP – Effective Modelling

- **BRP = Bounded Retransmission Protocol**
  - Alternating bit protocol with timers
  - 1997: exhaustive verification with SPIN and UPPAAL
  - 2002: optimised SPIN version
  - Shows the effectiveness of a tuned model

<table>
<thead>
<tr>
<th>Year</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

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

1997: bitstate hashing
1995: partial order reduction
1999: minimised automaton

Both verified with SPIN 3.4.x

Told up at 7 hours 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

Still in the pipeline...

Invariance

- \[ \{P \} \] where \( P \) is a state property
  - safety property
  - \( \{P\} \) where \( P \) is a state 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:
  - \[ \{P\} \]
  - \[ \{P\} \] mutex != 2
- SPIN supports (at least) 7 ways to check for invariance.

Invariance

\[ \{P\} \]

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

variant 3 - guarded monitor process

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

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

- The atomic statement only becomes executable when \( P \) itself is not true.

variant 4 - monitor process (do assert)

- 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.

variant 5 - never claim (do assert)

- also proposed in Spin's documentation

```
ever {
    do :: assert(P) 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.

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

variant 7 - unless (P \implies \cdots)

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

```plaintext
{ body } unless ( atomic ( P \implies \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 - 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 \texttt{pftp} model.
  - unfortunately, this method modifies the original \texttt{pftp} model!
  - the \texttt{pftp} model contains a \texttt{timeout} statement
  - because the do-assert loop is always executable, the \texttt{timeout} will never become executable
  ⇒ never use variant 4 in the presence of \texttt{timeout}s
- 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.
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

runspin & ppr

- \texttt{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
- \texttt{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