Promela/SPIN

Acknowledgements:

These notes used some of the material presented by Flavio Lerda as part of Ed Clarke's model-checking course

SPIN

- **For checking correctness of process interactions**
  - Specified using buffered channels, shared variables or combination
  - Focus: asynchronous control in software systems
  - Promela – program-like notation for specifying design choices
    - Models are bounded and have countably many distinct behaviors
- **Generate a C program that performs an efficient online verification of the system's correctness properties**
- **Types of properties:**
  - Deadlock, violated assertions, unreachable code
  - System invariants, general LTL properties
- **Random simulations of the system's execution**
- **"Proof approximation"**

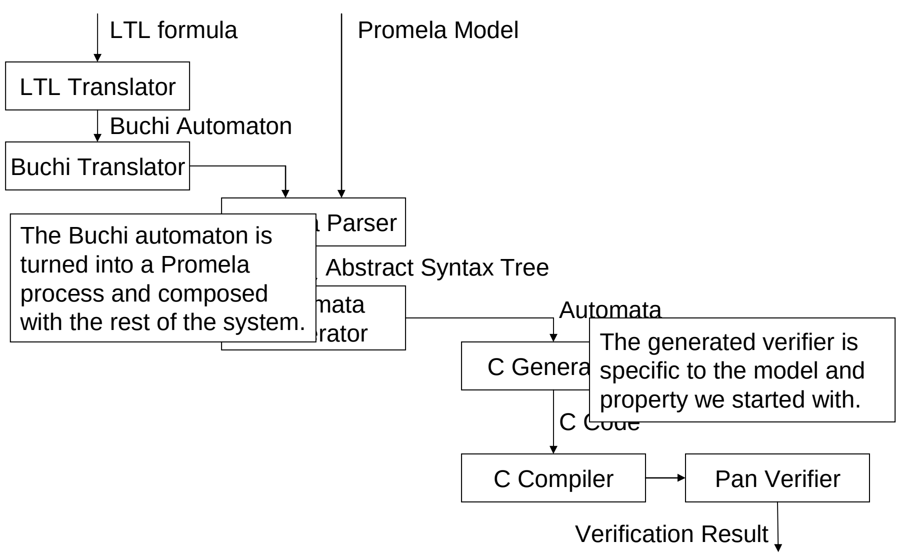
•2

Explicit State Model Checker

- **Represents the system as a finite state machine**
- **Visits each reachable state (state space) explicitly (using Nested DFS)**
- **Performs on-the-fly computation**
- **Uses partial order reduction**
- **Efficient memory usage**
  - State compression
  - Bit-state hashing
- **Version 4:**
  - Uninterpreted C code can be used as part of Promela model

•3

High Level Organization



•4

Promela (Process Meta Language)

- **Asynchronous composition of independent processes**
- **Communication using channels and global variables**
- **Non-deterministic choices and interleavings**
- **Based on Dijkstra's guarded command language**
  - Every statement guarded by a condition and blocks until condition becomes true

  Example:

```
while (a == b)
   skip /* wait for a == b */
vs
   (a == b)
```

•5

Process Types

- **State of variable or message channel can only be changed or inspected by processes (defined using `proctype`)**
- **; and -> are statement *separators* with same semantics.**
  - -> used informally to indicate causal relation between statements

  Example:

```
byte state = 2;
proctype A()
{       (state == 1) -> state = 3
}
proctype B()
{       state = state -1
}
```

- **`state` here is a global variable**

•6

Process Instantiation

- **Need to execute processes**
  - `proctype` only defines them
- **How to do it?**
  - By default, process of type `init` always executes
  - `run` starts processes
  - Alternatively, define them as `active` (see later)
- **Processes can receive parameters**
  - all basic data types and message channels.
  - Data arrays and process types are not allowed.

Example:

```
proctype A (byte state; short foo)
{      (state == 1) -> state = foo
}
init
{      run A(1, 3)
}
```

•7

Example

- **As mentioned earlier, no distinction between a statement and condition.**

```
bool a, b;
proctype p1()
{
  a = true;
  a & b;
  a = false;
}
proctype p2()
{
  b = false;
  a & b;
  b = true;
}
init { a = false; b = false; run p1(); run p2(); }
```

These statements are enabled only if both **a** and **b** are true.

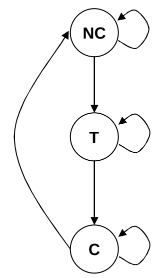
In this case **b** is always false and therefore there is a deadlock.

•8

An Example

```
mtype = { NONCRITICAL, TRYING, CRITICAL };
mtype state[2];
proctype process(int id) {
beginning:
noncritical:
   state[id] = NONCRITICAL;
   if
   :: goto noncritical;
   :: true;
   fi;
trying:
   state[id] = TRYING;
   if
   :: goto trying;
   :: true;
   fi;
critical:
   state[id] = CRITICAL;
   if
   :: goto critical;
   :: true;
   fi;
   goto beginning;}
init { run process(0); run process(1) }
```

At most one `mtype` can be declared



Other constructs

➲ Do loops

```
do
:: count = count + 1;
:: count = count - 1;
:: (count == 0) -> break
od
```

Other constructs

- Do loops
- Communication over channels

```
proctype sender(chan out)
{
  int x;

  if
  ::x=0;
  ::x=1;
  fi

 out ! x;
}
```

•11

Other constructs

- Do loops
- Communication over channels
- Assertions

```
proctype receiver(chan in)
{
 int value;
 out ? value;
 assert(value == 0 || value == 1)
}
```

•12

Other constructs

- Do loops
- Communication over channels
- Assertions
- Atomic Steps

```
int value;
proctype increment()
{ atomic
  {      x = value;
         x = x + 1;
         value = x;
  }
}
```

•13

Message Passing

`chan qname = [16] of {short}`

`qname!expr` **– writing (appending) to channel**

`qname?expr` **– reading (from head) of the channel**

`qname??expr` **– "peaking" (without removing content)**

`qname!!expr` **– checking if there is room to write**

can declare channel for exclusive read or write:

`chan in, out;   xr in;  xs out;`

`qname!exp1, exp2, exp3` **– writing several vars**

`qname!expr1(expr2, expr3)` **– type and params**

`qname?vari(var2, var3)`

`qname?cons1, var2, cons2` **– can send constants**

- Less parameters sent than received – others are undefined
- More parameters sent – remaining values are lost
- Constants sent must match with constants received

•14

Message Passing Example

```
proctype A(chan q1)
{  chan q2;
   q1?q2;
   q2!123
}
proctype B(chan qforb)
{  int x;
   qforb?x;
   print(“x=%d\n”, x)
}
init   {
   chan qname = [1] of {chan };
   chan qforb = [1] of {int };
   run A(gname);
   run B(qforb);
   qname!qforb
}
```

Prints: 123

•15

Randez-vous Communications

- **Buffer of size 0 – can pass but not store messages**
  - Message interactions by definition synchronous

Example:

```
#define msgtype 33
chan name = [0] of { byte, byte };
proctype A()
{    name!msgtype(123);
     name!msgtype(121); /* non-executable */
}
proctype B()
{    byte state;
     name?msgtype(state)
}
init
{    atomic {      run A(); run B() }
}
```

•16

Randez-Vous Communications (Cont’d)

- **If channel name has zero buffer capacity:**
  - Handshake on message `msgtype` and transfer of value 123 to variable `state`.
  - The second statement will not be executable since no matching receive operation in B
- **If channel name has size 1:**
  - Process A can complete its first send but blocks on the second since channel is filled.
  - B can retrieve this message and complete.
  - Then A completes, leaving the last message in the channel
- **If channel name has size 2 or more:**
  - A can finish its execution before B even starts

•17

Example – protocol

- **Channels `Ain` and `Bin`**
  - to be filled with token messages of type next and arbitrary values (ASCII chars)…
  - by unspecified background processes: the users of the transfer service
- **These users can also read received data from the channels `Aout` and `Bout`**
- **The channels are initialized in a single atomic statement…**
  - And started with the dummy `err` message.

•18

Example Cont'd

```
mtype = {ack, nak, err, next, accept};
proctype transfer (chan in, out, chin, chout)
{      byte o, I;
       in?next(o);
       do
       :: chin?nak(I) ->
                             out!accept(I);
                             chout!ack(o)
       :: chin?ack(I) ->
                             out!accept(I);
                             in?next(o);
                             chout!ack(o)
       :: chin?err(I) ->
                             chout!nak(o)
       od
}
```

•19

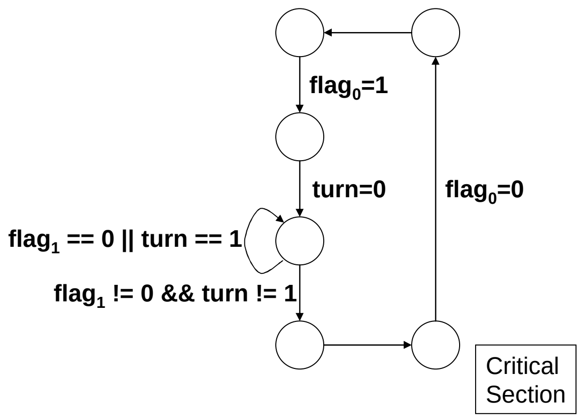
Example (Cont'd)

```
init
{      chan AtoB  = [1] if { mtype, byte };
       chan BtoA  = [1] of { mtype, byte };

       chan Ain   = [2] of { mtype, byte };
       chan Bin   = [2] of { mtype, byte };

       chan Aout  = [2] of { mtype, byte };
       chan Bout  = [2] of { mtype, byte };

       atomic {
          run transfer (Ain, Aout, AtoB, BtoA);
          run transfer (Bin, Bout, BtoA, AtoB);
        }
      AtoB!err(0)
}
```

•20

Mutual Exclusion

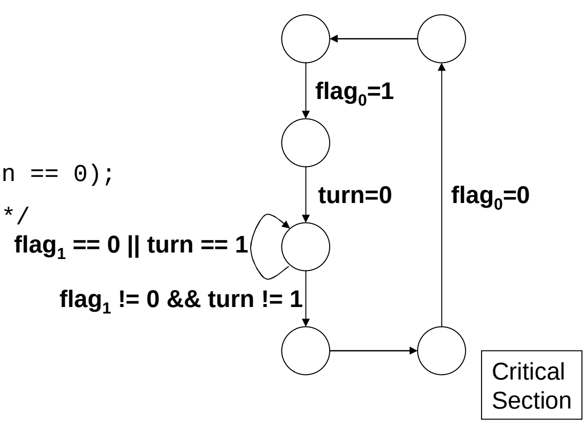
- Peterson's solution to the mutual exclusion problem



•21

Mutual Exclusion in SPIN

```
bool turn;
bool flag[2];
proctype mutex0() {
again:
  flag[0] = 1;
  turn = 0;
  (flag[1] == 0 || turn == 0);
  /* critical section */
  flag[0] = 0;
  goto again;
}
```



•22

Mutual Exclusion in SPIN

_pid:
Identifier of the process

assert:
Checks that there are only at most two instances with identifiers 0 and 1

```
bool turn, flag[2];

active [2] proctype user()
{
  assert(_pid == 0 || __pid == 1);
again:
  flag[_pid] = 1;
  turn = _pid;
  (flag[1 - _pid] == 0 || turn == 1 - _pid);

                    /* critical section */

  flag[_pid] = 0;
  goto again;
}
```

•23

Mutual Exclusion in SPIN

ncrit:
Counts the number of processes in the critical section

assert:
Checks that there is always at most one process in the critical section

```
bool turn, flag[2];
byte ncrit;

active [2] proctype user()
{
  assert(_pid == 0 || __pid == 1);
again:
  flag[_pid] = 1;
  turn = _pid;
  (flag[1 - _pid] == 0 || turn == 1 - _pid);

  ncrit++;
  assert(ncrit == 1); /* critical sec
  ncrit--;

  flag[_pid] = 0;
  goto again;
}
```

•24

Verification

- **Generate, compile and run the verifier**
  - to check for deadlocks and other major problems:

```
$ spin –a mutex
$ cc –O pan pan.c
$ pan
full statespace search for:
assertion violations and invalid endstates
vector 20 bytes, depth reached 19, errors: 0
79 states, stored
0  states, linked
38 states, matched  total: 117
hash conflicts:  4 (resolved)
(size s^18 states, stack frames:  3/0)
unreached code _init (proc 0);
  reached all 3 states
unreached code P (proc 1):
 reached all 12 states
```

•25

Mutual Exclusion

- **Verifier: Assertion can be violated**
  - Can use -t -p to find out the trace
    - Or use XSpin
- **Another way of catching the error**
  - Have another monitor process ran in parallel
  - Allows all possible relative timings of the processes
  - Elegant way to check validity of system invariant

•26

Mutual Exclusion in SPIN

```
bool turn, flag[2];
byte ncrit;

active [2] proctype user()
{
  assert(_pid == 0 || __pid == 1);
again:
  flag[_pid] = 1;
  turn = _pid;
  (flag[1 - _pid] == 0 || turn == 1 - _pid);

  ncrit++;
  /* critical section */
  ncrit--;

  flag[_pid] = 0;
  goto again;
}

active proctype monitor()
{  assert (ncrit == 0 || ncrit == 1) }
```


Finally,

- **Can specify an LTL formula and run the model-checker**

Example:

`#define p count <= 1`

- LTL claim: `[] p`

- **Note: all variables in LTL claims have to be global!**
- **LTL claim gets translated into `NEVER` claim and stored either in .ltl file or at the end of model file**
  - Only one LTL property can be verified at a time
- **Parameters can be set using XSpin**
  - Depth of search, available memory, etc.

Mutual Exclusion in SPIN

```
bool turn, flag[2];
bool critical[2];

active [2] proctype user()
{
  assert(_pid == 0 || __pid == 1);
again:
  flag[_pid] = 1;
  turn = _pid;
  (flag[1 - _pid] == 0 ||
     turn == 1 - _pid);

  critical[_pid] = 1;
  /* critical section */
  critical[_pid] = 0;

  flag[_pid] = 0;
  goto again;
}
```

LTL Properties:

```
[] (critial[0] || critical[1])

[] <> (critical[0])
[] <> (critical[1])

[] (critical[0] ->
  (critial[0] U
    (!critical[0] &&
      ((!critical[0] &&
        !critical[1]) U critical[1]))))

[] (critical[1] ->
  (critial[1] U
    (!critical[1] &&
      ((!critical[1] &&
        !critical[0]) U critical[0]))))
```

Note: critical[ ] is a global var!

•29

Alternatively,

```
#define p ncrit <= 1
#define q ncrit = 0
bool turn, flag[2];
byte ncrit;

active [2] proctype user()
{
  assert(_pid == 0 || __pid == 1);
again:
  flag[_pid] = 1;
  turn = _pid;
  (flag[1 - _pid] == 0 ||
     turn == 1 - _pid);

  ncrit++;
  /* critical section */
  ncrit--;

  flag[_pid] = 0;
  goto again;
}
```

LTL Properties:

```
[] (p)
[]<> (!q)
```

•30

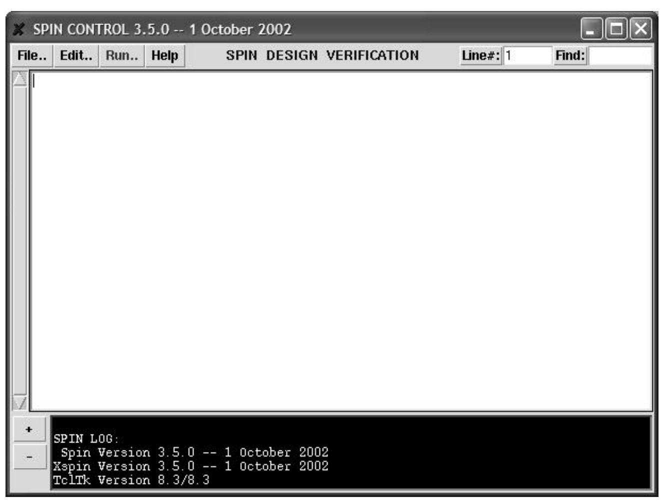
Command Line Tools

- **Spin**
  - Generates the Promela code for the LTL formula
    `$ spin –f "[]<>p"`
    - The proposition in the formula must correspond to ***#define***s
  - Generates the C source code
    `$ spin –a source.pro`
    - The property must be included in the source
- **Pan**
  - Performs the verification
    - Has many compile time options to enable different features
    - Optimized for performance

•31

Xspin

- **GUI for Spin**



•32

Simulator

- **Spin can also be used as a simulator**
  - Simulated the Promela program
- **It is used as a simulator when a counterexample is generated**
  - Steps through the trace
  - The trace itself is not “readable”
- **Can be used for random and manually guided simulation as well**

•33

A few examples

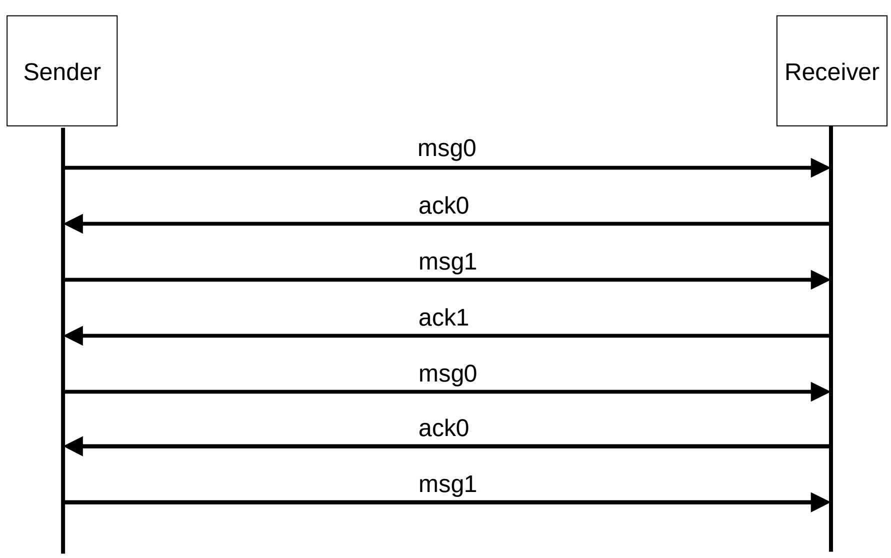
- **Alternating Bit Protocol**
- **Leader Election**

Alternating Bit Protocol

- Two processes want to communicate
- They want acknowledgement of received messages
- Sending window of one message
- Each message is identified by one bit
- Alternating values of the identifier

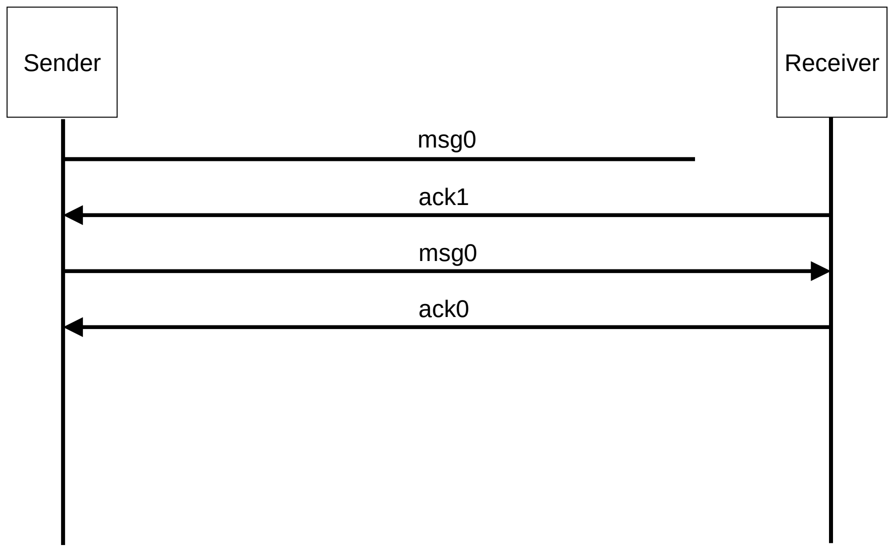
•35

Alternating Bit Protocol



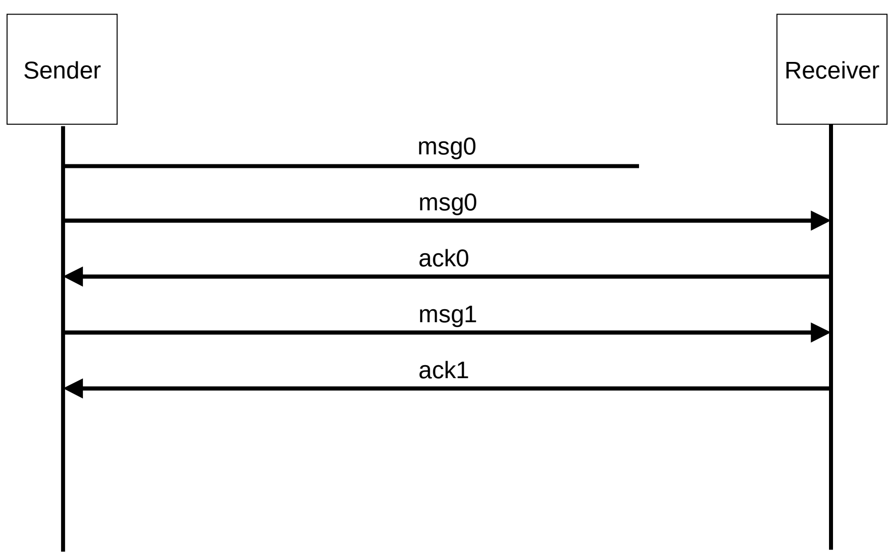
•36

Alternating Bit Protocol



•37

Alternating Bit Protocol



•38

Sender Process

```
active proctype Sender()
{
  do
  ::
    if
    :: receiver?msg0;
    :: skip
    fi;
    do
    :: sender?ack0 -> break
    :: sender?ack1
    :: timeout ->
        if
        :: receiver!msg0;
        :: skip
        fi;
    od;
  ::
    if
    :: receiver?msg1;
    :: skip
    fi;
    do
    :: sender?ack1 -> break
    :: sender?ack0
    :: timeout ->
        if
        :: receiver!msg1;
        :: skip
        fi;
    od;
  od;
}
```

•39

Receiver Process

```
active proctype Receiver()
{
  do
  ::
    do
    :: receiver?msg0 ->
        sender!ack0; break;
    :: receiver?msg1 ->
        server!ack1
    od

    do
    :: receiver?msg1 ->
        sender!ack1; break;
    :: receiver?msg0 ->
        server!ack0
    od
  od
}
```

```
mtype = { msg0, msg1, ack0, ack1 }
chan sender = [1] of { mtype };
chan receiver = [1] of { mtype };
```

•40

Leader Election

- **Elect leader in unidirectional ring.**
  - All processes participate in election
  - Cannot join after the execution started
- **Global property:**
  - It should not be possible for more than one process to declare to be the leader of the ring
    - LTL: `[] (nr_leaders <= 1)`
    - Use assertion (line 57)
      - `assert (nr_leaders == 1)`
      - this is much more efficient!
  - Eventually a leader is elected
    - `<> [] (nr_leaders == 1)`

•41

Verification of Leader Election

```
1  #define N   5    /* nr of processes */
2  #define I   3    /* node given the smallest number */
3  #define L   10   /* size of buffer (>= 2*N) */
4
5  mtype = {one, two, winner}; /* symb. message names */
6  chan q[N] = [L] of {mtype, byte} /* asynch channel */
7
8  byte nr_leaders = 0; /* count the number of processes
9    that think they are leader of the ring */
10 proctype node (chan in, out; byte mynumber)
11 { bit Active = 1, know_winner = 0;
12   byte nr, maximum = mynumber, neighbourR;
13
14   xr in;  /* claim exclusive recv access to in */
15   xs out; /* claims exclusive send access to out */
16
17   printf ("MSC: %d\n", mynumber);
18   out!one(mynumber) /* send msg of type one */
19   one:    do
20         :: in?one(nr) -> /* receive msg of type one */
```

•42

Verification of Leader Election

```
    if
    :: Active ->
       if
       :: nr != maximum ->
         out!two(nr);
         neighbourR = nr;
       :: else ->
         /* max is the greatest number */
         assert (nr == N);
         know_winner = 1;
         out!winner(nr);
       fi
    :: else ->
         out!one(nr)
    fi

:: in?two(nr) ->
    if
    :: Active ->
        if
```

•43

Verification of Leader Election

```
      :: neighbourR > nr && neighbourR > maximum
           maximum = neighbourR;
           out!one(neighbourR)
      :: else ->
             Active = 0
      fi
    :: else ->
      out !two (nr)
    fi
:: in?winner(nr) ->
    if
    :: nr != mynumber -> printf ("LOST\n");
    :: else ->
         printf ("Leader \n");
         nr_leaders++;
         assert(nr_leaders == 1);
    fi
    if
    :: know_winner
    :: else ->
         out!winner(nr)
```

•44

Verification of Leader Election

```
    fi;
    break
  od
}

init {
  byte proc;
      atomic { /* activate N copies of proc template */
           proc = 1;
           do
           :: proc <= N ->
                   run node (q[proc-1], q[proc%N],
                       (N+I-proc)% N+1);
                   proc++
           :: proc > N -> break
            od
      }
}
```

•45

Summary

- **Distinction between behavior and requirements on behavior**
  - Which are checked for their internal and mutual consistency
- **After verification, can refine decisions towards a full system implementation**
  - Promela is not a full programming language
- **Can simulate the design before verification starts**

•46

Comments

- **DFS does not necessarily find the shortest counterexample**
  - There might be a very short counterexample but the verification might go out of memory
  - If we don't finish, we might still have some sort of a result (coverage metrics)

•47

On-The-Fly

- **System is the asynchronous composition of processes**
- **The global transition relation is never build**
- **For each state the successor states are enumerated using the transition relation of each process**

•48

Visited Set

- **Hash table**
  - Efficient for testing even if the number of elements in it is very big ($\geq 10^6$)
- **Reduce memory usage**
  - Compress each state
- **Reduce the number of states**
  - Partial Order Reduction

When a transition is executed only a limited part of the state is modified

•49

SPIN and Bit-state Hashing

- **Command line:**
  - `cc –DBITSTATE –o run pan.c`
- **Can specify amount of available (non-virtual) memory directly…**
  - using `–w N` option, e.g., `-w 7` means 128 Mb of memory

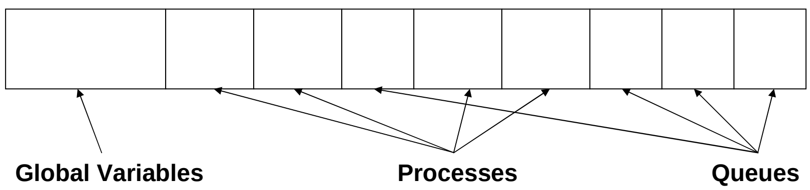
```
$ run
  assertion violated …
  pan aborted
  …
  hash factor:  67650.064516
    (size 2^22 states, stack frames:  0/5)
```

- **Hash factor:**
  - max number of states / actual number
  - Maximum number is 2^{22} or about 32 million
  - Hash factor > 100 – coverage around 100%
  - Hash factor = 1 – coverage approaches 0%

•50

State Representation

- Global variables
- Processes and local variables
- Queues



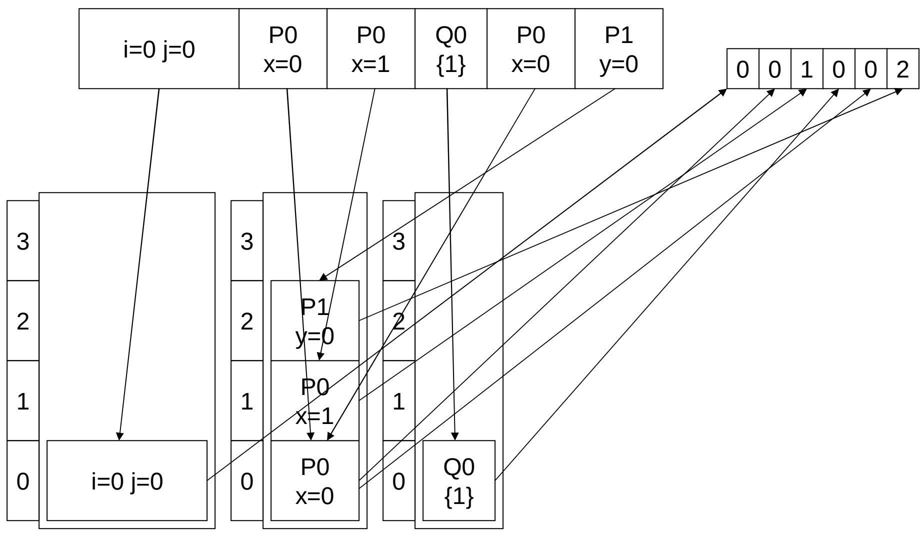
•51

Compression

- Each transition changes only a small part of the state
- Assign a code to each element dynamically
- Encoded states + basic elements use considerably less spaces than the uncompressed states

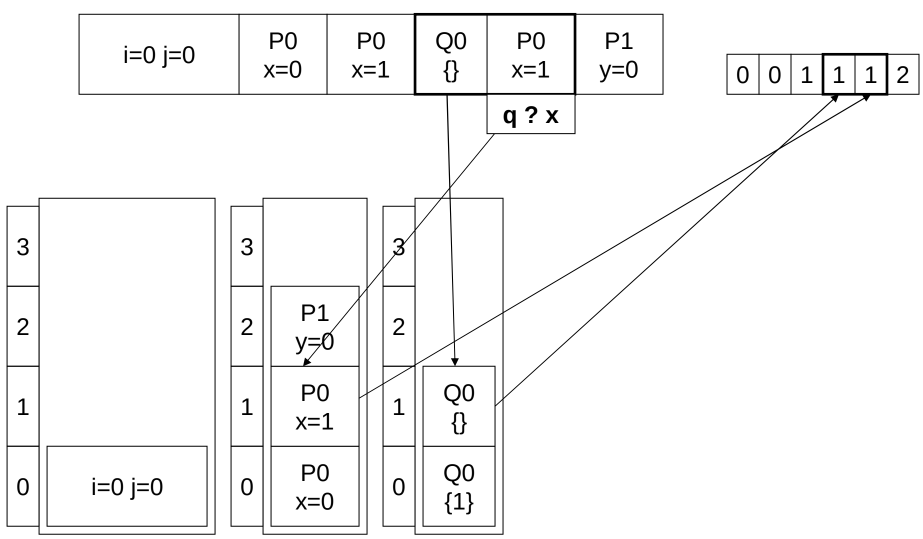
•52

Compression



•53

Compression



•54

Hash Compaction

- Uses a hashing function to store each state using only 2 bits
- There is a non-zero probability that two states are mapped into the same bits
- If the number of states is much smaller than the number of bits available there is a pretty good chance of not having conflicts
- The result is not (always) 100% correct!

•55

Minimized Automata Reduction

- Turns the state into a sequence of integers
- Constructs an automaton which accepts the states in the *visited* set
- Works like a BDD but on non-binary variables (MDD)
  - The variables are the components of the state
  - The automaton is minimal
  - The automaton is updated efficiently

•56

Partial Order Reduction

- **Optimal partial order reduction is as difficult as model checking!**
- **Compute an approximation based on syntactical information**
  - Independent
  - Invisible
  - Check (at run-time) for actions postponed at infinitum

Access to local variables
Receive on exclusive receive-access queues
Send on exclusive send-access queues

Not mentioned in the property

So called *stack proviso*

•57

References

- **http://spinroot.com/**
- *Design and Validation of Computer Protocols* **by Gerard Holzmann**
- *The Spin Model Checker* **by Gerard Holzmann**
- *An automata-theoretic approach to automatic program verification*, **by Moshe Y. Vardi, and Pierre Wolper**
- *An analysis of bitstate hashing*, **by G.J. Holzmann**
- *An Improvement in Formal Verification*, **by G.J. Holzmann and D. Peled**
- *Simple on-the-fly automatic verification of linear temporal logic*, **by Rob Gerth, Doron Peled, Moshe Vardi, and Pierre Wolper**
- *A Minimized automaton representation of reachable states*, **by A. Puri and G.J. Holzmann**

•58