SPIN - Introduction (1)

- SPIN (= Simple Promela Interpreter)
  - = is a tool for analysing the logical conisistency 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:

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

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

Documentation on SPIN

- SPIN's starting page:
  http://netlib.bell-labs.com/netlib/spin/whatispin.html
  - Basic SPIN manual
  - Getting started with Xspin
  - Getting started with SPIN
  - Examples and Exercises
  - Concise Promela Reference (by Rob Gerth)
  - Proceedings of all SPIN Workshops

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

- Gerard Holzmann's website for papers on SPIN:
  http://cm.bell-labs.com/cm/cs/who/gerard/
- SPIN version 1.0 is described in [Holzmann 1991].

Promela Model

- Promela model consist of:
  - type declarations
  - channel declarations
  - variable declarations
  - process declarations
  - [`init` process]
- A Promela model corresponds with a (usually very large, but) finite transition system, so
  - no unbounded data
  - no unbounded channels
  - no unbounded processes
  - no unbounded process creation

```
mtype = {MSG, ACK};
chan toS = ...
chan toR = ...
bool flag;

proctype Sender() {
   ...            process body
}

proctype Receiver() {
   ...
}

init {
   ...            creates processes
}
```

Processes (1)

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

name

formal parameters

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

local variables

body

The body consist of a sequence of statements.

Processes (2)

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

Processes (3)

- Process are created using the **run** statement (which returns the process id).
- Processes can be created at any point in the execution (within any process).
- Processes start executing after the **run** statement.
- Processes can also be created by adding **active** in front of the **proctype** declaration.

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

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

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

number of procs. (opt.)

parameters will be initialised to 0

DEMO

Hello World!

```
/* A "Hello World" Promela model for SPIN. */
active proctype Hello() {
    printf("Hello process, my pid is: %d\n", _pid);
}
init {
    int lastpid;
    printf("init process, my pid is: %d\n", _pid);
    lastpid = run Hello();
    printf("last pid was: %d\n", lastpid);
}
```

random seed

```
$ spin -n2 hello.pr
init process, my pid is: 1
        last pid was: 2
Hello process, my pid is: 0
                Hello process, my pid is: 2
3 processes created
```

running SPIN in random simulation mode

Variables and Types (1)

- Five different (integer) basic types.
- Arrays
- Records (structs)
- Type conflicts are detected at runtime.
- Default initial value of basic variables (local and global) is 0.

Basic types

`bit turn=1;`	[0..1]
`bool flag;`	[0..1]
`byte counter;`	[0..255]
`short s;`	$[-2^{16}-1 .. 2^{16}-1]$
`int msg;`	$[-2^{32}-1 .. 2^{32}-1]$

Arrays

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

array indicing start at 0

Typedef (records)

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

variable declaration

Variables and Types (2)

- Variables should be declared.
- Variables can be given a value by:
  - assignment
  - argument passing
  - message passing (see communication)
- Variables can be used in expressions.

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

```
int ii;
bit bb;

bb=1;
ii=2;

short s=-1;

typedef Foo {
   bit bb;
   int ii;
};
Foo f;
f.bb = 0;
f.ii = -2;

ii*s+27 == 23;
printf("value: %d", s*s);
```

assignment =

declaration + initialisation

equal test ==

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.

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

- 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)

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

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

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

Executable if `Noot` can be created…

Can only become executable if a some other process makes x greater than 2.

Statements (3)

- `assert(<expr>);`
  - The `assert`-statement is always executable.
  - If `<expr>` evaluates to zero, SPIN will exit with an error, as the `<expr>` "has been violated".
  - The `assert`-statement is often used within Promela models, to check whether certain properties are valid in a state.

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

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

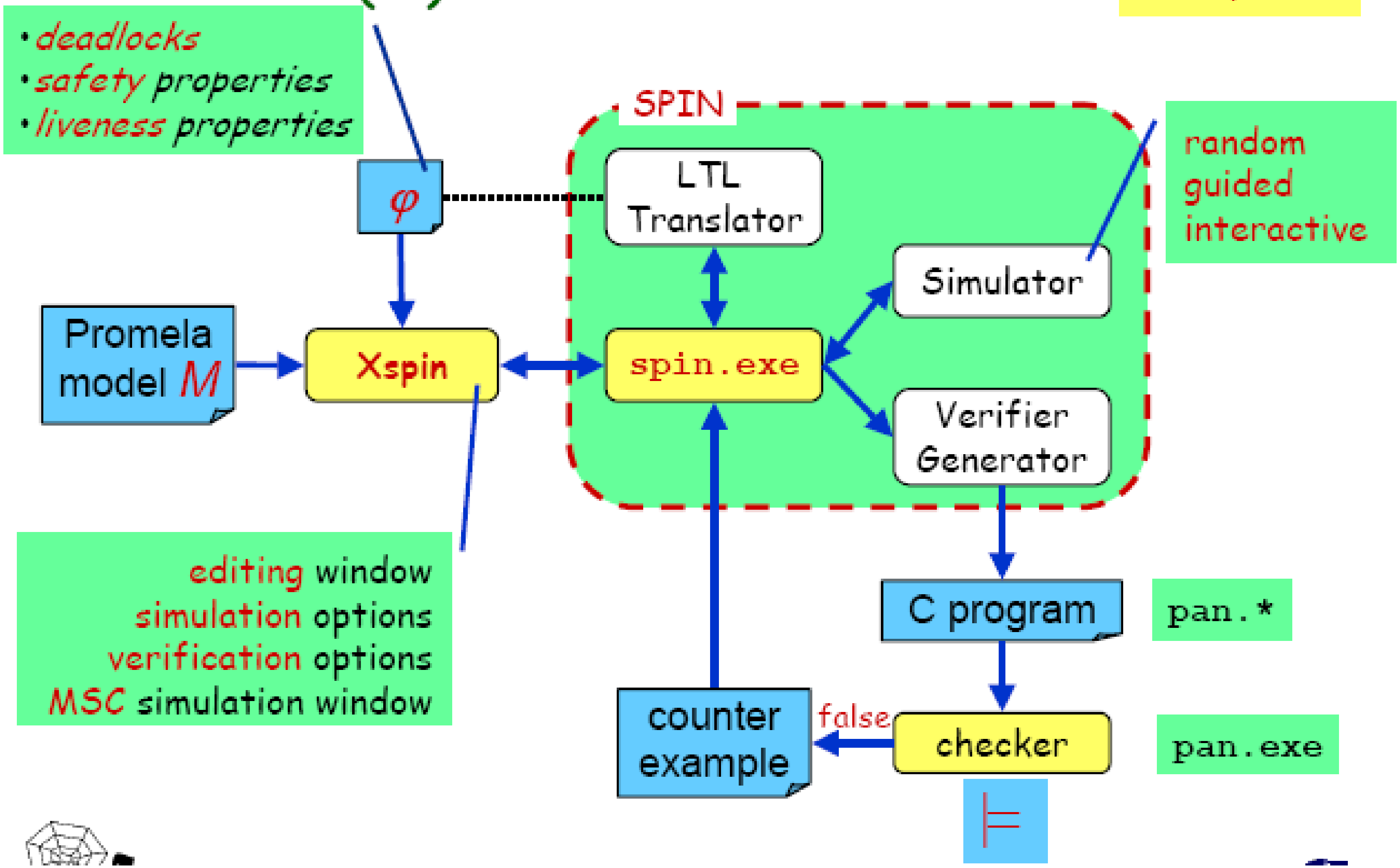
Interleaving Semantics

- Promela processes execute concurrently.
- Non-deterministic scheduling of the processes.
- Processes are interleaved (statements of different processes do not occur at the same time).
  - exception: rendez-vous communication.
- All statements are atomic; each statement is executed without interleaving with other processes.
- Each process may have several different possible actions enabled at each point of execution.
  - only one choice is made, non-deterministically.

= randomly

(X)SPIN Architecture

$M \models \varphi$



Xspin in a nutshell

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

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

DEMO

WRONG!

Mutual Exclusion (1)

```
bit  flag;     /* signal entering/leaving the section */
byte mutex;    /* # procs in the critical section.    */

proctype P(bit i) {
    flag != 1;
    flag  = 1;
    mutex++;
    printf("MSC: P(%d) has entered section.\n", i);
    mutex--;
    flag  = 0;
}

proctype monitor() {
    assert(mutex != 2);
}

init {
    atomic { run P(0); run P(1); run monitor(); }
}
```

models:
`while (flag == 1) /* wait */;`

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

starts two instances of process P

DEMO

WRONG!

Mutual Exclusion (2)

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


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

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

Process A waits for process B to end.

Problem: invalid-end-state!
Both processes can pass execute x = 1 and y = 1 "at the same time", and will then be waiting for each other.

DEMO

Dekker [1962]

Mutual Exclusion (3)

```
bit  x, y;      /* signal entering/leaving the section  */
byte mutex;     /* # of procs in the critical section.  */
byte turn;      /* who's turn is it?                     */

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

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

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

Can be generalised to a single process.

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

if-statement (1)

inspired by:
Dijkstra's guarded command language

```
if
:: choice1 -> stat1.1; stat1.2; stat1.3; …
:: choice2 -> stat2.1; stat2.2; stat2.3; …
:: …
:: choicen -> statn.1; statn.2; statn.3; …
fi;
```

- If there is at least one `choice`$_i$ (guard) executable, the `if`-statement is executable and SPIN non-deterministically chooses one of the executable choices.
- If no `choice`$_i$ is executable, the `if`-statement is blocked.
- The operator "`->`" is equivalent to "`;`". By convention, it is used within `if`-statements to separate the guards from the statements that follow the guards.

if-statement (2)

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

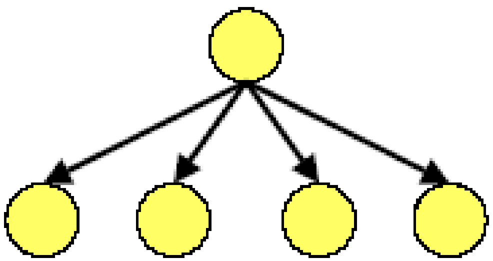
- The **else** guard becomes executable if none of the other guards is executable.

give n a random value

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

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

non-deterministic branching



do-statement (1)

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

- With respect to the choices, a `do`-statement behaves in the same way as an `if`-statement.
- However, instead of ending the statement at the end of the choosen list of statements, a `do`-statement repeats the choice selection.
- The (always executable) `break` statement exits a `do`-loop statement and transfers control to the end of the loop.

do-statement (2)

- Example – modelling a traffic light

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

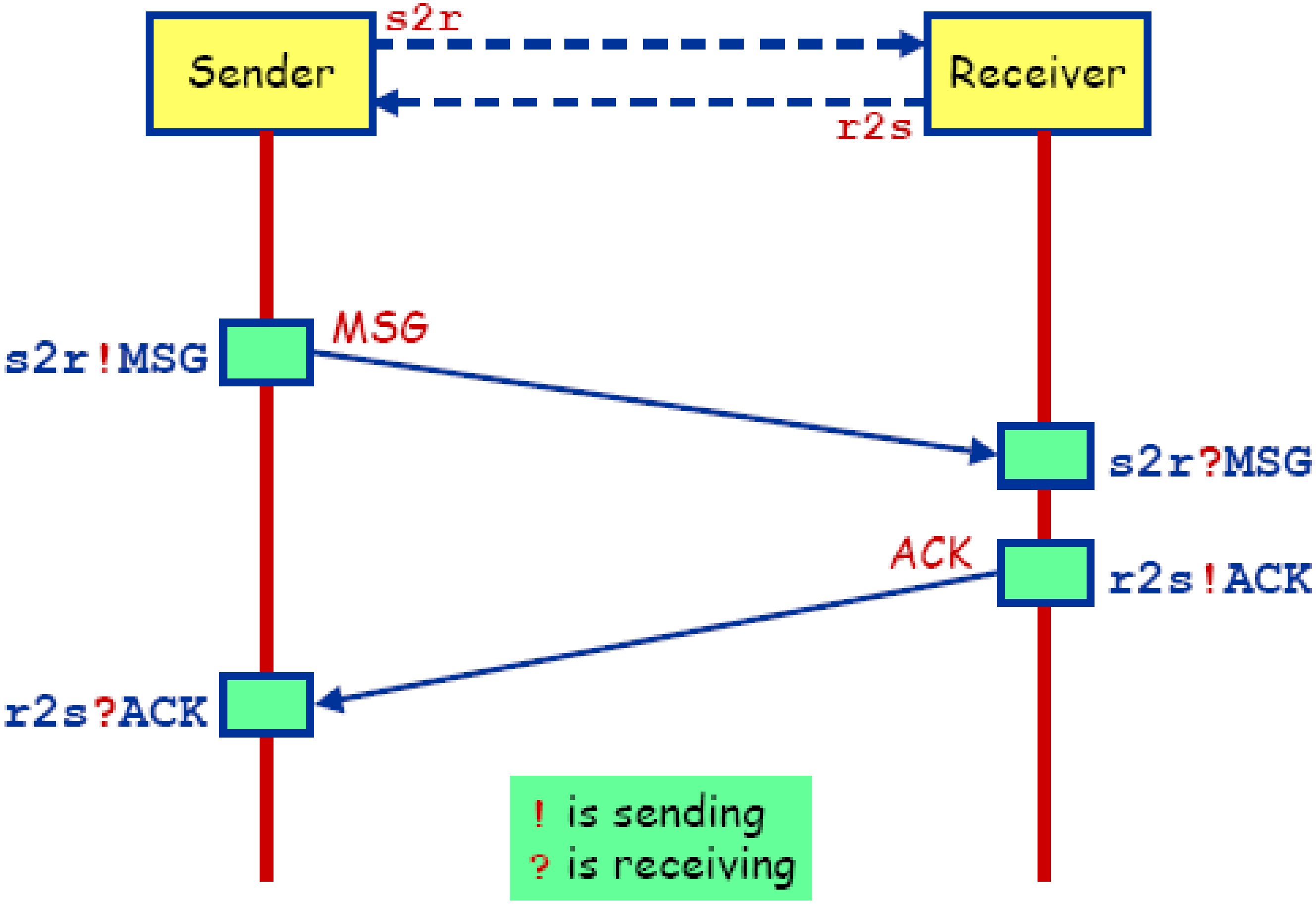
```
mtype = { RED, YELLOW, GREEN } ;
```

mtype (message type) models enumerations in Promela

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

Note: this do-loop does not contain any non-deterministic choice.

Communication (1)



Communication (2)

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

also called:
queue or buffer

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

name of
the channel

type of the elements that will be
transmitted over the channel

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

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

array of
channels

Communication (3)

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

! **Sending** - *putting a message into a channel*

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

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

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

<var> + <const> can be mixed

```
ch ? <var1>, <var2>, … <varn>;
```

message passing

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

```
ch ? <const1>, <const2>, … <constn>;
```

message testing

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

Communication (4)

- Rendez-vous communication

  `<dim>` == 0

  The number of elements in the channel is now zero.

  - If send `ch!` is enabled and if there is a corresponding receive `ch?` that can be executed simultaneously and the constants match, then both statements are enabled.
  - Both statements will “handshake” and together take the transition.

- *Example:*

  `chan ch = [0] of {bit, byte};`

  - P wants to do `ch ! 1, 3+7`
  - Q wants to do `ch ? 1, x`
  - Then after the communication, `x` will have the value `10`.

Promela Model

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

Promela statements

are either executable or blocked

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

atomic

atomic { $\mathtt{stat}_1$; $\mathtt{stat}_2$; ... $\mathtt{stat}_n$ }

- can be used to group statements into an atomic sequence; all statements are executed in a single step (no interleaving with statements of other processes)
- is executable if $\mathtt{stat}_1$ is executable — no pure atomicity
- if a $\mathtt{stat}_i$ (with `i>1`) is blocked, the "atomicity token" is (temporarily) lost and other processes may do a step

• (Hardware) solution to the mutual exclusion problem:

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

d_step

```
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 $\texttt{stat}_i$ (i>1) blocks.
- `d_step` is especially useful to perform intermediate computations in a single transition

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

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

`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:

```
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]).

goto

```
goto label
```

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

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

Timeout modelled by a channel.

Part of model of BRP

unless

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

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

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

macros - cpp preprocessor

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

```
#define MAX 4
```

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

  - macros

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

  - conditional Promela model fragments

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

inline - poor man's procedures

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

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

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

Be sure to reset temporary variables.

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

State vector

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

state vector = m bytes
state space = n states

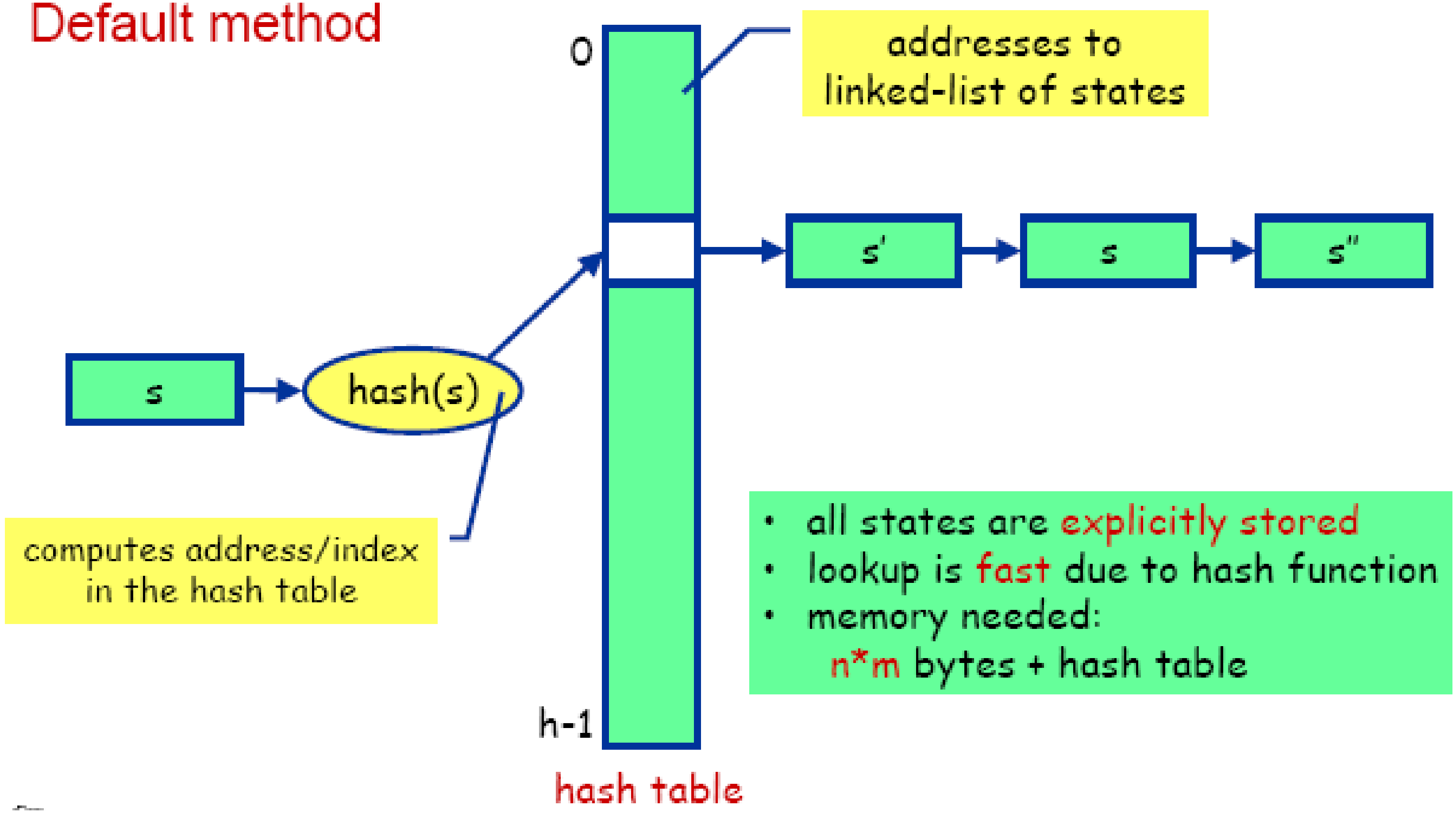
storing the state space may require n*m bytes

SPIN provides several algorithms to compress the state vector.

[Holzmann 1997 - State Compression]

Storing States in SPIN

Default method



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

SPIN Verification Report

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

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

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

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

the size of a single state

longest execution path

property was satisfied

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

total amount of memory used for this verification

A tutorial by Theo Ruys