NSWI101: System Behaviour Models and Verification

3. Spin

Jan Kofroň

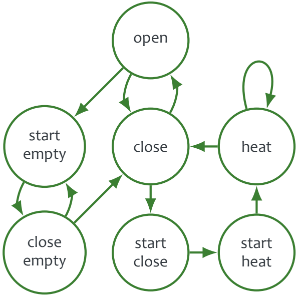
FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

Department of Distributed and Dependable Systems

Today

- Spin model checker

MODEL CHECKING



System model

$AG\ (start \rightarrow AF\ heat)$

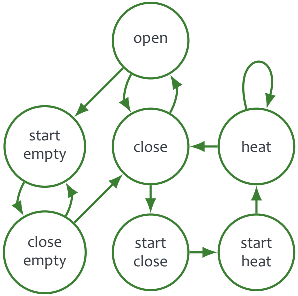
Property specification

Model Checker

Property satisfied

Property violated

Model checking



System model

AG (start → AF heat)

Property specification

Spin

Model Checker

Property satisfied

Property violated

Spin overview

Complete set of original slides used in this presentation available at:

- `http://spinroot.com/spin/Doc/SpinTutorial.pdf`
- `http://spinroot.com/spin/Doc/Spin_tutorial_2004.pdf`

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

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;           [-2^15-1..2^15-1]
int   msg;         [-2^31-1..2^31-1]
```

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

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);
  ...
}
```

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

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

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

Process A waits for process B to end.

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

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)

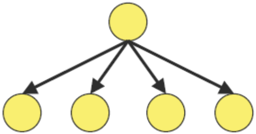
```
if
:: (n % 2 != 0) -> n=1
:: (n >= 0)     -> n=n-2
:: (n % 3 == 0) -> n=3
:: else         -> skip
fi
```

- 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
```

non-deterministic branching



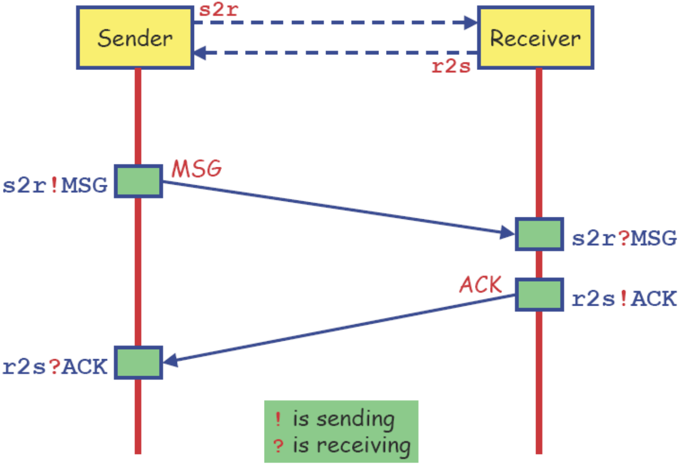
skips are redundant, because assignments are themselves always executable...

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 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.

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 ! <expr$_1$>, <expr$_2$>, … <expr$_n$>;

- 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 ? <var$_1$>, <var$_2$>, … <var$_n$>;

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 ? <const$_1$>, <const$_2$>, … <const$_n$>;

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.

DEMO

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`.

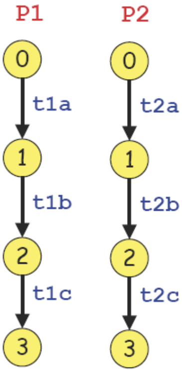
Interleaving of processes

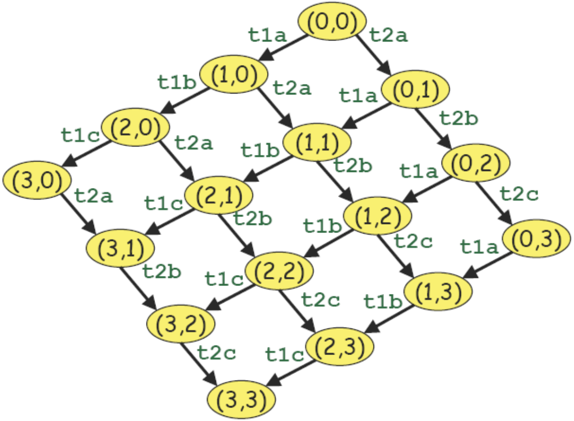
Interleaving semantics:

- Each time, process is selected, and its current statement is executed
- Selected process has to be enabled
- This is repeated
- Number of all possible interleavings may be very high
  $\Longrightarrow$ state space explosion $\Longrightarrow$ not verifiable models
- Mechanism to control the interleavings would be handy

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

No atomicity



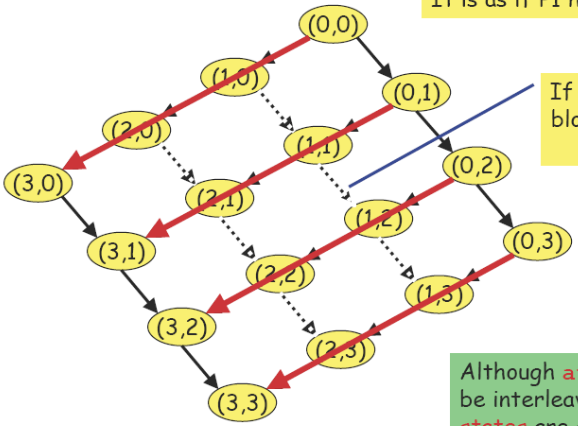


Not completely correct as each process has an implicit end-transition...

```
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…



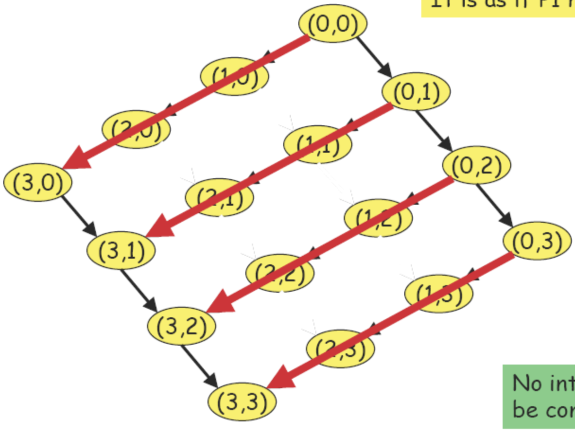
If one of P1's transitions blocks, these transitions may get executed

Although atomic clauses cannot be interleaved, the intermediate states are still constructed.

d_step

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

It is as if P1 has only one transition...



No intermediate states will be 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 {
  stat1;
  aflag=1;
  stat2

  ...

  statn
  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:

```
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

(random) Simulation Algorithm

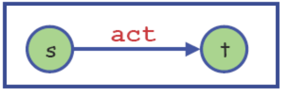
deadlock ≡ allBlocked

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

interactive simulation: act is chosen by the user

act is executed and the system enters a new state

Visit all processes and collect all executable actions .



Verification Algorithm (1)

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

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

> states are stored in a hash table

> requires state matching

> Only works for state properties.

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

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

Properties (1)

- Model checking tools automatically verify whether
  $$M \models \phi$$
  holds, where M is a (finite-state) model of a system and property ϕ is stated in some formal notation.

- With SPIN one may check the following type of properties:
  - deadlocks (invalid endstates)
  - assertions
  - unreachable code
  - LTL formulae
  - liveness properties
    - non-progress cycles (livelocks)
    - acceptance cycles

Property specification

LTL_{-X} is used in Spin

- LTL without X operator
- More efficient model checking algorithm
- Still expressive enough

Describing properties of states (or runs), not of transitions between states

EXAMPLE: ALTERNATING BIT PROTOCOL – ABP

Four versions with various properties:

1. Perfect lines
2. Loosing messages
3. Fixing deadlock
4. Checking for progress

ABP VERSION 1

```
#define MAX 4;
mtype {MSG, ACK};
chan toR = [1] of {mtype, byte, bit};
chan toS = [1] of {mtype, bit};

active proctype Sender()
{
  byte data;
  bit sendb, recvb;
  sendb = 0;
  data = 0;
  do
    :: toR ! MSG(data,sendb) ->
       toS ? ACK(recvb);
    if
      :: recvb == sendb -> sendb = 1-sendb;
         data = (data+1)%MAX;
      :: else -> skip; /* resend old data */
    fi
  od
}
```

```
active proctype Receiver()
{
  byte data, exp_data;
  bit ab, exp_ab;
  exp_ab = 0;
  exp_data = 0;
  do
    :: toR ? MSG(data,ab) ->
    if
      :: (ab == exp_ab) ->
           assert(data == exp_data);
           exp_ab = 1-exp_ab;
           exp_data = (exp_data+1)%MAX;
      :: else -> skip;
    fi;
    toS ! ACK(ab)
  od
}
```

ABP VERSION 2

Adding special stealing daemon process:

```
active proctype Daemon()
{
  do
    :: toR ? _, _, _
    :: toS ? _, _
  od
}
```

ABP VERSION 3

Fixing sender model to escape from deadlock:

```
do
  :: toR ! MSG(data,sendb) ->
     if
       ::toS ? ACK(recvb) ->
          if
            :: recvb == sendb -> sendb = 1-sendb;
                                 data = (data+1)%MAX;
            :: else /* resend old data */
          fi
       ::timeout /* message lost */
     fi
od
```

ABP VERSION 4

Augmenting receiver process to detect livelock:

```
do
  :: toR ? MSG(data,ab) ->
  if
    :: (ab == exp_ab) -> assert(data == exp_data);
                         exp_ab = 1-exp_ab;
                         progress:
                         exp_data = (exp_data+1)%MAX;
    :: else -> skip;
  fi;
  toS ! ACK(ab)
od
```

ALTERNATING BIT PROTOCOL – SUMMARY

We should be aware of all possible executions and issues in the model

If there is error due to simplification (abstraction), it can still be ok

- In our example we may know that messages *can* get lost but are *usually* delivered
- Consider possible errors beyond the ignored ones!

Model is not implementation!