Promela and SPIN

Mads Dam
Dept. Microelectronics and Information Technology
Royal Institute of Technology, KTH

Promela and SPIN

- Promela (<u>Pro</u>tocol <u>Me</u>ta <u>La</u>nguage):
  - Language for modelling discrete, event-driven systems as transition systems
- SPIN
  - Tool for performing simulations and full state-space validations of Promela models
- XSPIN
  - X-interface to SPIN with graphic and textual representation of execution traces, message sequence diagrams, state-transition diagrams when running a Promela model

Promela and SPIN

Promela and SPIN/XSPIN are

- Developed by Gerard Holzmann at Bell Labs
- Freeware for non-commercial use
- State-of-art model checker (another is SMV)
- Used by more than 2000 users

See course binder and SPIN home page for more information

Promela Models

- Describe (possibly very large but) **finite** transition system
- Essentially:
  - No unbounded data
  - No recursive processes
  - No unbounded process creation
- SPIN traverses the finite transition system
- States constructed as they are visited (on-the-fly)
  - CWB equivalence checker constructs state space in advance
- Temporal logic: Specifications represented as transition systems
- This lecture: Getting started with Promela

SPIN vs CCS

SPIN:

- Dressed up automata
- Verification by traversing states

- ”Realistic” program model
- Linear time
- Properties as automata
- On-the-fly state space exploration
- Sharable store
- Buffered comms

CCS:

- Dressed up automata
- Verification by traversing states

- Elegance of theory
- Branching time
- Properties as logic
- State space constructed ”in advance”
- No store
- Primitive, synchronous comms

Alternating Bit Protocol

```
mtype = { msg,ack };
chan to_sndr = [2] of { mtype,bit };
chan to_rcvr = [2] of { mtype,bit };

proctype Sender(chan in, chan out)
{
    bit sendVal, recVal;
    do
    :: out!msg(sendVal) ->
          in?ack(recVal);
          if
          :: recVal == sendVal ->
                   sendVal = 1 - recVal
          :: else -> skip
          fi
    od
}
```

```
proctype Receiver(chan in, chan out)
{
    bit recVal;
    do
    :: in?msg(recVal) ->
          out!ack(recVal)
    :: timeout ->
          out!ack(recVal)
    od
}

init
{
    run Sender(to_sndr,to_rcvr);
    run Receiver(to_rcvr,to_sndr)
}
```

Promela

Promela model:

- Process types
- Channel declarations
- Variable declarations
- Main program

```
chan to_sndr = ...

proctype Sender (chan in,m chan out)
{
   ...
}

proctype Receiver (chan in, chan out)
{
   ...
}

init
{
   ...
}
```


Processes

A process

- executes concurrently with all other processes, independent of speed and behaviour
- communicates with other processes using channels
- may access shared variables
- follows the description of a process type

There may be several processes of the same type

Each process has own local state

```
proctype Sender (chan in, chan out)
{
   ...
}
```

A Process Type

A process type consists of

- a name
- a list of formal parameters
- local variable declarations
- body

Process name

```
proctype Sender(chan in, chan out)
{
   bit sendVal, recVal;
   do
   :: out!msg(sendVal) ->
        in?ack(recVal);
        if
        :: recVal == sendVal ->
                sendVal !=
   recVal
        :: else -> skip
        fi
   od
}
```

Formal parameters

Local variable declarations

Body

2004 Mads Dam 9 2G1516/2G1521
IMIT, KTH Formal Methods

Process Creation

- Processes created by `run` statement
- Value of run statement is a process identifier
- Processes can be created at any point of execution
- Processes start executing after execution of run statement
- Processes can also be created by adding `active` in front of process type declaration

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

init
{   ...
   run p(a) ;
   pid4 = run p(b) ;
    ...
}
```

Number of processes created (optional)

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

Obs: No parameter when using active

2004 Mads Dam 10 2G1516/2G1521
IMIT, KTH Formal Methods

Data Types and Variables

- Five different types of integers as basic types
- Records and arrays for compound data
- Type conflicts detected at ”runtime”
- Default initial value 0

Integers

`bit flag, turn=1`	[0..1]
`bool isSet`	[0..1]
`byte counter`	[0..255]
`short balance`	$[-2^{15}..2^{15}-1]$
`int range`	$[-2^{31}..2^{31}-1]$

Records

```
typedef Field {
        short foo = 8 ;
        byte bar
} ;
```

Arrays

```
typedef Array {
        byte elmnt[4]
};
```


Tests and Assignment

- Assignment with single equals sign: `a = 2`
- Testing for equality: `a == 2`
- Inequality: `a != 2`
- Comparisons: `a >= 2, a <= 2`
- Logical conjunction: `foo && bar`
- Disjunction: `foo || bar`
- Negation: `!foo`

Channels

Channels model transfer of data between processes

Each channel has typed buffer of finite length

Special type mtype used for enumerating message types

Enumerated from 1 upwards

```
chan to_sndr = [2] of byte
```

Name — Size of buffer — Type of fields in each slot

```
mtype = { msg,ack }
```

Has value 1

```
chan to_rcvr = [2] of {mtype,bit}
```


Channels, cont'd

- Channel: Fifo buffer with number of slots, each with same number and types of fields
- Several processes can share same channel
- A channel is usually, but not always, used unidirectionally between two processes

Receive statement:

```
in_q?msg,recVal
```

Equivalently:

```
in_q?msg(recVal)
```

Executable only if buffer nonempty

Send statement:

```
out_q!ack,sendVal
```

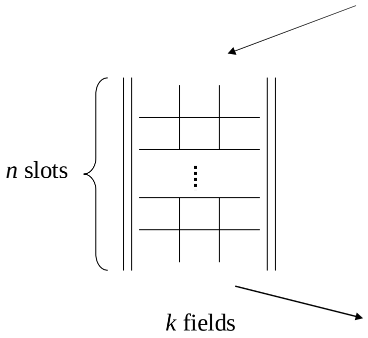
Equivalently:

```
out_q!ack(sendVal)
```

Default: Executable when buffer has at least 1 free slot

Channels, cont'd

So: A channel is a fifo buffer with *n* slots, each consisting of *k* fields




```
chan ch = [n] of { T1,...,Tk }
```


Channels, Variations

- Can change default behaviour to always send, loose message if buffer full
- Can receive more or less values than specified in receive statement
  - More values => message loss
  - Less values => params undefined
- Can match against constants:
  `in_q?chan1(recVal)`
- Queue operations:
  - `len(qname)`
  - `empty(qname)`
  - `full(qname)`
- Attention!
  ```
  !full(qname) ->
    qname!msg0
  ```
  not guaranteed to succeed
- Lookahead:
  `qname?[msg]`
- Also rendez-vous construction and sorted input/output

Concurrency

- Specification of process behaviour in Promela
- Processes execute concurrently
- Nondeterministic scheduling, interleaving
- All statements within a single process are executed sequentially
- Each process may have several different possible actions enabled at each point of execution
- One choice is made, nondeterministically

Semantics of Execution

- Transition has two components: Side-effect free condition and an atomic action
- A transition is executable if its condition holds, otherwise it is blocked
- Following rules apply:
  - Assignments are always executable
  - Run statements are executable is new processes can be created
  - Conditions are executable if the truth value is true
  - Send statements are executable if channel not full (or ...)
  - Receive statements are executable if channel is nonempty and patterns match
  - Skip statements are always executable

`if` Statement

- First statement in each entry acts as guard
- Collect all entries with executable guard
- Select one of these entries nondeterministically and execute it
- If no entry is executable then execute the `else` entry
- If no `else` entry exists, hang
- No restriction on type of guard

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


Statement Delimiters

- There are two types of statement delimiters to use between (not after) statements. These can be used interchangably:

  ; and ->

- Use the one most appropriate at the given situation
- Usually, ; is used between ordinary statements
- An -> is often used after ”guards” in a `do` or `if` statement, pointing at what comes next

do Statement

Repeat forever, or until a `break` or `goto` statement is encountered

First statement in each entry will act as guard

Collect all entries with executable guard; randomly select one for execution

If no executable entry exists, hang

```
do
:: ( n % 2 != 0 ) ->
      goto oddLabel
:: ( n  >= 0 ) ->
      n = n - 2
:: ( n % 3 == 0 ) ->
      m = 3
:: ( n == 3 ) -> break
od
```


skip

- Condition always true, no effect
- Useful when removing a statement, but state space should be unaffected
- (Sometimes needed in never claims when matching an arbitrary statement)

Alternating Bit Protocol

```
mtype = { msg,ack };
chan to_sndr = [2] of { mtype,bit };
chan to_rcvr = [2] of { mtype,bit };

proctype Sender(chan in, chan out)
{
    bit sendVal, recVal;
    do
    :: out!msg(sendVal) ->
          in?ack(recVal);
          if
          :: recVal == sendVal ->
                  sendVal = 1 - recVal
          :: else -> skip
          fi
    od
}
```

```
proctype Receiver(chan in, chan out)
{
    bit recVal;
    do
    :: in?msg(recVal) ->
          out!ack(recVal)
    :: timeout ->
          out!ack(recVal)
    od
}

init
{
    run Sender(to_sndr,to_rcvr);
    run Receiver(to_rcvr,to_sndr)
}
```


Modelling Loss of Messages

```
mtype = { msg,ack };
chan to_sndr = [2] of { mtype,bit };
chan to_rcvr = [2] of { mtype,bit };

active proctype Receiver()
{
    bit seq_in;
    do
    :: to_rcvr?msg(seq_in) ->
          to_sndr!ack(seq_in)
    :: to_rcvr?msg(seq_in) ->
          skip      /* message loss */
    :: timeout ->
          to_sndr!ack(seq_in)
    od
}
```

```
active proctype Sender()
{
    bit seq_in, seq_out;
    do
    :: to_rcvr!msg(seq_out) ->
        if
        :: to_sndr?ack(seq_in) ->
            if
            :: seq_in == seq_out ->
                seq_out = 1 - seq_in
             :: else -> skip
            fi
        :: to_sndr?ack(seq_in) ->
            skip     /* message loss */
        fi
    od
}
```

atomic Statement

Flag is a global variable. Will the loops terminate?

```
do
::     flag = 1 ;
   if
   :: (flag == 1) -> break
   :: else -> skip
   fi
od
```

```
do
:: atomic
   {   flag = 1 ;
       if
       :: (flag == 1) ->
   break
       :: else -> skip
       fi
   }
od
```

atomic statements used to prevent interference

d_step Statements

Atomic statements used to prevent interference, but individual states and transitions still present

d_step used to construct new primitive transitions

Requires:

- Determinacy
- No jumps in or out of d_steps
- No statements inside d_step must become unexecutable – else runtime error

Swap values of a and b:

```
d_step {
        tmp = b ;
        b = a ;
        a = tmp
      }
```

Labels

- A label is an identifier ending with a colon used for referring to specific statement
- Labels are used for jumps and for some validations
- Special labels start with one of
  - `accept`
  - `progress`
  - `end`

```
...
snd_loc: nxt!nxtVal(val);
...

endState:
   do
   :: to_rcvr?msg(seq_in)
   ->
progressLabel:
        to_sndr!ack(seq_in)
   :: to_rcvr?msg(seq_in)
acceptanceHere:     skip
   od
```


Validation

Four ways of representing validation information:

- Asserting conditions
- Adding special labels
  - End states
  - Progress cycles
  - Acceptance cycles
- Never (Büchi) automata
- Temporal logic – translated into never automata

Asserting a Condition

```
assert(a == 1 || b < 2)
```

- An assert statement can be inserted to express that condition must be fulfilled at certain point in execution
  - It is always executable
  - It has no effect provided result is non-zero
- Asserted expressions must be side-effect free
- Failing assertion will cause execution to be aborted

End States

When execution stops, each process has either reached the end or it is blocked

By default the only valid end states are those where process execution has completed

End labels used to indicate that also other states can represent valid end states

```
proctype S(chan in, chan out)
{  out!send(0) ;
   in?ack ;
   out!send(1) ;
   in?ack       }

proctype R(chan in, chan out)
{  bit val ;
end:
   do
   ::   out?send(val) ->
   in?ack
   od          }
```

Progress Cycles

- Loops may be just idling
- Loops may contribute useful work
- Non progress cycle analysis:
- Cycle which does not visit a progress labelled state will be an error

```
Proctype Send(chan in, chan out)
{
   bit sendVal, recVal ;
   do
   :: out!msg(sendVal) ->
        in?ack(recVal) ;
        if
        :: recVal == sendVal ->
Progress:    sendVal = 1 – recVal
        :: else -> skip
        fi
   od
}
```


Undesired Cycles

- May also be necessary to trace a cycle which is ”bad”
- Acceptance cycle analysis:
- Cycles which visit states labelled by acceptance label are in error

```
Proctype Send(chan in, chan out)
{
   bit sendVal, recVal ;
   do
   :: out!msg(sendVal) ->
        in?ack(recVal) ;
        if
        :: recVal == sendVal ->
             sendVal = 1 – recVal
        :: else ->
Accept:      skip
        fi
   od
}
```

Referencing Process States

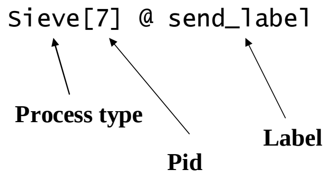
Process identifiers (pid's) are used to reference processes

Process referenced by process type along with pid

Process in particular state referenced by

- Process type
- Pid
- Label

Expression returns 0 if predicate false, o/w 1



Never Claims

How can we express a property such as *A*(*P U Q*) :

For all execution sequences, *P* remains true until *Q* becomes true, if ever ?

```
active proctype monitor()
{
   progress:
        do
        :: P -> Q
        od
}
```

"If *P* is sometime true, some time later, *Q* will become true"

```
active proctype monitor()
{
   progress:
        do
        :: P -> assert(P || Q)
        od
}
```

Execution can be interleaved

Q may become true undetected

Never Claims, cont'd

- Never claims used to *synchronously* monitor execution to detect prohibited execution sequences
- Never claim may "look" at execution, but not interfere
- So: no assignments, no message passing, no process creation, etc.
- Only one never claim in a Promela model
- Express the desired sequence in Linear Time Temporal Logic. Negate it. Transform to a Never claim. Verify.
- Supported by SPIN!

Temporal Logic

Temporal logic: Language for expressing properties (sequences) of states.

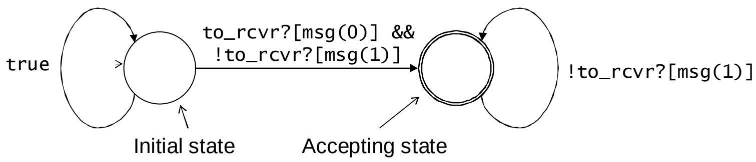
[]P	AGP	Always P
<>P	AFP	Eventually P
PUQ	A(PUQ)	P until Q
=>		Implication

In Alternating Bit Protocol it is always the case that if msg(0) has been sent then eventually msg(1) will be sent:

```
[](to_rcvr?[msg(0)] =>
        <>to_rcvr?[msg(1)])
```

Never Claim - Example

```
spec = [](to_rcvr?[msg(0)] => <>to_rcvr?[msg(1)])
```



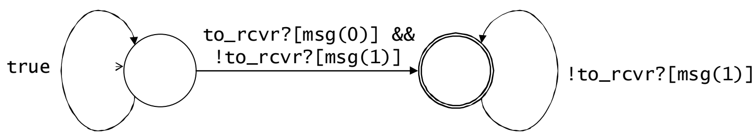
- Negation of `spec` – the ”bad” execution sequences:
- Some time

  `to_rcvr?[msg(0)] && !to_rcvr?[msg(1)]`

  becomes true and then forever after

  `!to_rcvr?[msg(1)]`

Never Claim Example, in Promela




```
never {
    do
    :: skip
    :: to_rcvr?[msg(0)] && !to_rcvr?[msg(1)] ->
         goto accept
    od ;
Accept:
    do
    :: !to_rcvr?[msg(1)]
    od }
```

If the Never claim can match the processes and detects an acceptance cycle then report an error

Some Practical Remarks

- Read chapter 6 and 7 in Holtzmann's book
- SPIN supports both simulation and exhaustive validation
- Use both!
- Do some simulations first
- Then try exhaustive validation
- Do not increase suggested available memory – it will only cause your computer to swap
- If SPIN reports out-of-memory switch to *supertrace*

Supertrace

- What if state size (*S*) and number of reachable states (*R*) do not fit into memory, i.e.

$$M < S*R \quad ?$$

- Use bit state hashing: Coverage can be increased dramatically by using two different hash functions
- Hash factor: Number of available bits / number of reached states
- Aim for hash factor > 100, otherwise you cannot have confidence in the result

More Hints

- If you find a deadlock at a large depth then do a revalidation but reduce max number of steps
- Use mtype in channel declarations to produce better MSC's

D-Spin

Experimental extension of Spin for

- Pointers
- Garbage collection
- Functions
- Function call stacks

Very useful for modelling Java-like programs (cf. JDK 1.2 assignment)