Modeling and Verification with SPIN

Wishnu Prasetya

wishnu@cs.uu.nl
www.cs.uu.nl/docs/vakken/pv
Overview

- Architecture & a bit more about SPIN
- SPIN’s modeling language
- Examples of models in SPIN

Acknowledgement: some slides are taken and adapted from Theo Ruys’s SPIN Tutorials.
Spin and Promela

- **SPIN** = Simple *Promela* *Interpreter*

- **Promela** = *Process Meta Language*
  - Is a *modelling* language! (not a language to build an application)

- **Strong features**:
  - Powerful constructs to synchronize concurrent processes
  - Cutting edge model checking technology
  - Simulation to support analysis (of the models)
Concurrency is a hot area again, now that we all use multi-core CPUs.

Other applications:


:: request?REQ2 -> { (resource == 0) ; resource=2 ; granted2?GRANTED }
od

active proctype customer1() {
  do
    { request!REQ1
      ; granted1?GRANTED
      ; skip /* representing customer1 uses the resource */
      ; resource = 0 /* after some time, freeing the resource again */
    }
  od
}
active proctype customer2() {
  do
    { request!REQ2
      ; granted2?GRANTED
      ; skip
      ; resource = 0
    }
  od
}

<starting simulation>
c:/apps/spin/spin425.exe -p

c:/apps/spin/spin425.exe -e -p

<done preprocessor>
c:/apps/spin/spin425.exe -f "![p]

#define p (resource==1)

Use Load to open a file or a template.

Symbol Definitions:

#define p (resource==1)

Never Claim:
:: (!([p])) -> goto accept_S4
:: (1) -> goto T0_init
fi;
accept_S4:
  if
    :: (!([p])) -> goto accept_S4
  fi;

Verification Result:
(X)SPIN Architecture

- deadlocks
- safety properties
- liveness properties

**Promela model** $M$ → **ispin** → **spin** → **spin command line tool**

- LTL Translator
- Simulator
- Verifier Generator

**C program** → **checker** → **false**

**Editing window**
- simulation options
- verification options
- MSC simulation window

**random guided interactive**

**pan.***

**pan.exe**
A system in SPIN consists of a set of interacting and concurrent processes.

Each process is sequential, but possibly non-deterministic.

Each process is built from atomic actions (transition).

Concurrent execution is modeled by interleaving.

Fairness can be imposed.
Interleaving model of concurrency

• Consider (with pseudo notation):

\[ P : \quad x++ \rightarrow x++ \]

\[ Q : \quad \text{print } x \]

Assume each arrow is \textit{atomic}.

• An execution of \( P \ | \ | Q \) abstractly proceeds as one of these paths:

\(\text{(note the interleaving)}\)
Degree of atomicity

- Whether it is reasonable to model a statement as ‘atomic’, depends on your situation.

  - `x++` usually no problem

  - `x>0 \rightarrow y:=x` ok, if we can lock both x and y

  - `0 \in S \rightarrow found:=true` ....?
Example

```plaintext
byte x = 1;

active proctype P1() { x++ ; assert (x==2) ; }

active proctype P2() { x-- ; }
```

(using a global variable to interact)
Data types

- Bit: 0,1
- Bool: true, false
- Byte: 0..255
- Short: \(-2^{15} .. 2^{15}-1\)
- Int: \(-2^{31} .. 2^{31}-1\)
- Pid: 0..255
- Mtype: 0..255 // user-def. enumeration
- Chan: 0..255

- One dimensional array
- Record
What you don’t have...

- No sophisticated data types
- No methods ; you have macro
- There are only 2 levels of scope:
  - global var (visible in the entire sys)
  - local var (visible only to the process that contains the declaration)
- there is no inner blocks
This process has 3 atomic actions.
The action “y==0”
- only enabled in a state where the expression is true
- it can only be executed when it is enabled; the effect is skip
- so, as long as it is disabled, the process will block
- if it is not enabled in the current state, a transition in another process may make it enabled in the next state.
- even if it is enabled in the current state, there is no guarantee the action will be selected for execution; but there is a way in SPIN to impose fairness.
Example

- Use it to **synchronize** between processes:

```c
byte x=0 , y=0

active proctype P { x++ ; (y>0) ; x-- }

active proctype Q { (x>0) ; y++ ; (x==0) ; y-- }

// both will terminate, but forcing Q to finish last
```
Multiprogramming is tricky…. 

- E.g. one or more processes can become stuck (deadlocked):

```c
byte x=0, y=0

active proctype P { x++; (y>0); x--; (y==0) }

active proctype Q { y++; (x>0); (x==0); y-- }
```

(6 potential executions…)
Processes can also synchronize with channels

```plaintext
chan c = [3] of {byte} ;

active proctype producer() {
    do
    :: c ! 0
    od
}

active proctype consumer() {
    do
    :: c ? x
    od
}
```
Channels

- for exchanging messages between processes
- finite sized and asynchronously, unless you set it to size 0 → synchronous channel
- Syntax:
  
  \[
  \begin{align*}
  \text{c ! 0} & \quad \text{sending over channel c; blocking if c is full} \\
  \text{c ? x} & \quad \text{receives from c, transfer it to x; blocking if c is empty} \\
  \text{d ? DATA, b, y} & \quad \text{match and receives}
  \end{align*}
  \]

- There are some more exotic channel operations: checking empty/full, testing head-value, copying instead of receiving, sorted send, random receive ... → check out the Manual

```
chan c       =  [0] of  {bit};
chan d       =  [2] of  {mtype, bit, byte};
chan e[2]    =  [1] of  {bit};
```

mtype = { DATA, ack }
The alternatives do not have to be atomic!
The first action in an alternative acts as its “guard”, which determines if the alternative is enabled on a given state.
Non-deterministically choose one enabled alternatives.
If there is none, the entire IF blocks.
“else” is a special expression that is enabled if all other alternatives block.
loop : do-statement

do
:: \textit{stmt}_1
:: ...
:: \textit{stmt}_n
od

- Non-deterministic, as in IF
- If no alternative is enabled, the entire loop blocks.
- Loop on forever, as long as there are enabled alternatives when the block cycle back.
- To exit you have explicitly do a \texttt{break}.
Non-determinism can be useful for modeling

```active proctype consumer() {
    do
    :: c ? x ;
    :: c ? x ; x=corrupted ; // to model occasional corrupted data
    od
}
```
Exiting a loop

\[
\text{do}
\begin{align*}
&:: \{ (i > 0) \ ; \ i-- \} \\
&:: \{ (i == 0) \ ; \ \textbf{break} \}
\end{align*}
\text{do}
\begin{align*}
&:: (i > 0) \rightarrow i-- \\
&:: (i == 0) \rightarrow \textbf{break}
\end{align*}
\text{do}
Label and jump

- Labels can also be useful in specification, e.g.

```plaintext
<> P@L0
```

- Referring to labels as above goes actually via a mechanism called “remote reference”, which can also be used to inspect the value of local variables for the purpose of specification.
Expressing local correctness with assertions

active proctype P …

active proctype Q { …; **assert** (x==0 && y==0) }  

*(here it implies that when Q terminates, x and y should be 0)*
But we can also express global invariant!

- Thanks to built-in non-determinism in the interleaving semantics, we can also use assertion to specify a global invariant!

```plaintext
byte x=0, y=0

active proctype P { x++ ; (y>0) ; x-- }

active proctype Q { (x>0) ; y++ ; (x==0) ; y-- }

active proctype Monitor { assert ((x==0 || x==1)) }
```

// implying that at any time during the run x is either 0 or 1
Deadlock checking

- When a system comes to a state where it has no enabled transition, but one of its processes is not in its terminal (end) state:
  - Deadlocked, will be reported by SPIN
  - But sometimes you want to model that this is ok → suppress it via the invalid-endstate option.

- The terminal state of a process P is by default just P’s textual end of code.
- You can specify additional terminal states by using end-label:
  - Of the form “end_1”, “end_blabla” etc
Expressing progress requirement

- We can mark some states as progress states
  - Using “progress*” labels

- Any *infinite execution* must pass through at least one progress label infinitely many often; else violation.

- We can ask SPIN (with an option) to verify no such violation exists (non-progress cycles option).
Dining philosophers

- N philosophers
- Each process:
  1. grab left and right fork simultaneously
  2. eat...
  3. release forks
  4. think............... then go back to 1
The processes in Promela

```
#define N 4
byte fork[N] ;
bool eating[N] ;

proctype P(byte i) {
    do
        :: (fork[i] == N && fork[(i + 1) % N] == N) -> {
            fork[i] = i
            fork[(i + 1) % N] = i ;

            eating[i] = 1 ;// eat ...
            eating[i] = 0 ;
            fork[i] = N ;
            fork [(i + 1) % N]= N
        }
    od
}
```
Creating processes and init { ... }

\textbf{init} { 
  byte \textit{i} ; 
  ... // initialize forks 
  \textit{i} = 0 ; 
  \textbf{do} 
  :: \textit{i}<\text{\textit{N}} \rightarrow \{ \text{run P(\textit{i})} ; \textit{i}++ ; \} 
  :: \textit{i}\geq\text{\textit{N}} \rightarrow \textbf{break} ; 
  \textbf{od} 
}

Put this in \textbf{atomic} { ... } ; 
Be aware of what it means!

What if we want to show that the algorithm is still correct for any initial value of forks, as long as you have at least one pair of forks free at the beginning, and that forks are only taken in pairs?
Using non-determinism to quantify over your data

```c
init {  
  // initializing the array x
  byte i = 0 ; byte v ;
  do  
  :: i>=N -> break ;  
  :: { if 
  :: v = N 
  :: v = i 
  fi ; 
  fork[i]=v ; fork[(i+1)%N]=v ; 
  i++ ;
  }  
  od ;
  ...
} // now create the processes as in the previous slide
```
How to express the specification?

```plaintext
proctype P(byte i) {
  do :: { atomic {
    (fork[i] == N && fork[(i + 1) % N] == N) ;
    fork[i] = i ;
    fork[(i + 1) % N] = i ;
  } ;
  eating[i] = 1 ;
  eating[i] = 0 ;
  fork[i] = N ;
  fork [(i + 1) % N] = N
  }
  od
}

assert (fork[i] == i && fork[(i+1)%N] == i)
```
Using a “monitor” process

```c
active proctype monitor() {
    byte i ;
    i = 0 ;
    do
    :: i>=N -> break ;
    :: i<N -> {
        assert (!eating[i] ||
                (fork[i]==i && fork[i+1%N]==i)) ;
        i++ ;
    }
  od
}
```

But we still can’t express that if a process is “hungry”, it will eventually eat. In this particular problem, we can still express it using progress labels. For more general temporal specification, we will look at the use of LTL formulas.
Example: Alternating bit protocol

- imperfect “connections”, but corrupted data can be detected (e.g. with checksum etc).

- Possible solution: send data, wait for a positive acknowledgement before sending the next one.

Just 1 bit is needed for the ack, hence the “bit” in the name.
You can think of several ways to work it out...


  - NPL Protocol
  - M<2 Protocol (we’ll discuss this one)

- For more, check out:

  http://spinroot.com/spin/Man/Exercises.html

  e.g. Go-Back-N Sliding Window Protocol
State 1 is the starting state, and its accepting state in the sense when the sender is in this state, it assumes the last data package it sent has been successfully received by the receiver, and so it fetches a new data package to send.
M<2 Protocol, Receiver part

1 // request Sender to resend

? error

2

!0

3

!1

?1, rd // Sender wants Receiver to resend

4

!1

?0, rd
Scenario: 1x error, corrected

Though each automaton is simple, the combined (and concrete) behavior is quite complex; \( \approx 100 \) states in my (abstract) SPIN model (there are more explicit states, if we take the “data” into account).
channel S2R = [BufSize] of { bit, byte };
channel R2S = [BufSize] of { bit };

proctype Sender (channel in, channel out) { ... }

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

init {
    run Sender(R2S, S2R) ;
    run Receiver(S2R, R2S)
}
Modelling in SPIN

```plaintext
proctype Receiver(chan in, out) {
  show byte rd ; /* received data */
  show bit cbit ; /* control bit */

  do
  :: in ? cbit, rd ;
  if
  :: (cbit == 1) -> out!1
  :: (cbit == 0) -> out!1
  :: printf("MSC: ERROR1\n") ; out!0
  fi
  od
}
```

So, how big the channels should be? Is 0 good enough?
A different style, with “goto”

```c
proc type Sender(chan in, out) {
    show byte data ; /* message data */
    show bit cbit ; /* received control bit */

    S1: data = (data+1) % MAX ; out!1,data ; goto S2;

    S2: in ? cbit ;
        if
        :: (cbit == 1) -> goto S1
        :: (cbit == 0) -> goto S3
        :: printf("MSC: AERROR1\n") -> goto S4
    fi ;

    S3: out!1,data ; goto S2 ;

    S4: out!0,data ; goto S2 ;
}
```
This time, not possible with assertions (at least not without the help of ‘something else’).

In LTL (to be discussed later), we can try something along this line:

\[ \square(\text{Receiver} @ S3 \rightarrow (\text{Receiver} @ rd == \text{Sender} @ data)) \]

But this still does not quite express the above.
Specification, using shadow variables

- Extend the model with ‘shadow variables’
  - Are used purely for expressing specifications
  - Must not influence the original behavior
- In our case:
  - exploit that sender generates new data by data+1
  - introduce a shadow variable “last” → previously accepted data
  - Impose this assertion on the acceptance state (of Receiver):

\[
\text{current data to be accepted} = \text{last} + 1
\]

Each data package, if accepted by the receiver, is accepted exactly once!
Extending the model

proctype Receiver(chan in, out) {
    show byte rd ; /* received data */
    show bit cbit ; /* control bit */
    do
        :: in?cbit,rd ;
        progress:
            if
                :: (cbit == 1) -> out!1
                :: (cbit == 0) -> out!1
                :: printf("MSC: ERROR1\n") ; out!0
            fi
    od
    assert (rd == (last+1) % MAX) ;
    last = rd ;
}

This is the Receiver’s accepting state S3
2x successive errors

Fetch a new data.

Ouch...
Ok... but suppose we still want to verify these:

**But, if error does not occur twice successively then: every pck sent, if accepted, is accepted exactly once.**

**If no error occur, every data sent will eventually be accepted.**

The first can be expressed simply by constraining the model, namely how it simulates error.

The 2\textsuperscript{nd} one can’t be expressed with just assertions and shadow variables.

Alternative: *LTL*. 
More on Promela
Exception/Escape

- S unless E

- Statement! Not to be confused with LTL “unless”.

- If E ever becomes enabled during the execution of S, then S is aborted and the execution continues with E.

  More precisely… check manual.
Predefined variables in Promela

- `_pid` (local var) current process’ instantiation number
- `_nr_pr` the number of active processes
- `np_` true when the model is not in a “progress state”
- `_last` the pid of process that executed last
- `else` true if no statement in the current process is executable
- `timeout` true if no statement in the system is executable

...
Timeout

- `timeout` becomes executable if there is no other process the system is executable/enabled
  - so, it models a global timeout
  - useful as a mechanism to avoid deadlock
  - beware of statements that are always executable.

```plaintext
do :: c ? x → ... :: timeout → break od
```