Introduction to Promela and SPIN

Cătălin Dima

LACL, Université Paris 12
Promela = Process Meta Language

- A **specification** language! No programming language!
- Used for system description:
  - Specify an **abstraction** of the system, not the system itself.
- Emphasize on **process synchronization & coordination**, not on computation.
- Promela uses **nondeterminism** as an abstraction technique.
- Suitable for software modeling, not for hardware.
A simulator for Promela programs.

And a verifier for the properties of Promela programs.

In simulation mode, SPIN gives quick impressions of system behavior.
  - Nondeterminism in specification is “randomly solved”.
  - No infinite behaviors.

In verification mode, SPIN generates a C program that constructs an implementation of the LTL model-checking algorithm for the given model.
  - Then one has to compile/run this C program to get the result.
  - ... which may provide a trace for the bugs in the model.
Promela program `hello.pml`:

```c
active proctype main(){
    printf("Hello world")
}
```

Simulating the program:

```bash
$ spin hello.pml
hello world
1 process created
```

- `proctype` = declares a new process type.
- `active` = instantiate one process of this type.
Producers/Consumers

mtype = { P, C }; /* symbols used */
mtype turn = P; /* shared variable */

active proctype producer(){
    do
        :: (turn == P) -> /* Guard */
            printf("Produce\n");
            turn = C
    od
}
active proctype consumer(){
    again:
        if
            :: (turn == C) -> /* Guard */
                printf("Consume\n");
                turn = P;
                goto again
        fi
}
Condition statements and nondeterminism

- **Proctype consumer rewritten:**

```c
again:
    (turn == C);
    printf("Consume\n");
    turn = P;
    goto again;
```

- **Condition statement**, blocking the process until the condition becomes true.

- **Nondeterminism:**

```c
byte count;
active proctype counter(){
    do
        :: count++
        :: count--
        :: (count==0) -> break
    od
}
```
Promela focuses on modeling distributed systems.

```promela
byte a;
active proctype p1()
{
    a=1;
    b=a+b
}
active proctype p2()
{
    a=2;
}
```

Atomicity needed for avoiding race conditions:

```promela
atomic{ a=1; b=a+b }
atomic{ tmp=y; y=x; x= tmp }
```
bool turn, flag[2];
byte cnt;
active [2] proctype proc(){
    pid i,j;
    i=_pid; j=1- _pid;
    again:
        flag[i]=true;
        turn=i;
        (flag[j]==false || turn !=i) ->
        cnt++;
        assert(cnt==1);
        cnt--;
        goto again;
}
Verifying Peterson’s algorithm

$ spin -a peterson.pml
   # creates several files named pan.c, pan.t,...
$ gcc -o pan pan.c
$ ./pan

Full statespace search for:
   never claim       - (none specified)
   assertion violations +
   acceptance cycles - (not selected)
   invalid end states +

State-vector 20 byte, depth reached 22, errors: 0
   38 states, stored
   25 states, matched
   63 transitions (= stored+matched)
   0 atomic steps

- Errors = 0 !
- The two assertions cnt==1 (one for each proctype) are satisfied in all runs !
Processes in more detail

- Process = instantiation of a proctype.
- Consisting of data declarations and statements.
- Always declared globally.
- Each running process has a unique pid,
  - Numbered in order of creation, starting with 0.
  - Referenced in the process by predefined \_pid.
- Possibility to differentiate output from one process from output from the others.
- Launching other processes with run:

  proctype pr(byte x) {
    printf("x=%d, pid = %d\n",x,_pid)
  }
  init {
    run pr(0); run pr(1);
  }

- init = declaration of a process active in the initial system state.
- Three processes created.
Promela processes behave like real processes.

\[ \text{run} \approx \text{fork+exec}. \]

\text{run} is an operator – a run statement may evaluate to false, and thus block the “parent” process!

- Number of created processes \( \leq 255 \).

\text{run} returns the PID of the launched process (like \text{fork}).

Process termination = end of its code.

A process can “die” only if all processes instantiated lated have died first.
bool var = true;
active proctype A() provided (var=true) {
    L: printf("A\n");
    var = false;
    goto L
}

active proctype A() provided (var=false) {
    L: printf("A\n");
    var = true;
    goto L:
}

- Each process may take a step only when its `provided` clause holds = invariant for that process.
- Strict alternation
Data objects

- Data can only be global or process local.
- Integer data types + bits + boolean.
- C syntax for variable declarations.
- One-dimensional arrays only.
- \texttt{mtype} = list of symbolic values, range 1..255.
  - A single list for a Promela program!
    \[
    \texttt{mtype} = \{ A, B, C \}; \\
    \texttt{mtype} = \{ 1, 2, 3 \}; /* union of the two sets */
    \]
- Record structures definable:
  
  \[
  \texttt{typedef Field}\{ \\
  \text{short f=3; byte g} \\
  \} \\
  \texttt{typedef Record}\{ \\
  \text{byte a[3];} \\
  \text{Field fld;} \\
  \}
  \]
- Can be used for defining multidimensional arrays.
Variables modeling communication channels between processes.
Must be declared globally, if needed by two distinct processes.

\[
\text{chan queue} = [10] \text{ of } \{ \text{mtype, short, Field} \}
\]

- 10 message buffer, each message composed of 3 fields.

Sending messages:

\[
\text{queue!expr1,expr2,expr3;}
\text{queue!expr1(expr2,expr3)}
\]

- expr1 used as message type indication.

Receiving messages:

\[
\text{queue?var1,var2,var3;}
\text{queue?var1(var2,var3)}
\]

Conditional reception:

\[
\text{queue?A(var2,var3);} \\
\text{queue?var1,100,var3} \\
\text{queue?eval(var1),100,var3}
\]

- Execute only when first field matches value of var1.
Other channel operations

- **Channel poll** – do not remove the message from the channel:

  ```
  queue?<eval(y),x>
a<b && queue?[msg] /* test for message, do not remove */
  ```

- **Sorted send**:

  ```
  queue!!msg /* inserted in lexicographic order */
  ```

- **Removing the first message matching some pattern**:

  ```
  queue??2,var2,var3
  ```

  Removes the first message whose first field is 2.

- **len(queue)** = buffer length.

- **Also** `empty(queue), nempty(queue), full(queue), nfull(queue)`.
Rendezvous communication

chan queue = [0] of { byte }

- The channel has zero buffering capacity.
- A send can only be executed when a corresponding receive is executed at the same time by some other process.

mtype = { id msg };
chan name = [0] of { mtype, byte };
active proctype A(){
    name!msg(100);
    name!id(10);
}
active proctype B(){
    byte var;
    if
    :: name?msg(var)  ->  printf("state = %d", var);
    :: name?id(var)    ->  printf("value = %d", var);
    fi
}

- Second send is blocking in proctype A.
Other channel operations

Channel values can be sent onto channels:

```promela
chan glob = [1] of { chan };
active proctype A(){
    chan loc = [1] of { byte }
    glob!loc;
    loc?var;
}
active proctype B(){
    chan who;
    glob?who;
    who!100;
}
```
Depending on system state, any statement is executable or blocked.

Expressions are statements that block when evaluating to \textit{false} or 0.

No need for “busy waiting”:

\begin{verbatim}
(a==b) /* behaves like while (a!=b) skip */ /* or like : 
do 
:: (a==b) -> break 
:: else -> skip 
od
\end{verbatim}
Control statements and inline definitions

- Have seen if, do and goto.
- break semantics as in C.
- Escape sequence:

  \{ P \} unless \{ E \}

  - Execute P unless first statement in E is executable.
  - When P terminates, the whole block terminates.

- Can define inline macros:

  ```
  inline swap(x, y) { 
    byte tmp = x; x = y; y = tmp 
  }
  ```

- No functions/procedures/modules.
- Reserved type `STDIN` for input:
  - Only one message type available on `STDIN: int`.
Correctness claims

The main part of Promela: placing claims on a program, that SPIN has to verify!

Various types:
- Basic assertions.
- End-state labels.
- Progress-state labels.
- Accept-state labels.
- Never claims.
- Trace assertions.
The simplest way to prove properties about a program: check that at a point in the program some property holds.

Basic assertion = `assert(expression)` : always executable.

But when `expression` evaluates to `false` or 0, an error message is triggered on output (and subsequent operations are done by SPIN).

Revisit the mutual exclusion example!

(Basic) assertions can be as complicated as desired:

```plaintext
assert(x=y && chan?[msg])
```
End states

- SPIN checks whether all processes reach terminate (i.e. reach their closing brace).
- Some processes are not intended to terminate:
  - Schedulers, servers, etc.
- Promela allows defining ending "states" (i.e. statements) in a process:
  - Not an error if the process linger in that state “forever”.
- Ending states for a process declared with labels starting with end.
- Example with a Dijsktra semaphore...
mtype = {p, v};
chan sema = [0] of { mtype }; active proctype Dijsktra() {
  end: do
    :: (count == 1) -> sema!p; count = 0;
    :: (count == 0) -> sema?v; count = 1;
  od
}
active [3] proctype user() {
  do
    :: sema?p;
    skip;
    sema!v
}
Progress states

- We may want to check that within each cycle through system states, something “desirable” happens.
- E.g.: lack of starvation.
- We may label some states with progress labels.
- This forces SPIN to check that each infinite execution passes through one of the statements labeled with progress labels.
- Special command-line options needed also for gcc/cc and pan:
  - `DNP` option for the compiler.
  - `-l` option for the verifier (i.e. `pan`).
bool turn, flag[2];
byte cnt;
active [2] proctype proc(){
    pid i,j;
    i=_pid;  j=1-_pid;
    again:
        flag[i]=true;
        turn=i;
        (flag[j]==false || turn !=i) ->
    progress:
        cnt++;  
        assert(cnt==1);
        cnt--;
        goto again;
}

Starvation freedom must be ensured in a correct mutual exclusion algorithm!
Weak fairness:

*If a process $P$ reaches a point where it has an executable statement, and the executability of that statement never changes, then $P$ should eventually proceed by executing the statement.*

Strong fairness:

*If a process $P$ reaches a point where it has an executable statement, and the executability of that statement occurs infinitely often from there on, then $P$ should eventually proceed by executing the statement.*

Enabling weak fairness: $\texttt{-f}$ option for $\texttt{pan}$. 
byte x;
active proctype A(){
    do
        :: x=2;
    progress: skip
    od
}
active proctype B(){
    do
        :: x=3;
    od
}

Each fair cycle is a progress cycle!

$ ./pan -l -f
Full statespace search for:
    ....
    non-progress cycles + (fairness enabled)
State-vector 20 byte, depth reached 4, errors: 0

Cătălin Dima          Promela & SPIN
“Never” claims

- Used for checking properties over sequences.
- Each temporal logic formula can be transformed into a “never” claim.
- “Never” claim = complement of the LTL formula that has to be checked.
- A “never” claim is like a new process whose traces must never occur as traces of the system.
- Similarly to an assert, SPIN checks that there exists no run of the system which is also a run of the code inside the “never” claim.

An example:

```promela
never{
  do
    :: !p -> break
    :: else
    od
}
```

- Checks that $p$ is true in any system state.
Never claims and temporal logic

- We would like to check the following property:

  *Every system state in which $p$ is true eventually leads to a system state in which $q$ holds, and in between $p$ must remain true.*

- An LTL formula for this:

  $$\Box(p \rightarrow pUq)$$

- A never claim for this:

  ```promela
ever{
  S0 : do
    :: p && !q -> break
    :: true
  od
  S1 :
  accept: do
    :: !q
    :: !(p || q) -> break
  od
}
```
Rules for specifying never claim

- Only statements that do not have side effects.
- Hence no assignments and no channel read/write.
- Channel poll operations and arbitrary condition statements are allowed.
- Some predefined variables can be used only in never claims.
- Accept states = formalize Büchi acceptance conditions for the never claim!
Can be generated from LTL formulas: the \( -f \) option for SPIN.

Grammar:

\[
\text{ltl ::= bop | (ltl) | ltl binop ltl | unop ltl}
\]

where

- **bop** is true or false.
- **unop** is [ ] (always), <-> (sometimes) or ! (negation).
- **binop** is \( U \) (until), \( V \) (dual of until), or the boolean operators: &&, ||, \( / \), \( \backslash \), \( - > \), <->.

The \( -D\text{NX}T \) option for SPIN adds also the nexttime operator \( X \).
Example:

```bash
$ spin -f '[]p'
never { /* []p */
accept_init:
T0_init:
    if
    :: ((p)) -> goto T0_init
    fi;
}
$ spin -f '!(<> p)'
....
$ spin -f '[] (p U (q U r))'
....
```

Atomic formulas \( p, q, r \) can be defined with macros in the Promela model:

```c
#define p (a<b)
#define q (len(x)<5 && a==b)
```
\_np is *false* in all system states where at least one running process is currently at a progress control-flow state.

\_last holds the instantiation number of the process that performed the last step.

\texttt{pc\_value(pid)} returns the current control state of the process having the \texttt{pid}.

\texttt{enabled(pid)} tells whether process \texttt{pid} has at least one statement executable in the current state.

\texttt{procname[pid]\_label} returns nonzero only if the next statement that can be executed by \texttt{pid} is labeled with \texttt{label}. 
Similar to never claims, but referring to message channels:

```plaintext
trace{
    do
        :: q1!a ; q2!b
    od
}
```

- Only simple send/receive statements (no ordering).
- No data objects can be declared in trace assertions.
- Don’t care values occurring on channels can be specified with the predefined variable `.`.
- May contain end states, progress states and accept states.
When the pan verifier generated by SPIN/gcc reports an error, it generates a trail file which shows the problem.

```
$ ./pan -l -f
pan: non-progress cycle (at depth 4)
pan: wrote fair.pml.trail
```

The trail file can be then interpreted by SPIN to show us the problem:

```
$ spin -t -p fair.pml
Starting A with pid 0
Starting B with pid 1
spin: couldn’t find claim (ignored)
....
```

Many other options for SPIN – check with spin --.