SPIN: Introduction and Examples*

Patrick Trentin
patrick.trentin@unitn.it
http://disi.unitn.it/~trentin

Formal Methods Lab Class, Feb 24, 2015

Università degli Studi di Trento

*These slides are derived from those by Stefano Tonetta, Alberto Griggio, Silvia Tomasi, Thi Thieu Hoa Le, Alessandra Giordani for FM lab 2005/14

Contents

The SPIN (= Simple Promela Interpreter) model checker

- Tool for formal verification of distributed and concurrent systems (e.g. operating systems, data communications protocols).
  - Developed at Bell Labs.
  - In 2002, recognized by the ACM with *Software System Award* (like Unix, TeX, Smalltalk, Postscript, TCP/IP, Tcl/Tk).
  - Automated tools convert programs written in Java or in C into SPIN models.
- The modelling language is called PROMELA.
- SPIN has a graphical user interface, ISPIN.
- Materials:
  - Homepage: `http://spinroot.com/spin/whatispin.html`
  - Manual: `http://spinroot.com/spin/Man/index.html`

PROMELA (= Protocol/Process Meta Language)

- PROMELA is suitable to describe concurrent systems:
  - dynamic creation of concurrent processes.
  - (synchronous/asynchronous) communication via message channels.
- Programs written in PROMELA can be simulated/verified.
- Simulation shows one execution.
  - *random*, *interactive* or *guided*.
  - not useful for finding bugs!
- Verification checks every execution looking for a counterexample.
  - *exhaustive* or *approximate* verification of correctness properties.
  - a counterexample is a computation that violates a correct property.

Basic commands

- To simulate a program:

  `spin system.pml`
- Interactively:

  `spin -i system.pml`
- To generate a verifier (`pan.c`):

  `spin -a system.pml`
- To run a guided simulation:

  `spin -t model.pml`
- To run ISPIN:

  `ispin model.pml`

Useful commands:

- To see available options: `spin --`
- To display processes moves at each simulation step: `spin -p system.pml`
- To display values of global variables: `spin -g system.pml`
- To display values of local variables: `spin -l -p system.pml`

Contents

Hello world!

```
active proctype main()
{
        printf("hello world\n")
}
```

- **active** instantiates one process of the type that follows.
- **proctype** denotes that *main* is a process type.
- *main* identifies the process type, it's not a keyword.
- Note that ';' is missing after **printf**:
    - ';' is a statement separator, not a statement terminator.

Hello world! Alternative

```
init {
        printf("hello world\n")
}
```

- **init** is a process that initializes the system.
- Initially just the initial process is executed.

Hello world! Alternative

```
init {
        printf("hello world\n")
}
```

- **init** is a process that initializes the system.
- Initially just the initial process is executed.

Simulation:

```
> spin hello.pml
      hello world
1 process created
```

- One process was created to simulate the execution of the model.

Producers/Consumers

```
mtype = { P, C };
mtype turn = P;
active proctype producer(){
        do
        :: (turn == P) ->
                printf("Produce\n");
                turn = C
        od
}
active proctype consumer(){
       do
       :: (turn == C) ->
                printf("Consume\n");
                turn = P
       od
}
```

Producers/Consumers (Language Details)

- **mtype** defines symbolic values
  (similar to an enum declaration in a C program).
- *turn* is a global variable.
- **do ... od** (do-statement) defines a loop.
- Every option of the loop must start with '`::`'.
- `(turn == P)` is the guard of the option.

- A `break`/`goto` statement can break the loop.
- `->` and `;` are equivalent
  (`->` indicates a causal relation between successive statements).
- If all guards are false, then the process blocks
  (no statement can be executed).
- If multiple guards are true, we get non-determinism.

Producers/Consumers

Simulation:

```
> spin prodcons.pml | more
Produce
        Consume
Produce
        Consume
Produce
        Consume
Produce
        Consume
Produce
        Consume
...
```

Producers/Consumers Extended

There can be multiple **running instances** of the same *proctype*:

```
active [2] proctype producer {...}
...
active [2] proctype consumer {...}
```

Simulation:

```
> spin prodcons2_flaw.pml | more
        Produce
                        Consume
                Consume
        Produce
Produce
                Consume
...
```

Producers/Consumers Extended

There can be multiple **running instances** of the same *proctype*:

```
active [2] proctype producer {...}
...
active [2] proctype consumer {...}
```

Simulation:

```
> spin prodcons2_flaw.pml | more
        Produce
                        Consume
                Consume
        Produce
Produce
                Consume
...
```

Concurrent execution: after each (**atomic**) statement, a new process can be (**randomly**) scheduled for execution.

Producers/Consumers Extended

Reason:

```
> spin -i prodcons2_flaw.pml
Select a statement
choice 3: proc  1 (producer) prodcons2_flaw.pml:7 (state 4) [((turn==P))]
choice 4: proc  0 (producer) prodcons2_flaw.pml:7 (state 4) [((turn==P))]
Select [1-4]: 3
Select a statement
choice 3: proc  1 (producer) prodcons2_flaw.pml:9 (state 2) [printf('Produce\\n')]
choice 4: proc  0 (producer) prodcons2_flaw.pml:7 (state 4) [((turn==P))]
Select [1-4]: 3
          Produce
Select a statement
choice 3: proc  1 (producer) prodcons2_flaw.pml:10 (state 3) [turn = C]
choice 4: proc  0 (producer) prodcons2_flaw.pml:7 (state 4) [((turn==P))]
Select [1-4]: 4
Select a statement
choice 3: proc  1 (producer) prodcons2_flaw.pml:10 (state 3) [turn = C]
choice 4: proc  0 (producer) prodcons2_flaw.pml:9 (state 2) [printf('Produce\\n')]
Select [1-4]:
```

Problem: Both processes can pass the guard `(turn == P)` and execute `printf("Produce")` before `turn` is set to `C`.

Producers/Consumers Extended

A correct declaration for the producer:

```
active [2] proctype producer()
{
        do
        :: request(turn, P, N) -> // if turn==P then turn=N
                printf("P%d\n", _pid);
                assert(who == _pid); // "who" is producing
                release(turn, C) // turn=C
        od
}
```

- `assert`: if expression is false (i.e. zero) then abort the program, else ignored.
- `_pid` is a predefined, local, read-only variable of type `pid` that stores the unique ID of the process.

Producers/Consumers Extended

Definition of `request`:

```
inline request(x, y, z) {
        atomic { x == y -> x = z; who = _pid }
}
```

- **inline** functions like C macros.
  - the body is directly pasted into the body of a proctype at each point of invocation.
- **atomic**: prevents the scheduler from changing the running process until all the statements are executed.
  - no interleaving with statements of other processes!
- The executability of the atomic sequence is determined by the first statement.
  - i.e. if `x==y` is true then the atomic block is executed.

Producers/Consumers Extended

File prodcons2.pml:

```
mtype = { P, C, N };
mtype turn = P;
pid   who;
... // request
inline release(x, y) { atomic { x = y; who = 0 } }
... // proctype producer
active [2] proctype consumer()
{
  do
    :: request(turn, C, N) ->
       printf("Consume %d\n", _pid);
       assert(who == _pid);
       release(turn, P)
  od
}
```

Producers/Consumers Extended

Simulation:

```
> spin prodcons2.pml | more
          P1
                    C3
      P0
                    C3
          P1
                    C3
          P1
               C2
      P0
                    C3
          P1
...
```

Producers/Consumers Extended

Simulation **can detect** errors:

```
init { assert(false) }

> spin false.pml
spin: line   1 "false.pml", Error: assertion violated
spin: text of failed assertion: assert(0)
#processes: 1
  1:    proc  0 (:init:) line   1 "false.pml" (state 1)
1 process created
```

However, simulation **can not prove** that the code is bug-free!

Producers/Consumers Extended

A **verifier** checks that an assertion is never violated.

We use SPIN to generate the verifier of *prodcons.pml*:

```
> spin -a prodcons2.pml
> 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 28 byte, depth reached 7, errors: 0
...
```

Producers/Consumers Extended

Back to the flawed Producers/Consumers

```
mtype = { P, C };

mtype turn = P;

int msgs;

active [2] proctype producer()
{
  do
    :: (turn == P) ->
         printf("Produce\n");
         msgs++;
         turn = C
  od
}
```

```
active [2] proctype consumer()
{
  do
    :: (turn == C) ->
         printf("Consume\n");
         msgs--;
         turn = P
  od
}


active proctype monitor() {
  assert(msgs >= 0 && msgs <= 1)
}
```

```
> spin -a prodcons2_flaw_msg.pml && gcc -o pan pan.c && ./pan
```

Producers/Consumers Extended (Trail File)

Trail File

`prodcons2_flaw.pml.trail` contains SPIN's transition markers corresponding to the contents of the stack of transitions leading to error states

Meaning:

- Step number in execution trace
- Id of the process moved in the current step
- Id of the transition taken in the current step

```
-4:-4:-4
1:1:0
2:1:1
3:1:2
4:1:3
5:3:8
6:3:9
7:3:10
8:2:8
9:2:9
10:3:11
11:2:10
12:4:16
```

```
> spin -t -p prodcons2_flaw_msg.pml
```

The Mutual Exclusion problem

General algorithm

```
active [2] proctype mutex()
{
again:
        /* trying section */

        cnt++;
        assert(cnt == 1);        /* critical section */
        cnt--;

        /* exit section */
        goto again
}
```

The Mutual Exclusion problem (First tentative)

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

active [2] proctype mutex() {
again:
  flag != 1;  /* It models "while (flag == 1) wait!" */
  flag = 1;
  cnt++;
  assert(cnt == 1);
  cnt--;
  flag = 0;
  goto again
}
```

The Mutual Exclusion problem (First tentative)

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

active [2] proctype mutex() {
again:
  flag != 1;  /* It models "while (flag == 1) wait!" */
  flag = 1;
  cnt++;
  assert(cnt == 1);
  cnt--;
  flag = 0;
  goto again
}
```

Assertion violation: Both processes can pass the `flag != 1` before flag is set to 1.

The Mutual Exclusion problem (Second tentative)

```
bit x, y;   /* signal entering/leaving the section */
byte cnt;
```

```
active proctype A() {
again:
  /* A waits for B to end */
  x = 1;
  y == 0;
  cnt++;
  /* critical section */
  assert(cnt == 1);
  cnt--;
  x = 0;
  goto again
}
```

```
active proctype B() {
again:
  y = 1;
  x == 0;
  cnt++;
  /* critical section */
  assert(cnt == 1);
  cnt--;
  y = 0;
  goto again
}
```

The Mutual Exclusion problem (Second tentative)

```
bit x, y;   /* signal entering/leaving the section */
byte cnt;

active proctype A() {
again:
  /* A waits for B to end */
  x = 1;
  y == 0;
  cnt++;
  /* critical section */
  assert(cnt == 1);
  cnt--;
  x = 0;
  goto again
}
```

```
active proctype B() {
again:
  y = 1;
  x == 0;
  cnt++;
  /* critical section */
  assert(cnt == 1);
  cnt--;
  y = 0;
  goto again
}
```

Invalid-end-state: Both processes can execute `x = 1` and `y = 1` at the same time and will then be waiting for each other.

Dekker/Dijkstra algorithm

```
/* trying section */
flag[i] = true;
do
   :: flag[j] ->
      if
         :: turn == j ->
            flag[i] = false;
            !(turn == j);
            flag[i] = true
         :: else -> skip
      fi
   :: else ->
      break
od;
```

Dekker/Dijkstra algorithm

```
/* trying section */
flag[i] = true;
do
    :: flag[j] ->
        if
            :: turn == j ->
                flag[i] = false;
                !(turn == j);
                flag[i] = true
            :: else -> skip
        fi
    :: else ->
        break
od;
```

```
/* initialization */
pid i = _pid;
pid j = 1 - _pid;
```

```
/* exit session */
turn = j;
flag[i] = false;
```

Dekker/Dijkstra algorithm

Verification:

```
> spin -a dekker.pml
> cc -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 67, errors: 0
...
```

Doran&Thomas change

Is the outer loop really necessary?

```
flag[i] = true;
if
:: flag[j] ->
        if
        :: turn == j ->
                flag[i] = false;
                !(turn == j);
                flag[i] = true
        :: else -> skip
        fi
:: else
fi;
```

Doran&Thomas change

Verification:

```
> spin -a doran.pml
> cc -o pan pan.c
> ./pan
...
pan: assertion violated (cnt==1) (at depth 117)
pan: wrote doran.pml.trail
...
```

`doran.pml.trail` contains a counterexample with length 117.

Doran&Thomas change

We can use a breadth-first search to find the shortest counterexample:

```
> cc -DBFS -o pan pan.c
> ./pan
...
pan: assertion violated (cnt==1) (at depth 12)
pan: wrote doran.pml.trail
...
```

Doran&Thomas change

Now, we can perform a guided simulation:

```
> spin -p -t doran.pml
  1:    proc  1 (mutex) line   8 ... [i = _pid]
  2:    proc  1 (mutex) line   9 ... [j = (1-_pid)]
  3:    proc  1 (mutex) line  11 ... [flag[i] = 1]
  4:    proc  1 (mutex) line  21 ... [else]
  5:    proc  1 (mutex) line  24 ... [cnt = (cnt+1)]
  6:    proc  0 (mutex) line   8 ... [i = _pid]
  7:    proc  0 (mutex) line   9 ... [j = (1-_pid)]
  8:    proc  0 (mutex) line  11 ... [flag[i] = 1]
  9:    proc  0 (mutex) line  13 ... [(flag[j])]
 10:    proc  0 (mutex) line  19 ... [else]
 11:    proc  0 (mutex) line  19 ... [(1)]
 12:    proc  0 (mutex) line  24 ... [cnt = (cnt+1)]
```

Peterson algorithm

A correct improvement:

trying session

```
flag[i] = true;
turn = i;
!(flag[j] && turn == i) ->
```

exit session

```
flag[i] = false;
```

Verification:

```
> spin -a peterson.pml
> cc -o pan pan.c
> ./pan
...
State-vector 20 byte, depth reached 41, errors: 0
...
```

Exercises

- Simulate `you_run2.pml` and `you_run3.pml`.
- Verify `prodcons3.pml`.
- Verify `mutex_flaw.pml`.
- Delete "`turn==i`" in Peterson and verify the correctness.

C Pan's Output Format

```
> ./pan
pan: assertion violated ((x!=0)) (at depth 11)
pan: wrote model.pml.trail
```

Assertion Violation

- SPIN has found a execution trace that violates the assertion
- the generated trace is 11 steps long and it is contained in `model.pml.trail`

C Pan's Output Format

```
(Spin Version 6.0.1 -- 16 December 2010)
+ Partial Order Reduction
```

Meaning

1. Version of Spin that generated the verifier
2. Optimized search technique

C Pan's Output Format

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

Meaning

1. Type of search: exhaustive search (Bitstate search for approx.)
2. No never claim was used for this run
3. The search checked for violations of user specified assertions
4. The search did not check for the presence of acceptance or non-progress cycles
5. The search checked for invalid endstates (i.e., for absence of deadlocks)

C Pan's Output Format

```
State-vector 32 byte, depth reached 13, errors: 0
```

Meaning

1. The complete description of a global system state required 32 bytes of memory (per state).
2. The longest depth-first search path contained 13 transitions from the initial system state.
   - `./pan -mN` set max search depth to `N` steps
3. No errors were found in this search.

C Pan's Output Format

```
      74 states, stored
      30 states, matched
     104 transitions (= stored+matched)
       1 atomic steps
1.533  memory usage (Mbyte)
```

Meaning

1. A total of 74 unique global system states were stored in the statespace.
2. In 30 cases the search returned to a previously visited state in the search tree.
3. A total of 104 transitions were explored in the search.
4. One of the transitions was part of an atomic sequence.
5. Total memory usage was 1.533 Megabytes,

C Pan's Output Format

```
unreached in proctype ProcA
      line 7, state 8, "Gaap = 4"
      (1 of 13 states)
unreached in proctype :init:
      line 21, state 14, "Gaap = 3"
      (1 of 19 states)
```

Meaning

A listing of the state numbers and approximate line numbers for the basic statements in the specification that were not reached $\Rightarrow$ since this is a full statespace search, these transitions are effectively unreachable (dead code).

C Pan's Output Format

```
error: max search depth too small
```

Meaning

It indicates that search was truncated by depth-bound (i.e. the depth bound prevented it from searching the complete statespace).

- `./pan -m50`
  sets a bound on the depth of the search

Nota Bene

When the search is bounded, SPIN will not be exploring part of the system statespace, and the omitted part may contain property violations that you want to detect $\Rightarrow$ you cannot assume that the system has no violations!