# SPIN: Introduction and Examples *

Alessandra Giordani
agiordani@disi.unitn.it
http://disi.unitn.it/~agiordani

Formal Methods Lab Class, September 28, 2014


Università degli Studi di Trento

*These slides are derived from those by Stefano Tonetta, Alberto Griggio, Silvia Tomasi, Thi Thieu Hoa Le for FM lab 2011/13

# 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 executed/simulated.
- 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 -I -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

The producer's definition is equivalent to:

```
active proctype producer()
{
again:  if
        :: (turn == P) ->
                printf("Produce\n");
                turn = C
        fi;
        goto again
}
```

- goto transfers control to the statement labeled by again.

# Producers/Consumers

Also equivalent to:

```
active proctype producer()
{
again:  if
        :: (turn == P) ->
                printf("Produce\n");
                turn = C
        :: else -> goto again
        fi;
        goto again
}
```

- `else` is only executable if all other options are not executable.

# Producers/Consumers

Also equivalent to:

```
active proctype producer()
{
again:  (turn == P) ->
                printf("Produce\n");
                turn = C;
        goto again
}
```

# Producers/Consumers

Also equivalent to:

```
active proctype producer()
{
again:  (turn == P) ->
                printf("Produce\n");
                turn = C;
        goto again
}
```

- If the boolean expression does not hold, execution blocks until it does.

# Producers/Consumers

Simulation:

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

# Producers/Consumers Extended

We can extend the example to more processes for each type:

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

The alternation is no more guaranteed. Simulation:

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

# 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` aborts the program if the expression is false (i.e. zero), otherwise it is just passed.
- `_pid` is a predefined, local, read-only variable of type `pid` that stores the instantiation number of the executing process.

# Producers/Consumers Extended

Definition of `request`:

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

- **inline** functions like C macros.
  - their body is directly pasted into the body of a proctype at each point of invocation.
- **atomic**: when it starts, the process will keep running until all steps will complete.
  - 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;

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

inline release(x, y) {
        atomic { x = y; who = 0 }
}
...
```

# 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:

```
> 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 cannot prove that errors do not exist!

# Producers/Consumers Extended

To prove that the assertions cannot be violated, we generate a verifier:

```
> 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.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.pml
```

# 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!