Spin: Introduction*

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

Formal Methods Lab Class, February 23, 2018

Università degli Studi di Trento

(compiled on 17/05/2018 at 12:49)

*These slides are derived from those by Stefano Tonetta, Alberto Griggio, Silvia Tomasi, Thi Thieu Hoa Le, Alessandra Giordani, Patrick Trentin for FM lab 2005/18

Course Overview

Course: covers two tools for model checking and formal verification

- Part I: SPIN
- Part II: NUXMV

Slides + Examples + Solutions:

http://disi.unitn.it/trentin/teaching/fm2018/fm2018.html

$\Longrightarrow$ the slides' content updated wrt. last year

Exam:

- examples + solutions will be provided
- short manuals of both tools available during exam

$\Longrightarrow$ thus: code that does not even compile is significantly penalized

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).
- The modelling language is called PROMELA.
  - dynamic creation of concurrent processes.
  - (synchronous/asynchronous) communication via message channels.

  $\Longrightarrow$ *(= Protocol/Process Meta Language)*
- Automated tools convert Java/C programs into SPIN models.
- SPIN has a graphical user interface, ISPIN.
- docs:
  - homepage: http://spinroot.com/spin/whatispin.html
  - manual: http://spinroot.com/spin/Man/index.html

Model Checking

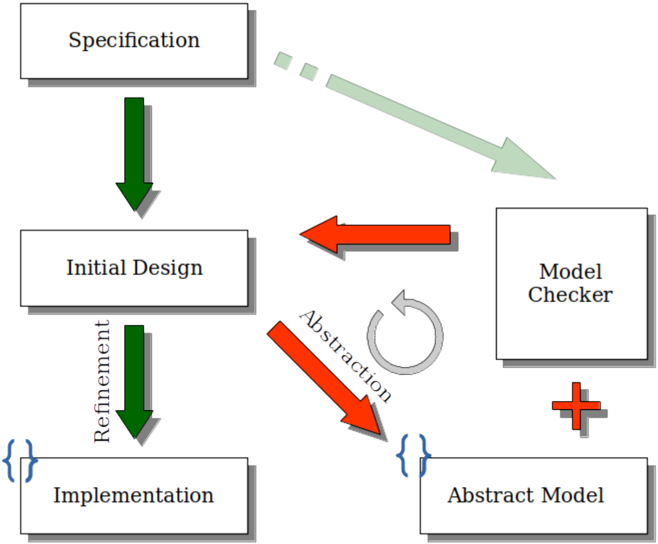
Model Checking: $M \models \varphi$

problem of **formally verifying**, in an *automated* and *exhaustive* way, that a given **system model** M **matches** a given **specification** φ, *i.e.* a set of *logical properties*.

Common Design Flaws

- **starvation**: no access to a resource
- **deadlock**: complete absence of progress
- **race conditions**: unprotected critical sections
- **constraints violation**: buffer overrun, overflows, ...
- **under-specified model**: unexpected behaviours
- **over-specified model**: dead code / unreachable states
- **...**

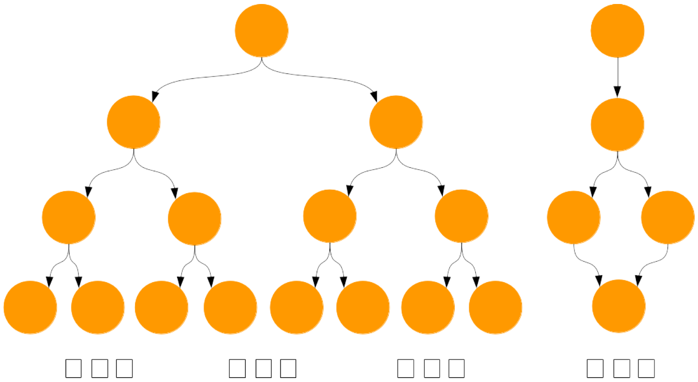
"Classic" Model Checking



Execution Model

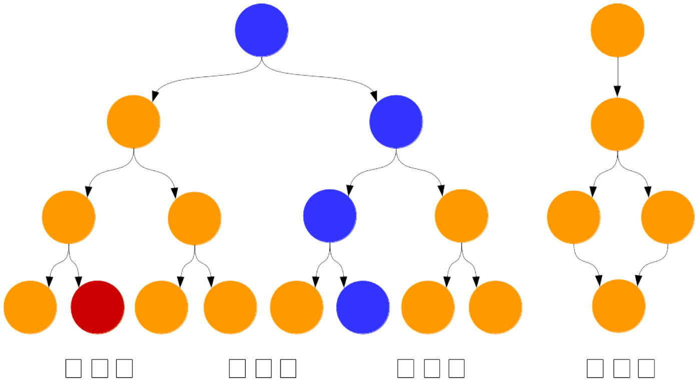
- one entry point

  $\Longrightarrow$ *no loss of generality, a single entry-point can always be imposed*
- finite number of states
- large # of execution paths *(possibly exponential)*



Model Simulation

- can be *random*, *interactive* or *guided*.
- useful for inspection of a PROMELA model
- **cannot prove** that code is bug free!

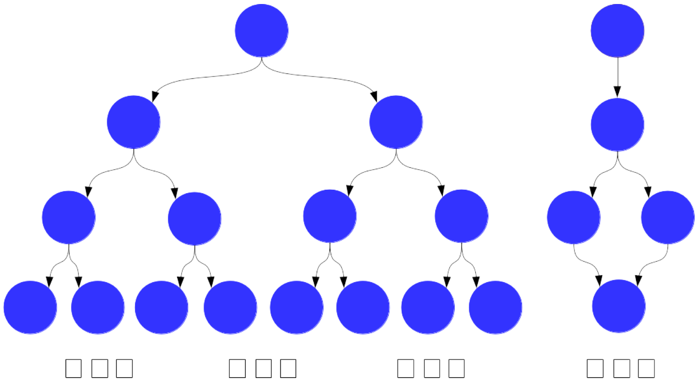


Model Verification

- looks for a property counterexample

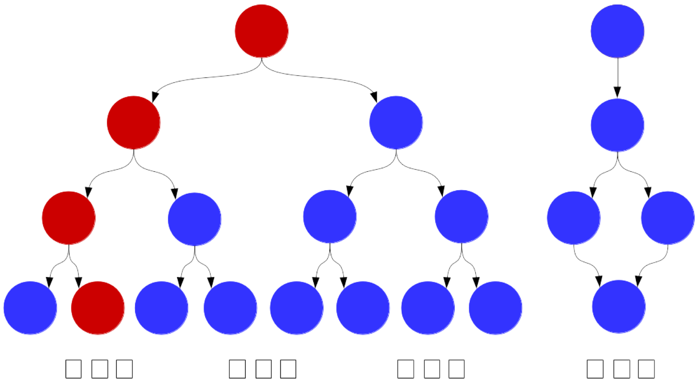
  $\Longrightarrow$ *an execution trace that* **falsifies** *a given property*

- can be *exhaustive* or *approximate*

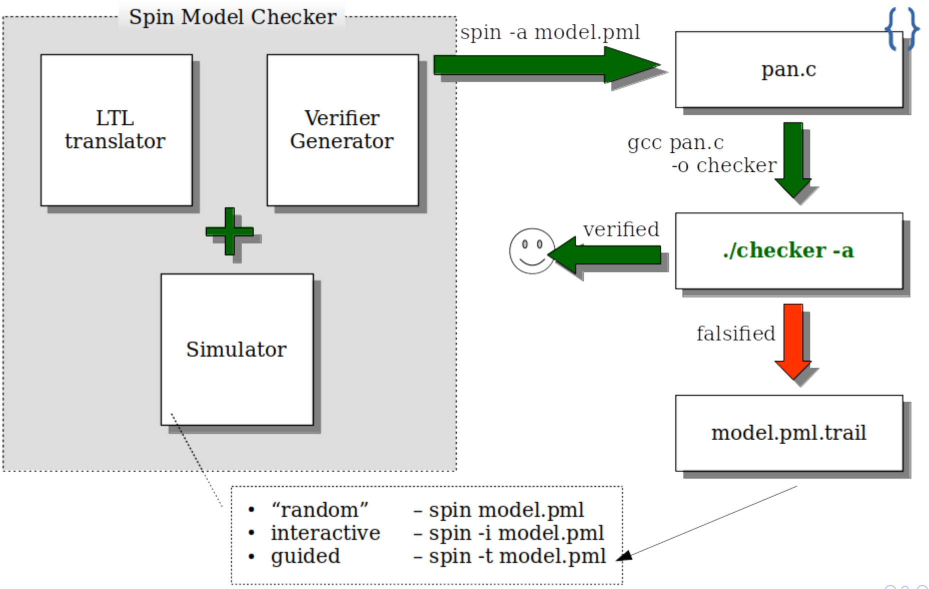


Model Verification: counterexample

- stored in the current directory with *".trail"* extension
- can be replayed with *-t* option



Spin Model Checker: usage [1/2]



Spin Model Checker: usage [2/2]

Other useful options:

- `spin --`: see available options
- `-p`: print each statement executed
- `-g`: print all global variables
- `-l`: print all local variables
- `-nN`: seed for random number generator
- `-search`: generate a verifier, compile and run it
  - `-dfs`: use depth-first search (default)
  - `-bfs`: use breadth-first search
  - `-ltl p`: verify property with name `p`
  - `-a`: search for acceptance cycles

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* is **not a keyword**: any *generic identifier* would do (e.g. foo)
- Note that ';' is missing after **printf**:
  - ';' is a statement separator, not a statement terminator.

Hello world! Alternative

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

- **init** identifies a *"special"* `active proctype`

  $\Longrightarrow$ *commonly used to initialize the system*

Hello world! Alternative

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

- **init** identifies a *"special"* `active proctype`

  $\Longrightarrow$ *commonly used to initialize the system*

Output:

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

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

Producers/Consumers [1/3]

```
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 [2/3]

- **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).
- a loop can have **multiple guards**:
  - 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 [3/3]

Output:

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

Producers/Consumers Extended [1/4]

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

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

Output:

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

Producers/Consumers Extended [1/4]

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

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

Output:

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

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

Producers/Consumers Extended [2/4]

```
> 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 producers can pass beyond the guard `(turn == P)` and execute `printf("Produce")` before `turn` is set to `C`.

Producers/Consumers Extended [3/4]

Use a **monitor** to check the number of items is alway between 0 and 1

```
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 [4/4]

Trail File

`prodcons2_flaw_msg.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
```

Fixed Producers/Consumers Extended [1/5]

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.

Fixed Producers/Consumers Extended [2/5]

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.

Fixed Producers/Consumers Extended [3/5]

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
}
```

Output:

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

⟹ The simulation looks fine, but it is not enough!

Fixed Producers/Consumers Extended [5/5]

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
...
```

$\Longrightarrow$ **verifier** necessary to ensure that the specification is never violated.

The Mutual Exclusion problem

Goal: two processes modify a shared resource without race conditions

General Approach:

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

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

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

The Mutual Exclusion problem (First attempt)

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

active [2] proctype mutex() {
again:
  flag != 1;  /* while (flag == 1) wait(); */
  flag = 1;

  cnt++;
  assert(cnt == 1);
  cnt--;

  flag = 0;
  goto again
}
```

The Mutual Exclusion problem (First attempt)

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

active [2] proctype mutex() {
again:
  flag != 1;  /* while (flag == 1) wait(); */
  flag = 1;

  cnt++;
  assert(cnt == 1);
  cnt--;

  flag = 0;
  goto again
}
```

Assertion Violation: both processes can go beyond the `flag != 1` barrier before flag is set to 1

The Mutual Exclusion problem (Second attempt)

```
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:
  /* B waits for A to end */
  y = 1;
  x == 0;

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

  y = 0;
  goto again
}
```

The Mutual Exclusion problem (Second attempt)

```
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:
  /* B waits for A to end */
  y = 1;
  x == 0;

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

  y = 0;
  goto again
}
```

Invalid End-State: A and B end up waiting for each other forever if they execute `x = 1` and `y = 1` at the same time

Dekker/Dijkstra algorithm [1/2]

```
/* 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 [2/2]

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
...
```

Peterson algorithm

Peterson Implementation:

```
/* 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 ⇒ 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 $\Rightarrow$ one cannot assume that the system has no violations!