Spin introduction

Simple Promela Interpreter

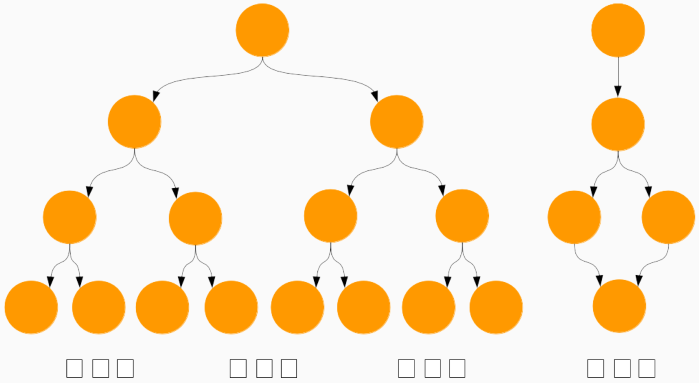
Enrico Magnago

University of Trento,
Fondazione Bruno Kessler

Theory recap

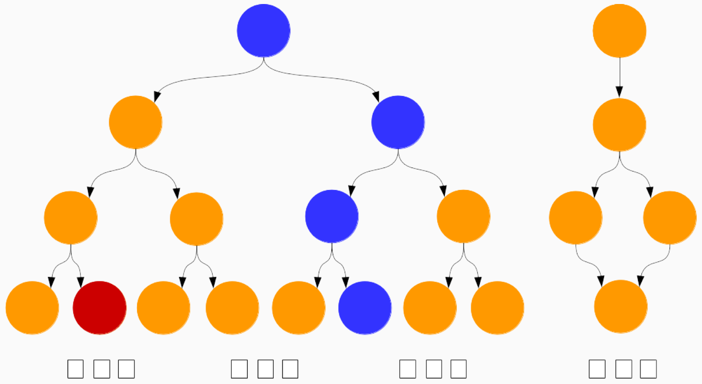
Model language

Model M: formal description of a system.
$\mathcal{L}(M)$: set of all possible executions.



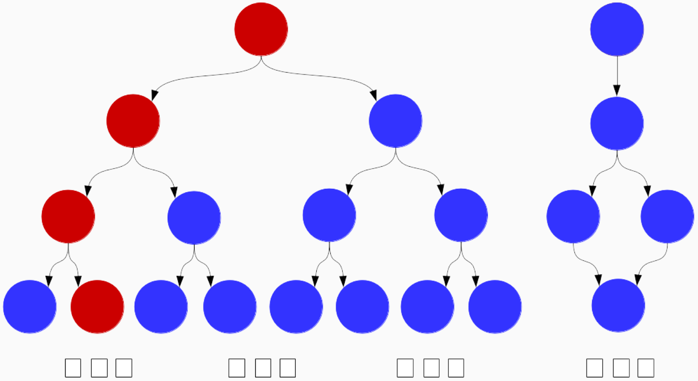
Model simulation

Inspect one of the possible executions: $\sigma \in \mathcal{L}(M)$.
Model simulation does NOT prove correctness, it can only find bugs (testing).



Model checking

Search for an execution that falsifies the property: **counter-example**.



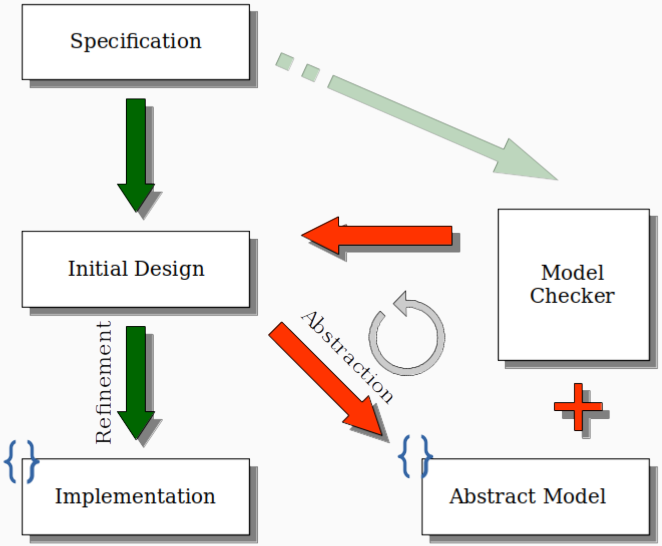
Model validation

Nice! We proved that the model is correct.
Then the system must be bug-free!
Absolutely FALSE!

What can go wrong?

- Is the model actually representing our system?
- Is our model capturing all relevant aspects of the system?
- Have we used the right level of abstraction?
- Have we missed some properties?
- ...

Development process



System Properties

Recall
Model checking problem $M \models \phi$, does ϕ hold in every possible execution of M?
If $M \not\models \phi$ the model checker provides a **counter-example**: an execution $\sigma \in \mathcal{L}(M)$ such that $\sigma \models \neg\phi$.

Properties

Catch design flaws:

- **deadlock**: no progress;
- **starvation**: no access to a resource;
- **over-specification**: unreachable states;
- **under-specification**: unexpected/undesired behaviours;

Two main categories of properties: **safety**, **liveness**.

Spin

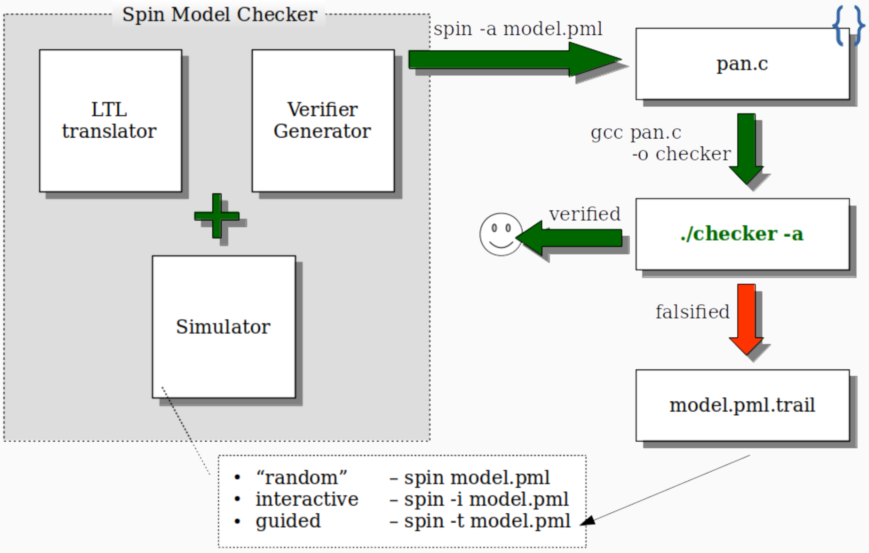
Spin model checker

- Developed at Bell Labs starting in 1980.
- SPIN performs the formal verification distributed, concurrent systems (e.g. mutual exclusion, communication protocols).
- Modelling language is PROMELA, it supports dynamic creation of concurrent processes and both synchronous and asynchronous communication via message channels.

resources

- web site: http://spinroot.com

Spin workflow



Spin main 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

Exercises

Download and install Spin

Setup Spin

Go to http://spinroot.com/spin/Man/README.html and follow the instructions

Hello World!

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

- **`proctype`**: `whatever` is a process type.
- **`active`**: the process is automatically created and started, otherwise we could decide when to instantiate it by implementing the `init` method.
- notice that there is no semicolon after the `printf` statement, in this case it is optional. In PROMELA the semicolon is used to separate statements.

hello.pml

Yet another Hello World!

```
init
{
  pid p;
  printf("Yet another...");
  p = run hello();
  printf("\n%d\n", p)
}

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

- this time we use the `init` function to explicitly instantiate the process.
- how many processes are being executed?
- what are the possible outcomes? (Hint: scheduling).

yet_another_hello.pml

Tired of Hello World? Producer-Consumer

```
mtype = { P, C };  /* define 2 symbolic values: P and C */
mtype turn = P;    /* global variable */
active proctype producer()
{
  do  /* loop */
  :: (turn == P) ->  /* guard of the case */
          printf("Produce\n");
          turn = C
  od
}
active proctype consumer()
{
  do
  :: (turn == C) ->
          printf("Consume\n");
          turn = P
  od
}
```

prodcons.pml

Producer-Consumer Cont.

- `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.

Producer-Consumer: let's get bigger

```
mtype = { P, C };
mtype turn = P;
active [2] proctype producer() /* create 2 producers */
{
  do
    :: (turn == P) ->
        printf("Produce\n");
        turn = C
  od
}
active [2] proctype consumer() /* create 2 consumers */
{
  do
    :: (turn == C) ->
       printf("Consume\n");
       turn = P
  od
}
```

prodcons2.pml

Producer-Consumer: not so fast

- Each statement is **atomic**, but **any** process can be scheduled for execution.
- Both producers can pass the guard `turn == P` and execute the `printf("Produce\n")` statement before the `turn` is set to `C`.

Producer-Consumer: let’s check

Add a **monitor** to check the number of items is always between 0 and 1.

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

`msgs` is a global variable of type `int`, the producers increment this variable by 1, and the consumers decrement it by 1. Use the following commands to ask SPIN to check if there exists an execution that violates the assertion:

- `spin -a prodcons2_flaw_msgs.pml`
- `gcc -o prodcons_monitor pan.c`
- `./prodcons_monitor`

prodcons2_monitor.pml

Producer-Consumer: counter-example

Trail File

`prodcons2_flaw_msgs.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
```

Re-execute trace using the command:

```
spin -t -p prodcons2_flaw_msgs.pml
```

Producer-Consumer: fix-it, producer

```
pid who;  /* global variable */
inline request(x, y, z) { /* perform statements atomically */
  atomic { x == y -> x = z; who = _pid }
}
inline release(x, y) {
  atomic { x = y; who = 0 }
}
active [2] proctype producer()
{
  do
    :: request(turn, P, N) ->  /* atomic compare and set */
         printf("Produce %d\n", _pid);  /* built-in _pid */
         assert(who == _pid);  /* I am producing */
         release(turn, C)  /* turn = C */
  od
}
```

The `atomic` statements are executable iff the first statement is `true`.

Producer-Consumer: fix-it, consumer

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

Consumer symmetric to the producer.
Simulate using: `spin prodcons2.pml` prodcons2.pml

Producer-Consumer: verify

The following commands can be used to let Spin search for an execution that violates the assertions.

- `spin -a prodcons2.pml`
- `gcc -o prodcons2 pan.c`
- `./prodcons2`

Output:

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

State-vector 52 byte, depth reached 7, errors: 0
...
```

Mutual exclusion

Problem
Multiple processes want to access a shared resource without race conditions.

General solution structure
The access to the **critical section** (CS) is protected by the **trying section** and followed by the **exit section**.

- The **trying section** must guarantee that only 1 process can be in CS at a time.
- The **exit section** must release all resources acquired by the trying section, and allow other processes to enter CS.

Mutual exclusion: naive

```
 1 bit flag; /* signal entering/leaving the section */
 2 byte cnt; /* # procs in the critical section */
 3 active [2] proctype mutex() {
 4 again:
 5   flag != 1; /* equivalent to 'while(flag == 1) wait' */
 6   flag = 1;
 7
 8   cnt++;
 9   printf("Process %d entered critical section.\n", _pid);
10   assert(cnt == 1);
11   cnt--;
12
13   printf("Process %d exited critical section.\n", _pid);
14   flag = 0;
15   goto again
16 }
```

BUG: both processes can pass line 5 before line 6 is ever executed.
mutex_naive.pml

Mutual exclusion: deadlock

```
bit x, y;
byte cnt;
```

```
active proctype A() {
again:
  x = 1;
  y == 0;

  cnt++;
  /* critical section */
  printf("A in CS\n");
  assert(cnt == 1);
  cnt--;

  printf("A not in CS\n");
  x = 0;
  goto again
}
```

```
active proctype B() {
again:
  y = 1;
  x == 0;

  cnt++;
  /* critical section */
  printf("B in CS\n");
  assert(cnt == 1);
  cnt--;

  printf("B not in CS\n");
  y = 0;
  goto again
}
```

Mutual exclusion: Dekker-Dijkstra algorithm

```
bool turn;
bool flag[2];
byte cnt;
active [2] proctype mutex()
{
  pid i, j;
  i = _pid;
  j = 1 - _pid;
again:
  flag[i] = true;
  do
    :: flag[j] -> if
         :: turn == j ->
              flag[i] = false;
              !(turn == j);
              flag[i] = true
         :: else -> skip
       fi
    :: else -> break
  od;
```

```
  cnt++;
  assert(cnt == 1); /* CS */
  cnt--;

  turn = j;
  flag[i] = false;
  goto again
}
```

dekker.pml

Mutual exclusion: verification

Commands

- `spin -a dekker2.pml`
- `gcc -o dekker pan.c`
- `./dekker`

Output

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

State-vector 28 byte, depth reached 51, errors: 0
...
```

Mutual exclusion: Peterson algorithm

```
bool turn, flag[2];
byte cnt;
active [2] proctype mutex()
{
  pid i, j;
  i = _pid;
  j = 1 - _pid;
again:
  flag[i] = true;
  turn = i;
  !(flag[j] && turn == i) ->
  cnt++; assert(cnt == 1);
  cnt--;
  flag[i] = false;
  goto again
}
```

peterson.pml

- `spin -a peterson.pml`
- `gcc -o peterson pan.c`
- `./peterson`

Spin's output

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!

Exercises

- simulate you_run2.pml and you_run3.pml, what are the possible outcomes?
- verify prodcons3.pml
- verify mutex_flaw.pml
- what happens if we delete `turn == i` in the Peterson algorithm?