Software Validation and Verification
Section II: Model Checking

Topic 3. Promela and SPIN

Pedro Cabalar

Department of Computer Science and IT
University of Corunna, SPAIN
cabalar@udc.es

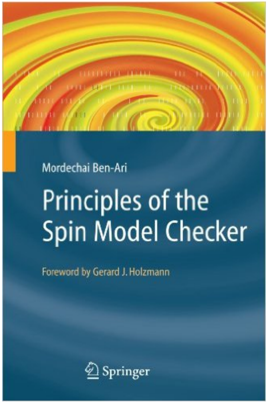
22 de febrero de 2023

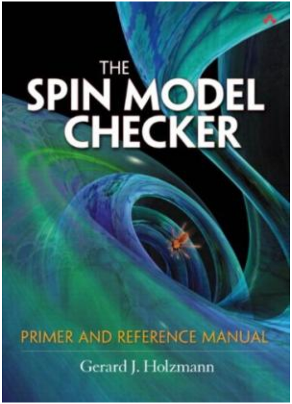
Outline

1. Sequential programs
2. Concurrency

Promela/SPIN

- We will use the tool SPIN (Simple Promela INterpreter)
  `http://spinroot.com`
- Bibliography:



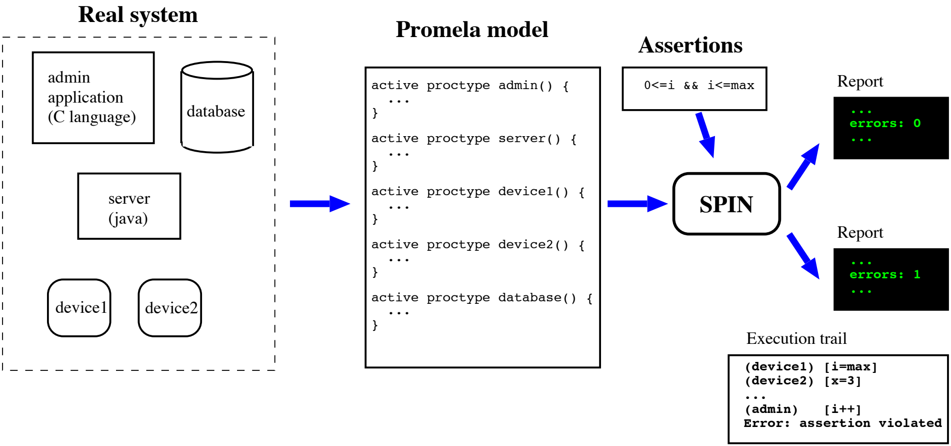


Promela language

- Language = PROMELA
  PROtocol/PROcess MEta LAnguage
- It is not a "programming" language!
  The analyzed (target) code will be written in Java, C, python, . . .
  or even in all of them!
- It is a modelling and specification language that allows describing the (concurrent) behavior of a protocol or a set of processes

SPIN model checker

- The specification is the input of model checker SPIN
- SPIN performs exhaustive search for errors



Promela

- Hello world!

```
init {
  printf("Hello World!\n")
}
```

- A first example of sequential program.

```
init {
 int value=123;
 int reversed;
 reversed=
     (value % 10)*100     \
   +((value/10)%10)*10   \
   +(value/100);
 printf("value=%d,reversed=%d\n",value,reversed)
}
```

Promela

- Operators and assignments: mostly like C or Java
- Exception: `++` and `--` can only be postfix and shouldn't be combined with other assignments.
- Small variation: C conditional operator `(cond ? expr1 : expr2)` is written `(cond -> expr1 : expr2)`. Example:

```
active proctype P() {
  int a=1,b=3;
  int max = (a>=b -> a : b);
  printf("max=%d\n",max)
}
```

- `define` statements work as in C

```
#define N 10
#define sum(a,b) ((a)+(b))
```

- Types

Type	Values	Size (bits)
`bit, bool`	$0, 1$,`false`,`true`	1
`byte`	$0 \ldots 255$	8
`short`	$-32768 \ldots 32767$	16
`int`	$-2^{31} \ldots 2^{31} - 1$	32
`unsigned`	$0 \ldots 2^{n} - 1$	≤ 32

- We also have a special type called `mtype` that allows symbolic values

```
mtype={red,yellow,green};
mtype light=green;
```

- We can only define `mtype` once. We cannot define different `mtype`'s

Promela

- The `printf` statement works as in C but limited to:

`%c`	`a single character`
`%d`	`a decimal value`
`%e`	`an mtype constant`
`%o`	`an unsigned octal value`
`%u`	`an unsigned integer value`
`%x`	`a hexadecimal value`

- We have a `skip` statement: increases the instruction pointer but does nothing else

Promela

- Conditional statement:

$$
\begin{array}{llll}
\texttt{if} & & & \\
\texttt{::} & \textit{condition}_1 & \texttt{->} & \textit{statement}_1 \\
 & & \vdots & \\
\texttt{::} & \textit{condition}_n & \texttt{->} & \textit{statement}_n \\
\texttt{fi} & & &
\end{array}
$$

- Semantics: non-deterministically take any true condition and proceed to execute its statement
- If all conditions are false, then wait until one becomes true (or forever!)

Promela

- An example:

```
init {
  int i=3,x;
  if
  :: i>1  -> x=1
  :: i==5 -> x=2
  :: i<0  -> x=3
  fi;
  printf("x=%d\n",x);
}
```

- Try with `i=5`, `i=-2` or `i=0`
- Message `timeout` means no enabled command to execute next

Promela

- Watchout: there is no default else

```
if
:: i==0 -> j++
fi
```

this gets blocked if $i \neq 0$. For a default else you should write

```
if
:: i==0 -> j++
:: i!=0 -> skip
fi
```

or use the `else` clause (the conjunction of negations for rest of conditions)

```
if
:: i==0 -> j++
:: else -> skip
fi
```

Promela

- An example using non-deterministic conditional

```
init {
  int a=3, b=3, max, branch;
  if
    :: a>=b -> max=a; branch=1
    :: a<=b -> max=b; branch=2
  fi;
  printf("max=%d, branch=%d\n",max,branch);
}
```

- Make several executions. Do you always get the same value for `branch`?
- Exercise: make a program that prints $x = 1$, $x = 2$ or $x = 3$ non-deterministically

- Iterative statement:

$$\begin{array}{llll} \texttt{do} & & & \\ \texttt{::} & \mathit{condition}_1 & \texttt{->} & \mathit{statement}_1 \\ & & \vdots & \\ \texttt{::} & \mathit{condition}_n & \texttt{->} & \mathit{statement}_n \\ \texttt{od} & & & \end{array}$$

- Semantics: non-deterministically take any true condition and proceed to execute its statement.
- If that statement is `break`, exit the loop. Otherwise, repeat.
- If all conditions are false, then wait until one is true (or forever!)
- Exercise: print all numbers from 1 to 10

Promela

- An example combining `do` and `if`

```
mtype = {red, yellow, green};
mtype light=green;

init {
  do
  :: if
     :: light==red    -> light=green
     :: light==yellow -> light=red
     :: light==green  -> light=yellow
     fi;
     printf("The light is now %e\n",light)
  od
}
```

Promela

- If you execute

```
$ spin tlight.pml
```

  you get an infinite loop

- You can fix a limit in the number of execution steps ($\neq$ loop iterations)

```
$ spin -u40 tlight.pml
```

- Exercise: remove the 3rd branch fixing `light=red` and execute again. What happens? Why does it happen?
- When we reach `light==yellow` the `if` gets "blocked" (no true condition to follow).
- Notice the difference with respect to an infinite (but enabled) execution

Promela

- In Promela, a condition can be used as a statement

```
int x=1;
init {
  (x>0);
  printf("%d\n",x);
}
```

- Semantics: `(x>0)` is a test. If the condition holds, we skip to the next statement.
- If not, it waits forever (`timeout`)...
  or perhaps until another process modifies `x`
  (`x` is a global variable = shared memory)

Promela

- An example of loop: Euclid's algorithm for Greatest Common Divisor

```
init {
  int x=15, y=20;
  int a=x, b=y;
  do
  :: a>b -> a=a-b
  :: b>a -> b=b-a
  :: a==b -> break
  od;
 printf("The GCD of %d and %d is = %d\n",x,y,a);
}
```

- Note the need for a `break` statement

- Exercise: make a program that non-deterministically prints any number between `1` and some constant `N`. Use, for instance:

```
#define N 10
```

Assertions

- We can use the `assert condition` clause to introduce assertions. If the condition is not satisfied, execution stops and shows the violated assertion
- Try this (erroneous) variation:

```
init {
  int x=15, y=20;
  int a=x, b=y;
  do
  :: a>b -> a=a-b+1
  :: b>a -> b=b-a
  :: a==b -> break
  od;
 printf("The GCD of %d and %d is = %d\n",x,y,a);
 assert (x%a==0 && y%a==0);
  // necessary test: a is some common divisor
}
```

Assertions

Exercise: fill the gap for the postcondition

```
init {
  int x=15, y=4;
  int q, r;
  assert (x>=0 && y>0); // precondition
  q=0;
  r=x;
  do
  :: r>y  -> q++; r=r-y
  :: else -> break
  od;
  printf("%d/%d = %d; remainder %d\n", x,y,q,r);
  assert ________________;
}
```

Try with $x = 15$ and $y = 3$. Does it work?

Assertions

- Consider this (wrong) program

```
init {
  int a=3, b=3, max;
  if
    :: a>=b -> max=a
    :: a<=b -> max=b+1
  fi;
  assert (max==a || max==b) && max>=a && max>=b;
}
```

- The program is wrong, but in some executions, the assertion violation is not raised
- This is because we are using simulation mode. However, the interesting use of SPIN is exhaustive verification mode (model checking).

Model checking

- To throw an exhaustive verification, we must actually perform these steps

  ```
  spin -a max.pml
  gcc -o pan pan.c
  pan
  ```

  or abbreviated in the single call

  ```
  spin -search max.pml
  ```

  `gcc -o pan pan.c -Wformat-overflow=0`
  Use this option in Linux gcc to avoid excessive number of warnings
- If errors are found (deadlocks or violated assertions) a `.trail` file will be generated (counterexample). In this case `max.pml.trail`
- We can execute the file using `-t` option. We can use `-p` to see all the steps.

  ```
  spin -t -p -l max.pml
  ```

Model checking

- Exercise: execute this program and try to get 100

```
#define N 100
init {
  int i=1;

  do
    :: break
    :: i<N -> i++
  od;
  printf("%d\n",i)
}
```

- Is 100 a probable output? (compute the probability)
- Is 100 a possible output? use the model checker to answer

Model checking

http://spinroot.com/spin/Man/1_Exercises.html

- Exercise: simulate this program

```
init {          // file: ex_1a.pml
 byte i=0;
 do
 :: i++
 od
}
```

```
spin -u514 -p -l ex_1a.pml
```

Try the following:

- Simulate without `-u514`.
- Exhaustive search: how many reachable states should you we get?
- Change the variable type to `bool` or to `short`
  Hint: default search depth = 10000 steps (truncates at 9999). Use option `-m70000`

Outline

Macros

Promela has no procedures or subroutines. We can only use macros:

```
// single line macro
#define printnum(n) printf("%d\n",n)

// multiple line macro
inline printnums(a,b) {
  int i=a;
  do
  :: i<=b -> printnum(i); i++
  :: i>b -> break;
  od
}

init
{ printnums(1,10) }
```

Concurrent processes

- Instead of procedures, we can define concurrent processes
- We may declare a process using `proctype` and simultaneously launch it using `active`

```
active proctype P() {
    ...
}
active proctype Q() {
    ...
}
```

- Processes are assigned a number (`_pid`) by their creation order in the source code. `P` will be process 0 and `Q` process 1.

Interleaving

- Important: after creation, execution is interleaved. Example:

```
byte n=0;
active proctype P() {
    n=1;
    printf("Process P, n = %d\n",n);
}
active proctype Q() {
    n=2;
    printf("Process Q, n = %d\n",n);
}
```

- How many different executions can we get?

Interleaving

- Important: after creation, execution is interleaved. Example:

```
byte n=0;
active proctype P() {
    n=1;
    printf("Process P, n = %d\n",n);
}
active proctype Q() {
    n=2;
    printf("Process Q, n = %d\n",n);
}
```

- How many different executions can we get?

Exercise 1

We have two sequential processes P and Q, with $m > 0$ and $n > 0$ instructions respectively. How many different interleaved executions do we get?

Interleaving

- We can declare a group of statements to be `atomic`
- Example: modify the previous code as follows

```
byte n=0;
active proctype P() {
    atomic {
      n=1;
      printf("Process P, n = %d\n",n);
    }
}
active proctype Q() {
    n=2;
    printf("Process Q, n = %d\n",n);
}
```

Interleaving

- Non-determinism: choose the next enabled instruction. Example:

```
int n = 5;    // try also n=0, n=-5
active proctype P() {
  do
  :: n<10 -> printf("%d\n",n); n++
  :: n==5 -> n = 7
  :: n==10 -> break
  od
}
active proctype Q() {
  if   :: n<0 -> n=0   fi
}
active proctype R() {
  (n==5);
  printf("five\n")
}
```

Example: critical section

- An example of critical section problem. P and Q use some common resource

```
active proctype P() {
  do
  :: printf("Non-critical section P\n");
     wantP=true;
     printf("Critical section P\n");
     wantP=false;
  od
}
active proctype Q() {
  do
  :: printf("Non-critical section Q\n");
     wantQ=true;
     printf("Critical section Q\n");
     wantQ=false;
  od
```

Mutual exclusion

- How can we check that both do not enter the critical section?
- Solution 1: use a global counter `critical` plus an assertion

```
active proctype P() {
  do
  :: printf("Non-critical section P\n");
     wantP=true;
     critical++;
     printf("Critical section P\n");
     assert (critical<=1);
     critical--;
     wantP=false;
  od
}
active proctype Q() {
... // idem
}
```

Mutual exclusion

- If you try, `spin -p critical.pml` several times, you'll probably violate the assertion at some point
- If you want to be exhaustive, generate a `pan.c` as before
- Suppose we try to avoid this using a "stop condition". For instance, P waits for `wantQ` to be false and vice versa

Synchronization

```
bool wantP=false, wantQ=false;
byte critical=0;
#define mutex (critical <=1)

active proctype P() {
  do
  :: printf("Non-critical section P\n");
     wantP=true;
     !wantQ;
     critical++;
     printf("Critical section P\n");
     critical--;
     wantP=false;
  od
}
```

Synchronization

```
active proctype Q() {
  do
  :: printf("Non-critical section Q\n");
     wantQ=true;
     !wantP;
     critical++;
     printf("Critical section Q\n");
     critical--;
     wantQ=false;
  od
}
```

- Execute several times: assertion is not violated but ...
- New surprise: if we execute several times, simulation stops with a `timeout` !

Deadlock

- Spin generates a `timeout` when there is no next statement available
- In other words: all processes are waiting = deadlock
- If you use exhaustive verification
  `spin -search critical3.pml`
  you'll find a trail where both `wantP` and `wantQ` become 1, and both processes are waiting for them to be 0

Deadlock vs mutual exclusion

- But can we guarantee that critical section satisfies mutual exclusion exhaustively?
- For this, we can use `spin -a` combined with a temporal formula.
- Example: remove the `asserts` and add instead

```
#define mutex (critical<2)
```

- To check that $\Box$*mutex* is true (critical is always lower than 2), we negate the formula and use option `-f` as follows:

```
spin -a -f '![]mutex' critical4.pml
```

Deadlock vs mutual exclusion

- Instead of using “ghost” variable `critical` we can use statement labels

```
bool wantP=false, wantQ=false;
active proctype P() {
  do
  :: printf("Non-critical section P\n");
     wantP=true;
     !wantQ;
 cs: printf("Critical section P\n");
     wantP=false;
  od
}
```

Deadlock vs mutual exclusion

```
active proctype Q() {
  do
  :: printf("Non-critical section Q\n");
     wantQ=true;
     !wantP;
cs:  printf("Critical section Q\n");
     wantQ=false;
  od
```

- To check mutual exclusion we would write

```
spin -a -f '![]!(P@cs && Q@cs)' critical5.pml
```

or simply

```
spin -a -f '<>(P@cs && Q@cs)' critical5.pml
```

Semaphores

- We still get an error due to the deadlock. If we want to ignore deadlocks we must use `-E` option in `pan` executable

```
pan -E
```

- Can we avoid the deadlock? Idea: using a semaphore
- Two atomic operations: wait (or P) and signal (or V)
- In our case: wait for process P would be

```
atomic {
  !wantQ;
  wantP=true;
}
```

and signal for P would just be `wantP=false`

Semaphores

- A general semaphore is just an integer variable `sem`.

```
inline wait(sem) {    // macro definition
  atomic {
    sem>0;
    sem--
  }
}
inline signal(sem) {
  sem++
}
```

Semaphores

- A binary semaphore is just a Boolean variable `sem`.

```
inline wait(sem) {    // macro definition
  atomic {
    sem;
    sem=false
  }
}
inline signal(sem) {
  sem=true
}
```

- Exercise: modify the example to use a single binary semaphore `mutex` instead WantP and WantQ. Check mutual exclusion and absence of deadlocks with `spin -a -f ...`

Multiple copies of processes

- We can define multiple copies of a given process by declaring `active[N]`

```
active[3] proctype P() {
  do
  :: printf("Non-critical section P\n");
     wait(mutex);
 cs: printf("Critical section P\n");
     signal(mutex)
  od
}
```

- Process numbers are assigned incrementally following the source code ordering.
- Variable `_pid` returns the current process identifier.
  Variable `_nr_pr` returns the number of active processes

Process `init`

- For a more elaborated creation of processes, we can use `init`. If defined, `init` becomes process 0 and the first to be executed.
- We define the process code without the `active` keyword. We can use parameters.

```
proctype P(byte c; int n) {
  printf("Executing process %c (%d)\n", c, n);
}
```

- Then, the `init` process may create active copies of a process with the special instruction `run`.

```
init {
  printf("Starting...\n");
  run P('A', 0);
  run P('B', 1);
  printf("All process terminated\n")
}
```

Process `init`

- Exercise: the previous example does not guarantee that message `All process terminated` is the last one to be printed. Can you find a way to force that situation?

Liveness

- Suppose that process Q has this modified code

```
active proctype Q() {
  do
  :: printf("Non-critical section Q\n");
     atomic {
       !wantP;
       wantQ=true;
     }
     false;
cs:  printf("Critical section Q\n");
     wantQ=false;
  od
}
```

- The critical section `cs` is unreachable!

Liveness

- We can use instruction labels in formulas: `procname@label` to check that `procname` is at instruction `label`. With multiple copies of a process, we use the `pid` number: `procname[pid]@label`
- To check liveness, i.e., that Q reaches `cs`, we use:

```
spin -a -f '!<>(Q@cs)' critical7.pml
gcc -o pan pan.c
pan -a -f
```

- `pan -a -f` looks for a counterexample in the form of infinite loop (we repeat an infinite sequence of steps that prevents reaching `cs`)
- When we execute the trail, `spin -p -t critical7.pml` it will show «`START OF CYCLE`» to show the infinite loop
- To check a fairness condition, we would use `-f '![]<>cs'` (cs is reached infinitely often)

Interleaving

Solution: we will execute $m+n$ instructions we can locate at

1	2	3	4	5	...	$m+n$
P	Q	P	P	Q	...	P

- Each interleaved execution can be seen as a selection of a set of positions where P takes the CPU (the rest will be taken by Q).
- In the example: P's positions are the m elements in $\{1,3,4,\ldots,m+n\}$
- The number of possible sets that can be formed corresponds to combinations of $m+n$ taken in groups of m:

$$\binom{m+n}{m} = \frac{(m+n)!}{(m+n-m)!\ m!} = \frac{(m+n)!}{n!\ m!}$$

- For instance, for $m=n=2$ we get $\frac{4!}{2!\ 2!}=6$ but for $m=n=3$ we obtain $\frac{6!}{3!\ 3!}=20$