Model Checking with SPIN

Deepak Kathayat
Somay Jain

Hello World!

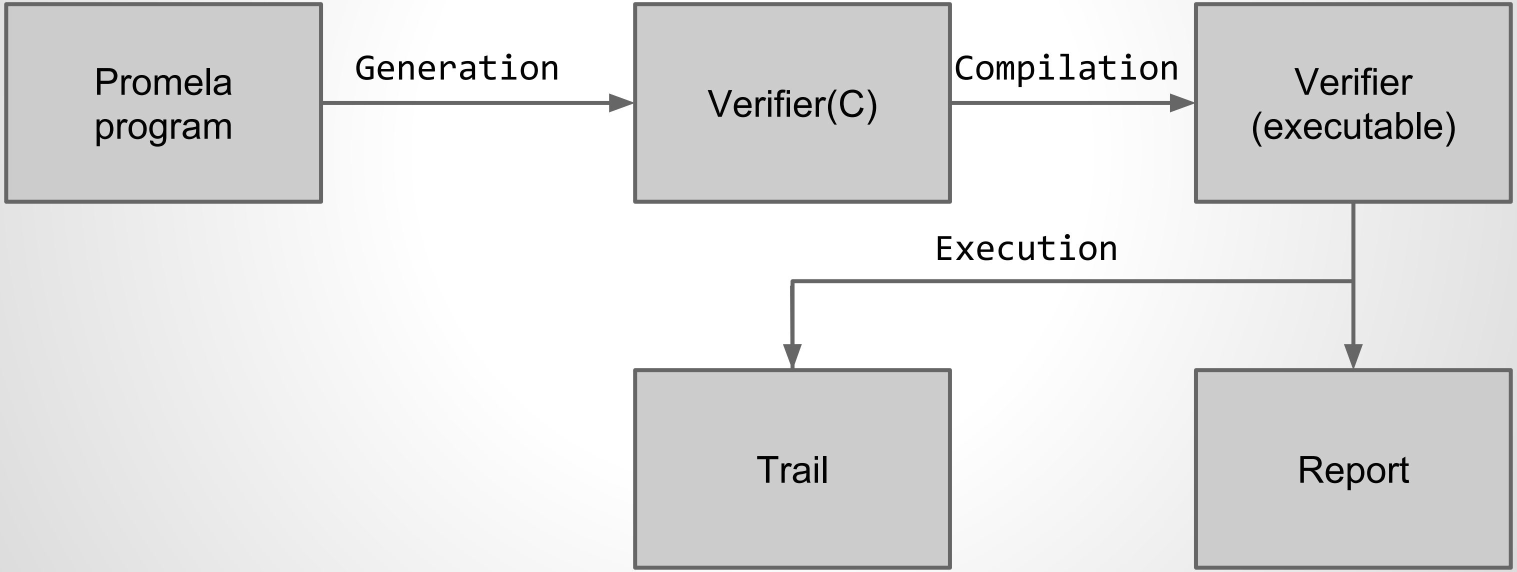
```
active proctype Hello(){
   printf("Hello process, my pid is %d\n", _pid);
}

init{
   int lastpid;
   printf("init process, my pid is: %d\n", _pid);
   lastpid = run Hello();
   printf("last pid was: %d\n", lastpid);
}

$ spin hello.pml
     init process, my pid is: 1
     last pid was: 2
  Hello process, my pid is 0
          Hello process, my pid is 2
3 processes created
```

Architecture of SPIN

Verification in SPIN is a three-step process as shown in this figure:



PROMELA Model

A promela model consists of -

- type declarations
- channel declarations
- global variable declarations
- process declarations
- [init process]

Data types in PROMELA

Basic Types	Values	Size(bits)
bit,bool	0,1, False,True	1
byte	0..255	8
short	-32768..32767	16
int	−231 ..231 − 1	32
unsigned	0..2n − 1	<=32

Arrays (indexing starts at 0)

```
byte a[27];
bit flag[4];
```

Typedef (records)

```
typedef Record {
        short f1;
        byte f2;
}

Record rr;
rr.f1 = ..
```

Type Declarations

- #define N 10
  - N is textually substituted whenever encountered.
- mtype
  - Eg - mtype = {red, yellow, green};
  - Internally, the values of the mtype are represented as positive byte values, so there can be at most 255 values of the type.
  - Advantage over #define is that they can be printed using the %e specifier.
- typedef
- constants

Processes

A process type (**proctype**) consists of

- a name
- a list of formal parameters
- local variable declarations
- body

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

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

Statements

- Can be either -
  - executable - ready to be executed
  - blocked - cannot be executed
- An expression which evaluates to a non-zero value is also executable

```
int value = 123;
active proctype P() {
     int reversed;
     reversed = (value % 10) * 100 +
               ((value / 10) % 10) * 10 +
               (value / 100);
     reversed == 1;  /* Will wait until reversed = 1, i.e. indefinitely */
     printf("value = %d, reversed = %d\n", value, reversed)
}
```

Statements

• skip	always executable
• assert(<expr>)	always executable
• expression	executable if not zero
• assignment	always executable
• if	executable if at least one guard is executable
• do	executable if at least one guard is executable
• break	always executable

Introducing atomicity

- atomic { stat1; stat2; ... statn }
  - used to group statements into an atomic sequence
  - no interleaving with statements of other processes
  - executable if stat1 is executable
  - if a statement is blocked, the “atomicity token” is (temporarily) lost and other processes may do a step
- d_step { stat1; stat2; ... statn }
  - more efficient version of atomic: no intermediate states are generated and stored
  - only deterministic steps
  - runtime error if stat_i ($i>1$) blocks
- atomic and d_step are used to lower the number of states of the model

Concurrency

```
int flag = 0;
active proctype P(){
   printf("In process P, flag is now %d\n", flag);
   flag == 1; // Will wait until flag is not 1
   printf("In process P, flag is now %d\n", flag);
}
active proctype Q(){
   printf("In process Q\n");
   flag = 1;

}
```

```
At runtime -

      In process P, flag is now 0

            In process Q

      In process P, flag is now 1

2 processes created
```

Mutual exclusion problem

```
if
:: a != 0 ->
     c = b / a
:: else ->
     c = b
fi
```

Assume a is a global variable.

Between the evaluation of the guard a != 0 and the execution of the assignment statement c = b / a, some other process might have assigned zero to a.

Division by zero seems possible!

Another example...

```
bit flag;
byte mutex;

proctype P(bit i) {
   flag != 1;
   flag = 1;
   mutex++;
   printf("MSC: P(%d) has entered section.\n", i);
   mutex--;
   flag = 0;
}
proctype monitor() {
 assert(mutex != 2);
}

init {
   atomic { run P(0); run P(1); run monitor(); }
}
```

```
models:
while (flag == 1) /* wait */;
```

Problem: assertion violation!

Both processes can pass the flag != 1 “at the same time”, i.e. before flag is set to 1.

Solution?

```
proctype P(bit i) {
   atomic{
       flag != 1;
       flag = 1;
   }
   mutex++;
   printf("MSC: P(%d) has entered section.\n", i);
   mutex--;
   flag = 0;

}
```

Checking for flag!=1 and assignment of flag = 1 happens atomically.

So, two processes cannot enter the critical section at the same time.

Deadlock

```
bit x, y;
byte mutex;

active proctype A() {
    x = 1;
    y == 0;
    mutex++;
    mutex--;
    x = 0;
}

active proctype monitor() {
    assert(mutex != 2);
}
```

```
active proctype B() {
    y = 1;
    x == 0;
    mutex++;
    mutex--;
    y = 0;
}
```

Process A waiting for Process B to end

Problem: invalid-end-state!
Both processes can pass execute
x = 1 and y = 1 “at the same time”,
and will then be waiting for each other.

Solution?

```
bit x, y;
byte mutex;

active proctype A() {
    atomic{
        y == 0;
        x = 1;
    }
    mutex++;
    mutex--;
    x = 0;
}

active proctype B() {
    atomic{
        x == 0;
        y = 1;
    }
    mutex++;
    mutex--;
    y = 0;
}

active proctype monitor() {
    assert(mutex != 2);
}
```

Without loss of generality, assume that A is executed first.
y==0 is checked and x=1 is set in the same atomic instruction.

Now, B has to wait until x becomes 0 again, which happens at the end of process A.

References

- Principles of the spin model checker by Holtzmann
- http://spinroot.com/spin/Doc/SpinTutorial.pdf