

Introduction to the Model Checker Spin

Wim H. Hesselink, October 6, 2008

1 Model Checking with Spin

Model checking is a verification technique that yields results much more quickly than theorem proving. It is based on the idea of exhaustive exploration of the reachable state space of a system. Therefore, currently, it can only be applied to systems with a finite state space, with not too many states. In practice, this is a severe limitation. There are experiments with symbolic model checkers that should be able to deal with infinite state spaces, but that is for the future.

The model checker Spin [2] has two principle modes of operation: simulation and verification. Verification indeed requires exhaustive search, whereas simulation does not and therefore can deal with bigger state spaces. Just as all other forms of testing, *simulation* can only indicate errors and can never show the absence of errors. It is definitely useful, but we shall concentrate on *verification*.

In Spin, verification is subdivided into two aspects: *safety* (nothing bad happens) and *liveness* (eventually something good happens). Just as with theorem proving, verification of safety is usually done first, it is more critical, and it is easier.

A model to be checked by Spin is declared by means of a script in the Protocol Metalanguage *Promela* [1, 2]. Such a script defines a number of processes or threads, that run concurrently (by interleaving semantics) and that need not terminate! So, the first source of nondeterminism is the choice of an arbitrary interleaving of the processes. The processes are defined with the keyword **proctype**.

The main primitive steps of processes are assignments and guards. A *guard* B is a Boolean expression that is used as a statement with the meaning: “wait until B holds”. A sequential composition $B;S$ can be written as $B \rightarrow S$. This is preferably done (only) when B is a guard. Then we call it a *guarded command*. It is said to be *enabled* if B holds.

Steps can be combined by means of sequential composition and the constructs **if fi** for selection and **do od** for repetition. These two constructs are nondeterministic. They can have any positive number of branches, each preceded by **::**.

```
if                                do
  :: S1 ;                          :: S1 ;
  :: S2 ;                          :: S2 ;
fi                                  do
```

Both constructs can have one or more alternatives, which are (possibly guarded) commands. Their meaning is a single or repeated nondeterministic choice between the enabled alternatives. If none of the alternatives is enabled, the construct blocks until some other process might modify a guard. Execution can leave the **do..od** repetition only via **break** or **goto**. For example: one can give **i** an arbitrary value in the range 0..5 by

```
i = 0 ;
do
  :: i < 5 -> i++ ;
  :: break ;
od
```

2 An Example Verification: Peterson

We may have seen Peterson's mutual exclusion algorithm for two processes in PVS. There are two processes and the algorithm must ensure that they are never at the same time in the critical section CS. We construct a Spin model of it to verify this property.

```
    bit aktief[2] ; bit last ; // initially (0, 0) and 0

    proctype threadMX (bit self) {
11      do
          :: break
          :: aktief[self] = 1 ;
12         last = self ;
13         (last == 1-self || ! aktief[1-self]) ; // AWAIT
14         // CS
          aktief[self] = 0 ;
        od
15    }

    init {
1      run threadMX (0) ;
2      run threadMX (1) ;
3    }
```

Apart from the line numbers, this is correct Promela and it corresponds to Peterson's algorithm as discussed earlier. It has the shared variable `last` of type `bit` ($= \{0, 1\}$) and `aktief`, an array of bits. We use the identifier `aktief` since `active` is a Promela keyword. Process `init` spawns two processes `threadMX` with parameters 0 and 1.

We now want to express the safety property that the two processes are never simultaneously in the critical section CS. The easiest way to do this, is to introduce a third shared variable `crit` of type `bit`, to indicate that CS is occupied. Therefore, `crit` is set to 1 when a process enters CS and set to 0 when it leaves CS. When a process enters CS, it first verifies that `crit = 0` by means of `assert`. Therefore the line CS is replaced by

```
    assert crit == 0 ;
    crit = 1 ;
    crit = 0 ;
```

How to verify this with Spin? We first make a file with this program in our favorite editor. We then open Spin by means of the shell command "`xspin &`". We open our file via the file menu. We then consult the *RUN* menu. Always first run syntax check to avoid unaccountable errors! Then choose "set verification parameters" in the runmenu: here, we choose safety with checking for asserts and invalid end-states. Then choose "run verification" and we get after some seconds output in a new window.

In order to appreciate this output, let us change the program in such a way that it is incorrect. First completely remove the await statement in line 13. Running the verification now yields an assert violation and gives you the option to run a guided simulation that violates the assert statement. Try it.

Now replace the await statement by either disjunct, e.g. by `(last == 1-self)`. Now verification should result in an invalid end-state: one of the processes hangs at the await statement and the other one has terminated. You could legitimize the end-state by putting a label "`end:`" before the await statement.

To better understand the model, we come back to the Promela script without the things about `crit`. The `init` process has three locations 1, 2, 3. The processes `threadMX` have five locations: 11, 12, 13, 14, 15. At 11, the process can choose to execute either branch of the `do` loop. In the branch with `break` it goes to 15. In the other branch, it sets `aktief` and goes to 12. At 13, the process can only make a step when the boolean holds true. Such a command is called a *guard*. At 14, the process resets `aktief` and goes to 11.

2.1 Grain of Atomicity

The executions of a Promela model are finite or infinite sequences of atomic steps. Each atomic step can be performed by any of the processes. An execution is thus an interleaving of atomic steps of the different processes. This gives Spin very many different executions to consider. Indeed, in verification Spin considers (in some sense) all possible executions!

What is atomic? Unless specified otherwise, every single assignment or guard is treated as atomic. Often, we want to investigate systems with bigger atomic commands. These can be created in two ways:

```
atomic {statements}
d_step{statements} // only if statements is deterministic
```

See [1] for details and differences.

In Peterson's algorithm, it is useful to shorten the critical section to

```
atomic { assert crit == 0 ; crit = 1 ; }
crit = 0 ;
```

What would happen, however, when you use:

```
atomic { assert crit == 0 ; crit = 1 ; crit = 0 ; } ?
```

2.2 Bounded Overtaking

Let us now use Spin to answer a more difficult question. When a process is waiting at 13, how many times can the other process enter CS and leave again? For this purpose we introduce an array `wtime` of bytes, with the intention that `wtime[p] - 1` is the number of times the process $\neq p$ slips through. We use `wtime[p] = 0` when process `p` is not at 13.

```
byte wtime[2] ;
#define tryout 2

proctype threadMX(bit self) {
11     do
        :: break
        :: aktief[self] = 1 ;
12     atomic { last = self ; wtime[self] = 1 ; }
13     (last == 1-self || ! aktief[1-self]) ; // AWAIT
14     atomic {
        wtime[1-self] =
            (wtime[1-self] > 0 -> wtime[1-self]+1 : 0) ;
        assert wtime[1-self] < tryout ;
        wtime[self] = 0 ;
        aktief[self] = 0 ; }
        od
15 } }
```

In the first assignment of 14, a conditional expression is used. Note that Promela's conditional expression uses an arrow instead of a question mark, because the question mark is reserved in Promela for reception of messages.

The variable `wtime` is only a ghost variable, used by the verifier to see what happens in the system, but not a program variable of the system itself. We are therefore free to add its modifications atomically to the actions on other variables. This has the advantage that it does not unnecessarily increase the number of states the model checker has to reckon with. For the presentation, it has the advantage that there is no need to renumber the locations. Checking yields an assert violation if and only if `tryout < 3`.

Remark. Always force Spin first to *complain* about errors, so that you see that it is indeed testing for them! Otherwise, Spin can easily fool you by not testing for the thing you are interested in!

2.3 Mutual Exclusion: a Channel as Semaphore

```
#define Nproc 5
chan sema = [1] of {bit} ; // buffered channel; contents are irrelevant
byte crit, proc ;

proctype member (byte self) {
  do
    :: proc -- ; break // invariant: proc = # active processes
    :: sema ? _ ;
      atomic{ assert crit == 0 ; crit ++ ; }
      atomic{ crit -- ; sema ! 0 ; }
  od
}

init {
  atomic {
    do
      :: proc < Nproc -> run member(proc) ; proc ++ ;
      :: else -> break
    od ;
  }
  assert proc == Nproc ; // is violated, therefore remove
  assert crit == 0 ; sema ! 0 ;
  proc == 0 ; sema ? 1 // gives invalid end-state because it waits for 1
}
```

No invalid end-state is found when the last statement is removed.

3 Liveness

A *liveness* property is one that asserts something good happens eventually or often enough. The simplest liveness property is termination, but there are many other kinds of liveness properties. Before treating these, we need to be more careful about the executions we are considering.

An *execution* is a finite or infinite sequence of states, starting in an initial state, in which consecutive states are valid steps of the model.

An execution is called *minimally fair* if it is either infinite, or finite and in the last state none of the processes is enabled.

An execution is called *weakly fair* iff every process acts infinitely often or is disabled infinitely often.

An execution is called *strongly fair* iff every process acts infinitely often or is continuously disabled from some point onward.

A system is called *minimally fair*, *weakly fair*, *strongly fair* if all its executions are *minimally fair*, *weakly fair*, *strongly fair*, respectively.

Spin is minimally fair, but it can also be made weakly fair. For many practical computer systems one generally assumes that they are weakly fair, for some physical or probabilistic reason.

In many verifications, one proves that a system satisfies its specification under the assumption that it is, say, weakly fair.

In finite-state model-checkers like Spin, infinite executions are represented by cycles, i.e., infinite executions (sequences of states) xs such that, for some offset n and period $p > 0$, we have $xs(p+i) = xs(i)$ for all $i \geq n$. (This does not mean that all infinite executions are cycles, but only that, for all properties that can be tested by the checker, it suffices to test the cycles.)

3.1 Testing for Liveness without the LTL manager

Back to Spin. Every Promela command can be preceded by a label (string) followed by a colon. A label that begins with **accept** or **progress** is called an *acceptance* or *progress* label, respectively. A state is called an *acceptance* or *progress* state iff at least one of the processes is at an acceptance or progress label, respectively.

An *acceptance cycle* is a cycle that infinitely often visits an acceptance state. A *non-progress cycle* is a cycle that does not visit a progress state infinitely often.

When testing for acceptance cycles or non-progress cycles, Spin reports an error when it finds an acceptance cycle or a non-progress cycle, respectively.

3.2 Termination

Acceptance labels can be used to show termination.

```
#define threshold 3
byte x ; bit stop ;

active proctype stepper () {
    do
        :: ! stop -> accept0: x = (x < 7 -> x + 1 : 3) ;
        :: stop    -> break ;
    od
}

active proctype stopper () { x >= threshold -> stop = 1 ; }
```

The keyword **active** means that these processes are started immediately. We therefore do not need the standard process **init** to start them (**init** is active by default). We now want to know whether the processes do terminate. This is a liveness property. Let us first look at the code to see what we can expect. The stepper performs a repetition, which stops when the stopper sets **stop** = 1. The stopper first waits until $x \geq \text{threshold}$ and then sets **stop** = 1. This suggests that the system terminates.

We verify it by testing for “acceptance cycles”, by placing the accept label **accept0** in the repeated alternative of the repetition. The stepper does not terminate if and only if there is an acceptance cycle.

When we test this, Spin reports an error: an acceptance cycle. When we ask Spin for a guided simulation, it shows such a cycle, in which the stopper never executes command 22. This execution is not fair. More precisely, it is not *weakly fair*. In this cycle, the stepper acts infinitely often, but the stopper acts at most once, while it is always enabled after the first 3 updates of `x`. It is therefore not disabled infinitely often.

We can also ask Spin to test for acceptance cycles *with weak fairness*. Then it reports no errors, i.e., no acceptance cycles. Indeed, under weak fairness, both processes terminate and are then disabled infinitely often. This remains valid if we choose `threshold = 2`.

If we choose `threshold = 4`, however, the acceptance cycle reappears! The reason is that the stopper can be infinitely often disabled at 21. Note that the stopper is not continuously disabled for some point onward. The system would therefore terminate under strong fairness, but this cannot easily be tested with Spin.

NB. The word “accept” does not refer to an accepting state of a finite state machine. The name “reject” might have been better.

An even simpler case where Spin without weak fairness finds an acceptance cycle:

```
bit x ;
active proctype stepper () { do :: x = ! x od }
active proctype acceptor () { x = ! x ; accept: skip }
```

The cycle occurs when `stepper` loops while `acceptor` is at its label. Weak fairness, however, forces `acceptor` to eventually leave its label (and hence terminate). So, in that case, there is no acceptance cycle.

3.3 Progress

Termination is the absence of infinite behaviour. Testing for acceptance cycles is generally testing for the absence of unwanted infinite behaviour. Progress however is the guarantee of wanted infinite behaviour. This is done by testing for non-progress cycles with a progress label at the command we want to ensure.

In the following example, progress means that `x` is modified infinitely often (in the previous version of this note, this example was treated incorrectly).

```
byte x, y, z ;
active proctype stepper () {
  do :: x == y -> progress0: x = (x+2) % 5 ; od
}
active proctype equalizer () { do :: y = x od }
```

When we test progress, the verifier reports an error, noticing a non-progress cycle where the stepper remains at the guard and the equalizer performs its steps (which do not change the state).

Under weak fairness, the verifier confirms correctness, because `equalizer` will always eventually enable `stepper` by `y = x`.

References

- [1] R. Gerth. Concise Promela reference. [//spinroot.com/spin/Man/Quick.html](http://spinroot.com/spin/Man/Quick.html), 1997.
- [2] G.J. Holzmann. *The SPIN Model Checker, primer and reference manual*. Addison-Wesley, 2004.