Further Steps with SPIN and PROMELA

David Kendall

1 Objectives

- Make further progress with SPIN and PROMELA in modelling and analysing some simple concurrent systems
- Use PROMELA and SPIN to model and analyse simple mutex protocols with respect to safety properties

2 Checking Safety Properties of Mutex Using SPIN

Use Promela/Spin to model and verify the algorithms below for maintaining mutually exclusive access to a critical region in a 2 process system.

You should check for the following properties:

- Mutual exclusion is preserved. i.e. at most one process is in its critical section at any one time.
- Deadlock is avoided.

You can assume that no process will remain in its critical section forever. You can not assume that a process will ever try to enter its critical section.

Verification of these properties can be carried out with Spin by using the following techniques:

Assertions – for checking for preservation of mutual exclusion. You will need to think of an assertion (or assertions) which you can include in your program, whose truth guarantees that mutual exclusion is preserved. Use Spin to check that your assertions are not violated.

Invalid end-states – by default Spin will check for invalid end states (i.e. states from which it is not possible to make further progress, which either a) do not occur as the result of termination, or b) are not labelled as valid end states (by using an "end:" label in Promela)). The existence of an invalid end-state indicates the possibility of deadlock.

You should refer to the Basic SPIN Manual, if necessary, for an introduction to the use of assertions and end-labels.

The algorithms are described below using pseudo-code.

Algorithm 1 Uses a single shared variable called ‘entry’ which can take one of two values, either ‘allowed’ or ‘forbidden’

Process 1 and 2

```
entry = allowed;
repeat
  while entry = forbidden do
    /* nothing */
  end while;
  entry = forbidden;
  /* do critical section */
  entry = allowed;
  /* do processing outside critical section */
forever
```

Algorithm 2 Uses 2 shared variables ‘need1’ which is set by process 1 when it needs to enter its critical section and ‘need2’ which is set similarly by process 2.

Process 1

```
need1 = false;
repeat
  need1 = true;
  while need2 do
    /* nothing */
  end while;
  /* do critical section */
  need1 = 0;
  /* do processing outside critical section */
forever
```

Process 2

```
need2 = false;
repeat
  need2 = true;
  while need1 do
    /* nothing */
  end while;
  /* do critical section */
  need2 = false;
  /* do processing outside critical section */
forever
```