Theo C. Ruys - SPIN Beginners’ Tutorial version: Friday, 13 September 2002

SPIN 2002 Workshop, Grenoble, 11-13 April ...

atomic

atomic { stat; stat; ... stat }

- can be used to group statements into an atomic sequence;
  all statements are executed in a single step
  (no interleaving with statements of other processes)
  - is executable if stat is executable
  - if a stat, (with i>1) is blocked, the “atomicity token” is
    temporarily lost and other processes may do a step

- (Hardware) solution to the mutual exclusion problem:

P1() { tla; tlb; tlc }
P2() { t2a; t2b; t2c }
init { run P1(); run P2() }

atomic { tla; tlb; tlc }

It is as if P1 has only one transition...

If one of P1's transitions blocks, these transitions may get executed

Although atomic clauses cannot be interleaved, the intermediate
states are still constructed.

It is as if P1 has only one transition...

No intermediate states will be constructed.
timeout

- Promela does not have real-time features.
  - In Promela we can only specify functional behaviour.
  - Most protocols, however, use timers or a timeout mechanism to resend messages or acknowledgements.

- timeout
  - SPIN's timeout becomes executable if there is no other process in the system which is executable
  - so, timeout models a global timeout
  - timeout provides an escape from deadlock states
  - beware of statements that are always executable...

unless

{ <stats> } unless { guard; <stats> }

- Statements in <stats> are executed until the first statement (guard) in the escape sequence becomes executable.
- resembles exception handling in languages like Java
- Example:

  proctype MicroProcessor() {
   ...
   // execute normal instructions */
   } unless { port ? INTERRUPT; ... } }

goto

- transfers execution to label
- each Promela statement might be labelled
- quite useful in modelling communication protocols

Communication

Major models of communication

1. Shared variables
   - one writes, many read later

2. Point-to-Point synchronous message passing
   - one sends, one other receives at the same time
   - send blocks until receive can happen

3. Point-to-Point asynchronous message passing
   - one sends, one other receives some time later
   - send never blocks

4. Point-to-Point buffered message passing
   - When buffer not full behaves like asynchronous
   - When buffer full, two variations: block or drop message

5. Synchronous broadcast
   - one sends, many receive synchronously
   - First variation: send never blocks process may receive if ready to ready
   - Second variation: send blocks until all possible recipients ready to receive

Communication in SPIN

- With more or less complexity each can implement the others
- Spin supports 1 and 4 (blocks send when buffer full), but with bounded buffers
- Buffer size = 0 \(\rightarrow\) synchronous communication
- Large buffer size approximates asynchronous communication
**Communication (2)**

- Communication between processes is via channels:
  - message passing
  - rendez-vous synchronisation (handshake)
- Both are defined as channels:
  ```c
  chan <name> = [(dim)] or [<t>, <t>, ..., <t>];
  ```
  also called queue or buffer
- name of the channel
- type of the elements that will be transmitted over the channel
- number of elements in the channel
- dim==0 is special case: rendez-vous

**Communication (3)**

- channel = FIFO-buffer (for dim>0)
  ```c
  ! Sending - putting a message into a channel
  ```
  ```c
  ch ! <expr>, <expr>, ..., <expr>;
  ```
  - The values of <expr> should correspond with the types of the channel declaration.
  - A send-statement is executable if the channel is not full.
  ```c
  ? Receiving - getting a message out of a channel
  ```
  ```c
  ch ? <var>, <var>, ..., <var>;
  ```
  - If the channel is not empty, the message is fetched from the channel and the individual parts of the message are stored into the <var>s.
  - A receive-statement is executable if the channel is not empty and the message at the front of the channel evaluates to the individual <var>s.
- If the channel is empty, the statement is executable and the message is removed from the channel.

**Alternating Bit Protocol (1)**

- Alternating Bit Protocol
  ```c
  – To every message, the sender adds a bit.
  ```
  ```c
  – The receiver acknowledges each message by sending the received bit back.
  ```
  ```c
  – To receive only excepts messages with a bit that it excepted to receive.
  ```
  ```c
  – If the sender is sure that the receiver has correctly received the previous message, it sends a new message and it alternates the accompanying bit.

**Alternating Bit Protocol (2)**

```c
proctype Sender(chan in, out) {
  bit sendbit, recvbit;  
  do  if out ! MSG, sendbit ->
    in ? ACK(recvbit) -> out ! ACK(sendbit);  
    od
  if recvbit == sendbit ->
    sendbit = 1-sendbit;
  else
    fi
  od
}
```

```c
proctype Receiver(chan in, out) {
  bit recvbit;  
  do  if in ? MSG(recvbit) ->
    out ! ACK(sendbit);  
    od
  init
  {  
    run Sender(toS, toS);  
    run Receiver(toR, toS);
  }
```

**mutextwrng1.pml**

```pml
bit flag; /* signal entering/leaving the section */
byte mutex; /* # proc in the critical section. */
proctype P(bit i) {
  flag := 1;
  flag := 1;
  mutex++;  
  printf("MSG: 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(); }
**SPIN as Simulator**

bash-3.2$ spin mutexwrong1.pml
MSC: P(0) has entered section.
MSC: P(1) has entered section.
4 processes created

bash-3.2$ !s
spin mutexwrong1.pml
MSC: P(1) has entered section.
MSC: P(0) has entered section.
4 processes created

**SPIN as Model Checker**

bash-3.2$ spin -a mutexwrong1.pml
bash-3.2$ ls -ltr
total 3520
-rw-r--r-- 1 elsa staff 335 Apr 11 23:27 mutexwrong1.pml
-rw-r--r-- 1 elsa staff 18801 Apr 11 23:28 pan.t
-rw-r--r-- 1 elsa staff 54243 Apr 11 23:28 pan.p
-rw-r--r-- 1 elsa staff 3450 Apr 11 23:28 pan.m
-rw-r--r-- 1 elsa staff 16489 Apr 11 23:28 pan.h
-rw-r--r-- 1 elsa staff 309382 Apr 11 23:28 pan.c
-rw-r--r-- 1 elsa staff 919 Apr 11 23:28 pan.b

bash-3.2$ cc -o pan pan.c
bash-3.2$ ./pan
hint: this search is more efficient if pan.c is compiled -DSAFETY
pan:1: assertion violated (mutex!=2) (at depth 11)
pan: wrote mutexwrong1.pml.trail
(Spin Version 6.2.4 -- 8 March 2013)
Warning: Search not completed
* Partial Order Reduction
Full statespace search for:
never claim - (none specified)
assertion violations +
acceptance cycles - (not selected)
invalid end states +

**SPIN as Model Checker**

State-vector 44 byte, depth reached 20, errors: 1
121 states, stored
47 states, matched
168 transitions (= stored+matched)
2 atomic steps
hash conflicts: 0 (resolved)
Stats on memory usage (in Megabytes):
0.008 equivalent memory usage for states
(stored*(State-vector + overhead))
0.291 actual memory usage for states
128.000 memory used for hash table (~w24)
0.534 memory used for DFS stack (~m10000)
128.730 total actual memory usage

**SPIN as Model Checker**

Spin: elapsed time 0 seconds

**mutextwrong1.pml Error Trace**

bash-3.2$ spin -t -p mutexwrong1.pml
using statement merging
Starting P with pid 1
1: proc 0 (:init:) mutexwrong1.pml:14 (state 1) [(run P(0))]
Starting P with pid 2
2: proc 0 (:init:) mutexwrong1.pml:14 (state 2) [(run P(1))]
Starting monitor with pid 3
3: proc 0 (:init:) mutexwrong1.pml:14 (state 3) [(run monitor())]
4: proc 2 (P) mutexwrong1.pml:4 (state 1) [[[flag!=1]]]
5: proc 1 (P) mutexwrong1.pml:4 (state 1) [[[flag!=1]]]
6: proc 2 (P) mutexwrong1.pml:5 (state 2) [flag = 1]
7: proc 2 (P) mutexwrong1.pml:6 (state 3) [mutex = (mutex+1)]
8: proc 2 (P) mutexwrong1.pml:7 (state 4)
[print("(MSC: P(\%d) has entered section.\",i)])
9: proc 1 (P) mutexwrong1.pml:5 (state 2) [flag = 1]

**mutextwrong1.pml Error Trace**

10: proc 1 (P) mutexwrong1.pml:6 (state 3) [mutex = (mutex+1)]
   MSC: P(0) has entered section.
11: proc 1 (P) mutexwrong1.pml:7 (state 4)
   [print("(MSC: P(\%d) has entered section.\n",i)])
   spin: mutexwrong1.pml:11, Error: assertion violated
   spin: text of failed assertion: assert((mutex!=2))
   12: proc 3 (monitor) mutexwrong1.pml:11 (state 1)
   [assert((mutex!=2))]
   spin: trail ends after 12 steps
#processes: 4
flag = 1
mutex = 2
12: proc 3 (monitor) mutexwrong1.pml:12 (state 2) <valid end
12: proc 2 (P) mutexwrong1.pml:7 (state 5)
12: proc 1 (P) mutexwrong1.pml:7 (state 5)
12: proc 0 (:init:) mutexwrong1.pml:15 (state 5) <valid end
4 processes created
bash-3.2$