Assertion Violation: mutextwrong1.pml

bit flag; /* signal entering/leaving the section */
byte mutex; /* # procs in the critical section. */

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();
    }
}

Examining Error Traces: mutextwrong1.pml

How did mutextwrong1.pml go wrong?

bash-3.2$ cc -o pan pan.c
bash-3.2$ /pan
pan:1: assertion violated (mutex!="2") (at depth 11)
pan: wrote mutextwrong1.pml.trail

(Spin Version 6.4.8 -- 2 March 2018)
Warning: Search not completed
+ Partial Order Reduction

Full statespace search for:
never claim  - (none specified)
assertion violations +
acceptance cycles  - (not selected)
invalid end states +

Simulator options (incomplete):
+p: Print at each state which process took which step
+s: Print send statements and their effects
+r: Print receive statements and their effects
+v: verbose
+nN: Use N as random seed, instead of clock (good for reproducibility)
+1: Show changes to local variables
+g: Show changes to global variables
+uN Limit number of steps taken to N
+kfilename use the trail file stored in filename
Examining Error Traces: mutexwrong1.pml

How did mutexwrong1.pml go wrong?

spin: mutexwrong1.pml:0, warning, proctype P, 'bit i' variable is never used (other than in print stmts) using statement merging
Starting P with pid 1
1: proc 0 (:init::1) mutexwrong1.pml:14 (state 1) [(run P(0))]
Starting P with pid 2
2: proc 0 (:init::1) mutexwrong1.pml:14 (state 2) [(run P(1))]
Starting monitor with pid 3
3: proc 0 (:init::1) mutexwrong1.pml:14 (state 3)
   [(run monitor())]
4: proc 2 (P:1) mutexwrong1.pml:4 (state 1) [((flag!=1))]
5: proc 1 (P:1) mutexwrong1.pml:4 (state 1) [((flag!=1))]
6: proc 2 (P:1) mutexwrong1.pml:5 (state 2) [flag = 1]
flag = 1
7: proc 2 (P:1) mutexwrong1.pml:6 (state 3) [mutex = (mutex+1)]
mutex = 1
MSC: P(1) has entered section.
8: proc 2 (P:1) mutexwrong1.pml:7 (state 4) [printf('MSC: P(%d) has entered section.\n',i)]
9: proc 1 (P:1) mutexwrong1.pml:5 (state 2) [flag = 1]
10: proc 1 (P:1) mutexwrong1.pml:6 (state 3) [mutex = (mutex+1)]
mutex = 2
MSC: P(0) has entered section.
11: proc 1 (P:1) mutexwrong1.pml:7 (state 4) [printf('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:1) mutexwrong1.pml:11 (state 1)
   [assert((mutex!=2))]

spin: trail ends after 12 steps
#processes: 4
flag = 1
mutex = 2
12: proc 3 (monitor:1) mutexwrong1.pml:12 (state 2) <valid end state>
12: proc 2 (P:1) mutexwrong1.pml:15 (state 5) <valid end state>
4 processes created

Examining Error Traces: mutexwrong1.pml

Deadlock: mutexwrong2.pml

bit x, y; /* signal entering/leaving the section */
byte mutex; /* # of procs in the critical section. */
active proctype A() {
    x = 1;
    y == 0;
    mutex++;
    printf("Process A is in the critical section\n");
    mutex--;
    y = 0;
}
active proctype B() {
    y = 1;
    x == 0;
    mutex++;
    printf("Process B is in the critical section\n");
    mutex--;
    y = 0;
}
active proctype monitor() {
    assert(mutex != 2);
}

SPIN as Simulator

bash-3.2$ spin mutexwrong2.pml
Process A is in the critical section
Process B is in the critical section
3 processes created
bash-3.2$ spin mutexwrong2.pml
timeout
#processes: 2
x = 1
y = 1
mutex = 0
3: proc 1 (B:1) mutexwrong2.pml:15 (state 2)
5: proc 0 (A:1) mutexwrong2.pml:6 (state 2)
3 processes created
Deadlock Detection in SPIN

bash-3.2$ spin -a mutexwrong2.pml
bash-3.2$ cc -o pan pan.c
bash-3.2$ ./pan
pan:1: invalid end state (at depth 3)

(Spin Version 6.4.8 -- 2 March 2018)
Warning: Search not completed
* Partial Order Reduction

Full statespace search for:
never claim = (none specified)
assertion violations +
acceptance cycles = (not selected)
invalid end states +

Examining Error Traces: mutexwrong2.pml

atomic

atomic { stat_1; stat_2; ... stat_n }
  - 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_1 is executable / no pure atomicity
  - if a stat_i (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:

No atomicity

Not completely correct as each process has an implicit end-transition...
Checking for pure atomicity

- Suppose we want to check that none of the atomic clauses in our model are ever blocked (i.e. pure atomicity).

1. Add a global bit variable:
   ```
   bit aflag;
   ```

2. Change all atomic clauses to:
   ```
   atomic {
     stat,
     aflag=1;
     ...
     stat,
     aflag=0;
   }
   ```

3. Check that aflag is always 0.

   e.g. active process monitor {
     assert(!aflag);
   }

e.g. to verify that all atomic clauses are being executed.


goto label

- Transfers execution to label
- Each Promela statement may be labelled
- Quite useful in modelling communication protocols

```text
wait_ack: if :?ACK -> ab1=ab ; goto success :: ChunkTimeout?SHAKE -> if : (rc < MAX) -> rc++; !'i=1, (i<=n), ab,d[i]; goto wait_ack :: (rc >= MAX) -> goto error fi fi ;
```

Port of model of BRP

**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 for deadlock states
- Beware of statements that are always executable...

**unless**

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

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

Major models of communication

- **Shared variables**
  - one writes, many read later
- **Point-to-Point** synchronous message passing
  - one sends, one other receives at the same time
  - send blocks until receive can happen
- **Point-to-Point** asynchronous message passing
  - one sends, one other receives some time later
  - send never blocks
- **Point-to-Point** buffered message passing
  - When buffer not full behaves like asynchronous
  - When buffer full, two variations: block or drop message
  - send never blocks
- **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 (1)**

![Diagram of communication between Sender and Receiver](image)

**Communication (2)**

- Communication between processes is via channels:
  - message passing
  - rendez-vous synchronisation
- Both are defined as channels:
  - `chan <name> = [ <dim> ] of { <t1>, <t2>, … <tn> };`
  - `chan c = [1] of { bit };`
  - `chan tok = [2] of { mtype, bit };`
  - `chan line[2] = [1] of { mtype, Record };`
- `dim == 0` is special case: rendez-vous handshake

**Communication (3)**

- `channel = FIFO-buffer (for dim > 0)`
- **Sending** - putting a message into a channel
  - `ch! <expr1>, <expr2>, … <exprn>;`
  - The values of `<expr1>`, `<expr2>`, … `<exprn>` should correspond with the types of the channel declaration.
  - A send-statement is executable if the channel is not full.
- **Receiving** - getting a message out of a channel
  - `ch? <var1>, <var2>, … <varn>;`
  - If the channel is not empty, the message fetched from the channel and the individual parts of the message are stored into the `<var1>`, `<var2>`, … `<varn>`.

**Communication (4)**

- **Rendez-vous communication**
  - `<dim> == 0`
  - The number of elements in the channel is now zero.
  - If `ch!` is enabled and if there is a corresponding `ch?` that can be executed simultaneously and the constants match, then both statements are enabled.
  - Both statements will “handshake” and together take the transition.
- **Example:**
  - `chan ch = [0] of { bit, byte };`
  - P wants to do `ch! 1, 3+7`
  - Q wants to do `ch? 1, x`
  - Then after the communication, `x` will have the value 10. 
• **Alternating Bit Protocol**
  
  – To every message, the *sender* adds a *bit*.
  
  – The *receiver* acknowledges each message by sending the *received* bit back.
  
  – To *receiver* only *excepts* messages with a bit that it *excepted* to receive.
  
  – 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.

```vhdl
mtype {MSG, ACK};
chan toS = [2] of {mtype, bit};
chan toR = [2] of {mtype, bit};
proctype Sender(chan in, out) {
  bit sendbit, recvbit;
  do :: out ! MSG, sendbit ->
    in ? ACK, recvbit;
    if :: recvbit == sendbit ->
      sendbit = 1-sendbit
    :: else
      od
  fi
}
proctype Receiver(chan in, out) {
  bit recvbit;
  do :: in ? MSG(recvbit) ->
    out ! ACK(recvbit);
    od
} init {
  run Sender(toS, toR);
  run Receiver(toR, toS);
}
```

**Alternative notation:**

- `ch ! MSG(par1, ...)`
- `ch ? MSG(par1, ...)`

---

**Cookie:** "hippies" problem

Germany

Holland

\[
\begin{align*}
\leq & 2 \text{ pers} \\
\leq & 60 \text{ min?} \\
\text{holes} \quad \text{[Ruys & Brinksma 1998]} \\
5 & \ldots \\
10 & \ldots \\
20 & \ldots \\
25 & \ldots \\
\text{coffee shop}
\end{align*}
\]