CS477 Formal Software Development Methods

Elsa L Gunter
2112 SC, UIUC
egunter@illinois.edu
http://courses.engr.illinois.edu/cs477

Slides mostly a reproduction of Theo C. Ruys – SPIN Beginners' Tutorial

April 12, 2013

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(); }
}
```

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.pm
-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
```

SPIN as Model Checker

```
bash-3.2$ cc -o pan pan.c
bash-3.2$ ./pan
hint: this search is more efficient if pan.c is compiled -DSAF
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 +

State-vector 44 byte, depth reached 20, errors: 1
```

mutextwrong2.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 criical section\n");
  mutex--;
  x = 0;
}
```

mutextwrong2.pml

```
active proctype B() {
  y = 1;
  x == 0;
  mutex++;
  printf ("Process B is in the criical 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 criical section
          Process B is in the criical section
3 processes created
bash-3.2$ spin mutexwrong2.pml
      timeout
#processes: 2
x = 1
y = 1
mutex = 0
  3: proc  1 (B) mutexwrong2.pml:15 (state 2)
  3: proc  0 (A) mutexwrong2.pml:6 (state 2)
3 processes created
```

SPIN as Simulator

```
bash-3.2$ spin -a mutexwrong2.pml
bash-3.2$ cc -o pan pan.c
bash-3.2$ ./pan
hint: this search is more efficient if pan.c is compiled -DSAF
pan:1: invalid end state (at depth 3)
pan: wrote mutexwrong2.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 +
```

Communication

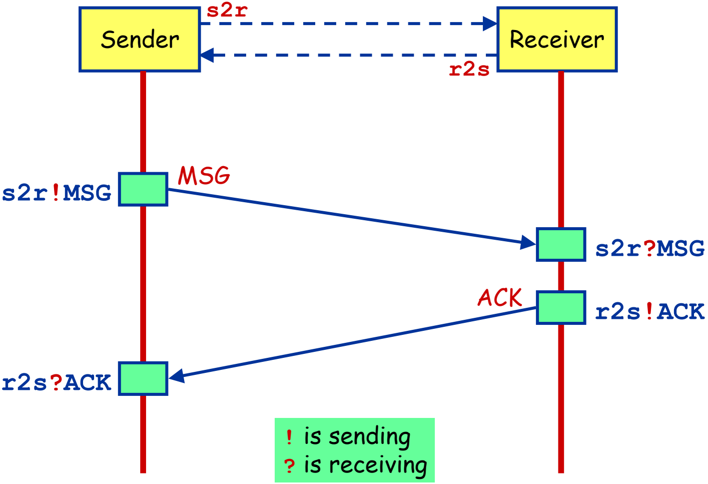
Major models of communication

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 receieve 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
   - send never blocks
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 $= 0 \Longrightarrow$ synchronous communication
- Large buffer size approximates asynchronous communication

Communication (1)



Communication (2)

- Communication between processes is via channels:
  - message passing
  - rendez-vous synchronisation (handshake)
- Both are defined as channels:

also called: queue or buffer

```
chan <name> = [<dim>] of {<t1>,<t2>, … <tn>};
```

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

```
chan c       = [1] of {bit};
chan toR     = [2] of {mtype, bit};
chan line[2] = [1] of {mtype, Record};
```

array of channels

Communication (3)

- channel = FIFO-buffer (for `dim>0`)

! **Sending** - *putting a message into a channel*

```
ch ! <expr1>, <expr2>, … <exprn>;
```

- The values of <$expr_i$> 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*

<var> + <const> can be mixed

```
ch ? <var1>, <var2>, … <varn>;
```

message passing

- 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_i>s.

```
ch ? <const1>, <const2>, … <constn>;
```

message testing

- If the channel is not empty and the message at the front of the channel evaluates to the individual <$const_i$>, the statement is executable and the message is removed from the channel.

Communication (4)

- Rendez-vous communication

  `<dim>` == 0

  The number of elements in the channel is now zero.

  – If send `ch!` is enabled and if there is a corresponding receive `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`.

DEMO

Alternating Bit Protocol (1)

- 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.

DEMO

Alternating Bit Protocol (2)

channel length of 2

```
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
       fi
  od
}
```

```
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, …)`

atomic

`atomic { stat1; stat2; ... statn }`

– 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:

```
proctype P(bit i) {
  atomic {flag != 1; flag = 1; }
  mutex++;
  mutex--;
  flag  = 0;
}
```

d_step

`d_step { stat`$_1$`; stat`$_2$`; ... stat`$_n$` }`

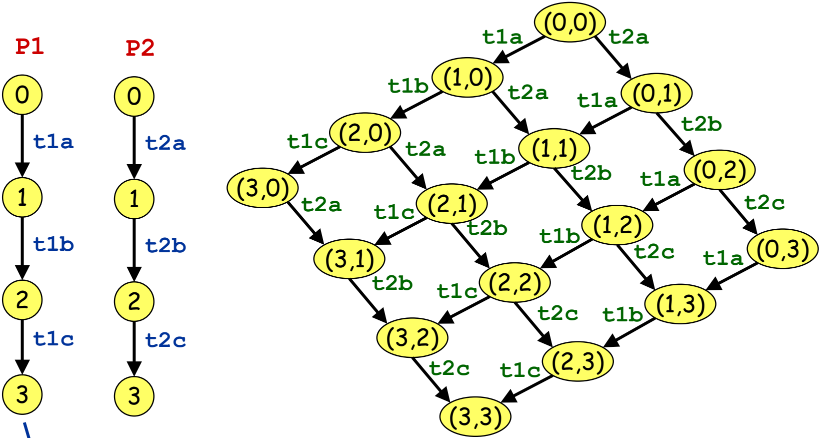
- more efficient version of **atomic**: no intermediate states are generated and stored
- may only contain deterministic steps
- it is a run-time error if **stat**$_i$ **(i>1)** blocks.
- **d_step** is especially useful to perform intermediate computations in a single transition

```
::  Rout?i(v) -> d_step {
       k++;
       e[k].ind = i;
       e[k].val = v;
       i=0; v=0 ;
    }
```

- **atomic** and **d_step** can be used to lower the number of states of the model

```
proctype P1() { t1a; t1b; t1c }
proctype P2() { t2a; t2b; t2c }
init { run P1(); run P2() }
```

No atomicity

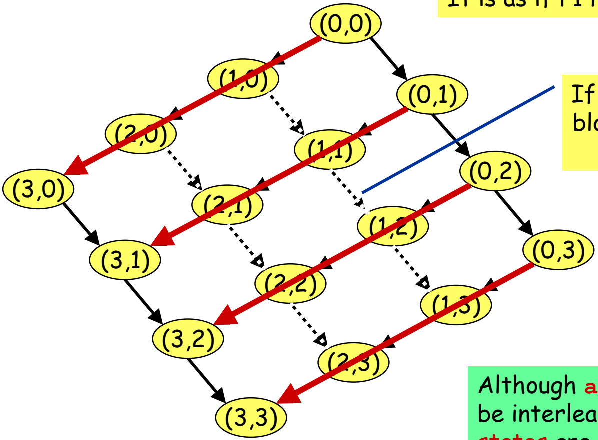


Not completely correct as each process has an implicit end-transition...

```
proctype P1() { atomic {t1a; t1b; t1c} }
proctype P2() { t2a; t2b; t2c }
init { run P1(); run P2() }
```

atomic

It is as if P1 has only one transition…



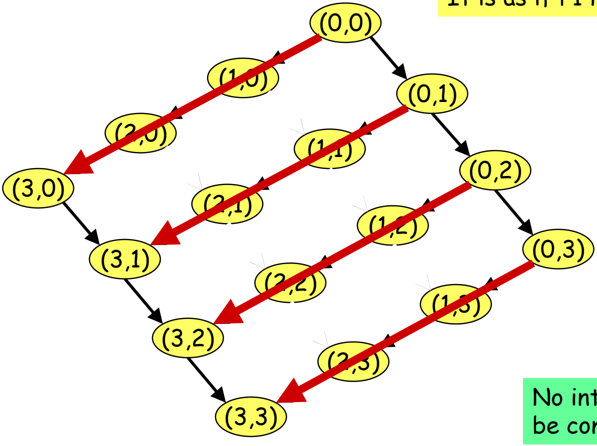
If one of P1's transitions blocks, these transitions may get executed

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

```
proctype P1() { d_step {t1a; t1b; t1c} }
proctype P2() { t2a; t2b; t2c }
init { run P1(); run P2() }
```

d_step

It is as if P1 has only one transition…



No intermediate states will be constructed.

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_1;
   aflag=1;
   stat_2
   
   ...
   
   stat_n
   aflag=0;
}
```

3. Check that `aflag` is always 0.

```
[]!aflag
```

e.g.

```
active process monitor {
   assert(!aflag);
}
```

timeout (1)

- 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…

timeout (1)

- 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…

goto

goto label

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

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

Timeout modelled by a channel.

Part of model of BRP

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; ... }
}
```

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; ... }
}
```

inline - poor man's procedures

- Promela also has its own macro-expansion feature using the **inline**-construct.

```
inline init_array(a) {
  d_step {
    i=0;
    do
    :: i<N  -> a[i] = 0; i++
    :: else -> break
    od;
    i=0;
  }
}
```

Should be declared somewhere else (probably as a local variable). *(refers to the first `i=0;`)*

Be sure to reset temporary variables. *(refers to the final `i=0;`)*

  - error messages are more useful than when using **#define**
  - cannot be used as expression
  - all variables should be declared somewhere else