

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 25, 2014

atomic

atomic { stat₁; stat₂; ... 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 **stat₁** 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 { stat₁; stat₂; ... 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**

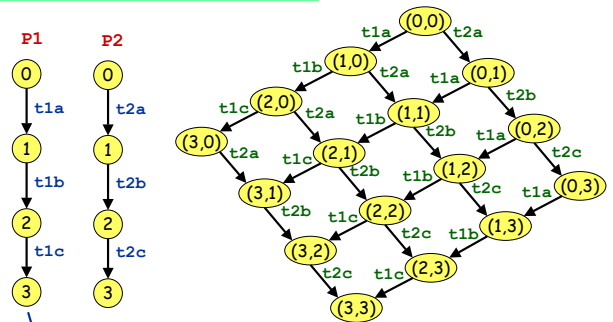
```
:: Rout?i(v) -> d_step {
  k++;
  e[k].ind = i;
  e[k].val = v;
  i=0; v=0;
}
```

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



No atomicity

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

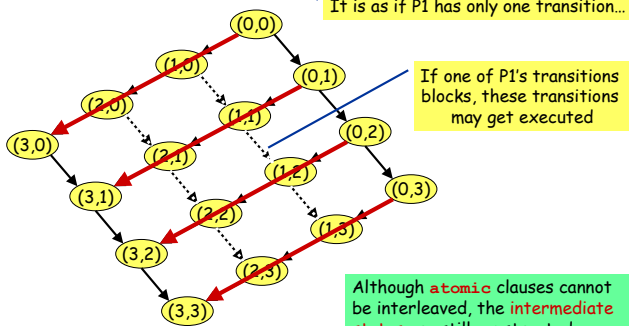


Not completely correct as each process has an implicit end-transition...



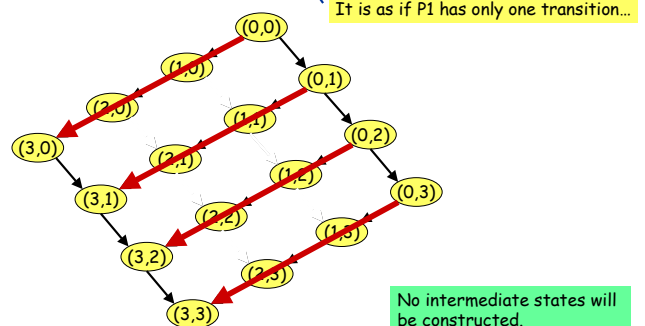
```
proctype P1() { atomic { t1a; t1b; t1c } }
proctype P2() { t2a; t2b; t2c }
init { run P1(); run P2() }
```

atomic



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

d_step



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...



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial

53



Elsa L. Gunter

CS477 Formal Software Development Method

/ 29

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];
    :: (rc >= MAX) -> goto error
    fi
fi ;
```

Timeout modelled by a channel.

Part of model of BRP



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial

56



Elsa L. Gunter

CS477 Formal Software Development Method

/ 29

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; ... }
}
```



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial

57



Elsa L. Gunter

CS477 Formal Software Development Method

/ 29

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

Navigation icons

Elsa L. Gunter

CS477 Formal Software Development Method

/ 29

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 \implies **synchronous** communication
- Large buffer size approximates **asynchronous** communication

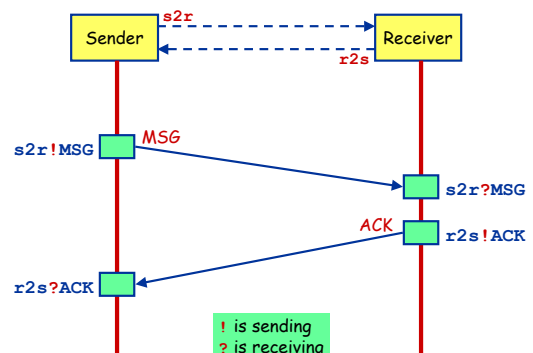
Navigation icons

Elsa L. Gunter

CS477 Formal Software Development Method

/ 29

Communication (1)



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial

36



Elsa L. Gunter

CS477 Formal Software Development Method

/ 29

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



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial

37



Elsa L. Gunter

CS477 Formal Software Development Method

/ 29

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>;
```

- 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.
- 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.



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial

38



Elsa L. Gunter

CS477 Formal Software Development Method

/ 29

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**.



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial

39



Elsa L. Gunter

CS477 Formal Software Development Method

/ 29

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 **expected** 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**.



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial

40



Elsa L. Gunter

CS477 Formal Software Development Method

/ 29

DEMO

Alternating Bit Protocol (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);
}
```

channel length of 2

*Alternative notation:
ch ! MSG(par1, ...)
ch ? MSG(par1, ...)*



Thursday 11-Apr-2002

Theo C. Ruys - SPIN Beginners' Tutorial

41



Elsa L. Gunter

CS477 Formal Software Development Method

/ 29

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(); }
}
```

Elsa L. Gunter

CS477 Formal Software Development Method

/ 29

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
```

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 -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
```

mutexwrong1.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 monit
  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)
          MSC: P(1) has entered section.
  8: proc 2 (P) mutexwrong1.pml:7 (state 4)
[printf('MSC: P(%d) has entered section.\n',i)]
  9: proc 1 (P) mutexwrong1.pml:5 (state 2) [flag = 1]
```

mutexwrong1.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)
[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) mutexwrong1.pml:11 (state 1)
[assert((mutex!=2))]
spin: trail ends after 12 steps
```

mutextwrong1.pml Error Trace

```
#processes: 4
flag = 1
mutex = 2
12: proc 3 (monitor) mutextwrong1.pml:12 (state 2) <valid end
12: proc 2 (P) mutextwrong1.pml:7 (state 5)
12: proc 1 (P) mutextwrong1.pml:7 (state 5)
12: proc 0 (:init:) mutextwrong1.pml:15 (state 5) <valid end
4 processes created
bash-3.2$
```