

Spin: introduction and examples*

Silvia Tomasi

`silvia.tomasi@disi.unitn.it`

`http://ares.science.unitn.it/~silvia.tomasi/st/Formal_Methods.html`

Course on Formal Methods
Lab Class April 20, 2010

Contents

1	Introduction	1
2	PROMELA examples	3
2.1	Hello world!	3
2.2	Producers/Consumers	4
2.3	Mutual Exclusion	9
3	PROMELA overview	13
3.1	Processes	13
3.2	Data objects	14
3.3	Message Channels	15
3.4	Executability	17
4	Asynchronous Network Problems	18
4.1	Reliable FIFO Communication	18
4.2	Leader Election	20

1 Introduction

The SPIN model checker

- Open-source software tool used for formal verification of distributed software systems.

*These slides are derived from those by Stefano Tonetta (FBK-IRST) and Alberto Griggio (FBK-IRST), used for FM lab 2005-2009

- Developed at Bell Labs.
- In 2002, recognized by the ACM with *Software System Award* (like Unix, TeX, Smalltalk, Postscript, TCP/IP, Tcl/Tk).
- Automated tools convert programs written in Java or in C into SPIN models.
- The specification language is called PROMELA.
- SPIN has a graphical user interface, XSPIN.
- Homepage: <http://spinroot.com/spin/whatispin.html>
- Manual: <http://spinroot.com/spin/Man/index.html>

PROMELA

- PROMELA is suitable to describe concurrent software.
 - dynamic creation of concurrent processes
 - (synchronous/asynchronous) communication via message channels
- Programs written in PROMELA can be executed/simulated.
- *Simulation* shows one execution.
- *Verification* checks every execution.

Basic commands

- To simulate a program:


```
spin system.pml
```
- Interactively:


```
spin -i system.pml
```
- To generate a verifier (`pan.c`):


```
spin -a system.pml
```

2 PROMELA examples

2.1 Hello world!

Hello world!

```
active proctype main()
{
    printf("hello world\n")
}
```

- **active** instantiates one process of the type that follows.
- **proctype** denotes that *main* is a process type.
- *main* identifies the process type, it's not a keyword.
- Note that ';' is missing after **printf**: ';' is a statement separator, not a statement terminator.

Remark

- Each process instance has a unique, positive instantiation number.
- A process-instance remains active until the process' body terminates (if ever).

Hello world! Alternative

```
init {
    printf("hello world\n")
}
```

- **init** is a process that initializes the system.
- Initially just the initial process is executed.

Simulation:

```
> spin hello.pml
    hello world
1 process created
```

2.2 Producers/Consumers

Producers/Consumers

File `prodcons.pml`:

```
mtype = { P, C };

mtype turn = P;

active proctype producer()
{
    do
        :: (turn == P) ->
            printf("Produce\n");
            turn = C
    od
}
...
```

Producers/Consumers

- **mtype** defines symbolic values.
- *turn* is a global variable.
- **do ... od** (i.e. repetition statement) defines a loop.
- Only a **break** statement can break the loop.
- Every option of the loop must start with `::`.
- `(turn == P)` is the guard of the option.
- `->` and `;` are equivalent (`->` indicates a causal relation between successive statements).
- If all guards are false, then the process blocks.
- If multiple guards are true, we get non-determinism.

Producers/Consumers

The producer's definition is equivalent to:

```
active proctype producer()
{
again:  if
        :: (turn == P) ->
            printf("Produce\n");
            turn = C
        fi;
}
```

```

        goto again
    }

```

- goto transfers control to the statement labeled by `again`.

Producers/Consumers

Also equivalent to:

```

active proctype producer()
{
again:  (turn == P) ->
        printf("Produce\n");
        turn = C;
        goto again
}

```

- If the guard does not hold, execution blocks until it does.

Producers/Consumers

Also equivalent to:

```

active proctype producer()
{
again:  if
        :: (turn == P) ->
            printf("Produce\n");
            turn = C
        :: else -> goto again
        fi;
        goto again
}

```

- `else` is only executable (true) if all other options are not executable.

Producers/Consumers

Simulation:

```

> spin prodcons.pml | more
Produce
    Consume
Produce
    Consume
Produce
    Consume
Produce
    Consume
Produce
    Consume
...

```

Producers/Consumers Extended

We can extend the example to more processes for each type:

```
active [2] proctype producer {...}
```

The alternation is no more guaranteed. Simulation:

```
> spin prodcons2_flaw.pml | more
Produce
                                Consume
                                Consume
Produce
                                Consume
Produce
Produce
Produce
Produce
                                Consume
...
```

Producers/Consumers Extended

Reason:

```
> spin -i prodcons2_flaw.pml
Select a statement
    choice 3: proc  1 (producer) line  7 "prodcons2_flaw.pml" (state
4) [((turn==P))]
    choice 4: proc  0 (producer) line  7 "prodcons2_flaw.pml" (state
4) [((turn==P))]
Select [1-4]: 3
Select a statement
    choice 3: proc  1 (producer) line  9 "prodcons2_flaw.pml" (state
2) [printf('Produce\\n')]
    choice 4: proc  0 (producer) line  7 "prodcons2_flaw.pml" (state
4) [((turn==P))]
Select [1-4]: 4
```

Producers/Consumers Extended

A correct declaration for the producer:

```
active [2] proctype producer()
{
    do
        :: request(turn, P, N) ->
            printf("P%d\\n", _pid);
            assert(who == _pid);
            release(turn, C)
    od
}
```

- **assert** aborts the program if the expression returns a zero result, otherwise it is just passed.

Producers/Consumers Extended

Definition of `request`:

```
inline request(x, y, z) {
    atomic { x == y -> x = z; who = _pid }
}
```

- **inline** functions like C macros (their body is directly pasted into the body of a prototype at each point of invocation)
- **atomic**: when it starts, the process will keep running until all steps will complete.
- The executability of the atomic sequence is determined by the first statement.

Producers/Consumers Extended

File `prodcons2.pml`:

```
mtype = { P, C, N };

mtype turn = P;
pid   who;

inline request(x, y, z) {
    atomic { x == y -> x = z; who = _pid }
}

inline release(x, y) {
    atomic { x = y; who = 0 }
}
...
```

Producers/Consumers Extended

Simulation:

```
> spin prodcons2.pml | more
      P1
          C3
P0
          C3
      P1
          C3
      P1
```

```

          C2
    P0
          C3
    P1
...

```

Producers/Consumers Extended

Simulation can detect errors:

```

> spin false.pml
spin: line 1 "false.pml", Error: assertion violated
spin: text of failed assertion: assert(0)
#processes: 1
1:   proc 0 (:init:) line 1 "false.pml" (state 1)
1 process created

```

However, simulation cannot prove that error do not exist.

Producers/Consumers Extended

To prove that the assertions cannot be violated, we generate a verifier:

```

> spin -a prodcons2.pml
> cc -o pan pan.c
> ./pan
...
Full statespace search for:
    never claim           - (none specified)
    assertion violations  +
    acceptance  cycles   - (not selected)
    invalid end states    +

State-vector 28 byte, depth reached 7, errors: 0
...

```

Producers/Consumers Extended

Back to the flawed Producers/Consumers


```

mtype = { P, C };

mtype turn = P;

int msgs;

active [2] proctype producer()
{
    do
        :: (turn == P) ->
            printf("Produce\n");
            msgs++;
            turn = C
    od
}

active [2] proctype consumer()
{
    do
        :: (turn == C) ->
            printf("Consume\n");
            msgs--;
            turn = P
    od
}

active proctype monitor() {
    assert(msgs >= 0 && msgs <= 1)
}

```

```
spin -a prodcons2_flaw.pml && gcc -o pan pan.c && ./pan spin -t -p prodcons2_flaw.pml
```

2.3 Mutual Exclusion

The Mutual Exclusion problem

General algorithm

```

active [2] proctype mutex()
{
again:
    /* trying section */

    cnt++;
    assert(cnt == 1);          /* critical section */
    cnt--;

    /* exit section */
    goto again
}

```

The Mutual Exclusion problem

First tentative

```

bit flag;
byte cnt;

active [2] proctype mutex() {
again:
    flag != 1;

```

```

    flag = 1;
    cnt++;
    assert(cnt == 1);
    cnt--;
    flag = 0;
    goto again
}

```

The Mutual Exclusion problem

Second tentative

```

bit x, y;
byte cnt;

```

```

active proctype A() {
again:
    x = 1;
    y == 0;
    cnt++;
    /* critical section */
    assert(cnt == 1);
    cnt--;
    x = 0;
    goto again
}

```

```

active proctype B() {
again:
    y = 1;
    x == 0;
    cnt++;
    /* critical section */
    assert(cnt == 1);
    cnt--;
    y = 0;
    goto again
}

```

Dekker/Dijkstra algorithm

Trying session:

```

    flag[i] = true;
do
    :: flag[j] ->
        if
            :: turn == j ->
                flag[i] = false;
                !(turn == j);
                flag[i] = true
            :: else -> skip
        fi
    :: else ->
        break
od;

```

- flag indicates an intention to enter the critical section
- turn indicates who has priority between the two processes.

Dekker/Dijkstra algorithm

Exit session:

```
turn = j;  
flag[i] = false;
```

Dekker/Dijkstra algorithm

Verification:

```
> spin -a dekker.pml  
> cc -o pan pan.c  
> ./pan
```

...

Full statespace search for:

never claim	- (none specified)
assertion violations	+
acceptance cycles	- (not selected)
invalid end states	+

State-vector 20 byte, depth reached 67, errors: 0

...

Doran&Thomas change

Is the outer loop really necessary?

```
flag[i] = true;  
if  
:: flag[j] ->  
    if  
    :: turn == j ->  
        flag[i] = false;  
        !(turn == j);  
        flag[i] = true  
    :: else -> skip  
    fi  
:: else  
fi;
```

Doran&Thomas change

Verification:

```
> spin -a doran.pml  
> cc -o pan pan.c  
> ./pan
```

...

pan: assertion violated (cnt==1) (at depth 117)

pan: wrote doran.pml.trail

...

doran.pml.trail contains a counterexample with length 117.

Doran&Thomas change

We can use a breadth-first search to find the shortest counterexample:

```
> cc -DBFS -o pan pan.c
> ./pan
...
pan: assertion violated (cnt==1) (at depth 12)
pan: wrote doran.pml.trail
...
```

Doran&Thomas change

Now, we can perform a guided simulation:

```
> spin -p -t doran.pml
1:  proc 1 (mutex) line 8 ... [i = _pid]
2:  proc 1 (mutex) line 9 ... [j = (1-_pid)]
3:  proc 1 (mutex) line 11 ... [flag[i] = 1]
4:  proc 1 (mutex) line 21 ... [else]
5:  proc 1 (mutex) line 24 ... [cnt = (cnt+1)]
6:  proc 0 (mutex) line 8 ... [i = _pid]
7:  proc 0 (mutex) line 9 ... [j = (1-_pid)]
8:  proc 0 (mutex) line 11 ... [flag[i] = 1]
9:  proc 0 (mutex) line 13 ... [(flag[j])]
10: proc 0 (mutex) line 19 ... [else]
11: proc 0 (mutex) line 19 ... [(1)]
12: proc 0 (mutex) line 24 ... [cnt = (cnt+1)]
```

Peterson algorithm

A correct improvement: trying session

```
    flag[i] = true;
    turn = i;
    !(flag[j] && turn == i) ->
```

exit session

```
    flag[i] = false;
```

Verification:

```
> spin -a peterson.pml
> cc -o pan pan.c
> ./pan
...
State-vector 20 byte, depth reached 41, errors: 0
...
```

3 PROMELA overview

PROMELA

- PROMELA design is focused on process interaction at the system level
- Consequent features:
 - non-deterministic control structures,
 - primitives for process creation,
 - primitives for interprocess communication.
- Consequent lacks:
 - functions with return values,
 - expressions with side-effects,
 - data and functions pointers.

PROMELA is a language for building verification models (not a programming language)!

Types of objects

Three basic types of objects:

- processes
- data objects
- message channels

3.1 Processes

Process Initialization

- By means of **active** (instantiate an initial set of processes):

```
active [2] proctype you_run()
{
    printf("my pid is: %d\n", _pid)
}
```

- By means of **run** (creating new processes):

```
proctype you_run(byte x)
{
    printf("x = %d, pid = %d\n", x, _pid)
}
init {
    run you_run(0);
    run you_run(1)
}
```

Notes

- We cannot pass parameter values to *init* or to active processes.
- A newly created process may not start right after its initialization.
- To keep the system finite, only 256 processes can be alive in the same moment.
- A process “terminates” when it reaches the end of its code.
- A process “dies” when it has terminated and all processes instantiated later have died.
- A process may terminate without dying.

3.2 Data objects

Variable Scope

There are only two levels of scope:

- global: if it is declared outside all process declarations,
- process local: if it is declared within a process declaration.

```
init { /* x declared in outer block */
    int x;
    { /* y declared in inner block */
        int y;
        printf("x = %d, y = %d\n", x, y);
        x++;
        y++;
    }
    /* y remains in scope */
    printf("x = %d, y = %d\n", x, y);
}
```

Basic types

Type	Typical Range
bit	0, 1
bool	<i>false, true</i>
byte	0..255
chan	1..255
mtype	1..255
pid	0..255
short	$-2^{15} .. 2^{15} - 1$
int	$-2^{31} .. 2^{31} - 1$
unsigned	$0 .. 2^n - 1$

Typical declarations

```
bit x, y;           /* two single bits, initially 0 */
bool turn = true;   /* boolean value, initially true */
byte a[12];         /* all elements initialized to 0 */
chan m;            /* uninitialized message channel */
mtype n;           /* uninitialized mtype variable */
short b[4] = 89;    /* all elements initialized to 89 */
int cnt = 67;       /* integer scalar, initially 67 */
unsigned v : 5;     /* unsigned stored in 5 bits */
unsigned w : 3 = 5; /* value range 0..7, initially 5 */
```

Data structures

```
typedef Field {
    short f = 3;
    byte g
};
typedef Record {
    byte a[3];
    int fld1;
    Field fld2;
    chan p[3];
    bit b
};
proctype me(Field z) { z.g = 12 }
init { Record goo; Field foo;
    run me(foo)
}
```

Arrays and Data structures

- A structure can be passed as argument to a **run** statement, provided it contains no arrays. (In the example, *foo* can be passed, *goo* cannot.)
- Multi-dimensional arrays are not supported, although there are indirect ways:

```
typedef Array {
    byte el[4]
};

Array a[4];
```

3.3 Message Channels

Message Channels

- Channels are used to transfer messages between active processes.
- They store messages in first-in first-out order.
- Two types:
 - buffered channels,
 - rendezvous ports, also called synchronous channels.

Buffered Channels

- Declaration:

```
chan qname = [16] of { short, byte, bool }
```

This channel can store up to 16 messages, each consisting of 3 fields of the types listed.

- A field can be a user-defined type, but not an array.
- Sending a message:

```
qname!expr1,expr2,expr3
```

The process blocks if the channel is full.

- Receiving a message:

```
qname?var1,var2,var3
```

The process blocks if the channel is empty.

Alternative

- The first message field is a message type indication:

```
qname!expr1(expr2,expr3)
qname?var1(var2,var3)
```

- Some parameters can be given as constants:

```
qname?cons1,var2,cons2
```

The process blocks if the channel is empty and if the sent values do not match the constants.

- The built-in function `len` can be used to get the number of messages in a given channel:

```
len(qname)
```


Rendezvous Ports

- Declaration of a rendezvous port: it can pass single byte messages
- ```
chan port = [0] of { byte }
```
- The channel size is zero: the channel port can pass, but can not store messages.
  - Message interaction is synchronous: two processes execute a send and a receive statement at the same time.

```
mtype = { msgtype };
chan name = [0] of { mtype, byte };
active proctype A()
{
 name!msgtype(124);
 name!msgtype(121)
}
active proctype B()
{
 byte state;
 name?msgtype(state)
}
```

## Channels of channels

- Message parameters are always passed by value.
- We can pass the value of a channel from a process to another.

```
mtype = { msgtype };
chan glob = [0] of { chan };
active proctype A()
{
 chan loc = [0] of { mtype, byte };
 glob!loc;
 loc?msgtype(121)
}
active proctype B()
{
 chan who;
 glob?who;
 who!msgtype(121)
}
```

## 3.4 Executability

### Statements

- Every statement is either *executable* or *blocked*.

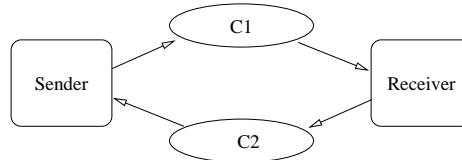
- Three main types of statements:
  - print statements
  - assignments
  - expression statements
- Print statements and assignments are always executable.
- Expression statements are executable iff they evaluate to true.
- Expressions must be side effect free.
- Exception: the `run` statement can be considered as a blocking expression:
  - it blocks when there are 256 processes alive;
  - if it does not block, it creates a new process.

## 4 Asynchronous Network Problems

### 4.1 Reliable FIFO Communication

#### Reliable FIFO Communication Problem

- We want to implement a reliable FIFO communication using less reliable channels.
- A user *Sender* sends messages to another user *Receiver* by means of two channels  $C_1$  and  $C_2$
- $C_1$  and  $C_2$  are non-reliable channels.
- The non-reliable channels may lose or duplicate the messages.



#### Alternating Bit Protocol

- *Sender* tags the messages with an alternating bit (e.g. it sends  $(0, \text{msg1})$ ,  $(1, \text{msg2})$ ,  $(0, \text{msg3})$ , ... ).
- *Sender* repeatedly sends a message with its tag until it receives a bit acknowledgment from *Receiver*.

- Suppose *Sender* has sent (**tag**, **msg**) and receives **b** as acknowledgment. If **b** is equal to **tag**, then it means that *Receiver* has received the right message, so it obtains a new message and tags it with a different value; otherwise it sends (**tag**, **msg**) again.
- Similarly, suppose *Receiver* receives (**tag**, **msg**). If **tag** is different from the last received bit, then it means that it is a new message; otherwise, the message is old. In both cases, *Receiver* sends **tag** back to *Sender* to communicate the correct receipt of the message.

### Alternating Bit Protocol

```

mtype = { msg, ack };

chan to_sndr = [2] of { mtype, bit };
chan to_rcvr = [2] of { mtype, bit };

active proctype Sender()
{
 ...
}

active proctype Receiver()
{
 ...
}
```

### Alternating Bit Protocol

```

active proctype Sender()
{
 bit seq_out, seq_in;

 /* obtain first message */
 do
 :: to_rcvr!msg(seq_out) ->
 to_sndr?ack(seq_in);
 if
 :: seq_in == seq_out ->
 /* obtain new message */
 seq_out = 1 - seq_out;
 :: else
 fi
 od
}
```

## Alternating Bit Protocol

```
active proctype Receiver()
{
 bit seq_in;
 do
 :: to_rcvr?msg(seq_in)
 :: to_sndr!ack(seq_in)
 od
}
```

### Example of simulation

- Receiver sends 2 ack (then it is blocked)
- Sender sends one message
- two possibilities:
  - Receiver receives the message
  - Sender receives one ack

### Exercise

- Try with:

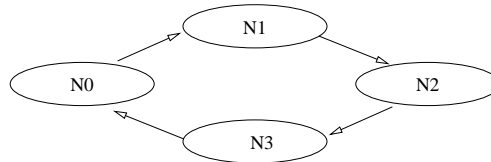
```
active proctype Receiver()
{
 bit seq_in, last_seq_in;
 int received;
 do
 :: to_rcvr?msg(seq_in, received) ->
 if
 :: (seq_in != last_seq_in) ->
 printf("Received: %d\n", received);
 last_seq_in = seq_in
 :: else
 fi;
 to_sndr!ack(seq_in)
 od
}
```

## 4.2 Leader Election

### Leader Election Problem

- $N$  processes are the nodes of a unidirectional ring network: each process can send messages to its clockwise neighbor and receive messages from its counterclockwise neighbor.

- The requirement is that, eventually, only one process will output that it is the leader.
- We assume that every process has a unique identifier.
- The leader must have the highest identifier



### Le Lann, Chang, Roberts (LCR) solution

- Initially, every process passes its identifier to its successor.
- When a process receives an identifier from its predecessor, then:
  - if it is greater than its own, it keeps passing the identifier;
  - if it is smaller, it discards the identifier.
  - if it is equal to its own identifier, it declares itself leader.
    - \* The leader informs the others that it is the leader.
    - \* After a process receives the message with the id of the leader, it exits.

Hint:

```

mtype = { candidate, leader };
chan c[N] = [BUFSIZE] of { mtype, byte };

proctype node(chan prev, next; byte mynumber) { ... }

init {
 ...
 do
 :: proc <= N -> run node(...);
 ...
}

```

### Exercises

- Implement the Le LCR algorithm.
- Verify there can be at most one leader (using Xspin, next lecture).