Theory of concurrency

Seminar 1

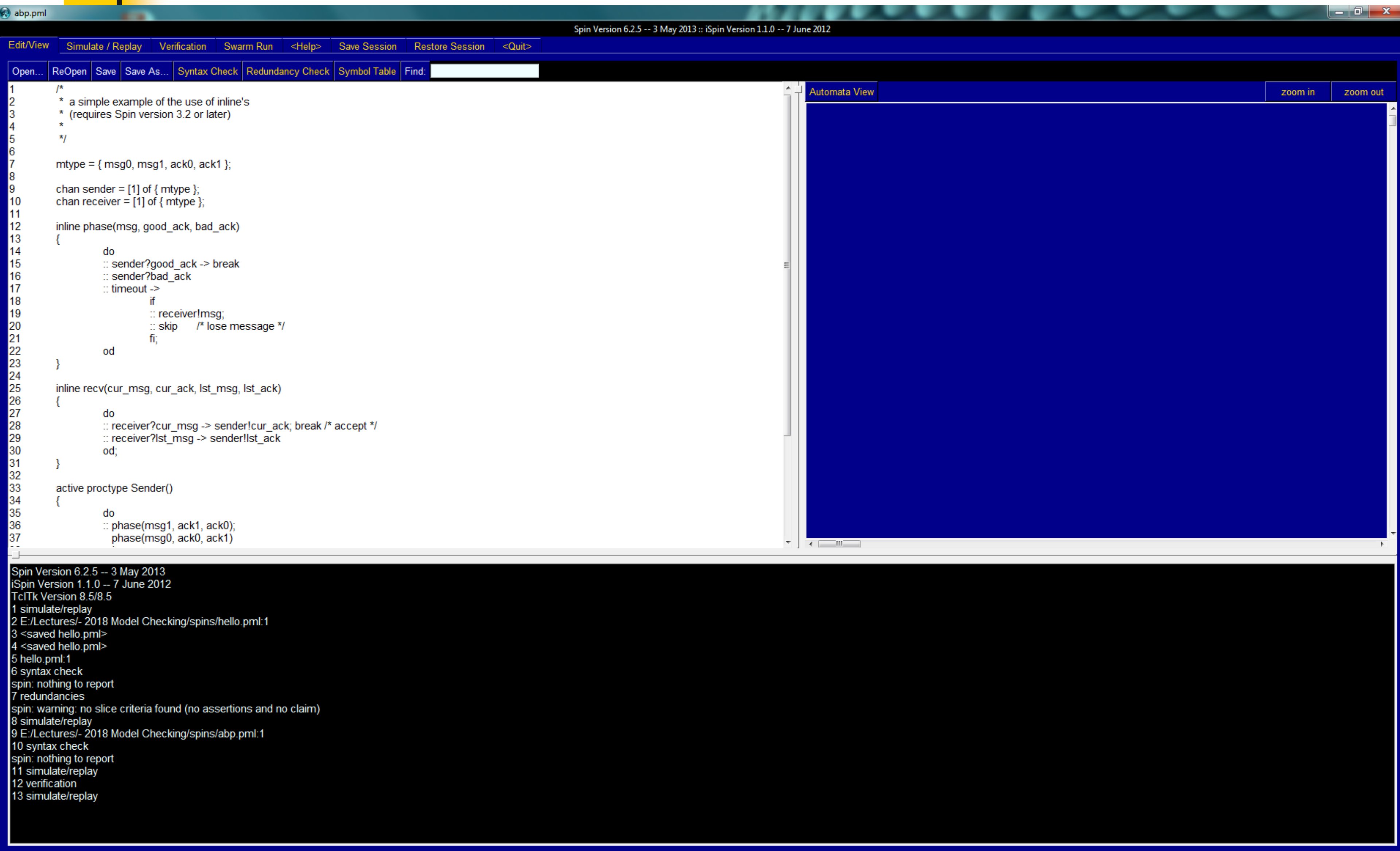
SPIN

http://spinroot.com/spin/whatispin.html

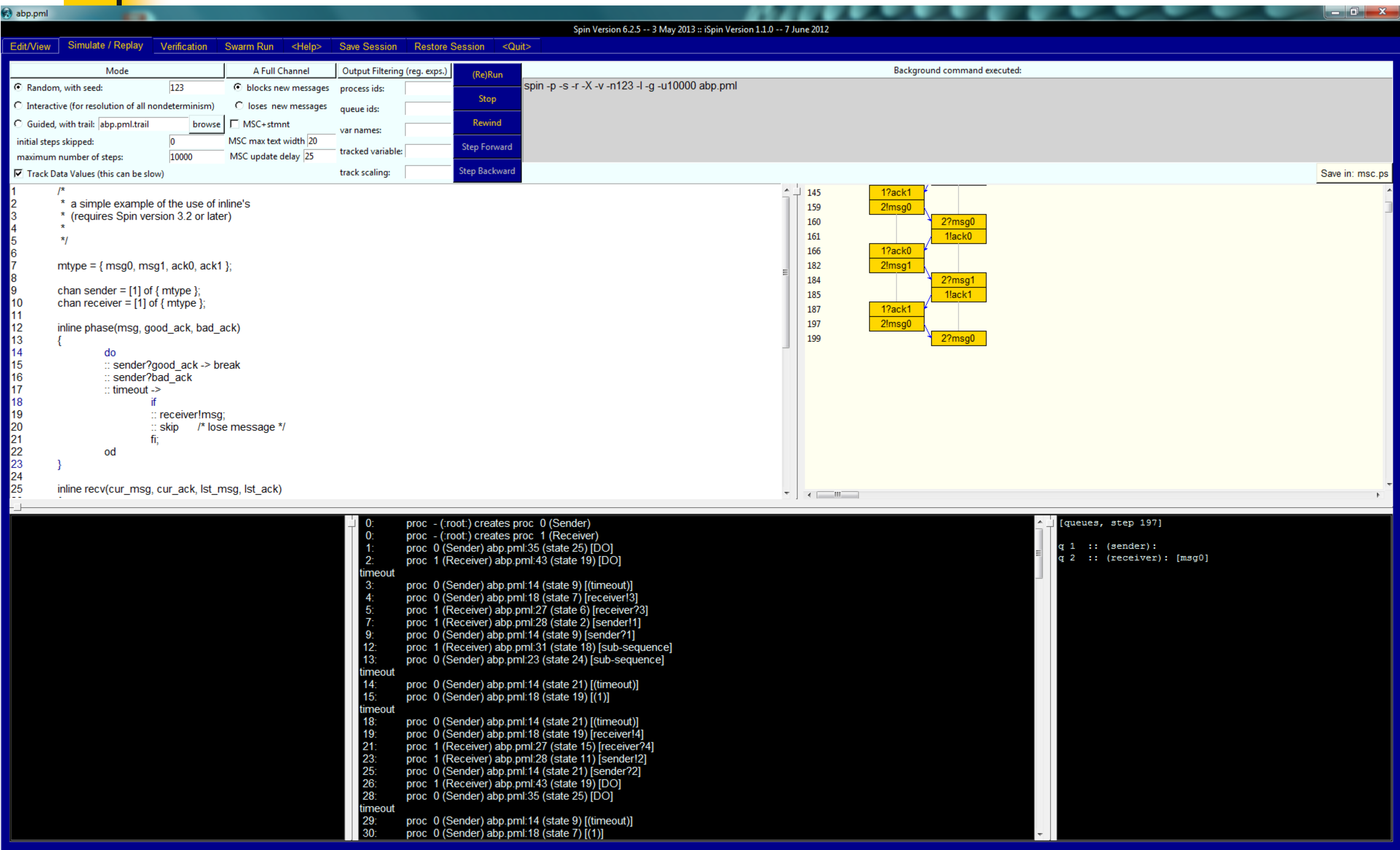
iSPIN

GUI

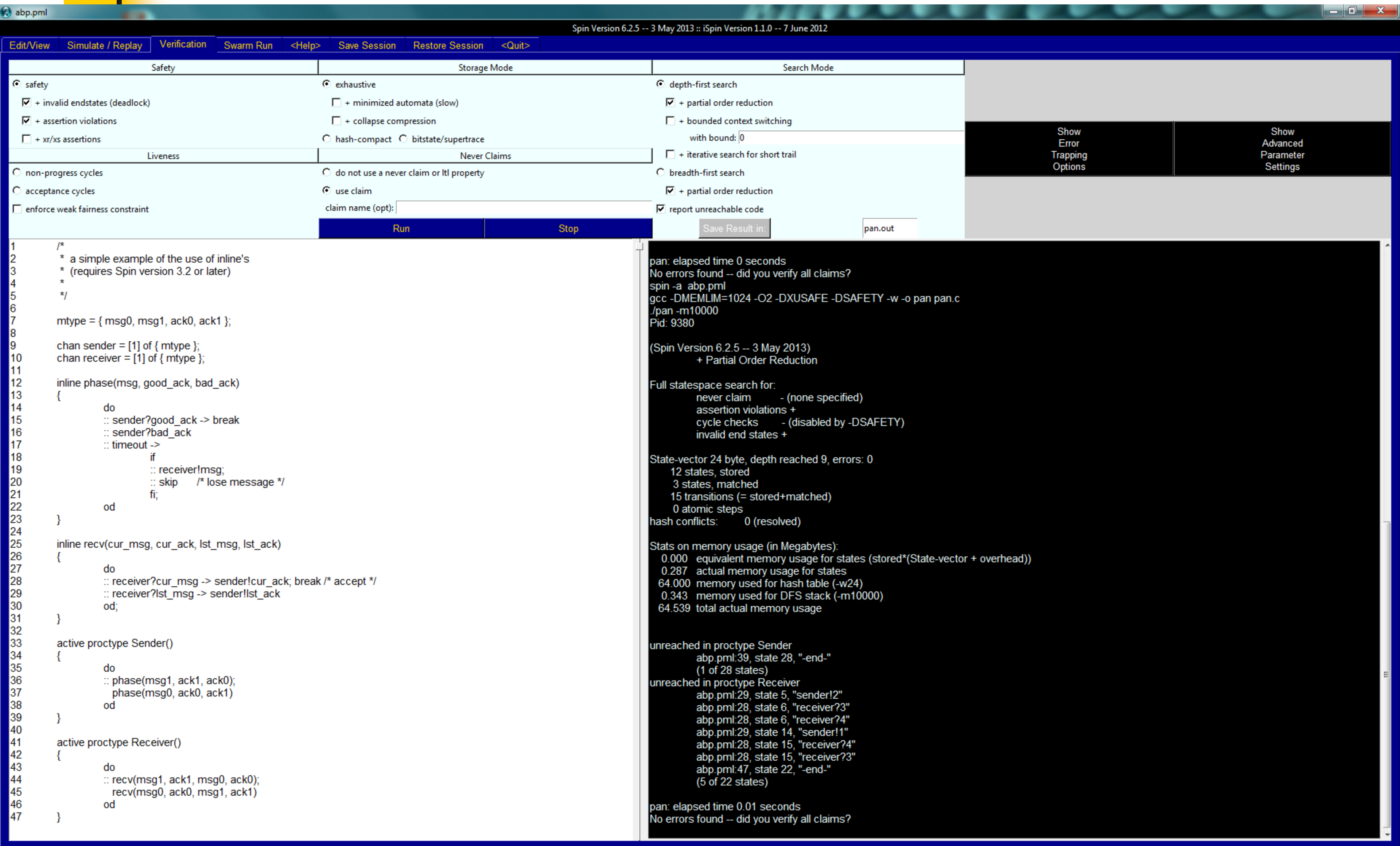
iSPIN


abp.pml
Spin Version 6.2.5 -- 3 May 2013 :: iSpin Version 1.1.0 -- 7 June 2012
Edit/View
Simulate / Replay
Verification
Swarm Run
<Help>
Save Session
Restore Session
<Quit>
Open...
ReOpen
Save
Save As...
Syntax Check
Redundancy Check
Symbol Table
Find:
/*
 * a simple example of the use of inline's
 * (requires Spin version 3.2 or later)
 *
 */
mtype = { msg0, msg1, ack0, ack1 };
chan sender = [1] of { mtype };
chan receiver = [1] of { mtype };
inline phase(msg, good_ack, bad_ack)
{
do
:: sender?good_ack -> break
:: sender?bad_ack
:: timeout ->
if
:: receiver!msg;
:: skip /* lose message */
fi;
od
}
inline recv(cur_msg, cur_ack, lst_msg, lst_ack)
{
do
:: receiver?cur_msg -> sender!cur_ack; break /* accept */
:: receiver?lst_msg -> sender!lst_ack
od;
}
active proctype Sender()
{
do
:: phase(msg1, ack1, ack0);
phase(msg0, ack0, ack1)
Automata View
zoom in
zoom out
Spin Version 6.2.5 -- 3 May 2013
iSpin Version 1.1.0 -- 7 June 2012
TclTk Version 8.5/8.5
1 simulate/replay
2 E:/Lectures/- 2018 Model Checking/spins/hello.pml:1
3 <saved hello.pml>
4 <saved hello.pml>
5 hello.pml:1
6 syntax check
spin: nothing to report
7 redundancies
spin: warning: no slice criteria found (no assertions and no claim)
8 simulate/replay
9 E:/Lectures/- 2018 Model Checking/spins/abp.pml:1
10 syntax check
spin: nothing to report
11 simulate/replay
12 verification
13 simulate/replay

iSPIN


abp.pml
Spin Version 6.2.5 -- 3 May 2013 :: iSpin Version 1.1.0 -- 7 June 2012
Edit/View
Simulate / Replay
Verification
Swarm Run
<Help>
Save Session
Restore Session
<Quit>
Mode
Random, with seed: 123
Interactive (for resolution of all nondeterminism)
Guided, with trail: abp.pml.trail browse
initial steps skipped: 0
maximum number of steps: 10000
Track Data Values (this can be slow)
A Full Channel
blocks new messages
loses new messages
MSC+stmnt
MSC max text width 20
MSC update delay 25
Output Filtering (reg. exps.)
process ids:
queue ids:
var names:
tracked variable:
track scaling:
(Re)Run
Stop
Rewind
Step Forward
Step Backward
Background command executed:
spin -p -s -r -X -v -n123 -l -g -u10000 abp.pml
Save in: msc.ps
/*
 * a simple example of the use of inline's
 * (requires Spin version 3.2 or later)
 *
 */
mtype = { msg0, msg1, ack0, ack1 };
chan sender = [1] of { mtype };
chan receiver = [1] of { mtype };
inline phase(msg, good_ack, bad_ack)
{
do
:: sender?good_ack -> break
:: sender?bad_ack
:: timeout ->
if
:: receiver!msg;
:: skip /* lose message */
fi;
od
}
inline recv(cur_msg, cur_ack, lst_msg, lst_ack)
145 1?ack1
159 2!msg0
160 2?msg0
161 1!ack0
166 1?ack0
182 2!msg1
184 2?msg1
185 1!ack1
187 1?ack1
197 2!msg0
199 2?msg0
0: proc - (:root:) creates proc 0 (Sender)
0: proc - (:root:) creates proc 1 (Receiver)
1: proc 0 (Sender) abp.pml:35 (state 25) [DO]
2: proc 1 (Receiver) abp.pml:43 (state 19) [DO]
timeout
3: proc 0 (Sender) abp.pml:14 (state 9) [(timeout)]
4: proc 0 (Sender) abp.pml:18 (state 7) [receiver!3]
5: proc 1 (Receiver) abp.pml:27 (state 6) [receiver?3]
7: proc 1 (Receiver) abp.pml:28 (state 2) [sender!1]
9: proc 0 (Sender) abp.pml:14 (state 9) [sender?1]
12: proc 1 (Receiver) abp.pml:31 (state 18) [sub-sequence]
13: proc 0 (Sender) abp.pml:23 (state 24) [sub-sequence]
timeout
14: proc 0 (Sender) abp.pml:14 (state 21) [(timeout)]
15: proc 0 (Sender) abp.pml:18 (state 19) [(1)]
timeout
18: proc 0 (Sender) abp.pml:14 (state 21) [(timeout)]
19: proc 0 (Sender) abp.pml:18 (state 19) [receiver!4]
21: proc 1 (Receiver) abp.pml:27 (state 15) [receiver?4]
23: proc 1 (Receiver) abp.pml:28 (state 11) [sender!2]
25: proc 0 (Sender) abp.pml:14 (state 21) [sender?2]
26: proc 1 (Receiver) abp.pml:43 (state 19) [DO]
28: proc 0 (Sender) abp.pml:35 (state 25) [DO]
timeout
29: proc 0 (Sender) abp.pml:14 (state 9) [(timeout)]
30: proc 0 (Sender) abp.pml:18 (state 7) [(1)]
[queues, step 197]
q 1 :: (sender):
q 2 :: (receiver): [msg0]

iSPIN


abp.pml
Spin Version 6.2.5 -- 3 May 2013 :: iSpin Version 1.1.0 -- 7 June 2012
Edit/View
Simulate / Replay
Verification
Swarm Run
<Help>
Save Session
Restore Session
<Quit>
Safety
safety
+ invalid endstates (deadlock)
+ assertion violations
+ xr/xs assertions
Liveness
non-progress cycles
acceptance cycles
enforce weak fairness constraint
Storage Mode
exhaustive
+ minimized automata (slow)
+ collapse compression
hash-compact
bitstate/supertrace
Never Claims
do not use a never claim or ltl property
use claim
claim name (opt):
Run
Stop
Search Mode
depth-first search
+ partial order reduction
+ bounded context switching
with bound: 0
+ iterative search for short trail
breadth-first search
+ partial order reduction
report unreachable code
Save Result in:
pan.out
Show Error Trapping Options
Show Advanced Parameter Settings
/*
 * a simple example of the use of inline's
 * (requires Spin version 3.2 or later)
 *
 */
mtype = { msg0, msg1, ack0, ack1 };
chan sender = [1] of { mtype };
chan receiver = [1] of { mtype };
inline phase(msg, good_ack, bad_ack)
{
do
:: sender?good_ack -> break
:: sender?bad_ack
:: timeout ->
if
:: receiver!msg;
:: skip /* lose message */
fi;
od
}
inline recv(cur_msg, cur_ack, lst_msg, lst_ack)
{
do
:: receiver?cur_msg -> sender!cur_ack; break /* accept */
:: receiver?lst_msg -> sender!lst_ack
od;
}
active proctype Sender()
{
do
:: phase(msg1, ack1, ack0);
phase(msg0, ack0, ack1)
od
}
active proctype Receiver()
{
do
:: recv(msg1, ack1, msg0, ack0);
recv(msg0, ack0, msg1, ack1)
od
}
pan: elapsed time 0 seconds
No errors found -- did you verify all claims?
spin -a abp.pml
gcc -DMEMLIM=1024 -O2 -DXUSAFE -DSAFETY -w -o pan pan.c
./pan -m10000
Pid: 9380
(Spin Version 6.2.5 -- 3 May 2013)
+ Partial Order Reduction
Full statespace search for:
never claim - (none specified)
assertion violations +
cycle checks - (disabled by -DSAFETY)
invalid end states +
State-vector 24 byte, depth reached 9, errors: 0
12 states, stored
3 states, matched
15 transitions (= stored+matched)
0 atomic steps
hash conflicts: 0 (resolved)
Stats on memory usage (in Megabytes):
0.000 equivalent memory usage for states (stored*(State-vector + overhead))
0.287 actual memory usage for states
64.000 memory used for hash table (-w24)
0.343 memory used for DFS stack (-m10000)
64.539 total actual memory usage
unreached in proctype Sender
abp.pml:39, state 28, "-end-"
(1 of 28 states)
unreached in proctype Receiver
abp.pml:29, state 5, "sender!2"
abp.pml:28, state 6, "receiver?3"
abp.pml:28, state 6, "receiver?4"
abp.pml:29, state 14, "sender!1"
abp.pml:28, state 15, "receiver?4"
abp.pml:28, state 15, "receiver?3"
abp.pml:47, state 22, "-end-"
(5 of 22 states)
pan: elapsed time 0.01 seconds
No errors found -- did you verify all claims?

Promela

Specification language

Objects

Objects: processes, message channels, and variables.

Processes

- Declaration `proctype`
  - at least one process
  - announced globally
  - defines the behavior, but does not run the process
- Instance of process starts
  - Prefix `active` :
    - `active [2] proctype foo(){printf("MSC: my pid is:% d \ n",_pid)}`
  - `run` statement:
    - `active proctype bar () {run foo ()}`
- `_pid` - the reserved variable for the non-negative value of the unique instance process identifier

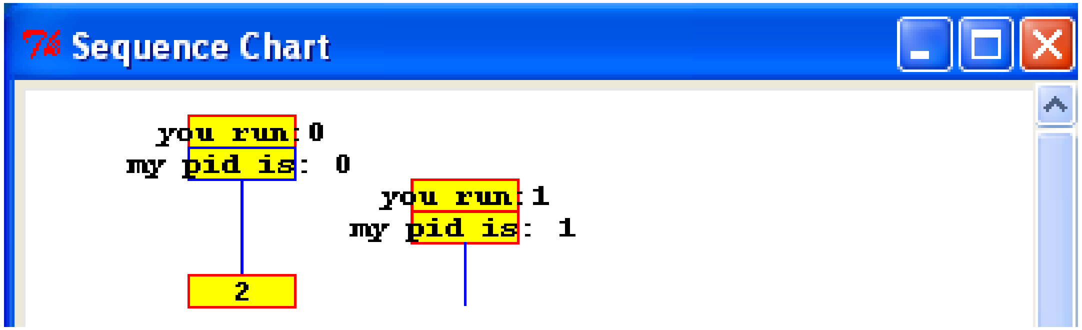
Objects

- Process body:
  - declarations of data and operators (may be empty)
  - operator separators:
    - semicolon "`;`"
      - empty operator allowed `; ; ; ;`
    - arrow “`->`”
      - for indicating a causal relationship between two operators

- Values of variables or messages in channels
  - changes or is checked only in processes.

Objects

- Simulation window: two processes of type **`you_run`** are created




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

- Sequence Chart
  - each column displays one running process.

Objects

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

- `init` is basic process in Promela
  - is always activated in an initial state of the model
  - cannot take parameters or be copied.
  - its identifier `_pid` is always 0.
  - can be unnecessary process which increase the size of the model
- A running process terminates when
  - it reaches the end of its body, but no later than the processes that it run.
- The number of processes in Spin is no more than 256.

Objects

Data type	Range
`bit`	0,1
`bool`	false, true
`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`	$0 .. 2^{32}-1$

Objects

- Variables
  - global vars are declared outside the process description
  - local vars are declared in the process description
    - you cannot restrict access to a local variable for a part of the process
      - no block or scope
  - initialized to zeros (`false`).

Objects

- One-dimensional arrays
  - `byte state[N]`
    - `state[0] = state[3] + 5 * state[3*2/n]`
      - `n` – constant or variable.
  - numbering from 0
  - array index is any expression with an integer value
    - out of range `0..N-1` result not defined
- Multidimensional arrays can be specified implicitly using the construct `typedef`.

Objects

- Enumerated type
  - **mtype**
  - symbolic values of variables
  - one or more declarations
    - all variables declared as **mtype** can take all declared values
  - 255 values in **mtype**.

```
mtype = { grandad, grandma, granddaughter, dog, cat, mice, turnip };
mtype = { mammal, vegetable };
init {
       mtype n = grandad; /* initializing n by value grandad */
       printf("MSC: %e ", n);
       n = vegetable; /* assigning n by value vegetable */
       printf("MSC: is not %e\n ", n)
}
```

Objects

- Operator `printf`
  - two arguments: a string and a list of arguments
  - formats of output variables
    - `d` is an integer in decimal format,
    - `i` is an unsigned integer value,
    - `c` is one character,
    - `e` is a constant of type `mtype`.
  - To display a message on the iSpin interaction diagram, the line starts with the characters "**MSC:**".

Objects

Channels

- Model data transfer from one process to another.
- Declared locally or globally with reserved word **`chan`**:
  - **`chan qname = [16] of {short}`**
  - **`qname`** channel with buffer 16 for **`short`** messages
- Send messages in FIFO order: first in - first out.
- The operator of sending messages “**`!`**”:
  - **`qname ! expr`**
  - adds this value to the end of the queue in the channel
  - executed if the destination channel is not full, otherwise it is blocked.
- The operator of receiving the message “**`?`**”:
  - **`qname ? msg`**
  - saves the value from the beginning of the queue in the channel to the **`msg`** variable
  - executed if the destination channel is not empty, otherwise it is blocked.

Objects

- Composed messages
  - a finite number of fields
  - `chan pname = [16] of {byte, int, chan}`
    - one eight-bit value (of type `byte`)
    - one 32-bit value (of type `int`)
    - channel name.
- Sending a channel identifier from one process to another
  - in the message
  - as a parameter to a process instance
- No arrays in message fields.
- Sending multiple values in a single message
  - `pname ! expr1, expr2, expr3`
- Receive such a message
  - `pname ? var1, var2, var3`

Objects

- Using the first message field to specify the type of message
  - In channels, `mtype` data is always interpreted symbolically, not numerically.

```
/* the declaration of message type */
mtype = { ack, nak, err, next, accept }
/* the declaration of veriable of this type */
mtype msgtype1, msgtype2;
```

- `chan tname = [4] of { mtype, int, bit };`
- `tname ! msgtype (data, b)`
  - `tname ! msgtype, data, b`

Objects

- Sending or receiving constants:

```
tname ! ack, var, 0
tname ? ack (data, 1)
```

- The operator of receiving a message with constants is executable,
  - only if the message at the beginning of the channel buffer in the corresponding fields has the values of the specified constants.
    - Otherwise, it will be blocked.
  - The operator of receiving a message is not executed if
    - the message `<ack, 15, 0>` is at the beginning of the buffer.
- Unexecuted statements pause the process until they become executable.
  - In this case, operations on channels may simulate point-to-point communication of several processes connected by one channel.

Objects

- Channel functions
  - `len`, `empty`, `nempty`, `full`, `nfull`.
- `len(qname)`
  - the number of messages in channel **qname**
  - unexecuted if used as an operator on the right side of the assignment and the channel is empty,
    - because it returns a null result, which by definition means that the statement is temporarily unexecuted.
- Sending **msgtype** messages if the **qname** channel is not full:
  - `(len(qname) < MAX) -> qname ! msgtype`
  - If access to the **qname** channel is shared by several processes, the execution of the second statement will not necessarily occur immediately after the execution of the first test statement.

Objects

Rendezvous interaction

- `chan port = [0] of {byte}`
- Rendezvous channel
  - the rendezvous channel buffer is zero: can transmit but cannot store messages.
  - Process interactions on rendezvous channels are synchronous by definition

```
#define msgtype 1
chan name = [0] of { byte, byte };
proctype A() {
        name ! msgtype(4);
        name ! msgtype(1) }
proctype B() {
        byte state;
        name ? msgtype(state) }
init {
        atomic { run A(); run B() }
}
```

Objects

- The channel **`name`** is declared as a global rendezvous channel.
- Two processes will synchronously execute their first statements:
  - handshake according to **`msgtype`** message
  - passing the value 13 to the local variable **`state`**.
- The second send statement in process **`A`** is not executed.
  - there is no corresponding message receiving operation in process **`B`**.

```
#define msgtype 1
chan name = [0] of { byte, byte };
proctype A() {
        name ! msgtype(13);
        name ! msgtype(1) }
proctype B() {
        byte state;
        name ? msgtype(state) }
init {
        atomic { run A(); run B() }
}
```

Objects

```
#define msgtype 1
chan name = [0] of { byte, byte };
proctype A() {
        name ! msgtype(4);
        name ! msgtype(1) }
proctype B() {
        byte state;
        name ? msgtype(state) }
init { atomic { run A(); run B()}}
```

- The size of buffer **`name`** is 2
  - **`A`** may terminate execution before **`B`** starts working.
- The size of buffer **`name`** is 1
  - Process **`A`** may terminate its first sending action
    - blocked on the second action, because now the channel is full.
  - Process **`B`** reads the first message and terminates.
  - At this point, **`A`** becomes executed again and terminates, leaving its last message on the channel.
- Binary rendezvous interactions:
  - only two processes, sender and receiver, can be synchronized.

Operators

A loop of wait:

```
while (a != b) -> skip
(a == b)
```

The rule of executability

- Executability provides the foundation for modeling process synchronization.
- Any statement is either executable or blocked.
- The main types of operators:
  - `printf` variable output statement (always executable),
  - assignment operator (always executable),
  - S/R operators (when transmitting data over channels),
  - expression operators.
- If the process reaches the code point with the unexecuted statement, then the process is blocked.
  - An operator can become executable if another active process performs actions that allow the operator blocked in this process to execute further.

Operators

- Two processes share access to the global variable **`state`**.
- Processes await the condition **`(state == 1)`**.
- If program terminates, then **`state`** can have a value: 0, 1, or 2.
- If one of the processes changes the value of **`state`** before the condition is checked by another process, then the other process will be blocked.

```
byte state = 1;
active proctype A(){
        byte tmp;
        (state==1) -> tmp = state;
        tmp = tmp+1;
        state = tmp }
active proctype B(){
        byte tmp;
        (state==1) -> tmp = state;
        tmp = tmp-1;
        state = tmp }
```

Operators

Expressions

- Expressions in Promela are statements
  - tested for true/false in any context.
- Expression executable iff
  - its boolean value is true
    - equivalent to any nonzero value.

Operators	
`()` `[]`	brackets, array brackets
`!` `++` `--`	negation, plus 1, minus 1
`*` `/` `%`	multiplication, division, modulo division
`+` `-`	addition, subtraction
`<<` `>>`	left shift, right shift
`<` `<=` `>` `>=`	comparison
`==` `!=`	equality, non-equality
`&`	bitwise and
`^`	bitwise xor
`|`	bitwise or
`&&`	logical and
`||`	logical or
`->` `:`	if operator
`=`	assignment

Operators

The assignment operator

- **`variable = expression`**
  - is not an statement, just like the **`print`** operator
  1. the expression to the right is evaluated
  2. the result of the expression is converted to the variable type
  3. variable gets this value.
- Increment and decrement
  - only postfix (**`a++`**, not **`++a`**)
  - only in the expression, but not in the assignment operator.

Operators

Sending/Receiving operators

- is executable if message sending/receiving is possible
  - otherwise the process is blocked.
- How to test the ability to send/receive without execution.
  - take operator arguments in square brackets
    - the operator is not executed, but its value is computed as an expression
  - **`qname ? [ack, var, 0]`**
    - test that the next message in the **`qname`** channel is a structure consisting of
      - the mnemonic value **`ack`**, some value of the **`var`** variable, and 0.
  - If the statement is executable, then 1 is returned, otherwise 0 is returned.

Operators

- Invalid expressions of the form
  - `(qname ? var == 0)` or `(a > b && qname ! 123)`
  - cannot be computed without side effects
    - attempts to perform S/R operations
- The sending and receiving operators are not expressions.

- Operator `printf`
  - output of variable values or text
  - `printf` and `skip` do not change the state of the system
    - used when an executable step is necessary
  - text output is only in simulation mode
    - is a side effect of the operator.

Control Flow

Composition operators: `atomic`, `d_step`, *choice* and *repetition*.

Block `atomic`

- `atomic {op1 op2 ... opn}`
  - block **`op1`**, **`op2`**, ..., **`opn`** is executed as an indivisible module, not alternating with other processes.
  - similar to using semaphore.
  - other processes "see" shared global variables and channels used in the `atomic` block, either *before* or *after* the execution of the entire sequence of statements.
  - if the statement inside `atomic` is not executable, then the whole block is not executable and *another process may act*.
  - reducing the complexity of models
    - decreasing the number of global states
      - `atomic` blocks limit the number of interleavings
    - Ensure the correct implementation of `atomic` blocks.

Control Flow

- `atomic` prevents a competing process from accessing a global variable.
- The final value of `state` is 0 or 2.

```
byte state = 1;
active proctype A(){
        atomic {
        (state==1) -> state = state+1 } }
active proctype B(){
        atomic {
        (state==1) -> state = state-1 } }
```

Control Flow

Block of deterministic steps `d_step`

- `d_step {op1 op2 ... opn}`
  - block **op1**, **op2**, ..., **opn** is executed as an indivisible module, not alternating with other processes.
  - other processes "see" shared global variables and channels used in the `d_step` block, either *before* or *after* the execution of the entire sequence of statements.
  - if the statement inside `d_step` is not executable, then the whole block is not executable and this is a *modeling error*.
  - more powerful decreasing verification complexity
    - for a sequence in `atomic` Spin generates transitions for other processes, unlike `d_step`.

Control Flow

```
if
  :: (a != b) -> option1
  :: (a == b) -> option2
fi
```

The guarded choice

- `if`
  - contains at least two sequences of operators.
  - only one sequence from the list of executable is performed.
    - a sequence is executable if its first statement is executable.
    - the first statement is called a *guard* or *condition*.
  - if several operators are executable, one is chosen nondeterministically
    - the order of listing alternatives does not matter
  - if all conditions are not true, then the process will be blocked until one of the conditions becomes true.
  - There are no restrictions on the types of expressions for conditions.

Control Flow

- `option1` is executable if the channel contains message **a**.
- `option2` is executable if the channel contains message **b**.
- Which operator will be executed depends on the relative speeds of the processes.

```
#define a 1
#define b 2
chan ch = [1] of { byte };
proctype A(){ ch ! a }
proctype B(){ ch ! b }
proctype C(){
        if
           :: ch ? a -> option1
           :: ch ? b -> option2
        fi }
init{
        atomic {
                run A(); run B(); run C()}}
```

Control Flow

- The process for changing the value of variable **`count`**.
- Both expressions in the example are always executable.
  - the choice between them is completely non-deterministic.

```
byte count;
proctype counter(){
        if
            :: count = count + 1
            :: count = count - 1
        fi
}
```

Control Flow

The loop operator

- only one option can be chosen for execution
- after terminating the selected option, control moves to the beginning of the loop
- exit the loop with using the `break` statement
- In the example
  - the loop will be terminated when `count == 0`.
  - two other statements are always executable
    - the result is non-deterministic

```
byte count;
proctype counter(){
        do
            :: count = count + 1
            :: count = count - 1
            :: (count == 0) -> break
        od
}
```

Control Flow

- Guarantee of termination of the loop under the desirable condition.

```
byte count;
proctype counter(){
        do
            :: (count != 0) ->
                if
                    :: count = count + 1
                    :: count = count - 1
                fi
            :: (count == 0) -> break
        od
}
```

Control Flow

- `else` condition.
  - is executable in choice or loop statements only if no other condition is executable.

```
byte count;
proctype counter(){
        do
            :: (count != 0) ->
                if
                    :: count = count + 1
                    :: count = count - 1
                fi
            :: else -> break
        od
}
```

  - the `else` condition is true when
    - `!(count != 0)` ≅ `(count == 0)`.

Control Flow

- Unconditional jump statement **goto**
  - is always executable if there is a label to which the jump is made.
- Euclidean algorithm for finding GCD:

```
proctype Euclid (int x, y){
        do
            :: (x > y) -> x = x - y
            :: (x < y) -> y = y - x
            :: (x == y) -> goto done
        od;
        done: skip
}
```

- The *label* can only be placed before the operator.
  - empty **skip** statement
    - is always executable, but has no effect.

Example: Message filter

- It receives messages from channel **`ch`**
- It divides them into two channels **`large`** and **`small`**
- **`ch`** channel is empty: the process is blocked

```
#define N 128
#define size 16
chan ch = [size] of { short };
chan large = [size] of { short };
chan small = [size] of { short };
proctype split(){
        short data;
        do :: ch ? data ->
                 if
                   :: (data >= N) -> large ! data
                   :: (data < N)  -> small ! data
                 fi
        od }
init{ run split() }
```

Example: split and merge

```
#define N 128
#define size 16
chan ch = [size] of { short };
chan large = [size] of { short };
chan small = [size] of { short };
proctype split(){
        short data;
        do :: ch ? data ->
                  if :: (data >= N) -> large ! data
                     :: (data < N)  -> small ! data
                  fi
        od }
proctype merge(){
        short data;
        do :: if :: large ? data
                 :: small ? data
              fi;
              ch ! data
        od }
init{   ch ! 345; ch ! 13; ch ! 6777; ch!32; ch ! 0;
        run split(); run merge() }
```

- Nonterminating processes for splitting and merging.

Example: a recursive process

- The return value is passed back to the calling process in a global variable or message.

```
proctype fact( int n; chan p) {
        chan child = [1] of { int };
        int result;
        if
          :: (n <= 1) -> p ! 1
          :: (n >= 2) -> run fact(n-1, child);
                         child ? result;
                         p ! n*result
        fi }
init{
        chan child = [1] of { int };
        int result;
        run fact(7, child);
        child ? result;
        printf("MSC: result: %d\n", result) }
```