The SPIN Model Checker

[Based on The SPIN Model Checker: Primer and Reference Manual, Gerard J. Holzmann]

Yi-Hsiung Wang

Dept. of Information Management
National Taiwan University

May 12, 2010

Agenda

- An Introduction to SPIN
- An Overview of PROMELA
- Verification in SPIN
- DEMO
- References

Agenda

- An Introduction to SPIN
  - History of SPIN
  - What is SPIN
- An Overview of PROMELA
- Verification in SPIN
- DEMO
- References

History of SPIN

- The tool was developed at Bell Labs in the original Unix group of the Computing Sciences Research Center, starting in 1980 by Gerard Holzmann and others.
- The software has been available freely since 1991, and continues to evolve to keep pace with new developments in the field.
- In April 2002 the tool was awarded the prestigious System Software Award for 2001 by the ACM.

What is SPIN

- SPIN (Simple PROMELA INterpreter)
  - Is a popular open-source software that can be used for formal verification of distributed software systems.
  - It supports the design and verification of asynchronous process system.
  - The verification models of SPIN are focused on proving the correctness of process interactions, and abstract from internal sequential computations.

What is SPIN (cont.)

- As a formal methods tool, SPIN aims to provide:
  - an intuitive, program-like notation for specifying design choices unambiguously, without implementation detail,
  - a powerful, concise notation for expressing general correctness requirements,
  - a methodology for establishing the logical consistency of the design from above.
- The tool supports a high level language to specify system description, called PROMELA (PROcess MEta LAnguage).

What is SPIN (cont.)

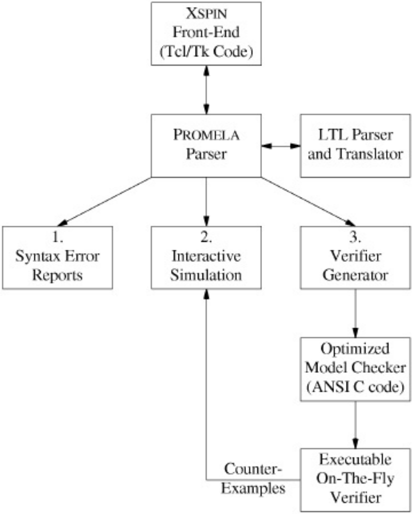


Fig. 1. The structure of SPIN simulation and verification.

Agenda

- An Introduction to SPIN
- An Overview of PROMELA
  - What is PROMELA
  - PROMELA Model
  - Correctness Claim
  - PROMELA Semantic
- Verification in SPIN
- DEMO
- References

What is PROMELA

- PROMELA (PROcess MEta-LAnguage)
  - PROMELA is not an implementation language but a system description language.
  - The emphasis is on the modeling of process synchronization and coordination, not on computation.
  - resembles the programming language C.

What is PROMELA (cont.)

- Models that can be specified in PROMELA are required to be bounded:
  - There can be only finitely many running processes.
  - There can only be finitely many statements in a proctype.
  - All data types have a finite range.
  - All message channels have an a bounded capacity.
- Enforcing that restriction helps to guarantee that any correctness property that can be stated in PROMELA is decidable.

What is PROMELA (cont.)

- A PROMELA model is constructed from three basic types of objects:
  - Processes
  - Data objects
  - Message channels

Process

- Defined by using proctype keyword or init keyword.
- There are two ways to instantiation a process:
  - Adding the prefix active to a proctype declaration
  - Using a run operator

Example1: Hello World

```
active proctype begin(){
    printf("Hello World\n")
}
```

Example2: Hello World

```
proctype begin2(){
    printf("Hello World Again\n")
}
init{
    run begin2()
}
```

- Note: Semicolon is defined as a separator, not terminator.

Process (cont.)

- By using run operator, we can pass the value to process (passing by value).
- If processes created through active keyword, their parameters are initialized to zero.

```
proctype value_pass ( byte x ){
    printf(" x = %d\n ",x)
}

init{
    run value_pass (0);
    run value_pass (1);
}
```

Process (cont.)

- We can create multiple instantiations by adding the desired number in square brackets.
- Processes are executes concurrently with all other processes.
- They can interleave their statement executions in arbitrary ways with other processes.
- Each running process has a unique process instantiation number, and can be accessed by local variable _pid.

Example:Hello World

```
active [2] proctype main(){

    printf("my pid is: %d\n",_pid)

}

/*   Output will be:   my pid is: 0   */
/*                     my pid is: 1   */
```

Process termination

- A process "terminates" when it reaches the end of its code (the closing curly brace).
- A process can only "die" and be removed if all processes instantiated later than this process have died first.
- Process can terminate in any order, but they can only die in the reverse order of their creation.

Data Objects

- The default initial value of all data objects is zero.

Type	Typical Range	Sample Declaration
bit	0, 1	bit turn = 1
bool	false, true	bool flag = true
byte	0..255	byte cnt
chan	1..255	chan q
mtype	1..255	mtype msg
pid	0..255	pid p
short	$-2^{15}..2^{15}-1$	short s = 100
int	$-2^{31}..2^{31}-1$	int x = 1
unsigned	$0..2^{n}-1$, $1 \leq n \leq 32$	unsigned w : 3 = 5

- Support array.
- unsigned w : 3 = 5 means w ranged from 0 to 7, and initially is 5.

Data Objects (cont.)

- There are only 2 levels of scope in PROMELA models:
  - global (visible in the entire model)
  - process local (visible only to the process that contains the declaration)

```
active proctype main(){
    int x;
    {
        int y;
        printf("x = %d,y = %d",x,y);   /* x=0 , y=0 */
        x++;
        y++;
    }
    printf("x = %d,y = %d",x,y);       /* x=1 , y=1 */
}
```

Data Objects (cont.)

- Enumerated Types is a set of symbolic constants:
  - mtype stands for message type.
  - There can be multiple mtype declarations but they are equivalent to a single mtype declaration that contains the concatenation of all separate lists of symbolic names.

```
mtype = { appel, pear, orange, banana };
mtype n = pear;
```

- User defined data type:

```
typedef record{
    short f1;
    byte f2 = 4
};
```

Message Channels

- Used to model the exchange of data between processes.
- They are declared either locally or globally, but the channel itself is always a global object.
- A locally declared and instantiated channel disappears, when the process that declare it dies.

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

- According to the capacity of channel, there are two types of channel:
  - *capacity* > 0: a FIFO buffered channel is initialized (asynchronous).
  - *capacity* $= 0$: a rendezvous channel is initialized (synchronous).

Message Passing

```
/*send message*/
qname ! expre1, expr2, expr3

/*receive message*/
qname ? var1, var2, var3
```

- Send a message to channel with corresponding values.
- Retrieves a message from the channel, and copies the values into corresponding variables.
- The message will be removed from the channel buffer (optional).
- It is an error to send or receive either more of fewer message fields than declared.

Message Passing (cont.)

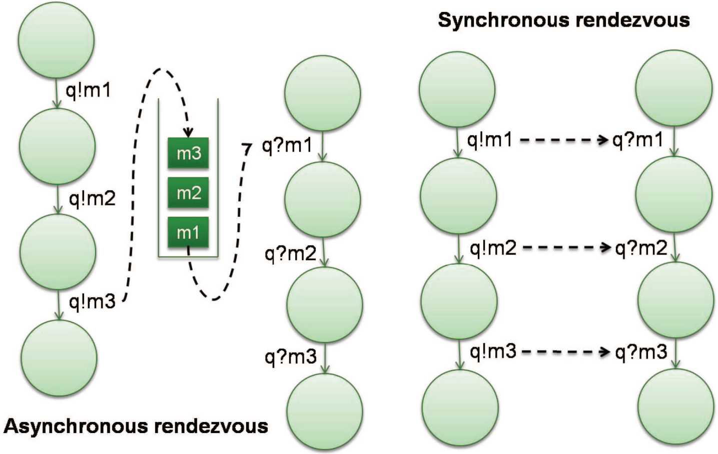
- A send statement on buffered channel is executable when the target channel is non-full.
- A send statement on rendezvous channel contains two steps:
  - a rendezvous offer: can be made at any time.
  - a rendezvous accept: can be accepted only if another process can perform the matching receive operation immediately (i.e., with no intervening steps by any process).
- A receive statement is executable if the first message in the channel match the pattern from the receive statement.
- A match of a message is obtained if all message fields that contain constant values in the receive statement equal the values of the corresponding message fields in the message.

Rendezvous Communication

- The size of the channel is set to zero.
- That is, the channel can pass, but cannot store messages.

```
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
}
```

Asynchronous and Synchronous Message Passing



Rules for executability

- Any statement in PROMELA is either executable or blocked.
- 6 types of basic PROMELA statements: assign, print, assert, expression, communication (send/receive)
  - A statement(expression) is executable iff evaluates to true or to a non-zero integer value.
  - A statement is blocked iff there is no executable statements left to execute.
  - Print and assignment are always executable.

```
/*   In c language we have to write like that:   */

while (a!=b) {}

 /*   But we can achieve the same effect in PROMELA by   */

(a==b);
```

Control Flow

- Atomic sequences, making things indivisible:
  - atomic{...}
  - d_step{...}
- Non-deterministic selection and iteration
  - if...fi
  - do...od
- Goto, break and labels
- Escape sequences:
  - {...} unless {...}

Atomic Sequences

- atomic { guard -> stmt_1; stmt_2; ...; stmt_n; }
  - Executable if the guard statement is executable.
  - Any statement can serve as the guard statement.
  - Executes all statements in the sequence without interleaving with other processes.
  - If any statement other than the guard blocks, atomicity is lost. Atomicity can be regained when the statement becomes executable.

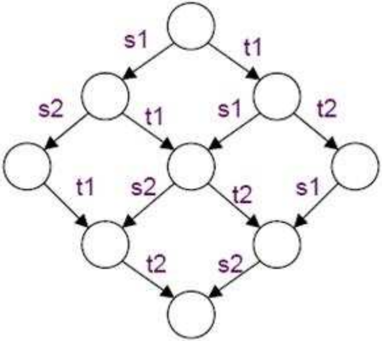
```
atomic{

    /* swap the values of a and b */
    tmp = b;
    b = a;
    a = tmp
}
```

D_step Sequences

- d_step { guard -> stmt_1; stmt_2; ...; stmt_n; }
  - Like atomic sequence, but must be deterministic and may not block anywhere inside the sequence.
  - It will be an error if any statement except the guard statement in a d_step sequence be unexecutable.
  - A Goto statement into or out of d_step sequences are forbidden.
  - Atomic and d_step sequences are often used as a model reduction method, to lower complexity of large models.

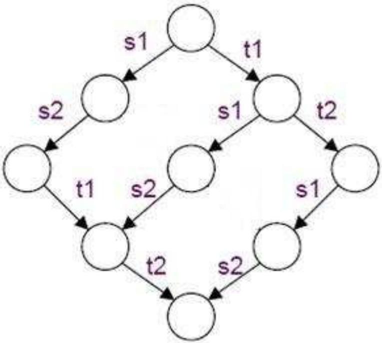
Atomic and D_step Sequences Example (1/3)

```
active proctype A() { s1; s2 }
active proctype B() { t1; t2 }
```



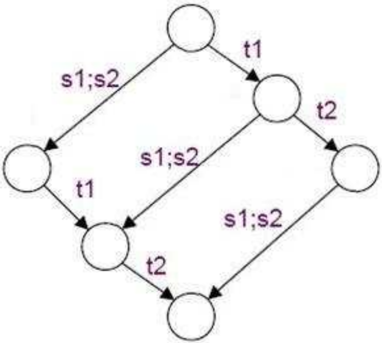
Atomic and D_step Sequences Example (2/3)

```
active proctype A() { atomic{ s1; s2 } }
active proctype B() { t1; t2 }
```



Atomic and D_step Sequences Example (3/3)

```
active proctype A() { d_step{ s1; s2 } }
active proctype B() { t1; t2 }
```



Selection

```
if
:: guard_1 -> stmt_1.1 ; stmt_1.2 ; ...
:: guard_2 -> stmt_2.1 ; stmt_2.2 ; ...
:: ...
:: guard_n -> stmt_n.1 ; stmt_n.2 ;...
fi
```

- The if statement is executable if at least one guard is executable.
- If more than one guard is executable, than selected non-deterministically.
- If none of the guard statements is executable, the if statement blocks until at least one of them can be selected.
- Any type of basic or compound statement can be used as a guard.

Repetition

```
do
:: guard_1 -> stmt_1.1 ; stmt_1.2 ;...
:: guard_2 -> stmt_2.1 ; stmt_2.2 ;...
:: ...
:: guard_n -> stmt_n.1 ; stmt_n.2 ;...
od
```

- The execution of the repetition structure is repeated.
- If there is none executable statement in the do-loop, the entire loop blocks.
- Any type of basic or compound statement can be used as a guard.
- Only a break or a goto can exit from a do-loop.

Timeout v.s. Else

- A special type of statement in selection and repetition is the else statement.
- An else statement become executable only if no other statement within same process, at the same control-flow point, is executable.
- Another similar global variable is timeout.
- Timeout becomes true iff there are no executable statements in all of currently running processes.

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

Label

- To exit the repetition we can use goto statement and labeling.
- Multiple labels may be used to label the same statement.

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

done: printf("answer: %d\n", x)
}
```

Unless Statement

- S unless E
  - S and E is any PROMELA fragments.
  - The statement of S has a lower execution priority than the statement of E.
  - The executability of S is constraint to the non-executability of guard statements in E.
  - If E ever becomes enabled during the execution of S, then S is aborted and the execution continues with E.

```
do
:: b1 -> B1
:: b2 -> B2
od unless { c -> C };
```

Correctness Claims

- Two types of correctness requirements:
  - Safety: the set of properties that the system may not violate.
  - Liveness: the set of properties that the system must satisfy.
- Correctness properties can be specified as system or process invariants (using assertions), as linear temporal logic requirements (LTL), as formal Bu"chi Automata in the syntax of never claims.

Correctness Claims (cont.)

- Correctness properties in PROMELA are formalized with following constructs:
  - Basic assertions
  - End-state labels
  - Progress-state labels
  - Never claims

Basic assertions

```
assert ( expression )
```

- Is always executable.
- If the expression evaluates to true, then has no effect.
- If the expression evaluates to false, an error message will be trigger during verifications with SPIN.
- An assertions statement is the only type of correctness property in PROMELA that can be checked during simulation runs with SPIN.

Basic assertions (cont.)

- If SPIN fails to find an assertion violation in simulation runs, this does not mean that assertions cannot be violated,
- Only a verification run with SPIN can assure that assertion wont be violated.
- The assertion statement can be used to check safety properties.
- An assertion statement can be use as a system invariant.
  - Because it is in an asynchronous process, this statement may be executed at any time.

End-state labels

- The verifier must be able to distinguish valid system end states from invalid ones (deadlock).
- By default, the only valid end states are the end of its code (the closing curly brace).
- But not all PROMELA processes are meant to reach the end of the code.
- We can use end-state label to tell the verifier that these states are also valid.
- There can be any number of end-state labels, but in the same process, they have to have unique identifer (by prefix with end).

End-state labels

```
mtype {p,v};

chan sema = [0] of {mtype};

active proctype Dijkstra(){

    byte count = 1;

end: do
     :: (count == 1) ->
            sema ! p ; count = 0
     :: (count == 0) ->
            sema ? v ; count = 1
     od
}

active [3] proctype user() {
    do
    :: sema ? p;  /*enter*/
       skip;      /*leave*/
       sema ! v;
    od
}
```

Progress-state labels

- Checking whether a statement is idling or waiting for other process to make progress.
- A progress label states that at least one of the labeled states must be visited infinitely often in any infinite system execution.
- Any violation of this requirement can be reported by verifier as a non-progress cycle.
- The progress-state label can be used to check liveness properties.

```
active proctype Dijkstra(){      /* modify the last slide's example Dijkstra() */
                                 /* no non-progress cycles are found  */
    byte count = 1;

  end:  do
        :: (count == 1) ->
progress:   sema ! p ; count = 0
        :: (count == 0) ->
            sema ? v ; count = 1
        od
}
```

Progress-state labels (cont.)

- Below is a case where there is a non-progress cycle:

```
byte x = 2;

active proctype A()
{
    do
    ::x = 3 - x; progress: skip
    od
}

active proctype B()
{
    do
    ::x = 3 - x
    od
}
```

Never Claims

- A never claim gives us the capability to check properties just before and just after each statement execution
- Originally, a never claim was meant to match behavior that should never occur.
- That is, the verifier will flag it as an error if the full behavior specified in the claim be matched by any feasible system execution.

```
never{       /*  if p becomes false, an error occured  */
    do
    :: !p -> break
    :: else
    od
}
```

Never Claims (cont.)

- Never claim can either be written by hands or generated mechanically from LTL formula (SPIN has built-in translator).
- To translate an LTL formulae into a never claim, we have to consider the property:
  - Positive property (good behavior): we have to negate it at first.
  - Negative property (bad behavior): just translate it.
- For example, we want to check the positive property [] p (SPIN LTL syntax):

```
    never {          /*  ![]p = <>!p  */
         do
         :: true
         :: !p -> break
         od
}
```

SPIN's LTL Syntax

```
f ::=  p
     | true
     | false
     | ( f )
     | f binop f
     | unop f

uniop ::= []      (always)
        | <>      (eventually)
        | !       (logical negation)

binop ::= U       (until)
        | &&      (logical and)
        | ||      (logical or)
        | ->      (implication)
        | <->     (equivalence)
```

Specifying LTL properties

- LTL Formulae examples:

Formula	Pronounced	Type/Template
[] p	always p	invariance
<> p	eventually p	guarantee
p -> (<> q)	p implies eventually q	response
p -> (q U r)	p implies q until r	precedence
[] <> p	always, eventually p	recurrence (progress)
<> [] p	eventually, always p	stability (non-progress)
(<> p) -> (<> q)	eventually p implies eventually q	correlation

PROMELA Semantics

- SPIN translates each process into a finite automaton.
- The global behavior of the concurrent system is obtained by computing an asynchronous interleaving product of automata, one automaton per asynchronous process behavior.
- The resulting global system behavior is itself again represented by an automaton.
- This interleaving product is often referred to as the state space of the system, and, because it can easily be represented as a graph, it is also commonly referred to as the global reachability graph.

PROMELA Semantics (cont.)

- By simulating the execution of a SPIN model we can generate a reachability graph.
- The PROMELA semantics rules define how the global reachability graph for any given PROMELA model is to be generated.
- Basic correctness claims in PROMELA can be interpreted as the presence or absence of specific types of nodes or edges.
- LTL properties can be interpreted as the presence or absence of specific types of sub-graph, or paths.

Transition Relation

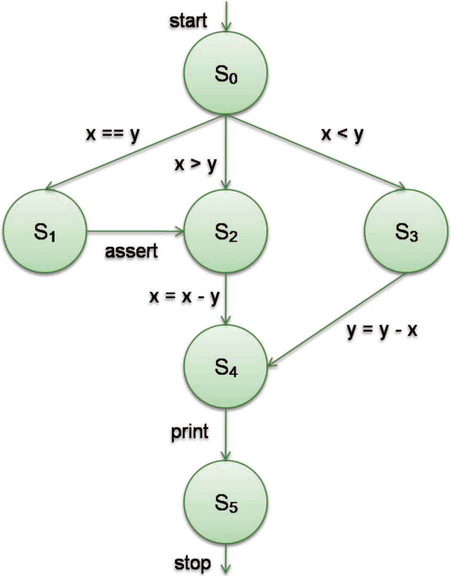
- Every PROMELA proctype defines a finite state automaton, (S, s_0, L, T, F)

Symbol	Finite State Automaton	PROMELA Model
S	Set of states	Possible points of control within the proctype
L	Transition label set	Specific basic statement (six basic types)
T	Transition relation	Flow of control
F	Set of final states	End-state

Proctype and Automata(1/2)

```
active proctype not_euclid(int x , y)
{
    if
    ::  (x > y) -> L: x = x - y
    ::  (x < y) -> y = y -x
    ::  (x == y) -> assert (x != y); goto L
    fi;
    printf(``%d\n'', x)
}
```

Proctype and Automata(2/2)



Operational Model(1/8)

- To define the semantics of the modeling language, we can define an operational model in terms of states and state transitions.
  - We have to define what a "state" is.
  - We have to define what a "transition" is.
    - i.e., how the 'next-state' relation is defined.
- Global system states are defined in terms of a small number of primitive objects:
  - We have to define: variables, messages, message channels, and processes.

Operational Model(2/8)

- State transitions require the definition of 3 things:
  - transition executability rules
  - transition selection rules
  - the effect of transition
- We only have to define one-step semantics to define the full language.
- The 3 parts of the semantics definition are defined over 4 types of objects:
  - variables, messages, channels, processes
- Well define these first.

Operational Model(3/8)

variables, messages, channels, processes, transitions, global states

- A PROMELA variable is defined by a five-tuple
  { name, scope, domain, inival, curval }

```
short x=2, y=1; /* global */
active proctype not_euclid(){
    S:  if  /* curval of x at S: 2 */
        :: x > y -> L: x = x - y
        :: x < y -> y = y - x
        :: x == y -> assert(x != y); goto L
        fi;
    E:  printf(``%d\n'', x) /* curval of x at E: 1 */
}
```

- note: domain is a finite set of integers.

Operational Model(3/8)

variables, messages, channels, processes, transitions, global states

- A PROMELA variable is defined by a five-tuple
  { name, scope, domain, inival, curval }

```
short x=2, y=1; /* global */
active proctype not_euclid(){
    S:  if  /* curval of x at S: 2 */
        :: x > y -> L: x = x - y
        :: x < y -> y = y - x
        :: x == y -> assert(x != y); goto L
        fi;
    E:  printf(‘‘%d\n’’, x) /* curval of x at E: 1 */
}
```

- note: domain is a finite set of integers.

Operational Model(3/8)

variables, messages, channels, processes, transitions, global states

- A PROMELA variable is defined by a five-tuple { name, scope, domain, inival, curval }

```
short x=2, y=1; /* global */
active proctype not_euclid(){
    S:  if  /* curval of x at S: 2 */
        :: x > y -> L: x = x - y
        :: x < y -> y = y - x
        :: x == y -> assert(x != y); goto L
        fi;
    E:  printf(``%d\n'', x) /* curval of x at E: 1 */
}
```

- note: domain is a finite set of integers.

Operational Model(3/8)

variables, messages, channels, processes, transitions, global states

- A PROMELA variable is defined by a five-tuple
  { name, scope, domain, inival, curval }

```
short x=2, y=1; /* global */
active proctype not_euclid(){
    S:  if  /* curval of x at S: 2 */
        :: x > y -> L: x = x - y
        :: x < y -> y = y - x
        :: x == y -> assert(x != y); goto L
        fi;
    E:  printf(``%d\n'', x) /* curval of x at E: 1 */
}
```

- note: domain is a finite set of integers.

Operational Model(3/8)

variables, messages, channels, processes, transitions, global states

- A PROMELA variable is defined by a five-tuple { name, scope, domain, inival, curval }

```
short x=2, y=1; /* global */
active proctype not_euclid(){
    S:  if  /* curval of x at S: 2 */
        :: x > y -> L: x = x - y
        :: x < y -> y = y - x
        :: x == y -> assert(x != y); goto L
        fi;
    E:  printf(``%d\n'', x) /* curval of x at E: 1 */
}
```

- note: domain is a finite set of integers.

Operational Model(4/8)

variables, messages, channels, processes, transitions, global states

- A message is a finite, ordered set of variables
  (Messages are stored in channels - defined next.)

Operational Model(5/8)

variables, messages, channels, processes, transitions, global states

- A message channel is defined by a 3-tuple
  { ch_id, nslots, contents }

```
chan q = [2] of { mtype, bit };
```

  - Channels always have global scope.
  - A ch_id is a positive integer uniquely identifies the channel.
  - An ordered set of messages with maximally nslots elements:
    { {slot1.field1 ,slot1.field2 }, {slot2.field1 ,slot2.field2 } }

Operational Model(5/8)

variables, messages, channels, processes, transitions, global states

- A message channel is defined by a 3-tuple
  { ch_id, nslots, contents }

```
chan q = [2] of { mtype, bit };
```

  - Channels always have global scope.
  - A ch_id is a positive integer uniquely identifies the channel.
  - An ordered set of messages with maximally nslots elements:
    { {slot1.field1 ,slot1.field2 }, {slot2.field1 ,slot2.field2 } }

Operational Model(5/8)

variables, messages, channels, processes, transitions, global states

- A message channel is defined by a 3-tuple
  { ch_id, nslots, contents }

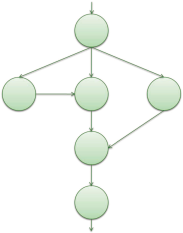
```
chan q = [2] of { mtype, bit };
```

  - Channels always have global scope.
  - A ch_id is a positive integer uniquely identifies the channel.
  - An ordered set of messages with maximally nslots elements:
    { {slot1.field1 ,slot1.field2 }, {slot2.field1 ,slot2.field2 } }

Operational Model(6/8)

variables, messages, channels, processes, transitions, global states

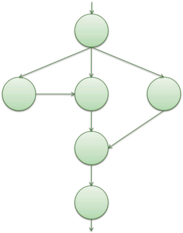
- A process is defined by a six-tuple
  { pid, lvars, lstates, inistate, curstate, transitions }
  - process instantiation number
  - finite set of local variables
  - a finite set of integers defining local proc states
  - the initial state
  - the current state
  - a finite set of transitions (to be defined) between elements of lstates



Operational Model(6/8)

variables, messages, channels, processes, transitions, global states

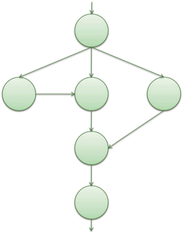
- A process is defined by a six-tuple
  { pid, lvars, lstates, inistate, curstate, transitions }
  - process instantiation number
  - finite set of local variables
  - a finite set of integers defining local proc states
  - the initial state
  - the current state
  - a finite set of transitions (to be defined) between elements of lstates



Operational Model(6/8)

variables, messages, channels, processes, transitions, global states

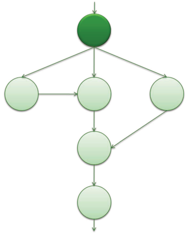
- A process is defined by a six-tuple
  { pid, lvars, lstates, inistate, curstate, transitions }
  - process instantiation number
  - finite set of local variables
  - a finite set of integers defining local proc states
  - the initial state
  - the current state
  - a finite set of transitions (to be defined) between elements of lstates



Operational Model(6/8)

variables, messages, channels, processes, transitions, global states

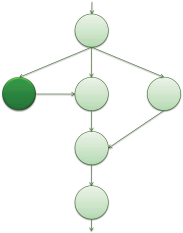
- A process is defined by a six-tuple
  { pid, lvars, lstates, inistate, curstate, transitions }
  - process instantiation number
  - finite set of local variables
  - a finite set of integers defining local proc states
  - the initial state
  - the current state
  - a finite set of transitions (to be defined) between elements of lstates



Operational Model(6/8)

variables, messages, channels, processes, transitions, global states

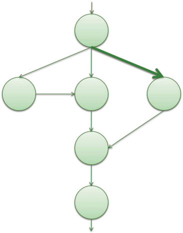
- A process is defined by a six-tuple
  { pid, lvars, lstates, inistate, curstate, transitions }
  - process instantiation number
  - finite set of local variables
  - a finite set of integers defining local proc states
  - the initial state
  - the current state
  - a finite set of transitions (to be defined) between elements of lstates



Operational Model(6/8)

variables, messages, channels, processes, transitions, global states

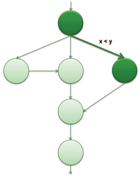
- A process is defined by a six-tuple
  { pid, lvars, lstates, inistate, curstate, transitions }
  - process instantiation number
  - finite set of local variables
  - a finite set of integers defining local proc states
  - the initial state
  - the current state
  - a finite set of transitions (to be defined) between elements of lstates



Operational Model(7/8)

variables, messages, channels, processes, transitions, global states

- A transition in process P is defined by a seven-tuple
  { tr_id, source-state, target-state, cond, effect, priority, rv }

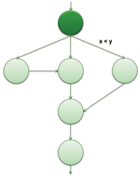


  - source-state and target-state are elements from set P.lstates
  - Condition and effect are defined for each basic statement, and they are typically defined on variable and channel values, possibly also on process states.
  - Predefined system variables that are used to define the semantics of unless and rendezvous.

Operational Model(7/8)

variables, messages, channels, processes, transitions, global states

- A transition in process P is defined by a seven-tuple
  { tr_id, source-state, target-state, cond, effect, priority, rv }

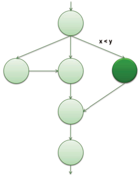


- source-state and target-state are elements from set P.lstates
- Condition and effect are defined for each basic statement, and they are typically defined on variable and channel values, possibly also on process states.
- Predefined system variables that are used to define the semantics of unless and rendezvous.

Operational Model(7/8)

variables, messages, channels, processes, transitions, global states

- A transition in process P is defined by a seven-tuple
  { tr_id, source-state, target-state, cond, effect, priority, rv }

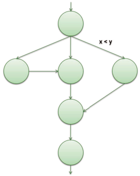


  - source-state and target-state are elements from set P.lstates
  - Condition and effect are defined for each basic statement, and they are typically defined on variable and channel values, possibly also on process states.
  - Predefined system variables that are used to define the semantics of unless and rendezvous.

Operational Model(7/8)

variables, messages, channels, processes, transitions, global states

- A transition in process P is defined by a seven-tuple
  { tr_id, source-state, target-state, cond, effect, priority, rv }

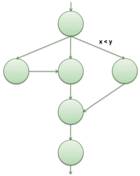


- source-state and target-state are elements from set P.lstates
- Condition and effect are defined for each basic statement, and they are typically defined on variable and channel values, possibly also on process states.
- Predefined system variables that are used to define the semantics of unless and rendezvous.

Operational Model(7/8)

variables, messages, channels, processes, transitions, global states

- A transition in process P is defined by a seven-tuple
  { tr_id, source-state, target-state, cond, effect, priority, rv }



  - source-state and target-state are elements from set P.lstates
  - Condition and effect are defined for each basic statement, and they are typically defined on variable and channel values, possibly also on process states.
  - Predefined system variables that are used to define the semantics of unless and rendezvous.

Operational Model(8/8)

variables, messages, channels, processes, transitions, global states

- A global system state is defined by a eight-tuple
  { gvars, procs, chans, exclusive, handshake, timeout, else, stutter }
  - a finite set of global variables
  - a finite set of processes
  - a finite set of message channels
  - predefined integer system variables that are used to define the semantics of atomic, d_step
  - predefined integer system variables that are used to define the semantics of rendezvous
  - predefined Boolean system variables
  - for stutter extension rule

Operational Model(8/8)

variables, messages, channels, processes, transitions, global states

- A global system state is defined by a eight-tuple { gvars, procs, chans, exclusive, handshake, timeout, else, stutter }
  - a finite set of global variables
  - a finite set of processes
  - a finite set of message channels
  - predefined integer system variables that are used to define the semantics of atomic, d_step
  - predefined integer system variables that are used to define the semantics of rendezvous
  - predefined Boolean system variables
  - for stutter extension rule

Operational Model(8/8)

variables, messages, channels, processes, transitions, global states

- A global system state is defined by a eight-tuple
  { gvars, procs, chans, exclusive, handshake, timeout, else, stutter }
  - a finite set of global variables
  - a finite set of processes
  - a finite set of message channels
  - predefined integer system variables that are used to define the semantics of atomic, d_step
  - predefined integer system variables that are used to define the semantics of rendezvous
  - predefined Boolean system variables
  - for stutter extension rule

Operational Model(8/8)

variables, messages, channels, processes, transitions, global states

- A global system state is defined by a eight-tuple
  { gvars, procs, chans, exclusive, handshake, timeout, else, stutter }
  - a finite set of global variables
  - a finite set of processes
  - a finite set of message channels
  - predefined integer system variables that are used to define the semantics of atomic, d_step
  - predefined integer system variables that are used to define the semantics of rendezvous
  - predefined Boolean system variables
  - for stutter extension rule

Operational Model(8/8)

- A global system state is defined by a eight-tuple
  { gvars, procs, chans, exclusive, handshake, timeout, else, stutter }
  - a finite set of global variables
  - a finite set of processes
  - a finite set of message channels
  - predefined integer system variables that are used to define the semantics of atomic, d_step
  - predefined integer system variables that are used to define the semantics of rendezvous
  - predefined Boolean system variables
  - for stutter extension rule

Operational Model(8/8)

variables, messages, channels, processes, transitions, global states

- A global system state is defined by a eight-tuple { gvars, procs, chans, exclusive, handshake, timeout, else, stutter }
  - a finite set of global variables
  - a finite set of processes
  - a finite set of message channels
  - predefined integer system variables that are used to define the semantics of atomic, d_step
  - predefined integer system variables that are used to define the semantics of rendezvous
  - predefined Boolean system variables
  - for stutter extension rule

Operational Model(8/8)

variables, messages, channels, processes, transitions, global states

- A global system state is defined by a eight-tuple
  { gvars, procs, chans, exclusive, handshake, timeout, else, stutter }
  - a finite set of global variables
  - a finite set of processes
  - a finite set of message channels
  - predefined integer system variables that are used to define the semantics of atomic, d_step
  - predefined integer system variables that are used to define the semantics of rendezvous
  - predefined Boolean system variables
  - for stutter extension rule

Stutter extension

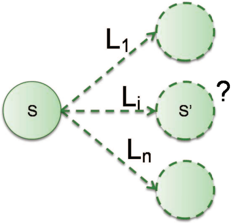
- The reason why we have to use stutter extension is because PROMELA model is finite.
- When we use LTL as a correctness claim, the LTL formula will be translated into Buchi automaton.
- In Buchi automaton acceptance condition, there will be an infinite cycle pass at least one of the element of accept sets.
- If we want to do the interleaving product of the Buchi automaton with PROMELA model, we have to deal with the infinite execution.
- In stutter extension, we make the final state have a transition target to itself, with label ε.

One-Step Semantics(1/2)

- Given an arbitrary global state of the system, determine the set of possible immediate successor states.
  - To define a one-step semantics, we have to define 3 more things:
    - transition executability rules, transition selection rules, the effect of transition

One-Step Semantics(2/2)

- We do so by defining an algorithm: an implementation-independent "semantics engine" for Spin.
  - The semantics engine executes the model in a stepwise manner: selection and executing one basic statement at a time
  - At the highest level of abstraction, the behavior of this engine is defined as follows:



$L_1, \ldots, L_i, \ldots, L_n$

- **assignment statement**
- **assertion statement**
- **expression statement**
- **print statement**
- **send statement**
- **receive statement**

PROMELA Semantics Engine

```
global states s, s'
processes p, p'
transitions t, t'
//E is a set of pairs (p,t)

while ((E = executable(s)) != {}){
    for some (p, t) from E{
        s' = apply(t.effect, s)

           s = s'
             p.curstate = t.target













  }
}

while (stutter){
   s = s    /* `stutter' extension*/
}
```

Executability Rules(1/5)

```
global states s, s'
processes p, p'
transitions t, t'

Set
executable (State s){
    new Set E
    new Set e



    AllProcs:
    ...












    return E    /* executable transitions */
}
```

next: extenstion for timeout, else, rendezvous, atomic, unless

Executability Rules(1/5)

```
global states s, s'
processes p, p'
transitions t, t'

Set
executable (State s){
    new Set E
    new Set e

    E = {}
    timeout = false
    AllProcs:
    ...

    if (E == {} and timeout == false){
        timeout == true
        goto AllProcs
    }

    return E    /* executable transitions */
}
```

next: extenstion for else

Executability Rules(2/5)

```
AllProcs:
for each active process p{



            e = {};


            OneProc:
               for each transition t in p.trans{
                   if (t.source == p.curstate
                     and eval(t.cond == true)){
                       add (p, t) to set e
                   }
               }


                   add all elements of e to E








}
```

Executability Rules(2/5)

```
AllProcs:
for each active process p{
            e = {};
            else = false
            OneProc:
               for each transition t in p.trans{
                   if (t.source == p.curstate
                     and eval(t.cond == true)){
                       add (p, t) to set e
                   }
               }
               if (e != {}){
                   add all elements of e to E
                   break    /* on to next process */
               } else if (else == false){
                   else = true
                   goto OneProc
               }
}
```

next: extension for extension for rendezvous

Adding Semantics for Rendezvous

```
global states s, s'
processes p, p'
transitions t, t'
//E is a set of pairs (p,t)

while ((E = executable(s)) != {}){
    for some (p, t) from E{
        s' = apply(t.effect, s)

            s = s'
            p.curstate = t.target










    }
}

while (stutter){
    s = s    /* stutter extension */
}
```

effect of issuing a rendezvous offer is to set handshake to channel's identity

Adding Semantics for Rendezvous

```
global states s, s'
processes p, p'
transitions t, t'
//E is a set of pairs (p,t)

while ((E = executable(s)) != {}){
    for some (p, t) from E{
        s' = apply(t.effect, s)
        if (handshake == 0){
            s = s'
            p.curstate = t.target
        } else{
                /* try to complete rv handshake */








                handshake = 0
        }
    }
}

while (stutter){
    s = s    /* stutter extension */
}
```

effect of issuing a rendezvous offer is to set handshake to channel's identity

Executability Rules(3/5)

```
AllProcs:
for each active process p{
            e = {};
            else = false
            OneProc:
               for each transition t in p.trans{
                   if (t.source == p.curstate
                     and eval(t.cond == true)){
                       add (p, t) to set e
                   }
               }
               if (e != {}){
                   add all elements of e to E
                   break    /* on to next process */
               } else if (else == false){
                   else = true
                   goto OneProc
               }
}
```

Executability Rules(3/5)

```
AllProcs:
for each active process p{



            e = {};
            else = false

            OneProc:
               for each transition t in p.trans{
                   if (t.source == p.curstate                  and (handshake == 0 or handshake == t.rv)
                     and eval(t.cond == true)){
                       add (p, t) to set e
                   }
               }

               if (e != {}){
                   add all elements of e to E
                   break    /* on to next process */
               } else if (else == false){
                   else = true
                   goto OneProc
               }


}
```

next: extenstion for atomic

Executability Rules(3/5)

```
AllProcs:
for each active process p{
    if (exclusive == 0 or exclusive == p.pid){


            e = {};
            else = false

            OneProc:
               for each transition t in p.trans{
                   if (t.source == p.curstate                 and (handshake == 0 or handshake == t.rv)
                     and eval(t.cond == true)){
                       add (p, t) to set e
                   }
               }

               if (e != {}){
                   add all elements of e to E
                   break    /* on to next process */
               } else if (else == false){
                   else = true
                   goto OneProc
               }

    }
}
```

Executability Rules(4/5)

```
global states s, s’
processes p, p’
transitions t, t’

Set
executable (State s){
    new Set E
    new Set e

    E = {}
    timeout = false
    AllProcs:
    ...







    if (E == {} and timeout == false){
        timeout == true
        goto AllProcs
    }

    return E /* executable transition */
}
```

Executability Rules(4/5)

```
global states s, s'
processes p, p'
transitions t, t'

Set
executable (State s){
    new Set E
    new Set e

    E = {}
    timeout = false
    AllProcs:
    ...


    if (E == {} and exclusive != 0){
        exclusive = 0
        goto AllProcs
    }

    if (E == {} and timeout == false){
        timeout == true
        goto AllProcs
    }

    return E /* executable transition */
}
```

Executability Rules(4/5)

```
global states s, s'
processes p, p'
transitions t, t'

Set
executable (State s){
    new Set E
    new Set e

    E = {}
    timeout = false
    AllProcs:
    ...


    if (E == {} and exclusive != 0){
        exclusive = 0
        goto AllProcs
    }

    if (E == {} and timeout == false){
        timeout == true
        goto AllProcs
    }

    return E /* executable transition */
}
```

next: extenstion for unless (priorities)

Executability Rules(5/5)

```
AllProcs:
for each active process p{
    if (exclusive == 0 or exclusive == p.pid){


            e = {};
            else = false

            OneProc:
               for each transition t in p.trans{
                   if (t.source == p.curstate                 and (handshake == 0 or handshake == t.rv)
                     and eval(t.cond == true)){
                       add (p, t) to set e
                   }
               }

               if (e != {}){
                   add all elements of e to E
                   break    /* on to next process */
               } else if (else == false){
                   else = true
                   goto OneProc
               }

    }
}
```

Executability Rules(5/5)

```
AllProcs:
for each active process p{
    if (exclusive == 0 or exclusive == p.pid){
        /* priority */
        for u from high to low{
            e = {};
            else = false

            OneProc:
               for each transition t in p.trans{
                   if (t.source == p.curstate and t.prty == u and (handshake == 0 or handshake == t.rv)
                     and eval(t.cond == true)){
                       add (p, t) to set e
                   }
               }

               if (e != {}){
                   add all elements of e to E
                   break    /* on to next process */
               } else if (else == false){
                   else = true
                   goto OneProc
               }  /* or else lower the priority */
       }
    }
}
```

PROMELA Semantics Engine

```
global states s, s'
processes p, p'
transitions t, t'
//E is a set of pairs (p,t)

while ((E = executable(s)) != {}){
    for some (p, t) from E{
        s' = apply(t.effect, s)
        if (handshake == 0){
            s = s'
            p.curstate = t.target
        } else{
            /* try to complete rv handshake */
            E' = executable(s')
            /* if E' is {}, s is unchanged */

            for some (p', t') from E'{
                s = apply(t'.effect, s')
                p.curstate = t.target
                p'.curstate = t'.target
            }
            handshake = 0
        }
    }
}

while (stutter){
    s = s    /* stutter extension */
}
```

Interpreting PROMELA models

- The semantic engine
  - manipulate the basic objects of a PROMELA model.
  - does not have to know anything about control-flow constructs.
    - e.g., if, do, break, and goto
  - merely deals with local states and transitions.
- Three examples

PROMELA Models(1/2)

```
chan x = [0] of {bit};
chan y = [0] of {bit};
active proctype A() {x?0 unless y!0}
active proctype B() {y?0 unless x!0}
```

```
chan x = [0] of {bit};
chan y = [0] of {bit};
active proctype A() {x!0 unless y!0}
active proctype B() {y?0 unless x?0}
```

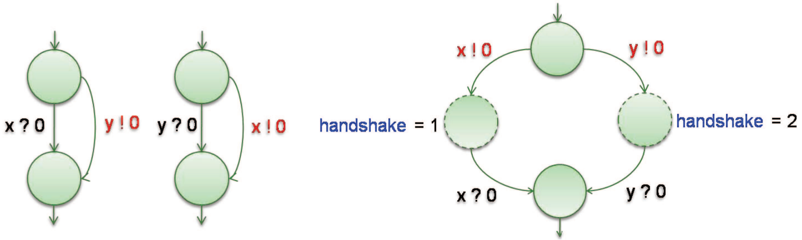
```
chan x = [0] of {bit};
chan y = [0] of {bit};
active proctype A() {x!0 unless y?0}
active proctype B() {y!0 unless x?0}
```

PROMELA Models(2/2)

- Rendezvous handshakes occur in two parts:
  - Sender offers
  - Receiver accepts

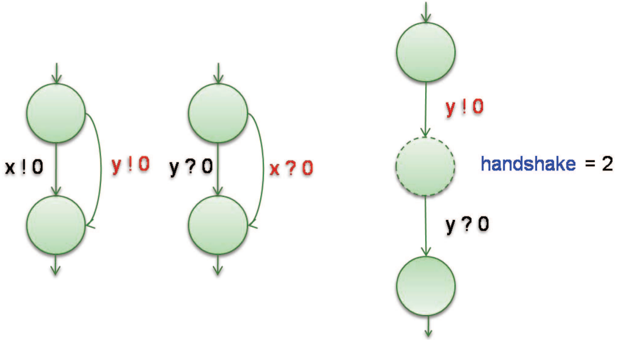
Example 1:3

```
chan x = [0] of {bit};
chan y = [0] of {bit};
active proctype A() {x?0 unless y!0}
active proctype B() {y?0 unless x!0}
```



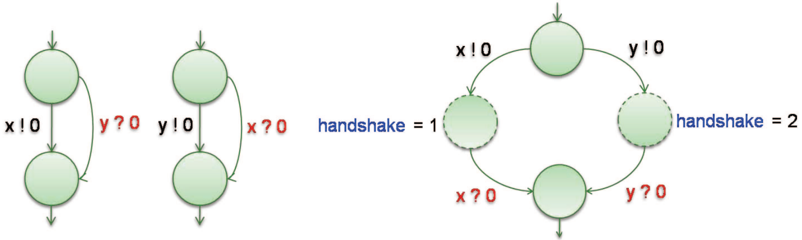
Example 2:3

```
chan x = [0] of {bit};
chan y = [0] of {bit};
active proctype A() {x!0 unless y!0}
active proctype B() {y?0 unless x?0}
```



Example 3:3

```
chan x = [0] of {bit};
chan y = [0] of {bit};
active proctype A() {x!0 unless y?0}
active proctype B() {y!0 unless x?0}
```



Agenda

- An Introduction to SPIN
- An Overview of PROMELA
- Verification in SPIN
- DEMO
- References

Verification in SPIN

- The goal of system verification is to establish what is possible and what is not.
- When performing verification we are interested in whether design requirements could be violated, not how likely or unlikely such violations might be.
- To perform verification, SPIN takes a correctness claim that is specified as a LTL, converts that formula into a Buchi automaton, and computes the synchronous product of this claim and the automaton representing the global state space.
- The result is again a Buchi automaton.
- If the language accepted by this automaton is empty, this means that the original claim is not satisfied for the given system.
- If the language is nonempty, it contains precisely those behaviors that satisfy the original temporal logic formula.

Agenda

- An Introduction to SPIN
- An Overview of PROMELA
- Verification in SPIN
- DEMO
- References

DEMO

- You can use the SPIN model checker in three types:
  - Using Command Line
  - Using XSPIN
  - Using JSPIN

DEMO

- Mutual_Exclusion_1.pml
  - This example is a software solution to the mutual exclusion problem proposed by Hyman.
  - Find a counterexample to demonstrate that this solution is incorrect.
- Mutual_Exclusion_2.pml (using assertion)
- Mutual_Exclusion_3.pml (using a monitor as invariant)
- Mutual_Exclusion_4.pml (using LTL property)
- Peterson_Mutual_Exclusion.pml (using LTL property)

Agenda

- An Introduction to SPIN
- An Overview of PROMELA
- Verification in SPIN
- DEMO
- References

References

- G.J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, Addison-Wesley, 2003
- G.J. Holzmann, *The Model Checker SPIN*, IEEE Trans. Software Eng., vol. 23, no. 5, May 1997.
- SPIN Official website