The SPIN Model Checker

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

Sheng-Feng Yu

Dept. of Information Management
National Taiwan University

May 15, 2009

Agenda

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

Agenda

- An Introduction to SPIN
  - History of SPIN
  - What is SPIN
    - 3 Types of Objects
  - (X)SPIN Architecture
  - DEMO in Command Line
    - Hello_World.pml
    - Generic.pml
- An Overview of PROMELA
- Verification in SPIN
- DEMO with XSPIN
- References

History of SPIN

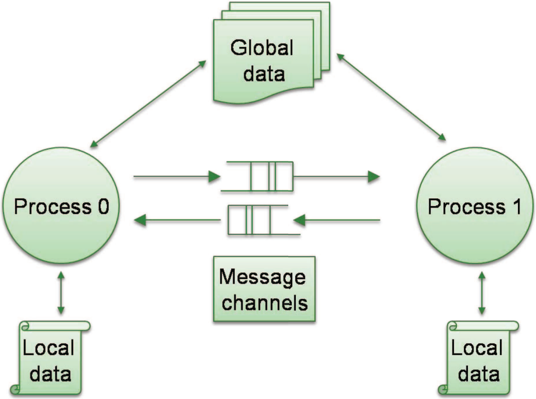
- The tool was developed at Bell Labs in the original Unix group of the Computing Sciences Research Center, starting in 1980 by Gerard Holzmann and others.
- The software has been available freely since 1991, and continues to evolve to keep pace with new developments in the field.
- In April 2002 the tool was awarded the prestigious System Software Award for 2001 by the ACM.
- Since 1995, (approximately) annual SPIN workshops have been held for SPIN users, researchers, and those generally interested in model checking.

What is SPIN

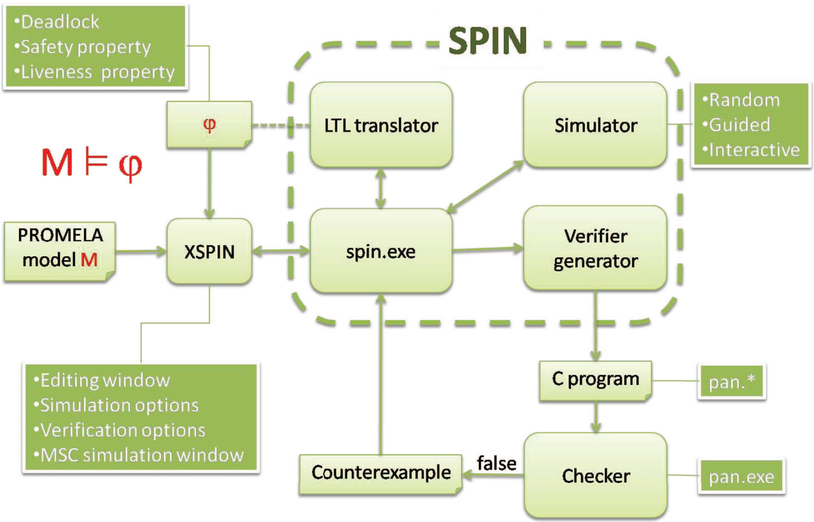
- SPIN (Simple PROMELA INterpreter)
  - Is a tool for analyzing the logical consistency of concurrent systems, specifically of data communication protocols.
  - Can check that the behavior specification (the system design) is logically consistent with the requirements specification (the desired properties of the design).
  - The system is described in a modeling language called PROMELA (PROcess MEta LAnguage).

3 Types of Objects

- Processes
- Global and local data objects
- Message channels



(X)SPIN Architecture



DEMO in Command Line

- Hello_World.pml
- Generic.pml

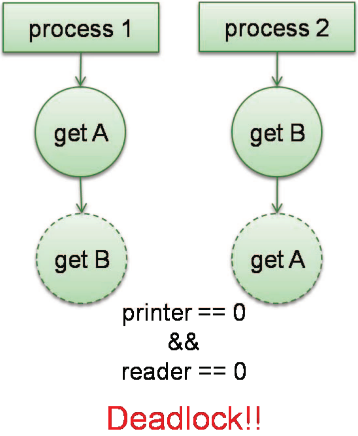
Hello_World.pml

- Simulation run
  - spin Hello_World.pml
- Verification run
  - spin -a Hello_World.pml
  - gcc -o pan pan.c
  - ./pan
  - -a produces a model checker pan.*

Generic.pml

- Simulation run
  - spin -v -u20 Generic.pml
- Verification run
  - spin -a Generic.pml
  - gcc -DBFS -o pan pan.c
  - ./pan
  - -DBFS use a breadth-first-search algorithm to find a short error path.
- Inspection of the error trail
  - spin -t -v Generic.pml
  - -t performs a guided simulation.
  - -v is verbose mode, adds some more detail, and generates more hints and warnings about the model.
  - Invalid end state is euphemism for a deadlock.

Deadlock Diagram



Agenda

- An Introduction to SPIN
- An Overview of PROMELA
  - What is PROMELA
  - PROMELA Model
    - Variable
    - Data type
    - Process
    - Message channel
    - Statement
  - PROMELA Semantic
- Verification in SPIN
- DEMO with XSPIN

What is PROMELA

- PROMELA (PROcess MEta LAnguage)
  - resembles the programming language C.
  - is a specification language to describe finite-state distributed systems.
    - Enforcing that restriction helps to guarantee that any correctness property that can be stated in PROMELA is decidable.
- PROMELA models are always finite-state:
  - 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.

PROMELA Model

- A PROMELA model consist of:
  - Global variable declarations
    - Can be access by all processes
  - Type declarations
    - mtype, typedef, constants
  - Process declarations
    - Behavior of the processes: local variables + statements
  - Channel declarations
    - chan ch = [dim] of {type, ...}
    - Asynchronous: $0 < \text{dim}$
    - Rendezvous: $\text{dim} == 0$
  - [init process]
    - Initializes variables and starts processes

Variables

- There are only 2 levels of scope:
  - global variable (visible in the entire system)
  - local variable (visible only to the process that contains the declaration)
- Predefined variables in PROMELA.
  - _pid
    - current process's instantiation number
  - _nr_pr
    - the number of active processes
  - timeout
    - true if no statement in the system is executable
  - else
    - true if no condition statement in the current process is executable

Data Type(1/2)

- The default initial value of all data objects (global and local) 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	$0..2^{n}-1,\ 0 \leq n \leq 32$	unsigned w : 3 = 5

Data Type(2/2)

- Enumerated Types is a set of symbolic constants:
  - mtype = {apple, banana, cherry}
  - Note: A process can only contain one mtype declaration which must be global.
- User defined data type

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

record rr;
rr.f1 = 5
```

Process

- Executes concurrently with all other processes
- Is defined by proctype declaration
- Has its **program counter** and **local variables**
- Communicates with other processes using channels or global variables
- Can be instantiated in two ways:
  - Adding the prefix active to a proctype declaration
  - Using a run operator

Example:proctype eager

```
active [2] proctype eager(){
    run eager();
    run eager()
}
```

- Note: The maximum number of processes is 255.

Process Synchronization with Provided Clauses

- A process can only execute statements if its provided clause evaluates to true.

toggle.pml

```
bool toggle = true;    /* global variable */
int cnt;    /* default initial value 0 */

active proctype A() provided (toggle == true){
    L:  cnt++;    /* increment cnt by 1 */
     printf("A: cnt=%d\n", cnt);
     toggle = false;    /* yield control to B */
     goto L
}

active proctype B() provided (toggle == false){
    L:  cnt--;    /* decrement cnt by 1 */
     printf("B: cnt=%d\n", cnt);
        toggle = true;    /* yield control to A */
     goto L
}
```

Message Channels

- Is an FIFO buffer for exchanging messages between processes.
- The name of a channel can be local or global, but the channel itself is always a global object.
- If the name of a channel is local, then its lifetime is depended on the local process lifetime.

Statements

- A PROMELA statement is either
  - executable the statement can be executed, or
  - blocked the statement cannot be executed (yet).
- Statement executions from different processes can be interleaved arbitrarily in time.
- Rules for Executability
  - Basic statements define primitive state transformers in PROMELA.
  - They end up labeling the edges (transitions) in the finite state automata.
  - 6 types of basic PROMELA statements: assign, print, assert, expression, communication(send/receive)
- Control Flow
  - goto, if, do, break, atomic, d_step, unless, ...

Assignment and Print Statement

- Assign statement
  - is always unconditionally executable, changes value of precisely one variable, specified on the left-hand side of the '=' operator.
- Print statement
  - is always unconditionally executable, no effect on state.

Assertion Statement

- assert(expression)
  - An assertion statement is always executable and has no effect on the state of the system when executed.
  - If the expression does not necessarily hold, the assertion statement will produce an error during verifications with SPIN.
  - The assertion statement can be used to check safety properties.
  - An assertion statement can be as a system invariant.
    - Because it is in an asynchronous process, this statement can be executed at any time.

Expression Statement

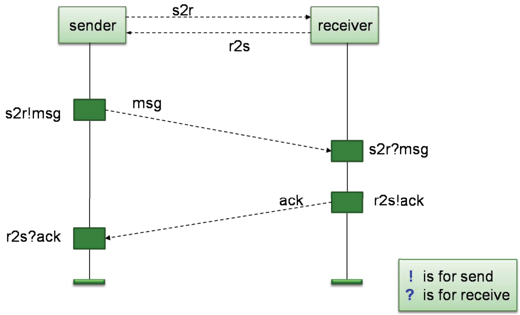
- Executable only if expression evaluates to non-zero (true)
- Example: run P(), else, timeout
  - run
    - returns 0 if the max number of processes would be exceeded by the creation of a new process (the number of processes is bounded).
    - Otherwise, returns the pid of the new process.
  - else
    - is true iff none of the other guards in the same process is executable.
  - timeout
    - is true iff no other statement in any process is executable.
    - can be as a mechanism to avoid deadlock.

timeout vs. else

- timeout and else are related
  - Both are predefined variables.
  - They evaluate to true or false, depending on context.
- timeout is like a system level else, but
  - else cannot be combined with other conditionals.
  - timeout can be combined, e.g. as in (timeout && a $<$ b).

Message Passing

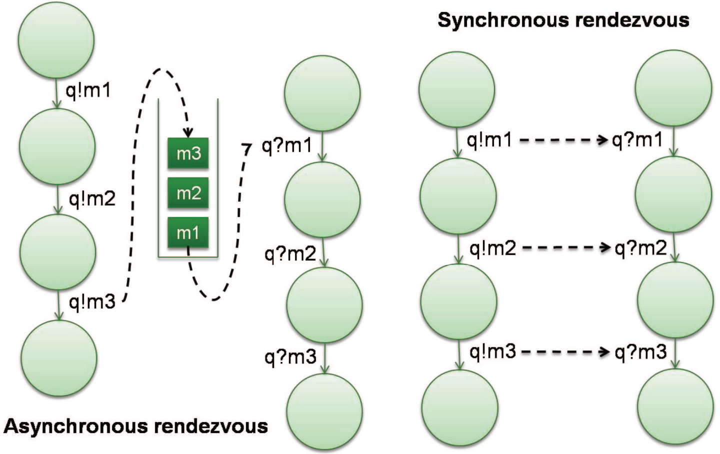
- ! Sending a data over channel
  - Executable when target channel is non-full
  - Q!X : send the value of the variable x through the channel q
- ? Receiving a data from channel
  - Executable when target channel is non-empty
  - Q?X : receive the value of the variable x through the channel q



Rendezvous Communication

- The size of the channel is set to zero.
- A send operation is enabled iff there is a matching receive operation that can be executed simultaneously, with all constant fields matching.
- On a match, both send and receive are executed atomically.
- Example:
  - chan ch = [0] of {bit, byte}
  - Sender offers: ch!1, 3+7
  - Receiver accepts: ch?1, x
  - After the rendezvous handshake completes, x is 10.

Asynchronous and Synchronous Message Passing



Control Flow

- Defining control flow:
  - Semi-colons, goto, break and labels
  - Non-deterministic selection and iteration
    - if...fi
    - do...od
  - Escape sequences:
    - {...} unless {...}
  - Atomic sequences, making things indivisible:
    - atomic{...}
    - d_step{...}

Case Selection

```
if
:: guard₁ -> stmt1.1;stmt1.2;stmt1.3;...
```

if
:: $guard_1$ -> $stmt_{1.1}$;$stmt_{1.2}$;$stmt_{1.3}$;...
:: $guard_2$ -> $stmt_{2.1}$;$stmt_{2.2}$;$stmt_{2.3}$;...
:: ...
:: $guard_n$ -> $stmt_{n.1}$;$stmt_{n.2}$;$stmt_{n.3}$;...
fi

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

Repetition

```
do
:: guard_1 -> stmt_1.1;stmt_1.2;stmt_1.3;...
:: guard_2 -> stmt_2.1;stmt_2.2;stmt_2.3;...
:: ...
:: guard_n -> stmt_n.1;stmt_n.2;stmt_n.3;...
od
```

- If there is none executable statement in a 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.

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 statements in other processes.
  - If any statement other than the guard blocks, atomicity is lost atomicity can be regained when the statement becomes executable.

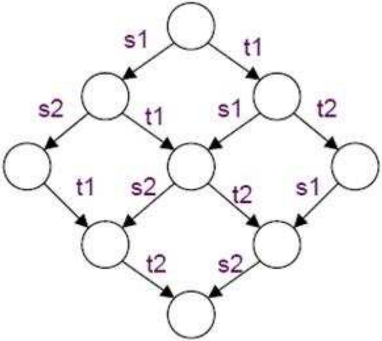
```
atomic{
    /* swap the values of a and b */
    tmp = b;
    b = a;
    a = tmp
}
```

D_step Sequences

- d_step { guard -> $stmt_1$; $stmt_2$; ...; $stmt_n$; }
  - Like an atomic, but must be deterministic and may not block anywhere inside the sequence.
  - Useful to perform intermediate computations with a deterministic result, in a single indivisible step .
  - goto into and 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.

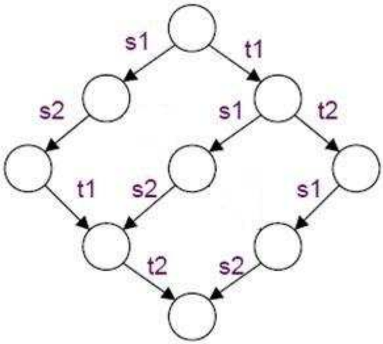
Atomic and D_step Sequences Example(1/3)

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



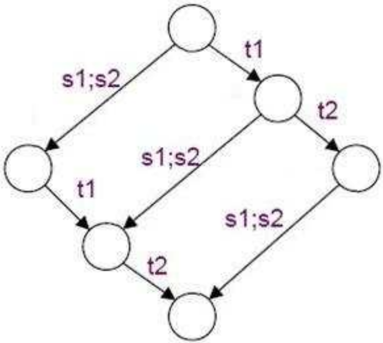
Atomic and D_step Sequences Example(2/3)

```
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 }
```



Unless Statement

- S unless E
  - Is a method to distinguish between higher and lower priority of transitions within a single process.
  - If E ever becomes enabled during the execution of S, then S is aborted and the execution continues with E.

PROMELA Semantics

- By simulating the execution of a SPIN model we can generate a large directed graph of all reachable system states.
- The PROMELA semantics rules define how the global reachability graph for any given PROMELA model is to be generated.

Transition Relation

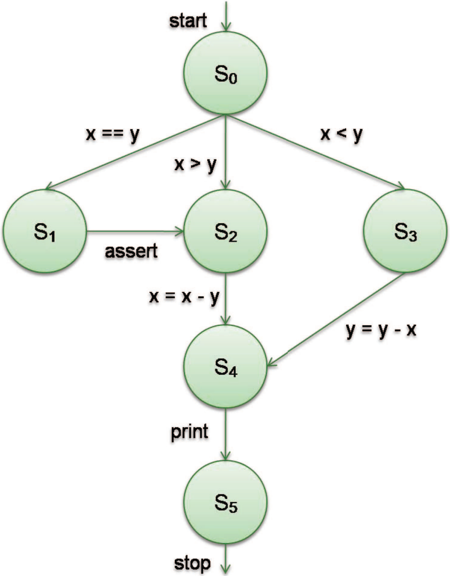
- Every PROMELA proctype defines a finite state automaton, (S, s_0, L, T, F)

Symbol	Finite State Automaton	PROMELA Model
S	Set of states	Possible points of control within the proctype
L	Transition label set	Specific basic statement
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 single-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 */
}
```

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 */
}
```

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 */
}
```

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 */
}
```

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 */
}
```

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 an integer 1..MAXQ that can be stored in a variable.
  - 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 an integer 1..MAXQ that can be stored in a variable.
  - 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 }

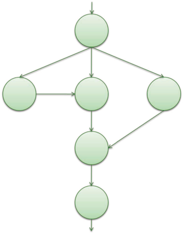
```
chan q = [2] of { mtype, bit };
```

  - Channels always have global scope.
  - A ch_id is an integer 1..MAXQ that can be stored in a variable.
  - 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

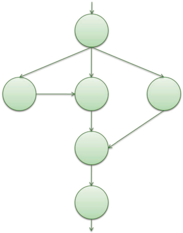
- A process is defined by a six-tuple
  { pid, lvars, lstates, inistate, curstate, transitions }
  - process instantiation number
  - finite set of local variables
  - a finite set of integers defining local proc states
  - the initial state
  - the current state
  - a finite set of transitions (to be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



Operational Model(6/8)

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

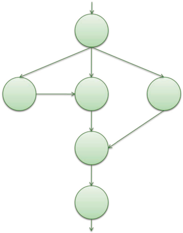
- A process is defined by a six-tuple
  { pid, lvars, lstates, inistate, curstate, transitions }
  - process instantiation number
  - finite set of local variables
  - a finite set of integers defining local proc states
  - the initial state
  - the current state
  - a finite set of transitions (to be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



Operational Model(6/8)

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

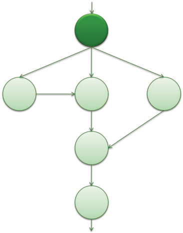
- A process is defined by a six-tuple
  { pid, lvars, lstates, inistate, curstate, transitions }
  - process instantiation number
  - finite set of local variables
  - a finite set of integers defining local proc states
  - the initial state
  - the current state
  - a finite set of transitions (to be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



Operational Model(6/8)

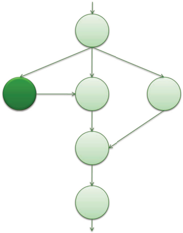
variables, messages, channels, processes, transitions, global states

- 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)

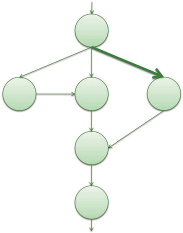
- A process is defined by a six-tuple
  { pid, lvars, lstates, inistate, curstate, transitions }
  - process instantiation number
  - finite set of local variables
  - a finite set of integers defining local proc states
  - the initial state
  - the current state
  - a finite set of transitions (to be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



Operational Model(6/8)

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

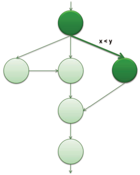
- A process is defined by a six-tuple
  { pid, lvars, lstates, inistate, curstate, transitions }
  - process instantiation number
  - finite set of local variables
  - a finite set of integers defining local proc states
  - the initial state
  - the current state
  - a finite set of transitions (to be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



Operational Model(7/8)

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

- A transition is defined by a seven-tuple
  { tri_id, source-state, target-state, cond, effect, priority, rv }

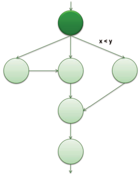


  - Condition and effect are defined for each basic statement, and they are typically defined on variable and channel values, possibly also on process states.
  - Predefined system variables that are used to define the semantics of unless and rendezvous.

Operational Model(7/8)

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

- A transition is defined by a seven-tuple
  { tri_id, source-state, target-state, cond, effect, priority, rv }

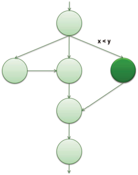


  - Condition and effect are defined for each basic statement, and they are typically defined on variable and channel values, possibly also on process states.
  - Predefined system variables that are used to define the semantics of unless and rendezvous.

Operational Model(7/8)

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

- A transition is defined by a seven-tuple
  { tri_id, source-state, target-state, cond, effect, priority, rv }

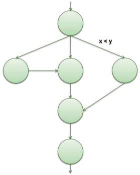


  - Condition and effect are defined for each basic statement, and they are typically defined on variable and channel values, possibly also on process states.
  - Predefined system variables that are used to define the semantics of unless and rendezvous.

Operational Model(7/8)

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

- A transition is defined by a seven-tuple
  { tri_id, source-state, target-state, cond, effect, priority, rv }

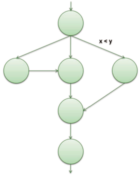


  - Condition and effect are defined for each basic statement, and they are typically defined on variable and channel values, possibly also on process states.
  - Predefined system variables that are used to define the semantics of unless and rendezvous.

Operational Model(7/8)

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

- A transition is defined by a seven-tuple
  { tri_id, source-state, target-state, cond, effect, priority, rv }



  - 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)

- a global 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 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 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 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 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 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

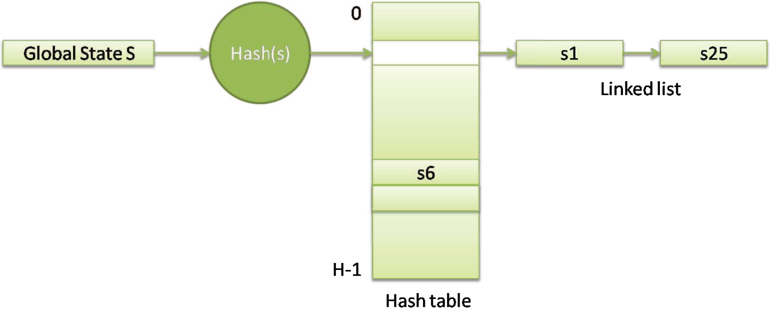
- a global state is defined by a eight-tuple
  { gvars, procs, chans, exclusive, handshake, timeout, else, stutter }
  - a finite set of global variables
  - a finite set of processes
  - a finite set of message channels
  - predefined integer system variables that are used to define the semantics of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

State Vector

- A state vector is the information to uniquely identify a global state.
- It is important to minimize the size of the state vector.
  - state vector = m bytes
  - state space = n states
  - Storing the state space may require n*m bytes.

Storing State in SPIN

- Hash function computes address(index) in the hash table.
- Hash table addresses to linked list states.



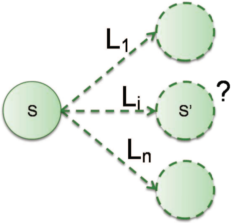
- All states are explicitly stored.
- Lookup is fast due to hash function.
- Memory needed: n*m bytes + hash table.

One-Step Semantics(1/2)

- Given an arbitrary global state of the system, determine the set of possible immediate successor states.
  - We've defined the only 4 types of objects that hold state:
    - variables, messages, channels, processes
  - 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 system 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**

The Next-State Relation

```
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 */
}
```

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{










        }
    }
}

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

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
  - 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}
```

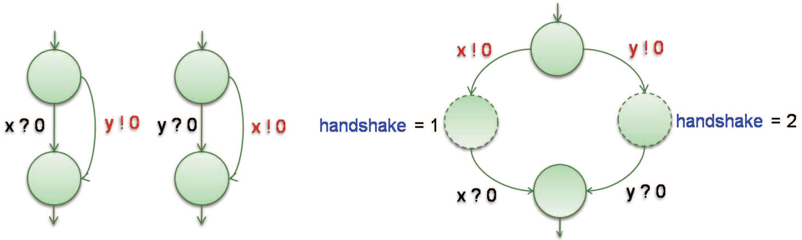
```
chan x = [0] of {bit}
chan y  = [0] of {bit}
active proctype A() {x!0 unless y?0}
active proctype B() {y!0 unless x?0}
```

PROMELA Models(2/2)

- The unless keyword has a lower execution priority than the statement that follows it
- Rendezvous handshakes occur in two parts:
  - Sender offers
  - Receiver accepts

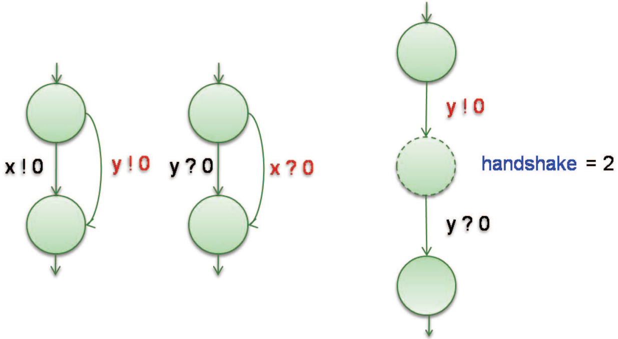
Example 1:3

```
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}
```



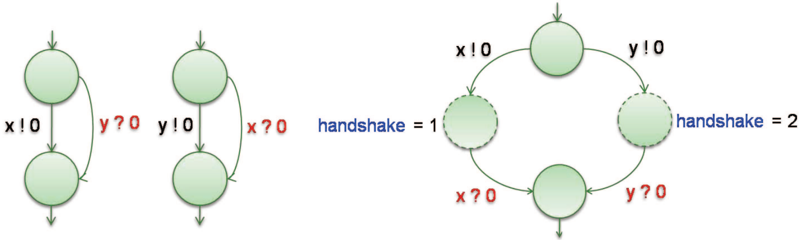
Example 2:3

```
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
  - Correctness Property
  - SPIN's LTL Syntax
  - LTL Semantic
  - Specifying LTL properties
- DEMO with XSPIN

Correctness Property

- With SPIN one may check the following type of properties:
  - Assertions
  - LTL formulae
  - Safety properties: nothing bad happens
    - Deadlocks (default)
    - Unreachable code(default)
  - Liveness properties: eventually something good happens
    - Non-progress cycles (livelocks)
    - Acceptance cycles

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)
```

LTL semantics

- Given an infinite trace $\tau = t_0, t_1, t_2, \ldots$ and a LTL formula φ we can decide if $\tau \models \varphi$ depending on the structure of φ
- $\tau \models []\varphi$, iff $\tau_i \models \varphi$, $\forall i \geq 0$
- $\tau \models <> \varphi$, iff $\exists i \geq 0$ s.t. $\tau_i \models \varphi$
- $\tau \models !\varphi$, iff $\neg(\tau \models \varphi)$
- $\tau \models \varphi_1$ U φ_2, iff $\exists j \geq 0$ s.t. $\tau_i \models \varphi_1$, for $0 \leq i < j$ and $\tau_j \models \varphi_2$

Specifying LTL properties

- LTL Formulae examples:

[] 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

Agenda

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

DEMO with XSPIN

- Introduction to XSPIN
- DEMO

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.
  - It is interesting to note that even the Communication of the ACM was fooled on this one.
- 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 with XSPIN
- 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