The SPIN Model Checker

Jinesh M.K

AMRITA VISHWA VIDYAPEETHAM UNIVERSITY

18 August 2017

jinesh@am.amrita.edu

Introduction to SPIN

- SPIN = **S**imple **P**romela **In**terpreter
- Popular open-source model checker
- Formal verification of asynchronous and distributed software systems
- Developed at Bell Labs during 1980's and '90s
- Gerard Holzmann won the ACM software award for SPIN
- Written in ANSI standard C, and is a portable across multiple platforms
- SPIN homepage www.spinroot.com
- Concurrent systems are specified in the modeling language called Promela

SPIN Basic Concepts

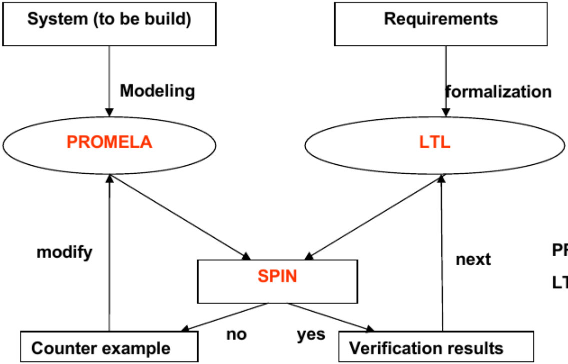
Common flaws in DS

- Deadlock
- Livelock, starvation
- Underspecification
- Overspecification

SPIN Basic Concepts

- **Simulator**- To get a quick impression of the behavior
  - guided simulation
  - random and interactive simulation
- **Verifier**- When a counterexample is generated, it uses simulation to step through the trace
  - to check assertions and temporal formula

SPIN Basic Modes



Process of the SPIN Model Checker

PROMELA → Kripke Structure

LTL → Büchi Automaton

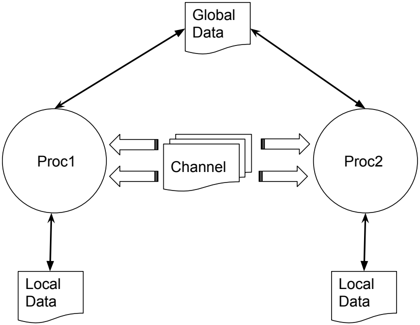
[1]

- C-like notation (Promela) for specifying the finite-state abstraction
- Expressing general correctness requirements as LTL formula

[1] Simple Promela Interpretor (SPIN) Model Checker By Prabhu Shankar Kaliappan

Promela Introduction

- PROMELA = **Pro**cess/**Pro**tocol **Me**ta **La**nguage
- Allows for the dynamic creation of concurrent processes
- Non-deterministic , guarded command language
- C language in some of the syntax and notational conventions
- CSP like message channel and global variable for inter-process communication



Promela

What is possible ?

- Process behavior
- Variables, data types
- Message channels

What is valid ?

- Assertions
- End-state, progress-state, and acceptance state labels
- Never claims (LTL formula)
- Trace assertions
- Default properties
  - Absence of system deadlock
  - Absence of unreachable code

Promela Model

- Process
  - Global Objects
  - Specify the behavior
  - communicating over channels and shared variables
  - Processes execute asynchronously
  - Keyword: `proctype`
- Message channels
  - Synchronous and asynchronous channel
  - Inter-process communication
  - Keyword: `chan`
- Variables
  - Local and global
  - Data types: `int`, `byte`, `mtype` `etc`

```
mtype = {MSG, ACK};
chan sch=...;
chan rch=...;
bool flag;

active proctype Sender(){
...process body...
}

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

Data Types

- Variables can be local or global
- Default initial value of both local and global variables is 0
- Variables can be assigned a value by an assignment, argument passing or message passing
- Variables can be used in expressions which includes most arithmetic, relational and logical operators
- Multi-dimensional arrays can be defined indirectly with the help of the `typedef` construct

- Basic types
  - `bit` – [0,1]
  - `bool` – [true,false]
  - `byte` – [0..255]
  - `short` – [-2^{15}..$2^{15}-1$]
  - `int` – [-2^{31}..$2^{31}-1$]
- Array

  eg. bool name[N];
- Records type

```
typedef Msg{
    bit a[10],b;
    chan c;
}
Msg msg;
msg.a[1]=10;
```

- Enumeration type for messages

```
mtype ={msg, ack, rec}
```

Promela Process

- A process executes concurrently with other processes
- A process also communicates with other processes by sending/receiving messages across channels by using shared (global) variables with other processes
- Variable/message channel can only be changed/inspected by processes
- Local state of a process is defined by process counter (defines the location of the process) and the values of the local variables of the process
- `atomic` blocks avoid concurrent update problems.
- Defined using `proctype` keyword and optional `active` keyword for process creation

```
[active] proctype <process_identifier> (<formal parameter>)
{ local variable declaration and statements }
```

- Process creation using `run` keyword

```
run <name>(<actual parameter>)
```

Promela Process with atomic blocks

```
byte state = 0;
proctype A(){
    atomic {state = state + 10}
}

proctype B(){
    atomic {state = state + 20}
}
init{ run A(); run B() }
```

- atomic block executes without being interrupted by other processes
- `d-step{stmt1;,...stmtn;}`- same as atomic but if one of the statements stmti blocks,it is a run-time error

Message Channel

- Communication between processes through channels

  `chan` *name* = [*buffer size*] `of` {*data type* }

- FIFO
- There can be two types of communications:
  - Message-passing or asynchronous
  - Rendezvous or synchronous (channel of dimension 0)
- Sending message (!)
  ch!0 - sending over channel ch; block if c is full
- Receiving message (?)
  ch?c - receives from channel ch and pass to c ; block if ch is empty
- It is an error to send or receive either more or fewer parameters per message than was declared for the message channel

```
chan c   = [0] of {bit};
chan d   = [2] of {mtype, bit, byte};
chan e[2] =  [1] of {mtype, record};
```

Message Passing

```
proctype A(chan q1){
  chan q2;
  q1?q2;
  q2!123
}
proctype B(chan qforb){
  int x;
  qforb?x;
  printf("x=%d\n",x);
}
init {
  chan qname = [1] of { chan };
  chan qforb = [1] of { int };
  run A(qname);
  run B(qforb);
  qname!qforb
}
```

Promela statements

- Statements are separated by a semi-colon
- Assignments and expressions are statements
- skip statement: does nothing, only changes the process counter
- printf statement: not evaluated during verification
- `assert(expr)`: Assert statement is used to check if the property specified by the expression expr is valid within a state.
- Semi-colon is used a statement separator not a statement terminator
  - Last statement does not need semi-colon
  - Often replaced by $->$ to indicate causality between two successive statements
    $(a == b); c = c + 1$
    $(a == b) -> c = c + 1$

Case Selection: Conditional

```
if
 :: alternative1 -> stat1.1; stat1.2;
 :: ...
 :: alternativen -> statn.1; statn.2;
fi;
```

- Only one executes
- Non-deterministically select one enabled alternatives.
- If none exists, the whole “if” blocks.
- `else` condition is executable iff no other statement is executable
- `goto` : Unconditional Jump

Example

```
if
:: a > b -> printf("a");
:: a == b -> prinf("b");
:: else -> ...
fi
```

Repetition - loop

```
do
 :: alternative1
 :: alternativen
do;
```

- The first action in an alternative acts as its “guard”, which determines if the alternative is enabled on a given state
- At each iteration, non-deterministically choose one enabled alternatives
- If there is none, the entire loop blocks
- `break` is used to terminate the repetition structure

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

Assertions

`assert` *(any_boolean_condition)*

- Assert statements are always executable
- If the boolean condition specified holds, the statement has no effect
- If condition doesn't hold, the statement will produce an error report during verification.

For stating simple safety properties

```
assert(x+1 != 2)
assert(y>2)
```

`assert(false)` - checks reachability of certain locations in proctype body

Timeouts

- The timeout models a special condition that allows a process to abort the waiting for a condition that may never become true.
- Becomes true only when no other statements within the distributed system is executable.

```
proctype watchdog() {
  do
    :: timeout -> guard!reset
  od
}
```

Statements

- A statement is either
  - executable - immediately execute
  - blocked - a statement cannot be executed
- An assignment,`skip`, `break` are always executable
- An expression is also a statement; it is executable if it evaluates to non-zero
  - $5 < 6$ - always executable
  - $x < 5$ - executable only if x is less than 5
- A `run` statement is only executable if a new process can be created
- `printf` statement is always executable
- `if` and `do` statement are executable ,if at least one choice is executable

Simple Mutual Exclusion

```
bool busy
byte mutex
active[2] proctype P(){
  (!busy) -> busy =true;
  mutex++;
CS: printf("P-%d in CS \n", _pid);
  assert(mutex <=1);
  mutex--;
  busy = false;
}
```

Verification

pan:1: assertion violated (mutex<=1) (at depth 9)

Both process can access `!busy` at same time.

```
bool x,y
byte mutex
active proctype A(){
   x = true;
   y == false;
   mutex++;
CS1: //CS
   assert(mutex<=1)
   mutex--;
   x = false;
}
active proctype B()
{
   y = true;
   x== false;
   mutex++;
CS2: //CS
   assert(mutex<=1)
   mutex--;
   y = false;
}
```

Verification

pan:1: invalid end state (at depth 1)

In-valid end state: state where not all active processes are either at the end of their code or at a local state that is marked with and end-state label

Simple Mutual Exclusion

```
bool x,y ,t
byte mutex
active proctype A()
{
    x = true;
    t = true;
    y == false || t == false ;
    mutex++;
CS1: //CS
     assert(mutex<=1)
     mutex--;
     x = false;
}
```

```
active proctype B()
{
    y = true;
    t = false;
    x== false || t == true;
    mutex++;
CS2: //CS
    assert(mutex<=1)
    mutex--;
    y = false;
}
```

Verification

No errors found – did you verify all claims?

Linear Temporal Logic

- LTL formula can be used to express both safety and liveness properties
- An LTL formula *f* may contain any lowercase propositional symbol p, combined with unary or binary, Boolean and/or temporal operators

```
f ::= p | true | false | (f) | f binop f | unop f
unop ::= [] | <> | !
binop ::= U | && | || | -> | <->
```

- A Promela specification defines a model, *M* i.e., a set of sequences.
- The LTL formula specifies a set of behaviors, *L*, that must hold.
- Correctness of the model requires that $M \subseteq L$
- SPIN checks that $M \bigcap L^c =$, where L^c is the complement of L
- L^c is specified as a "never claim"
- In-line specification

`ltl` *<name>* {*<formula>*}

- LTL properties in mutual exclusion algorithm

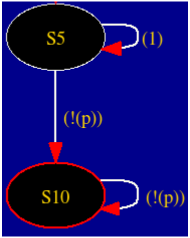
```
ltl claim1 {[](A@CS1 -> mutex <=1)}
```

Never Claim

- Used to specify either finite or infinite system behavior that should never occur
- Defined as a series of propositions, or boolean expressions, on the system state that must become true in the sequence specified for the behavior of interest to be matched
- `spin -f` can be used to generate promela never claim code from LTL formula

Example- spin -f [](<>p)

```
never  {
T0_init:do
:: ((p)) -> goto accept_S10
:: (1) -> goto T0_init
od;
accept_S10:
do
:: (1) -> goto T0_init
od;
}
```



Vending Machine

```
#define CPRICE 10
#define TPRICE   5
#define COFFEE   1
#define TEA    0
chan d_chan=[1] of {bit};
chan c_chan=[1] of {byte};
bool paid;
bool happy;
ltl p0 {[](paid -> <>happy)}
proctype vender() {
 byte price;
 coin_channel?price;
 if
 ::price==CPRICE -> d_chan!COFFEE;
 ::price==TPRICE -> d_chan!TEA;
 ::else ->skip;
 fi
}
```

```
proctype customer(byte price){
 happy=0; paid=0;
 if
 ::price!=CPRICE&&price!=TPRICE
     -> goto end;
 ::else->skip;
 fi;
 bit drink;
 c_chan!price; paid=1;
 d_chan?drink;
 if
 ::price==CPRICE&&drink==COFFEE
     -> happy=1;
 ::price==TPRICE&&drink==TEA
    -> happy=1;
 ::else ->skip;
 fi;
 end: printf("Happy=%d", happy);
}
```

Verification in SPIN

- How to specify the correctness properties
  - Safety properties
    - Assertion
    - Invalid Endstates
  - Liveness Properties
    - Non-progress cycles
    - acceptance cycles
- Never claim
  - Express the safety and liveness property through LTL
  - Conversion of LTL properties into Büchi automaton is called Never Claim which is done automatically

Invalid Endstates

- Identified using **end** state label
- label-name prefix **end** for marking valid termination states
- Distinguishes between valid and invalid state
- In-valid end state leads to deadlock
- Can be conditional or unconditional goto

```
active proctype dijkstra()
{
end1:   do
         :: sema!p -> sema?v
        od
}
```

Non-Progress Cycles

- To ensure that the process is making an effective progress(liveness)
- Progress statement can label that starts with the eight-character sequence **progress**

Peterson algorithm

```
active proctype dijkstra()
{
  do
   :: sema!p ->
   progress: sema?v
  od
}
```

Any infinite system execution contains infinitely many executions of the statement sema?v

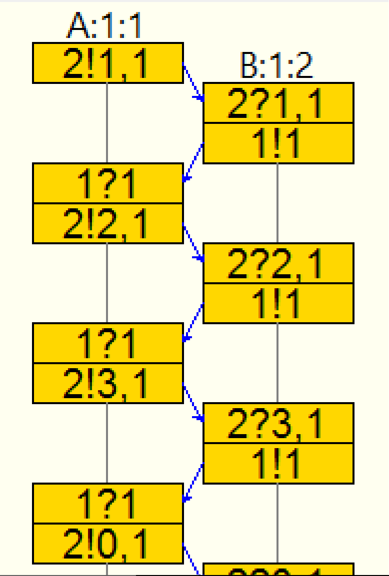
Acceptance cycle

- Mark a state with a label name that starts with the six-character sequence **accept**
- Reserved for 'never' clause

```
  dell: spin -f '[]<>(p U q)'
never {    /* []<>(p U q) */
T0_init:
        if
        :: (q) -> goto accept_S9
        :: (1) -> goto T0_init
        fi;
accept_S9:
        if
        :: (1) -> goto T0_init
        fi;
}
```

Sequence Diagram

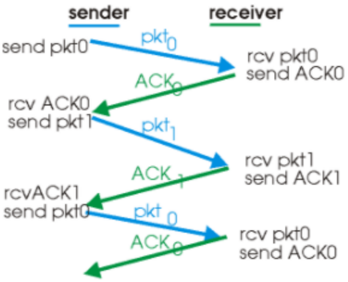
A Sequence Diagram that clarifies the sending/receiving of messages between processes on Promela channels



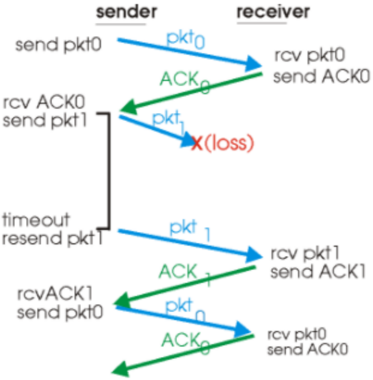
Alternate Bit Protocol

- ABP is a simple network protocol operating at the data link layer that retransmits lost or corrupted messages.
- Messages are sent from transmitter A to receiver B. Each message from A to B contains a data part and a one-bit sequence number, a value that is 0 or 1. B has two acknowledge characters that it can send to A: ACK0 and ACK1.
- When A sends a message, it resends it continuously, with the same sequence number, until it receives an acknowledgment from B that contains the same sequence number. Then, A complements the sequence number and starts transmitting the next message.
- When B receives a message that is not corrupted and has sequence number 0, it starts sending ACK0 and keeps doing so until it receives a valid message with number 1. Then it starts sending ACK1.

Alternate Bit Protocol



(a) operation with no loss



(b) lost packet

Promela Specification of ABP

```
mtype = {msg, ack};
chan tosndr = [2] of {mtype, bit};
chan torcvr = [2] of {mtype, bit};
active proctype sender()
{
  bool seqout, seqin;
  do
   :: torcvr!msg,seqout ->
              tosndr?ack,seqin;
  if
    :: seqin == seqout ->
       seqout = 1- seqout ;
    ::else->skip
  fi
  od
}
active proctype receiver()
{
 bool seqin;
 do
   :: torcvr?msg,seqin ->
              tosndr!ack,seqin;
od
}
```

Sybil Attack: a Case Study

- Single entity can gain control over a substantial fraction of the system by presenting multiple identities
- Two types
  - A single node presents multiple identities
  - Node uses the identity of another node
- Violates the fundamental assumption of one-to-one correspondence with of a node with its identity
- Create issue in
  - Routing
  - Tampering with Voting and Reputation Systems
  - Fair Resource Allocation
    Data Aggregation

Sybil Attack Detection

- Based on a traditional public key certificate together with position verification
- Scheme is founded on the concept of a location certificate issued by a RSU for communication with other vehicles under the same RSU
- No dependence on specialized hardware
- Central Authority (CA) and RSUs both participate in detection
- Node authentication depends on geo-location information
- Support for high vehicular mobility
- Sybil nodes are isolated from the network

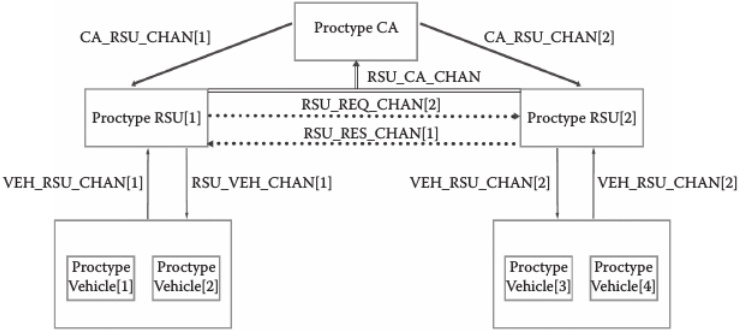


Figure 3.7: Promela channels and process types [2]

[2] Sybil Attack Detection in Vehicular Networks,*Jinesh M.K, Bharat Jayaraman and Krishnashree Achuthan*, Security and Privacy in Internet of Things (IoTs) Models, Algorithms, and Implementations, CRC Press

Some Inspiring Applications of Spin

- Verification of the control algorithms for the flood control barrier built in the Netherlands
- Verification of handoff algorithms for the dual control CPUs
- Correctness of Mar's Exploration Rovers
- NASA's investigation of the control software of the Toyota Camry MY05
- Verification of medical device transmission protocols
- Verification of cryptographic protocols

Hands-on Experiment with SPIN & iSPIN

- Download and install SPIN and iSPIN
- On-line Manuals

  SPIN- http://spinroot.com/spin/Man/Manual.html
  Promela- http://spinroot.com/spin/Man/promela.html
  Examples- http://spinroot.com/spin/Man/Exercises.html
- Write a promela program to model simple traffic light and verify the liveness properties using LTL formula
- Modify the ABP promela code to handle message lose in communication [Hint: use `timeout` statement]
- Extend the simple traffic light to two-way traffic light and verify the safety and liveness properties[You can remove 'yellow' transition for convenience]