Model-Checker SPIN

- For proving correctness of process interactions
- These are specified using buffered channels, shared variables, or a combination
- Focus - asynchronous control in software systems
- has program-like notation for specifying design choices (Promela)
  - models are bounded and have countably many distinct behaviors
- powerful notation for expressing general correctness requirements (LTL)
- methodology for establishing logical consistency of the design choices against correctness requirements (model-checker SPIN)

Structure of SPIN simulation and verification

Fig 1, p. 2 of TSE paper

Overview of SPIN

- Promela
- SPIN
  - random simulations of the system's exection
  - generate a C program that performs an efficient online verification of the system's correctness properties
  - check for absence of deadlock, unspecified receptions, and non-executable code
  - verify correctness of system invariants, find non-program execution cycles, and verify correctness properties expressed in LTL.
- LTL (Linear Temporal Logic)
  - How to write LTL?
  - How to store LTL formulae? (Buchi automata)
- How does SPIN work?
- Some examples

Extra References

[1] W. Thomas, "Automata on Infine Objects", *Handbook on Theoretical Computer Science*, J. Van Leeuwen, ed., pp. 133-187, Elsevier Science, 1990

[2] Vardi and Wolper, "Reasoning About Infinite Computations", *Information and Computation*, vol. 115, pp. 1-37, 1994.

[3]Vardi and Wolper, "An Automata-Theoretic Approach to Automatic Program Verification", *Proc. First IEEE Symp. Logic in Computer Science*, pp. 322-331, 1986.

Promela (Process Meta Language)

- Model and verify relevant behavior
- construct increasingly more detailed Promela models, verified under different assumptions about the environment
- once correctness has been established, it can be used in verification of subsequent models
  - programs consist of processes, message channels and variables
  - every statement is guarded by a condition. It is executable when the condition is true. Otherwise, it blocks until condition becomes true

```
while (a != b)
    skip  /* wait for a == b */
```

vs

```
    (a == b)
```

Process Types

- State of variable or message channel can only be changed or inspected by processes (defined using `proctype`)
- `;` and `->` are statement *separators* with same semantics. `->` is used informally to indicate causal relation between statements

Example:

```
byte state = 2;
proctype A()
{      (state == 1) -> state = 3
}
proctype B()
{      state = state - 1
}
```

- State here is a global variable

Process Instantiation

- Need to execute processes (`proctype` only defines them)
- By default, process of type `init` always executes.
- `run` starts processes
- processes can receive parameters: all basic data types and message channels. Data arrays and process types are not allowed.

Example:

```
proctype A(byte state: short foo)
{      (state == 1) -> state = foo
}
init
{      run A(1,3)
}
```

Process Instantiation (Cont'd)

- If have several processes allowed to read and write the value of a shared variable, have necessity for mutual exclusion. Here is one solution:

```
#define true 1
#define false 0
#define Aturn false
#define Bturn true
bool x,y,t;
proctype A()                  proctype B()
{ x = true;                   {  y = true;
  t = Bturn;                     t = Aturn;
  (y == false || t == Aturn);    (x == false) ||
  /* critical section */            (t == Bturn);
  x = false                     /* critical section */
}                               y = false }
init
{ run A();  run B()}
```

Atomic Sequences

- Keyword `atomic` takes care of the *test and set* problem
- this prohibits interleaving during this operation and reduces complexity of verification model

Example:

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

Message Passing

`chan qname = [16] of {short}` - declaration

`qname!expr` - writing (appending) to the channel

`qname?expr` - reading (from head) of the channel

`qname!expr1,expr2,expr3` - writing several vars

`qname?var1,var2,var3` - reading several vars

`qname!expr1(expr2,expr3)` - message type and

`qname?var1(var2,var3)` params

`qname?cons1,var2,cons2` - can send constants

- less parameters sent than received - others are undefined
- more parameters sent - remaining values are lost
- constants sent must match with constants received

Message Passing - Example

```
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
}
```

this prints 123

Rendez-Vous Communications

- Buffers of size 0 - can pass but not store messages
- these message interactions are by definition synchronous
- defined only on two processes, a sender and a receiver

Example:

```
#define msqtype 33
chan name = [0] of { byte, byte };
proctype A(0
{             name!msgtype(124);
              name!msgtype(121); /* non-executable */
}
proctype B()
{             byte state;
              name?msgtype(state)
}
init
{             atomic {  run A();  run B()  }
}
```

Rendez-Vous Communications (Cont’d)

- If channel `name` has zero buffer capacity:

  handshake on message `msgtype` and transfer of value `123` to variable `state`. The second statement in `A` will be unexecutable since no matching receive operation in `B`
- If channel `name` has size 1:

  process `A` can complete its first send, but blocks on second since channel is filled. `B` can retrieve the first message and complete. Then `A` completes, leaving its last message as a residual in the channel
- If channel `name` has size 2 or more:

  `A` can finish its execution before `B` even starts

Example using Control-Flow: Dijkstra Semaphore using rendezvous

```
#define p    0
#define v    1
chan sema = [0] of { bit };

proctype dijkstra()
{  byte count = 1;
   do
   :: (count == 1) -> sema!p; count = 0
   :: (count == 0) -> sema? v; count = 1
   od
}
proctype user()
{  do
   ::  sema? p;
       /* crit. sect */
       sema!v;
       /* non-crit. sect. */
   od
}
init
{      run dijkstra(); run user();
       run user(); run user()
}
```

Other Promela Features

- Can model procedures and recursion
- all sorts of control flow (loops, cases, ifs, breaks, gotos)
- timeouts
- assertions
- message type definitions
- pseudo statements

See Web pages (Promela.html) for more description

Example - protocol

- Channels `Ain` and `Bin` are to be filled in with token messages of type `next` and arbitrary values (ASCII chars) by unspecified background processes: the users of the transfer service.
- These users can also read received data from the channels `Aout` and `Bout`.
- The channels are initialized in a single atomic statement, and started with the dummy `err` message.

Another Example

```
mtype = {ack, nak, err, next, accept};
proctype transfer(chan in,out,chin,chout)
{      byte o, I;
       in?next(o);
       do
       :: chin?nak(I) ->
                    out!accept(I);
                    chout!ack(o)
       :: chin?ack(I) ->
                    out!accept(I);
                    in?next(o);
                    chout!ack(o)
       :: chin?err(I) ->
                    chout!nak(o)
       od
}
```

Example (Cont’d)

```
Init
{      chan AtoB = [1] of { mtype, byte };
       chan BtoA = [1] of { mtype, byte };

       chan Ain    = [2] of { mtype, byte };
       chan Bin    = [2] of { mtype, byte };

       chan Aout  = [2] of { mtype, byte };
       chan Bout  = [2] of { mtype, byte };

       atomic {
              run transfer(Ain, Aout, AtoB, BtoA);
              run transfer(Bin, Bout, BtoA, AtoB);
       };
       AtoB!err(0)
}
```

LTL and Buchi Automata

- Can use LTL to express safety and liveness properties
  - [] (p U q) - always p remains true at least until q becomes true
  - ! (<> (p U q)) - never is there a point in the execution such that p remains true at least until q becomes true
  - [] (<> (p || q)) - at any point of execution it is guaranteed that eventually either p or q will become true at least once more
  - ! (p U ([] (q U r))) - it is not true that p is true at least until the point such that for all paths q is true at least until r is true

LTL and Buchi Automata (Cont’d)

- Can automatically convert from an LTL formula to a Buchi automaton expressed in terms of Promela code
- The automaton accepts a system execution iff that execution forces it to pass through one or more of its accepting states infinitely often (acceptance states).
- Conditions are on the transitions of the automaton, not on the states
- To prove that no execution sequence of the system matches the negated correctness claim, it suffices to prove the absence of acceptance cycles in the combined execution of the system and the Buchi automaton representing the claim.
- This execution is formally defined by a synchronous product of the system and the claim

From LTL to Buchi Automata - Examples

```
• $ spin -f “[]<>(p||q)”
never {
TO_init:
       if
       :: (1) -> goto TO_init
       :: (p||q) -> goto accept_S10
       fi
accept_S10:
       if
       :: (1) -> goto TO
       fi
accept_all:
       skip
}
```

Examples (Cont’d)

```
• $ spin -f “![]<>(p||q)”
never {
TO_init:
       if
       :: (1) -> goto TO_init
       :: (!p && !q) -> goto accept_S2
       fi;
accept_S2:
       if
       :: (!p && !q) -> goto TO_S2
       fi;
TO_S2:
       if
       :: (!p && !q) -> goto accept_S2
       fi;
accept_all:
       skip
}
```

Examples - Cont’d

```
• spin -f “!( [] (p U q) )”
never {
TO_init:
       if
       :: (1) -> goto TO_init
       :: (!p && !q) -> goto accept_all
       :: (! q) -> goto accept_S2
       fi;
accept_S2:
       if
       :: (!p && !q) -> goto accept_all
       :: (!q) -> goto accept_S2
       fi;
TO_S2:
       if
       :: (!p && !q) -> goto accept_all
       :: (!q) -> goto accept_S2
       fi;
accept_all:
       skip
}
```

LTL to Buchi Automaton - Algorithm

Idea: compute the set of subformulas that must hold in each reachable state and in each of its successor states

convert formula into normal form

create initial state, marked with the formula to be matched and a dummy incoming edge

create `accept_all` state

recursively:

take subformula to be satisfied

leading operators much split the current state into two, with each copy inheriting a different part of the subformula

make connections to `accept_all` state (using until operator)

Nested Depth-First Search

- Problem: need to determine cycles. And the method needs to be compatible with all modes of verification
- Solution (Tarjan) - construct strongly-connected components in linear time by adding 2 integers: *dfs*-number and *lowlink*-number (32 bits of storage each because of huge state space)
- Idea: visit each state twice, but storing every state only once. Only 2 bits of overhead instead of 64 by using encoding
- For an accepting cycle to exist in the reachability graph, at least one accepting state must be both reachable from the initial system state (root) and must be reachable from itself

Nested Depth-First Search (Cont'd)

Using depth-first search find accepting states reachable from the root.

For each such state

use depth-first search to see if this state is reachable from itself

if so, we found an *acceptance cycle:* a counter-example to a user-defined correctness claim

- can only generate one acceptance cycle, not all, but will always find at least one if it exists
- can also extend the algorithm with weak fairness constraint: every process that contains at least one transition that remains enabled infinitely long, is guaranteed to execute that transition within finite time

Partial Order Reduction

- Idea: validity of an LTL formula is often insensitive to the order in which concurrent and independently executed events are interleaved in the depth-first search
- Thus can generate a state-space with only representatives of classes of execution sequences that are indistinguishable for a given correctness property
- In a typical case, the reduction in the state space size and in the memory requirements are linear in the size of the model, yielding savings in memory and runtime from 10 to 90 percent.
- This method cannot lead to noticeable increase in memory requirements
- Method not sensitive to decisions about process or variable orderings (unlike BDDs)

Memory Management

- Size of interleaving product can grow exponentially with the number of processes!
- For LTL properties, the verification time in the worst case is exponential in the number of temporal operators (unlike branching-time logic!)
- Goal: create algorithms that can economize the memory requirement of a reachability analysis, without incurring unrealistic increases in runtime requirements.
- Examples: state compression and bit-state hashing

State Compression

- About 10-20 percent run-time overhead in return for 60-70 percent reduction in memory utilization
- Every process and every channel in a PROMELA specification has only relatively small number of unique local states - so store them separately and use unique indices into the local state tables
- So, 256 distinct local states = 1 byte of memory within the global state descriptor. 256 and fever - 8 bits.
- User can set up this information (size of index) to 1, 2, 3, or 4 bytes.

Bit-State Hashing

- Sometimes cannot have exhaustive verification, so all other techniques stop when they run out of memory.
- With amount of memory M and number of states R and S bytes to store each state, the checker exhausts memory after M/S states. *Problem coverage* is M/(R * S).
  - Example: with 64 bytes of memory to encode each state and total of 2 Mb, we can store 32,768 states.
- Bit-state hashing usually does much better than that.
- Each reachable state is stored using two bits of information
- command line:

```
cc -DBITSTATE -o run pan.c
```

- Can specify amount of available (non-virtual) memory directly, using `-w N` option, e.g., `-w27` means that we have 128 Mb of memory.

Bit-Size Hashing

- Exact algorithm could not be determined, but here is an example:

```
$ run
assertion violated (I == ((last_I + 1))
pan: aborted
search interrupted
…
hash factor: 67650.064516
(size 2^22 states, stack frames: 0/5)
```

- Hash factor: maximum number of states/actual number
- Maximum number of states is 2^{22} bytes or about 32 million bits = states
- Hash factor > 100 - coverage around 100%
- Hash factor = 1 -> coverage approaches 0%

Verification example: mutual exclusion

```
bool want[2];  /* Bool array b */
bool turn;     /* integer    k */

proctype P(bool I)
{
  want[I] = 1;
  do
  :: (turn != I) ->
          (!want[1-I]);
          turn = I
   :: (turn == I) ->
          break
   od
   skip;  /* critical section */
   want[I] = 0
 }

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

Mutual Exclusion (Cont’d)

- Generate, compile and run the verifier to check for deadlock and other major problems. Result:

```
$ spin -a hyman0
$ cc -o pan pan.c
$ pan
full statespace search for:
assertion violations and invalid endstates
vector 20 bytes, depth reached 19, errors: 0
  79 states, stored
  0   states, linked
    38 states, matched     total: 117
hash conflicts: 4 (resolved)
(size 2^18 states, stack frames: 3/0)
unreached code _init (proc 0):
  reached all 3 states
unreached code P (proc 1):
  reached all 12 states
```

Mutual exclusion (Cont’d)

- Want to check mutual exclusion.

```
bool want[2];  /* Bool array b */
bool turn;     /* integer    k */
byte cnt;
proctype P(bool I)
{
  want[I] = 1;
  do
  :: (turn != I) ->
          (!want[1-I]);
          turn = I
   :: (turn == I) ->
          break
   od
   skip;  /* critical section */
   cnt = cnt+1;
   assert(cnt == 1);
   cnt = cnt-1
   want[I] = 0
 }
 init { run P(0); run P(1) }
```

Mutual Exclusion (Cont’d)

- Verifier says that assertion can be violated, and we can use options -t -p to find out the trace (or do the same thing using Xspin’s nice graphic capabilities)
- Another way of catching the error : having another process with the assertion, allowing all possible relative timings of the processes.
- This is an elegant way to check the validity of a system invariant

Mutual Exclusion (Cont’d)

```
bool want[2];  /* Bool array b */
bool turn;     /* integer    k */
byte cnt;
proctype P(bool I)
{
  want[I] = 1;
  do
  :: (turn != I) ->
          (!want[1-I]);
          turn = I
   :: (turn == I) ->
          break
   od
   cnt = cnt+1;
   skip;  /* critical section */
   cnt = cnt-1;
   want[I] = 0
 }
 proctype monitor()
 {  assert (cnt == 0 || cnt == 1) }
 init { run P(0); run P(1); run monitor() }
```

Verification example: leader Election

- Leader election in a unidirectional ring. All processes participate in the election (cannot join in after the execution started)
- Global property: it should not be possible for more than one process to declare to be the leader of the ring
- To check this property, either specify it using LTL:
  `[] (nr_leaders <= 1)`
- Or (much more efficiently) use assertion (line 57)
  `assert (nr_leaders == 1)`
- Also want to specify that eventually a leader is elected:
  `<>[] (nr_leaders == 1)`

Verification Model of Leader Election

```
1  #define N  5   /* nr of processes */
2  #define I  3   /* node given the smallest number */
3  #define L 10   /* size of buffer (>= 2*N) */
4
5  mtype = { one, two, winner}; /* symb. Msg. Names */
6  chan q[N] = [L] of {mtype, byte} /* assynch. Chnl */
7
8  byte nr_leaders = 0;  /* count the number of process
9         that think they are leader of the ring */
10 proctype node (chan in, out; byte mynumber)
11 { bit Active = 1, know_winner = 0;
12   byte nr, maximum = mynumber, neighbourR;
13
14   xr in;  /* claim exclusive recv access to in */
15   xs out; /* claim exclusive send access to out */
16
17   printf(“MSC: pecent\n”, mynumber);
18   out!one(mynumber) /* send msg of type one */
19 end:       do
20   :: in?one(nr) ->  /* receive msg of type one */
```

Verification Model of Leader Election (Cont’d)

```
21     if
22     :: Active ->
23            if
24            :: nr != maximum ->
25                   out!two(nr);
26                   neighbourR = nr;
27            :: else ->
28                   /* max is the greatest number */
29                   assert(nr == N);
30                   know_winner = 1;
31                   out!winner(nr);
32            fi
33     :: else ->
34            out!one(nr)
35     fi
36
37  :: in?two(nr) ->
38     if
39     :: Active ->
40            if
```

Verification Model of Leader Election (Cont’d)

```
41 _          :: neighbourR > nr && neighbourR > maximum
42                   maximum = neighbourR;
43                   out!one(neighbourR)
44            :: else ->
45                   Active = 0
46            fi
47     :: else ->
48            out!two(nr)
49     fi
50   :: in?winner(nr) ->
51     if
52     :: nr != mynumber ->
53            printf(“MSC: LOST\n”);
54     :: else ->
55            printf(“MSC: LEADER\n”);
56            nr_leaders++;
57            assert(nrleaders == 1)
58     fi
59     if
60            know_winner
61            else -> out!winner(nr)
```

Verification Model of Leader Election (Cont’d)

```
62     fi;
63     break
64   od
65 }
66
67  init {
68    byte proc;
69    atomic {/* activate N copies of proc template */
70            proc = 1;
71            do
72            :: proc <= N ->
73                run node (q[proc-1], q[procpercentN],
74                   (N+I-proc)percentN+1);
75                proc++
76            :: proc > N ->
77                break
78            od
79     }
80  }
```

Conclusion

- Distinction between behavior and requirements on behavior (invariants, deadlock-detection, LTL formulae)
- Requirements and behavior s are checked for both their internal and their mutual consistency
- Design is revised until its critical correctness properties can be successfully proven. Then can refine the design decisions further toward a full systems implementation (PROMELA is not a full programming language - no data structures, for example)
- Can also simulate the design before the verification starts, to make sure that the design “seems” correct - no change for “vacuous” verification as in SMV.

Comment

- On-line information is very good:
  - lots of examples
  - nicely-written manual and on-line help
  - GUI-based input, xspin
  - more familiar, C-like syntax
- seems like no place where “black magic” is involved
- bring to everyone’s attention if and how you can trick the system into dealing with vacuous models