SMV Model Checker
CS 4271

Abhik Roychoudhury
http://www.comp.nus.edu.sg/~abhik

Copyright 2012 by Abhik Roychoudhury

SMV

- Symbolic Model Verifier
- Several versions exist, we will use Cadence SMV
  - http://www-cad.eecs.berkeley.edu/~kenmcmill/smv/
- Familiarize yourself via the tutorial at
  - http://www-cad.eecs.berkeley.edu/~kenmcmill/tutorial.ps
  - You should preferably use it in an online mode by trying out the
    examples, rather than offline reading.
- We will have a full case study in a later class.

Recap: the big picture

System to be built (Dream)

System Model (Rough Idea)

Properties to Satisfy (caution)

Checking Method (Automated)

Refine the model

Today’s lecture is a checking tool.
Yes, the tool comes before the method!

Before starting …

- If a property is false, a counter-example trace is generated.
  - Details of counter-example generation is not covered in our course.
  - We only present and discuss model checking as a yes/no decision
    procedure in class with no other output.
  - However, studying the counter-example trace is of utmost importance
    for detecting errors in your design, when you are using Cadence SMV as
    a validation tool.

SMV vs. SPIN

- LTL Property
- System Model
- SMV input model – more suitable for modeling
  - hardware / processors ...
- SMV model checker – check hardware / processors ....
  - Yes, with Counter-example trace
  - No, with Counter-example trace

Organization

- So Far
  - What is a Model?
  - ATC – Running Example
  - How to model such requirements
  - How to validate the models
    - Simulations,
    - Model-based testing,
    - Model Checking
  - Model Checkers
    - SMV
Modeling in SMV

- Can model state machines.
- States given by valuation of signals.
- How each signal changes is captured by individual assignment statements.
- Let us start with a simple combinational circuit; then we go to sequential circuits.

Example circuit

![Example circuit diagram](image)

A combinational circuit

```plaintext
module main(req1, req2, ack1, ack2) {
  input req1, req2 : boolean;
  output ack1, ack2 : boolean;
  ack1 := req1 & ~req2;
  ack2 := ~req1 & req2;
  serve: assert (req1 | req2) -> (ack1 | ack2);
}
```

Inputs and outputs

- Input signals come with finite types
  - Can assume any valuation within that type.
  - SMV has to try out all possible valuations of all input signals.
- Output signals are computed from input
  - In our combinational circuit, they are simple boolean formulae of inputs.

Verifying the circuit

- req1 = 1, req2 = 1, ack1 = 0, ack2 = 0
  - Combinational circuit; this state repeats forever.
  - A counter-example trace for `serve`.
- `serve` is a propositional property
  - For sequential circuits, we verify temporal properties specified in LTL.
    - Temporal properties were discussed earlier.

A slight modification

```plaintext
module main(req1, req2, ack1, ack2) {
  input req1, req2 : boolean;
  output ack1, ack2 : boolean;
  ack1 := req1;
  ack2 := ~req1 & req2;
  serve: assert (req1 | req2) -> (ack1 | ack2);
}
```
A slight modification

- \(\text{ack1}\) is set whenever \(\text{req1}\) is set
- If \(\text{req1}\) is always set
  - This will starve \(\text{ack2}\)
- Need a bit of memory to remember for how long \(\text{req1}\) is set
  - A sequential circuit …

Modeling sequential circuits

- module main(req1, req2, ack1, ack2)
  
  - \{ \}
  - input req1, req2 : boolean;
  - output ack1, ack2 : boolean;
  - bit : boolean; // a latch has been added
  - 
  - next(bit) := ack1;
  - ack1 := bit \& req1 \& \neg req2 \& \neg req1;
  - ack2 := bit \& req2 \& \neg req2 \& \neg req1;
  - \}

Assignment statement in SMV

- Assignments for \(\text{ack1}\) and \(\text{ack2}\) signals are conditional.
- SMV also allows direct usage of (see manual)
  - If-then-else statement
  - Case statement
- Assignments may involve the \texttt{next} operator
  - Value of a signal \(s\) in the next clock cycle is computed using the value of various signals (possibly including \(s\)) in the current cycle.

Properties to be proved

- Cadence SMV allows user to specify properties in LTL.
- Properties are distinguished via \texttt{assert} keyword.
- There is an option to verify all LTL properties described in your spec. file.
- You can also assume properties to prove other properties
  - More about this later …

Starvation of low priority \(\text{req1}\)?

- module main(req1, req2, ack1, ack2)
  
  - \{ \}
  - input req1, req2 : boolean;
  - output ack1, ack2 : boolean;
  - bit : boolean; // if not initialized
  - 
  - next(bit) := ack1;
  - ack1 := bit \& req1 \& \neg req2 \& \neg req1;
  - ack2 := bit \& req2 \& \neg req2 \& \neg req1;
  - 
  - no_starve: assert G F (\neg req2 \&\& \neg ack2);
  - \}
Exercise

- Draw the underlying state machine for this SMV specification.
- Verify the non-starvation property manually using this state machine.
- GF denotes infinitely often
- GF (¬req2 | ack2) denotes infinitely often
  - Either req2 is not set,
  - Or ack2 is set.

Traffic Control

A Traffic Light Controller

```
module main(N_s, S_s, E_s, N_g, S_g, E_g)

input  N_s, S_s, E_s : boolean;
output N_g, S_g, E_g : boolean;
...
```

Inputs/outputs of controller

- \(N_s = 1\) (similarly \(S_s, E_s\))
- Traffic going North is sensed
- \(N_g = 1\) (similarly \(N_g, E_g\))
- Green light allowing traffic to go North.

Consider all possible values of the input variables.

Internal variables of controller

- \(N_r, S_r, E_r\)
  - Latch sensor outputs from the three directions
  - Requests sensed, but not served.
- NS_lock
  - Convenient way of disabling \(E_g\)
  - Set exactly when traffic is enabled in North and/or South directions.

Initializations

- \(N_g, E_g, S_g, N_r, E_r, S_r\)
  - All green lights are initially 0
- NS_lock
  - Initially 0.
- Use the init command
  - init(N_g) := 0;

The full spec. comes with the Cadence SMV distribution
Look under ./doc/smv/examples
Let us take a quick look at a few salient issues.
Properties
- safety: assert $G \neg(E_s \land (N_s \lor S_s))$;
- $N_{\text{live}}$: assert $G (N_s \rightarrow F N_g)$;
- $S_{\text{live}}$: assert $G (S_s \rightarrow F S_g)$;
- $E_{\text{live}}$: assert $G (E_s \rightarrow F E_g)$;
- Once again these are LTL properties.
- The actual “liveness” can only hold if drivers do not wait forever at a green light.
- But, this is something we are not verifying.
- We assume the humans to co-operate.
- Alternatively, traffic may always be coming from an enabled direction, starving other directions?

Verification
- We instruct SMV to explore only fair paths.
  - using $N_{\text{fair}}, S_{\text{fair}}, E_{\text{fair}}$
  - prove $N_{\text{live}}, S_{\text{live}}, E_{\text{live}}$
  - assume $N_{\text{fair}}, S_{\text{fair}}, E_{\text{fair}}$
- In general, we can instruct SMV to assume any arbitrary temporal property
  - Corresponds to implementation details which are not modeled in SMV,
  - but are required for verification.
- A very useful feature, from my personal experience!
  - Use of implementation assumptions which are temporal properties!

Exercises
- Try out the traffic light controller verification.
  - Fix the counter-example(s) obtained.
- Try out an alternate modeling where $NS_{\text{lock}}$ is simply defined by the eqn
  - $NS_{\text{lock}} = N_g \lor S_g$
- Look under ./doc/smv/examples/traffic
  - contains other versions of the controller

Overall structure
```c
MODULE main() {
  p1 : processor(a.GRANT1, s.RESP);
  p2 : processor(a.GRANT2, s.RESP);
  s : slave(a.GRANT1, a.GRANT2);
  a : arbirter(p1.REQUEST, p2.REQUEST);
  mutex: assert $G(\neg(a.GRANT1 \land a.GRANT2))$;
  nostarve1: assert $G(p1.REQUEST \rightarrow F a.GRANT1)$;
  nostarve2: assert $G(p2.REQUEST \rightarrow F a.GRANT2)$;
  using mutex prove nostarve1, nostarve2;
  assume mutex;
}
```
Advantages
- Can now under-specify the arbiter.
- Advantages
  - No need to worry about implementation details.
  - Verification not dependent on specific arbitration policy.
  - Can thus even deliberately under-specify!

```plaintext
MODULE arbiter(REQUEST1, REQUEST2)
{
  GRANT1, GRANT2 : boolean;
  next(GRANT1) := case{
    REQUEST1 : {0,1};
    default: 0;
  }
  next(GRANT2) := case{
    REQUEST2 : {0,1};
    default: 0;
  }
}
```

Assigning Signals
- Within a module
  - A signal can be assigned through “default” block nestings as shown in traffic light controller
  - Or, a less error-prone method is use a switch statement (called “case” in SMV).
  - This is illustrated in the following example.

Example: ABP
- Both channels are lossy
  - Msg / Ack may be lost
  - Fairness is needed for progress of the protocol.
  - Msg / Ack cannot be dropped forever.
  - Sender resends message until an ack with the expected bit is received.
  - Receiver resends previous ack until a message with the expected bit is received.

Example: ABP
- Alternating bit protocol
  - Sender
  - Receiver
  - Data_Chan
  - Ack_Chan
  - Sender sends msg with bit 0
  - Receiver sends ack with bit 0
  - Sender sends msg with bit 1
  - Receiver sends ack with bit 1

Protocol Architecture
Protocol Architecture

module main
{    S: process sender(ack_chan.output);
    R: process receiver(msg_chan.output);
    msg_chan: process one-bit-chan(S.msg);
    ack_chan : process one-bit-chan(R.ack);

    init(S.msg) := 0;
    init(R.expect):= 0; init(R.ack) := 1;
    init(msg_chan.output) := 1;
    init(ack_chan.output) := 1;

delivery: assert G(S.status = sent -> F R.status = received)
using fair_chan prove delivery assume fair_chan;
}

Channel

module one-bit-chan(input)
{
    output: boolean;

    next(output) := {input, output};

    fair_chan : assert G
    (input = 0 -> F output = 0) &
    G(input = 1 -> F output = 1)
}

Sender

module sender(ack)
{
    status : {send, sent};
    msg: boolean;  // the control bit

    init(status) := send;
    init(msg) := 0;

    next(status) := case{
        status = send & ack = msg : sent;
        1 : send;
        
        next(msg) := case{
            status = sent ! msg;
            1 : msg;
        }
    }

Receiver

module receiver(bit)
{
    status : {receiving, received};
    ack, expect : boolean;

    init(status) := receiving;

    next(status) := case{
        bit = expect & status = receiving: received;
        1 : receiving;

        next(ack) := case{
            status = received: bit;
            1 : ack;
        }

        next(expect) := (status = received) ? ! expect : expect;
    }

Some key points about ABP

- Illustrates the alternate modeling style
- Transition of each signal modeled by a separate case statement.
- No use of "default" nestings.
- Illustrates assume-guarantee proofs
- Assumptions about channel are crucial for proving data delivery.
- These assumptions refer to impl. and are hence not dispensed using SMV.
- More about this issue in the revision hour!

Some points about the properties verified

- Data values are not modeled.
- Cannot verify properties like:
  - If a message with value x is sent, the same uncorrupted message is eventually received.
  - What is the domain of x?
  - If it is unbounded, what to do?
So far …

- Basics of modeling
  - Includes details of SMV syntax
- Toy examples
  - ABP, Traffic Light Controller
- In the remaining time
  - Modeling exercises in SMV

Ex 1: Modeling a Counter

- A normal three bit counter can also be described as a mod 8 counter since its contents vary from 0 to 7 by following the sequence
  \[ 0 \rightarrow 1 \rightarrow \ldots \rightarrow 7 \rightarrow 0 \rightarrow 1 \ldots \]
- Construct the Kripke Structure for a mod 7 counter whose contents vary from 0 to 6 by following a similar sequence.
- Encode the mod 7 counter in SMV. Use only boolean variables.

More Exercises

- Model a shift register in SMV
  - Prove that a signal when fed from left goes out eventually through the right end.
- Model the crude mutual exclusion protocol involving “turn” studied earlier in our lectures.
  - Prove mutual exclusion.

Right shifts only

```plaintext
MODULE main(left, inleft)
{
  input left : boolean;
  input inleft : boolean;
  bit0 : cell(left, inleft);
  bit1 : cell(left, bit0.content);
  bit2 : cell(left, bit1.content);
  bit3 : cell(left, bit2.content);
  left_live : assert G ( ( ( G left ) & inleft ) -> F bit3.content);

  prove left_live;
}
```

Each cell

```plaintext
MODULE cell(left, lval)
{
  content: boolean;
  init(content) := 0;
  next(content) := case{
    left : lval;
    1 : content;
    };
}
```
Left and right shifts

- Need to have more input variables
- What do we do when there is input to be fed from each side?
  - Can we then prove the liveness properties for each direction of shift?

Shift Register

```plaintext
MODULE main(left, right, inleft, inright)
{
    input left, right: boolean;
    input inleft, inright : boolean;
    bit0 : cell(left, right, inleft, bit1.content);
    bit1 : cell(left, right, bit0.content, bit2.content);
    bit2 : cell(left, right, bit1.content, bit3.content);
    bit3 : cell(left, right, bit2.content, inright);
    left_live : assert G ( ( ( G left ) & inleft ) -> F bit3.content);
    right_live : assert G( ( (G right) & inright ) -> F bit0.content);
    prove left_live, right_live;
}
```

Each cell

```plaintext
MODULE cell(left, right, lval, rval)
{
    content: boolean;
    init(content) := 0;
    next(content) := case{
        left  : lval;
        right : rval;
        1     : content;
    };
}
```

A Concurrent Program

```plaintext
PO || PI

l0: while true do
l1: wait(turn = 0);
l2: turn := 1;
l3: endwhile

m0: while true do
m1: wait(turn = 1);
m2: turn := 0;
m3: endwhile
```

Models a crude protocol for entry/exit to critical section without modeling the critical section itself.

SMV modeling

```plaintext
MODULE main;
{
    pc0 : { l0, l1, l2, l3 };
    pc1 : { m0, m1, m2, m3 };
    turn : boolean;
    schedule : boolean;
    schedule := {0, 1};
    init(turn) := 0;
    next(turn) := case{
        (schedule = 0 & pc0 = l2) : 1;
        (schedule = 1 & pc1 = m2) : 0;
        1 : turn;
    };
    init(pc0) := l0;
    next(pc0) := case{
        (schedule = 0 & pc0 = l0) : l1;
        (schedule = 0 & pc0 = l1 & turn = 0 ) : l2;
        (schedule = 0 & pc0 = l2) : l3;
        (schedule = 0 & pc0 = l3) : l0;
        1 : pc0;
    };
    init(pc1) := m0;
    next(pc1) := case{
        (schedule = 1 & pc1 = m0) : m1;
        (schedule = 1 & pc1 = m1 & turn = 1) : m2;
        (schedule = 1 & pc1 = m2) : m3;
        (schedule = 1 & pc1 = m3) : m0;
        1 : pc1;
    };
    mutual_excl: assert G( !(pc0 = l2 & pc1 = m2));
    prove mutual_excl;
}
```

SMV modeling