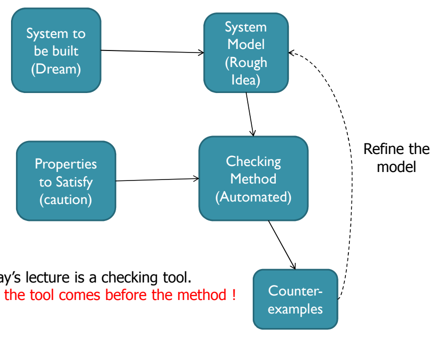
Model Checking Tools (SMV)

Abhik Roychoudhury
CS 4271
National University of Singapore

SMV

- Symbolic Model Verifier
- Several versions exist, we will use Cadence SMV
  - http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/
- Familiarize yourself via the tutorial at
  - http://www-cad.eecs.berkeley.edu/~kenmcmil/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
Counter-examples

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

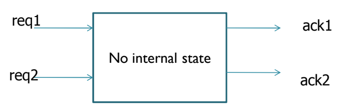
Before starting …

- If a property is false, a counter-example trace is generated.
  - Details of counter-example generation algorithm 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.

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



A combinational circuit

- 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

- 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

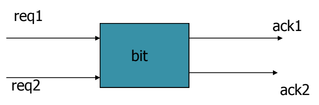
- ack1 is set whenever
  - req1 is set
- If req1 is always set
  - This will starve ack2
- Need a bit of memory to remember for how long req1 is set
  - A sequential circuit …

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 & ~req2 : req1;
- ack2 := bit ? req2 : req2 & ~req1;
- }

Block diagram



Assignment statement in SMV

- Assignments for ack1 and ack2 signals are conditional.
- SMV also allows direct usage of (see manual)
  - If-then-else statement
  - Case statement
- Assignments may involve the 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 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 …

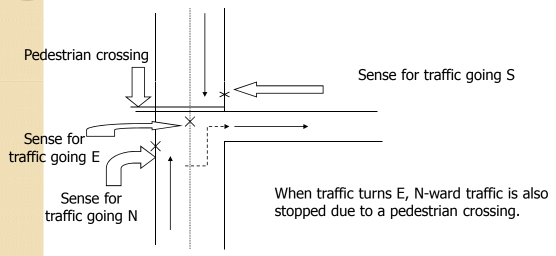
Starvation of low priority req.?

```
module main(req1, req2, ack1, ack2)
{
  input req1, req2 : boolean;
  output ack1, ack2 : boolean;
  bit : boolean;      // not initialized

  next(bit) := ack1;
  ack1 := bit ? req1 & ~req2 : req1;
  ack2 := bit ? req2 : req2 & ~req1;

  no_starve: assert  G F (~req2 | ack2);
}
```

Exercise

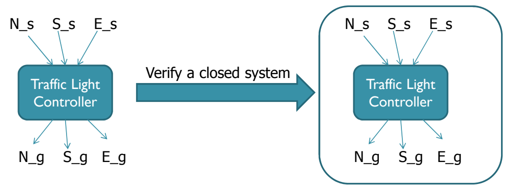
- Draw the underlying state machine for this SMV specification.
- Verify the non-starvation property manually using this state machine.
  - Formal description of temporal properties not discussed.
  - GF denotes *infinitely often*
    - *GF (~req2 | ack2) denotes infinitely often*
      - *Either req2 is not set,*
      - *Or ack2 is set.*

Traffic Control


Pedestrian crossing
Sense for traffic going S
Sense for traffic going E
Sense for traffic going N
When traffic turns E, N-ward traffic is also stopped due to a pedestrian crossing.

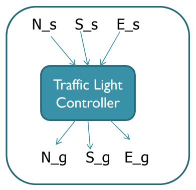
A Traffic Light Controller

- 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;

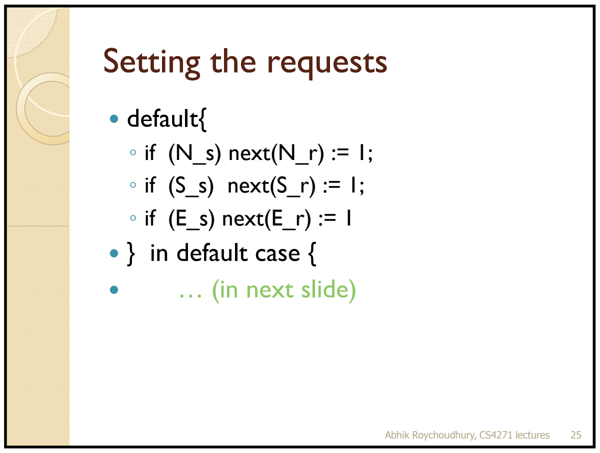
- *How to update the values of signals?*

The full spec.

- Comes with the Cadence SMV distribution
  - Look under **./doc/smv/examples**
- Let us take a quick peek to get a full picture of the story.

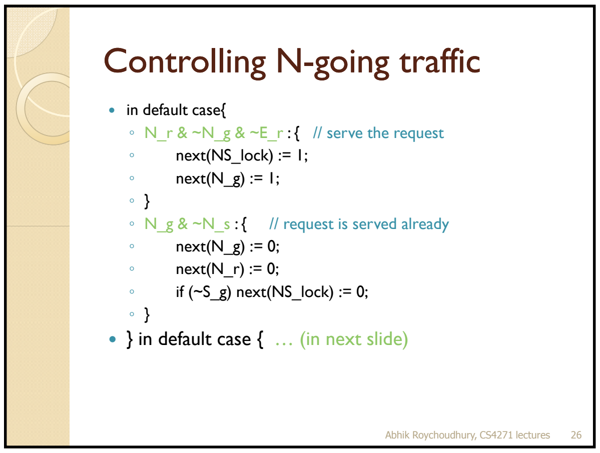
Single Assignment rule

- default {block1}
- in {block2}
  - Assignments in block2 get priority.
  - Only one assignment to a signal is to be active at a time.
  - The "default" keyword allows a nice nesting of blocks rather than enumerating all cases explicitly for each signal.



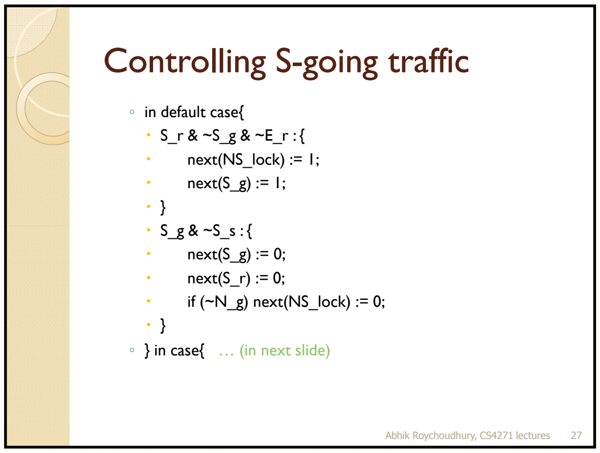
Setting the requests

- default{
  - if (N_s) next(N_r) := 1;
  - if (S_s) next(S_r) := 1;
  - if (E_s) next(E_r) := 1
- } in default case {
- … (in next slide)



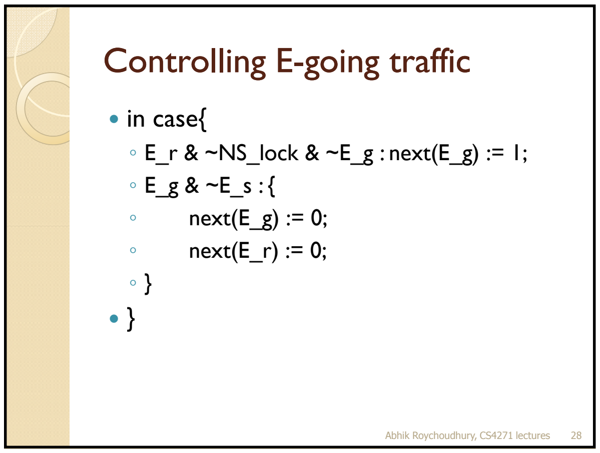
Controlling N-going traffic

- in default case{
  - N_r & ~N_g & ~E_r : { // serve the request
  - next(NS_lock) := 1;
  - next(N_g) := 1;
  - }
  - N_g & ~N_s : { // request is served already
  - next(N_g) := 0;
  - next(N_r) := 0;
  - if (~S_g) next(NS_lock) := 0;
  - }
- } in default case { … (in next slide)



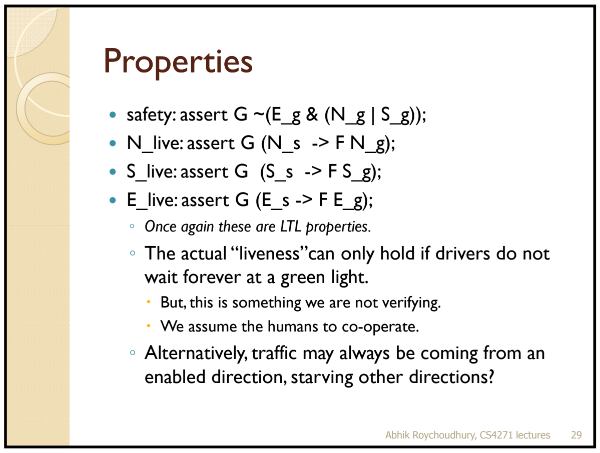
Controlling S-going traffic

- in default case{
  - S_r & ~S_g & ~E_r : {
  - next(NS_lock) := 1;
  - next(S_g) := 1;
  - }
  - S_g & ~S_s : {
  - next(S_g) := 0;
  - next(S_r) := 0;
  - if (~N_g) next(NS_lock) := 0;
  - }
- } in case{ … (in next slide)



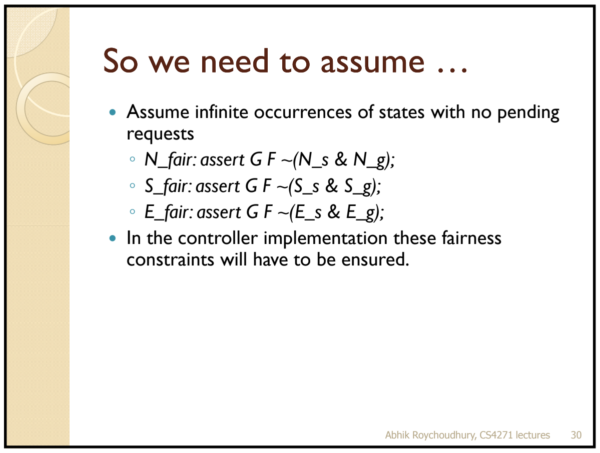
Controlling E-going traffic

- in case{
  - E_r & ~NS_lock & ~E_g : next(E_g) := 1;
  - E_g & ~E_s : {
  - next(E_g) := 0;
  - next(E_r) := 0;
  - }
- }



Properties

- safety: assert G ~(E_g & (N_g | S_g));
- N_live: assert G (N_s -> F N_g);
- S_live: assert G (S_s -> F S_g);
- E_live: assert G (E_s -> 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?



So we need to assume …

- Assume infinite occurrences of states with no pending requests
  - *N_fair: assert G F ~(N_s & N_g);*
  - *S_fair: assert G F ~(S_s & S_g);*
  - *E_fair: assert G F ~(E_s & E_g);*
- In the controller implementation these fairness constraints will have to be ensured.

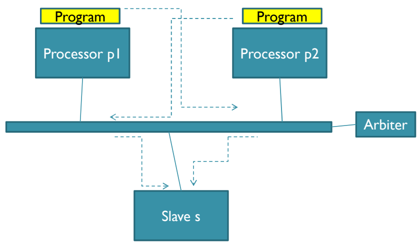
Verification

- We instruct SMV to explore only fair paths.
  - *using N_ fair, S_ fair, E_ fair*
  - *prove N_live, S_live, E_live*
  - *assume N_fair, S_fair, E_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_lock is simply defined by the eqn
  - *NS_lock := N_g | S_g*
- Look under **./doc/smv/examples/traffic**
  - contains other versions of the controller

More on assumptions



Model and verify the bus access protocol using SMV

Overall structure

```
MODULE main() {
   p1 : processor(a.GRANT1, s.RESP);
   p2 : processor(a.GRANT2, s.RESP);
   s : slave(a.GRANT1, a.GRANT2);
   a : arbiter(p1.REQUEST, p2.REQUEST);

   mutex: assert G( ~(a.GRANT1 & a.GRANT2) );
   nostarve1: assert G( p1.REQUEST -> F a.GRANT1 );
   nostarve2: assert G( p2.REQUEST -> 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!

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

Composing modules

- Your design consists of a number of components
  - Each component is a module
  - Default composition of modules is synchronous.
  - Asynchronous composition is enabled by declaring each component as process in the main module.

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.

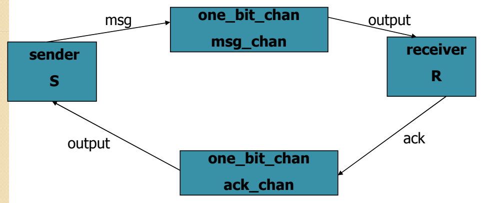
Example: ABP

- 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

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.

Protocol Architecture

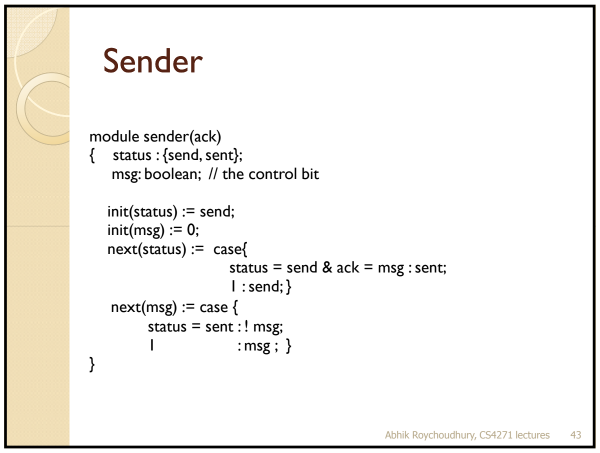


Protocol Architecture

```
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;
}
```

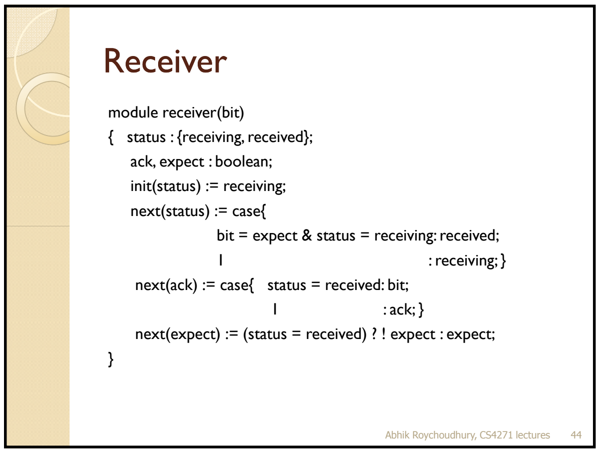
Channel

```
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)
}
```



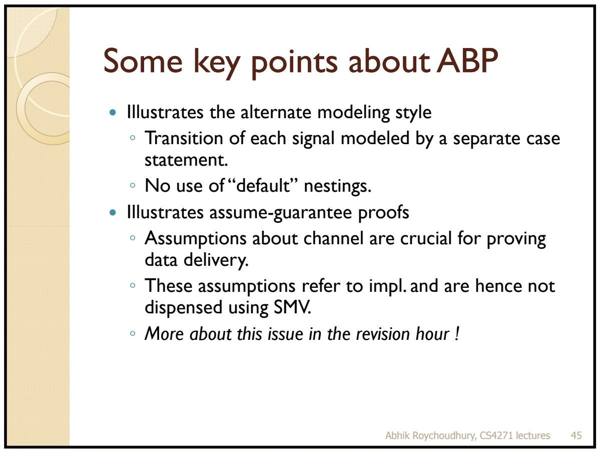
Sender

```
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 ;  }
}
```



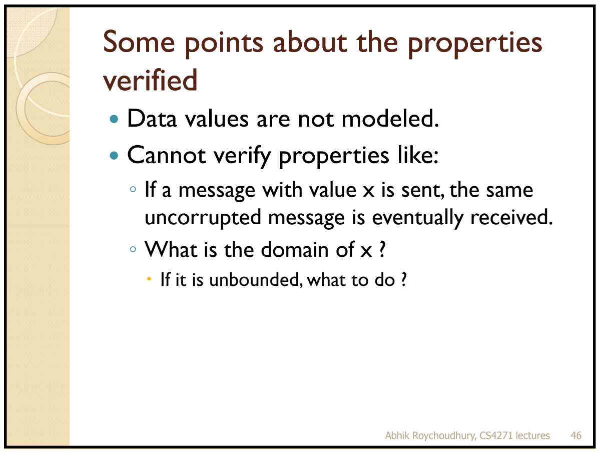
Receiver

```
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;
}
```



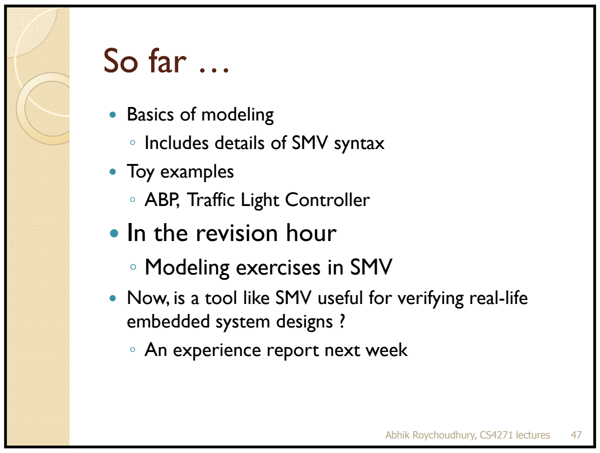
Some key points about ABP

- 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 revision hour
  - Modeling exercises in SMV
- Now, is a tool like SMV useful for verifying real-life embedded system designs ?
  - An experience report next week