Model Checking Tools (SMV)

Abhik Roychoudhury
Department of Computer Science
National University of Singapore

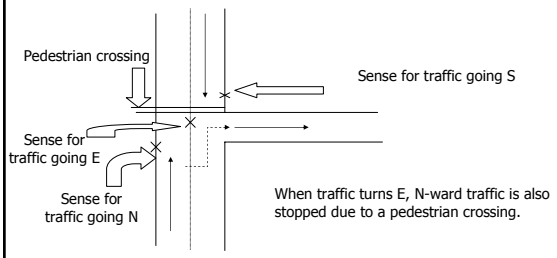
SMV

- 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 two large case studies one today and one in the next class.

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 presented 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.

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.

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**

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
  - 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);
- The liveness properties can only hold if drivers do not wait forever at a green light
  - Otherwise the sensors will remain set.

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 !

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

Composing modules

- We looked at a monolithic controller just now.
- Your design may consist of a number of components
  - Each component is a module
  - 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.

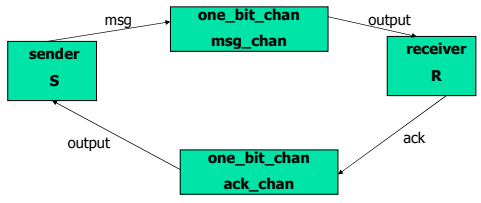
Example: ABP

- 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 later!*

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 ?
      - **At least can we specify the property ??**

So far …

- Basics of modeling
  - Includes details of SMV syntax
- Toy examples
  - ABP, Traffic Light Controller
- Now, is a tool like SMV useful for verifying real-life designs ?
  - An experience report now !!

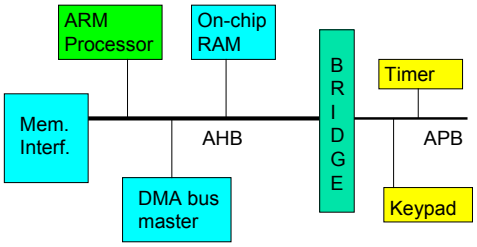
Experience Report on using SMV on a real-life bus protocol

Abhik Roychoudhury
Department of Computer Science
National University of Singapore

A real-life application

- Verifying the AMBA Advanced High-performance (AHB) bus protocol in ARM processors.
- Cannot discuss all the details, but it gives a feel of:
  - the practical usage of model checking
  - Simplifications involved in modeling
    - *The original AMBA AHB document runs to 60 pages, see the handout AMBA.pdf (Chapter 3) from IVLE lesson Plan*
    - More in-depth on this issue in the next class.
  - How subtle errors can creep into real designs !!

Bus-based SoC design



Bus Protocols

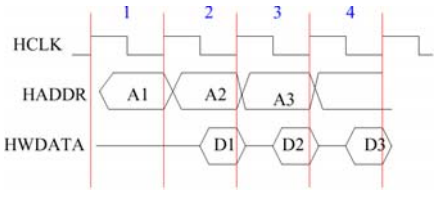
- Popularity of bus-based SoC designs necessitate the verification of bus protocols.
- Different from testing/validating the cores.
- SoC Bus Protocols often involve advanced features for high speed data transfer, leading to corner cases
  - **Pipelining**
  - **Wait Cycles**
  - **Split Transfers**
- Case study: AMBA AHB protocol from ARM.

Bus architecture

- Several masters and slaves are connected to AHB.
- An arbiter decides which master will transfer data.
- Data is transferred from a master to a slave in bursts.
- Any burst involves read/write of a sequence of addresses.
- The slave to service a burst is chosen depending on the addresses (decided by a decoder).
- AHB is connected to APB via a bus bridge.
- Let us study the transfer features of AHB protocol

Pipelining within a burst

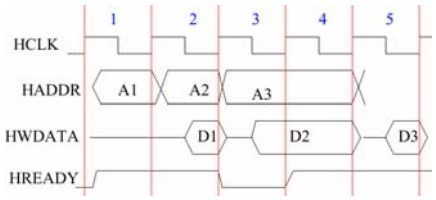
Address and data of consecutive transfers are transmitted in same clock cycle,



Wait cycles

Slave may not be ready to service request.

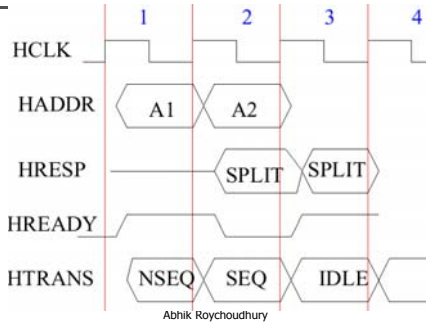
Inserts Wait cycle(s) by de-asserting HREADY



Split response

- If the slave thinks it may take too long to service a request
  - Insert too many wait cycles, or better …
  - Suspend the request (via SPLIT response)
    - Arbiter informed via bus
    - Corresponding master is suspended
    - Other masters can access the bus
    - Later slave informs arbiter that it is ready.

Transfer Cancellation



Split response

- Cycle i
  - Master M drives address A on bus
- Cycle i+1
  - Slave S thinks it can take too long to service A, issues SPLIT response
  - Arbiter snoops on SPLIT response, records **current master**.
  - Issued in (i+1, i+2) to kill already initiated transfers
- Cycle i+2
  - Arbiter disables bus access to **current master**. Others can now access the bus.

Mechanism of split

- Slave issues split to master m.
- Arbiter records that m has been split
  - Even if m requests bus access during this period, it is not even considered by the arbitration algorithm
- Slave informs arbiter later that it can now service m.
- Arbiter now enables potential access
  - m may still not get bus access immediately.

Model Checking

- Developed a formal specification of the protocol.
- Various kinds of components
  - > 1 Masters
  - Slave(s)
  - Arbiter
- Bus interface of each component modeled as a finite state machine.
- Protocol = Synchronous composition of these FSMs
- Model check temporal properties (e.g. non-starvation) using Cadence SMV tool.

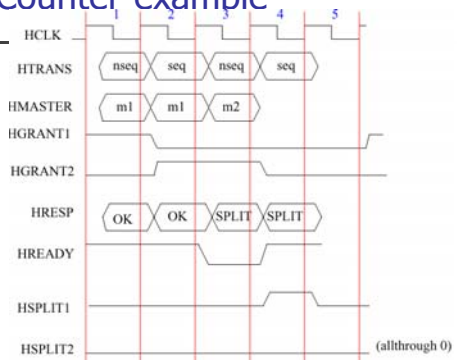
Checking for no-starvation

- Property in Temporal Logic LTL
  - G ($HBUSREQ_m \Rightarrow F\ HGRANT_m$)
  - $HBUSREQ_m$: Master m requests bus access
  - $HGRANT_m$: Master m granted access by arbiter
- Do not consider starvation introduced by incorrect implementation of the cores, e.g.
  - The arbiter is unfair.
  - A slave is not guaranteed to service every split transfer eventually.

Checking for no-starvation

- Check for starvation introduced by corner cases of the protocol
- However, do not fix any implementation of the cores
  - Do not code up a fair arbiter.
  - Do not code up a slave which is guaranteed to eventually service split responses.
- Use assertions to denote these features
  - **using fair, slave_live prove no_starve;**
  - **assume fair, slave_live;**
- Restrict model checking to executions satisfying these assertions (possible in Cadence SMV).

Counter-example



Conclusion

- Case study in model checking of a bus protocol with non-trivial data transfer features
- Interaction of these features (Pipelining + Splits) leads to a corner case starvation scenario.
  - Essentially an incompleteness in the spec.
- Source of starvation was suspected via human understanding of the protocol.

Conclusion

- Model checking effort was taken up to:
  - verify our suspicion.
  - Find a detailed counter-example trace.
- References
  - SMV Code (feel free to take a sneak peek)
    - http://www.comp.nus.edu.sg/~abhik/software/amba/ahb.smv
  - A relevant paper
    - http://www.comp.nus.edu.sg/~abhik/pdf/date03.pdf
- **Discussion of Assignment 1 at this stage.**

A Full Case Study using SMV

Abhik Roychoudhury
Department of Computer Science
National University of Singapore

So far …

- **Basics of modeling**
  - Includes details of SMV syntax
- **Toy examples**
  - ABP, Traffic Light Controller
- **Motivational practical applications**
  - AMBA AHB protocol
- We need to model/verify a medium sized problem completely to get the feel

Case Studies

- Many well-publicized successes of Model Checking in the verif. of processors / cache coherence protocols
  - Encore Gigamax Cache coherence protocol
  - IEEE FutureBus+ standard
  - T9000 virtual channel processor
- Our case study is a slightly more software centric coherence protocol, one for distributed file systems

Note of Caution

- Following slides use CMU SMV syntax which is a bit different from Cadence SMV syntax.
  - This has been done to ensure uniformity with the reading material.
  - The syntax differences are however minimal e.g.
    - MODULE x(…) instead of MODULE x(…)
    - VAR {

Before starting …

- SMV employs symbolic model checking
  - More space efficient than the explicit state MC algorithm which proceeds by graph search.
  - Uses a data structure called Binary Decision Diagrams for compact internal representation of state space.
  - This is covered in your other courses (e.g. Verification course by Prof. Deepak D'Souza), and hence not covered here to avoid repetition.
  - But you do not need to understand symbolic MC for modeling and verifying using SMV.

Success of Model Checking

- Primarily in hardware verification
- Routinely used for processor verification (or modules of it) in Intel etc
- Employed by CAD giants and processor design companies
  - Cadence, Intel, Motorola …

Verification or Bug Hunting

- Model Checking verifies only the design, not the implementation
- So, MC is more of a bug detection mechanism
  - But "Automated" bug detection
- This line will be more acceptable to practitioners than presenting it as a formal verification technique.

Reference

- A Case Study in Model Checking Software Systems
  - Jeanette Wing and M. Vaziri-Farhana
  - FSE 1995, CMU Tech report CMU-CS-96-124
  - Another version in "Science of Computer Programming", v20, 1997, 273-299.

File System Cache coherence

- Several clients and servers
- Each file is authorized by a single server
- Clients cache files on obtaining them from the server
- Clients talk directly only to the server, not to other clients.

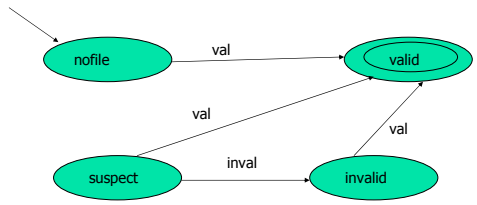
The Cache Coherence Problem

- A client can have a copy of a file, but
  - It may not be sure that it is the latest copy
  - It knows that it is an invalid copy
- The client will
  - Request for a validation from the server
  - Request a fresh copy

Global State

- Composition of Client & Server's local states
  - Client
    - Data in files
    - Belief about each file --- valid/invalid
  - Server
    - Master copy of files
    - Belief about each client's local copy of each file
  - Are the server and client copies in synch. ?
    - Not really a global knowledge about the copies.
    - Pair-wise knowledge across client and server.

Client's belief about its copy



Client receives val or inval from server

Client – State Variables

- out – request to the server.
  - This can be a request to
    - Fetch a file copy
    - Validate a file copy
- belief
  - the current belief of the client about the status of its cached copy.

Simplification # 1

- Do not model contents of shared files.
  - The file contents does not affect the coherence mechanism.

Client – State Variables

- Possible values of the "belief" variable
  - Valid
    - the client knows that the cache copy is valid
  - Invalid
    - the client knows that the cached copy is invalid
  - Suspect
    - the client is not sure of the status of the cached copy
  - Nofile
    - client does not have a cached copy

Simplification # 2

- In reality
  - The client has a status for every shared file
- We model
  - Only one shared file, since the coherence issues of each file is independent
    - Hence modeling one server is enough.

Client SMV description

```
MODULE client(input)
VAR
      out: {0, fetch, validate};
      belief: {valid, invalid, suspect,nofile};
ASSIGN
       …
```

- Receives input from server
- out and belief are state variables

A peculiarity of synch. comp.

- One possible value of out is 0
  - Signifies nothing being sent out.
  - In synch. comp., each variable is set a value in every time unit.
    - Different from event triggered system modeling.

Client- Trans. Rel. in SMV

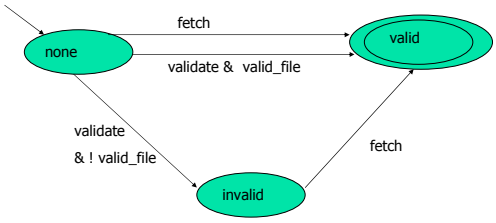
```
ASSIGN
      init(out) := 0;
      next(out) := case
               (belief = nofile) : fetch;
               (belief = invalid) : fetch;
               (belief = suspect): validate;
               1: 0;
       esac
…
```

Client – Trans. Rel. in SMV

```
init(belief) := nofile, suspect;
next(belief):=
case
  (belief=nofile)&(input=val): valid;
  (belief=suspect)&(input=val):valid;
  (belief=suspect)&(input=inval): invalid;
  (belief=invalid)&(input=val): valid;
    1   : belief
esac
```

- `belief` is updated based on current value, and input from server.

Server – State Diagram



Server receives fetch, validate from Client.

Server – State Variables

- belief : the belief of the server about the status of a cached copy in a particular client
  - Valid
  - Invalid
  - None : Server has no knowledge (the client might or might not have a cached copy)
- For each client i, the server should model a belief[i]
  - But we will simplify further and model only one client !!

Server – State Variables

- out : the output of the server to the client
  - val: indicates to the client that its cached copy is valid
  - inval : indicates to the client that its cached copy is invalid
  - 0 : default output. Ignored by client.

Simplification # 3

- Server has another state variable valid_file
  - Boolean variable
- Models updates to the server by other clients
  - Which need not be modeled explicitly
- Suppose C.belief = suspect
  - Then C sends a validate message to Server
- If Server has received a file update message by another client C′ by now
  - Then server deems cached copy of C as invalid
  - Else …

Modeling other clients

- Whether Server receives a new update from another client is modeled by the variable valid_file
  - Set non-deterministically since we do not model the other clients explicitly
- Modeled one client, and exploited the star topology to implicitly model the effects of other clients on the server
  - Drastically cuts down the state space.

Simplification #4 (uncommon)

- Our modeling considers only finite traces
  - "Final" states in Client and Server
  - Restricted modeling of a single session
  - But can be used for verifying invariant properties
    - AG(boolean formula of atomic propositions)
  - If we still find a violation of such properties they would have occurred in the generic modeling with infinite execution traces as well
    - The finite execution traces modeled are possible prefixes of the actual traces if the system was modeled in details

System description in SMV

```
MODULE main
VAR
      Client : client(Server.out);
      Server: server(Client.out);
```

- Client module already presented
- Let us look at the Server.

Server : SMV description

- MODULE server(input)
- VAR
  - out : {0, val, inval};
  - belief : {none, valid, invalid };
  - valid_file : boolean;
- ASSIGN
  - valid_file := {0,1};
  - …

Server's Transition Relation

```
init(belief) := none;
next(belief):=
case
  (belief=none)&(input=fetch): valid;
  (belief=none)&(input=validate)&valid_file:valid;
  (belief=none)&(input=validate)& !valid_file:
                                        invalid;
  (belief=invalid)&(input=fetch): valid;
    1  : belief
esac
```


Server's Transition Relation

```
init(out) := 0;
next(out):=
case
  (belief=none)&(input=fetch): val;
  (belief=none)&(input=validate)&valid_file:
                                        val;
  (belief=none)&(input=validate)& !valid_file:
                                        inval;
  (belief=invalid)&(input=fetch): val;
    1  : 0
esac
```


Properties to Verify # 1

- If client C believes cached copy of file f is valid, it does not go to server
  - In this case, server also should believe that C's cached copy of f is valid
  - AG (C.belief=valid $\Rightarrow$ S.belief = valid)
    - Verified to be true by CMU SMV

Property # 2

- We can also check
  - AG(S.belief = valid ⇒ C.belief = valid)
  - Otherwise, client might sometimes unnecessarily go to the server for validation
    - Inefficiency : additional traffic from client to server !!
  - SMV produces a counter-example.

SMV Counter-example

- **state 1.1**
- C.out =0, C.belief = nofile
- S.out =0, S.belief =none, S.valid_file = 0
- **State 1.2**
- C.out = fetch
- **State 1.3**
- S.out = val, S. belief = valid

SMV counterexample

- If client C does not have a copy of a shared file (cache miss), it requests from the server via fetch
- Server S sends a fresh copy and updates its belief about the status of cached copy at C
- Due to the transit delay between S and C, the client still has not updated its belief, but it will do in a few steps
  - This leads to the counter-example
  - Not a cause of concern in terms of additional traffic from the client to the server.

Some useful simplifications …

- … employed in today's case study
  - Do not model data of the data items
    - Files in this case !
  - Model only a single data item
    - Hence model a single server
  - Model only one client
    - Other clients modeled implicitly by considering their effect on the server.

Class Practice

P0 || P1

- 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

```
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;
  };
```

SMV modeling

```
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;
}
```


More modular design

- Do not specify P0, P1 separately
  - They are instances of the same process specification.
  - Asynchronous composition of the process instances required.
  - Use the "process" keyword.

Class Practice (from an old exam question set at NUS)

Consider a traffic light controller which initially shows green light. It senses traffic data every second. Thus, starting from time=0, the traffic is sensed at time = 1 sec, 2 sec, and so on.
If there is no traffic movement, the light turns from green to yellow.
If there is traffic movement, the light stays green, but it can stay green for a maximum of 3 seconds at a stretch.
The light always stays yellow for exactly one second after which it turns red. Once the light is red, again traffic is sensed every second. If there is traffic, then the light becomes green after staying red for a minimum of 2 seconds. If there is no traffic, the light eventually becomes green, after staying red for 3 seconds.

Class Practice

- Try to model the system as a Kripke Structure and/or SMV program.
  - What aspects of the informal requirements cannot be modeled?
  - Is it enough to just model the controller, for modeling/verifying the system in SMV?

Class Practice (from an old exam question set at NUS)

1. Consider two *parallel* processes which communicate via matching actions shown below. In the figure, some of the transitions are labeled. Any transition $s \rightarrow s'$ which is not labeled denotes an internal action; it is always enabled once control reaches s. But any transition $s \xrightarrow{in(a)} s'$ ($s \xrightarrow{out(a)} s'$) can only take place together with another process making a transition labeled with $out(a)$ ($in(a)$). In other words, such transitions denote a handshake between two processes. At any time step, any process can make (a) an internal action or (b) an input/output action provided the other process makes a matching output/input action. If the other process is not ready to make such a matching move, and there are no internal moves to make, then a process remains blocked. *At any time step, all the processes which are not blocked execute an action, that is, the processes are running in parallel.*

SENDER RECEIVER

Model the example in SMV's input language. Your modeling should preferably be modular, that is, you should not put all the code into a main module.

Class Practice

```
module main()
{
   S: myproc(R.label);
   R: myproc(S.label);
}

module myproc(msg)
{
   label: {a, none};
   state: {initial, busy};

   init(state) := initial;
   next(state) := case{
       state = initial & label = a & msg = a : busy;
       state = busy : initial;
       1    :  state;
   };
   label := case{
       state = initial : {a, none};
       state = busy    : none;
   };
}
```