Introduction to SMV
Part 2

Arie Gurfinkel (SEI/CMU)

based on material by Prof. Clarke and others
Brief Review
Symbolic Model Verifier (SMV)


Finite-state Systems described in a specialized language

Specifications given as CTL formulas

Internal representation using ROBDDs

Automatically verifies specification or produces a counterexample
A Sample SMV Program (short.smv)

MODULE main
VAR
    request: boolean;
    state: {ready, busy};
ASSIGN
    init(state) := ready;
    next(state) :=
        case
            state=ready & request: busy;
            TRUE : {ready, busy};
        esac;
SPEC AG(request -> AF (state = busy))
A Three-Bit Counter

MODULE main
VAR
  bit0 : counter_cell(TRUE);
  bit1 : counter_cell(bit0.carry_out);
  bit2 : counter_cell(bit1.carry_out);

SPEC AG AF bit2.carry_out

MODULE counter_cell(carry_in)
VAR
  value : boolean;
ASSIGN
  init(value) := FALSE;
  next(value) := value xor carry_in;
DEFINE
  carry_out := value & carry_in;

value + carry_in mod 2
module declaration

module instantiations

bit0

bit1

bit2
Inverter Ring

MODULE main
VAR
  gate1 : process inverter(gate3.output);
  gate2 : process inverter(gate1.output);
  gate3 : process inverter(gate2.output);

SPEC (AG AF gate1.output) & (AG AF !gate1.output)

MODULE inverter(input)
VAR
  output : boolean;
ASSIGN
  init(output) := FALSE;
  next(output) := !input;

FAIRNESS
  running
Fairness

- Assumed to be true infinitely often
- Model checker only explores paths satisfying fairness constraint
- Each fairness constraint must be true infinitely often

If there are no fair paths
- All existential formulas are false
- All universal formulas are true
Can A TRUE Result of Model Checker be Trusted

Antecedent Failure [Beatty & Bryant 1994]

- A temporal formula $AG (p \Rightarrow q)$ suffers an antecedent failure in model $M$ iff $M \not\models AG (p \Rightarrow q)$ AND $M \not\models AG (\neg p)$

Vacuity [Beer et al. 1997]

- A temporal formula $\phi$ is satisfied vacuously by $M$ iff there exists a sub-formula $p$ of $\phi$ such that $M \models \phi[p\leftarrow q]$ for every other formula $q$
- e.g., $M \models AG (r \Rightarrow AF a)$ and $M \models AG (r \Rightarrow AF \neg a)$ and $AG (r \Rightarrow AF \neg r)$ and $AG (r \Rightarrow AF FALSE)$, …
Vacuity Detection: Single Occurrence

\( \varphi \) is vacuous in \( M \) iff there exists an occurrence of a subformula \( p \) such that

- \( M \models \varphi[p \leftarrow \text{TRUE}] \) and \( M \not\models \varphi[p \leftarrow \text{FALSE}] \)

\[
\begin{align*}
M \not\models \text{AG (req } \Rightarrow \text{ AF TRUE) } & \quad M \not\models \text{AG (req } \Rightarrow \text{ AF FALSE) } \\
M \models \text{AG TRUE} & \quad M \not\models \text{AG } \neg\text{req }
\end{align*}
\]

\[
\begin{align*}
M \not\models \text{AG (TRUE } \Rightarrow \text{ AF ack) } & \quad M \not\models \text{AG (FALSE } \Rightarrow \text{ AF ack) } \\
M \models \text{AG AF ack} & \quad M \models \text{AG TRUE}
\end{align*}
\]
Detecting Vacuity in Multiple Occurrences: ACTL

An *ACTL* \( \varphi \) is vacuous in \( M \) iff there exists an a subformula \( p \) such that

- \( M \Vdash \varphi[p \leftarrow x] \), where \( x \) is a non-deterministic variable

Is \( AG (req \Rightarrow AF req) \) vacuous? Should it be?

\[
\begin{align*}
M \Vdash AG (x \Rightarrow AF x) & \quad \text{Always vacuous!!!} \\
\hline
M \Vdash AG TRUE & \\
\end{align*}
\]

Is \( AG (req \Rightarrow AX req) \) vacuous? Should it be?

\[
\begin{align*}
M \Vdash AG (x \Rightarrow AX x) & \quad \text{Can be vacuous!!!} \\
\text{can’t reduce} & \\
\end{align*}
\]
Run NuSMV

NuSMV [options] inputfile

- `-int` interactive mode
- `-lp` list all properties
- `-n X` check property number X
- `-ctt` check totality of transition relation
- `-old` compatibility mode
- `-ofm` file output flattened model
Using NuSMV in Interactive Mode

Basic Usage

• **go**
  – prepare model for verification

• **check_ctlspec**
  – verify properties

Simulation

• **pick_state [-i] [-r]**
  – pick initial state for simulation [interactively] or [randomly]

• **simulate [-i] [r] s**
  – simulate the model for ‘s’ steps [interactively] or [randomly]

• **show_traces**
  – show active traces
Useful Links

NuSMV home page
  • http://nusmv.fbk.eu/

NuSMV tutorial
  • http://nusmv.fbk.eu/NuSMV/tutorial/v25/tutorial.pdf

NuSMV user manual
  • http://nusmv.fbk.eu/NuSMV/userman/v25/nusmv.pdf

NuSMV FAQ
  • http://nusmv.fbk.eu/faq.html

NuSMV on Andrew
  • /afs/andrew.cmu.edu/usr6/soonhok/public/NuSMV-zchaff-2.5.3-x86_64-redhat-linux-gnu/

NuSMV examples
  • <NuSMV>/share/nusmv/examples

Today: 3 Examples

Mutual Exclusion

Bus Protocol

Traffic Light Controller
Example 1: Mutual Exclusion

Two processes want access to a shared resource
- they go through idle, trying, critical states

Safety (Mutual Exclusion)
- Only one process can be in the critical section at any given time
  - \(\mathcal{AG} \ (\neg(p_0 = \text{critical} \land p_1 = \text{critical})\)\)

Liveness (No Starvation)
- If a process is trying to enter critical section, it eventually enters it
  - \(\mathcal{AG} \ (p_0 = \text{trying} \rightarrow \mathcal{AF} \ p_0 = \text{critical})\)
SMV Example: Bus Protocol

Ed Clarke
Daniel Kroening
Carnegie Mellon University
Overview

Preliminaries:
- Single, shared bus
- Every node can broadcast on this bus

Design goals:
- Collision free operation
- Priorities for the nodes

Similar busses are used in the automotive industry
- CAN
- Byteflight
Basic Idea

Operation Principle
- Round based algorithm
- First person to start sending gets the bus

Achieve design goals by:
- Assign unique time to each node
- Guarantees Collision free operation
- The node with the lower time gets priority
Example

Node 1

Node 2

Node 3

Node 4

Bus

time
Example

Hm, I won’t send

Bus
Example

Node 1

Node 2

Node 3

Node 4

I will send!

Bus

time
Example

Node 1

Node 2

Node 3

Node 4

Bus

time
Example

Node 1  Node 2  Node 3  Node 4

Bus

time
Example

Node 1 → Node 2 → Node 3 → Node 4

Bus

time
Hm, I won’t send
SMV Model

Design:
- A state machine controls each node
- Counter keeps track of clock

Counter:
- Reset if someone sends
- Increment otherwise

```plaintext
MODULE node(bus_active)
VAR counter: 0 .. 99;

ASSIGN
next(counter):=
  case
  bus_active : 0;
  counter < 99: counter + 1;
  TRUE: 99;
  esac;
```
SMV Model

Design:

- A state machine controls each node
- Counter keeps track of the clock
SMV Model

**MODULE** node(priority, bus_active)

**VAR**
- counter: 0..99;
- state: { busy, skip, waiting, sending };

**ASSIGN**
- init(state):=busy;

next(state):= case
  - state=busy & beginning : { skip, waiting };
  - state=busy : busy;
  - state=skip & bus_active : busy;
  - state=skip : skip;
  - state=waiting & bus_active : waiting;
  - state=waiting & counter=priority: sending;
  - state=waiting: waiting;
  - state=sending: { busy, sending };

esac;
SMV Model

MODULE main
VAR
  node1: node(1, bus_active);
  node2: node(2, bus_active);
  node3: node(3, bus_active);
  node4: node(4, bus_active);
DEFINE
  bus_active:=node1.is_sending | node2.is_sending | node3.is_sending | node4.is_sending;
Properties

Desired Properties

- Safety: Only one node uses the bus at a given time

SPEC AG (node1.is_sending -> (!node2.is_sending & !node3.is_sending & !node4.is_sending))
SPEC AG (node2.is_sending -> (!node1.is_sending & !node3.is_sending & !node4.is_sending))
SPEC AG (node3.is_sending -> (!node1.is_sending & !node2.is_sending & !node4.is_sending))
SPEC AG (node4.is_sending -> (!node1.is_sending & !node2.is_sending & !node3.is_sending))
Properties

Desired Properties

- **Liveness**: a node that is waiting for the bus will eventually get it, given that the nodes with higher priority are fair

FAIRNESS node1.is_skipping
FAIRNESS node1.is_skipping & node2.is_skipping
FAIRNESS node1.is_skipping & node2.is_skipping & node3.is_skipping

SPEC AG AF bus_active
SPEC AG(node1.is_waiting -> AF node1.is_sending)
SPEC AG(node2.is_waiting -> AF node2.is_sending)
SPEC AG(node3.is_waiting -> AF node3.is_sending)
SPEC AG(node4.is_waiting -> AF node4.is_sending)
Traffic Light Controller

based on slides by Himanshu Jain
Outline

- Modeling Traffic Light Controller in SMV
- Properties to Check
- Four different SMV models for traffic light controller
Scenario
Binary traffic lights
Safety Property

This should not happen
Safety Property

This should not happen
Liveness Property

When will the stupid light become green again?
Liveness Property

Thank God!

Traffic in each direction must be served
Let’s Model all of this in NuSMV
SMV variables

Three Boolean variables track the status of the lights

North.Go=F
South.Go=F
West.Go=T
SMV variables

Three Boolean variables sense the traffic in each direction:

North.Sense = T
South.Sense = T
West.Sense = F
North.Sense = T
Properties

- Mutual exclusion
  - AG !(West.Go & (North.Go | South.Go))

- Liveness (e.g., in North direction)
  - AG (North.Sense & !North.Go -> AF North.Go)

- Similarly for South and West directions
Properties

- No strict sequencing
  - We don’t want the traffic lights to give turns to each other (if there is no need for it)
  - For example, if there is no traffic on west lane, we do not want West.Go becoming TRUE periodically

- We can specify such properties partially
  - See code for other such properties
  - We want these properties to FAIL
SMV modules

North module will control

South module will control

West module will control

Main module
- Initialize variables
- Start s all modules

N

S

W
What if north light is always green and there is always traffic in north direction???
Fairness Constraints

- What if a light is always green and there is always traffic in its direction

- We will avoid such scenarios by means of fairness constraints
  - FAIRNESS !(Sense & Go)
  - FAIRNESS running

- In any infinite execution, there are infinite number of states where either the light is red or there is no traffic in its direction
Implementations...
Some more variables

- To ensure mutual exclusion
  - We will have two Boolean variables
  - NS_lock: denotes locking of north/south lane
  - EW_lock: denotes locking of west lane

- To remember that there is traffic on a lane
  - Boolean variable: North.Req
    - If North.Sense becomes TRUE, then North.Req is set to TRUE
  - Similarly, for South.Req and West.Req
MODULE main

VAR
   -- lock for North-South direction
   NS_lock : boolean;
   -- lock for East-West direction
   EW_lock : boolean;

   North : process North (NS_lock, EW_lock, South.Go);
   South : process South (NS_lock, EW_lock, North.Go);
   West : process West (EW_lock, NS_lock);

ASSIGN
   init (NS_lock) := FALSE;
   init (EW_lock) := FALSE;
   ...

MODULE North (NS_lock, EW_lock, FriendGo)

VAR
Go : boolean;
Sense : boolean;
Req : boolean;
State : {idle, entering, critical, exiting};

ASSIGN
  init (State) := idle;
  next (State) :=
    case
      State = idle & Req : entering;
      State = entering & !EW_lock : critical;
      State = critical & !Sense : exiting;
      State = exiting : idle;
      TRUE : State;
    esac;
next (NS_lock) :=
  case
    State = entering & !EW_lock : TRUE;
    State = exiting & !FriendGo : FALSE;
    TRUE : NS_lock;
  esac;
init (Req) := FALSE;
next (Req) :=
  case
    State = exiting : FALSE;
    Sense: TRUE;
    TRUE : Req;
  esac;
init (Go) := FALSE;
next (Go) :=
  case
    State = critical : TRUE;
    State = exiting : FALSE;
    TRUE : Go;
  esac;
South is symmetric

West is a bit simpler (no East)

Let’s run NuSMV!!!
Mutual Exclusion CEX

1. All variables FALSE
2. North.Sense = T
   (North Run)
4. North.State = entering
5. **NS_lock=T**, North.Sense=T, North.State=critical
   (South Run)
6. South.Sense=T
7. South.Sense=F, South.Req=T
8. South.State = entering
9. South.State = critical
10. South.Go = T, South.State = exiting
11. West.Sense=T
   (West Run)
12. West.Sense=F, West.Req=T
13. West.State=entering
    (South Run)
15. **EW_lock=T**, West.State=critical, West.Sense=T
    (West Run)
    (North Run)
17. **West.Go=T**, West.Sense=F
    (West Run)
MODULE North (NS_lock, EW_lock, FriendGo)
VAR
Go : boolean;
Sense : boolean;
Req : boolean;
State : {idle, entering, critical, exiting};

ASSIGN
init (State) := idle;
next (State) :=
  case
    State = idle & Req : entering;
    State = entering & !EW_lock : critical;
    State = critical & !Sense : exiting;
    State = exiting : idle;
    TRUE : State;
  esac;
next (Go) :=
  case
    State = critical : TRUE;
    State = exiting : FALSE;
    TRUE : Go;
  esac;
init (Go) := FALSE;
traffic2.smv: fix

DEFINE
  EnterCritical := \text{State} = \text{entering} \& \neg \text{EW\_lock};

ASSIGN
  init (\text{State}) := \text{idle};
  next (\text{State}) :=
   \begin{cases} 
   \text{State} = \text{idle} \& \text{Req} : \text{entering}; \\
   \text{EnterCritical} : \text{critical}; \\
   \text{State} = \text{critical} \& \neg \text{Sense} : \text{exiting}; \\
   \text{State} = \text{exiting} : \text{idle}; \\
   \text{TRUE} : \text{State}; \\
   \end{cases}
  init (\text{Go}) := \text{FALSE};
  next (\text{Go}) :=
   \begin{cases} 
   \text{EnterCritical} : \text{TRUE}; \\
   \text{State} = \text{exiting} : \text{FALSE}; \\
   \text{TRUE} : \text{Go}; \\
   \end{cases}
  esac;
Model checking traffic2.smv

- Mutual exclusion property is satisfied

- Liveness property for North direction fails
  - $\text{AG ( (Sense \& \sim\text{Go}) \rightarrow AF \text{Go}) \text{ IN North}}$
CEX for Liveness is a Fair Cycle

1.6
North.State = entering
North.EnterCritical = T
all others are idle

1.10
South is given a turn, but does nothing

1.15
West.State = critical

1.16
North is given a turn, but can’t get a lock

1.19
West.State = idle
Add ‘Turn’ to Ensure Liveness

- This is in traffic3.smv

- Use Peterson’s mutual exclusion algorithm

- Introduce a variable Turn
  - Turn : \{nst, wt\}

  - If I have just exited the critical section, offer Turn to others
**traffic3.smv: Adding Turn**

**DEFINE**

EnterCritical :=
  State = entering & !EW_lock & (Turn = nst | !OtherReq);

next (Turn) :=
  case
  State = exiting & Turn = nst & !FriendReq : wt;
  TRUE : Turn;
  esac;
Model check again…

- Mutual still exclusion holds!

- What about liveness properties
  - In north direction? HOLDS
  - In south direction? HOLDS
  - In west direction? FAILS 😞
traffic4.smv

- Two extra variables to distinguish between North and South completion
  - North.Done, South.Done

- When North exits critical section
  - North.Done is set to TRUE
  - Similarly for South.Done

- When West exits
  - both South.Done and North.Done are set to FALSE
init (Done) := FALSE;
next (Done) :=
  case
    State = exiting : TRUE;
    TRUE : Done;
  esac;
next (Turn) :=
  case
    State = exiting & Turn = nst & !FriendReq : wt;
    State = exiting & Turn = nst &
      FriendDone & OtherReq : wt;
    TRUE : Turn;
  esac;
Hurray!

- Mutual exclusion holds
- Liveness for all three directions holds
- No Strict sequencing
Possible extensions

- Allow for north, south, east, and west traffic
- Allow for cars to turns
- Replace specific modules by a single generic one
  - Instantiate it four times
  - Once for each direction
- Ensure properties without using fairness constraints