Lecture 9: SMV: An Introduction

Pankaj Chauhan*

*Based on lectures by Ed Clarke, Dong Wang and Marius Minea.

Useful Links

CMU Model checking homepage
www.cs.cmu.edu/~modelcheck/

SMV versions for Unix/WinNT
www.cs.cmu.edu/~modelcheck/smv.html

SMV man page (must read)
www.cs.cmu.edu/~dongw/smv.txt

SMV manual
www.cs.cmu.edu/~modelcheck/smv/smvmanual.ps

NuSMV
http://nusmv.irst.itc.it/

SMV: Symbolic Model Verifier

Ken McMillan, *Symbolic Model Checking: An Approach to State Explosion Problem*, 1993.

Finite-state Systems described in a specialized language

Specifications given as CTL formulas

Internal representation using BDDs

Automatically verifies specification or produce counterexamples

Language Characteristics

Allows description of synchronous and asynchronous systems

Modularized and hierarchical descriptions

Finite data types: boolean, enum, int etc

Nondeterminism

Variety of specifications: safety, liveness, deadlock freedom

A Sample SMV Program

```
MODULE main
VAR
    request: boolean;
    state: {ready, busy};
ASSIGN
    init(state) := ready;
    next(state) :=
                case
                        state=ready & request: busy;
                        1: {ready, busy};
                esac;
-- Comments start with two -- (dashes)
SPEC AG(request –> AF (state = busy))
```

Variable Assignments

Assignment to initial state: init(value) := 0;

Assignment to next state: transition relation

next(value) := value + carry_in mod 2;

Assignment to current state: invariant relation

carry_out := value & carry_in;

SMV is a parallel assignment language

Specifications

EF p : from all initial states, a state where p holds is reachable.

A[p U q] : p remains true until q is true.

AG AF p : p is true infinitely often on every computation path.

AG (req –> AF ack) : any request will be eventually acknowledged.

ASSIGN and DEFINE

VAR a: boolean;
ASSIGN a := b | c;

- **declares a new state variable a**
- **becomes part of invariant relation**

DEFINE d:= b | c;

- **is effectively a macro definition, each occurrence of d is replaced by b | c**
- **no extra BDD variable is generated for d**
- **the BDD for b | c becomes part of each expression using d**

More on Case Statement

Case statement is converted to if-then-else internally, so all the guards are evaluated sequentially.

If none of the guards are true, an arbitrary valid value is returned.

Nondeterminism

Completely unassigned variable can model unconstrained input.

{val_1, …, val_n} is an expression taking on any of the given values nondeterministically.

Nondeterministic choice

- **to model an implementation that has not been refined yet**
- **In abstract models where a value of some state variable cannot be completely determined**

Modules and Hierarchy

Modules can be instantiated.

Each program has a module main

Scoping

- Variables declared outside a module can be passed as parameters.
- Internal variables of a module can be used in enclosing modules (submodel.varname).

Parameters are passed by reference.

Modules and Hierarchy - Example

```
MODULE main
VAR   bit0 : counter_cell(1);
      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) := 0;
  next(value) := value + carry_in mod 2;
DEFINE  carry_out := value & carry_in;
```

Module Composition

Synchronous composition

- All assignments are executed in parallel and synchronously.
- A single step of the resulting model corresponds to a step in each of the components.

Asynchronous composition

- A step of the composition is a step by exactly one process.
- Variables, not assigned in that process, are left unchanged.

Asynchronous Composition

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

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

MODULE inverter(input)
VAR   output: boolean;
ASSIGN
    init(output) := 0;
    next(output) := !input;
```

Counterexamples

-- specification AG AF (!gate1.output) is false
-- as demonstrated by the following execution
state 2.1:
gate1.output = 0
gate2.output = 0
gate3.output = 0

state 2.2:
[executing process gate1]

-- loop starts here --
state 2.3:
gate1.output = 1
[stuttering]

state 2.4:
[stuttering]

Fairness

Fairness constraint

- FAIRNESS ctl_formulae
- Assumed to be true infinitely often
- Model checker only explores paths satisfying fairness constraint

Special fairness constraint:

FAIRNESS running

Counter Revisited

```
MODULE main
VAR
        count_enable: boolean;
        bit0 : counter_cell(count_enable);
        bit1 : counter_cell(bit0.carry_out);
        bit2 : counter_cell(bit1.carry_out);
SPEC      AG AF bit2.carry_out
FAIRNESS count_enable
```

Modeling Shared Variables

```
MODULE main
VAR   x: boolean;
      z: process zero(x);
      o: process one(x);
SPEC   AG AF (x = 0)

MODULE zero(a)
ASSIGN  next(a) := 0;
FAIRNESS running

MODULE one(b)
ASSIGN  next(b) := 1;
FAIRNESS running
```

Implicit Modeling

INIT boolean_expr
- Initial states will be those satisfy boolean_expr.
- There is no next operator in boolean_expr.

INVAR boolean_expr
- The set of states is restricted to those satisfy boolean_expr
- There is no next operator in boolean_expr.

TRANS boolean_expr
- Restrict the transition relation.

Implicit Modeling Example

INVAR
(!enable -> stutter)

TRANS
((state = idle & next(state) = request) |
(state = request & sema & turn = id &
(next(state) = critical & next(sema) = 0) |
(state = critical & next(state) = release))

TRANS

Advantages

- Group assignments to different variables
- Good for modeling guarded commands

Disadvantages - easy to make mistakes

- Contradictory constraints
  - Transition relation is empty, reachable states is 0.
  - Transition relation is not total.
- Missing cases

Shared Data Example - main module

Two users assign pid to shared data in turn

```
MODULE main
VAR
  data: boolean;
  turn: boolean;
  user0: user(0, data, turn);
  user1: user(1, data, !turn);
ASSIGN
       next(turn) := !turn;
SPEC
  AG (AF data & AF (!data))
```

Shared Data Example - user module 1

Using ASSIGN and case statement won't work (constraining sema all the time).

```
MODULE user(pid, data, turn)
ASSIGN
 next(data) := case
    turn: pid;
    1: data;
  esac;
```

line 3: multiple assignment: next(data)

Shared Data Example - user module 2

TRANS is useful for changing shared data in a synchronous system between modules.

```
MODULE user(pid, data, turn)
TRANS
  turn –> next(data) = pid
```

Guarded Commands

guard1 : action1
guard2 : action2
…
otherwise: nop

TRANS
(guard1 & action1) |
(guard2 & action2) |
…
(!guard1 & !gard2 & … & “nop”)

Guarded Commands Pitfall

TRANS
guard1 –> action1 &
guard2 –> action2 &
…
(!guard1 & !guard2 & ... –> “nop”)

For example
true -> next(b) = 0 &
true -> next(b) = 1 & ...
This results in an empty transition relation

TRANS Guidelines

Try to use ASSIGN instead

Write in a disjunction of conjunctions format

Do not constrain current state variable, use INVAR for that

Try to cover all cases

Try to make guards disjoint

SMV Steps

read_model: read the input smv file

flatten_hierarchy: instantiate modules and processes

build_model: compile the model into BDDs (initial state, invar, transition relation)

check_spec: checking specification bottom-up

Synchronous vs. Asynchronous

Synchronous

- **Conjunct transition relation from each module**

Asynchronous

- ***N (V, V') = N_0(V, V') | .. | N_{n-1}(V,V')*** **where**
  N_i (V,V') = (v_i' = F_i(V)) & L_{j^1i} (v_j'= v_j)
- **Figure out the variables each process modifies**

Fact - SMV does not support modules with TRANS to be a process, why?

Run SMV

smv [options] <inputfile>

- -c cache-size
- -k key-table-size
- -m mini-cache-size
- -v verbose
- -r
  - prints out statistics about reachable state space
- -checktrans
  - checks whether the transition relation is total

SMV Options

–f

- computes set of reachable staes first
- Model checking algorithms traverse only set of reachable states instead of complete state space.
- useful if reachable state space is a small fraction of total state space

SMV Options: Variable ordering

Variable ordering is crucial for small BDD sizes and speed.

Generally, variables which are related need to be close in the ordering.

–i <input_order> –o <output_order>

- Input, output BDD variable ordering to given file.

-reorder

- Invokes automatic variable reordering

SMV Options: Transition relation

smv –cp part_limit

- **Conjunctive partitioning. Transition relation not evaluated as a whole, instead individual next() assignments are grouped into partitions that do not exceed part_limit.**
- **This method generally uses less memory and can benefit from early quantification.**
- **Only for synchronous models**

Mutual Exclusion -1

```
MODULE user(turn, id, other)
VAR         state: {n, t, c};
ASSIGN      init(state) := n;
  next(state) :=
    case
      state = n : {n, t};
      state = t & other = n: c;
      state = t & other = t & turn = id: c;
      state = c: n;
      1: state;
    esac;
SPEC AG(state = t –> AF (state = c))
```

Mutual Exclusion -2

```
MODULE main
VAR       turn: {1, 2};
      user1: user(turn, 1, user2.state);
      user2: user(turn, 2, user1.state);
ASSIGN   init(turn) := 1;
 next(turn) :=
   case
     user1.state = n & user2.state = t: 2;
     user2.state = n & user1.state = t: 1;
     1: turn;
   esac;
SPEC AG !(user1.state = c & user2.state = c)
```

Mutual Exclusion: Counterexample

Specification AG (user1.state != c) is false

state 1.1
- **turn = 1**
- **user1.state = n**
- **user2.state = n**

state 1.2:
- **user1.state = t**

state 1.3
- **user1.state = c**