Introduction to SMV

Arie Gurfinkel (SEI/CMU)

based on material by Prof. Clarke and others
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
Overview of SMV

SMV Input Language

- Finite State Kripke Structure
- Specification – CTL Formula

Backend

- OBDD based Symbolic Model Checking
- CounterExample

Yes

No
SMV Variants

- CMU SMV
  - Oldest Version
  - No GUI

- NuSMV
  - Strong abstraction functions
  - GUI
  - New language

- Cadence SMV

Two versions
- 2.x: Open Source, many new features, BDD and SAT based backends
- 1.x: Original version, had a GUI
NuSMV2 Architecture

- Flattening
- Simulation
- Trace Manipulation

- Boolean Encoding
  - encode scalar vars
  - build boolean functions for scalar propositions

- BDD-Based Verification
  - reachability
  - fair CTL model checking
  - LTL model checking
  - quantitative analysis

- Confluence of Influence
- BDD-Based Model Construction

- BDD Package

- Trace Reconstruction

- Different SAT Solver Interfaces
  - SAT solver N
  - SAT solver 2

- DIMACS Printer

- Bounded Model Checking
  - CNF conversion

- RBC Engine
SMV Language

Allows description of completely synchronous to asynchronous systems, detailed to abstract systems

Modularized and hierarchical descriptions

Finite data types: Boolean and enumerated

Parallel-assignment syntax

Non-determinism
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))
```
Kripke structure

AG(request → AF (state = busy))

Computation tree

holds after one step

holds in the initial state
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
        esac
            : {ready, busy};;
SPEC AG(request -> AX (state = busy))
```

what if AF is changed to $AX$?
$AG(request \rightarrow AX \,(state = busy))$ is false
SMV Syntax: Expressions

Expr ::=
  atom -- symbolic constant
  | number -- numeric constant
  | id -- variable identifier
  | “!” Expr -- logical not
  | Expr & Expr -- logical and
  | Expr | Expr -- logical or
  | Expr -> Expr -- logical implication
  | Expr <-> Expr -- logical equivalence
  | “next” “(“ id “)” -- next value
  | Case_expr
  | Set_expr
The Case Expression

Case_expr :: “case”
  
  expr_a1 “:” expr_b2 “;”
  ...
  expr_an “:” expr_bn “;”
  “esac”

Guards are evaluated sequentially

The first one that is true determines the resulting value

Cases must be exhaustive

It is an error if all expressions on the left hand side evaluate to FALSE
Variables and Assignments

Decl :: “VAR”
  atom1 “:” type1 “;”
  atom2 “:” type2 “;”
  ...

Decl :: “ASSIGN”
  dest1 “:=“ Expr1 “;”
  dest2 “:=“ Expr2 “;”
  ...

Dest :: atom  -- current
  | “init” “(“ atom “)”  -- initial
  | “next” “(“ atom “)”  -- next-state
Variables and Assignments (cont’d)

State is an assignment of values to a set of state variables
Type of a variable – boolean, scalar, user defined module, or array.

Assignment to initial state:
• init(value) := FALSE;

Assignment to next state (transition relation)
• next(value) := value xor carry_in;

Assignment to current state (invariant)
• carry_out := value & carry_in;

Either init-next or invar should be used, but not both

SMV is a parallel assignment language
Circular Definitions

... are not allowed

```plaintext
init(a) := 0;
next(a) := !b;

init(b) := 1;
next(b) := !a;
```
Nondeterminism

Completely unassigned variable model unconstrained input

\{\text{val}_1, \ldots, \text{val}_n\} \text{ is an expression taking on any of the given values nondeterministically}

- \text{next}(b) := \{\text{TRUE, FALSE}\};

Nondeterministic choice can be used to:

- Model an environment that is outside of the control of the system
- Model an implementation that has not been refined yet
- Abstract behavior
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;

- 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
SPEC Declaration

Decl :: “SPEC” ctlform

Ctlform :: expr -- bool expression
| “!” ctlform
| Ctlform <op> Ctlform
| “E” Pathform
| “A” Pathform

Pathform :: “X” Ctlform
| “F” Ctlform
| “G” Ctlform
| Ctlform “U” Ctlform
Modules

Modules can be instantiated many times, each instantiation creates a copy of the local variables

Each program must have a module **main**

Scoping

- Variables declared outside a module can be passed as parameters

Parameters are passed by reference.
Pass by reference

DEFINE
  a := 0;

VAR
  b : bar(a);

...
MODULE bar(x)
DEFINE
  a := 1;
  y := x;

DEFINE
  a := 0;
  b.y := 0;
  b.a := 1;
Pass by reference

VAR
  a : boolean;
  b : foo(a);
...
MODULE foo(x)
VAR
  y : boolean;
ASSIGN
  x := TRUE;
  y := FALSE;

VAR
  a : boolean;
  b.y : boolean;
ASSIGN
  a := TRUE;
  b.y := FALSE;
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
AG AF bit2.carry_out is true

bit0
\{
\begin{align*}
\text{in} &= \begin{bmatrix} 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \end{bmatrix} \\
\text{val} &= \begin{bmatrix} 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \end{bmatrix} \\
\text{out} &= \begin{bmatrix} 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \end{bmatrix}
\end{align*}
\}

bit1
\{
\begin{align*}
\text{in} &= \begin{bmatrix} 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \end{bmatrix} \\
\text{val} &= \begin{bmatrix} 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \end{bmatrix} \\
\text{out} &= \begin{bmatrix} 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \end{bmatrix}
\end{align*}
\}

bit2
\{
\begin{align*}
\text{in} &= \begin{bmatrix} 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \end{bmatrix} \\
\text{val} &= \begin{bmatrix} 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \end{bmatrix} \\
\text{out} &= \begin{bmatrix} 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \end{bmatrix}
\end{align*}
\}

bit2.carry_out is true
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 (!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;
AG (!bit2.carry_out) is false

bit0
\[
\begin{array}{ccccccccccc}
\text{in} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\text{val} & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
\text{out} & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
\end{array}
\]

bit1
\[
\begin{array}{ccccccccccc}
\text{in} & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
\text{val} & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
\text{out} & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
\end{array}
\]

bit2
\[
\begin{array}{ccccccccccc}
\text{in} & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\
\text{val} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{out} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

bit2.carry_out is true
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.
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
In asynchronous composition, a step of the computation is a step by exactly one component. The process to execute is assumed to choose gate0, gate1, and gate2 repeatedly.

\[(\text{AG AF gate1.output}) \& (\text{AG AF !gate1.output}) \text{ is true}\]
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
Synchronous vs Asynchronous

In Asynchronous process, need not combine transition relation of each process

Complexity of representing set of states reachable in \( n \) steps higher in asynchronous processes occasionally due to higher number of interleaving

SMV models asynchronous composition by a synchronous one
Implicit Modeling

**INIT Expr**

Boolean valued expression giving initial states

**INVAR Expr**

Boolean valued expression restricting set of all states of model

**TRANS Expr**

Boolean valued expression restricting transition relation of system
### Implicit Modeling Example

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

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

MODULE inverter(input)
VAR
  output : boolean;
INIT
  output = FALSE;
TRANS
  next(output) = !input | next(output) = output
```
Advantages

- Group assignments to different variables
- Good for modeling guarded commands
  - IF guard THEN new state

Disadvantages

- Logical absurdities can lead to unimplementable descriptions
Shared Data Example

Two users assign PID to Data in turn

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

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

Error: multiple assignment: next(data)
Shared Data Example with TRANS

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

MODULE user(pid, data, turn)
TRANS
  turn=pid -> next(data) = pid;
TRANS Pitfalls

Inconsistencies in TRANS result in an empty transition relation

All universal properties are satisfied

All existential properties are refuted
**TRANS Guidelines**

Use ASSIGN if you can!

Validate your model with simulation and sanity checks

Check that transition relation is total (-ctt option)

Write in a disjunction of conjunction format

Cover all cases

Make guards disjoint
MODULE main

VAR
  send : {s0,s1,s2};
  recv : {r0,r1,r2};
  ack : boolean;
  req : boolean;

ASSIGN
  init(ack):=FALSE;
  init(req):=FALSE;
  init(send):= s0;
  init(recv):= r0;

next (send) :=
  case
    send=s0:{s0,s1};
    send=s1:s2;
    send=s2&ack:s0;
    TRUE:send;
  esac;

next (recv) :=
  case
    recv=r0&req:r1;
    recv=r1:r2;
    recv=r2:r0;
    TRUE: recv;
  esac;

next (ack) :=
  case
    recv=r2:TRUE;
    TRUE: ack;
  esac;

next (req) :=
  case
    send=s1:FALSE;
    TRUE: req;
  esac;

SPEC AG (req -> AF ack)
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 $M \vDash AG (p \Rightarrow q)$ AND $M \vDash AG (\neg p)$

Vacuity [Beer et al. 1997]

• A temporal formula $\varphi$ is satisfied vacuously by $M$ iff there exists a sub-formula $p$ of $\varphi$ such that $M \vDash \varphi[p \leftarrow q]$ for every other formula $q$
• e.g., $M \vDash AG (r \Rightarrow AF a)$ and $M \vDash AG (r \Rightarrow AF \neg a)$ 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

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

<table>
<thead>
<tr>
<th>Statement</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M \not\models \text{AG (req } \Rightarrow \text{AF TRUE)} )</td>
<td>( M \not\models \text{AG TRUE} )</td>
</tr>
<tr>
<td>( M \not\models \text{AG (req } \Rightarrow \text{AF FALSE)} )</td>
<td>( M \not\models \text{AG } \neg \text{req} )</td>
</tr>
<tr>
<td>( M \not\models \text{AG (TRUE } \Rightarrow \text{AF ack)} )</td>
<td>( M \not\models \text{AG AF ack} )</td>
</tr>
<tr>
<td>( M \not\models \text{AG (FALSE } \Rightarrow \text{AF ack)} )</td>
<td>( M \not\models \text{AG TRUE} )</td>
</tr>
</tbody>
</table>
Detecting Vacuity in Multiple Occurrences

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

\[
\begin{array}{ll}
M \not\models AG (TRUE \Rightarrow AF \, TRUE) & M \not\models AG (FALSE \Rightarrow AF \, FALSE) \\
M \not\models AG TRUE & M \not\models AG TRUE
\end{array}
\]

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

\[
\begin{array}{ll}
M \not\models AG (TRUE \Rightarrow AX \, TRUE) & M \not\models AG (FALSE \Rightarrow AX \, FALSE) \\
M \not\models AG TRUE & M \not\models AG TRUE
\end{array}
\]
Detecting Vacuity in Multiple Occurrences: ACTL

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

- $M \not\models \varphi[p \leftarrow x]$, where $x$ is a non-deterministic variable

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

\[
\begin{align*}
M \not\models AG \ (x \Rightarrow AF \ x) & \quad \text{Always vacuous!!!} \\
M \not\models AG \ TRUE
\end{align*}
\]

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

\[
\begin{align*}
M \not\models AG \ (x \Rightarrow AX \ x) & \quad \text{Can be vacuous!!!} \\
can't \ reduce
\end{align*}
\]
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
Ken McMillan, Symbolic Model Checking: An Approach to the State Explosion Problem, 1993