An Introduction to SMV

Sérgio Campos, Edmund Clarke

Symbolic Model Verifier

- Ken McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem*, Kluwer Academic Publishers, 1993.
- finite-state systems described in a specialized language
- specifications given as CTL formulas
- internal representation using binary decision diagrams
- automatically verifies specification or produces counterexamples

SMV Language Characteristics

- allows description of synchronous and asynchronous systems
- modularized and hierarchical descriptions
- finite data types: booleans, enumerations, integers
- nondeterminism and partial implementations
- variety of specifications: safety, liveness, deadlock freedom

An Example 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;
SPEC
  AG (request -> AF (state = busy))
```

Variable Assigments

- 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
- all assignments executed in parallel and simultaneously
- SMV checks for circularities and duplicate assignments

ASSIGN and DEFINE

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

- declares a new state variable `a` (and a new BDD variable)
- assignment becomes part of *invariant relation*
- generally suitable for more complex assignments

```
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`
- generally suitable for simple expressions

Transition X Invariant Relation

1. Long expression goes into transition relation:
   ```
   ASSIGN
     NEXT(a) := case
                  long_complex_expression;
                esac;
   ```
2. Long expression goes into invariant relation:
   ```
   ASSIGN
     anext := long_complex_expression;

     NEXT(a) := anext;
   ```

$$EX(f) := \underbrace{Invar}_{1} \wedge (\exists \bar{v}' \left[f(\bar{v}') \wedge \underbrace{N(\bar{v}, \bar{v}')}_{2} \right])$$

- Moving expressions between *Invar* and *N* can affect performance significantly!

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 between several values can be used
  - 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

```
MODULE main
VAR
  bit0: counter_cell(1);
  bit1: counter_cell(bit0.carry_out);
  bit2: counter_cell(bit1.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;
```


Modules and hierarchy

- modules are instantiated in variable declarations
- each program is required to have module `main`
- `module_name.var_name` used to reference variable in a module
- scoping:
  - values, expressions, or variables declared outside a module can be passed as parameters
  - internal variables of a module can be used in enclosing modules
- parameters passed by reference
- modules may contain fairness constraints and specs

Module Composition

- Modules represent individual finite-state machines. These can be composed either synchronously or asynchronously.
- In *synchronous composition* (default), 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* is specified by prefixing a module with the keyword `process`.
  - In asynchronous composition, a step of the composition is a step by *exactly one* process.
  - The process is chosen nondeterministically.
  - 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.out) & (AG AF !gate1.out)


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

Specifications

Have to hold in each initial state. Typical CTL formulas:

- **EF** p
  from all initial states, a state where p holds is reachable
- **AG EF** p
  from every state, it is possible to get to a state where p holds
- **AF** p
  p is eventually reached on every path from an initial state
- **AG AF** p
  p is true infinitely often on every computation path
- **AG** (req -> **AF** ack)
  any request will be eventually acknowledged

Asynchronous Composition

```
AG (!bit1.carry_out) is false


.bit0.value = 0
.bit1.value = 0
.bit2.value = 0


next state
.bit0.value = 1


next state
.bit0.value = 0
.bit1.value = 1


next state
.bit0.value = 1
```


Fairness

A *fairness constraint* is a condition assumed to be true infinitely often.

Model checker only explores paths satisfying fairness constraint.

Example: modified 3-bit counter:

```
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
FAIR  count_enable;
```


Modeling shared variables

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

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

MODULE main
VAR x: boolean;
    z: process zero(x);
    o: process one(x);

SPEC  AG AF (x = 0)
```

SMV Options

smv -f

- computes set of reachable states *before* verifying properties
- 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
- tradeoff: extra time needed to compute reachable states

smv -r
prints out statistics about reachable state space

smv -v 1 or **smv -v 2**
verbose mode: useful to follow progress of verification

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.

smv -o `filename`
Outputs BDD variable ordering to given file. Can be used to do manual variable reordering.

smv -i `filename`
Inputs BDD variable ordering from specified file

smv -reorder
Invokes automatic variable reordering as soon as BDD size exceeds a certain limit.

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 variable quantification.

smv -inc
Incremental evaluation of transition relation. At each step in forward search, the transition relation is restricted to the reached state set, which can reduce its size at the expense of time overhead.

Sample program: mutual exclusion

```
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 (cont'd)

```
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))
```


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
```