SMV

Symbolic model verifier - a program that uses symbolic model checking algorithm to evaluate formulas of CTL w.r.t. a finite state model - a finite Kripke structure. The language for describing the model is a simple parallel assignment.

Some points about the language:
- A program may consist of several modules
- Variables are boolean , enumeration type or integer subrange
- Should contain a module `main`, with no formal parameters
- Value of variables in each state is defined using `init` and `next`
- Transitions can be chosen non-deterministically

SMV Syntax - Variables

- Variables - any sequence of characters in the set {`A-Z,a-z,0-9,_,-`}, beginning with an alphabetic character.

`AB-3` - variable

`AB - 3` - expression

In general, spaces in expression are not necessary, except in case of `-`.
- Variables can be boolean, enumerated types, user defined modules ("objects"), or an array of any of these:

```
VAR
  state :  {on, off};
  state1 :  array 2..5 of {on, off};
  state2 :  computeState(1);
  state3 :  compute;
  state4 :  array 2..5 of state; - error
  state5 :  array on..off of boolean; - error
```


Some More SMV Syntax

- Comment - anything starting with -- and ending with a newline.
- Boolean values are 0 and 1.
- Subscripts of an array should evaluate to integer constants.
- There are no loops, so array expressions should be handled separately for each subscript.
- Variables are passed into modules by reference.

First SMV Example

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

- Why non-deterministic choices? To describe systems which are not fully implemented, or abstract models of complex systems.
- Note that `request` never receives an assignment - models input.

Another 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;
```

- *a.b* - component *b* of module *a*.
- DEFINE - same as ASSIGN but
  - don't require introducing a new variable in BDD representation
  - cannot be given values non-deterministically
  - are dynamically typed.

Models of Concurrency

Maximum parallelism - "simultaneous execution of atomic actions in all system modules capable of performing an operation"

Interleaving - "concurrent execution of modules is represented by the interleaving of their atomic actions".

Example:

Typechecking Issues

- Check for multiple assignments:
  `a := a + 1`
- Check for circular dependencies between variables and defined symbols, making sure that all legal SMV programs can be implemented
- Check for ambiguous identifiers:

```
VAR
  state :  {on, off}
  on :  boolean
```

- Check for correct types:

```
VAR
  on :  boolean
  on := 4;
```


Modeling Interleaving

`process` keyword. Allows interleaving execution. The program executes a step by non-deterministically choosing a process, then executing all of its assignment statements in parallel.

```
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) := 0;
  next(output) := !input;
```

Output of Running SMV

```
-- specification AG AF gate1.output &
   AG AF (!gate1.outpu... is false
-- as demonstrated by the following execution sequence
-- loop starts here --
state 1.1:
gate1.output = 0
gate2.output = 0
gate3.output = 0

state 1.2:

resources used:
user time: 0.07 s, system time: 0.13 s
BDD nodes allocated: 246
Bytes allocated: 917504
BDD nodes representing transition relation: 32 + 1
```

What went wrong? We never specified that each process has to execute infinitely often - a *fairness* constraint.

Notion of Fairness

Fairness has been an issue since the original '86 paper. Fairness cannot be expressed in CTL, so other means are necessary.

Three types of fairness properties:

- *Impartiality* - requires that every process is executed infinitely often.
- *Just* computation - every process is either infinitely often disabled or else it is infinitely often executed.
- *Fair* computation - if the process is infinitely often enabled, then it will be infinitely often executed.

Specifying Fairness

```
FAIRNESS
  running
```

`running` - special variable which is true if and only if that process is currently executing.

This specifies *impartiality*. '86 paper contained the following lemma: *Given any finite structure M=(S, R, P), collection F =* $\{G_1, ..., G_k\}$ *of subsets of S, and state* $s_0 \in S$*, the following conditions are equivalent:*
(1) There exists a fair path in M starting at s_0
(2) There exists a strongly connected component C of (the graph of) M such that
(a) there is a finite path from s_0 *to a state* $t \in C$*, and*
(b) for each G_i*, there is a state* $t_i \in C \cap G_i$

Implementing Fairness

Have system Pr of concurrent processes with interleaving model.
Model Pr by *(S, R, P)* and labeling function $L : R \rightarrow Pr$, where
S is the set of global states of the system
R is the single-step execution relation of the system
for each transition in R, L gives the process that caused the transition.
By duplicating each state in S at most card(Pr) times, can model the system by a structure (S', R', R', F), where
each state in s' is reached by the execution of at most one process
F is a partitioning of S' s.t. each element in F is the set of states reached by the execution of one process. So,
card(F) = card(Pr)

Example - Inverter Ring

```
MODULE main
VAR
  gate1 : inverter(gate3.output);
  gate2 : inverter(gate2.output);
  gate3 : inverter(gate3.output);
SPEC
  (AG AF gate1.output) & (AG AF !gate1.output)

MODULE inverter(input)
VAR
  output : boolean;
DEFINE
  temp := !input;
ASSIGN
  init(output) := 0;
  next(output) := {temp} union {output};

FAIRNESS
  output = temp
```


Inverters + Fairness + Output

```
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) := 0;
  next(output) := !input;
FAIRNESS
  running
```

Output

```
-- specification AG AF gate1.output &
AG AF (!gate1.outpu... is true

resources used:
user time: 0.06 s, system time: 0.12 s
BDD nodes allocated: 239
Bytes allocated: 917504
BDD nodes representing transition relation: 32 + 1
```


Inverter Ring - Cont'd

The number of possible transitions from a given state can be as high as s^n, where n is the number of gates. In this example, this is not the case.

```
-- specification AG AF gate1.output &
   AG AF (!gate1.outpu... is true

resources used:
user time: 0.02 s, system time: 0.16 s
BDD nodes allocated: 31
Bytes allocated: 917504
BDD nodes representing transition relation: 7 + 1
```


Advantages to Interleaving Model

- Allows a particularly efficient BDD representation of the transition relation

The set of states reachable by one step of the program is the union of the sets reachable by each individual process. So, do not need reachability graph.

- But, sometimes have increased complexity in representing the set of states reachable in n steps.

What if we want to model asynchronous circuit without using processes?
  - All gates would execute simultaneously
  - We would allow each gate the non-deterministic choice of evaluating its output, or keeping the same output value.

Mutual Exclusion Example

Each process has four states: `idle`, `entering`, `critical`, and `exiting`. `entering` - process wants to enter its critical region. If variable `semaphore` is zero, it goes to `critical` and sets `semaphore` to one. On exiting the critical region, it sets `semaphore` to zero again.

Mutual exclusion property:
"two processes cannot be in critical section at the same time"
AG !(proc1.state = critical & proc2.state = critical)

Mutual Exclusion (Cont'd)

```
MODULE main
VAR
  semaphore : boolean;
  proc1 : process user(semaphore);
  proc2 : process user(semaphore);
ASSIGN
  init(semaphore) := 0;
SPEC
  AG !(proc1.state = critical & proc2.state = critical)

MODULE user(semaphore)
VAR
  state : {idle, entering, critical, exiting};
ASSIGN
  init(state) := idle;
  next(state) := case
     state = idle : {idle, entering};
     state = entering & !semaphore : critical;
     state = critical : {critical, exiting};
     state = exiting : idle;
     1 : state;
  esac;
  next(semaphore) := case
     state = entering : 1;
     state = exiting : 0;
     1 : semaphore;
  esac;
FAIRNESS
  running
```


Is the Spec Correct?

```
-- specification AG (!(proc1.state = critical &
   proc2.sta... is true

resources used:
user time: 0.09 s, system time: 0.13 s
BDD nodes allocated: 560
Bytes allocated: 917504
BDD nodes representing transition relation: 69 + 1
```

Let's add a "lack of starvation" criteria:

```
AG (proc1.state = entering -> AF proc1.state = critical)
```

Output now:

```
-- specification AG (proc1.state = entering ->
   AF proc1.s... is false
-- as demonstrated by the following execution sequence
state 1.1:
semaphore = 0
proc1.state = idle
proc2.state = idle

state 1.2:
[executing process proc1]

-- loop starts here --
state 1.3:
proc1.state = entering
```


Output (Cont'd)

```
state 1.4:
[executing process proc2]

state 1.5:
[executing process proc2]
proc2.state = entering

state 1.6:
[executing process proc1]
semaphore = 1
proc2.state = critical

state 1.7:
[executing process proc2]

state 1.8:
[executing process proc2]
proc2.state = exiting

state 1.9:
semaphore = 0
proc2.state = idle

resources used:
user time: 0.11 s, system time: 0.11 s
BDD nodes allocated: 1421
Bytes allocated: 917504
BDD nodes representing transition relation: 69 + 1
```

Attempt to Fix the Spec

- Introduce a variable `turn`
- `turn` would ensure that processes take turns entering the critical section.

Spec:

```
MODULE main
VAR
  semaphore : boolean;
  turn : boolean;
  proc1 : process user(semaphore, 1, turn);
  proc2 : process user(semaphore, 0, turn);
ASSIGN
  init(semaphore) := 0;
  init(turn) := 1;
SPEC
  AG (proc1.state = entering -> AF proc1.state = critical)
SPEC
  AG !(proc1.state=critical & proc2.state=critical)

MODULE user(semaphore, number, turn)
VAR
  state : {idle,entering,critical,exiting};
ASSIGN
  init(state) := idle;
  next(state) :=
```


Fixed Spec (Cont'd)

```
    case
      state = idle : {idle,entering};
      state = entering & !semaphore&(turn=number):critical;
      state = critical : {critical,exiting};
      state = exiting : idle;
      1 : state;
    esac;
  next(semaphore) :=
    case
      state = entering : 1;
      state = exiting : 0;
      1 : semaphore;
    esac;
  next(turn) :=
    case
      state = exiting : !turn;
      1 : turn;
    esac;
FAIRNESS
  running
FAIRNESS
  turn = number
```


Output

```
-- specification AG (proc1.state = entering
   -> AF proc1.s... is true
-- specification AG (!(proc1.state = critical &
   proc2.sta... is true

resources used:
user time: 0.14 s, system time: 0.13 s
BDD nodes allocated: 2530
Bytes allocated: 917504
BDD nodes representing transition relation: 104 + 1
```

I've tried it about 10 times before getting it quite right!

TRANS relation

Parallel assignment should be suitable to most purposes. Still, it is possible to specify the transition relation directly as a propositional formula in terms of the current and next values of state variables.

- Any current/next state pair is in the transition relation iff the value of the formula is one.
- It is possible to give the set of possible initial states as a formula in terms of only the current state variables.
- It is done using TRANS and INIT.

Inverter using INIT and TRANS

```
MODULE main
VAR
  gate1 : inverter(gate3.output);
  gate2 : inverter(gate1.output);
  gate3 : inverter(gate2.output);
SPEC
  (AG AF gate1.output) & (AG AF !gate1.output)

MODULE inverter(input)
VAR
  output : boolean;
INIT
  output = 0
TRANS
  next(output) = !input |
  next(output) = output
```

Actually, this spec is wrong! Try to fix it!

Synchronous Arbiter

- A synchronous finite state machine.
- Composed of a "daisy chain" of arbiter cells.
- The arbiter grants the bus on each clock cycle to the requester with the highest priority.
- Each arbiter cell receives a "bus-grant" input from the next higher priority cell
- If this signal is true, and the cell's "request" input is true, then the cell activates its "acknowledge" output and negates "bus-grant" to the next lower priority sell.
- If the signal is false, the "bus grant" input is passed along to the next cell via the "bus-grant" output.
- When the bus approaches saturation, the arbiter reverts to a round-robin scheme.
- This is accomplished by a "token" which is passed in a cyclic manner from the first cell down to the last, and then back to the first.

Synchronous Arbiter (Cont'd)

- The "token" moves once each clock cycle.
- When the "token" passes a cell whose "request" is active, it sets a flag "waiting", which remains set as long as the request persists.
- When the token returns to that cell, if the "waiting" flag is still set, the cell receives immediate highest priority, accomplished by asserting an output called "override".
- This signal propagates to the highest priority cell and negates its "bus-grant" input.

Specification:
1. No two acknowledge outputs are asserted simultaneously
2. Every request is eventually acknowledged.
3. Acknowledge is not asserted without request.

Synchronous Arbiter Spec

```
MODULE arbiter-element(above,below,init-token)

VAR
  Persistent : boolean;
  Token : boolean;
  Request : boolean;

ASSIGN
  init(Token) := init-token;
  next(Token) := token-in;
  init(Persistent) := 0;
  next(Persistent) := Request & (Persistent | Token);

DEFINE
  above.token-in := Token;
  override-out := above.override-out | (Persistent & Token);
  grant-out := !Request & below.grant-out;
  ack-out := Request & (Persistent & Token | below.grant-out

SPEC
  AG ((ack-out -> Request) & AF (!Request | ack-out))

MODULE main

VAR
  e5 : arbiter-element(self,e4,0);
  e4 : arbiter-element(e5,e3,0);
  e3 : arbiter-element(e4,e2,0);
  e2 : arbiter-element(e3,e1,0);
  e1 : arbiter-element(e2,self,1);
```

Arbiter Spec (Cont'd)

```
DEFINE
  grant-in := 1;
  e1.token-in := token-in;
  override-out := 0;
  grant-out := grant-in & !e1.override-out;
SPEC
  AG (
      !(e1.ack-out & e2.ack-out)

      & !(e1.ack-out & e3.ack-out)
      & !(e2.ack-out & e3.ack-out)

      & !(e1.ack-out & e4.ack-out)
      & !(e2.ack-out & e4.ack-out)
      & !(e3.ack-out & e4.ack-out)

      & !(e1.ack-out & e5.ack-out)
      & !(e2.ack-out & e5.ack-out)
      & !(e3.ack-out & e5.ack-out)
      & !(e4.ack-out & e5.ack-out)
     )
```


Runtime Options

- `-f` - forward search of the state space of the model should be made before checking the specifications, i.e., first compute reachable states, then verify formulas.
- `-v` <`level`> - verbose level - prints all sorts of gibberish
- `-o` <`output-order`> - write variable ordering to a file after parsing. Do not do evaluation
- `-i` <`input-order`> - read variable ordering from a file. Usually, `-o`, reorder by hand, `-i`.
- other options, usually aimed at figuring out a good BDD ordering. See `man` pages for `smv`.

Running times

```
smv syncarb5.smv

resources used:
user time: 0.19 s, system time: 0.12 s
BDD nodes allocated: 4802
Bytes allocated: 917504
BDD nodes representing transition relation: 144+1

smv -f syncarb5.smv

resources used:
user time: 0.13 s, system time: 0.09 s
BDD nodes allocated: 1830
Bytes allocated: 917504
BDD nodes representing transition relation: 144+1
```


A Final Note

Current implementation of SMV does much more than what is specified in the manuals. For example, modules can be inherited, including multiple inheritance.
Example (`gigamax.smv`)

```
MODULE bus-device
VAR
  master  : boolean;
  waiting : boolean;
  reply-stall : boolean;
...

MODULE processor(CMD,REPLY-OWNED,REPLY-WAITING,REPLY-STALL)
ISA bus-device
ISA cache-device
...

MODULE cache-device
VAR
  state : {invalid,shared,owned};

DEFINE
  readable := ((state = shared) | (state = owned)) & !waitir
  writable := (state = owned) & !waiting;
...
```