CS 745 (Fall 2004): Computer Aided Verification (Introduction to Formal Methods)

Lecture 9: CTL, SMV

Nancy Day

DC 2335, nday@cs.uwaterloo.ca

Office Hours: Mon 11:30-12:30, Thurs 3-4pm

`http://www.student.cs.uwaterloo.ca/˜cs745`

uw.cs.cs745

Today's Agenda

- CTL
- SMV
- SMV modules
- Concurrency and fairness

Symbolic Model Verifier (SMV)

SMV is a program that checks $\mathcal{M} \models \phi$ for a ϕ that is a property in the temporal logics LTL and CTL. We will be using LTL.

- Developed to verify synchronous circuits
- Extended to verify asychronous circuits
- Successfully used to verify models of reactive systems

Next we are going to talk about modelling systems using SMV. The SMV symbolic model checking algorithm will be described next week.

Getting and installing SMV

Cadence SMV (extended SMV) can be found at:

`http://www-cad.eecs.berkeley.edu/˜kenmcmil/smv`

You have to fill in a registration form. You can get SMV for various platforms.

Run the executable `vw`. Like HOL, you edit your model description as a text file and load it into SMV.

Look in smv/doc/smv for a tutorial, and SMV examples.

Reactive Systems

- System interacts with its environment, monitoring and responding to environmental events
- Computation may not terminate
- System behaviour changes over time, in reaction to history of inputs
- Complexity is due to concurrency and interactions among components
- Examples: operating systems, embedded systems, process-control systems, financial trading systems, automated banking machines, etc.

Compared to Transformational Programs

- Program computes a function from inputs to outputs
- Complexity is in data transformations
- Examples: compilers, filters, payroll systems, scientific computations

SMV Modelling

- Goal is to describe control and interaction. Hence, no complex data structures, not much data manipulation.
- SMV Language: Communicating Finite State Machines (FSMs with variables and communication channels)
- System may consist of several modules
- Modules consist of several simple parallel assignments
- Model may also specify constraints on environment's behaviour

SMV Modelling

A system is described as a set of modules. Each module is a reactive system interacting with other modules and the system's environment.

Each modules has variables that it reacts to, and that is manipulates.

In each module, there are variable declarations, assignments to variables, and properties that we want to check.

The main module is like a main program. In the simplest SMV descriptions we use only the main module, and no sub-modules.

Modules can be parameterized and the main module can creating instances of modules to describe the system.

Example Specification (echo1.smv)

```
MODULE main ()
{
   signal : boolean;
   echo   : boolean;

   init(echo) := 0;

   next(echo) := signal;

   mypropname: assert G (signal <-> X (echo));

}
```

Example Specification (echo1.smv)

Example Traces:

signal:	01001000100001...
echo:	001001000100001...
signal:	11001100110011...
echo:	011001100110011...

At the prompt, type “vw echo1.smv &“

Example Specification (echo2.smv)

```
MODULE main ()
{
   signal : boolean;
   echo   : boolean;

   init(echo) := 0;

   next(echo) := signal & 1 ;

   mypropname: assert G (signal <-> X (echo));

}
```

See echo2.smv

Example Specification (echo3.smv)

```
MODULE main ()
{
   signal : boolean;
   echo   : boolean;

   init(echo) := 0;

   next(echo) := signal & 0 ;

   mypropname: assert G (signal <-> X (echo));

}
```

See echo3.smv

SMV Modelling

Recall that the SMV modelling notation is used to describe communicating finite state machines.

It consists of a set of modules, with one main module.

In each module there are:

- variables declarations,
- variable initialization,
- assignments, and
- properties that we want to check.

Variables

Variables can be boolean, enumerated types, integer subranges, user-defined modules, or an array of any of these:

```
var1 : {on, off};
var2 : array 2..5 of {on, off};
var3 : array 1..10 of array 2..5 of boolean;
var4 : Inverter(1);
var5 : array on..off of boolean;   -- error
var6 : {enabled, active, off};     -- error
var7 : 0..6;
```

Variables

- Enumerated values must be distinct, from each other and from integer values
- Boolean values are 0 and 1
- Boolean operations are: $\sim$ (not), & (and), | (or), ˆ (xor), $->$ (implies), $<->$ (iff)
- Array subscripts must evaluate to integers (preferably constants)
- Comments are delimited by $--$ and a newline

Execution Model

- System state $\mathcal{S}$ is defined by the variables' values
  $v_1{:}T_1;\ v_2{:}T_2;\ \cdots;\ v_n{:}T_n;$
  $\mathcal{S} \in T_1 x T_2 x \cdots x T_n$
- Each variable is either controlled by the system (i.e., the model explicitly assigns values to the variable) or is controlled by the environment (i.e., can be thought of as an input variable).

Execution Model

- Round-based execution: initialization round followed by a sequence of update rounds. In each round
  - the environment-controlled variables non-deterministically change value
  - the system executes its parallel assignment statements
  - the result is a new system state
- Variable assignments may either take effect in the next system state (sequential/unit delay)

  next(x) := y + 3;

  or be effective immediately (combinational/derived)

  x := y + 3;

Assignments

Simple Assignments

```
x := y + 3;          – derived
next(y) := z;        – unit delay
```

Conditional Assignments: if-then-else returns the value assigned. The condition or guard is a Boolean expression over variables.

Examples:
In a case statement the conditions are evaluated in order.

```
next(y) := (a | b) ? x : x+1;
next(k) := case {
               c1 : x1;
               c2 : x2;
               ...
               default : z;
           };
```

Example: Thermostat (See therm1.smv)

```
MODULE main()
{
   temp : {hot, cold, justright}; -- room temperature
   active : boolean;              -- is thermostat on?
   heat : boolean;                -- is furnace on?
   aircond : boolean;             -- is aircond on?

   init(heat) := 0;
   next(heat) := case {
        active & (temp=cold) : 1 ;
        ˜active | ˜(temp=cold) : 0;
   };
   init(aircond) := 0;
   next(aircond) := case {
        active & (temp=hot) : 1 ;
        ˜active | ˜(temp=hot) : 0 ;
   };
   prop1: assert G((active & temp=hot) -> aircond);
}
```

Example: Thermostat

Better properties:

```
prop2: assert G
          ((active & temp=hot) -> X aircond);
prop3: assert G
          ((active & temp=cold) -> X heat);
```

See therm2.smv

Case Statements

Case statements are evaluated in top-down order. In the thermostat example, the guards are mutually exclusive. Let's add a user override:

```
input turnheatoff: boolean;
next(heat) := case {
      active & (temp=cold): 1;
      ~active | ~(temp=cold) : 0;
      turnheatoff : 0;
 };
 p4: assert G (turnheatoff -> X ~heat);
```

Can we prove prop3 and prop4?
See therm3.smv, therm4.smv, therm5.smv

Non-Determinism

Non-determinism: more than one outcome possible.

Ways to have non-determinism in SMV models:

- Input A variable not assigned any value.
- Non-deterministic assignments
  ```
        x := {1,2,3,4};
  ```
- Undefined assignments
  A variable of undefined value may take on any value in its type. See examples next page. Note: undefined assignments are not a good idea!

Undefined Assignments

- Example: A variable that is assigned a value outside the range of its declared type:
  ```
            on : boolean;
            on := 4;
  ```
- Example: If a conditional assignment is not total (i.e., there exists a condition not covered by any clause):
  ```
  y := c1 ? x;           z := case {
                                c1 : x1;
                                c2 : x2;
                                c3 : x3;
                               };
  ```
  where $c1 \lor c2 \lor c3$ isn't a tautology.

Environmental Assumptions

- Some counter-examples exhibit undesirable system behaviour in response to invalid environmental input. These are called infeasible paths.
- Sometimes we need to constrain how environmental variables change value, in order to model a realistic environment. One option is include a model of the environment in your system description:
  ```
  next(temp) := case {
    temp=hot : {justright, hot};
    temp=justright : {cold, justright, hot};
    temp=cold : {cold, justright}
  };
  ```

The verification is valid as long as environmental assumptions are true.

Common Errors

SMV must be able to compute set of possible next states

Single Assignment Rule
Can only have one assignment statement for each variable

- The following is invalid
```
      next(x) := y;
      next(x) := z;
```
- Can assign value to:
  - x, or
  - `next(x)` and `init(x)`,

  but not both.

Common Errors

Circular Dependency Rule
Cannot have a cycle of dependencies all of whose assignments take effect immediately. That is, the system can't have a set of variables whose current values depends on the current values of each other and vice versa (can't have a combinational loop):

```
    x := y;
    y := z;
    z := x;
```

SMV should give you a syntax error for all of these mistakes.

SMV Models and Kripke Structures

An SMV model represents a Kripke structure.

```
MODULE main()
{
    request: boolean;
    status: {ready, busy};

    init(status) := ready;
    next(status) := case {
                     request : busy;
                     default: {ready, busy};
                    };
}
```

From: Huth and Ryan [HR00], p. 182–183.

SMV Modules

So far, we've only been using the “main” module.

A module bundles together definitions (type declarations and assignments). These definitions can then be reused.

Modules have three kinds of variables: input, output, private.

Input and output variables are labelled as such (keywords: INPUT and OUTPUT) and included in the list of formal parameters of the module. These variables are pass by reference.

SMV Modules

The input variables of the main module are the environmental variables.

All module variables are visible to other modules and to the environment.

However, for modularity. modules shouldn't refer to each other's private variables directly!

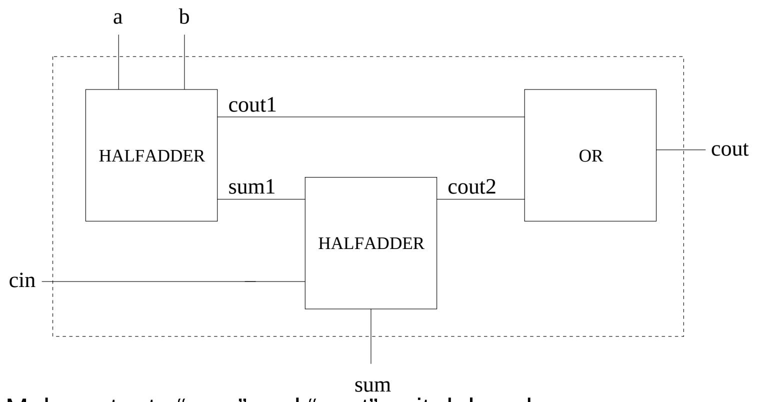
Example Module

```
MODULE half_adder(a, b, cout, sum)
{
  INPUT a, b : boolean;
  OUTPUT cout, sum : boolean;

  -- combinational
  sum := (a & ~b) | (a & ~b);
  cout := a & b;

}
```

Full Adder



Make outputs "sum" and "cout" unit delayed.
See adder.smv
Note: in our properties, we can't refer to the value at the previous time step, so we have to write more properties to describe the behaviour of an adder.

Modules with Types

"A module with only type declarations and no parameters or assignments acts like a structured data type." [McM01]

```
MODULE point(start)
{
    x,y: start..6;
}

MODULE main()
{
    mypt: point(1);
    j: 0..6;

    next(j) := mypt.x;
}
```

Today's Agenda

- CTL
- SMV
- SMV modules
- Concurrency and fairness

Concurrency

Maximum Parallelism (synchronous): All assignments are executed simultaneously, i.e., all modules perform all of their atomic assignments at the same time. This is the default in SMV.

Interleaving Parallelism (asynchronous): Module executions are interleaved. Each module performing all of its atomic assignments in isolation. Multiple modules do not execute in the same step. In the modules that aren't executing in a step, the variables do not change their values.

Interleaving in SMV

To have modules interleave their execute, use the SMV keyword `process`.

In the counterexamples, a new variable `running` appears for each module and for the overall system.

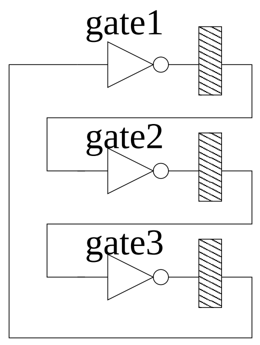
When `running` is 1 for a module, it means that module is running.

When `running` is 1 for the system, it means no module is running. (This is always an option for interleaving.)

Only one `running` variable can be true at a time.

Note: this keyword isn't mentioned in the latest SMV documentation.

Concurrency Example



Normally hardware would run in parallel mode (synchronous), but in this example each gate is in a separate module that we will run in interleaving mode.

Src: McMillan [McM92]

Concurrency Example (interleaving1.sml)

```
MODULE main()
{
  gate1: process invertor(gate3.out, gate2.inp);
  gate2: process invertor(gate1.out, gate3.inp);
  gate3: process invertor(gate2.out, gate1.inp);

  p1: assert G (F gate1.out);
  p2: assert G (F !gate1.out);
}

MODULE invertor(inp,out)
{
  INPUT inp:boolean;
  OUTPUT out:boolean;

  init(out) := 0;
  next(out) := !inp;
}
```

Fairness

We want to limit the set of paths considered to those that are *fair*, i.e., those where all processes gets a chance to execute. For example, an arbiter shouldn't ignore one of its input requests all the time.

There are multiple kinds of fairness constraints. (Here, we are speaking more generally than just SMV.)

We can talk about fairness constrains in terms of processes, which are enabled and occur.

"occurs" means the process runs (In SMV, this means the variable "running" is true for that process.)

"enabled" means the process is able to do something (This doesn't have a good match in SMV, it's more intended for event-based languages where an event triggers some behaviour.)

Types of Fairness Properties

Impartiality: every process is executed infinitely often.

Weak Fairness (justice): every process is either infinitely often disabled or else is infinitely often executed.

Strong Fairness (compassion, fair): if a process is infinitely often enabled it will occur infinitely often

Fairness and LTL

These fairness can be expressed in LTL:

Impartiality: **GF** oc
A path must contain a state that satisfies oc infinitely often.

Weak fairness (justice): **GF** $(en \Rightarrow oc)$
A path must then contain a state that satisfies $en \Rightarrow oc$ infinitely often.

Strong fairness (compassion, fair): $($**GF** $en) \Rightarrow ($**GF** $oc)$
If the path satisfies en infinitely often then it also must satisfy oc infinitely often.

These can be used as antecedents of the property that we want to check.

Fairness and CTL

Fairness constraints can't be expressed in CTL, and therefore they have to be added separately.

This was true in the old SMV, which only model checked CTL, but the old SMV included a way to add the impartiality and justice constraints.

Fairness Example (Impartiality)

```
p1: assert (G F gate1.running &
            G F gate2.running &
            G F gate3.running)
                 -> G (F gate1.out);
p2: assert (G F gate1.running &
            G F gate2.running &
            G F gate3.running)
                 -> G (F !gate1.out);
```

See interleaving2.sml

Today's Agenda

- CTL
- SMV
- SMV modules
- Concurrency and fairness

References

[HR00] Michael R. A. Huth and Mark D. Ryan. *Logic in Computer Science*. Cambridge University Press, Cambridge, 2000. DC Library: QA76.9.L63 H88 2000.

[McM92] Kenneth L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, May 1992.

[McM01] K. L. McMillan. The smv language, 2001. On-line language reference manual.