The Model Checker SMV

Somesh Jha
Computer Sciences Department
University of Wisconsin
Madison, WI 53706

1 A Simple Example

We will introduce the model checker SMV using a simple example. The example is two process mutual exclusion.

1.1 The `main` module

The `main` module is what SMV uses to construct the model. The `main` module is analogous to the `main` function in C. All the global variables should appear in the `main` module and all other modules are instantiated in this module. Consider the fragment of SMV code shown in Figure 1.

```
MODULE main
VAR
  semaphore: boolean;
  proc1: process user(semaphore);
  proc2: process user(semaphore);
```

Figure 1: A fragment of SMV code

Here are some points to notice from Figure 1.

- The name of the main module is `main`.
- All variables are declared after the keyword `VAR`.
- Variable `semaphore` is of type `boolean`.
- Variable `proc1` is a state-machine of type `user`. In other words, `proc1` is an instantiation of the definition `user`. This is similar to instantiating a class definition to construct an object instance in object oriented languages such as `Java`.
- `MODULE user` will be discussed later.
- Notice that `semaphore` is passed as a parameter to `user`. We will discuss this bug in detail later.

1.2 What does the `process` keyword mean?

The keyword `process` instructs SMV to use *asynchronous* composition. This means that at every step either a transition from `proc1` or `proc2` (but not both) is *taken* or *executed*. This is what leads to a bug in the program. We will discuss this later.

1.3 Declaring transitions

Transitions and the initial state of the system are described after the keyword `ASSIGN`. In case of the `main` module we only define the initial value for the semaphore.

```
ASSIGN
   init(semaphore) := 0;
```

1.4 Specifications in CTL

Specifications in SMV are written in CTL and follow the keyword `SPEC`. One can have multiple specifications, but the keyword `SPEC` should appear before every specification. In this example, we only have one specification. The CTL specification is shown in Figure 2. (**Question:** Can you describe what the specification expresses?)

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

Figure 2: A CTL specification

1.5 The `user` module

The `user` module is a *template* or a *definition* of a model. Notice that the type for parameter `semaphore` is not specified in the declaration. SMV will figure out the type for `semaphore` while constructing the model. A fragment of the declaration for the `user` module is shown in Figure 3.

```
MODULE user(semaphore)
VAR
state : {idle,entering,critical,exiting};
```

Figure 3: A Fragment of the `user` module definition

Variable `state` is an enumerated type and can have any of the four specified values. Internally, SMV codes everything using boolean variables.

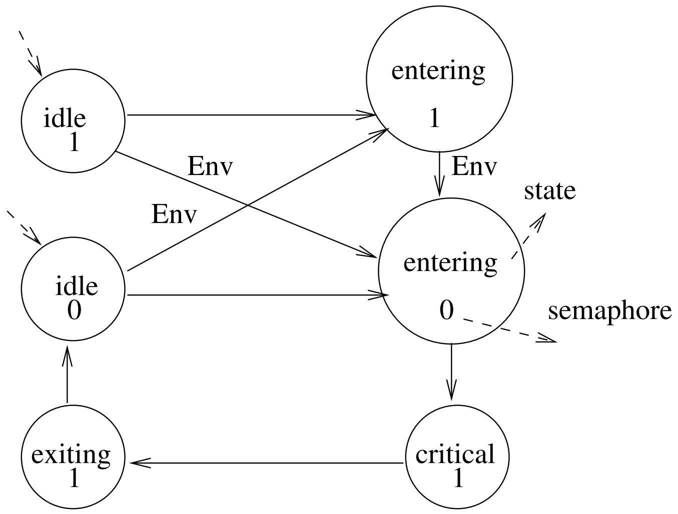

Figure 4: User state diagram

1.6 `FAIRNESS` condition

Recall that the system will arbitrarily pick one transition from `proc1` and `proc2` and execute it. Given no restrictions, there might be paths where a process (say `proc1`) never gets to *execute* `FAIRNESS running` (see the end of the `MODULE user` definition) precludes that. SMV has an internal variable for each process (called `running`) which is set equal to *true* when a transition from that process executes. The fairness condition states that the variable `running` should be true infinitely often along a path. Formally, we only explore *fair paths* while verifying the specification, where a fair path is a path where the fairness condition is true infinitely often.

1.7 Running SMV

If we execute SMV on the specification just described, we obtain the following counter-example. The counter-example is simply shown as a sequence of states. The next state in the sequence only shows variables that have changed and not the entire state, i.e., bindings of all the variables in the state. Notice that at the end of the counter-example a synopsis is also printed.

```
-- 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
[stuttering]

state 1.2:
[executing process proc1]

-- loop starts here --
```

```
state 1.3:
proc1.state = entering
[stuttering]

state 1.4:
[executing process proc2]

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

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

state 1.7:
[executing process proc2]

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

state 1.9:
semaphore = 0
proc2.state = idle
[stuttering]


resources used:
user time: 0.0833333 s, system time: 0.166667 s
BDD nodes allocated: 1202
Bytes allocated: 1245184
BDD nodes representing transition relation: 69 + 1
```

1.8 Structure of the counter-example

The negation of the specification is given below:

$$\begin{array}{l}\mathbf{EF}(\text{proc1.state} = \text{entering} \wedge \\ \mathbf{EG}(\text{proc1.state} \neq \text{critical}))\end{array}$$

A trace where the property given above is true *is the* counter-example for the specification given earlier. Next we explain the counter-example in detail.

- State 1.1

Variables `semaphore`, `proc1.state`, and `proc2.state` are 0, `idle`, and `idle` respectively.

- State 1.2
  Same state as 1.1. SMV only shows variables that change in the transition. We execute a transition from `proc1`.

- State 1.3
  Loop or a cycle is formed by states 1.3 through 1.9. Notice that on this cycle `proc1.state` is never equal to `critical` and `proc1` changes its state to `entering`.

- State 1.4
  Same state as 1.3 but going to execute a transition from `proc2`.

- State 1.5
  Process `proc2` changes state to `entering` and we are going to execute a transition from process `proc2`.

- State 1.6
  Process `proc2` sets the `semaphore` to 1 and moves to the `critical` state. Going to execute `proc1`.

- State 1.7
  Semaphore is set to 1 so `proc1` stays in `entering` state. We are going to execute `proc2`.

- State 1.8
  Process `proc2` moves to the `exiting` state. We are going to execute a transition from `proc2`.

- State 1.9
  Variable `semaphore` reset to 0 and `proc2` moves to the `idle` state. We can stay in this state for arbitrarily long time (this indicated by `stuttering` in the example). Notice that this is the same state as state 1.3. Therefore, we have discovered a loop.

Here are some points to notice about the counter-example.

- Process `proc1` was never in the `critical` state in the loop.
- In the loop process `proc1` did execute (state 1.6 to 1.7). Hence `FAIRNESS running` is true. Process `proc1` could not do much because the `semaphore` was set to 1 by `proc2`.
- Process `proc1` was *stuck in* the state `entering` and was never chosen to make the transition to the `critical` state.

The counter-example suggest the following fix:

Assert that process is not in the state `entering` *infinitely often.*

We achieve this by changing the *fairness* constraint to:

```
FAIRNESS
running & !(state=entering)
```

After the fix is made, SMV says that the specification is true and prints out a synopsis.

```
-- specification AG (proc1.state = entering -> AF proc1.s... is true

resources used:
user time: 0.0833333 s, system time: 0.133333 s
BDD nodes allocated: 615
Bytes allocated: 1245184
BDD nodes representing transition relation: 69 + 1
```

2 A Vending machine model

Next we discuss a model of a vending machine which is slightly more complicated than the mutual-exclusion example discussed earlier. We only provide a brief synopsis of the example. The reader should consult the entire SMV code for additional details.

2.1 A Brief Synopsis

A vending machine model has one user and one vending machine. We assume that it takes two coins to buy a beverage. The state variables corresponding to the model of the user are:

- variable `state` has the following set of values:
  - `initial, one-coin, two-coins, making-choice, waiting`
- variable `choice` has the following set of values:
  - `none, coke, diet-coke, sprite, mountain-dew`

The model for the vending machine has the following variables:

- variable `state` has the following set of values:
  - `initial, one-coin, two-coins, get-choice, dispense`

Initial values for the state variables for the model of the user are:

- initial value for the variable state
  `state := initial;`
- initial value for the variable choice
  `choice := none;`

Transitions for the user model are listed below:

- if `state` is `initial` and state of the vending machine is in state `initial`, stay in `initial` or transition to `one-coin` using *non-determinism*

```
state = initial & (vending-machine-state = initial) :
   { initial, one-coin };
```

- if state is `one-coin`, move to `two-coins` indicating that the user has deposited the second coin
- If the state is `making-choice`, then user makes choice of beverage (see the transitions for the variable `choice`). The choice between beverages is completely non-deterministic.
- After the beverage has been dispensed, the value of the variable `state` becomes `none`.

Initial values for the state variables of vending machine model are:

- initial value of `state` is `initial`
- `user-state` and `choice` are passed as parameters from the user module. These parameters make the vending machine aware of what the user is doing.

Transition for the vending machine are only a few. Locate all the transitions in the code. We only describe two transitions below.

- change state from `initial` to `one-coin` if the `user-state` is `one-coin`, i.e., the vending machine is recording that the user has deposited one coin.
- transition to state `dispense` after `get-choice`. This corresponds to the vending machine dispensing their drink.

2.2 Macros

In SMV you can define *macros* using the keyword `DEFINE`. In module `vending-machine` we define a macro `DISPENSED`, which describes that the vending machine is dispensing a beverage.

```
DEFINE
   DISPENSED := state = dispense;
```

2.3 Instantiating the modules

The vending machine model is created by instantiating the modules `vending-machine` and `user`. As we already pointed out this is equivalent to instantiating a class definition in object oriented programming. The fragment of the code in `MAIN` module is shown in Figure 5.

```
MODULE main
VAR
  machine: vending-machine(wisc-user.state,
                           wisc-user.choice);
  wisc-user: user(machine.DISPENSED,
                  machine.state);
```

Figure 5: Fragment of the main module.

```
SPEC
  AG(((wisc-user.state = one-coin) &
      (machine.state = initial))
      -> AF(machine.state = dispense))
```

Figure 6: Specification for the vending machine

2.4 Specification

We want to state that *if* the user state is `one-coin` and vending machine state is `initial`, *then always eventually* vending machine state is `dispensed`, i.e., a user always eventually gets his/her drink. This specification is shown in Figure 6.

3 Enumerating behaviors

Model checkers are very useful for demonstrating scenarios with certain properties. For example, suppose you construct a model of a distributed database system. You, the designer, might want to see a trace where a transaction is issued and then eventually finishes. The designer can then visualize what steps the system takes to finish a transaction. For example, in order to gain insight into the vending machine module suppose a designer wants to see a trace where the vending machine dispenses a beverage. The negation of the property corresponding the vending machine not dispensing a beverage is:

```
!machine.DISPENSED
```

The designer writes a property asserting that `!machine.DISPENSED` is never true. This specification is shown below.

```
SPEC
  AG(!machine.DISPENSED)
```

The Negation of the property is **EF**(machine.dispensed). Hence, the counterexample to the previous spec is a *trace* where the vending machine eventually dispenses.