Lecture 4 (Introduce SMV)
Analysis of Software Artifacts
Somesh Jha

`MAIN` module

- This is what SMV uses to build a model. Very similar to the `main` function in C.
- Put all your global variables in the `MAIN` module.
- Instantiate all modules here.
- Consider the following fragment of the SMV code:

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

`MAIN` module (Contd)

- The name of the `MAIN` module is `main`.
- After the keyword `VAR` declare all your variables.
- Variable `semaphore` is of type boolean.
- `proc1` is a component/state-machine of type `user`.
- `MODULE user` will be defined later.
- Notice that `semaphore` is passed as a parameter to `user`. Will become clear later.

What is that `process` thingy?

- The keyword `process` tells 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 creates the bug. Will get to that later.

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;
```

Specifications in CTL

- Specifications are written in CTL and follow the keyword `SPEC`.

- You can have multiple specifications. Here we have only specification.

- The spec looks like:

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

- What does it say?

The `user` module

- The `user` module is a *template* or a `type` of a state machine.

- Notice that no type for parameter `semaphore` is specified in the declaration.

- SMV will figure out the type. I don't like this.

The `user` module (Contd)

- The declaration for the `user` module is:

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

- Variable `state` is an enumerated type and can have any of the four specified values.

- Internally, SMV codes everything as booleans.

`FAIRNESS` condition

- Recall that the system will pick one of `proc1` and `proc2` arbitrarily and execute a transition from that process.
- 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.

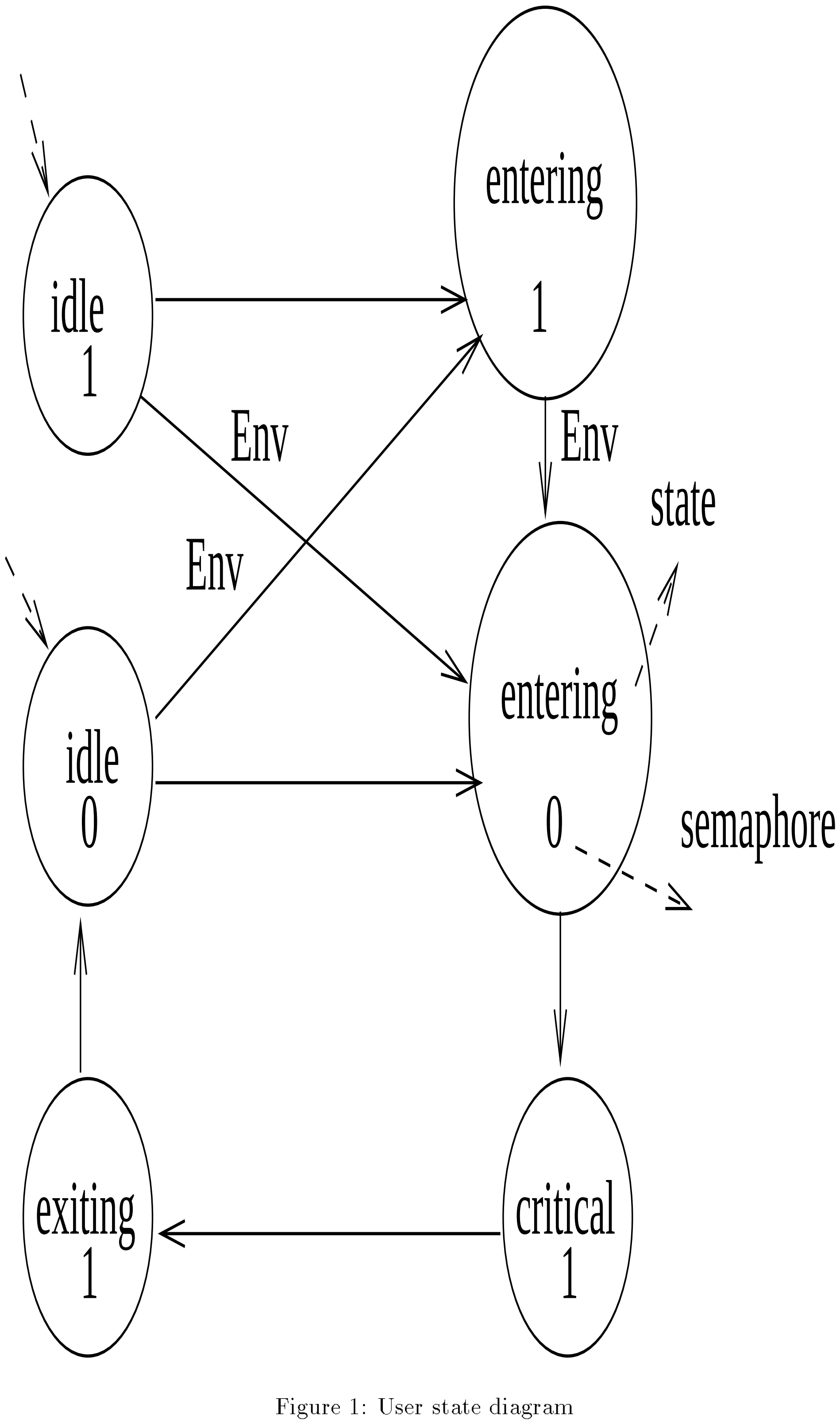

Figure 1: User state diagram

Running SMV

```
-- 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
```

Structure of the counter-example

- Negation of the specification looks like

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

- How does the counter-example look?

Counter-example explained

- 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. `proc1` changes its state to `entering`.

Counter-example (Contd)

- 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`.

Counter-example (Contd)

- 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 `idle` state. We can stay in this state for arbitrarily long time (`stuttering`). Notice that this is the same state as State 1.3. We have a loop.

Points to notice

- 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. Poor `proc1` couldn't do much because the `semaphore` was set to 1 by `proc2`

Explaining the counter-example

- Process `proc1` was *stuck in* the state `entering` and was never chosen to make the transition to the `critical` state.

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

- Change the *fairness* constraint to:

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

Everything is fine

- SMV says that the specification is true:

```
-- 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
```