SMV Tutorial

CS4211 - Formal Methods for Software Engineering

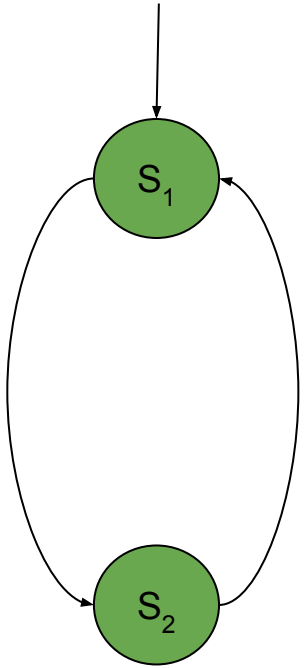
Prepared by: Ridwan Shariffdeen

Writing First Program

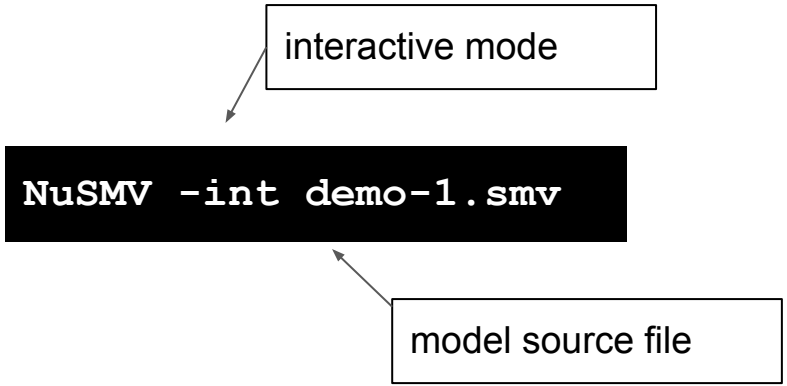
Simple Transition

Demo Code

```
MODULE main
VAR
    state: {s1, s2};

ASSIGN
    init(state) := s1;
    next(state) := case
             state = s1 : s2;
             state = s2 : s1;
           esac;
```



Build Model



Commands:

- go
  - read_model
  - flatten_hierarchy
  - encode_variables
  - build_model
- goto_state
- pick_state
- print_current_state
- print_reachable_states
- simulate
- show_traces

Load & Build Model

```
NuSMV > read_model -i demo-1.smv
NuSMV > flatten_hierarchy
NuSMV > encode_variables
NuSMV > build_model
```

```
NuSMV > go
```

Describe a Model

```
NuSMV > print_reachable_states -v
```

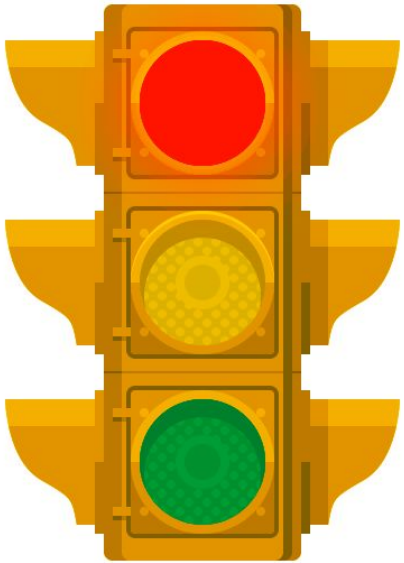
```
###############################################################################
system diameter: 2
reachable states: 2 (2^1) out of 2 (2^1)
  ------- State     1 ------
  state = s2

  ------- State     2 ------
  state = s1

  -------------------------
###############################################################################
```

Infinite Loop

Traffic Light

Traffic Light Code

```
MODULE main
VAR
    light: {red, green, yellow};
ASSIGN
        next(light) := case
                           light = red : green;
                           light = green : yellow;
                           light = yellow : red;
                       esac;
```



Run a Simulation (random)

```
NuSMV > pick_state -r
NuSMV > simulate -r -k 5
NuSMV > print_current_state -v
```

```
********  Simulation Starting From State 1.1   ********
Current state is 1.6
light = red
```

Show Trace

```
NuSMV > show_traces -t
NuSMV > show_traces 1
```

```
<!-- ################## Trace number: 1 ################## -->
Trace Description: Simulation Trace
Trace Type: Simulation
  -> State: 1.1 <-
     light = green
  -> State: 1.2 <-
     light = yellow
  -> State: 1.3 <-
     light = red
  -> State: 1.4 <-
     light = green
```

Run a Simulation (interactive)

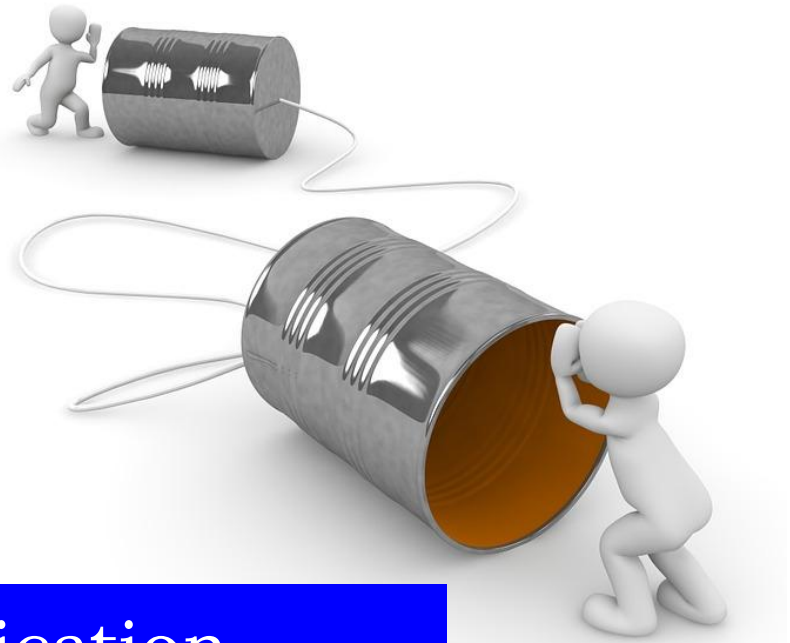
```
NuSMV > pick_state -i
NuSMV > simulate -i -k 5
NuSMV > print_current_state -v
```

```
********  Simulation Starting From State 1.1   ********
Current state is 1.6
light = green
```

Revisit a State and Replay

```
NuSMV > goto_state 1.11
NuSMV > simluate -r -k 4
NuSMV > show_traces -v
```

```
   <!-- ################## Trace number: 3 ################## →
Trace Description: Simulation Trace
Trace Type: Simulation
  -> State: 3.1 <-
     light = green
  -> State: 3.2 <-
     light = yellow
  -> State: 3.3 <-
     light = red
```



Process Communication

Alternate Bit Protocol

Tutorial - Spin

CS4211 - Formal Methods for Software Engineering

NUS National University of Singapore

ABP Code

```
MODULE main
VAR
    toS: boolean;
    toR: boolean;
    rcvr: process receiver(toR, toS);
    sndr: process sender(toS, toR);

ASSIGN
    init(toS):= FALSE;
    init(toR):= FALSE;
```

```
MODULE receiver(input,output)
ASSIGN
    next(output) := input;
FAIRNESS
    running
```

```
MODULE receiver(input,output)
ASSIGN
    next(output) := !input;
FAIRNESS
    running
```

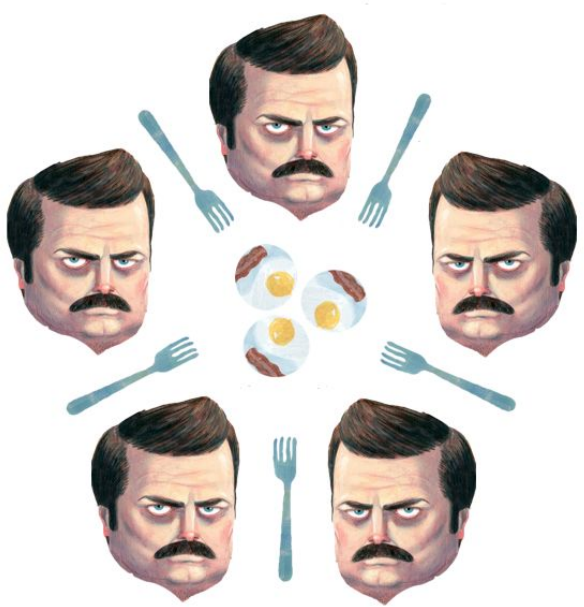
Multiple Processes

Dining Troopers

Dining Philosophers Problem

The dining philosophers problem is a classic concurrency problem dealing with synchronization.

Source:
http://adit.io/posts/2013-05-11-The-Dining-Philosophers-Problem-With-Ron-Swanson.html



Dining Philosophers Problem

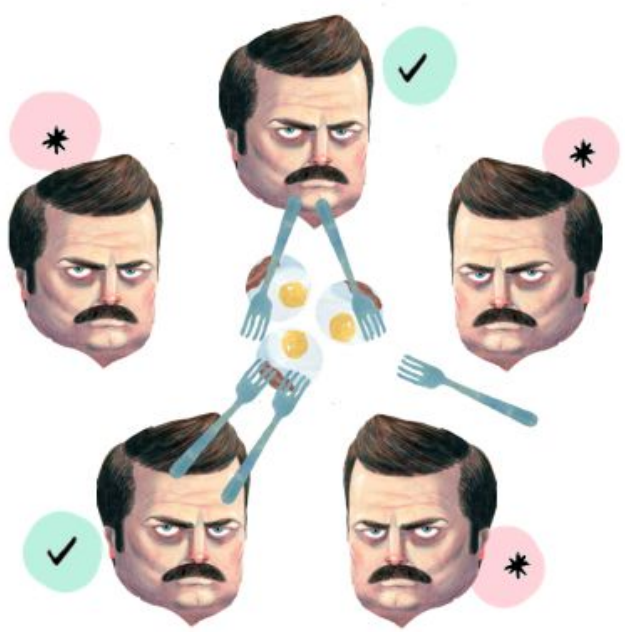
Now, each philosopher has two forks: left fork and right fork. If a Philosopher gets two forks, he can eat!

If Ron gets two forks, he can eat!

If he only has one fork he can't eat :(

Dining Philosophers Problem

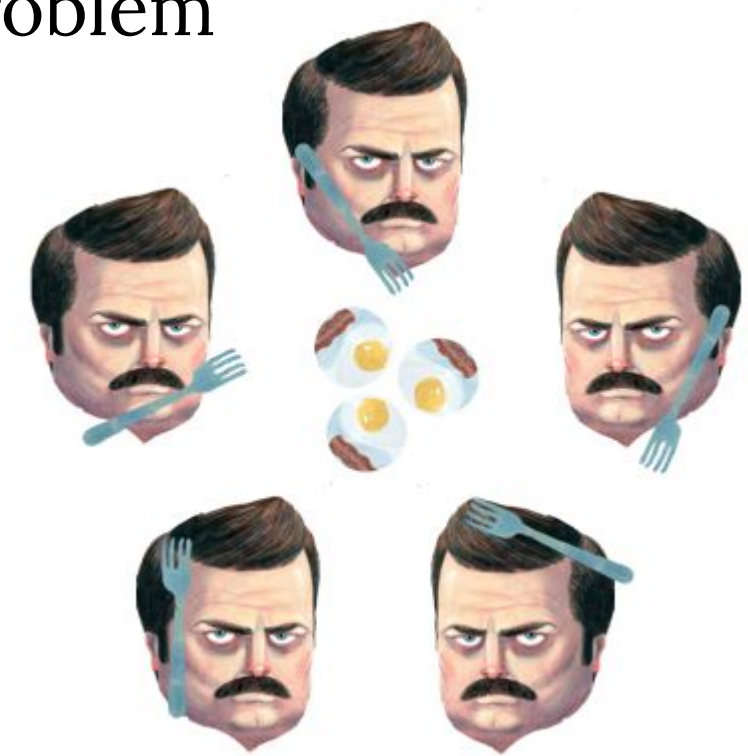
The dining philosophers problem is: **how do you make sure every Philosopher gets to eat?**



Dining Philosophers Problem

every Philosopher grabbed a fork: and then they waited for someone to give up their fork so they could eat. But of course, "a Philosopher never gives up his fork!" *sigh* So they waited forever and eventually died in their cabin.

Great job, guys. When all Philosophers are stuck, that is called ***deadlock***.



Dining Troopers Code

```
MODULE trooper(forks)
VAR
     state: {not-eating, eating, waiting};
ASSIGN
     init(state) := not-eating;
     next(state) := case
                         state = not-eating & forks > 0 : waiting;
                         state = waiting & forks > 0 : eating;
                         state = eating : not-eating;
                         TRUE: state;
                    esac;
     next(forks) := case
                    state = not-eating & forks > 0 : forks -1;
                    state = waiting & forks > 0 : forks -1;
                    state = eating :  2;
                    TRUE: forks;
                    esac;
```

Dining Troopers Code

```
MODULE main
VAR
     forks: 0..2;
     troop_1: process trooper(forks);
     troop_2: process trooper(forks);

ASSIGN
     init(forks) := 2;
```

Dining Troopers

Is everything implemented correctly?

How can we verify our implementation is correct?

Dining Troopers Code

```
LTLSPEC G !((troop_1.state = eating) & (troop_2.state = eating))
LTLSPEC G(F (troop_1.state = eating) | F (troop_2.state = eating))
```

```
NuSMV troops.smv
-- specification  G !(troop1.state = eating & troop2.state =
eating)  is true

-- specification  G ( F troop1.state = eating |  F
troop2.state = eating)  is false

-- as demonstrated by the following execution sequence

Trace Description: LTL Counterexample
Trace Type: Counterexample
```

DEADLOCK

Dining Troopers Code

```
MODULE trooper(forks)
VAR
     state: {not-eating, eating, waiting};
ASSIGN
     init(state) := not-eating;
     next(state) := case
                         state = not-eating & forks > 0 : waiting;
                         state = waiting & forks > 0 : eating;
                         state = waiting & forks = 0 : not-eating;
                         state = eating : not-eating;
                         TRUE: state;
                    esac;
     next(forks) := case
                    state = not-eating & forks > 0 : forks -1;
                    state = waiting & forks > 0 : forks -1;
                    state = waiting & forks = 0 : forks + 1;
                    state = eating :  2;
                    TRUE: forks;
                    esac;
```

Dining Troopers Code

```
LTLSPEC G !((troop_1.state = eating) & (troop_2.state = eating))
LTLSPEC G(F (troop1.state = eating) | F (troop2.state = eating))
```

```
NuSMV troops.smv
-- specification  G !(troop1.state = eating & troop2.state =
eating)  is true

-- specification  G ( F troop1.state = eating |  F
troop2.state = eating)  is false

-- as demonstrated by the following execution sequence

Trace Description: LTL Counterexample
Trace Type: Counterexample
```

LIVELOCK

Dining Troopers Code

```
MODULE trooper(forks)
VAR
     state: {not-eating, eating};
ASSIGN
     init(state) := not-eating;
     next(state) := case
                         state = not-eating & forks = 2 : eating;
                         state = eating : not-eating;
                         TRUE: state;
                    esac;
     next(forks) := case
                    state = not-eating & forks = 2 : 0;
                    state = eating :  2;
                    TRUE: forks;
                    esac;
```

Dining Troopers Code

```
LTLSPEC G !((troop_1.state = eating) & (troop_2.state = eating))
LTLSPEC G(F (troop1.state = eating) | F (troop2.state = eating))
LTLSPEC G(F (troop1.state = eating) & F (troop2.state = eating))
```

```
NuSMV troops.smv
-- specification  G !(troop1.state = eating & troop2.state =
eating)  is true

-- specification  G ( F troop1.state = eating |  F
troop2.state = eating)  is true

-- specification  G ( F troop1.state = eating &  F
troop2.state = eating)  is false

-- as demonstrated by the following execution sequence
```

STARVATION

Q&A

Thank You!