Introduction to Model Checking and NuSMV

Cyrille Artho and Musard Balliu
KTH Royal Institute of Technology, Stockholm, Sweden
School of Electrical Engineering and Computer Science
Theoretical Computer Science
artho@kth.se

2019-03-27

Summary of last lecture

Temporal logics

Linear temporal logic (LTL):

- No branching.
- Defined on paths.

Computational tree logic (CTL):

- Branching.
- Defined on transition systems.

LTL $\neq$ CTL

1. **FG**(p)
2. **EX**(p)

- *Which logic can express the formulas above?*
- *What is the semantics of each formula?*
- *Where does the counterpart fail to express it?*

CTL*: Combines LTL and CTL.

Safety vs. Liveness

Safety: Something bad will never happen.

Ensures absence of defects and hazards.

Liveness: Something good eventually happens.

Ensures progress.

Which temporal logic operators are suitable for which type of property?

Outline of today's lecture

1. Introduction to NuSMV.

2. NuSMV by example.

Ferryman puzzle

 $\xrightarrow{?}$

- Ferryman wants to cross river with cabbage (c), goat (g), wolf (w).
- Goat will eat cabbage when left alone; wolf will eat goat.
- Ferry carries only one „passenger”.

Can the ferryman bring all things to the other side, safely?

Wolf, goat, ferryman, river icons made by Freepik; cabbage icon made by Nikita Golubev; icons from www.flaticon.com

NuSMV model of the ferryman puzzle state

```
-- Ferryman by Bow-Yaw Wang
MODULE main
VAR
  ferryman : boolean;
  goat : boolean;
  cabbage : boolean;
  wolf : boolean;
  carry : { g, c, w, n };
```

```
ASSIGN
  init (ferryman) := FALSE;
  init (goat) := FALSE;
  init (cabbage) := FALSE;
  init (wolf) := FALSE;
  init (carry) := n;
```

- Boolean variables **`ferryman`**, **`goat`**, **`cabbage`**, **`wolf`** denote the location of the ferryman, goat, cabbage, wolf.
- Initially, all are on the same side (**`FALSE`**).
- The variable carry denotes the good carried by the ferryman: **`g`** (goat), **`c`** (cabbage), **`w`** (wolf), or **`n`** (none).

Modeling the ferryman and his passengers

```
next (ferryman) := { FALSE, TRUE };
next (goat) := case
  ferryman = goat & next (carry) = g: next (ferryman);
  TRUE: goat;
esac;
next (cabbage) := case
  ferryman = cabbage & next (carry) = c: next (ferryman);
  TRUE: cabbage;
esac;
next (wolf) := case
  ferryman = wolf & next (carry) = w: next (ferryman);
  TRUE: wolf;
esac;
```

- The ferryman is non-deterministic (we don't know the right strategy).
- The passengers follow the ferryman iff he carries them to other side.

Modeling the possibility to carry a passenger

```
TRANS
   (next(carry) = n) |
   (ferryman = goat & next(carry) = g) |
   (ferryman = cabbage & next(carry) = c) |
   (ferryman = wolf & next(carry) = w);
```

- The ferryman can carry nothing, or. . .
- starts from the same place as the item he will carry in the next turn.
- **`TRANS`** is another way to model transition relation.

How to describe the puzzle and its solution?

- Remember: ferryman needs to watch if goat is together with cabbage or wolf.
- Therefore: if goat is one the same side as cabbage or wolf, ferryman must be on that side, too.
- Once all four are on the other side, puzzle is solved.
- Use **until** operator:

rules of puzzle are followed **U** *solution achieved.*

Encoding the puzzle in LTL

((goat = cabbage | goat = wolf) $\rightarrow$ goat = ferryman) *[rule]*

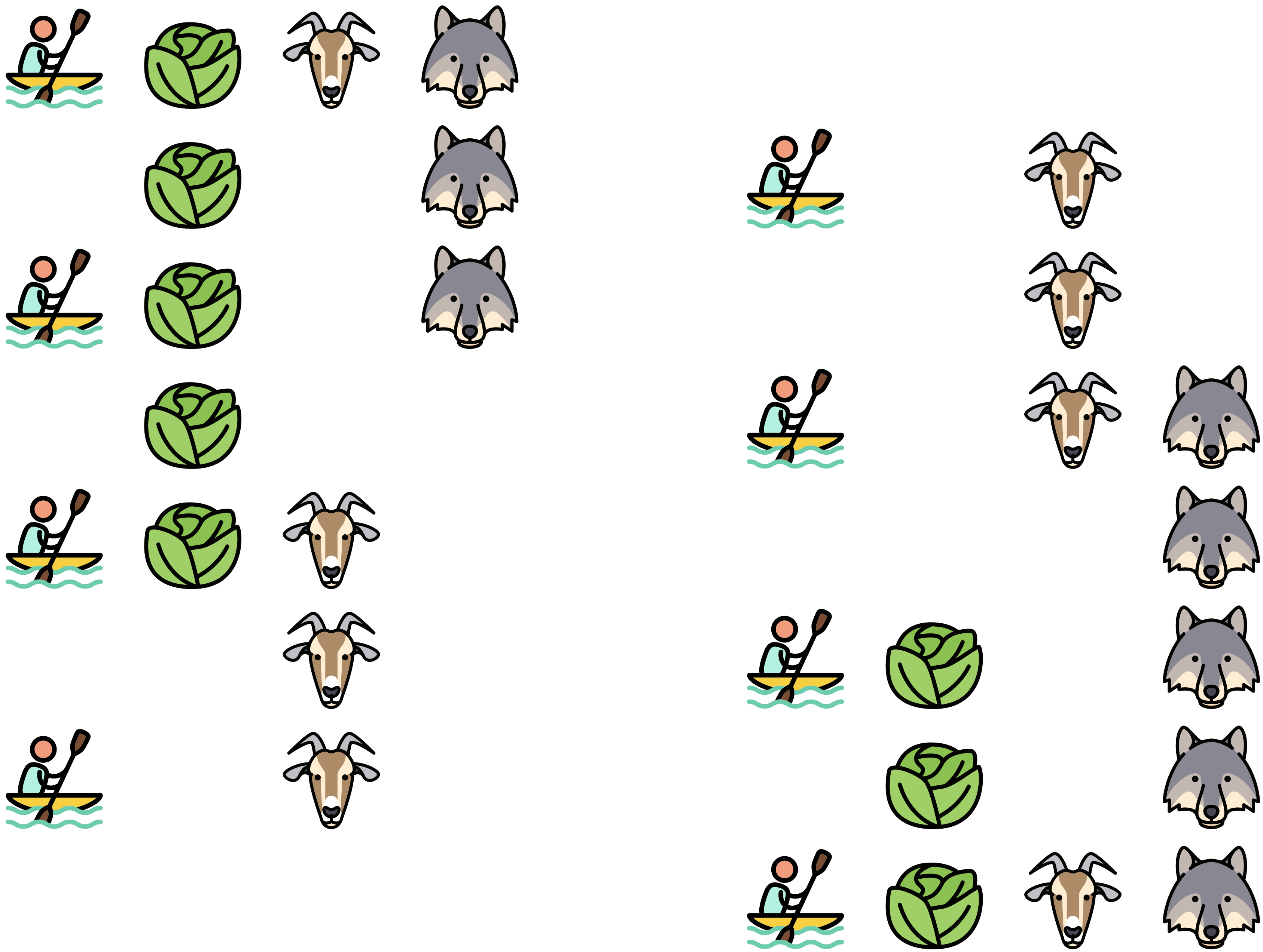
U (cabbage & goat & wolf & ferryman) *[solution]*

We want to see the solution!

We negate the whole property, stating
„I can't follow the puzzle rules **until** the solution is achieved".

```
!( ((goat = cabbage | goat = wolf) -> goat = ferryman)
    U (cabbage & goat & wolf & ferryman) )
```

Counterexample = solution



Another bridge crossing puzzle: „Bridge and torch problem"

- A, B, C, and D want to cross a bridge at night.
- They have only one weak torch that gives light for up to two people.
- Torch must be carried across the bridge (cannot be thrown across).
- The time taken for each crossing depends on the slowest person:

A	1
B	2
C	5
D	10

- Can all four cross within 17 time units?

An example crossing

Left side		Right side	Time
A B C D			0
		A B	2
A C D		B	3
D		A B C	8
A D		B C	9
		A B C D	19

Can we do better?

NuSMV model: Variables

- Location of A, B, C, D are booleans (or an array of booleans).
- Another array of booleans denote of A, B, C, D are *traveling*.
- Torch location is also a boolean.
- Time is a number between 0 and 100.

Some transitions

- Torch can change location only if someone travels.
  Also possible to model that torch always changes location until solution achieved, since we are interested in efficient solutions, and time does not increase if nobody moves.

- Choice of who travels is not specified.

- Location of A, B, C, D is updated iff

  1. they want to travel,
  2. the torch is at their place.

Timekeeping

- Time advances according to the slowest person who travels.
  If you model torch moves as optional (see above), then ensure that „empty moves” do not count towards time, by incrementing time only if location(x) != next(location(x));

- You need to have a non-overflow rule at top, e. g.,

```
next(time) := case
  time > 90: 90;
```

The final formula and game rule

- At most two people travel: use **`count`**:
  `count(..., ..., ...)`
  returns the number of true predicates in the list.
- A, B, C, D have to arrive at the other side within N time units.
- Use !„follow game rules **U** goal" as template;
  solution is counterexample.
- You can add other consistency checks, such as
  `G (location[0] & location[1] & location[2] & location[3] -> torch)`
- The time limit is another conjunct in the goal condition.
- What happens if you lower the time limit further?

Lab exercise 2 (in assignment 1)

1. Develop the full game model and the formula to find the solution.

2. Study the resulting trace(s) and show the optimal solution.

Semaphores

Photo by Dave F

- Inspired by railway signal.
- Binary semaphore:
  Resource can be available or in use.
- Value of semaphore guards access to exclusive shared resource („critical section” in concurrent code).
- Anyone who wants access to resource in use, has to wait.

Semaphore model

```
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;
    TRUE : state;
  esac;
  next(semaphore) := case
    state = entering: TRUE;
    state = exiting : FALSE;
    TRUE : semaphore;
  esac;
```

- Process („user") may only acquire semaphore when not in use.

Semaphore model – 2

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

- System is *asynchronous:* only one process runs at a time!
- Therefore, semaphore is only updated by active process; this ensures that no two processes can obtain it at once.

Liveness property does not hold!

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

```
Trace Type: Counterexample
  -> State: 1.1 <-
    semaphore = FALSE
    proc1.state = idle
    proc2.state = idle
  -> Input: 1.2 <-
    _process_selector_ = proc1
    running = FALSE
    proc2.running = FALSE
    proc1.running = TRUE
  -- Loop starts here
  -> State: 1.2 <-
    proc1.state = entering
  -> Input: 1.3 <-
    _process_selector_ = proc2
    proc2.running = TRUE
    proc1.running = FALSE
  -> State: 1.3 <-
```

- Trace shows interleaving between processes.
- Process 1 is selected, tries to acquire semaphore.
- After that, process 2 is always selected.
- Process 1 never gets another turn.

This is not fair!

- A good „scheduler” would always eventually run process 1.
- Error traces from unfair runs are not always relevant.
- We can forbid such scenarios with *fairness constraints:*

```
FAIRNESS running;
```

- Only traces where **`running`** is true infinitely often are considered.

It is possible to list multiple fairness conditions!

Another error trace

```
-> State: 1.1 <-
  semaphore = FALSE
  proc1.state = idle
  proc2.state = idle
-> Input: 1.2 <-
  _process_selector_ = proc1
  running = FALSE
  proc2.running = FALSE
  proc1.running = TRUE
-- Loop starts here
-> State: 1.2 <-
  proc1.state = entering
-> Input: 1.3 <-
  _process_selector_ = proc2
  proc2.running = TRUE
  proc1.running = FALSE
-- Loop starts here
-> State: 1.3 <-
-> Input: 1.4 <-
-> State: 1.4 <-
  proc2.state = entering
-> Input: 1.5 <-
-> State: 1.5 <-
  semaphore = TRUE
  proc2.state = critical
-> Input: 1.6 <-
  _process_selector_ = proc1
  proc2.running = FALSE
  proc1.running = TRUE
-> State: 1.6 <-
-> Input: 1.7 <-
  _process_selector_ = proc2
  proc2.running = TRUE
  proc1.running = FALSE
-> State: 1.7 <-
  proc2.state = exiting
-> Input: 1.8 <-
-> State: 1.8 <-
  semaphore = FALSE
  proc2.state = idle
```

What goes wrong here?

Analyze the error trace with your lab partner(s).

- What happens in this error trace?
- What is the problem (the infinite loop)?
- Is the fairness condition fulfilled?
- What could be changed to avoid such behavior?

Processes in NuSMV

- Used to model systems where only one component is active at a time.
- No „global clock" (synchronous behavior): *asynchronous systems.*
- Problem:

```
WARNING *** Processes are still supported, but deprecated.      ***
WARNING *** In the future processes may be no longer supported. ***
```

Write your own model code to "schedule" modules.

How to model processes with variables

- NuSMV selects the active process in between regular model steps.
- We need a variable that holds the ID of the active process:

```
VAR running: 0..N;
```

- We cannot interleave the choice of „running" with other modules!

Use `next(running) = PID` as scheduling predicate.

Lab exercise 3 (assignment 1): Change in semaphore model

1. User module

```
MODULE user(semaphore, active)
-- state transitions only fire when active
```

2. Semaphore module

- Semaphore is now executed globally, once per turn.
- Condition needs to be repeated for each user instance, but...
  state only changes if condition is true for active process!

Adapt semaphore model and update the fairness conditions.

Guidance for exercise 3

1. Transition rule of module „user” easy to adapt:
   State changes only if „active” is true.

2. Process scheduling:
```
running: 0..1;
proc1 : user(semaphore, next(running) = 0);
proc2 : user(semaphore, next(running) = 1);
```

3. **Semaphore transition cannot be specified inside „user”!**
   Reason: Semaphore is executed globally, only once per step;
   so it has to be specified globally.

   (a) Copy/paste transition function; adapt for each user process.
   (b) Transition has to take into account „active” flag of each process.

4. Fairness condition: Each process has to run infinitely often.
   More fairness conditions may be needed to avoid livelock, fulfill properties.

Assignment 1: Three exercises

1. Vending machine: fix, add counter.
2. Bridge crossing: implement, find optimum.
3. Semaphore: find solution without NuSMV processes.

- All parts: Add brief comments (starting with `--`) for variables, modules, transitions, properties.

$\rightarrow$ What does variable represent, why is starting state as is.
$\rightarrow$ What does transition rule/property mean, why was it changed.

- Part 2: Add explanation of optimum (error trace) as comment at end. Multi-line comments start with `/--`, end with `--/`.

- Part 3: Add explanation of error trace when fairness is disabled.

Lab exercise and peer review

Lab: opportunity to ask questions and work on exercise.

1. Introductory exercise:
   Submit on Canvas before Friday! (Imperfect solution ok)

2. Peer review on Canvas.

3. Large (graded) exercise in labs 2 and 3 of each cycle.

Advanced NuSMV features

- How to denote changes in **`x`**: **`next(x)`**
- Non-deterministic assignment.
- Evaluation order.
- Fairness.

`next(x)`: the future state of a variable

```
MODULE main
VAR
  x: 0..10;
  y: 0..10;
ASSIGN
  init(x) := 0;
  next(x) := x < 10 ? x + 1 : x;
  next(y) := x;
-- LTLSPEC G (x + 2 = next(next(x)));
-- canNOT nest "next"! Use auxiliary variable
LTLSPEC G (x + 2 = next(y)); -- ok
```

Non-deterministic assignment

Revisit the ferryman and semaphore examples:

```
next (ferryman) := { FALSE, TRUE }; -- ferryman

state = critical : { critical, exiting };
  -- inside case expression in semaphore
```

Meaning: Both values are possible for the next state.
NuSMV will explore all outcomes in the set of assigned values.

Uses:

- Leave inputs unspecified.
- Optional state transition.

Evaluation order

```
next (goat) := case
  ferryman = goat & next (carry) = g: next (ferryman);
  TRUE: goat;
esac;
```

- **Case statements** are evaluated top to bottom; first match triggers.
- **Variable updates** are (conceptually) applied simultaneously (synchronously) to all variables, changing x to $\mathrm{next}(x)$ atomically.
  If there are dependencies between updates, they are applied in the right order.
  Cyclic dependencies between variables updates result in compile error.

Fairness

- *Fairness constraints* have to hold infinitely often during a valid trace:

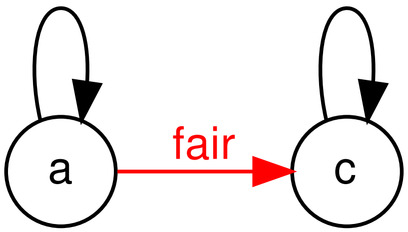
```
FAIRNESS running;
```

- Straightforward for booleans, but not for temporal formulas.

Weak and strong fairness

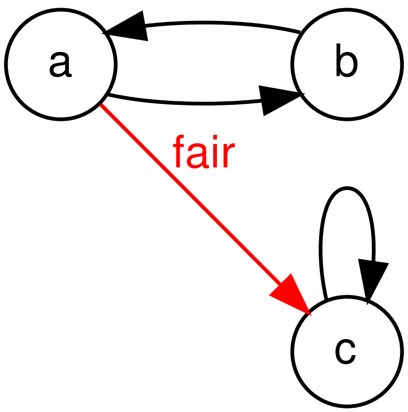
Weak (Büchi) fairness: $\Box\Diamond(\mathit{fairness} \rightarrow \mathit{property})$

- A set of transitions cannot be enabled **forever** without being taken.
- **`JUSTICE`** or **`FAIRNESS`** in NuSMV (same functionality).



Strong (Streett) fairness: $(\Box\Diamond\mathit{fairness}) \rightarrow (\Box\Diamond\mathit{property})$

- Set of transitions cannot be enabled **infinitely often** without being taken.



- **`COMPASSION(p, q)`** in NuSMV.

„Fair states set is empty"

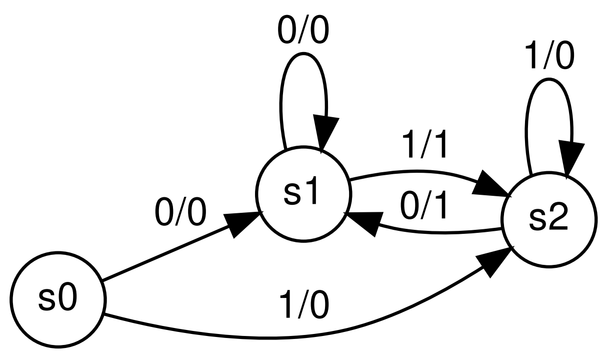
```
********* WARNING *********
Fair states set of the finite state machine is empty.
This might make results of model checking not trustable.
********* END WARNING *********
```

Explanation: You have a fairness expression that is never true.

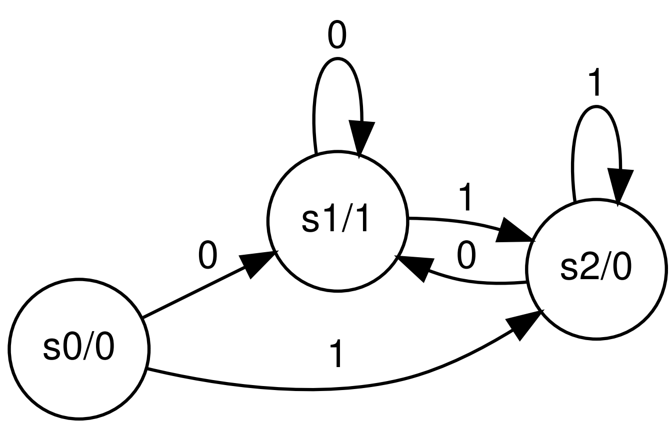
- The desired fair state(s) are never reached!
- Usually the result of a deadlock in your FSM.
- You cannot trust the result until you fix this warning.

Different types of state machines

Mealy machine: Output depends on current states and input.



Moore machine: Output depends only on current state.



Mealy vs. Moore machines

Mealy	Moore
Output on n^2 edges.	Output on n states.
Fewer states needed.	More states needed.
More responsive.	Output delayed until clock.
Risk of asynchronous feedback when connecting machines.	Output on clock. Synchronous product of machines.
Output expression contains input variable.	No use of **`next(...)`** in output expression.

Design choice

- Vending machine example: M   machine.
  Clear semantics about when item is dropped.
  Closer to real world, as accepting payment incurs delay.

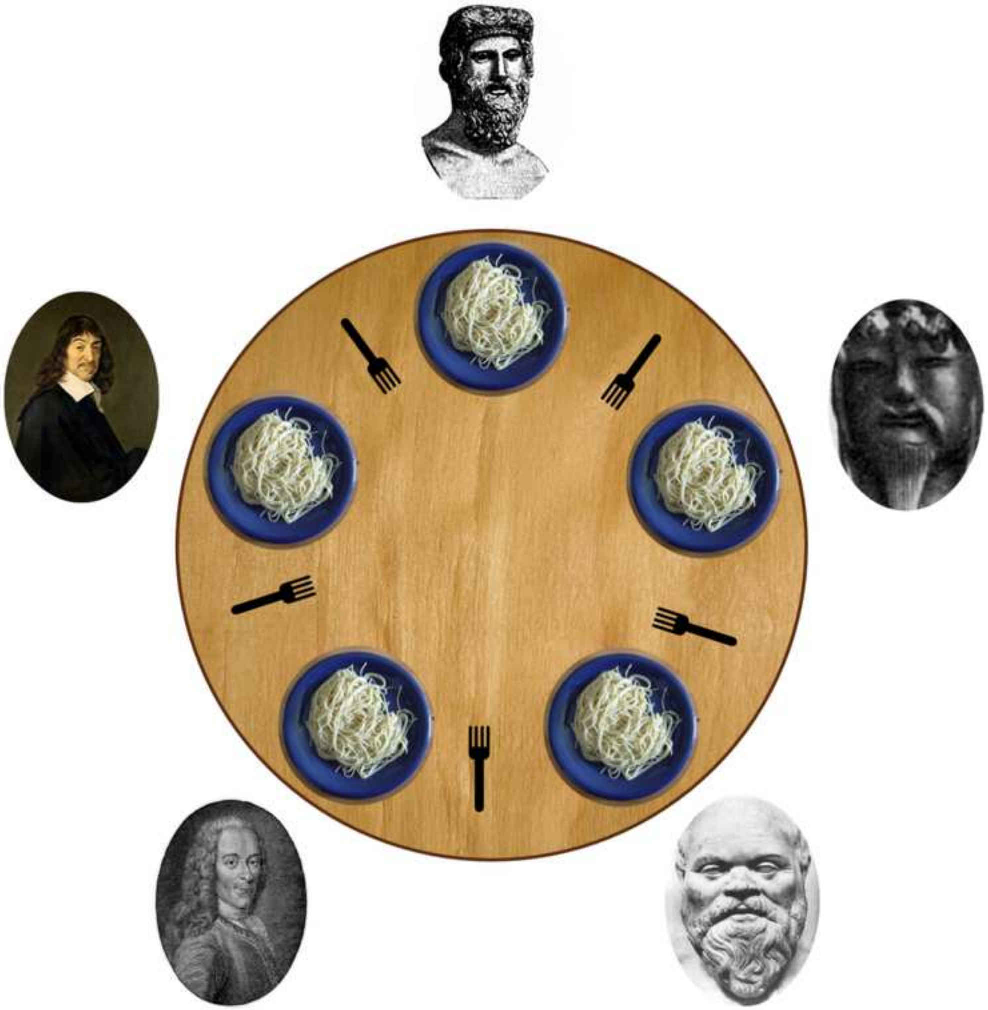
- Bridge crossing example: M   machine.
  Simpler implementation of rule,
  can read error trace directly without referring to past state.

- Semaphore: M   machine.
  Model behavior of process selector.

Advanced modeling: parametrized modules

- Instantiate multiple modules with different settings.
- Same transition rules for each instance.
- Avoid „copy pasta”.

Example: Dining Philosophers



- Five philosphers.
  - → Think eat.
  - → Need forks to eat.
- Five plates.
- Five forks.
  - → Exclusive access.
  - → Atomic access.

Can we get mutual exclusion without starvation?

Main module

```
MODULE main
VAR
  fork: array 1..5 of 0..5; -- 0: unused; > 0: held by X
  phil1: process philosopher(1, fork[1], fork[2]);
  phil2: process philosopher(2, fork[2], fork[3]);
  phil3: process philosopher(3, fork[3], fork[4]);
  phil4: process philosopher(4, fork[4], fork[5]);
  phil5: process philosopher(5, fork[5], fork[1]);
DEFINE
  available := 0;
ASSIGN
  init(fork[1]) := available;
  init(fork[2]) := available;
  init(fork[3]) := available;
  init(fork[4]) := available;
  init(fork[5]) := available;
```

Philosopher module

```
MODULE philosopher(id, leftFork, rightFork)
DEFINE
  owned := id;
  available := 0;
VAR
  state: {think, eat, done};
ASSIGN
  init(state) := think;
  next(state) := case
    (state = think) & (leftFork = owned) &
      (rightFork = owned): eat;
    (state = eat): {eat, done};
    (state = done): think;
    TRUE: think; -- (state = think) but forks not both taken
  esac;
```

Avoid copying module definition by using parameters.

Fork transitions, properties

```
  next(leftFork) := case
    (state = think) & (leftFork = available):
      {available, owned};
    state = done: available;
    TRUE: leftFork;
  esac;
  next(rightFork) := case
    (state = think) & (rightFork = available):
      {available, owned};
    state = done: available;
    TRUE: rightFork;
  esac;
SPEC   AG ((state = eat) ->
           ((leftFork = owned) & (rightFork = owned)))
SPEC   AG EF (state = eat);
```

Which properties hold?

Summary

Model checking

Enumerate all reachable states.
Check reachable states against temporal properties.

NuSMV

Uses domain-specific modeling language; simple data types.

- Module: unit with variables (state), transitions.
- Transitions: as formula or case block.
- Properties: LTL or CTL.