NuSMV

Cyrille Artho
KTH Royal Institute of Technology, Stockholm, Sweden
Computer Science and Communication
Theoretical Computer Science
`artho@kth.se`

2017-03-29

Assignments on Canvas

1. Everyone has to be in a group **in group set Cycles_1+2+5.**
2. Ask before joining a group.
3. Submit assignments as a group (once per group);
   write your names on assignment.

- If you joined your group too late for the written assignment, use alternative assignment (see Canvas announcement).
- For later learning cycles, we will use new groups in group set Cycles_3+4 (to be created).

Outline

NuSMV

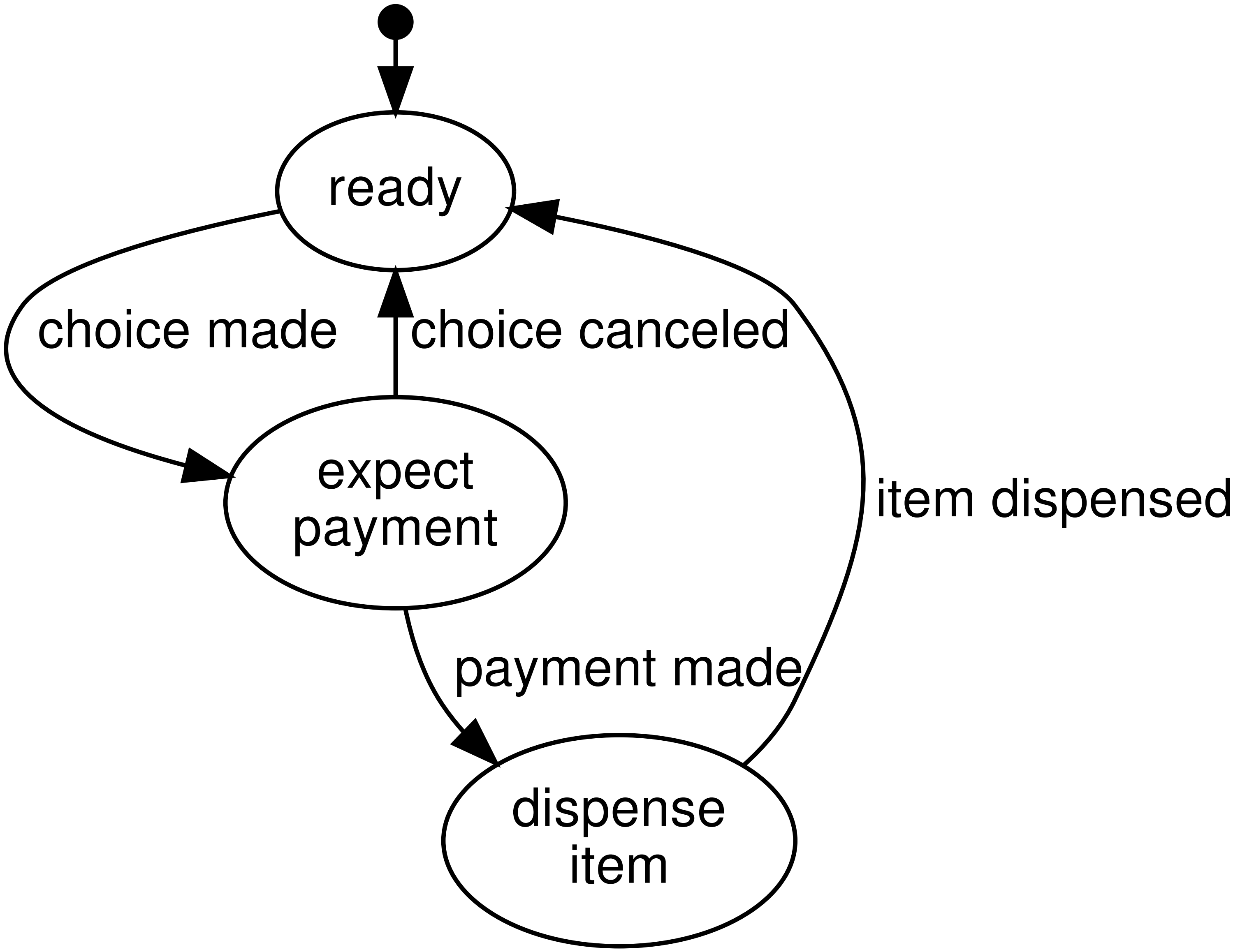
1. Vending machine.
2. Planning problems.
3. Processes.

NuSMV

- Re-implementation NuSMV (open source) at IRST Trento, Italy.
- NuSMV is still being maintained and developed.
- Current version is 2.6.0 (used in this course).

Assignment due tomorrow: Install NuSMV!

Example: Vending machine



Details not modeled

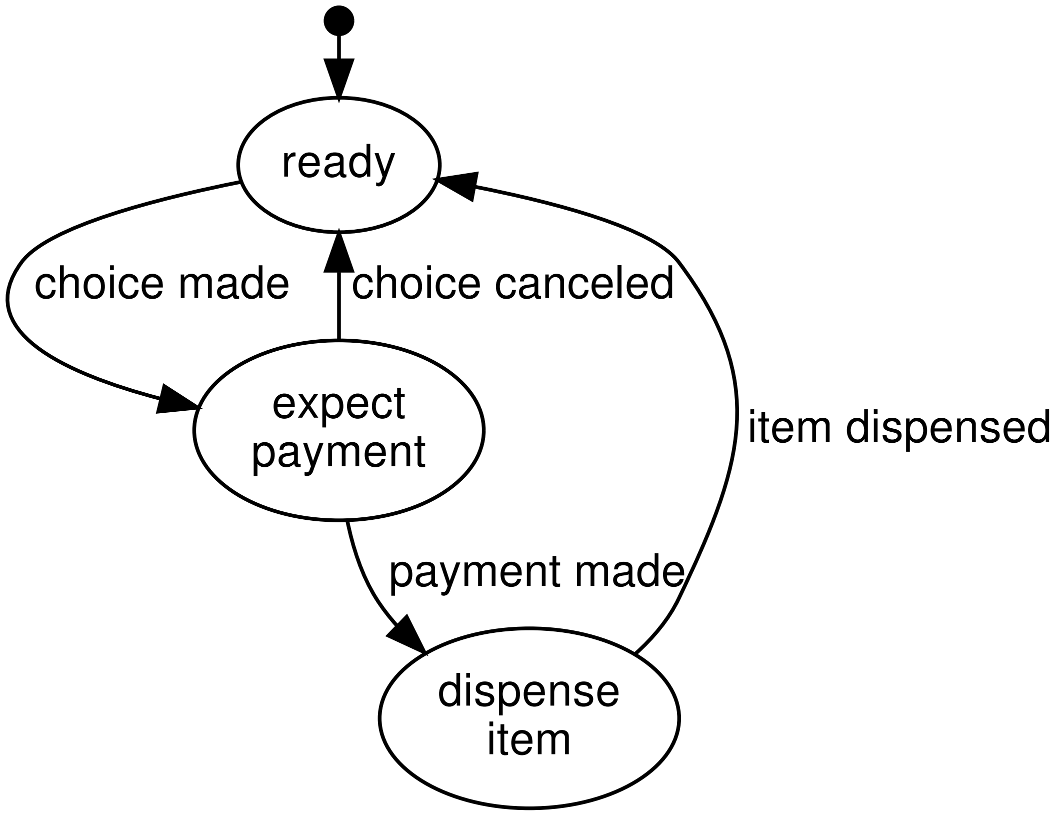
- How choice is made (how many choices) and canceled.

  $\rightarrow$ Button, timeout, both?

- How payment is handled/accepted; price of goods.

- Item dispenser mechanism, time taken to dispense item.

Example: Vending machine—2



- Three states: ready, expect_payment, dispense_item.
- Two user inputs (non-deterministic):

1. Choice (of item); can be cancelled.
2. Payment (requires choice to be made first).

Vending machine in NuSMV

```
MODULE main
VAR
  choice, payment:  boolean;
  state:   { ready, expect_payment, dispense_item };
ASSIGN
  init (state) := ready;
  next (state) := case
    state = ready          & choice:  expect_payment;
    state = expect_payment & payment: dispense_item;
    state = expect_payment & !choice: ready;
    state = dispense_item:            ready;
  esac;
```

NuSMV syntax

- Module, variables, assignments.
- case (follows order of declaration); can have multiple outcomes.

Error message

```
file vending.smv: line 3: at token ",": syntax error
file vending.smv: line 3: Parser error
NuSMV terminated by a signal
```

- Cannot declare more than one variable per line!
- Try again...

Another error message

```
file vending.smv: line 14: case conditions are not exhaustive
```

- Recall the **`case`** block:

```
state = ready            & choice:  expect_payment;
state = expect_payment & payment: dispense_item;
state = expect_payment & !choice: ready;
state = dispense_item:            ready;
```

- State should remain the same by default: specify!

Complete `case` block

```
ASSIGN
  init (state) := ready;
  next (state) := case
    state = ready             & choice:  expect_payment;
    state = expect_payment & payment: dispense_item;
    state = expect_payment & !choice: ready;
    state = dispense_item:              ready;
    TRUE:                               state;
  esac;
```

- Transitions from ready and expect_payment depend on user choice.
- Cancellation is modeled as **`choice`** reverting to **`false`**.
- Transition from dispense_item back to ready is automatic.

Run NuSMV

- Nothing happens!
- We need properties. . .

```
LTLSPEC
  G(choice -> F state = dispense_item);
```

- „Every time I choose something, I eventually get it”.

Nice try!

```
-- specification
   G (choice ->  F state = dispense_item)  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -- Loop starts here
  -> State: 1.1 <-
    choice = TRUE
    payment = FALSE
    state = ready
  -> State: 1.2 <-
    choice = FALSE
    state = expect_payment
  -> State: 1.3 <-
    choice = TRUE
    state = ready
```

OK, I'll pay…

```
LTLSPEC
  G(payment -> F state = dispense_item);
-- specification
   G (payment ->  F state = dispense_item)  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -- Loop starts here
  -> State: 1.1 <-
    choice = FALSE
    payment = TRUE
    state = ready
  -> State: 1.2 <-
```

- State 1.2: no progress.
- Payment is accepted even when no choice has been made!

Accept payment only when choice is made

```
MODULE main
VAR
  choice:   boolean;
  payment: boolean;
  acc_payment: boolean;
  state:   { ready, expect_payment, dispense_item };
ASSIGN
  init (state) := ready;
  next (state) := case
    state = ready            & choice:   expect_payment;
    state = expect_payment & acc_payment: dispense_item;
    state = expect_payment & !choice: ready;
    state = dispense_item:              ready;
    TRUE:                                state;
  esac;
  init (acc_payment) := FALSE;
  next (acc_payment) := (state = expect_payment & payment);
```

Another problem?!

`G (acc_payment -> F state = dispense_item)` is false

```
-> State: 1.1 <-
   choice = FALSE
   payment = FALSE
   acc_payment = FALSE
   state = ready
-> State: 1.2 <-
   choice = TRUE
-> State: 1.3 <-
   choice = FALSE
   payment = TRUE
   state = expect_payment
-> State: 1.4 <-
   payment = FALSE
   acc_payment = TRUE
   state = ready
-- Loop starts here
-> State: 1.5 <-
   acc_payment = FALSE
-> State: 1.6 <-
```

State 1.3: Choice is made, next state = accept payment

State 1.4: accepting payment, but choice is canceled just now!

- Need a way to prevent this transition back to *ready*.
- Use stricter `case` condition!
- Lab exercise 1.

Extension of the vending machine

- Limited capacity of n items.
- Payment should not be accepted when no items available.
- Counting down items:

```
next(n_items) := case
  ...: n_items - 1;
  TRUE: n_items;
esac;
```

- Fails!

```
file vending3.smv: line 16:
  cannot assign value -1 to variable n_items
```

Counting without over- or underflow

- NuSMV recognized possible over- or underflow at compile time.

```
next(n_items) := case
    ... & n_items > 0: n_items - 1;
    TRUE: n_items;
esac;
```

- Underflow needs to be prevented in code.

Additional properties

1. Number of items should always be ≥ 0.
2. Payment should only be accepted if number of items > 0.
3. If an item is dispensed, the counter of items is always reduced by 1.

First lab exercise: Summary

1. Use NuSMV on **`vending1.smv`**.
   - (a) Study error trace.
   - (b) Refine transition (**`case`** condition).
2. Counting remaining items.
   - (a) Add a counter **`n_items`** (see above).
   - (b) Write LTL or CTL properties for the three properties above.
   - (c) Ensure your model fulfills all properties.

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]*

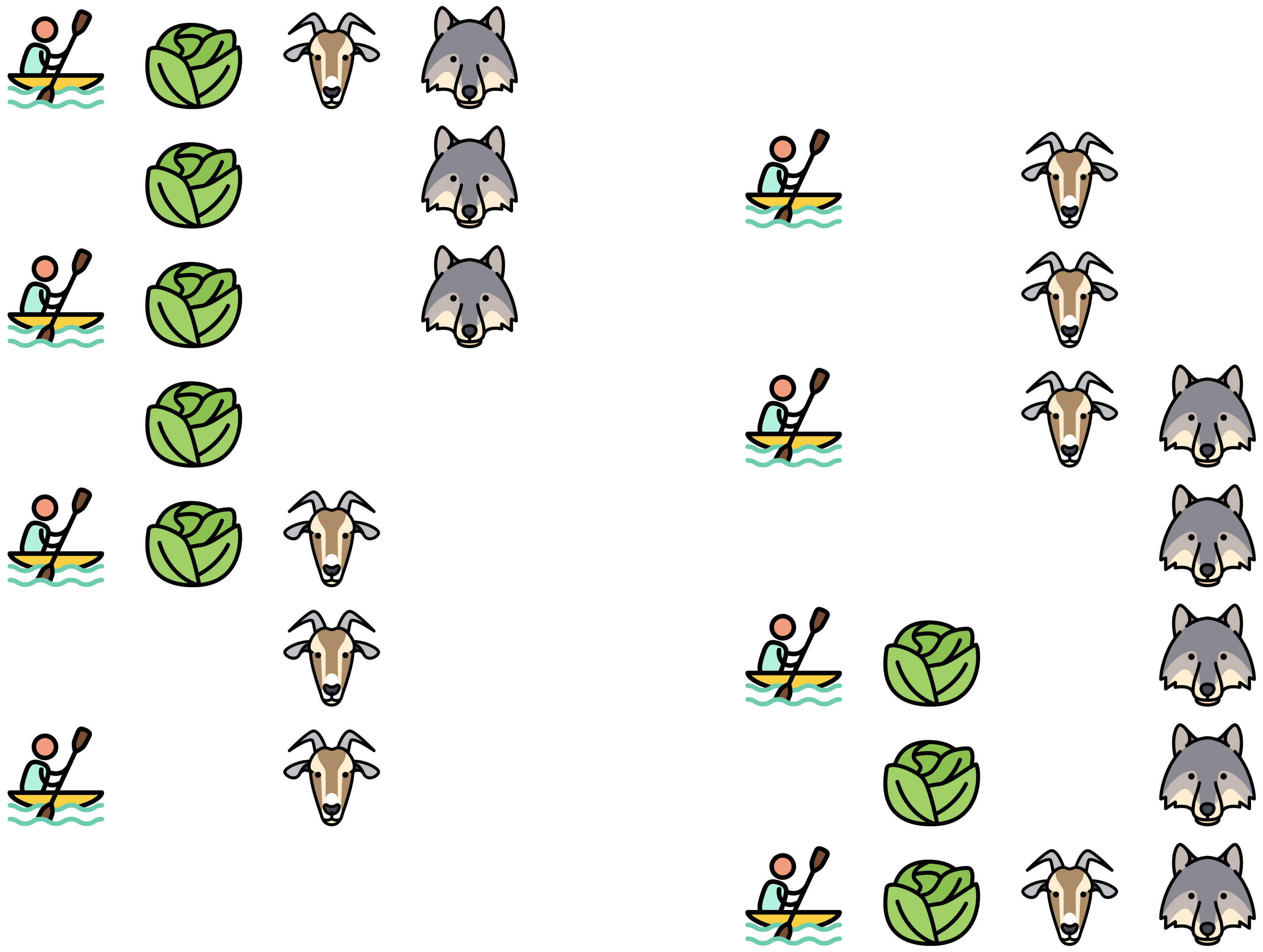
U (cabbage & goat & wolf & ferryman) *[solution]*

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 (same lab session as exercise 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: 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.

Summary

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.

Assignments

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. Use all four hours to work on problem, submit on Canvas, peer review (on Canvas) later.

2. Work in exercises for 2–3 hours, ask neighbor group to peer review, finish later.

   - Take notes of your peer review so you can submit a short summary.
   - In this case, let us know about which group you peer reviewed with.