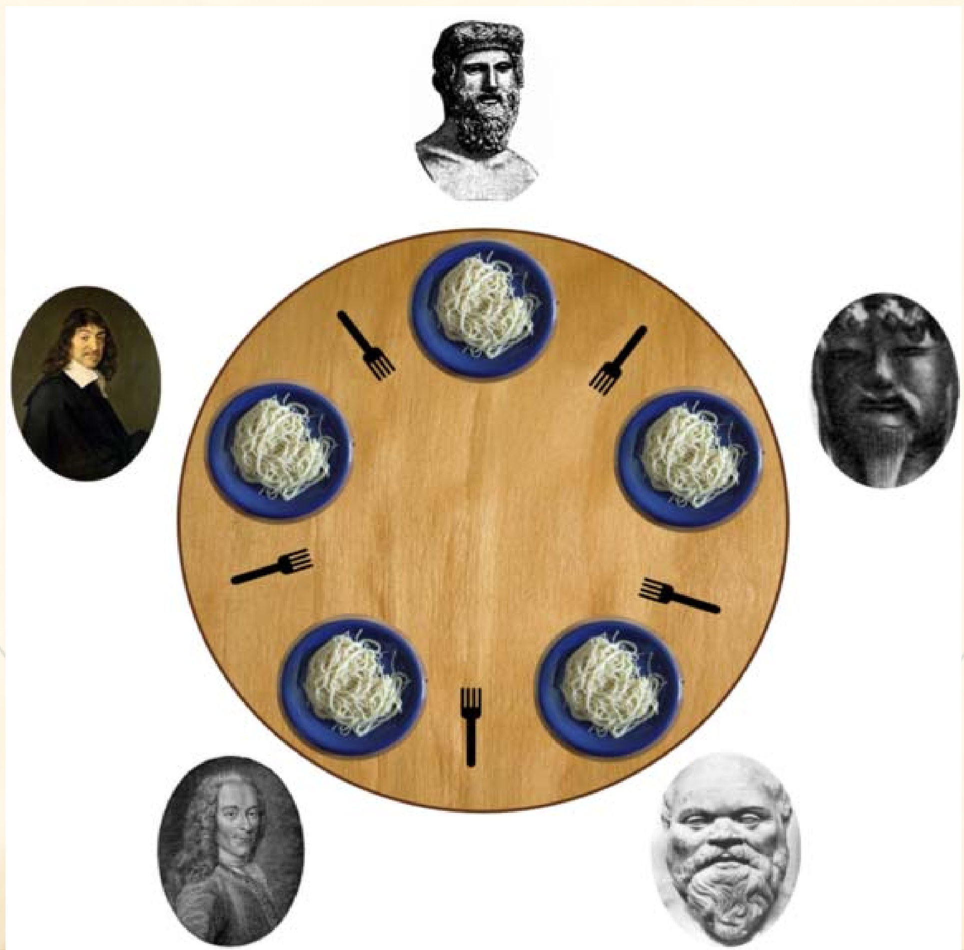
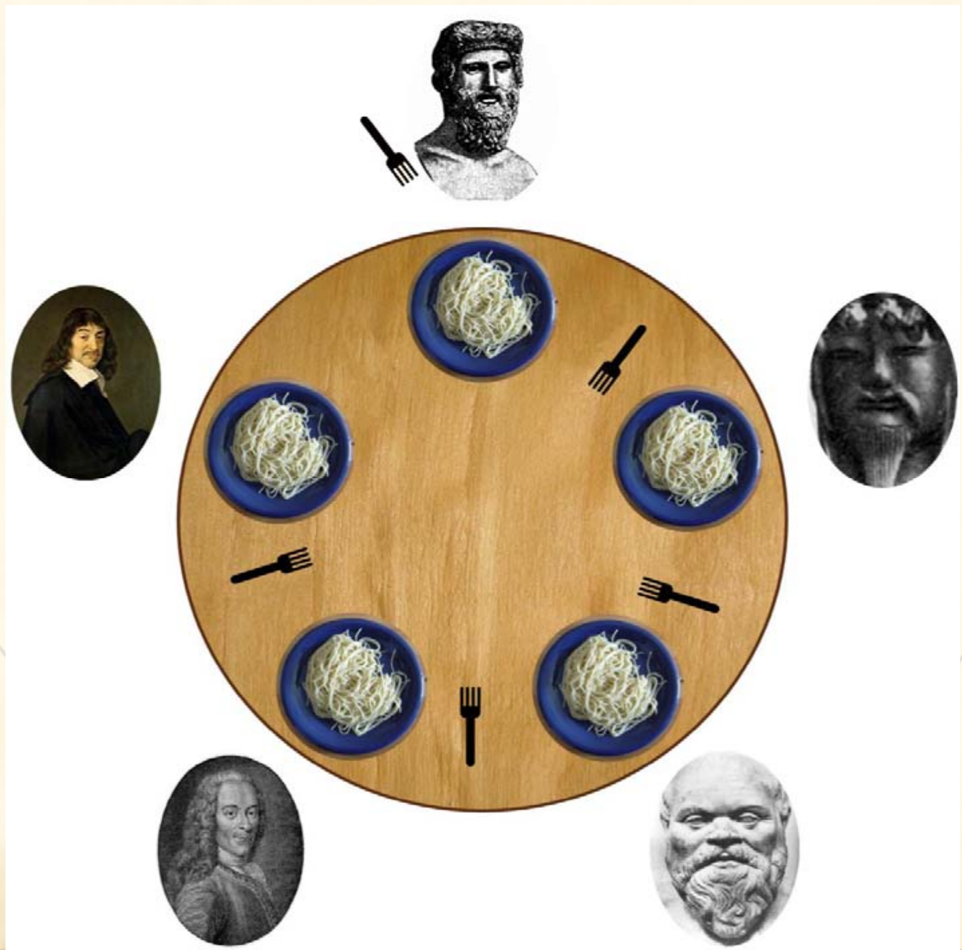
The Dining Philosophers

Specification and verification using CSP and PAT

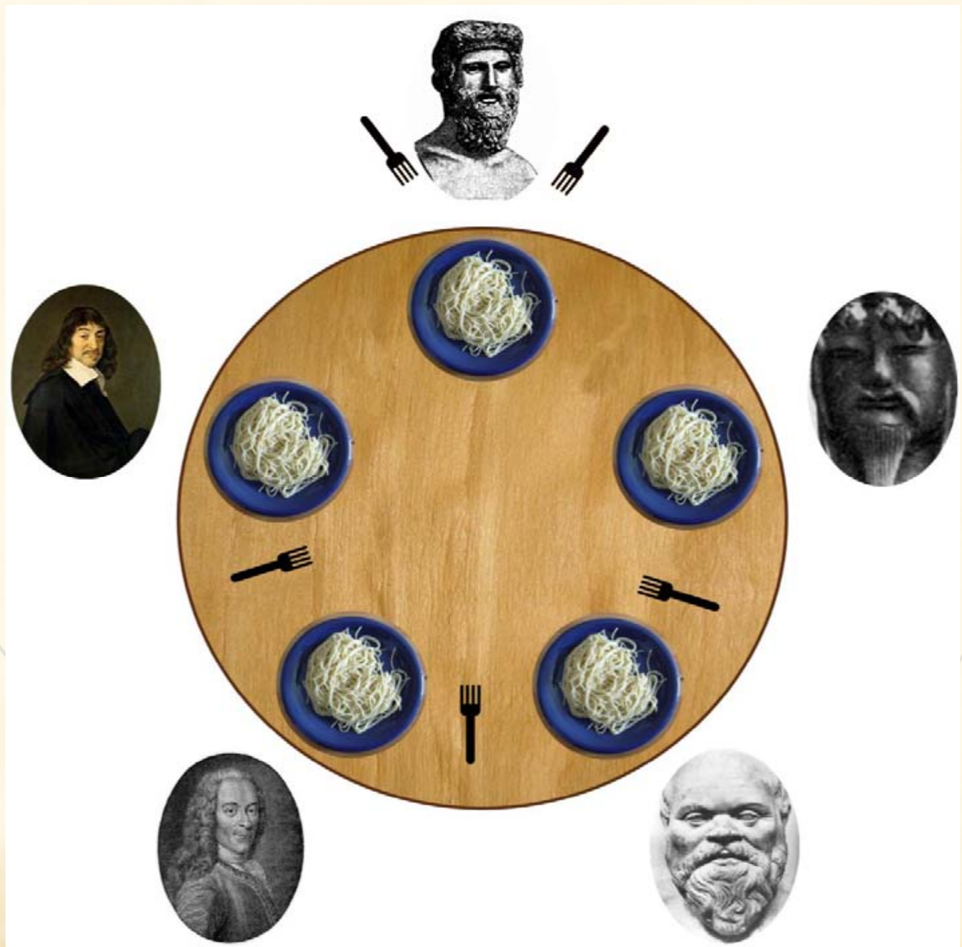
Sun Jun
sunj@comp.nus.edu.sg
http://www.comp.nus.edu.sg/~sunj



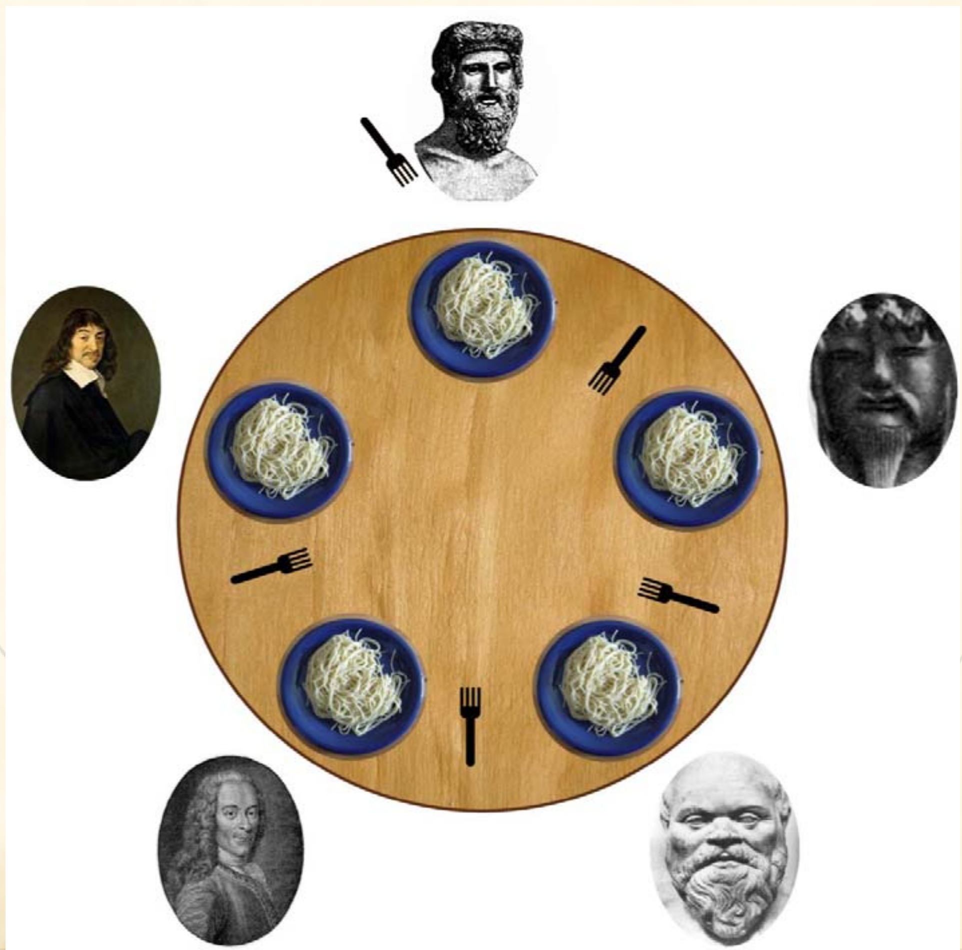
N=5



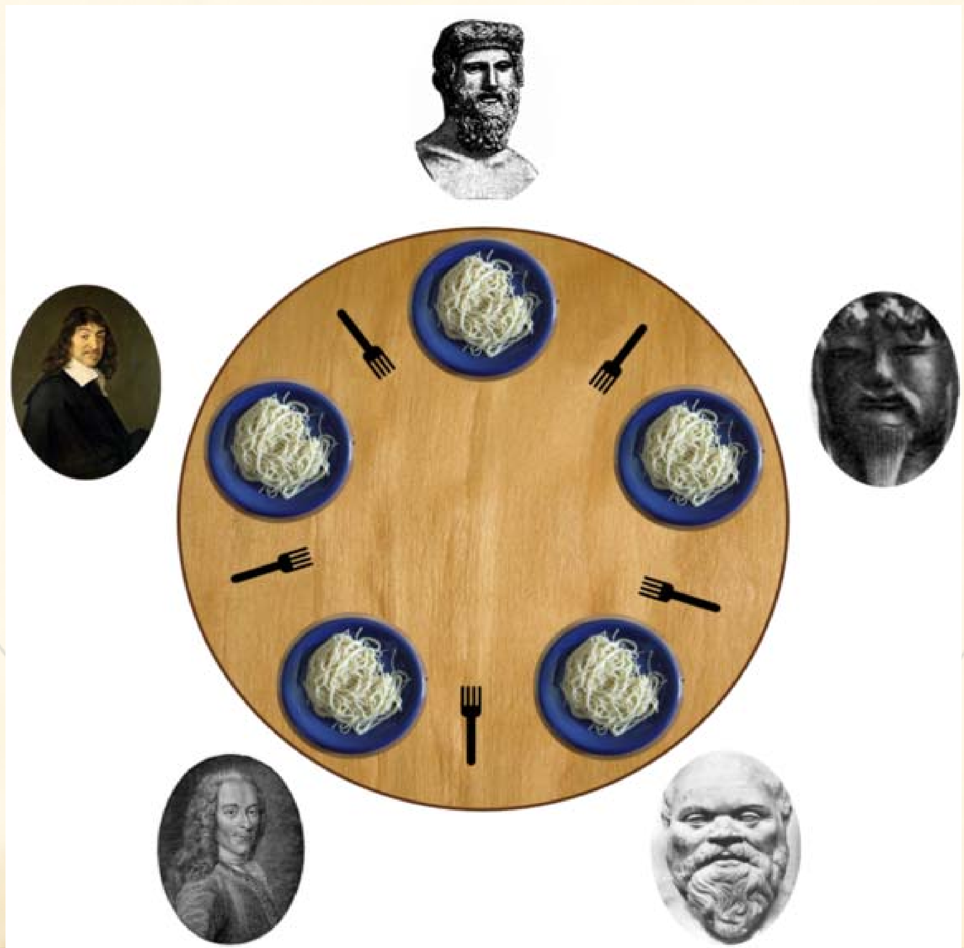
N=5



N=5



N=5



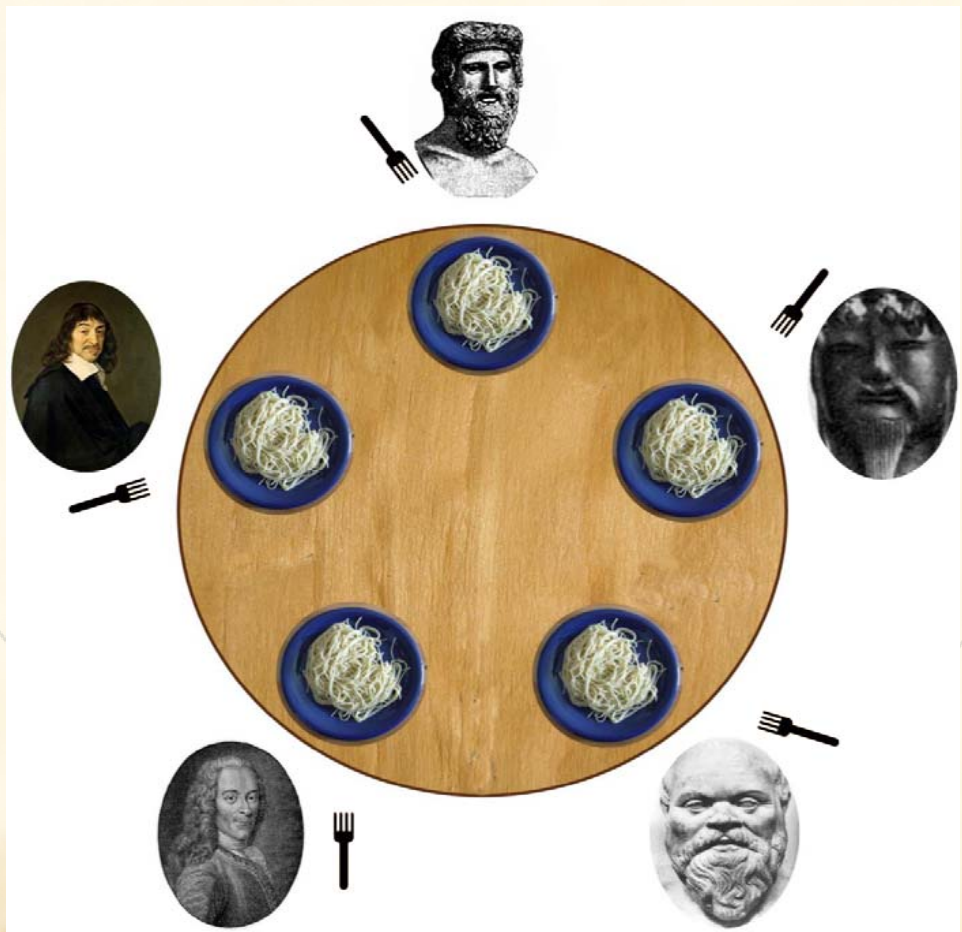
N=5

Requirements

- A philosopher must hold two forks to eat.
- A philosopher must take the fork on his right and the one on his left.
- A philosopher only puts down the forks after eating.

Questions

- Is it possible that all philosophers may starve to death?
- Is it possible that a philosopher may starve to death?
- How to prevent a philosopher from starving?

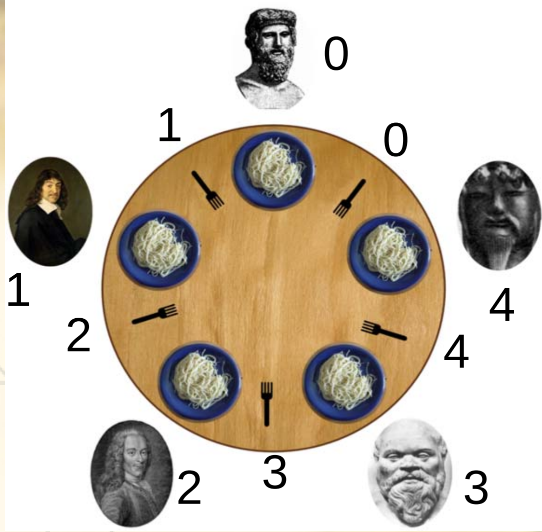


Starvation-Free?

- Introduce a waiter at the table?
- More forks?
- Ask one philosopher to pick up the forks in a different order?

Formal Specification and Verification

- Specification helps to capture the essence of the problem.
- Formal specification (e.g., CSP) has well defined semantics.
- Automated verification support is available.



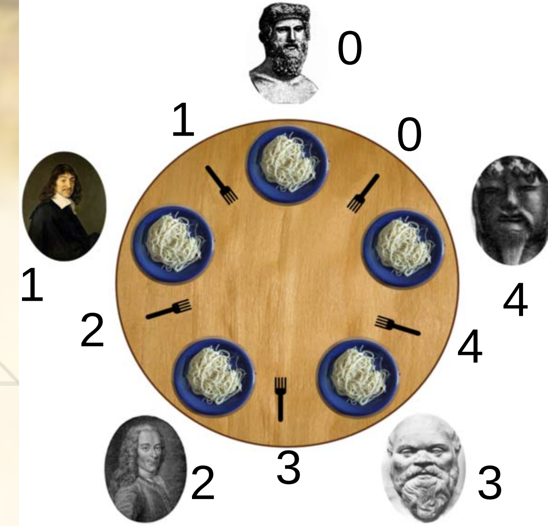
Modeling using CSP

Phil(i) = get.i.(i+1)%5 → get.i.i → eat.i → put.i.(i+1)%5 → put.i.i → Phil(i)

*get.i.j is the event of i-th philosopher picking up the j-th fork.

e.g., Phil(0) = get.0.1 → get.0.0 → eat.0 → put.0.1 → put.0.0 → Phil(0)

traces(Phil(0)) = ?



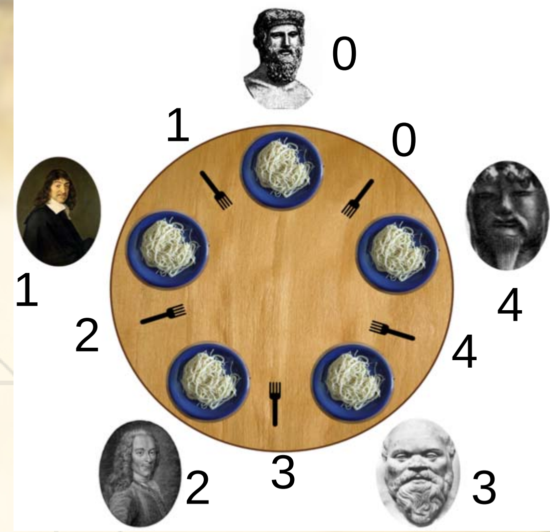
Modeling using CSP

Fork(i) = get.(i-1)%5.i → put. (i-1)%5.i →Fork(i) $\Box$ get.i.i → put.i.i → Fork(i)

*get.i.j is the event of i-th philosopher picking up the j-th fork.
*$\Box$ is external choice.

e.g., Fork(0) = get.4.0 → put.4.0 → Fork(0) $\Box$ get.0.0 → put.0.0 → Fork(0)

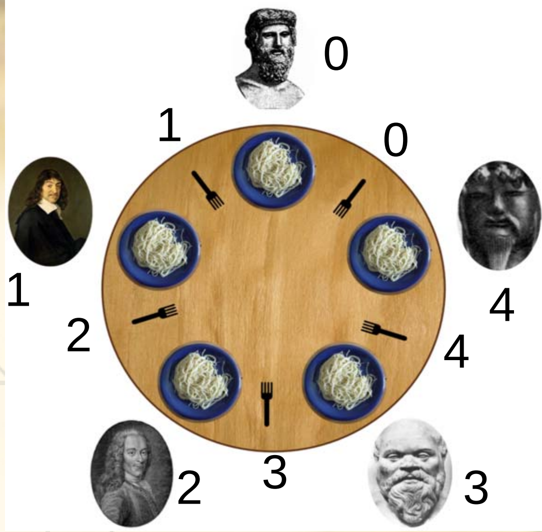
traces(Fork(0)) = ?



Operational Semantics

- CSP has well-defined operational semantics to say how the system executes.
  - Assume that initially the process is
    - Phil(0) || Fork(0)
  - After engaging in event get.0.1, the process becomes
    - (get.0.0 → eat.0 → put.0.1 → put.0.0 → Phil(0)) || Fork(0)
  - After engaging in event get.0.0, the process becomes
    - eat.0 → put.0.1 → put.0.0 → Phil(0) || put.0.0 → Fork(0)

Reminder: Phil(0) = get.0.1 → get.0.0 → eat.0 → put.0.1 → put.0.0 → Phil(0)
Fork(0) = get.4.0 → put.4.0 →Fork(0) □ get.0.0 → put.0.0 → Fork(0)



Modeling using CSP

College = Fork(0) || Phil(0) || Fork(1) || Phil(1) ||
Fork(2) || Phil(2) || Fork(3) || Phil(3) ||
Fork(4) || Phil(4)

Modeled, What Now?

- Model checking exhaustively searches through all possible behaviors of the system to verify whether the system satisfies certain properties.
- What are the questions we can ask about a given model?

Formal Verification

- Reachability Analysis
- Temporal Logic Properties
- Refinement Checking

Reachability Analysis

- whether certain state is reachable
  - Deadlock-freeness – whether the system may reach a state at which no further move is available.
  - Invariants – whether the system may reach a state at which a condition is satisfied (or violated).
    - e.g., a pointer will never be null, the index of an array is never negative, etc.

Reachability Analysis Algorithm

```
working := an empty stack;
working.push(initial state);

while (working is not empty) {
        let s := working.pop();
        if (s satisfies certain condition) {
                produce a counterexample and return false;
        }
        else {
                foreach successor s’ of s {
                        if (s’ is new) {
                                working.push(s’)
                        }
                }
        }
}

return true;
```

The bridge crossing puzzle

The bridge crossing puzzle

knight = 0; Lady = 0; King = 0; Queen = 0;
time = 0;

South () = [knight == 0 && Lady == 0]**go_knight_lady**{knight = 1; Lady= 1; time = time+2;} → North ()
☐ [knight == 0 && King == 0]**go_knight_king**{knight = 1; King= 1; time = time+5;} → North ()
☐ [knight == 0 && Queen == 0]**go_knight_queen**{knight = 1; Queen= 1; time = time+10;} → North ()
☐ [Lady == 0 && King == 0]**go_lady_king**{Lady = 1; King= 1; time = time+5;} → North ()
☐ [Lady == 0 && Queen == 0]**go_lady_queen**{Lady = 1; Queen= 1; time = time+10;} → North ()
☐ [King == 0 && Queen == 0]**go_king_queen**{King = 1; Queen= 1; time = time+10;} → North ()
☐ [knight == 0]**go_knight**{knight = 1; time = time+1;} → North ()
☐ [Lady == 0]**go_lady**{Lady = 1; time = time+2;} → North ()
☐ [King == 0]**go_king**{King = 1; time = time+5;} → North ()
☐ [Queen == 0]**go_queen**{Queen = 1; time = time+10;} → North ()

Formal Verification

- ✓ Reachability Analysis
- Temporal Logic Properties
- Refinement Checking

Temporal Logic Properties

- 0-th philosopher will never starve to death.
  - []<> eat.0 where [] reads as `always' and <> as `eventually'.
- No philosophers starve.
  - []<>eat.0 and []<>eat.1 and …

Temporal Logic Verification

- Given []<> eat.0, search for a loop which contains no transition labeled with eat.0.
- Two sets of algorithms can be used to find such a loop.
  - Nested depth-first-search.
  - Tarjan’s algorithm for finding maximum strongly connected components.

Nested Depth-First-Search

- Assume we are search for a loop containing a state which satsifies some condition.
- Use a depth-first-search to find such a state.
- Use a second depth-first-search to check if the state is reachable from itself.

Tarjan's SCC Algorithm

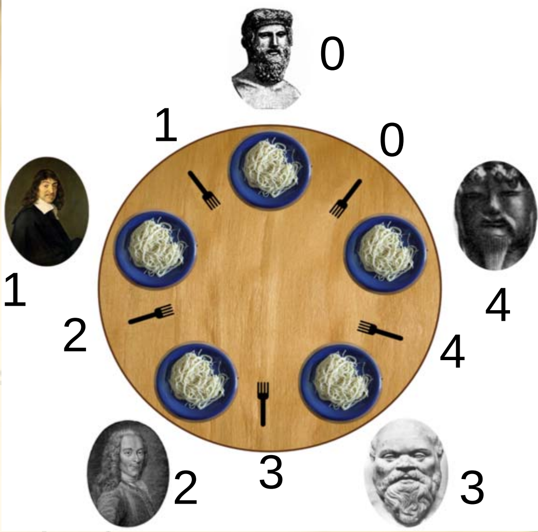
- The system can be viewed as a graph.
- A loop must be part of a Strongly Connected Component.
- Find the maximum SCC which contains such a state.

Temporal Logic Verification

- Given a (LTL) temporal logic formula, it's negation is translated to a Büchi automaton.
- The product of the Büchi automaton and the process is computed.
- A counterexample is a loop which contains at least one accepting state.

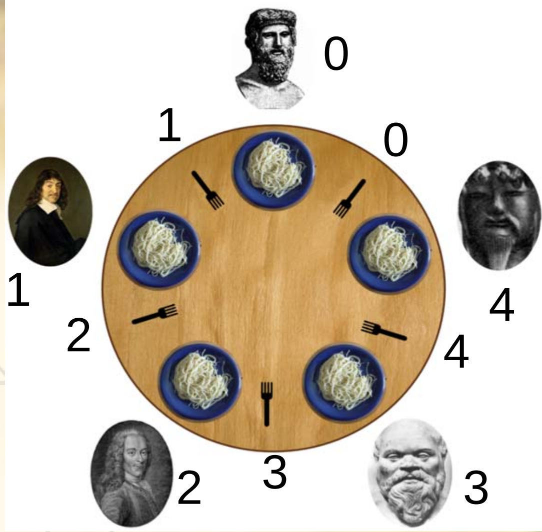
Lacking of Fairness

- []<>eat.0 – one philosopher eats greedily and leaves no chance to the neighbors.
- []<> a car will reach its destiny – A pedestrian may go back and forth on a zebra crossing forever.
- []<> my car will change to the right lane and right-turn – there might be infinite stream of cars on the right lane.
- ……



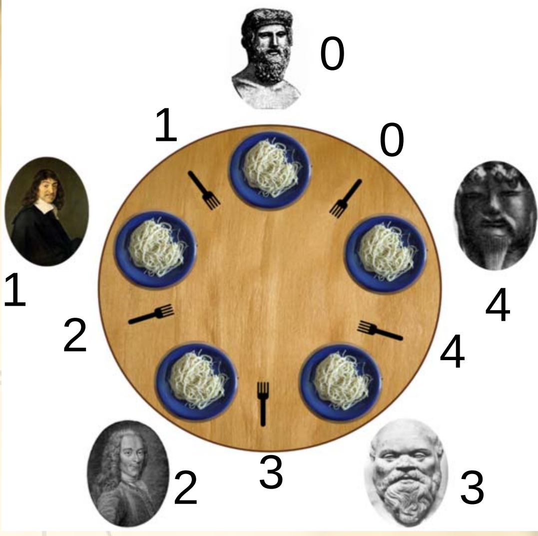
Weak Fairness

- If an event is always enabled, it must be eventually engaged.
  - e.g., during the loop <get.0.1, get.0.0,eat.0,put.0.1,put.0.0>, the event get.3.4 is always enabled.



Strong Fairness

- If an event is repeatedly enabled, it must be eventually engaged.
  - e.g., during the loop <get.0.1, get.0.0,eat.0,put.0.1,put.0.0>, the event get.4.0 is repeatedly enabled.



Modeling Fairness

- Each philosopher must eventually pick up a fork.
  - Phil(i) = wl(get.i.(i+1)%5) → get.i.i → eat.i → put.i.(i+1)%5 → put.i.i → Phil(i)

- A fork must be eventually picked up by both philosophers.
  - Fork(i) = sl(get.(i-1)%5.i) → put. (i-1)%5.i →Fork(i) □ sl(get.i.i) → put.i.i → Fork(i)

Verification under Fairness

- Assume that each philosopher will eventually get his turn to pick up a fork, is []<>eat.0?
  - Search for a loop such that during the loop each philosopher does pick up a fork and yet the loop contains no transition labeled with eat.0.

Formal Verification

- ✓ Reachability Analysis
- ✓ Temporal Logic Properties
- Refinement Checking

Refinement Checking

- Verifying properties by showing a refinement relationship between a process modeling the system and a process modeling the property.
  - Given process P and Q, whether Q can do whatever P can do.
  - Or whether Q can not do whatever P can not do.

Hiding

- Given a process P and a set of events X, the process P \ X hides occurrences of events from X.
  - e.g., College \{get.*.*,put.*.*}

Trace Refinement Checking

- Question: is it possible that the philosophers eat in the pre-defined order?
  - i.e., whether P = eat.0 $\rightarrow$ eat.1 $\rightarrow$ eat.2 $\rightarrow$ eat.3 $\rightarrow$ eat.4 $\rightarrow$ P trace-refines College \{get.*.*,put.*.*}?
- Question: is it that the philosophers only eat in the pre-defined order?
  - i.e., whether College \{get.*.*,put.*.*} trace-refines P?

Refinement Checking

- Failures Refinement Checking,
  - P failures-refines Q means that Q not only can do whatever P can do, but also cannot do whatever P cannot do.
  - e.g., P = eat.0 $\rightarrow$ P, Stop trace-refines P but not failures-refines P.
- Failures/Divergence Refinement Checking,
  - Assume there might be infinite loop (of invisible events).

Summary

- Formal modeling is the starting point for formal system analysis.
  - Formal modeling allows us to focus on essence of the problems.

- Model checking is an active research area for formal verification.
  - this year’s Turing award!