Modeling and Specification in OTS/CafeOBJ

CafeOBJ Team of JAIST

Topics

- What is QLOCK?
- Modeling and Description of QLOCK in OTS
- Formal specification of QLOCK in CafeOBJ
- Formal specification of mutual exclusion

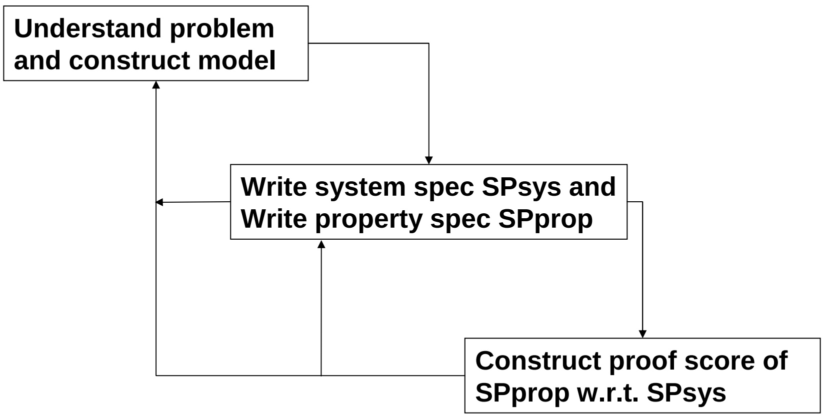
Modeling, Specifying, and Verifying (MSV) in CafeOBJ

1. By understanding a problem to be modeled/specified, determine <u>several sorts of objects</u> (entities, data, agents, states) and <u>operations</u> (functions, actions, events) over them for describing the problem

2. Define the meanings/functions of the operations by declaring <u>equations</u> over expressions/terms composed of the operations

3. Write <u>proof scores</u> for properties to be verified

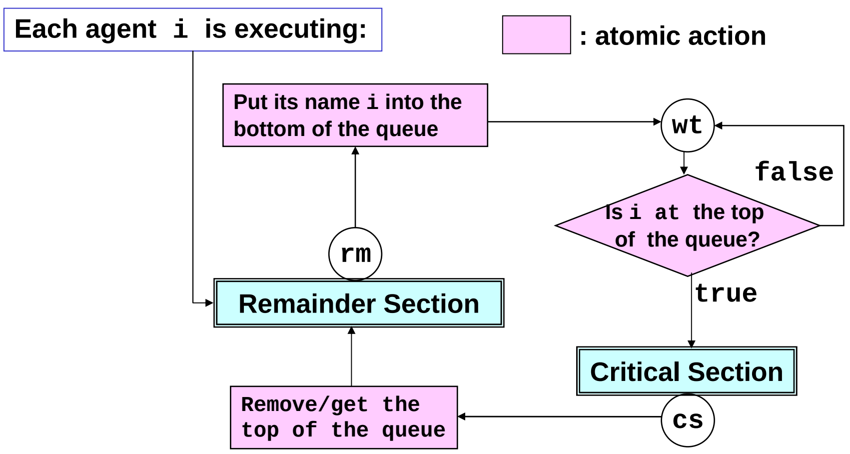
MSV with proof scores in CafeOBJ



An example: mutual exclusion protocol

Assume that many agents (or processes) are competing for a common equipment, but at any moment of time only one agent can use the equipment. That is, the agents are mutually excluded in using the equipment. A protocol (mechanism or algorithm) which can achieve the mutual exclusion is called “mutual exclusion protocol”.

QLOCK (locking with queue): a mutual exclusion protocol

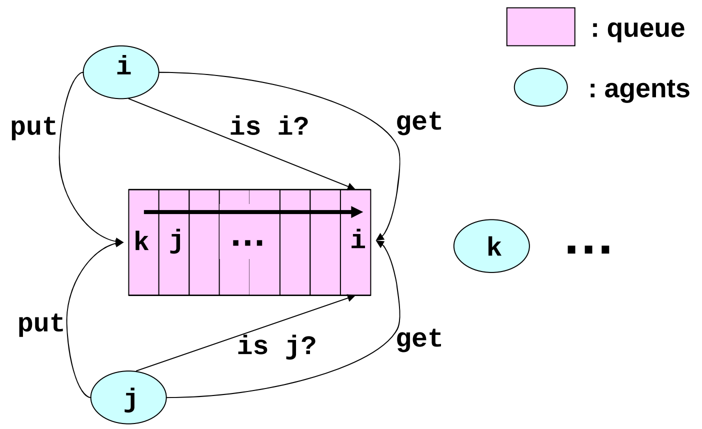


QLOCK: basic assumptions/characteristics

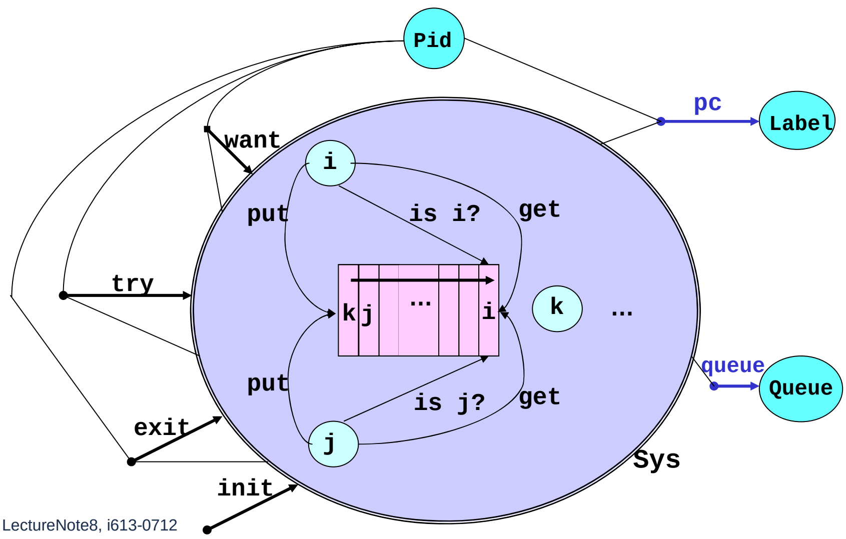
- There is only one queue and all agents/processes share the queue.
- Any basic action on the queue is inseparable (or atomic). That is, when any action is executed on the queue, no other action can be executed until the current action is finished.
- There may be unbounded number of agents.
- In the initial state, every agents are in the remainder section (or at the label `rm`), and the queue is empty.

The property to be shown is that at most one agent is in the critical section (or at the label `cs`) at any moment.

Global (or macro) view of QLOCK



Modeling QLOCK (via Signature Diagram) with OTS (Observational Transition System)



Signature for QLOCKwithOTS

- **`Sys` is the sort for representing the state space of the system.**
- **`Pid` is the sort for the set of agent/process names.**
- **`Label` is the sort for the set of labels; i.e. `{rm, wt, cs}`.**
- **`Queue` is the sort for the queues of `Pid`**
- **`pc` (program counter) is an observer returning a label where each agent resides.**
- **`queue` is an observer returning the current value of the waiting queue of `Pid`.**
- **`want` is an action for agent `i` of putting its name/id into the queue.**
- **`try` is an action for agent `i` of checking whether its name/id is at the top of the queue.**
- **`exit` is an action for agent `i` of removing/getting its name/id from the top of the queue.**

CafeOBJ signature for QLOCKwithOTS

```
-- state space of the system
*[Sys]*

-- visible sorts for observation
[Queue Pid Label]

-- observations
bop pc : Sys Pid -> Label
bop queue : Sys -> Queue

-- actions
bop want : Sys Pid -> Sys
bop try  : Sys Pid -> Sys
bop exit : Sys Pid -> Sys
```

Hiden sort declaration

visible sort declaration

Observation declaration

action declaration

Module LABEL specifying (via tight denotation/semantics) "labels"

qlock.mod

```
mod! LABEL {
  [Label]
  ops rm wt cs : -> Label
  pred (_=_) : Label Label {comm}
  var L : Label
  eq (L = L) = true .
  eq (rm = wt) = false .
  eq (rm = cs) = false .
  eq (wt = cs) = false .
}
```

Predicate (_ = _) defines identity relation among `rm`, `wt`, and `cs`.

Module PID specifying (via loose denotation) "agent/process names/identifiers"

qlock.mod

```
mod* PID {
  [Pid < PidErr]
  op none : -> PidErr
  pred (_=_) : PidErr PidErr {comm}
  var I : Pid .
  eq (I = I) = true .
  eq (none = I) = false .
  -- (none = none) is not defined intentionally
}
```

- **The constant none of the sort `PidErr` is intended to indicate the result of getting top of the empty queue.**
- **Any element in the sort `Pid` is defined not equal to none, that is, return `false` for predicate `(_ = _) .`**
- **Notice that `(none = none)` does not reduced to `true` or `false`.**

Module QUEUE specifying "queue" (1) -- an parameterized module

qlock.mod

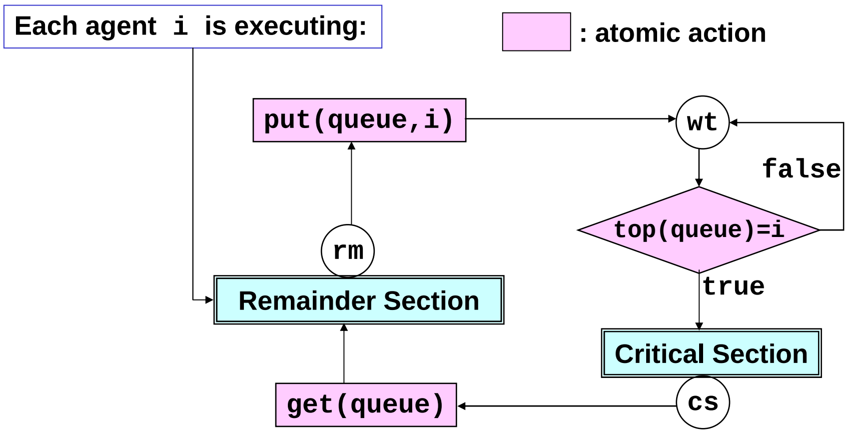
```
mod* TRIVerr {
  [Elt < EltErr]
  op none : -> EltErr
}
mod! QUEUE(D :: TRIVerr) {
  [Queue]
-- constructors
  op empty :              -> Queue {constr}
  op _,_    : Queue Elt.D -> Queue {constr l-assoc}
-- operators
  op put : Elt.D Queue -> Queue
  op get : Queue -> Queue
  op top : Queue -> EltErr.D
  op empty? : Queue -> Bool
```

Module QUEUE specifying "queue" (2) -- an parameterized module

qlock.mod

```
-- CafeOBJ variables
  var Q : Queue
  vars X Y : Elt.D
-- equations
  eq put(X,empty) = empty,X .
  eq put(X,(Q,Y)) = put(X,Q),Y .
  -- get(empty) is not defined intentionally
  eq get((Q,X)) = Q .
  eq top(empty) = (none):EltErr.D .
  eq top((Q,X)) = X .
  eq empty?(empty) = true .
  eq empty?((Q,X)) = false .
}
```


QLOCK using operators in the CafeOBJ module QUEUE



Module QLOCK specifying "QLOCK" (1-1)

qlock.mod

```
view TRIVerr2PID from TRIVerr to PID {
  sort Elt -> Pid,
  sort EltErr -> PidErr,
  op (none):EltErr -> (none):PidErr  }
```

```
mod* QLOCK {
  pr(LABEL)
  pr(QUEUE(D <= TRIVerr2PID))
  *[Sys]*
-- any initial state
  op init : -> Sys
-- observations
  bop pc    : Sys Pid -> Label
  bop queue : Sys -> Queue
-- actions
  bop want : Sys Pid -> Sys
  bop try  : Sys Pid -> Sys
  bop exit : Sys Pid -> Sys
-- for any initial state
  eq pc(init,I:Pid)  = rm .
  eq queue(init) = empty .
```


Module QLOCK specifying "QLOCK" (1-2)

qlock.mod

```
mod* QLOCK {
  pr(LABEL)
  pr(QUEUE(PID{sort Elt -> Pid,
               sort EltErr -> PidErr,
               op (none):EltErr -> none):PidErr}))
  *[Sys]*
-- any initial state
  op init : -> Sys
-- observations
  bop pc    : Sys Pid -> Label
  bop queue : Sys -> Queue
-- actions
  bop want : Sys Pid -> Sys
  bop try  : Sys Pid -> Sys
  bop exit : Sys Pid -> Sys
-- for any initial state
  eq pc(init,I:Pid)  = rm .
  eq queue(init) = empty .
```

Module QLOCK specifying "QLOCK" (2)

qlock.mod

```
var S : Sys . vars I J : Pid .
-- for want
op c-want : Sys Pid -> Bool {strat: (0 1 2)}
eq c-want(S,I) = (pc(S,I) = rm) .
--
ceq pc(want(S,I),J)
    = (if I = J then wt else pc(S,J) fi)
      if c-want(S,I) .
ceq queue(want(S,I)) = put(I,queue(S))
      if c-want(S,I) .
ceq want(S,I) = S
      if not c-want(S,I) .
```


Module QLOCK specifying "QLOCK" (3)

qlock.mod

```
-- for try
  op c-try : Sys Pid -> Bool {strat: (0 1 2)}
  eq c-try(S,I) = (pc(S,I) = wt and top(queue(S)) = I) .
  --
  ceq pc(try(S,I),J)
      = (if I = J then cs else pc(S,J) fi) if c-try(S,I) .
  eq  queue(try(S,I)) = queue(S) .
  ceq try(S,I)        = S                    if not c-try(S,I) .
-- for exit
  op c-exit : Sys Pid -> Bool {strat: (0 1 2)}
  eq c-exit(S,I) = (pc(S,I) = cs) .
  --
  ceq pc(exit(S,I),J)
      = (if I = J then rm else pc(S,J) fi) if c-exit(S,I) .
  ceq queue(exit(S,I)) = get(queue(S))     if c-exit(S,I) .
  ceq exit(S,I)        = S                 if not c-exit(S,I) .
}
```

(_ =*= _) is congruent for OTS

The binary relation (`S1:Sys =*= S2:Sys`) is defined to be true iff S1 and S2 have the same observation values.

OTS style of defining the possible changes of the values of observations is characterized by the equations of the form:

$o(a(s,d),d') = \ldots o_1(s,d_1) \ldots o_2(s,d_2) \ldots o_n(s,d_n) \ldots$

for appropriate data values of $d, d', d_1, d_2, \ldots, d_n$.

It can be shown that OTS style guarantees that (_ =*= _) is congruent with respect to all actions.

R_{QLOCK} (set of reachable states) of OTS_{QLOCK} (OTS defined by the module QLOCK)

Signature determining R_{QLOCK}

```
-- any initial state
  op init : -> Sys
-- actions
  bop want : Sys Pid -> Sys
  bop try  : Sys Pid -> Sys
  bop exit : Sys Pid -> Sys
```

Recursive definition of R_{QLOCK}

$$R_{QLOCK} = \{init\} \cup \{want(s,i) \mid s \in R_{QLOCK}, i \in Pid\} \cup \{try(s,i) \mid s \in R_{QLOCK}, i \in Pid\} \cup \{exit(s,i) \mid s \in R_{QLOCK}, i \in Pid\}$$

Mutual exclusion property as an invariant

invariants-0.mod

```
mod INV1 {
   pr(QLOCK)
-- declare a predicate to verify to be an invariant
   pred inv1 : Sys Pid Pid
-- CafeOBJ variables
   var S :  Sys .
   vars I J : Pid .
-- define inv1 to be the mutual exclusion property
   eq inv1(S,I,J)
       = (((pc(S,I) = cs) and (pc(S,J) = cs)) implies I = J) .
}
```

Formulation of proof goal for mutual exclusion property

INV1 |= $\forall s \in R_{QLOCK} \forall i,j \in$ Pid.inv1(s,i,j)

Induction scheme induced by the structure of R_{QLOCK}

mx(s) =def= $\forall i,j \in$ Pid.inv1(s,i,j)

{ INV1 |= mx(init),
INV1 ∪ {mx(s)=true} |= $\forall k$. mx(want(s,k)),
INV1 ∪ {mx(s)=true} |= $\forall k$. mx(try(s,k)),
INV1 ∪ {mx(s)=true} |= $\forall k$. mx(exit(s,k)) }
implies
INV1 |= $\forall s \in R_{QLOCK}$.mx(s)

Schematic signature diagram for OTS