I636: Specification and Verification of Distributed Systems
1. Outline of How to Analyze Systems with Maude

Kazuhiro Ogata & Kokichi Futatsugi

Outline

• The course investigates how to make mathematical models (i.e. state machines or transition systems) of distributed systems, specify such models in Maude, and model check that such models satisfy (or do not satisfy) properties with Maude.
• This lecture gives an outline of how to model a mutual exclusion protocol, specify the model in Maude, and model check that the model does not satisfy (and revised models satisfy) some properties with Maude.
Maude

- A spec. & prog. lang. based on membership equational logic and rewriting logic.
- A direct successor of OBJ3.
  (CafeOBJ is another direct successor of OBJ3.)
- Many excellent functionalities:
  - Fast Associative-Commutative (AC) rewriting
  - Model checking facilities
    (intensively used in this course)
  - Meta-programming facilities
  - ...

Mutual Exclusion Problem

- Many resources shared by multiple entities
  (processors, processes, threads, etc.) in computer systems.
- Some resources such as memory locations
  should be used by at most one entity at any given moment.
- The mutual exclusion (mutex) problem* is to find a mechanism that assures it.

* Edsger W. Dijkstra: Solution of a problem in concurrent programming control.
Commun. ACM 8(9): 569 (1965)
Mutual Exclusion Protocols

• Solutions of the mutex problem.
• Many mutex protocols have been proposed. Among them are
• Among the properties mutex protocols should satisfy are
  – The mutual exclusion property
    At most one entity is in a critical section at any given moment. A critical section is a fragment of a protocol in which a shared resource is used.
  – The lockout (starvation) freedom property
    Whenever an entity wants to enter a critical section, it will eventually enter it.

Qlock0: A (Flawed) Mutex Protocol (1)

The program for each process $i$:

```plaintext
Loop {
    Remainder Section
    rs: tmp := enqueue(queue, i);
    es: queue := tmp;
    ws: repeat until top(queue) = i;
        Critical Section
        cs: tmp := dequeue(queue);
        ds: queue := tmp
    }
```

• queue is a queue of process IDs, shared by all processes.
• tmp is a temporary variable.
• rs, es, ws, cs and ds are labels.
• Initially, queue and tmp are empty and each process is at label cs.
Qlock0: A (Flawed) Mutex Protocol (2)

\[
\text{Loop}\{ \\
\text{Remainder Section} \\
p_1(t: \text{empty}) \quad p_2(t: \text{empty}) \\
rs: \quad \text{tmp := enqueue}(\text{queue}, i); \\
\text{es:} \\
\text{queue := tmp;} \\
\text{ws:} \\
\text{repeat until top(queue) = i;} \\
\text{Critical Section} \\
p_1(t: p_1) \quad p_2(t: p_1 p_2) \\
cs: \quad \text{tmp := dequeue(queue);} \\
p_1(t: p_1) \quad p_2(t: p_1 p_2) \\
ds: \quad \text{queue := tmp} \\
\}
\]

\text{queue: } \text{p} \text{\#}\text{p}\text{p}\text{y} \quad \text{A nonempty queue is represented as } e_1 e_2 \ldots e_n

\text{The top element} \quad \text{The bottom element}

Modeling Qlock0 (1)

- Make a mathematical model of Qlock0 as a state machine (transition system) to check if Qlock0 satisfies desired properties.
- A state consists of, for each process \( p \), the \( p \)'s location and the value of the \( p \)'s tmp, and the value of queue.
- A state is depicted as follows.
Modeling Qlock0 (2)

Loop

Remainder Section
rs: tmp := enqueue(queue, i);
es: queue := tmp;
ws: repeat until top(queue) = i;
Critical Section
cs: tmp := dequeue(queue);
ds: queue := tmp

Modeling Qlock0 (3)

Loop

Remainder Section
rs: tmp := enqueue(queue, i);
es: queue := tmp;
ws: repeat until top(queue) = i;
Critical Section
cs: tmp := dequeue(queue);
ds: queue := tmp

rl [eq2]:
(pc[I]: es) (queue: Q) (tmp[I]: R)
=>
(pc[I]: ws) (queue: R) (tmp[I]: R) .
Modeling Qlock0 (4)

Loop

\[ rs: \text{tmp} := \text{enqueue(queue, } i) \];
\[ \text{es: queue := tmp;} \]
\[ ws: \text{repeat until top(queue) = I;} \]
\[ \text{cs: tmp := dequeue(queue);} \]
\[ \text{ds: queue := tmp} \]

\( q_1 \ldots q_n \)

\( \text{wt} \) If \( q_i \) is \( p \)

Remainder Section
\[ \text{rs: tmp := enqueue(queue, } i) \];
\[ \text{es: queue := tmp;} \]
\[ \text{ws: repeat until top(queue) = I;} \]
\[ \text{cs: tmp := dequeue(queue);} \]
\[ \text{ds: queue := tmp} \]

Modeling Qlock0 (5)

Loop

\[ rs: \text{tmp} := \text{enqueue(queue, } i) \];
\[ \text{es: queue := tmp;} \]
\[ ws: \text{repeat until top(queue) = i;} \]
\[ \text{cs: tmp := dequeue(queue);} \]
\[ \text{ds: queue := tmp} \]

\( q_1 \ldots q_n \)

\( \text{dq1} \) \( [dq1] \)

Remainder Section
\[ \text{rs: tmp := enqueue(queue, } i) \];
\[ \text{es: queue := tmp;} \]
\[ \text{ws: repeat until top(queue) = i;} \]
\[ \text{cs: tmp := dequeue(queue);} \]
\[ \text{ds: queue := tmp} \]

\( q_1 \ldots q_n \)

\( \text{dql} \) \( [dql] \)

\( q_1 \ldots q_n \)
Modeling Qlock0 (6)

```
Loop
  Remainder Section
  rs: tmp := enqueue(queue, i);
  es: queue := tmp;
  ws: repeat until top(queue) = i;
  Critical Section
  cs: tmp := dequeue(queue);
  ds: queue := tmp
}
```

System Specification of Qlock0

- Written as a system module.

```
mod QLOCK is
  pr LABEL . pr PID . pr QUEUE .
  sorts Val Sys .
  subsorts Val < Sys .
  *** Configurations
  *** State components
  op pc[_]:_ : Pid Label -> Val .
  op queue:_ : Queue -> Val .
  op tmp[_]:_ : Pid Queue -> Val .
  *** Maude variables
  vars Q R : Queue .
  var I : Pid .
  *** Rules
  ...
endm
```

The five rules are put here.
Data Used in System Specification (1)

• Written as functional modules

```plaintext
fmod LABEL is
  sort Label .
  ops rs es ws cs ds : -> Label .
endfm

fmod PID is
  sort Pid .
  ops p1 p2 nop : -> Pid .
  *** nop is used as a dummy ID.
endfm
```

Data Used in System Specification (2)

```plaintext
fmod QUEUE is
  pr PID .
  sort Queue .
  op empty : -> Queue .
  op _ _ : Pid Queue -> Queue .
  op enq : Queue Pid -> Queue .
  op deq : Queue -> Queue .
  op top : Queue -> Pid .
  var Q : Queue .
  vars X Y : Pid .
  eq enq(empty,X) = X empty .
  eq enq((Y Q),X) = Y enq(Q,X) .
  eq deq(empty) = empty .
  eq deq((X Q)) = Q .
  eq top(empty) = nop .
  eq top((X Q)) = X .
endfm
```
Requirements on System Specifications

• Requirements on a system specification (a set of equations + a set of rewriting rules (transitions)) so that the system specification can be effectively executed.

• Requirements on equations
  – Terminating & confluence.
  – Admissible

• Requirements on rewriting rules
  – Admissible
  – Coherence

Initial State

• Assume that there are two processes.
• The initial state is defined as follows:

```
mod QLOCK-INIT is
  pr QLOCK .
  op init : -> Sys .
  eq init = (pc[p1]: rs) (pc[p2]: rs) (queue: empty)
             (tmp[p1]: empty) (tmp[p2]: empty) .
endm
```

• Note that the state space is unbounded because a queue is used.
• Note that even the reachable state space is unbounded. Check it yourself.
Checking the Mutex Property (1)

• The mutex property is that at most one process is in the critical section in any reachable states.
• Command search traverses the reachable states from a given state to find states such that some conditions hold.
• The command can be used to find a counterexample showing that Qlock0 does not satisfy the property.

Checking the Mutex Property (2)

• The command is as follows:
  ```
  search [1] in QLOCK-INIT :
  init =>* (pc[p1]: cs) (pc[p2]: cs) (S:Sys) .
  ```
• The command traverses the reachable state space from the initial state init to find one state such that the two processes p1 and p2 are in the critical section simultaneously.
• The command finds state 33 as such as state.
• Command show path generates an execution sequence leading to a given state from the state with which search begins.
• A counterexample showing that Qlock0 does not satisfy the mutex property can be generated with the command:
  ```
  show path 33 .
  ```
Checking the Mutex Property (3)

• The counterexample generated is like
  state 0, Sys: queue: empty (pc[p1]: rs) (pc[p2]: rs) (tmp[p1]: empty) tmp[p2]: empty
  ... ===>
  state 13, Sys: queue: (p1 empty) (pc[p1]: cs) (pc[p2]: es) (tmp[p1]: p1 empty) tmp[p2]: p2 empty
  ... eq2 ... ===>
  state 23, Sys: queue: (p2 empty) (pc[p1]: cs) (pc[p2]: ws) (tmp[p1]: p1 empty) tmp[p2]: p2 empty
  ... wt ... ===>
  state 33, Sys: queue: (p2 empty) (pc[p1]: cs) (pc[p2]: cs) (tmp[p1]: p1 empty) tmp[p2]: p2 empty

• The reason why the counterexample can be generated is that process IDs are not put ATOMICALLY into queue.
  rs: tmp := enqueue(queue, i);
  es: queue := tmp;

So, even when p1 is in the critical section in state 13, p1 is deleted from queue and p2 is top of queue in state 23.

Qlock1: A Modification of Qlock0 (1)

The program for each process i:

```
Loop {
    Remainder Section
    rs: atomic_enqueue(queue, i);
    es: queue := tmp;
    ws: repeat until top(queue) = i;
    Critical Section
    cs: tmp := dequeue(queue);
    ds: queue := tmp
}
```

• atomic_enqueue puts an element into a queue at the bottom position atomically
Modeling Qlock1 (1)

Loop

\[
\begin{align*}
& \text{Remainder Section} \\
& \text{rs: atomic_enqueue(queue, I);} \\
& \text{ws: repeat until top(queue) = I;} \\
& \text{Critical Section} \\
& \text{cs: tmp := dequeue(queue);} \\
& \text{ds: queue := tmp} \\
\end{align*}
\]

Critical Section

\[
\begin{align*}
& rl_{\text{eq}}: \\
& (pc[I]: rs) (queue: Q) \\
& \Rightarrow \\
& (pc[I]: es) (queue: enq(Q, I)).
\end{align*}
\]

Modeling Qlock1 (2)

- Assume that the number of processes is limited to some fixed number, say two.
- The entire state space is still unbounded.
- But, the reachable state space is bounded.
Verification of the Mutex Property

• The command
  search [1] in QLOCK-INIT :
  init =>*
  (pc[p1]: cs) (pc[p2]: cs) (S:Sys) .
  successfully terminates and does not find any such states.
• This means that when there are two processes, Qlock1 satisfies the mutex property.

Another Way of the Mutex Property Verification (1)

• When the reachable state space is bounded, the model checker can be used to verify properties.
• To this end, some state predicates are defined, and LTL formulas corresponding to properties are defined based on the state predicates.
Another Way of the Mutex Property Verification (2)

- State predicates and LTL formulas are defined as follows:

```plaintext
mod QLOCK-PREDS is
   pr QLOCK-INIT.
   inc SATISFACTION.
   subsort Sys < State.
   op crit : Pid -> Prop.
   var P : Pid. var S : Sys.
   eq (pc[P] : cs) S |= crit(P) = true.
endm
```

- A given process is in the critical section

```plaintext
mod QLOCK-CHECK is
   inc QLOCK-PREDS.
   inc MODEL-CHECKER.
   inc LTL-SIMPLIFIER.
   op mutex : -> Formula.
   eq mutex = ([] ~(crit(p1) /
                     crit(p2))) .
endm
```

- Conjunction
- Henceforth (Always)
- Negation

Another Way of the Mutex Property Verification (3)

- The command

```plaintext
red in QLOCK-CHECK :
   modelCheck(init,mutex) .
```

verifies that Qlock1 satisfies the mutex property with the model checker when two processes are involved.

- The command successfully verifies it.
Checking the Lockout Freedom Property (1)

• The property is that whenever a process wants to enter the critical section, it will eventually enter it.
• For the checking, the following state predicate is defined in module QLOCK-PREDs:
  \[
  \text{op wait : Pid} \rightarrow \text{Prop}.
  \text{eq (pc[P] : ws) S |= wait(P) = true}.
  \]
• The following LTL formula is defined in module QLOCK-CHECK:
  \[
  \text{op lofree : Formula}.
  \text{eq lofree = (wait(p1) |-> crit(p1))}
  \backslash (\text{wait(p2) |-> crit(p2)}) .
  \]

Checking the Lockout Freedom Property (2)

• The command
  \[
  \text{red in QLOCK-CHECK :}
  \text{modelCheck(init,lofree)} .
  \]
find a counterexample showing that Qlock1 does not satisfy the lockout freedom property with the model checker.
Checking the Lockout Freedom Property (3)

• The counterexample generated is like

\[
\begin{align*}
\text{queue: empty (pc[p1]: rs) (pc[p2]: rs)} \\
\quad (\text{tmp}[p1]: \text{empty}) \text{ tmp}[p2]: \text{empty, 'eq}) \\
\quad \ldots \\
\text{queue: (p2 empty) (pc[p1]: rs) (pc[p2]: cs)} \\
\quad (\text{tmp}[p1]: \text{empty}) \text{ tmp}[p2]: \text{empty, 'eq}) \\
\text{queue: (p2 empty) (pc[p1]: rs) (pc[p2]: ds)} \\
\quad (\text{tmp}[p1]: \text{empty}) \text{ tmp}[p2]: \text{empty, 'eq}) \\
\end{align*}
\]

Why?: Process IDs are not deleted ATOMICALLY from queue.

Qlock2: A Modification of Qlock1 (1)

The program for each process \(i\):

```plaintext
Loop {
    Remainder Section
    rs: atomic_enqueue(queue, i);
    ws: repeat until top(queue) = i;
    Critical Section
    cs: tmp = dequeue(queue); 
    ds: queue = tmp
}
```

• atomic_dequeue deletes the top element from a queue atomically.
Modeling Qlock2

\[
\text{Loop} \quad \text{Remainder Section}
\begin{align*}
rs & : \text{atomic\_enqueue(queue,} i) ; \\
ws & : \text{repeat until top(queue) = } i ; \\
\text{Critical Section}
\end{align*}
\begin{align*}
rs & : \text{atomic\_dequeue(queue)}
\]

\[
q_1 \ldots q_n \\
eq
\]

Verification of the Lockout Freedom Property

- The command
  \[
  \text{red in QLOCK-CHECK : modelCheck(init,lofree) .}
  \]
  verifies that Qlock2 satisfies the lockout freedom property with the model checker when two processes are involved.
- The command successfully verifies it.
- It is also verified that Qlock2 satisfies the mutex property.
Fairness Assumptions (1)

- The mutex property is a *safety property*, while the lockout freedom property is a *liveness property*.
- Generally, some fairness should be assumed to verify (model check) liveness properties.
- The Maude model checker does not use any fairness assumptions.
- The reason why it is successfully verified that Qlock2 satisfies the lockout freedom property is because each process tries to enter the critical section infinitely often.

Fairness Assumptions (2)

- If a change is made such that each process does not try to enter the critical section infinitely often, verification of the lockout freedom property is not successful.
- The following rewriting rule is added:
  
  \[
  rl \ [xr] : (pc[I]: rs) (queue: Q) \\
  \Rightarrow (pc[I]: rs) (queue: Q) .
  \]

  The modification of Qlock2 in this way is called Qlock3.
- Some fairness assumption should be used to successfully verify the lockout freedom property.
Fairness Assumptions (3)

- The fairness assumption used is as follows:
  \[
  \text{op fair : } \rightarrow \text{Formula.} \quad \text{Eventually}
  \]
  \[
  \text{eq fair} = (\langle \langle \text{wait(p1)} \rangle \rangle \chi / \langle \langle \text{wait(p2)} \rangle \rangle).
  \]
- The fairness assumption says that each process tries to enter the critical section infinitely often.
- The command
  \[
  \text{red in QLOCK-CHECK :}
  \]
  \[
  \text{modelCheck(init,fair -> lofree).}
  \]
  successfully verifies that Qlock3 satisfies the lockout freedom property using the fairness assumption when two processes are involved.