I217: Functional Programming

7. Multisets

(Model Checking for Invariant Properties)

Kazuhiro Ogata, Canh Minh Do

Roadmap

- Transition rules
- Formal specification of a flawed version of TAS in which a record is used to express states and invariant model checking with the search predicate
- Multisets
- Use of multisets to express states
- State machines and invariant properties
- A flawed version of Qlokc and a corrected version of Qlock

Transition Rules

Declared as follows:

trans *LeftTerm* => *RightTerm* .

where *LeftTerm* and *RightTerm* are terms of a same sort.

If variables X_1, X_2, … of sorts S_1, S_2, … occur in the rule, then the equation says that for all X_1 of S_1, all X_2 of S_2, … *LeftTerm* changes to *RightTerm*.

(Precisely, the least sort of *RightTerm* is the same as or a sub-sort of the lest sort of *LeftTerm*, or equivalently the least sort of *LeftTerm* is a sort of *RightTerm*.)

Spec. of a Mutex Protocol with trans

Let us use the same mutex protocol as the one used in lecture note 6.

Loop: "Remainder Section"
rs: **while** *locked* = true {}
ms: *locked* := true;
"Critical Section"
cs: *locked* := false;

Spec. of a Mutex Protocol with trans

```
mod! TID {
  [Tid]
  ops t1 t2 t3 t4 : -> Tid {constr} .
}

mod! LOC {
  [Loc]
  ops rs ms cs ws es ds : -> Loc {constr} .
}

mod! STATE2 {
  pr(LOC)
  [State2]
  op (locked:_,pc1:_,pc2:_) : Bool Loc Loc -> State2 {constr} .
}
```


Spec. of a Mutex Protocol with trans

```
mod! FMUTEX2 {
  pr(STATE2)
  pr(TID)
  vars L1 L2 : Loc .
  var B : Bool .
  -- t1
  trans [want1] : (locked: false,pc1: rs,pc2: L2)
     => (locked: false,pc1: ms,pc2: L2) .
  trans [try1] : (locked: B,pc1: ms,pc2: L2)
     => (locked: true,pc1: cs,pc2: L2) .
  trans [exit1] : (locked: B,pc1: cs,pc2: L2)
     => (locked: false,pc1: rs,pc2: L2) .
```

Spec. of a Mutex Protocol with trans

```
-- t2
 trans [want2] : (locked: false,pc1: L1,pc2: rs)
   => (locked: false,pc1: L1,pc2: ms) .
 trans [try2] : (locked: B,pc1: L1,pc2: ms)
   => (locked: true,pc1: L1,pc2: cs) .
 trans [exit2] : (locked: B,pc1: L1,pc2: cs)
   => (locked: false,pc1: L1,pc2: rs) .
}
```


Invariant Model Checking with Search

red *initialState* =(1,*)=>* *statePattern* .

This searches the state space reachable from *initialState* for a state that matches *statePattern* by breadth-first search.

red *initialState* =(1,*)=>* *statePattern*
 suchThat *condition* .

This searches the state space reachable from *initialState* for a state that matches *statePattern* and for which *condition* holds by breadth-first search.

Invariant Model Checking with Search

```
open FMUTEX2 .
 red (locked: false, pc1: rs, pc2: rs)
  =(1,*)=>* (locked: B, pc1: cs, pc2: cs) .
close
```

A state is found:

```
** Found [state 0-8] (locked: true , pc1: cs , pc2: cs):State2
-- target: (locked: B , pc1: cs , pc2: cs)
 { B |-> true, B |-> true, L1 |-> cs }
```

should be { B |-> true, L1 |-> cs }

Invariant Model Checking with Search

```
open FMUTEX2 .
 red (locked: false, pc1: rs, pc2: rs)
  =(1,*)=>* (locked: B, pc1: cs, pc2: cs) .
 show path 0-8 .
close
```

A counterexample is displayed.

```
[state 0-0] (locked: false , pc1: rs , pc2: rs):State2
 trans [want1]: (locked: false , pc1: rs , pc2: L2) => (locked: false , pc1: ms , pc2: L2)
[state 0-1] (locked: false , pc1: ms , pc2: rs):State2
 trans [want2]: (locked: false , pc1: L1 , pc2: rs) => (locked: false , pc1: L1 , pc2: ms)
[state 0-4] (locked: false , pc1: ms , pc2: ms):State2
 trans [try1]: (locked: B , pc1: ms , pc2: L2) => (locked: true , pc1: cs , pc2: L2)
[state 0-6] (locked: true , pc1: cs , pc2: ms):State2
 trans [try2]: (locked: B , pc1: L1 , pc2: ms) => (locked: true , pc1: L1 , pc2: cs)
[state 0-8] (locked: true , pc1: cs , pc2: cs):State2
```

Multisets

Collections such that multiple occurrences of elements are permitted and the order in which elements are enumerated is irrelevant

{0,1,2,3} {0,1,0,2,1,3} {0,0,1,1,2,3} {3,2,1,0}

{0,1,2,3} is the same as {3,2,1,0}
but different from {0,1,0,2,1,3}

{0,1,0,2,1,3} is the same as {0,0,1,1,2,3}

The operator attributes assoc, comm, and id: make it possible to expresses multisets in CafeOBJ.

Multisets

```
mod! MULTISET(E :: TRIV) {
  [Elt.E < MSet]   An element is also a singleton multiset.
  op emp : -> MSet {constr} .   The empty multiset
  op _ _ : MSet MSet -> MSet {constr assoc comm id: emp} .
}
```

$(e_1\ e_2)\ e_3$ is the same as $e_1\ (e_2\ e_3)$ because of **assoc**(iativity).

$e_1\ e_2$ is the same as $e_2\ e_1$ because of **comm**(utativity).

ms emp is the same as *ms* and so is emp *ms* because of **id:** emp.

Multisets

```
open MULTISET(NAT) .
 red (1 2) 3 == 1 (2 3) .
 red 1 2 == 2 1 .
 red 1 1 1 2 2 == 1 2 1 1 2 .
 red emp 1 emp 2 emp 1 emp 1 emp 2 emp .
 red emp .
 red emp emp .
close
```


State Representation Using Multisets

An observable component is a name-value pair (*name*: *val*).

(pc[*t*]: *l*) thread *t* is located at location *l*.

(locked: *b*) the value stored in variable *locked* is *b*.

```
mod! OCOMP principal-sort OComp {
 pr(TID)
 pr(LOC)
 [OComp]
 op (pc[_]:_)  : Tid Loc -> OComp {constr} .
 op (locked:_) : Bool  -> OComp {constr} .
}
```

State Representation Using Multisets

A state is expressed as a braced multiset of observable components, such as {(locked: false) (pc[t1]: rs) (pc[t2]: rs)}.

```
mod! STATE {
 pr(MULTISET(OCOMP) * {sort MSet -> OComps})
 [State]
 op {_} : OComps -> State {constr} .
}
```


State Representation Using Multisets

```
mod! FMUTEX {
 pr(STATE)
 vars T T1 T2 : Tid .
 var B : Bool .
 var OCs : OComps .
 trans [want] : {(locked: false) (pc[T]: rs) OCs}
   => {(locked: false) (pc[T]: ms) OCs} .
 trans [try] : {(locked: B) (pc[T]: ms) OCs}
   => {(locked: true) (pc[T]: cs) OCs} .
 trans [exit] : {(locked: B) (pc[T]: cs) OCs}
   => {(locked: false) (pc[T]: rs) OCs} .
}
```

State Representation Using Multisets

```
open FMUTEX .
 red {(locked: false) (pc[t1]: rs) (pc[t2]: rs)}
  =(1,*)=>* {(pc[T1]: cs) (pc[T2]: cs) OCs} .
 show path 0-8 .
close
```


A Corrected Version of the Protocol

Loop: "Remainder Section"
  rs: **while** test&set(*locked*) = true {}
    "Critical Section"
  cs: *locked* := false;

test&set(x) does the following atomically:

$tmp := x;$
$x :=$ true;
return tmp;

The corrected version of the protocol is called TAS, while the flawed version is called FTAS.

A Corrected Version of the Protocol

```
mod! STATE3 {
 pr(LOC)
 [State3]
 op (locked:_,pc1:_,pc2:_,pc3:_) : Bool Loc Loc Loc -> State3 {constr} .
}

mod! MUTEX3 {
 pr(STATE3)
 pr(TID)
 vars L1 L2 L3 : Loc .
 var B : Bool .
 -- t1
 trans [try1] : (locked: false,pc1: rs,pc2: L2,pc3: L3)
   => (locked: true,pc1: cs,pc2: L2,pc3: L3) .
 trans [exit1] : (locked: B,pc1: cs,pc2: L2,pc3: L3)
   => (locked: false,pc1: rs,pc2: L2,pc3: L3) .
```


A Corrected Version of the Protocol

```
 -- t2
 trans [try2] : (locked: false,pc1: L1,pc2: rs,pc3: L3)
   => (locked: true,pc1: L1,pc2: cs,pc3: L3) .
 trans [exit2] : (locked: B,pc1: L1,pc2: cs,pc3: L3)
   => (locked: false,pc1: L1,pc2: rs,pc3: L3) .
 -- t3
 trans [try2] : (locked: false,pc1: L1,pc2: L2,pc3: rs)
   => (locked: true,pc1: L1,pc2: L2,pc3: cs) .
 trans [exit2] : (locked: B,pc1: L1,pc2: L2,pc3: cs)
   => (locked: false,pc1: L1,pc2: L2,pc3: rs) .
}
```

A Corrected Version of the Protocol

```
open MUTEX3 .
 red (locked: false, pc1: rs, pc2: rs, pc3: rs)
  =(1,*)=>* (locked: B, pc1: L1, pc2: L2, pc3: L3)
  suchThat (L1 == cs and L2 == cs) or
           (L1 == cs and L3 == cs) or (L2 == cs and L3 == cs) .
close
```


A Corrected Version of the Protocol

```
mod! MUTEX {
 pr(STATE)
 vars T T1 T2 : Tid .
 var B : Bool .
 var OCs : OComps .
 trans [try] : {(locked: false) (pc[T]: rs) OCs}
   => {(locked: true) (pc[T]: cs) OCs} .
 trans [exit] : {(locked: B) (pc[T]: cs) OCs}
   => {(locked: false) (pc[T]: rs) OCs} .
}

open MUTEX .
 red {(locked: false) (pc[t1]: rs) (pc[t2]: rs) (pc[t3]: rs)}
   =(1,*)=>* {(pc[T1]: cs) (pc[T2]: cs) OCs} .
close
```

State Machines & Invariants

A set of states

A set of state-state pairs (s, s') (which may be denoted $s \rightarrow s'$)

A state machine M is defined as $\langle S, I, T \rangle$.

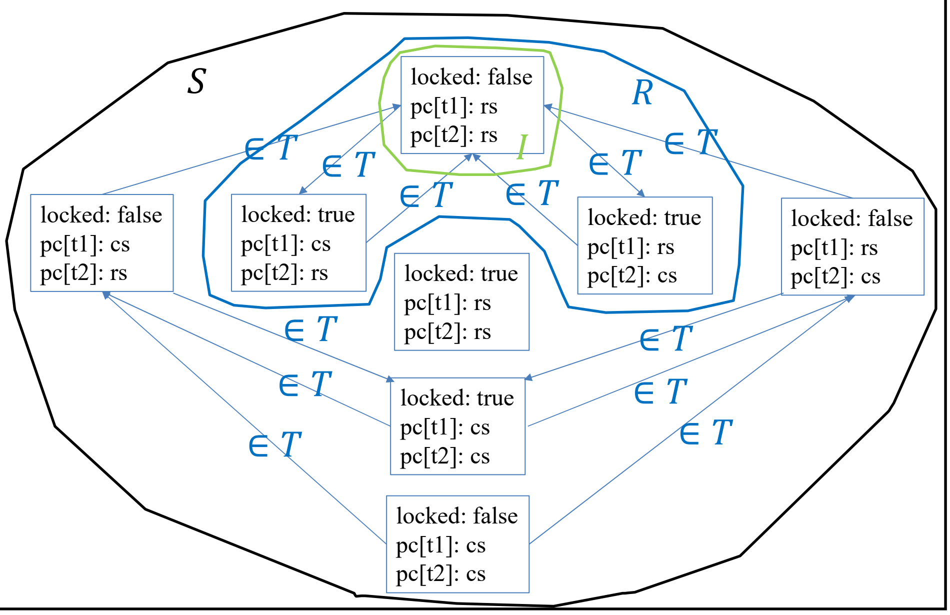
The set of initial states

The set R of reachable states w.r.t. M is defined as follows:

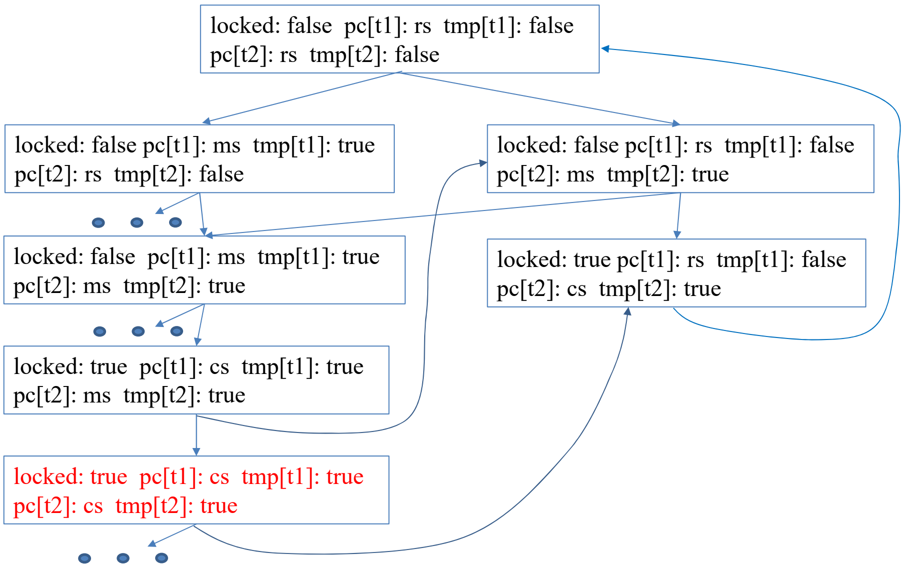
1. $I \subseteq R$
2. if $s \in R$ and $(s, s') \in T$, then $s' \in R$

A state predicate p of M is an invariant (or an invariant property) of M if and only if $(\forall s \in R)\ p(s)$ holds.

State Machines & Invariants



State Machines & Invariants



FQlock: A Flawed Version of Qlock

Alice Bob Cathy David Emma

How to make sure at most one person is given the permission to use the shared bike?

FQlock: A Flawed Version of Qlock

A queue may be used to do so.

Suppose Emma, David & Alice enqueues their initials into the queue in this order.

Emma is the 1st person who is given the permission. When her use is done, the queue is dequeued.

David is the 1st person who is given the permission.

FQlock: A Flawed Version of Qlock

Based on the idea mentioned so far, they could come up with a mutual exclusion protocol. The pseudo-code processed by each person (or process) i:

$$
\begin{array}{l}
\textbf{Loop}: \text{“Remainder Section”} \\
\quad \text{rs}: \ \mathit{tmp}_i := \text{enq}(\mathit{queue}, i); \\
\quad \text{es}: \ \mathit{queue} := \mathit{tmp}_i; \\
\quad \text{ws}: \ \textbf{repeat until } \text{top}(\mathit{queue}) = i; \\
\qquad\quad \text{“Critical Section”} \\
\quad \text{cs}: \ \mathit{tmp}_i := \text{deq}(\mathit{queue}); \\
\quad \text{ds}: \ \mathit{queue} := \mathit{tmp}_i;
\end{array}
$$

Each i is always located at one of rs, es, ws, cs and ds. Initially, each i is located at rs and *queue* is empty.

When i is located at cs, i has the permission to use the bike.

Let us call the protocol FQlock.

FQlock: A Flawed Version of Qlock

```
mod! QUEUE(E :: TRIV) {
  [Elt.E < Queue]
  op emq : -> Queue {constr} .
  op _;_ : Queue Queue -> Queue {constr assoc id: emq} .
}
```

```
mod! OCOMP principal-sort OComp {
 pr(QUEUE(TID))
 pr(LOC)
 [OComp]
 op (pc[_]:_)  : Tid Loc -> OComp {constr} .
 op (tmp[_]:_)  : Tid Queue -> OComp {constr} .
 op (queue:_) : Queue  -> OComp {constr} .
}
```


FQlock: A Flawed Version of Qlock

```
mod! STATE {
 pr(MULTISET(OCOMP) * {sort MSet -> OComps})
 [State]
 op {_} : OComps -> State {constr} .
}
```

FQlock: A Flawed Version of Qlock

```
mod! FQLOCK {
 pr(STATE)
 vars T T1 T2 : Tid .  vars Q1 Q2 : Queue .  var OCs : OComps .
 trans [want] : {(queue: Q1) (pc[T]: rs) (tmp[T]: Q2) OCs}
   => {(queue: Q1) (pc[T]: es) (tmp[T]: (Q1 ; T)) OCs} .
 trans [enq] : {(queue: Q1) (pc[T]: es) (tmp[T]: Q2) OCs}
   => {(queue: Q2) (pc[T]: ws) (tmp[T]: Q2) OCs} .
 trans [try] : {(queue: (T ; Q1)) (pc[T]: ws) (tmp[T]: Q2) OCs}
   => {(queue: (T ; Q1)) (pc[T]: cs) (tmp[T]: Q2) OCs} .
 trans [deq1] : {(queue: (T1 ; Q1)) (pc[T]: cs) (tmp[T]: Q2) OCs}
   => {(queue: (T1 ; Q1)) (pc[T]: ds) (tmp[T]: Q1) OCs} .
 trans [deq2] : {(queue: emq) (pc[T]: cs) (tmp[T]: Q2) OCs}
   => {(queue: emq) (pc[T]: ds) (tmp[T]: emq) OCs} .
 trans [exit] : {(queue: Q1) (pc[T]: ds) (tmp[T]: Q2) OCs}
   => {(queue: Q2) (pc[T]: rs) (tmp[T]: Q2) OCs} .
}
```


FQlock: A Flawed Version of Qlock

```
open FQLOCK .
 red {(queue: emq) (pc[t1]: rs) (pc[t2]: rs) (tmp[t1]: emq) (tmp[t2]: emq)}
   =(1,*)=>* {(pc[T1]: cs) (pc[T2]: cs) OCs} .
 show path 0-33 .
close
```

Qlock

Loop : "Remainder Section"
rs : enq($queue, i$);
ws : **repeat until** top($queue$) $= i$;
"Critical Section"
cs : deq($queue$);

Note that both enq & deq used in Qlock update *queue*.

Qlock

```
mod! QLOCK {
 pr(STATE)
 vars T T1 T2 : Tid .
 vars Q1 Q2 : Queue .
 var OCs : OComps .
 trans [want] : {(queue: Q1) (pc[T]: rs) OCs}
  => {(queue: (Q1 ; T)) (pc[T]: ws) OCs} .
 trans [try] : {(queue: (T ; Q1)) (pc[T]: ws) OCs}
  => {(queue: (T ; Q1)) (pc[T]: cs) OCs} .
 trans [exit1] : {(queue: (T1 ; Q1)) (pc[T]: cs) OCs}
  => {(queue: Q1) (pc[T]: rs) OCs} .
 trans [exit2] : {(queue: emq) (pc[T]: cs) OCs}
  => {(queue: emq) (pc[T]: rs) OCs} .
}
```

Qlock

```
open QLOCK .
 red {(queue: emq) (pc[t1]: rs) (pc[t2]: rs)}
   =(1,*)=>* {(pc[T1]: cs) (pc[T2]: cs) OCs} .
close

open QLOCK .
 red {(queue: emq) (pc[t1]: rs) (pc[t2]: rs) (pc[t3]: rs)}
   =(1,*)=>* {(pc[T1]: cs) (pc[T2]: cs) OCs} .
close

open QLOCK .
 red {(queue: emq) (pc[t1]: rs) (pc[t2]: rs) (pc[t3]: rs) (pc[t4]: rs)}
   =(1,*)=>* {(pc[T1]: cs) (pc[T2]: cs) OCs} .
close
```


Exercises

1. Write all programs (or formal specifications) in the slides, feed them into the CafeOBJ system, and observe the results returned by the CafeOBJ system.

2. Revise the two kinds of the programs (or formal specifications) of TAS such that there are four threads participating in the protocol, where one uses records to represent states and the other uses multisets to represent states, feed the revised versions into the CafeOBJ system, and observe the results returned by the CafeOBJ system.

Exercises

3. Make comparison of the two ways to represent states, where one uses records and the other uses multisets.

4. Draw a diagram for Qlock when there are two threads participating in Qlock like the one shown on p.23 for TAS.

Exercises

5. How many reachable states does FQlock have? Describe what can support your answer?
Hint – see the following Project Report:
http://hdl.handle.net/10119/18917

6. Experiment how much time it will take to complete the model checking that TAS and Qlock satisfy the mutex property as the number of threads increases.

Exercises

7. Investigate the state explosion problem in model checking.

8. Investigate some techniques that can mitigate the state explosion in model checking.

9. Investigate narrowing in which unification is used instead of pattern matching.

Exercises

10. The search predicate can be used to model check that mutex protocols, such as TAS, satisfy the mutex property when a small fixed number of threads participate in the protocols, but cannot be used to do so when an arbitrary number of threads do so. Narrowing can solve the problem. Investigate some techniques based on narrowing that can be used to model check that mutex protocols satisfy the mutex property when an arbitrary number of threads participate in the protocols. Note that Maude, a sibling language of CafeOBJ, supports narrowing and gives some commends based on narrowing.

Exercises

11. Investigate linear temporal logic (LTL), Kripke structures (an extension of state machines), the semantics of LTL based on Kripke structures, the Maude LTL model checker, and conduct case studies in which TAS and Qlock enjoy some liveness properties, say, the lockout freedom property that whenever a process wants to enter the critical section, it will eventually be in the critical section.

12. Investigate MCS mutex protocol and Suzuki-Kasami distributed mutex protocol, and conduct case studies that the protocols enjoy both the mutex property and the lockout freedom property.