Some mutual exclusion protocols

Duong Dinh Tran

April 24, 2024

1 TAS

The TAS mutual exclusion protocol has the pseudo-code as follows:

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

test&set(*locked*) atomically does the following: if *locked* is false, then it sets *locked* to true and returns false; otherwise, it just returns true. (Since the protocol uses test&set, it is called TAS.) Each process is located at either rs (Remainder Section) or cs (Critical Section) and initially at rs. *locked* is a Boolean variable shared by all processes and initially false.

CafeOBJ formal specification

In the specification of TAS in CafeOBJ, we use sorts `Sys`, `Label`, and `Pid` to represent the state space, the labels (i.e., rs and cs), and process IDs, respectively:

```
[Sys]     -- state space
[Label]   -- labels, i.e., rs and cs
[Pid]     -- process IDs
```

We introduce the two so-called *observation functions* to observe the location of each process and the value of *locked* in each given state:

```
op pc : Sys Pid -> Label
op locked : Sys -> Bool
```

Accordingly, `pc(S,P)` returns the location (label) where process `P` is located at state `S`. Similarly, `locked(S)` denotes the lock at state `S`.

We introduce constant `init` to denote any initial state, which satisfies the following equations:

```
-- Initial states
op init  :         -> Sys {constr} .
eq pc(init,P) = rs .
eq locked(init) = false .
```

To specify the protocol execution, we define two *transitions* `enter` and `exit`, modeling processes move to cs from rs and move back to rs from cs. With the former, we first define its *effective condition* `c-enter` (i.e., the condition triggering the transition).

```
-- moves to cs from rs
op enter   : Sys Pid -> Sys {constr} .
op c-enter : Sys Pid -> Bool .
eq c-enter(S,P) = (pc(S,P) = rs and locked(S) = false) .
ceq pc(enter(S,P),Q) = (if P = Q then cs else pc(S,Q) fi)
  if c-enter(S,P) .
ceq locked(enter(S,P)) = true if c-enter(S,P) .
ceq enter(S,P) = S if not c-enter(S,P) .
```

Note that `S` is a variable of `Sys`, while `P` and `Q` are variables of `Pid`. The effective condition requires that the process `P` is located at rs and the lock is false in state `S`. If it is met, then process `P` moves to cs in the successor state; otherwise, the state does not change.

Mutex property verification

One desired property TAS should enjoy is the mutual exclusion property whose informal description is that there is always at most one process located at the Critical Section. The property is specified by the following predicate `mutex`:

```
op mutex : Sys Prin Prin -> Bool
eq mutex(S,P,Q) = (pc(S,P) = cs and pc(S,Q) = cs) implies P = Q .
```

The invariant is proved by (simultaneous) structural induction on `S`. There are one base case and two induction cases. The following is the proof of the base case written in CafeOBJ:

```
open TAS .
  ops p q : -> Pid .
  red mutex(init,p,q) .   --> true
close
```

where `TAS` is the CafeOBJ module in which the specification and the predicate `mutex` are available, **`open`** makes the given module (or specification) available, **`close`** stops the use of the module, and **`red`** (abbreviation of `reduce`) reduces the given term. `p` & `q` are fresh constants of sort `Pid` representing arbitrary process IDs (which may or may not be equal to each other). CafeOBJ returns `true`, meaning that the case is discharged.

Let us consider the induction case associated with transition `enter`. When the most typical induction hypothesis is used, the proof is as follows:

```
open TAS .
  op s : -> Sys .  ops p q r : -> Pid .
  red mutex(s,p,q) implies mutex(enter(s,r),p,q) .
close
```

However, feeding this proof fragment into CafeOBJ, the returned result is neither `true` nor `false`, but instead a complicated term as follows:

```
(((pc(enter(s,r),q) = cs) and ((p = q) and (pc(enter(s,r),p) = cs))) xor
(((pc(enter(s,r),p) = cs) and ... )))
```

where ... stands for terms that are omitted. The term cannot be reduced because the module lacks information on the processes `r`, `p`, and `q`. Case splitting is used to overcome this situation. The case is first split into two sub-cases: (`enter-1`) `pc(s,r) = rs` and (`enter-2`) `(pc(s,r) = rs) = false`. The proof fragment (`enter-2`) for the latter is as follows:

```
open TAS .
  op s : -> Sys .  ops p q r : -> Pid .
  eq (pc(s,r) = rs) = false .
  red mutex(s,p,q) implies mutex(enter(s,r),p,q) .  --> true
close
```

CafeOBJ returns `true`, indicating that the sub-case is discharged. For the sub-case (`enter-1`), it is necessary to conduct case splitting several more times. Let us consider a sub-case of (`enter-1`), which has the following proof fragment:

```
open TAS .
  op s : -> Sys .        ops p q r : -> Pid .
  eq pc(s,r) = rs .      eq locked(s) = false .  eq p = r .
  eq (q = r) = false .   eq pc(s,q) = cs .
  red mutex(s,p,q) implies mutex(enter(s,r),p,q) .  --> false
close
```

CafeOBJ returns `false` for this fragment. We need to conjecture a lemma to discharge the sub-case. The lemma is as follows:

```
-- for any state S and process P,
--  if P is located at Critical Section, locked must be true
eq inv1(S,P) = (pc(S,P) = cs implies locked(S)) .
```

Then, in the open-close fragment above, `inv1` is used as a lemma to discharge the sub-case of (`enter-1`) as follows:

```
red inv1(s,q) implies mutex(s,p,q) implies mutex(enter(s,r),p,q) .
```

CafeOBJ now returns `true` for the proof fragment. The remaining proof is written likewise.

loop { "Remainder Section"
rs : enqueue($queue, i$);
ws : **repeat until** top($queue$) $= i$;
"Critical Section"
cs : dequeue($queue$); }

Figure 1: Qlock protocol

2 Qlock

The Qlock protocol has the pseudo-code as in Figure 1. In the figure, rs, ws, and cs stand for Remainder Section, Waiting Section, and Critical Section, respectively. *queue* is an atomic queue of process identifiers (IDs) shared by all processes. Initially, *queue* is empty and each process i is located at rs. If i wants to enter cs, it first enqueues its ID i into *queue* and moves to ws. While the top of *queue* is not i, it needs to wait there. When i leaves cs, it dequeues *queue* and goes back to rs.

3 Anderson

We suppose that there are N processes participating in Anderson protocol. The pseudo-code of Anderson protocol for each process i can be written as follows:

Loop "Remainder Section"
rs : $place[i]$:= fetch&incmod($next, N$);
ws : **repeat until** $array[place[i]]$;
"Critical Section"
cs : $array[place[i]]$,
$array[(place[i] + 1) \,\%\, \mathrm{N}]$:= false, true;

We suppose that each process is located at rs, ws or cs and initially located at rs. *place* is an array whose size is N and each of whose elements stores one from $\{0, 1, \ldots, N-1\}$. Initially, each element of *place* can be any from $\{0, 1, \ldots, N-1\}$ but is 0 in this paper. Although *place* is an array, each process i only uses $place[i]$ and then we can regard $place[i]$ as a local variable to each process i. *array* is a Boolean array whose size is N. Initially, $array[0]$ is true and $array[j]$ is false for any $j \in \{1, \ldots, N-1\}$. *next* is a natural number variable and initially set to 0. fetch&incmod($next, N$) atomically does the following: setting *next* to $(next + 1) \,\%\, N$ and returning the old value of *next*. $x, y := e_1, e_2$ is a concurrent assignment that is processed as

follows: calculating e_1 and e_2 independently and setting x and y to their values, respectively.

An abstract version of Anderson

Since initially we found trouble in verifying the original Anderson protocol, we made an abstract version of it, called A-Anderson, and verify this abstract version. We also suppose that there are N processes participating in an abstract version of Anderson protocol. The pseudo-code of A-Anderson protocol for each process i can be written as follows:

Loop "Remainder Section"
  rs : $place[i]$:= fetch&inc($next$);
  ws : **repeat until** $array[place[i]]$;
    "Critical Section"
  cs : $array[place[i]+1]$:= true;

We use an infinite Boolean array *array* instead of a finite one and do not use %. fetch&inc is used instead of fetch&incmod. fetch&inc($next$) atomically does the following: setting $next$ to $next+1$ and returning the old value of $next$. We also suppose that each process is located at rs, ws or cs and initially located at rs. Initially, each element of *place* can be any natural number but is 0 in this paper, $array[0]$ is true, $array[j]$ is false for any non-zero natural number j and $next$ is 0.

4 MCS

MCS is a mutual exclusion protocol invented by Mellor-Crummey and Scott. Variants of MCS have been used in Java VMs and therefore the 2006 Edsger W. Dijkstra Prize in Distributed Computing went to their paper. The algorithm inside the MCS protocol is a scalable algorithm for spin locks that generates $O(1)$ remote references per lock acquisition, independent of the number of processes attempting to acquire the lock. Figure 2 shows the pseudo-code of the protocol for each process p. MCS uses one global variable $glock$ and three local variables $next_p$, $prede_p$ and $lock_p$ for each process p. Process IDs are stored in $glock$, $next_p$, and $prede_p$, while a Boolean value is stored in $lock_p$. There is one special (dummy) process ID, i.e., nop, that is different from any real process IDs. Initially, each of $glock$, $next_p$ and $prede_p$ is set to nop and $lock_p$ is set to false. We suppose that each process is located at one of the labels, such as rs, l1, and cs. Initially, each process is located at rs. When a process wants to enter "Critical Section," it first moves to l1 from rs.

MCS uses two non-trivial atomic instructions: fetch&store and comp&swap. For a variable x and a value a, fetch&store(x, a) atomically does the following: x is set to a and the old value of x is returned. For a variable x and

```
rs  : "Remainder Section"
l1  : next_p := nop;
l2  : prede_p := fetch&store(glock, p);
l3  : if prede_p ≠ nop {
l4  :    lock_p := true;
l5  :    next_prede_p := p;
l6  :    repeat while lock_p; }
cs  : "Critical Section"
l7  : if next_p = nop {
l8  :    if comp&swap(glock, p, nop)
l9  :       goto rs;
l10 :    repeat while next_p = nop; }
l11 : lock_next_p := false;
l12 : goto rs;
```

Figure 2: MCS protocol

values a, b, comp&swap(x, a, b) atomically does the following: if x equals a, then x is set to b and true is returned; otherwise, false is just returned.

Figure 3 graphically visualizes the change of state of MCS when a process p moves to l3 from l2. In the state υ, which is represented by Figure 3 (a), processes p, q, and r, located at l2, l5, and cs, respectively; *glock* is q; *next* of r is q; and *prede* of q is r. When process p moves to $l3$, *glock* is set to itself, and its *prede* is set to q (Figure 3 (b)).

5 Suzuki-Kasami

Suzuki-Kasami is a distributed mutual exclusion protocol. The name Suzuki-Kasami came from its authors' names, namely Ichiro Suzuki and Tadao Kasami. The protocol is designed to work over the network with the participation of multiple nodes. The key idea of the protocol is a shared privilege, in which a node cannot enter the critical section unless it owns the privilege, and the privilege can be transferred between nodes in the network. Suppose that there are N nodes participating in the protocol, where $1, \ldots, N$ are used as their identifiers. The pseudo-code of the protocol for each node i is shown in Figure 4. A node i can send a request message, which is in the form of request(i, n), to another node to request for the privilege, where n is a natural number that identifies the request number. A node can send a privilege message, which is in the form of privilege(q, a), to another node to

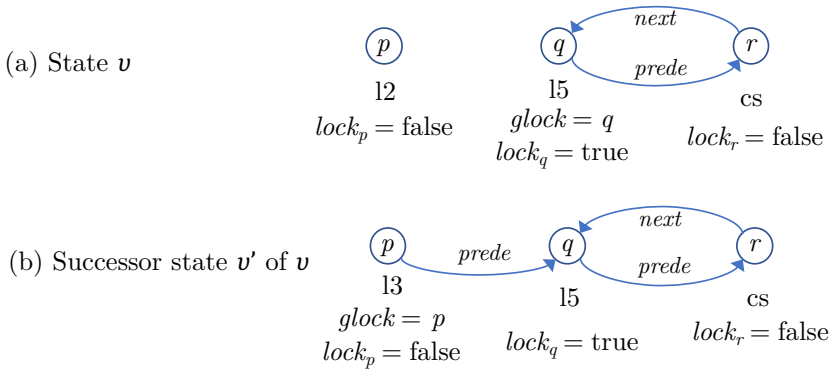

Figure 3: The change of state of MCS when a process p moves to l3 from l2

transfer the privilege after it exits the critical section, where q is a queue of node IDs and a is an N-array of natural numbers.

Each node i maintains the following local variables:

- *requesting*: a Boolean variable, which is true if the node wants to enter the critical section; otherwise, it is false.
- *have_privilege*: a Boolean variable, which is true if the node currently owns the privilege; otherwise, it is false.
- *queue*: a queue of node IDs that are requesting to enter the critical section.
- ln: an N-array of natural numbers, where $ln[j]$ is the request number of the request of node j granted most recently.
- rn: an N-array of natural numbers recording the largest request number ever received from each other node.

Figure 4 consists of two procedures, namely P1 and P2. The former is invoked when a node i attempts to enter the critical section. First, *requesting* is set to true. If i owns the privilege, it directly moves to the critical section. Otherwise, it increments $rn[i]$ and sends the request message, i.e., request$(i, rn[i])$, to all other nodes in the network. Then, i waits for the privilege. Once the privilege is received, it updates its queue and ln by the ones received in that privilege message, sets *have_privilege* to true, and moves to the critical section. When i leaves the critical section, it updates $ln[i]$ by $rn[i]$. After that, i checks for each node j if j is waiting to enter the critical section ($rn[j] = ln[j] + 1$) and j is not in the queue maintained by i ($j \notin queue$). If that is the case, j is put into the queue. After that, if the queue is empty, i sets *requesting* to false and keeps the privilege. Otherwise,

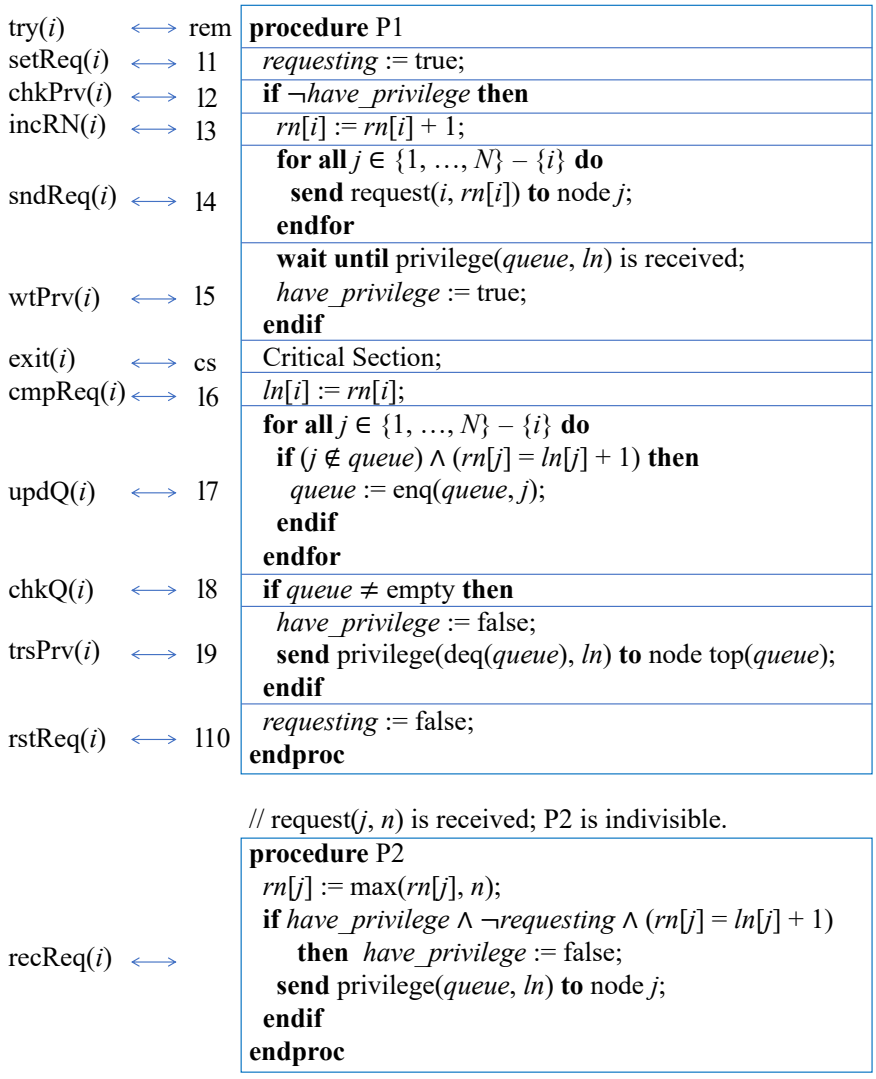

Figure 4: Suzuki-Kasami protocol

have_privilege is set to false and i transfers the privilege to the node at the top of the queue by sending the message privilege(deq($queue$), ln) to it.

Procedure P2 is invoked when node i receives a request message in the form of request(j, n) from node j. Note that the procedure is atomically executed.