university of groningen

Mechanical verification of a mutual exclusion algorithm with nonatomic variables

Wim H. Hesselink

Johann Bernoulli Institute for Mathematics and Computer Science
University of Groningen
The Netherlands

IPA, de Koningshof, 1 November 2010

Sources

Wim H. Hesselink
Email: w.h.hesselink@rug.nl
www.cs.rug.nl/~wim

L. Lamport: A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17: 453–455, 1974.

E.A. Lycklama and V. Hadzilacos: A first-come-first-served mutual-exclusion algorithm with small communication variables.
ACM TOPLAS 13: 558–576, 1991.

A.A. Aravind and W.H. Hesselink: A nonatomic dual bakery algorithm with bounded tokens. Submitted.

For the proof assistant PVS see: http://pvs.csl.sri.com

Overview

- Mutual exclusion
- Nonatomic variables (safeness)
- Modelling MX and FCFS
- Lamport's Bakery Algorithm
- Proof obligations
- PVS modelling
- Invariants (modelled in PVS)
- No deadlock, and FCFS
- Use of proof assistants
- Questions?

Mutual exclusion

I think, this is well known.

A number (N) of threads or processes (or processors)

need to communicate by shared variables

to obtain exclusive access to some shared resource.

Basic method: (busy) waiting for condition C to hold:

$$\textbf{await } C \quad \equiv \quad \textbf{while } \neg\, C \textbf{ do } \textit{skip} \textbf{ od} \ .$$

Nonatomic variables

A shared variable `x` is called atomic
iff the variable behaves logically as if
readings and writings are always interleaved
in some order that refines the temporal order.

A shared variable `x` is called safe (nonatomic)

iff, under the assumption that
it is never written by more than one thread,

any thread reading while no thread is writing
obtains the latest written value

and any thread reading while some thread is writing
obtains an arbitrary value.

Modelling safeness

Reading a safe variable $\mathtt{x}$ into a private variable v
can be regarded as atomic (it never interferes).

Written $v := \mathtt{x}$.

Writing a private expression E into a safe shared variable $\mathtt{x}$:

$$\ell : \ \mathtt{x} := (\text{flickering})\ E\ .$$

is modelled as a nondeterministic choice:

$$\begin{array}{l} \ell : \ (\ \mathtt{x} := \textit{arbitrary}\ ;\ \textbf{goto}\ \ell \\ \qquad [\!]\ \mathtt{x} := E\ ) \end{array}$$

Therefore, repeated attempts to write E that eventually succeed.

Modelling MX and FCFS

thread $(p : 0 \leq p < N)$:
  while *true* **do**
    NCS ;
    Doorway ;
    Waiting ;
    CS ;
    Exit ;
  od.

NCS and *CS* are given.

To design *Doorway*, *Waiting*, and *Exit* such that

MX: never more than 1 thread in *CS*.

Doorway and *Exit* are waitfree.

FCFS: if p has passed *Doorway* when q enters *Doorway*, then p enters *CS* before q.

Lamport's Bakery

When entering the bakery, you get a number higher than the maximum number present.

When your number is the lowest number present, it is your turn.

When you leave, reset your number.

Safe shared variables

bool $\texttt{act}[N] := \{\mathit{false}, \ldots, \mathit{false}\}$;

int $\texttt{num}[N] := \{0, \ldots, 0\}$.

Thread p only writes $\texttt{act}[p]$ en $\texttt{num}[p]$.

Lamport's Bakery Algorithm

```
thread (p) {
    int j, level ;
    while (true) {
        NCS ;
        act[p] := (flickering) true ; level := 0 ;
        for (j := 0 ; j < N ; j++) level := max(level, num[j]) ;
        num[p] := (flickering) level + 1 ;
        act[p] := (flickering) false ;
        for (j := 0 ; j < N ; j++) {
            await ¬ act[j] ;
            await num[j] ≤ 0 ∨ num[p] · N + p ≤ num[j] · N + j ;
        }
        CS ;
        num[p] := (flickering) 0 ;
    }
}
```

Bakery, the rough argument

In general, the thread q with the lowest value $\texttt{num}[q]$
gets priority in second loop.

When threads obtain the same numbers in the first loop,
priority is determined by their thread identifiers.

In the second loop, thread p first waits for thread $j.p$ to complete its *Doorway*
before comparing the numbers.

Bakery, remarks

The order of traversing the two loops does not matter:

We use private variables *set1*, *set2*, and *set3*
to hold the sets of thread numbers
that have yet to be treated in the loops.

The numbers $\mathtt{num}[p]$ are unbounded integers
and can get arbitrarly large.

In second loop, Lamport has the first conjunct $\mathtt{num}[p] = 0$.

He uses the lexical order on pairs $(\mathtt{num}[p], p)$.

Proof obligations

Mutual exclusion: $(MX)\quad q$ **at** $CS \;\wedge\; r$ **at** $CS \;\Rightarrow\; q = r$.

No deadlock

FCFS: first-come-first-served

FCFS: use a shared ghost variable `predec`;
`predec`$[q]$ holds the threads that thread q must gives priority to.

FCFS: $\quad q$ **at** $CS \;\Rightarrow\;$ `predec`$[q] = \emptyset$.

At 11: $\quad$ `predec`$[p] := \{q \mid q$ **in** $[15 \ldots 17]\}$.

At 17: $\quad \langle$ **for all** q **do remove** p **from** `predec`$[q]$ **od** $\rangle$.

Code extended

thread(p) :

10: $\quad$ *NCS* ; $\texttt{predec}[p] := \{q \mid q$ **in** $[15 \ldots 17]\}$;

11: $\quad$ $\texttt{act}[p] :=$ (flickering) *true* ; *level* $:= 0$; *set1* $:=$ *Thread* $\setminus \{p\}$;

12: $\quad$ **while** *nonempty*(*set1*) **do**

$\quad\quad$ **extract some** j **from** *set1* ;

$\quad\quad$ *level* $:= \max($*level*$, \texttt{num}[j])$ **od** ;

13: $\quad$ $\texttt{num}[p] :=$ (flickering) *level* $+ 1$;

14: $\quad$ $\texttt{act}[p] :=$ (flickering) *false* ; *set2* $:=$ *set3* $:=$ *Thread* $\setminus \{p\}$;

15: $\quad$ **while** *nonempty*(*set3*) **do**

$\quad\quad$ **choose some** $j \in$ *set3* ;

16: $\quad\quad$ **if** $j \in$ *set2* **then await** $\neg\, \texttt{act}[j]$; **remove** j **from** *set2*

$\quad\quad$ **else await** $\texttt{num}[j] \leq 0 \;\vee\; \texttt{num}[p] \cdot N + p \leq \texttt{num}[j] \cdot N + j$;

$\quad\quad\quad\quad$ **remove** j **from** *set3*

$\quad\quad$ **fi**

$\quad$ **od** ;

17: $\quad$ *CS* ; $\langle$ **for all** q **do remove** p **from** $\texttt{predec}[q]$ **od** $\rangle$;

18: $\quad$ $\texttt{num}[p] :=$ (flickering) 0 ; *cnt*++ ; **goto** 10 .

One access of a shared variable per transition

Moreover

Initialization:

$$\forall\, q : q \textbf{ at } 10 \;\wedge\; \neg\, \texttt{act}[q] \;\wedge\; \texttt{num}[q] = 0 \;\wedge\; \texttt{predec}[q] = \emptyset \;.$$

Fault tolerance.
Thread p can do at any time:

$$\begin{array}{l} \langle\; \texttt{act}[p] := \mathit{false} \;; \\ \quad \textbf{for all } q \textbf{ do remove } p \textbf{ from } \texttt{predec}[q] \textbf{ od} \;\; ; \\ \quad \textbf{goto } 18 \;\; \rangle \;. \end{array}$$

PVS modelling

```
state: TYPE = [#
  num: [Process -> int],
  act: pred[Process],
  predec: [Process -> pred[Process]],
  jj: [Process -> Process],
  set1, set2, set3: [Process -> finite_set[Process]],
  pc, lev, cnt: [Process -> nat]
#]

step12B(p, x, y): bool =
  x`pc(p) = 12 AND EXISTS q: x`set1(p)(q) AND
  y = x WITH [
       `set1(p) := remove(q, x`set1(p)) ,
       `lev(p) := max(x`lev(p), x`num(q))
      ]
```

PVS modelling continued

The total step relation becomes the union of about 14 step relations:
9 for the main steps,
2 for flickering of `act` and `num`,
2 jumps to the end of the loop,
and 1 for the fault step.

Construct an initial condition.

Invent and verify invariants.

Main invariants for MX

Iq0: q **in** $[15 \ldots 17] \ \wedge \ r$ **in** $[14 \ldots 17]$
$\Rightarrow \quad r \in \mathit{set3}.q \ \vee \ \mathtt{num}[q] \cdot N + q \leq \mathtt{num}[r] \cdot N + r$

Iq1: q **in** $[17 \ldots] \ \Rightarrow \ \mathit{set3}.q = \emptyset$

From *Iq0* and *Iq1* for q, r, we get

$$q \ \textbf{at} \ 17 \ \wedge \ r \ \textbf{at} \ 17 \ \Rightarrow \ \mathtt{num}[q] \cdot N + q = \mathtt{num}[r] \cdot N + r \ .$$

This implies *MX* because $0 \leq q, r < N$.

Invariance van *Iq1* is easy.

Invariance of *Iq0* requires some other invariants.

Indeed, somehow, the Booleans `act` must play a role.

Invariants in PVS

```
iq0(q, r, x): bool =
  15 <= x`pc(q) AND x`pc(q) <= 17
  AND 14 <= x`pc(r) AND x`pc(r) <= 17
  IMPLIES x`set3(q)(r) OR x`num(q)*N + q <= x`num(r)*N + r
```

Iq2: q **in** $[15 \ldots 17] \ \wedge \ r$ **in** $[12 \ldots 13]$
$\Rightarrow \quad r \in \mathit{set3}.q \ \vee \ q \in \mathit{set1}.r \ \vee \ \mathtt{num}[q] \leq \mathit{level}.r$

Iq3: q **in** $[13 \ldots] \ \Rightarrow \ \mathit{set1}.q = \emptyset$

```
iq0_step_13: LEMMA
  iq0(q, r, x) AND step13(p, x, y)
  AND iq2(q, r, x) AND iq3(r, x)
  IMPLIES iq0(q, r, y)
```

Preservation of $Iq0$ continued

Preservation of *Iq0* at line 16 follows from the obvious invariants

Iq4: q **in** $[14 \ldots 17] \;\Rightarrow\; \mathtt{num}[q] > 0$

Iq5: q **in** $[\ldots 12] \;\Rightarrow\; \mathtt{num}[q] = 0$

Nothing is said about $\mathtt{num}[q]$ at 13 or 18, because of flickering.

Preservation of $Iq2$

Iq2: q **in** $[15 \ldots 17] \ \wedge \ r$ **in** $[12 \ldots 13]$
$\Rightarrow \quad r \in \mathit{set3}.q \ \vee \ q \in \mathit{set1}.r \ \vee \ \mathtt{num}[q] \leq \mathit{level}.r$

```
iq2_step_rest: LEMMA
  iq2(q, r, x) AND step(p, x, y)
  IMPLIES iq2(q, r, y) OR step16(p, x, y)
```

Preservation of *Iq2* at line 16 (removal of j from *set3*) follows from:

Iq6: q **in** $[15 \ldots 16] \ \wedge \ r$ **in** $[12 \ldots 13]$
$\Rightarrow \quad r \in \mathit{set2}.q \ \vee \ q \in \mathit{set1}.r \ \vee \ \mathtt{num}[q] \leq \mathit{level}.r$.

Preservation of *Iq6* at line 16 follows from the obvious invariant

Iq7: q **in** $[12 \ldots 13] \ \Rightarrow \ \mathtt{act}[q]$.

No deadlock

Here, we also need the easy invariants

Iq8: q **in** $[10 \ldots 18]$,
Iq9: $\texttt{act}[q] \Rightarrow q$ **in** $[11 \ldots 14]$.

Then define `iqall(x)` to be the conjunction of the universal quantification of the predicates *Iq0* up to *Iq9*.

```
no_deadlock: LEMMA
  iqall(x) IMPLIES (FORALL p: x`pc(p) = 10)
  OR EXISTS p, y: forwardstep(p, x, y)
```

The proof required much work, because I had to instantiate the new state `y` for all different cases.

The argument is also nontrivial.

FCFS

FCFS: q **at** $17 \;\Rightarrow\; \texttt{predec}[q] = \emptyset$

follows logically from *Iq1* (q **at** $17 \;\Rightarrow\; \mathit{set3}.q = \emptyset$)
and the new invariant

Kq0: q **in** $[15 \ldots 17] \;\Rightarrow\; \texttt{predec}[q] \subseteq \mathit{set3}.q$.

Preservation of *Kq0* at 14 and 16 from

Kq1: $q \notin \texttt{predec}[q]$,
Kq2: $r \in \texttt{predec}[q] \;\Rightarrow\; r$ **in** $[15 \ldots 17]$,
Kq3: q **in** $[14 \ldots 17] \;\wedge\; r \in \texttt{predec}[q] \;\Rightarrow\; \texttt{num}[r] < \texttt{num}[q]$.

Preservation of *Kq3* at 13 and 18 from *Iq4*, *Kq2*, and

Kq4: q **in** $[12 \ldots 13] \;\wedge\; r \in \texttt{predec}[q] \;\Rightarrow\; r \in \mathit{set1}.q \;\vee\; \texttt{num}[r] \leq \mathit{level}.q$.

Preservation of *Kq4* at the flickering lines 13 and 18 uses *Kq2*.

Progress

Progress is more than absence of deadlock.

E.g. the loops need to terminate.

Every thread q gets a private ghost variable $\mathit{cnt}.q$,
incremented with 1 in line 18.

$$\begin{aligned}\mathit{vf}.q = {} & A \cdot \mathit{cnt}.q + \mathit{pc}.q \\ & + (q \textbf{ in } [12\ldots] \,?\, N - \#\mathit{set1}.q : 0) \\ & + (q \textbf{ in } [15\ldots] \,?\, 2 \cdot (2 \cdot N - \#\mathit{set2}.q - \#\mathit{set3}.q) : 0)\ .\end{aligned}$$

If $A \geq 5 \cdot N + 9$, then $\mathit{vf}.q$ never decreases,
and it increases with every forward step of q.

Moreover $A \cdot \mathit{cnt}.q \leq \mathit{vf}.q \leq A \cdot \mathit{cnt}.q + 5 \cdot N + 18$.

Therefore, if q does ∞ forward steps, then ∞ *CS* or ∞ Faults.

Liveness

Theorem 1. *Under assumption of weak fairness and boundedly many faults, every thread is always eventually at NCS.*

This is expressed by the temporal formula:

$$\mathit{Ex} \cap \mathit{WFa} \cap \mathit{BFa} \subseteq \Box\Diamond[\![\, q \textbf{ at } 10 \,]\!] \; .$$

```
liveness: THEOREM
   execution(xs) AND weakly_fair(xs) AND jumping
   AND bounded_faults(xs)
   IMPLIES box(diamond(sem1(idle(q))))(xs)
```

`jumping` : $A \geq 5 \cdot N + 9$

Available proof assistants

NQTHM and ACL2 (the Boyer-Moore provers)

PVS

HOL

Isabelle

Coq

etc. (Ask Freek Wiedijk, RU)

Interactive use of a proof assistant

To verify validity of all proof steps

It can do some very easy proofs by itself

No intelligence: it gives no direction for the proof

It can be used to derive proof obligations

and for proof administration

PVS gives the current proof tree

Use requires insight in the logic of your problem

Questions