

```

/* consensus protocol [AH90] */
/* gxn 20/11/00 */
/* randomization replaced with non-deterministic choice */
/* this file contains the agreement proof (no two processes decide on different values) */
/* note used values 1 and 2 not values 0,1 (use 0 to model bottom) */
/* THE FILE CONTAINS THE PROOF OF INVARIANT 6.3 */

/*-----*/

/* CONSTANTS */
/* number of processes */
#define N 10
/* set of processes as ordset to use induction */
ordset PROC 1..N;
/* round numbers as ordset to use induction */
ordset NUM 0..;
/* local phases */
typedef PC {INITIAL, READ1, CHECK1, READ2, CHECK2, DECIDE, NIL};

/*-----*/

module main(act){

/*-----INPUTS-----*/

/* scheduler */
act : PROC;
/* initial values of processes */
start : array PROC of 1..2;

/*-----THE PROTOCOL-----*/

/* LOCAL VARIABLES */
/* phase */
pc : array PROC of PC;
/* values[i][j] choice of j when last read by i */
values : array PROC of array PROC of 0..2;
/* rounds[i][j] round number of j when last read by i */
rounds : array PROC of array PROC of NUM;
/* counter used for loop when reading */
count : array PROC of PROC;

/* GLOBAL VARIABLES (MEMORY) */
/* value[i] current choice of i */
value : array PROC of 0..2;
/* round[i] current round number of i */
round : array PROC of NUM;

/* INITIAL VALUES */
forall (i in PROC) {
  init(pc[i]) := INITIAL;
  forall (j in PROC) {
    init(rounds[i][j]) := 0;
    init(values[i][j]) := 0;
  }
  init(round[i]) := 0;
  init(value[i]) := 0;
  init(count[i]) := 1;
}
}

```

```

/* NEXT VALUES (based on the phase of the process) */
/* note only the process being scheduled (process act) moves */
switch (pc[act]) {
  INITIAL : {
    next(value[act]) := start[act];
    next(round[act]) := round[act]+1;
    next(pc[act]) := READ1;
  }
  READ1 : {
    next(pc[act]) := (count[act]=N) ? CHECK1 : READ1;
    next(rounds[act][count[act]]) := round[count[act]];
    next(values[act][count[act]]) := value[count[act]];
    next(count[act]) := count[act]=N ? count[act] : count[act]+1;
  }
  CHECK1 : {
    if (decide[act]) {
      /* all who disagree trail by two and I am a leader */
      next(pc[act]) := DECIDE;
    }
    else if (agree[act][1]) {
      /* all leaders agree on 1 */
      next(pc[act]) := READ1;
      next(count[act]) := 1;
      next(value[act]) := 1;
      next(round[act]) := round[act]+1;
    }
    else if (agree[act][2]) {
      /* all leaders agree on 2 */
      next(pc[act]) := READ1;
      next(count[act]) := 1;
      next(value[act]) := 2;
      next(round[act]) := round[act]+1;
    }
    else {
      next(pc[act]) := READ2;
      next(count[act]) := 1;
      next(value[act]) := 0;
    }
  }
  READ2 : {
    next(pc[act]) := count[act]=N ? CHECK2 : READ2;
    next(rounds[act][count[act]]) := round[count[act]];
    next(values[act][count[act]]) := value[count[act]];
    next(count[act]) := count[act]=N ? count[act] : count[act]+1;
  }
  CHECK2 : {
    if (agree[act][1]) {
      /* all leaders agree on 1 */
      next(pc[act]) := READ1;
      next(count[act]) := 1;
      next(value[act]) := 1;
      next(round[act]) := round[act]+1;
    }
    else if (agree[act][2]) {
      /* all leaders agree on 2 */
      next(pc[act]) := READ1;
      next(count[act]) := 1;
      next(value[act]) := 2;
      next(round[act]) := round[act]+1;
    }
  }
}

```

```

    }
    else {
        /* guess new value */
        next(pc[act]) := READ1;
        next(count[act]) := 1;
        next(value[act]) := {1,2};
        next(round[act]) := round[act]+1;
    }
}
DECIDE : {
    next(pc[act]) := NIL;
}
};

/*-----END OF MAIN PROTOCOL-----*/

/*-----FORMULAE WE NEED FOR CHECKING PHASES----- */

/* decide[i] true if according to i all that disagree trail by 2 and i is a leader */
decide : array PROC of boolean;
/* array_agree[i][v][j] true if i has read j implies according to i if j is a leader then j agrees on v */
array_agree : array PROC of array 1..2 of array PROC of boolean;
/* agree[i][v] true if according to i all leaders read by process i agree on v */
agree : array PROC of array 1..2 of boolean;
/* array_minus1_agree[i][v][j] true if i has read j then rounds[i][j] ≥ fill_maxr[i]-1 → values[i][j]=1 */
array_minus1_agree : array PROC of array 1..2 of array PROC of boolean;
/* minus1_agree[i][v][j] true if according to i all process with round ≥ fill_maxr[i]-1 read by process i agree on v */
minus1_agree : array PROC of array 1..2 of boolean;

/*Note that inv5 and inv7 (proved in invariants.smv) allow us to use fill_maxr in the definition of array_agree etc */

/* INITIAL VALUES */
forall (i in PROC) {
    init(decide[i]) := 0;
    for(v = 1; v ≤ 2; v = v + 1) forall (j in PROC) {
        init(array_agree[i][v][j]) := 1;
        init(array_minus1_agree[i][v][j]) := 1;
    }
}

forall (i in PROC) {
    next(decide[i]) := ( next(minus1_agree[i][1]) ∨ next(minus1_agree[i][2]) ) ∧ next(fill_maxr[i])=next(round[i]);
}

forall (i in PROC) {
    for(v = 1; v ≤ 2; v = v + 1) {
        forall (j in PROC) {
            if (next(pc[i])=INITIAL ∨ ((next(pc[i])=READ1 ∨ next(pc[i])=READ2) ∧ next(count[i])≤j)) {
                /* not read yet */
                next(array_agree[i][v][j]) := 1;
                next(array_minus1_agree[i][v][j]) := 1;
            }
            else {
                /* already read */
                next(array_agree[i][v][j]) := next(rounds[i][j])≥next(fill_maxr[i]) ⇒ next(values[i][j])=v;
                next(array_minus1_agree[i][v][j]) := next(rounds[i][j])≥next(fill_maxr[i])-1 ⇒ next(values[i][j])=v;
            }
        }
    }
}

```

```

}
/* conjunction of arrays */
forall (i in PROC) for(v = 1; v ≤ 2; v = v + 1) {
  agree[i][v] := ∧[ array_agree[i][v][j] : j in PROC ];
  minus1_agree[i][v] := ∧[ array_minus1_agree[i][v][j] : j in PROC ];
}

/*-----EXTRA PREDICATES NEEDED FOR AGREEMENT PROOF-----*/

/* global_maxr the maximum round */
global_maxr : NUM;
/* fill_maxr[i] round i thinks is the maximum round after fill */
fill_maxr : array PROC of NUM;

/* we use +1 for global_maxr and fill_maxr as opposed to next(history_round) as it simplifies proofs */
/* from inv1 and inv2 (proved in invariants.smv) global_maxr is correct */
/* from inv4, inv5, inv6 and inv7 (proved in invariants.smv) fill_maxr is correct */

/* INITIAL VALUES */
init(global_maxr) := 0;
forall (i in PROC) init(fill_maxr[i]) := 0;

/* NEXT VALUES */
next(global_maxr) := next(history_round) > global_maxr ? global_maxr + 1 : global_maxr;
forall (i in PROC) {
  if (next(pc[i])=INITIAL ∨ ((next(pc[i])=READ1 ∨ next(pc[i])=READ2) ∧ next(count[i]=1)))
    /* not read any processes (obs=0) so use global_maxr */
    next(fill_maxr[i]) := next(history_round) > global_maxr ? global_maxr + 1 : global_maxr;
  else if ((next(pc[i])=READ1 ∨ next(pc[i])=READ2) ∧ next(count[i] ≤ act) {
    /* not read process act but read some processes update fill_maxr */
    next(fill_maxr[i]) := next(history_round) > fill_maxr[i] ? fill_maxr[i] + 1 : fill_maxr[i];
  }
}
}
/* array_fill_agree[i][v] true if according to i process j agrees on v after fill */
array_fill_agree : array PROC of array 1..2 of array PROC of boolean;
/* fill_agree[i][v] true if according to i all leaders agree on v after fill */
fill_agree : array PROC of array 1..2 of boolean;

/* INITIAL VALUES */
forall (i in PROC) for(v = 1; v ≤ 2; v = v + 1) forall (j in PROC) {
  init(array_fill_agree[i][v][j]) := 0;
}

/* NEXT VALUES */
forall (i in PROC) {
  for(v = 1; v ≤ 2; v = v + 1) {
    forall (j in PROC) {
      if (next(pc[i])=INITIAL ∨ ((next(pc[i])=READ1 ∨ next(pc[i])=READ2) ∧ next(count[i] ≤ j))) {
        /* not read yet so use global values */
        next(array_fill_agree[i][v][j]) := next(round[j]) ≥ next(fill_maxr[i]) ⇒ next(value[j])=v;
      }
      else {
        /* already read */
        next(array_fill_agree[i][v][j]) := next(rounds[i][j]) ≥ next(fill_maxr[i]) ⇒ next(values[i][j])=v;
      }
    }
  }
}
}
/* conjunction of arrays */
forall (i in PROC) for(v = 1; v ≤ 2; v = v + 1) {

```

```

    fill_agree[i][v] :=  $\wedge$ [ array_fill_agree[i][v][j] : j in PROC ];
}

/*-----EXTRA PREDICATES NEEDED FOR PROBABILISTIC PROGRESS PROOF-----*/

/* array_fillr_agree[i][r][v][j] true if according to i after fill j has round greater than r implies value[i][j]=v */
array_fillr_agree : array PROC of array NUM of array 1..2 of array PROC of boolean;
/* fill_agree[i][r][v] true if according to i after fill all processes with round greater than r agree on v */
fillr_agree : array PROC of array NUM of array 1..2 of boolean;

/* INITIAL VALUES */
forall (i in PROC) forall (r in NUM) for(v = 1; v ≤ 2; v = v + 1) forall (j in PROC) {
    init(array_fillr_agree[i][r][v][j]) := 0;
}
/* NEXT VALUES */
forall (i in PROC) forall (r in NUM) for(v = 1; v ≤ 2; v = v + 1) forall (j in PROC) {
    if (next(pc[i])=INITIAL ∨ ((next(pc[i])=READ1 ∨ next(pc[i])=READ2) ∧ next(count[i]≤j)) {
        /* not read yet so use global values */
        next(array_fillr_agree[i][r][v][j]) := next(round[j])>r ⇒ next(value[j])=v;
    }
    else {
        /* already read */
        next(array_fillr_agree[i][r][v][j]) := next(rounds[i][j])>r ⇒ next(values[i][j])=v;
    }
}
forall (i in PROC) forall (r in NUM) for(v = 1; v ≤ 2; v = v + 1) {
    fillr_agree[i][r][v] :=  $\wedge$ [ array_fillr_agree[i][r][v][j] : j in PROC ];
}

/* to simplify proofs we need the condition of the invariant 7.6 as a single variable */
inv76 : array NUM of boolean;
forall (r in NUM) {
    inv76[r] :=  $\wedge$ [ (round[i]=r ⇒ ¬fill_agree[i][2]) : i in PROC ] ∧  $\wedge$ [ fillr_agree[i][r][1] : i in PROC ] ∧
     $\wedge$ [ (round[i]>r ⇒ value[i]=1) : i in PROC ];
}

/*-----HISTORY VARIABLES-----*/

/* records the current round of the process being scheduled */
history_round : NUM;
init(history_round) := 0;
next(history_round) := next(round[act]);

/* records the process with the global maximum round */
history_maxr : PROC;
init(history_maxr) := act;
next(history_maxr) := next(history_round)>global_maxr ? act : history_maxr;

/* records the process j with round[j] or rounds[i][j] equal to fill_maxr[i] */
history_fill_maxr : array PROC of PROC;
forall (i in PROC) {
    init(history_fill_maxr[i]) := act;
    if (next(pc[i])=INITIAL ∨ ((next(pc[i])=READ1 ∨ next(pc[i])=READ2) ∧ next(count[i]=1))
        /* not read any processes (obs=0) so use global_maxr */
        next(history_fill_maxr[i]) := next(history_round)>global_maxr ? act : history_maxr;
    else if ((next(pc[i])=READ1 ∨ next(pc[i])=READ2) ∧ next(count[i]≤act) {
        /* not read process act but read some processes update fill_maxr */
        next(history_fill_maxr[i]) := next(history_round)>fill_maxr[i] ? act : history_fill_maxr[i];
    }
}
}

```

```

/*-----THE PROOF-----*/

/* we split the proof of inv63 up into three parts */
/* these correspond to the clauses (a), (b) and (c) of invariant 6.3 in [PSL00] */

/* additional invariants */
/* invariants proved in invariants.smv */
forall (i in PROC) {
  inv1[i] : assert G ( round[i] ≤ global_maxr );
  assume inv1[i];
}
forall (i in PROC) forall (j in PROC) {
  inv4[i][j] : assert G ( (count[i] ≤ j ∧ (pc[i]=READ1 ∨ pc[i]=READ2)) ⇒ fill_maxr[i] ≥ round[j]);
  assume inv4[i][j];
}
forall (i in PROC) {
  inv10[i] : assert G ( (count[i]=1 ∧ (pc[i]=READ1 ∨ pc[i]=READ2)) ⇒ fill_maxr[i]=global_maxr );
  assume inv10[i];
}
forall (i in PROC) {
  inv8[i] : assert G ( round[i] ≤ fill_maxr[i] );
  assume inv8[i];
}
forall (i in PROC) {
  inv12[i] : assert G ( (pc[i]=CHECK1 ∨ pc[i]=CHECK2) ⇒ ¬( agree[i][1] ∧ agree[i][2] ) );
  assume inv12[i];
}
forall (i in PROC) for(v = 1; v ≤ 2; v = v + 1) forall (j in PROC) {
  inv30[i][v][j] : assert G ( ( minus1_agree[i][v] ∧ fill_agree[i][v] ∧ (fill_maxr[i]=round[i]) ∧ value[i]=v ∧ pc[j]=INITIAL ∧
    round[i]+1=round[j] ) ⇒ count[i] ≤ j );
  assume inv30[i][v][j];
}
forall (i in PROC) for(v = 1; v ≤ 2; v = v + 1) forall (j in PROC) forall (k in PROC) {
  inv31[i][v][j][k] : assert G ( (fill_agree[i][v] ∧ (fill_maxr[i]=round[i]) ∧ value[i]=v ∧ count[i] ≤ k ∧
    (pc[i]=INITIAL ∨ pc[i]=READ1 ∨ pc[i]=READ2)) ⇒ (rounds[j][k] ≥ round[i] ⇒ values[j][k]=v) );
  assume inv31[i][v][j][k];
}
/* these are proved in lemma6.12.smv */
forall (i in PROC) for(v = 1; v ≤ 2; v = v + 1) {
  lemma612a[i][v] : assert G ( ( ¬(act=i) ∧ X ( fill_agree[i][v] ∧ minus1_agree[i][v] ∧ fill_maxr[i]=round[i] ) )
    ⇒ ( fill_agree[i][v] ∧ minus1_agree[i][v] ∧ fill_maxr[i]=round[i] ) );
  assume lemma612a[i][v];
}
forall (i in PROC) for(v = 1; v ≤ 2; v = v + 1) {
  lemma612b[i][v] : assert G ( ( act=i ∧ (pc[i]=READ1 ∨ pc[i]=READ2) ∧
    X ( fill_agree[i][v] ∧ minus1_agree[i][v] ∧ fill_maxr[i]=round[i] ) )
    ⇒ ( fill_agree[i][v] ∧ minus1_agree[i][v] ∧ fill_maxr[i]=round[i] ) );
  assume lemma612b[i][v];
}
forall (i in PROC) for(v = 1; v ≤ 2; v = v + 1) {
  lemma612c[i][v] : assert G ( ( act=i ∧ X ( (pc[i]=DECIDE ∨ pc[i]=NIL) ∧ fill_agree[i][v] ∧ minus1_agree[i][v] ∧
    fill_maxr[i]=round[i] ) ) ⇒ ( fill_agree[i][v] ∧ minus1_agree[i][v] ∧ fill_maxr[i]=round[i] ) );
  assume lemma612c[i][v];
}

/* invariant 6.3(a) */
/* note we do not need to use inv63c to prove this */
/* one problem since have conjunction of an array */
/* therefore first prove for a single value in the array */

```

```

forall (i in PROC) for(v = 1; v ≤ 2; v = v + 1) forall (j in PROC) forall (k in PROC) {
  lemma63ak[i][v][j][k] : assert G ( (minus1_agree[i][v] ∧ fill_agree[i][v] ∧ fill_maxr[i]=round[i] ∧ value[i]=v)
    ⇒ ((count[j]>k ∨ pc[j]=CHECK1 ∨ pc[j]=CHECK2) ⇒ (rounds[j][k]≥round[i] ⇒ values[j][k]=v)) );
  forall (r in NUM) {
    subcase lemma63ak[i][v][j][k][r] of lemma63ak[i][v][j][k]
      for round[i]=r;
      using (lemma63ak[i][v][j][k]),
      (inv63b[i][v][k]),
      /* lemmas for lemma63a */
      inv12[i],
      inv12[k],
      inv31[i][v][j][k],
      inv30[i][v][j],
      /* other invariants */
      inv8[i],
      lemma612a[i][v],
      lemma612b[i][v],
      lemma612c[i][v],
      /* required abstractions */
      NUM⇒{r-1..r},
      /* free variables in cone */
      agree//free,
      array_fill_agree//free,
      array_fill_agree[i][v][k],
      array_minus1_agree//free,
      array_minus1_agree[i][v][k],
      count//free,
      count[i],
      count[j],
      decide//free,
      fill_agree//free,
      fill_agree[i][v],
      fill_maxr//free,
      fill_maxr[i],
      history_round//free,
      global_maxr//free,
      minus1_agree//free,
      minus1_agree[i][v],
      pc//free,
      pc[i],
      pc[j],
      round//free,
      round[i],
      round[k],
      rounds//free,
      rounds[i][k],
      rounds[j][k],
      start//free,
      value//free,
      value[i],
      value[k],
      values//free,
      values[i][k],
      values[j][k]
      prove lemma63ak[i][v][j][k][r];
    }
  }
}
/* we now prove invariant 6.3(a) by a contradiction */
/* introduce a witness */
y : array PROC of array 1..2 of PROC;

```

```

forall (j in PROC) for(v = 1; v ≤ 2; v = v + 1) {
  y[j][v] := { k : k in PROC, ¬array_agree[j][v][k] };
}
forall (i in PROC) for(v = 1; v ≤ 2; v = v + 1) forall (j in PROC) {
  inv63a[i][v][j] : assert G ( (minus1_agree[i][v] ∧ fill_agree[i][v] ∧ fill_maxr[i]=round[i] ∧ value[i]=v)
    ⇒ (((pc[j]=CHECK1 ∨ pc[j]=CHECK2)) ⇒ (fill_maxr[j]≥round[i] ⇒ agree[j][v])) );
  forall (k in PROC) forall (r in NUM) {
    subcase inv63a[i][v][j][k][r] of inv63a[i][v][j]
      for k=y[j][v] ∧ round[i]=r;
      using lemma63ak[i][v][j][k],
      agree//free,
      agree[j][v],
      array_fill_agree//free,
      array_minus1_agree//free,
      count//free,
      decide//free,
      fill_agree//free,
      fill_agree[i][v],
      fill_maxr//free,
      fill_maxr[i],
      global_maxr//free,
      minus1_agree//free,
      minus1_agree[i][v],
      round//free,
      round[i],
      rounds//free,
      start//free,
      value//free,
      value[i],
      values//free
      prove inv63a[i][v][j][k][r];
  }
}
/* invariant 6.3(b) */
forall (i in PROC) for(v = 1; v ≤ 2; v = v + 1) forall (j in PROC) {
  inv63b[i][v][j] : assert G ( (minus1_agree[i][v] ∧ fill_agree[i][v] ∧ fill_maxr[i]=round[i] ∧ value[i]=v)
    ⇒ (round[j]≥round[i] ⇒ value[j]=v) );
  forall (r1 in NUM) forall (r2 in NUM) {
    subcase inv63b[i][v][j][r1][r2] of inv63b[i][v][j]
      for round[i]=r1 ∧ round[j]=r2;
      using (inv63b[i][v][j]),
      (inv63a[i][v][j]),
      (inv63c[i][v][j]),
      /* lemmas for inv63(b) */
      lemma612a[i][v],
      lemma612b[i][v],
      lemma612c[i][v],
      inv8[j],
      inv12[i],
      inv12[j],
      inv30[i][v][j],
      /* required abstractions */
      NUM⇒{r1-1..r1},
      /* free variables in cone */
      agree//free,
      agree[j][1],
      agree[j][2],
      array_agree//free,
      array_fill_agree//free,
      array_fill_agree[i][v][j],

```



```

array_minus1_agree//free,
array_minus1_agree[i][v][j],
count//free,
count[i],
fill_maxr//free,
fill_maxr[i],
fill_maxr[j],
fill_agree//free,
fill_agree[i][v],
history_round//free,
minus1_agree//free,
minus1_agree[i][v],
decide//free,
rounds//free,
rounds[i][j],
value//free,
value[i],
value[j],
values//free,
values[i][j],
start//free
prove inv63b[i][v][j][r1][r2];
}
}
/* invariant 6.3(c) */
forall (i in PROC) for(v = 1; v ≤ 2; v = v + 1) forall (j in PROC) {
  inv63c[i][v][j] : assert G ( (minus1_agree[i][v] ∧ fill_agree[i][v] ∧ fill_maxr[i]=round[i] ∧ value[i]=v)
    ⇒ ( ((count[i]>j ∨ pc[i]=CHECK1 ∨ pc[i]=CHECK2) ∧ round[j]=round[i]-1 ∧ ¬value[j]=v) ⇒ fill_maxr[j]≥round[i] ) );
  forall (r1 in NUM) forall (r2 in NUM) {
    subcase inv63c[i][v][j][r1][r2] of inv63c[i][v][j]
      for round[i]=r1 ∧ round[j]=r2;
      using (inv63c[i][v][j]),
      (inv63a[i][v][j]),
      (inv63b[i][v][j]),
      /* other invariants */
      inv1[j],
      inv4[j],
      inv10[j],
      inv12[i],
      inv12[j],
      /* required abstraction */
      NUM⇒{r1-1..r1},
      /* free variables in cone */
      agree//free,
      array_fill_agree//free,
      array_fill_agree[i][v][j],
      array_minus1_agree//free,
      array_minus1_agree[i][v][j],
      count//free,
      count[i],
      count[j],
      decide//free,
      fill_maxr//free,
      fill_maxr[i],
      fill_maxr[j],
      history_round//free,
      start//free,
      round//free,
      round[i],
      round[j],

```

```
    rounds//free,  
    rounds[i][j],  
    value//free,  
    value[i],  
    value[j],  
    values//free,  
    values[i][j]  
    prove inv63c[i][v][j][r1][r2];  
  }  
}  
  
/*-----END-----*/  
}
```