```
/* consensus protocol [AH90] */
/* gxn 24/11/00*/
/* randomization replaced with non-deterministic choice */
/* note used values 1 and 2 not values 0,1 (use 0 to model bottom) */
/* THIS FILE CONTAINS THE PROOF OF PROBABILISTIC PROGRESS */

/*————————————————————————————————————-*/

/* 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;
/* act left unspecified : the scheduler's choice */

/* 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] ≥ 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 ≥ maxr[i]-1 read by process i agree on v */
minus1_agree : array PROC of array 1..2 of boolean;

/* 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;
            }
        }
    }
}
```

```
/* conjuction 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 lemma1 and lemma2 global maxr is correct */
/* from lemma8 and lemma2 and lemma7 fill_maxr is correct */
/* also lemma8 and lemma7 allows us to use fill_maxr in the definition of array_agree etc */

/* 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 j 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;
            }
        }
    }
}
/* conjuction of arrays */
forall (i in PROC) for(v = 1; v ≤ 2; v = v + 1) {
```

```
    fill_agree[i][v] := ∧[ 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 if process j has round greater than r then value[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] := ∧[ 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] := ∧[ (round[i]=r ⇒ ¬fill_agree[i][2]) : i in PROC ] ∧ ∧[ fillr_agree[i][r][1] : i in PROC] ∧
                ∧[ (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] of 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 j but read some processes update fill_maxr */
        next(history_fill_maxr[i]) := next(history_round)>fill_maxr[i] ? act : history_fill_maxr[i];
    }
}
```

```
/*————————————PROOF OF PROBABILISTIC PROGRESS————————————————*/


/*————————————EXTRA INVARIANTS————————————————-*/

/* EXTRA INVARIANTS */
/* the following have been proved in invariants.smv */
forall (i in PROC) {
    inv1[i] : assert G ( round[i]≤global_maxr );
    assume inv1[i];
}
forall (i in PROC) {
    inv8[i] : assert G ( round[i]≤fill_maxr[i] );
    assume inv8[i];
}
forall (i in PROC) {
    inv9[i] : assert G ( fill_maxr[i]≤global_maxr );
    assume inv9[i];
}
forall (i in PROC) for(v = 1; v ≤ 2; v = v + 1) {
    inv24[i][v] : assert G ( (pc[i]=CHECK1 ∨ pc[i]=CHECK2 ∨ pc[i]= DECIDE ∨ pc[i]= NIL) ⇒ (agree[i][v] = fill_agree[i][v]) );
    assume inv24[i][v];
}
forall (i in PROC) forall (r in NUM) for(v = 1; v ≤ 2; v = v + 1) {
    inv27[i][r][v] : assert G ( round[i]>r ⇒ (fillr_agree[i][r][v] ⇒ agree[i][v]) );
    assume inv27[i][r][v];
}
forall (i in PROC) forall (r in NUM) for(v = 1; v ≤ 2; v = v + 1) {
    inv29[i][r][v] : assert G ( round[i]>r+1 ⇒ (fillr_agree[i][r][v] ⇒ minus1_agree[i][v]) );
    assume inv29[i][r][v];
}
forall (i in PROC) {
    inv64[i] : assert G ( pc[i]=INITIAL ⇒ round[i]=0 );
    assume inv64[i];
}

/*————————————————PROOF of inv73————————————————————-*/

/* we prove that from any state either we reach a state where the global max has increases by 1 */
/* there is a unique process with round=global_maxr */
/* and all other process have a round less than the global max */
/* or all processes decide */
/* (which we change to the fact that the global maximum round does not increase) */
/* this is sufficent since from */
/* note needed fairness for inv23 */
forall (r in NUM) forall (i in PROC) {
    inv25[r][i] : assert G ( round[i]=r ⇒ F G ( round[i]≥r ∨ pc[i]=NIL ) );
    assume inv25[r][i];
}
forall (i in PROC) forall (r in NUM) {
    inv73[i][r] : assert G (global_maxr=r ⇒ ( ( F ( global_maxr=r ∧ X (global_maxr=r+1) ∧ ¬(act=i)
                                              ⇒ X(round[i]≤r) ∧ (act=i) ⇒ X(round[i]=r+1 ∧ count[i]=1 ∧ (value[i]=1 ∨ value[i]=2))) )
                                              ∨ ( G (global_maxr≤r)) ));
            using
            inv1[i],
            /* required abstraction */
            PROC⇒{1,i,N},
            NUM⇒{r..r+1},
            /* free variables in cone */
```

```
            agree//free,
            array_agree//free,
            array_minus1_agree//free,
            count//free,
            count[i],
            decide//free
            prove inv73[i][r];
}

/*————————————————PROOF of inv76————————————————————————-*/

/* we prove if inv76 holds then inv76 holds in all future states */
/* under the following assumptions */

/* assumption that the coin flipper is (v,r)-global */
/* assume v=1 */
/*that is if the process is in round r and flips the coin it returns 1 */
forall (i in PROC) forall (r in NUM) {
    vr_global[i][r] : assert G ( (round[i]=r ∧ act=i ∧ pc[i]=CHECK2 ∧ ¬agree[i][1] ∧ ¬agree[i][2]) ⇒ X(value[i]=1) );
    assume vr_global[i][r];
}
/* assume that process l is the process when enters round r first with value 1 */
forall (l in PROC) forall (r in NUM) {
    assumption1[l][r] : assert G ( round[l]=r ⇒ ¬(value[l]=2) );
    assume assumption1[l][r];
}
/* to use this we must prove invariant 7.5 */
forall (i in PROC) forall (r in NUM) {
    inv75[i][r] : assert G ( (round[i]=r ∧ value[i]=1) ⇒ G (round[i]=r ⇒ ¬value[i]=2 ) );
        using
            NUM⇒{r..r+1},
            agree//free,
            array_agree//free,
            array_minus1_agree//free,
            count//free,
            decide//free,
            fill_maxr//free,
            global_maxr//free,
            minus1_agree//free,
            rounds//free,
            start//free,
            values//free
            prove inv75[i][r];
}
/* assume process i has round greater than r */
/* at the starting point round[i]=r and we have proved below (helper[r][i]) that round cannot decrease */
forall (l in PROC) forall (r in NUM) {
    assumption2[l][r] : assert G ( round[l]≥r );
    assume assumption2[l][r];
}
/* round always increases */
forall (r in NUM) forall (i in PROC) {
    helper[r][i] : assert G ( round[i]=r ⇒ G (round[i]≥r) );
}

/* CONDITION 1: round[i] =r -¿ !fill_agree[i][2] */
/* for this we need assumption 1 if round[l] then value[l] ≠ 2 */
/* where process l is the first to enter round r */
/* note added l to the case splitting since need to assume properties about process l */
    forall (i in PROC) forall (r in NUM) {
```

```
    inv76a[i][r] : assert G ( (r>0 ∧ inv76[r]) ⇒ X (round[i]=r ⇒ ¬fill_agree[i][2]) );
    forall (r1 in NUM) forall (l in PROC) {
        subcase inv76a[i][r][r1][l] of inv76a[i][r]
            for fill_maxr[i]=r1;
            using
            assumption1[l][r],
            assumption2[l][r],
            inv8[i][r],
            /* free variables in cone */
            agree//free,
            array_agree//free,
            array_fill_agree//free,
            array_fill_agree[i][2][l],
            array_minus1_agree//free,
            count//free,
            decide//free,
            global_maxr//free,
            history_round//free,
            fill_agree//free,
            fill_agree[i][2],
            fillr_agree//free,
            minus1_agree//free,
            start//free
            prove inv76a[i][r][r1][l];
    }
}
/* now need to prove for all processes by contradiction */
/* however, we first need to know the next value of round[i]=r -¿ !fill_agree[i][2] */
next1 : array PROC of array NUM of boolean;
/* then define it to be the next value of round[i]=r -¿ !fill_agree[i][2] */
forall (i in PROC) forall (r in NUM) {
    next1_assumption[i][r] : assert G ( next1[i][r] ⇒ X (round[i]=r ⇒ ¬fill_agree[i][2]) );
    assume next1_assumption[i][r];
}
z1 : array NUM of PROC;
forall (r in NUM) z1[r] := { i : i in PROC, ¬next1[i][r] };
/* now prove by contradiction */
/* again we need to case split on l and use assumption1 and assumption2 */
forall (r in NUM) {
    inv76a1[r] : assert G ( (r>0 ∧ inv76[r]) ⇒ X ( ∧[ round[i]=r ⇒ ¬fill_agree[i][2] : i in PROC] ) );
    forall (r1 in NUM) forall (i in PROC) forall (l in PROC) {
        subcase inv76a1[r][r1][i][l] of inv76a1[r]
            for fill_maxr[i]=r1 ∧ z1[r]=i;
            using
            inv76a[i][r][r1][l],
            assumption1[l][r],
            assumption2[l][r],
            inv8[i][r],
            next1_assumption[i][r],
            /* free variables in cone */
            agree//free,
            array_agree//free,
            array_fill_agree//free,
            array_fill_agree[i][2][l],
            array_minus1_agree//free,
            count//free,
            decide//free,
            global_maxr//free,
            history_round//free,
            fill_agree//free,
```

```
            fill_agree[i][2],
            fillr_agree//free,
            minus1_agree//free,
            start//free
            prove inv76a1[r][r1][i][l];
    }
}
/* CONDITION 2: fillr_agree[i][r][1] */
/* for one element of the array */
forall (i in PROC) forall (r in NUM) forall (j in PROC) {
    inv76b[i][r][j] : assert G ( (r>0 ∧ inv76[r]) ⇒ ( X ( array_fillr_agree[i][r][1][j] ) ) );
    forall (r1 in NUM) {
        subcase inv76b[i][r][j][r1] of inv76b[i][r][j]
            for round[j]=r1;
            using
            inv27[j][r][1],
            vr_global[j][r],
            inv64[j],
            inv24[j][1],
            inv24[j][2],
            /* free variables in cone */
            agree//free,
            agree[j][1],
            agree[j][2],
            array_agree//free,
            array_minus1_agree//free,
            array_fillr_agree//free,
            array_fillr_agree[i][r][1][j],
            count//free,
            count[i],
            decide//free,
            fill_agree//free,
            fill_agree[j][2],
            fillr_agree//free,
            fillr_agree[i][r][1],
            fillr_agree[j][r][1],
            fill_maxr//free,
            global_maxr//free,
            history_round//free,
            minus1_agree//free,
            round//free,
            round[j],
            rounds//free,
            rounds[i][j],
            start//free,
            value//free,
            value[j],
            values//free,
            values[i][j]
            prove inv76b[i][r][j][r1];
    }
}
/* similarly to inv76a1 we reach */
forall (i in PROC) forall (r in NUM) {
    inv76b1[i][r] : assert G ( (r>0 ∧ inv76[r]) ⇒ ( X ( ∧[ fillr_agree[i][r][1] : i in PROC ] ) ) );
    assume inv76b1[i][r];
}
/* CONDITION 3: round[j]¿r -¿ value[i]=1 */
forall (r in NUM) forall (i in PROC) {
    inv76c[r][i] : assert G ( (r>0 ∧ inv76[r]) ⇒ ( X (round[i]>r ⇒ value[i]=1) ) );
```

```
    forall (r1 in NUM) {
        subcase inv76c[r][i][r1] of inv76c[r][i]
            for round[i]=r1;
            using
            inv27[i][r][1],
            vr_global[i][r],
            inv64[i],
            inv24[i][1],
            inv24[i][2],
            /* free variables in cone */
            agree//free,
            agree[i][1],
            agree[i][2],
            array_agree//free,
            array_minus1_agree//free,
            array_fillr_agree//free,
            count//free,
            decide//free,
            decide[i],
            global_maxr//free,
            fill_agree//free,
            fill_agree[i][2],
            fillr_agree//free,
            fillr_agree[i][r][1],
            minus1_agree//free,
            round//free,
            round[i],
            rounds//free,
            start//free,
            value//free,
            value[i],
            values//free
            prove inv76c[r][i][r1];
    }
}
/* similarly to inv76a1 we reach */
forall (r in NUM) {
    inv76c1[r] : assert G ( (r>0 ∧ inv76[r]) ⇒ ( X ( ∧[ (round[i]>r ⇒ value[i]=1) : i in PROC] ) ) );
    assume inv76c1[r];
}
/* combining inv76a1, inv76b1 and inv76c1 */
/* we reach invariant 7.6 */
forall (r in NUM) {
    inv76d[r] : assert G ( inv76[r] ⇒ X(inv76[r]) );
    assume inv76d[r];
}
/*————————————————PROOF of inv77————————————————————-*/

/* under the assumption invariant 7.6 holds the global max round does not exceed r+2 */
/* first need the following helper lemma */
forall (i in PROC) forall (r in NUM) {
    inv77a[i][r] : assert G ( (inv76[r] ∧ fill_maxr[i]=round[i] ∧ round[i]>r+1) ⇒ decide[i] );
    forall (r1 in NUM) {
        subcase inv77a[i][r][r1] of inv77a[i][r]
            for round[i]=r1;
            using
            inv29[i][r][1],
            /* required abstraction */
            NUM⇒{r..r+1},
            /* free variables in cone */
```

```
            agree//free,
            array_agree//free,
            array_fill_agree//free,
            array_fillr_agree//free,
            array_minus1_agree//free,
            count//free,
            fill_agree//free,
            fillr_agree//free,
            fillr_agree[i][r][1],
            global_maxr//free,
            minus1_agree//free,
            minus1_agree[i][1],
            pc//free,
            rounds//free,
            start//free,
            value//free,
            values//free
            prove inv77a[i][r][r1];
    }
}
/* prove that processes do not enter READ2 with round[i]=r+2 */
/* under the assumption they start with round less than or equal to r and inv76 is true */
forall (i in PROC) forall (r in NUM) {
    inv77b[i][r] : assert G ( (r>0 ∧ round[i]≤r ∧ inv76[r]) ⇒ G (round[i]=r+2 ⇒ ¬(pc[i]=READ2 ∨ pc[i]=CHECK2) ) );
    forall (r1 in NUM) {
        subcase inv77b[i][r][r1] of inv77b[i][r]
            for round[i]=r1;
            using
            inv27[i][r][1],
            inv76d[r],
            /* required abstraction */
            NUM⇒{r..r+2},
            /* free variables in cone */
            agree//free,
            agree[i][1],
            array_agree//free,
            array_fill_agree//free,
            array_fillr_agree//free,
            array_minus1_agree//free,
            count//free,
            decide//free,
            fill_agree//free,
            fillr_agree//free,
            fillr_agree[i][1],
            fill_maxr//free,
            global_maxr//free,
            rounds//free,
            value//free,
            values//free
            prove inv77b[i][r][r1];
    }
}
/*from inv77b we can make the following assumption */
/* on all the paths that we consider */
forall (i in PROC) forall (r in NUM) {
    inv77_assumption[i][r] : assert G ( round[i]=r+2 ⇒ ¬(pc[i]=READ2 ∨ pc[i]=CHECK2) );
    assume inv77_assumption[i][r];
}
/* next we show that global max round does not increase in one time step */
/* since the value of global max depends on the process scheduled */
```

```
/* we need to include this as a parameter (e.g. act is a parameter) */
forall (r in NUM) {
    inv77c[r] : assert G ( (r>0 ∧ inv76[r] ∧ global_maxr≤r+2) ⇒ X (global_maxr≤r+2) );
    forall (i in PROC) forall (r1 in NUM) {
        subcase inv77c[r][i][r1] of inv77c[r]
            for act=i ∧ round[i]=r1;
            using
            inv76d[r],
            inv8,
            inv9,
            inv77a[i][r],
            inv77_assumption[i][r],
            /* required abstraction */
            NUM⇒{r..r+2},
            /* free variables in cone */
            agree//free,
            array_agree//free,
            array_fill_agree//free,
            array_fillr_agree//free,
            array_minus1_agree//free,
            count//free,
            decide//free,
            decide[i],
            fill_agree//free,
            fill_maxr//free,
            fill_maxr[i],
            fillr_agree//free,
            fillr_agree[i][r][1],
            minus1_agree//free,
            start//free,
            value//free
            prove inv77c[r][i][r1];
    }
}
/* then above to show it holds along all paths under the assumption (inv76[r] → G(inv76[r]) holds */
forall (r in NUM) {
    inv77[r] : assert G ((r>0 ∧ global_maxr≤r ∧ inv76[r]) ⇒ G (global_maxr≤r+2) );
        using
            inv77c[r],
            inv76d[r],
            NUM⇒{r..r+2}
            prove inv77[r];
}

/*————————————————-END————————————————————*/

}
```