Majority and Byzantine Agreement[a]

N. Shankar
shankar@csl.sri.com
URL: http://www.csl.sri.com/~shankar/

Computer Science Laboratory
SRI International
Menlo Park, CA

[a] Partly based on the Ehdm work of John Rushby. Funded by NASA.

Overview

- Safety-critical systems are an important application area for mechanized formal methods.
- N-modular redundancy is used to achieve fault-tolerance.
- Redundancy management is itself the primary source of unreliability.
- Majority and Byzantine Agreement are basic operations in coordinating N-plex fault-tolerant systems.
- We present PVS specifications and proofs of these operations.
- The goal is to illustrate the formal development of mechanically checked proof for some clever algorithms.

Boyer and Moore's MJRTY

- A linear-time algorithm for computing majority.
- **Intuition:**
  - Pair off and eliminate conflicting votes until no conflicting votes remain.
  - The surviving candidate (if any) is the only possible winner.
- **Why?** Because the majority candidate cannot be eliminated by any pairing of conflicting votes.

The MJRTY Algorithm

The intuition can be translated to an iterative program over the array `poll` as shown.

```
cand, lead := noname, 0;
FOR i = 0 TO (N - 1) DO
 IF poll[i] = cand
   THEN lead := lead + 1
   ELSIF lead > 0
     THEN lead := lead - 1
   ELSE cand, lead := poll[i], 1
 ENDIF
```

MJRTY in PVS

`T` is the type of candidates,
`n` is the number of votes,
`poll` is an array of size `n` of `T`, and `caucus` is a subset from `0..n-1`.

```
mjrty [T : NONEMPTY_TYPE, n: posnat ]
: THEORY
  BEGIN
  index: TYPE = below[n]
  vector: TYPE = [index -> T]
  A, B, C : VAR T
  v, v1, v2: VAR vector
  p, q, r  : VAR index
  caucus : VAR PRED[index]
  i, j : VAR upto[n]
  m : VAR nat
  votes: TYPE = vector
  noname : T
  poll : VAR votes
  IMPORTING
   cardinality@finite_cardinality[below[n], n,
                                  id[below[n]]],
   cardinality@card_set[below[n], n, id[below[n]]]
```


MJRTY in PVS

`mjrty` computes the majority candidate (and lead) from among a `caucus` of the votes.

```
  result: TYPE = [T, nat]
  P,Q: VAR result

  mjrty(poll, caucus, i): RECURSIVE result   =
    (IF i> 0
      THEN
       IF caucus(i-1)
        THEN (LET (A, m) = mjrty(poll,caucus, i-1)
              IN
              (IF poll(i-1) =  A
                THEN (A, m+1)
              ELSIF m > 0
                    THEN (A, m -1)
                    ELSE (poll(i-1), 1)
              ENDIF))
        ELSE mjrty(poll, caucus,i-1)
       ENDIF
      ELSE (noname, 0)
      ENDIF)
    MEASURE i
```


MJRTY in PVS

Majority candidate is the first component of the `mjrty` result.

`count_votes` counts votes for A in `caucus` of votes in `poll` from `0..i-1`. `fincardi` (from the `cardinality` library) computes the cardinality of a subset of numbers from `0..i-1`.

```
  maj(caucus, v): T = proj_1(mjrty(v, caucus, n))

  count_votes(poll, caucus, A, i): nat =
    fincardi((LAMBDA p: caucus(p)
                        AND poll(p) = A),
             i)
```


MJRTY Invariant

The main invariant asserts that if `A` has lead `m` out of `i` votes, and `B` $\neq$ `A`,
then
`B` is over `m` votes shy of a majority.

```
  invariant(poll,caucus, i):  bool =
   (LET (A, m) = mjrty(poll,caucus, i)
    IN (FORALL (B : T) :
         2* (count_votes(poll, caucus, B, i) +
         (IF B = A THEN 0 ELSE m ENDIF))
          <= fincardi(caucus, i)+ m))

  invariant_holds: LEMMA
    invariant(poll, caucus, i)
```

Proved by

```
("" (INDUCT-AND-SIMPLIFY "i"))
```

MJRTY Correctness

We can then easily deduce that

1. No non-winner has a majority.
2. The majority candidate wins.
3. `mjrty` only depends on the `caucus` votes.

```
correctness_step2: LEMMA
 (LET (B, m) = mjrty(poll, caucus,i)
  IN
  (A /= B
  IMPLIES 2*count_votes(poll,caucus, A, i)
         <= fincardi(caucus, i)))

maj_correctness:
LEMMA
   2 * count_votes(poll, caucus, A, n)
    > fincard(caucus)
  IMPLIES maj(caucus, poll) = A

maj_ext: LEMMA
  (FORALL p : caucus(p) IMPLIES v1(p) = v2(p))
    IMPLIES maj(caucus, v1) = maj(caucus, v2)
```


The Byzantine Generals Problem

- The *Byzantine Generals* (BG) problem: a Commanding General has an order that must be reliably conveyed to a set of Lieutenant Generals where a bounded subset of the generals can be unreliable.
- **Requirements:** There are n generals, of whom as many as m can be faulty (including the Commander)

  BG1: nonfaulty lieutenants agree on the value received from the Commander

  BG2: if the Commander is nonfaulty, the value received by every nonfaulty lieutenant is the value he sent

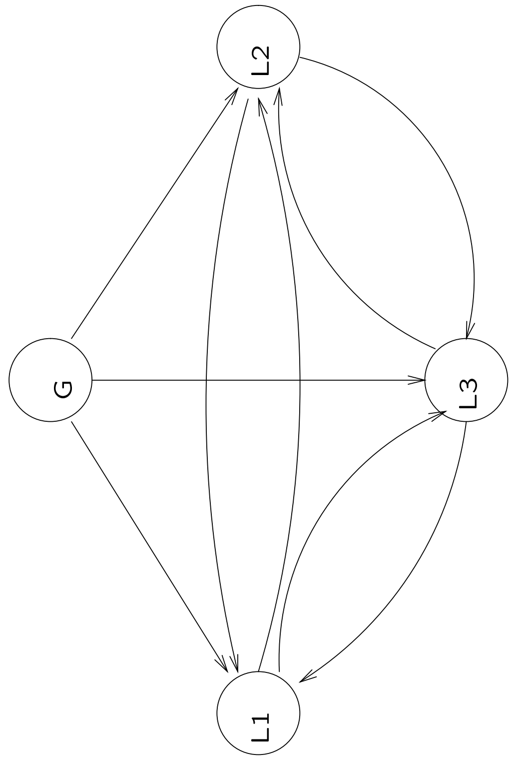
The Oral Messages (OM) Algorithm for BG

- Requires $n > 3m$
- And $m + 1$ rounds of message exchange
- So need four channels and two rounds to withstand a single fault

The Oral Messages Algorithm OMBG

- The Commander broadcasts a value to the lieutenants.
- If $m = 0$, each lieutenant accepts the value received.
- Otherwise, each lieutenant takes the part of the general in $OMBG(m - 1)$ to broadcast the value received to the other lieutenants.
- Each lieutenant takes majority vote of the values received directly from Commander and via the other lieutenants.

Oral Messages



Proof of OMBG: Validity

Quoting from Lamport, Shostak, and Pease (LSP):

> **Lemma 0.1** *For any m and k, Algorithm OMBG(m) satisfies BG2 if there are more than $2k + m$ participants and at most k traitors.*
>
> **Proof.** The proof is by induction on m. BG2 only specifies what must happen if the General is loyal. Using A1, it is easy to see that the trivial algorithm OMBG(0) works if the General is loyal, so the lemma is true for $m = 0$. We now assume it is true for $m - 1$, $m > 0$, and prove it for m.
>
> In step (1), the loyal General sends a value v to all $n - 1$ lieutenants. In step (2), each loyal lieutenant applies OMBG($m - 1$) with $n - 1$ generals. Since by hypothesis $n > 2k + m$, we have $n - 1 > 2k + (m - 1)$, so we can apply the induction hypothesis to conclude that every loyal lieutenant gets $v_j = v$ for each loyal Lieutenant j. Since there are at most k traitors, and $n - 1 > 2k + (m - 1) \geq 2k$, a majority of the $n - 1$ lieutenants are loyal. Hence, each loyal lieutenant has $v_i = v$ for a majority of the $n - 1$ values i, so he obtains $majority(v_1, \ldots, v_{n-1}) = v$ in step (3), proving BG2. ∎

Proof of OMBG: Agreement

Theorem 0.2 *For any m, Algorithm OMBG(m) satisfies conditions BG1 and BG2 if there are more than $3m$ participants and at most m traitors.*

Proof. The proof is by induction on m. If there are no traitors, then it is easy to see that OMBG(0) satisfies BG1 and BG2. We therefore assume that the theorem is true for OMBG($m - 1$) and prove it for OMBG(m), $m > 0$.

We first consider the case in which the General is loyal. By taking k equal to m in Lemma 1, we see that OMBG(m) satisfies BG2. BG1 follows from BG2 if the General is loyal, so we only need verify BG1 in the case the General is a traitor.

There are at most m traitors, and the General is one of them, so at most $m - 1$ of the lieutenants are traitors. Since there are more than $3m$ generals, there are more than $3m - 1$ lieutenants, and $3m - 1 > 3(m - 1)$. We may therefore apply the induction hypothesis to conclude that OMBG($m - 1$) satisfies conditions BG1 and BG2. Hence, for each j, any two loyal lieutenants get the same value for v_j in step (3). (This follows from BG2 if one of the two lieutenants is Lieutenant j, and from BG1 otherwise). Hence, any two loyal lieutenants get the same vector of values $v_1, \ldots, v_{n-1}$, and therefore obtain the same value $majority(v_1, \ldots, v_{n-1})$ in step (3), proving BG1. ∎

OMBG in PVS

Assuming $3m < n$ for n processors with at most m faulty, i.e., not `ok`.

`T` is the message type.

```
byzantine[m: nat, n: posnat, T: NONEMPTY_TYPE]
  : THEORY
  BEGIN
  ASSUMING

    mn_prop: ASSUMPTION 3 * m < n

  ENDASSUMING

  npos: LEMMA n > 0
  proc: TYPE = below[n]
  rounds: TYPE = upto[m]
  vector: TYPE = [proc -> T]
  procset: TYPE = setof[proc]
  ok: procset
  G, p, q, z: VAR proc
  r: VAR rounds
  caucus: VAR procset
  v: VAR vector
  t: VAR T
```

OMBG in PVS: Cardinalities

`nok(caucus)` is the subset of faulty processor ids in the `caucus`.

`allprocs` is the set of all processors.

Axiom `max_faults` bounds the cardinality of `nok(allprocs)` by `m`.

```
IMPORTING mjrty[T, n]

nok(caucus): procset =
 z: proc | member(z, caucus) AND NOT ok(z)

allprocs: procset = fullset[proc]

allset_card: LEMMA fincard(allprocs) = n

max_faults: AXIOM fincard(nok(allprocs)) <= m
```


OMBG in PVS: Cardinality Lemmas

Removing an element decrements cardinality by one.

Removing an `ok` element leaves cardinality of `nok` elements unchanged.

Removing a `nok` element decrements cardinality of `nok` elements by one.

```
remove_lemma: LEMMA
  member(p, caucus)
 IMPLIES
  fincard(remove(p, caucus))
  = fincard(caucus) - 1

nok_lemma1: LEMMA
  ok(p)
 IMPLIES
   fincard(nok(remove(p, caucus)))
   = fincard(nok(caucus))

nok_lemma3: LEMMA
  NOT ok(p) AND member(p, caucus)
    IMPLIES fincard(nok(remove(p, caucus)))
            = fincard(nok(caucus)) - 1
```


OMBG: Communication

`maj_ax` restates `maj_correctness`.

`send(t, p, q)` is the message received by `q` when `p` sends `t`.

`send_ax`: communication between two reliable processors is reliable

```
maj_ax: LEMMA
  fincard(caucus) > 2 * fincard(nok(caucus))
    AND (FORALL p: ok(p) AND
            member(p, caucus)
          => v(p) = t)
    IMPLIES maj(caucus, v) = t

send: [T, proc, proc -> T]

send_ax: AXIOM ok(p) AND ok(q)
        => send(t, p, q) = t

distr(t, p)(z : proc): T =  send(t, p, z)
```


The OMBG Algorithm in PVS

- `OMBG(G, r, v, caucus)` computes a result vector from the general G's initial broadcast vector `v` given round `r` and current `caucus`.
- In round 0, result vector equals broadcast vector.
- In round `r+1`, result vector `u` has `u(G) = v(G)` and for `p` $\neq$ `G` is the majority value from `caucus` - {`G`} of vector `w` where `w(q)` is `p`'s receipt of `q`'s broadcast (using `OMBG(r-1)`) of `v(q)`.

The OMBG Algorithm in PVS

```
OMBG(G, r, v, caucus): RECURSIVE vector =
  (IF r = 0
    THEN v
    ELSE
     (LAMBDA p:
      IF p=G
       THEN v(G)
       ELSE
        maj(remove(G, caucus),
            (LAMBDA q:
              OMBG(q, r-1, distr(v(q),q),
                   remove(G, caucus))(p)))
       ENDIF)
    ENDIF)
 MEASURE r
```


OMBG Validity in PVS

Lemma C2 Corresponds to LSP Validity lemma.

```
C2_prop(r): bool =
  (FORALL p, G, t, caucus:
    ok(p)
    AND ok(G)
    AND member(p, caucus)
    AND member(G, caucus)
    AND fincard(caucus)
        > 2 * fincard(nok(caucus)) + r
 IMPLIES OMBG(G, r, distr(t, G), caucus)(p)
         = t)

C2: LEMMA C2_prop(r)

C2_final: THEOREM ok(p) AND ok(G)
   IMPLIES
     OMBG(G, m, distr(t, G), allprocs)(p) = t
```


OMBG Validity Proof in PVS

The PVS proof has 11 user-supplied steps.

Proof of C2 follows the LSP outline and is at least as terse.

```
(""
 (AUTO-REWRITE-THEORY "byzantine")
 (INDUCT "r")
 (("1" (GRIND))
  ("2"
   (SKOSIMP)
   (EXPAND "C2_prop" +)
   (SKOSIMP)
   (SMASH)
   (REWRITE "maj_ax")
   (HIDE 3)
   (SKOSIMP)
   (INST?)
   (GRIND :EXCLUDE ("fincard" "nok")))))
```


OMBG Agreement in PVS

```
C1_prop(r): bool =
  (FORALL p, q, G, caucus, t:
     ok(p)
      AND ok(q)
      AND member(p, caucus)
      AND member(q, caucus)
      AND member(G, caucus)
      AND fincard(caucus) > 3 * r
      AND r >= fincard(nok(caucus))
   IMPLIES OMBG(G, r, distr(t, G), caucus)(p)
           = OMBG(G, r, distr(t, G), caucus)(q))

C1: LEMMA C1_prop(r)

C1_final: THEOREM
  ok(p) AND ok(q)
   IMPLIES
    FORALL (G: proc):
      OMBG(G, m, distr(t, G), allprocs)(p)
      = OMBG(G, m, distr(t, G), allprocs)(q)
```

OMBG Agreement Proof in PVS

```
(""
 (AUTO-REWRITE-THEORY "byzantine")
 (INDUCT "r")
 (("1"
   (EXPAND "C1_prop")
   (SKOSIMP*)
   (CASE "ok(G!1)")
   (("1" (GRIND))
    ("2"
     (USE "fincard_non_empty" :IF-MATCH ALL)
     (HIDE -1 -3)
     (ASSERT)
     (INST 1 "G!1")
     (ASSERT))))
  ("2"
   (STOP-REWRITE)
   (SKOSIMP)
   (EXPAND "C1_prop" +)
   (SKOSIMP)
   (ASSERT)
   ⋮
   ))
```


OMBG Agreement Proof (contd.)

```
    ⋮
    (CASE "ok(G!1)")
    (("1" (LEMMA "C2") (EXPAND "C2_prop")
          (AUTO-REWRITE -1) (ASSERT))
     ("2"
      (EXPAND "OMBG")
      (GROUND)
      (REWRITE "maj_ext")
      (HIDE 3)
      (SKOSIMP)
      (EXPAND "C1_prop")
      (INST?)
      (GRIND :EXCLUDE "fincard" :REWRITES
       ("remove_lemma" "nok_lemma3")))))
```


Conclusion

- Lincoln and Rushby have developed and verified agreement algorithms for combinations of crash, symmetric, and Byzantine faults.
- Butler and colleagues are developing and verifying the architecture of a reliable computing platform based on fault-tolerant clock synchronization, sensor distribution, and fault masking.
- Majority and Byzantine Agreement are crucial algorithms for building fault-tolerant systems.
- These algorithms can be modelled as recursive functions in PVS.
- The verification relies heavily on the use of decision procedures.
- The proofs follow the informal outline in structure as well as level of detail.