SAL Lunch 10 Feb 1999.

# Simple Verification of a Distributed Algorithm Using Abstraction

John Rushby

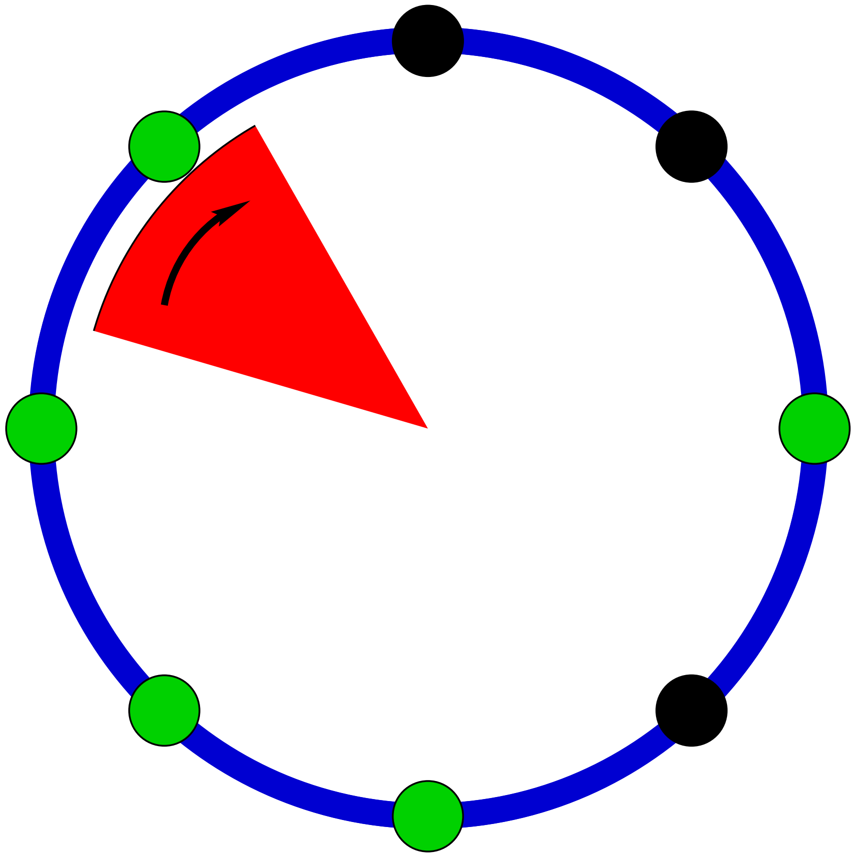
Computer Science Laboratory
SRI International
Menlo Park, CA

## Context: The Time-Triggered Architecture (TTA)

- An architecture for safety-critical real-time embedded control
- Specifically, automobiles
  - Drive-by-wire, not “body electronics”
- It’s based on a protocol called the Time-Triggered Protocol (TTP)
- It’s been adopted by Daimler-Benz, and Motorola have committed to adding TTP controllers to the their embedded processor chips
- It targets the same $10^{-9}$ requirement as flight-critical avionics
- And has quite a lot in common with Arinc 659 (the Honeywell SafeBus) used in the Boeing 777 AIMS
- And may be used by Airbus in the A3XX

# Context: The Time-Triggered Protocol (TTP)

Creates a synchronous, TDMA ring on a broadcast bus



## Context: The Time-Triggered Protocol (TTP)

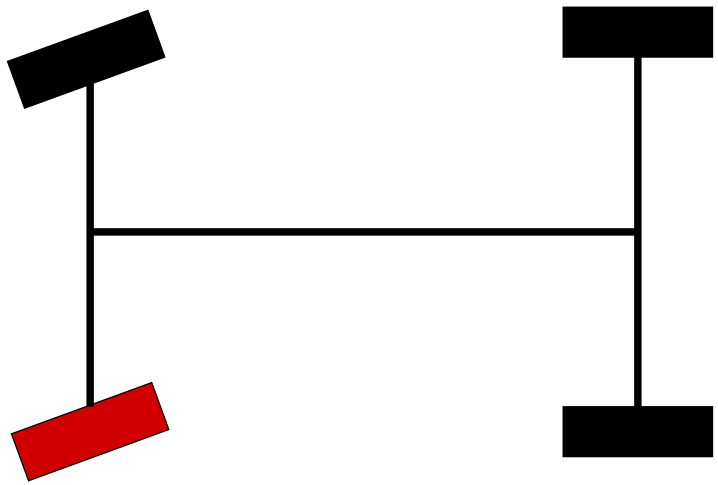
TTP provides several services in a tightly integrated manner

- Clock synchronization (verified in PVS by Pfeifer, Schwier, and Friedrich W. von Henke, DCCA '99)
- Time-triggered implementation of synchronous algorithms (verified in PVS by Rushby, DCCA '97)
- Time Division Multiple Access (TDMA)
  - Deterministic mutual exclusion
  - Data communication
- Group membership (today's topic)

Assumes (and enforces) fail silence

# The Need for Group Membership

- Consider a brake-by-wire application
- Separate computers at each wheel adjust braking force according to inputs from brake pedal, accelerometers, steering angle, wheel-spin sensors etc.
- Suppose one of these computers fails



- The others need to redistribute the braking force
- So must have consistent opinion about who has failed

## Requirements For Group Membership

Each processor maintains a membership set

**Validity:** the membership sets of nonfaulty processors contain all the nonfaulty processors

- And nothing else—not possible because it takes some time to diagnose a faulty processor
- So allow at most one faulty processor

**Agreement:** all nonfaulty processors have the same membership sets

**Self-Diagnosis:** faulty processors eventually remove themselves from their own membership sets (and fail silently)

Subject to assumptions about possible fault modes, fault arrival rate, and maximum number of faults

## Group membership: Algorithms

- Must be achieved with very low overhead
- WDAG algorithm (Katz, Lincoln, Rushby WDAG'97) does it with one bit added to each broadcast
- I'll describe a similar, but simpler algorithm, that does not achieve full self-diagnosis
- Each processor maintains an ack bit and appends it to its message when it broadcasts
  - Interpretation is
    "I heard the previous expected broadcast"
    - ⋆ For WDAG it is "I retained the previous expected broadcaster in my membership"
- The actual TTP algorithm is different: uses two bits, encoded in the CRC checksum, has slower self-diagnosis, tricky special cases

## Fault Assumptions

**Fault modes:** very restrictive assumptions

Send Fault

- A faulty processor might not broadcast (in some rounds)
  - Broadcasts are received either by all or none of the nonfaulty processors

Receive Fault

- A faulty processor might not receive some broadcasts

**Fault arrivals:** faults must be “rare”

- Roughly, no more than one per round
- More precisely, may arrive only when system is “stable”
  - And after a fault it returns to a stable state very quickly

**Maximum faults:** no restriction

- WDAG requires at least two nonfaulty processors for self diagnosis

## Group membership: Algorithm (Modifications for WDAG)

**Initially:** `mem = fullset; ack = true`

**Broadcaster:** `mem' := mem; ack' := true`

**Receiver (p):** `LET b be current broadcaster IN`

```
not mem(b)              -> mem' := mem;     ack' := ack
ack and no msg          -> mem' := mem-{b}; ack' := false
ack and ack(b)          -> mem' := mem;     ack' := true
ack and not ack(b)      -> mem' := mem-{b}; ack' := true
ack and not ack(b)      -> mem' := mem-{b}; ack' := false
not ack and no msg      -> not mem'(p)
not ack and not ack(b)  -> mem' := mem;     ack' := true
not ack and ack(b)      -> not mem'(p)
|mem'| < 2              -> not mem'(p)   fixes WDAG bug
```

(Better fix has additional state bit and larger modifications)

## Verification

- Standard way to verify safety properties is by an inductive invariance argument
  - Prove that the initial state satisfies the property
  - Prove that if a pre-state satisfies the property, then the post-state resulting from any transition also satisfies the property
- Property of interest is seldom inductive
  (i.e., cannot complete these proofs)
- Must be strengthened by conjoining additional properties
  - These can often be discovered systematically by precondition strengthening
  - But are often numerous, and effort becomes daunting

## The WDAG Invariant

The WDAG paper presents a pen-and-paper inductive invariance argument

The invariant has the following conjuncts.

1. All nonfaulty processors have the same membership sets.
2. All nonfaulty processors are in their own membership sets.
3. All nonfaulty processors have the same value for $ack$.
4. For each processor $p$, $ack(p)$ is true iff in the most recent previous step in which $p$ expected a broadcast from a processor $b$, either $p$ was $b$, or $arrived(b, p) \wedge (ack(b) \vee \neg ack(p))$ in that step.

5. If a processor $p$ became faulty less than $n$ steps ago and $q$ is a nonfaulty processor, either $p$ is the present broadcaster or the present broadcaster is in $p$'s local membership set iff it is in $q$'s.

6. If a receive fault occurred to processor $p$ less than $n$ steps ago, then either $p$ is not the broadcaster or $ack(p)$ is $false$ while all nonfaulty $q$ have $ack(q) = true$, or $p$ is not in its local membership set.

7. If in the previous step $b$ is in the broadcaster slot, $p$ is a nonfaulty processor, and $arrived(b, p)$ does not hold, then $b$ is faulty in the current step.

8. If the broadcaster $b$ is expected by a nonfaulty processor, then $b$ is either nonfaulty, or became faulty less than $n$ steps ago.

## The WDAG Proof

- Is incorrect
  - And the algorithm has a bug
- Has defeated attempts by Pat Lincoln and me to formally verify it in PVS
  - Because of its horrible complexity

## New Approach: Disjunctive Invariants

- Instead of making the invariant a big conjunction
- Suppose we made it a big disjunction
- Each disjunct conjoins the property of interest P with some strengthenings
- For each disjunct, prove post-state resulting from any transition satisfies some collection of disjuncts

(P & a & ... & ...)

or

(P & b & ... & ...)

or

(P & c & ... & ...)

or.

.

.

.

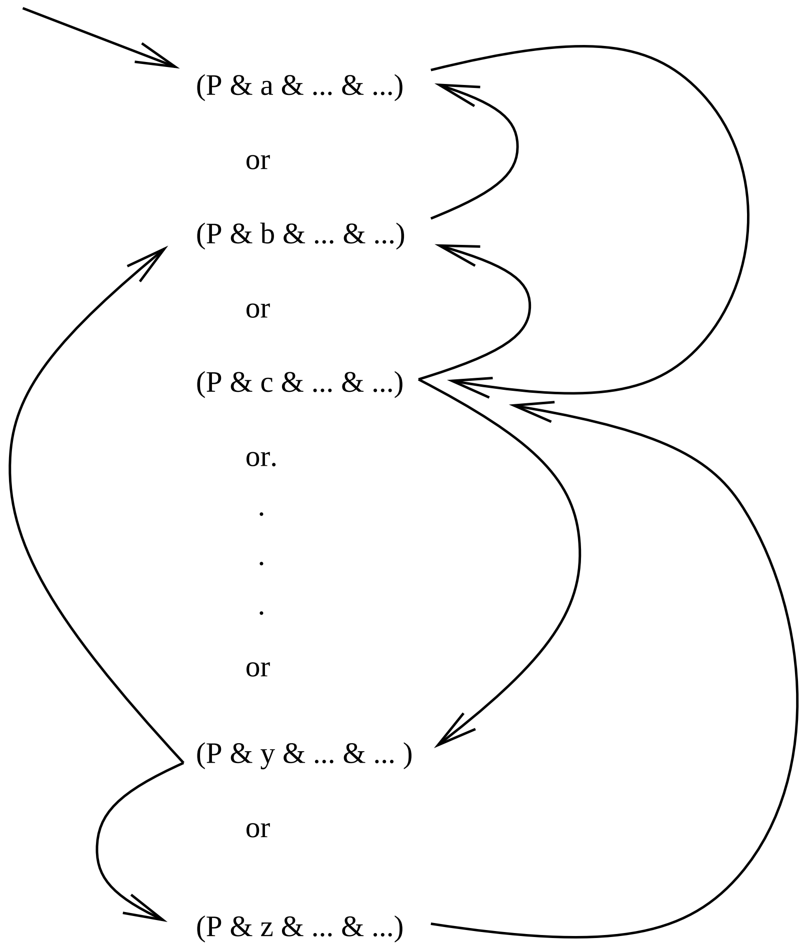
or

(P & y & ... & ... )

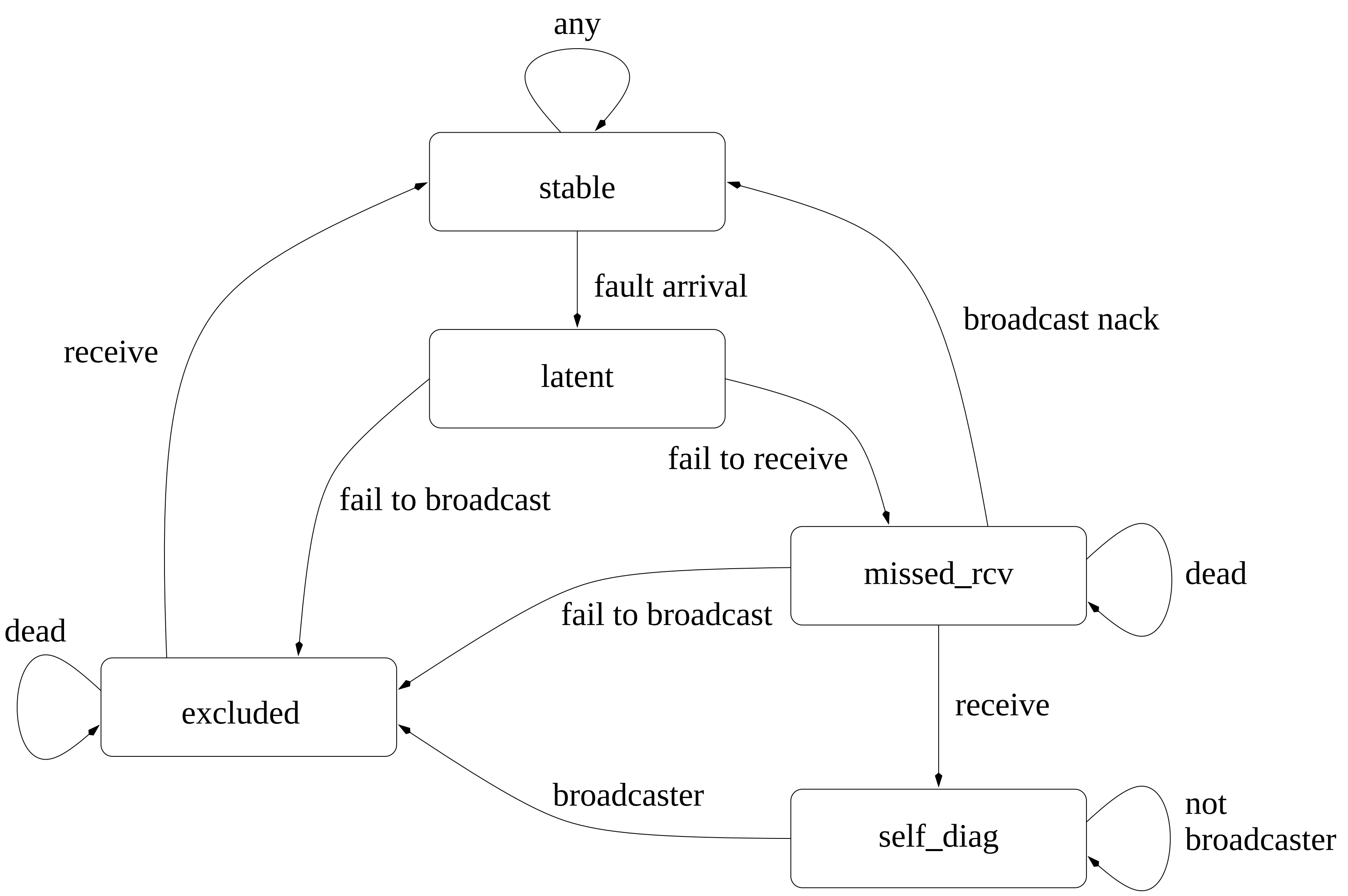
or

(P & z & ... & ...)

# The “Transitions” on a Disjunctive Invariant Look Like a State Machine



# Abstract State Machine for Group Membership Protocol



## Disjuncts for the Group Membership Protocol

```
the_mem(t)(p): bool = not faulty(t, p)

stable(t): bool =
  FORALL p: NOT faulty(t, p) IMPLIES
       membership(t, p) = the_mem(t) AND ack(t, p)

latent((t:postime), x): bool =
  NOT faulty(t-1, x) AND the_mem(t) = remove(x, the_mem(t-1))
    AND membership(t, x) = add(x, the_mem(t)) AND ack(t, x)
       AND FORALL p: NOT faulty(t, p) IMPLIES
           membership(t, p) = add(x, the_mem(t)) AND ack(t, p)

excluded(t, x): bool =
  FORALL p: NOT faulty(t, p) IMPLIES
      membership(t, p) = the_mem(t) AND NOT ack(t, p)
```

```
before(t, x, y): bool =
  FORALL (r:nat): y = broadcaster(t+r)
    IMPLIES EXISTS s: s<r AND x = broadcaster(t+s)

missed_rcv(t, y, x): bool =
  faulty(t, x) AND  x /= y AND before(t, x, y)
    AND membership(t, x) = remove(y, add(x, the_mem(t))) AND NOT ack(t, x)
        AND FORALL p: NOT faulty(t, p) IMPLIES
            membership(t, p) = add(x, the_mem(t)) AND ack(t, p)

self_diag(t, y, x): bool =
  faulty(t, x) AND x /= y
    AND membership(t, x) = remove(y, the_mem(t))
      AND FORALL p: NOT faulty(t, p) IMPLIES
          membership(t, p) = add(x, the_mem(t)) AND ack(t, p)
```

## Typical Lemmas

```
selfdiag_to_selfdiag: LEMMA
  self_diag(t, y, x)
    AND x /= broadcaster(t)
    AND the_mem(t+1) = the_mem(t)
        IMPLIES self_diag(t+1, y, x)

selfdiag_to_excluded: LEMMA
  self_diag(t, y, x)
    AND x = broadcaster(t)
    AND the_mem(t+1) = the_mem(t)
        IMPLIES excluded(t+1, x)

self_diag_exit: LEMMA
  self_diag(t,y,x) IMPLIES
     (FORALL j: x /= broadcaster(t+j))
     OR (EXISTS j: excluded(t+j, x))
```

## Typical Proof (selfdiag_to_selfdiag)

```
(SKOSIMP)
(EXPAND "self_diag")
(FLATTEN)
(INSTALL-REWRITES :DEFS NIL :THEORIES ("sets") :REWRITES
 ("the_mem" "faults_latch1" "set_prop2"))
(GROUND)
(("1" (ASSERT))
 ("2" (REWRITE "newmem") (GRND-REPL-ASSRT))
 ("3"
  (SKOSIMP) (FORWARD-CHAIN "faults_latch1x")
  (COPY -3) (FORWARD-CHAIN -) (GROUND)
  (("1"
    (REWRITE "newmem") (LIFT-IF) (GRND-REPL-ASSRT)
    (("1" (FORWARD-CHAIN -) (USE-GRND-REPL-ASSRT "arrival"))
     ("2" (FORWARD-CHAIN -))))
   ("2"
    (REWRITE "newack") (GRND-REPL-ASSRT) (FORWARD-CHAIN -)
    (USE-GRND-REPL-ASSRT "arrival")))))
```

## Proofs and Theorems

By defining some convenient strategies (macros), most of the proofs become very systematic

I have proved all three of the following

```
validity: THEOREM NOT faulty(t,p) IMPLIES
   membership(t,p) = the_mem(t)
    OR EXISTS x: faulty(t,x) AND membership(t,p) = add(x,the_mem(t))

agreement: THEOREM NOT faulty(t,p) AND NOT faulty(t,q)
     IMPLIES  membership(t,p) = membership(t,q)

liveness: THEOREM stable(t) IMPLIES EXISTS j: j>0 AND stable(t+j)
```

## The WDAG Algorithm

I have also proved the corresponding properties for the full (corrected) WDAG Algorithm

```
validity: THEOREM IF faulty(t,p)
 THEN
    NOT membership(t,p)(p)
      OR subset?(membership(t,p), add(p,the_mem(t)))
 ELSE
    membership(t,p) = the_mem(t)
      OR EXISTS x: faulty(t,x) AND membership(t,p) = add(x,the_mem(t))
 ENDIF

 stable(t): bool =
     FORALL p: IF faulty(t, p) THEN NOT membership(t,p)(p)
               ELSE membership(t, p) = the_mem(t)
                    AND ack(t, p) AND NOT pnack(t, p)
               ENDIF
```

The abstract state machine is about twice as complex

## Conclusions and Questions

- This approach makes mechanized verification of these algorithms very intuitive and straightforward
- Holger Pfeifer (Ulm) has used the approach to verify "most" of the actual TTP algorithm (it is much less uniform than WDAG)
- Is it new?
- How is it related to predicate abstraction and to proof diagrams?
- To what class of problems is it applicable?
- How can mechanized support be provided?