Can Inductive Invariants be used to enhance efficiency of Bounded Model Checking (BMC)

A case study applied to Raft Consensus Protocol

Shobhit Singh   Akhoury Shauryam

May 25, 2025

Chennai Mathematical Institute

Under the guidance of

Prof. M. Praveen & **Prof. M.K. Srivas**

Overview

Motivation

Goals and Contributions

Preliminaries

What is Raft

Models with Different Levels of Abstraction

Targeted Bugs

Verification Properties

Motivation

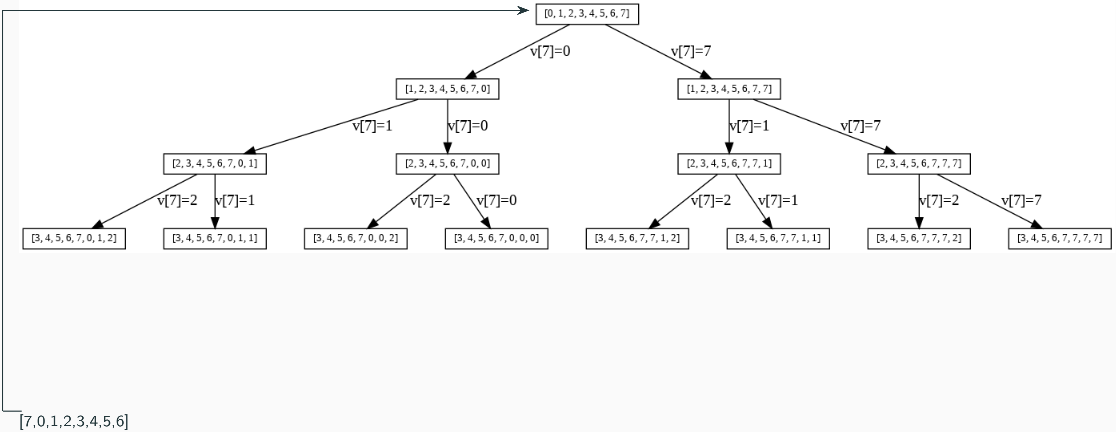
A toy example – Left-shift with a bug

```
vars  v[0..7] : 0..7           -- 8 registers
      i       : 0..80          -- step counter
      N       : 1..10          -- cycles (const)

init  v[k] := k                -- k = 0..7
      i := 0

loop  while i < 8*N do         -- one step = shift by 1
        for k = 0..6: v[k] := v[k+1]
            v[7] := v[0]  -- NOBUG
            v[7] := choose(v[0], v[7])  -- BUG
        i := i + 1
      end
INV   (i mod 8 = 0)  v = (0..7)
LTL   G(v[0] = 0  ->  X v[0] = 1)
```

Traces


[0, 1, 2, 3, 4, 5, 6, 7]
v[7]=0
v[7]=7
[1, 2, 3, 4, 5, 6, 7, 0]
[1, 2, 3, 4, 5, 6, 7, 7]
v[7]=1
v[7]=0
v[7]=1
v[7]=7
[2, 3, 4, 5, 6, 7, 0, 1]
[2, 3, 4, 5, 6, 7, 0, 0]
[2, 3, 4, 5, 6, 7, 7, 1]
[2, 3, 4, 5, 6, 7, 7, 7]
v[7]=2
v[7]=1
v[7]=2
v[7]=0
v[7]=2
v[7]=1
v[7]=2
v[7]=7
[3, 4, 5, 6, 7, 0, 1, 2]
[3, 4, 5, 6, 7, 0, 1, 1]
[3, 4, 5, 6, 7, 0, 0, 2]
[3, 4, 5, 6, 7, 0, 0, 0]
[3, 4, 5, 6, 7, 7, 1, 2]
[3, 4, 5, 6, 7, 7, 1, 1]
[3, 4, 5, 6, 7, 7, 7, 2]
[3, 4, 5, 6, 7, 7, 7, 7]
[7,0,1,2,3,4,5,6]

Techniques to catch the bug

- Symbolic Model Checking
- Bounded Model Checking
- K-Induction

Verification Techniques — Quick Scan

Technique	**Upside**	**Downside**
SMC	finds *all* bugs	state blow-up
BMC (k)	fast for shallow bugs	$k >$ bug depth
k-Induction	full proof if small k	Large k for long props
BMC + Invariants	Cuts runtime	Craft invariants

Adding Invariants → Faster BMC (Hopefully)

Idea. Inject auxiliary *inductive invariants*

$$\mathsf{Inv}_1, \ldots, \mathsf{Inv}_m \;\; s.t. \;\; \neg\mathsf{Inv}_j \;\Rightarrow\; \neg\mathsf{Safety}.$$

- Each invariant is an extra *trip-wire*. Solver may hit any one counter-example sooner.
- More trip-wires → smaller unwind bound k → runtime drop.

Adding Invariants Faster BMC (8-register Rotator)

Model recap.

```
step  if i < 8·N then
         for k=0..6: v[k] := v[k+1]
         v[7] := choose(v[0], v[7])   -- BUG: non-det choice
         i := i+1
      fi
prop  AG(i=8 → v == (0..7))           -- must reset after 8 steps
```

Auxiliary invariant

$$\boxed{\mathrm{Distinct}(v[0], \ldots, v[7])}$$

- Initially true (values 07).
- We never need to unroll to $8N$; extra "distinct" property is a cheap early violation that still implies the main safety failure.

Goals and Contributions

Project Goal

Big Picture

Apply **inductive-invariant–aided BMC** to speed up formal verification & debugging of a realistic consensus protocol model.

- **Protocol:** Raft (Leader-Election Phase only).
- **Challenge:** state explosion due to network $+$timing.
- **Hypothesis:** carefully-chosen invariants $\Rightarrow$ earlier counter-examples & shorter k in k-induction.

Contribution 1 — Abstract Raft-LEP Model

- **Balanced abstraction**
  - enough detail - captures leader election logic;
  - enough abstraction - tractable in NuSMV.
- **Network abstraction**
  - use *non-determinism* to model message delay/loss;
  - sidestep explicit channel state explosion.

Contribution 2 — Safety Property & Inductive Invariants

Target safety: *At most one leader per term*

Safety Property

At most one leader per term.

Inductive invariants crafted

- **Unique Quorum:** There can only be one node with majority votes.
- **Unique Vote:** A voter grants at most one vote per term.
- **Leader Uniqueness:** If two nodes think they're leaders, terms differ.

Contribution 3 — Proof Experiments

1. **Baseline** Model-check correct model M — no counter-example.
2. **Bug injection** Produce faulty model M' (drop "unique vote" rule).
3. **Experiments on M'**
   - BMC alone — finds safety violation after deep unwind.
   - BMC + invariants — violation within few steps.
   - k-Induction with invariants — fast proof of *unsafety*.
4. **Metrics collected** #SAT calls, max k, wall-clock time.

Preliminaries

Model Checking

- **Model Checking** is an automated technique to verify whether a system satisfies a temporal logic specification (e.g., in LTL or CTL).
- The system is modeled as a transition system:

$$\mathcal{M} = (S, I, T)$$

  where:

  - S: Set of states
  - $I(s)$: Predicate defining initial states
  - $T(s, s')$: Transition relation

BMC vs SMC

Two Approaches:

- **Symbolic Model Checking (with BDDs):**
  - Uses Binary Decision Diagrams to represent I, T, and sets of states.
  - Explores the full reachable state space.
  - Can prove properties but may suffer from BDD blowup.
- **Bounded Model Checking (BMC):**
  - Unrolls the system up to depth k and searches for counterexamples.
  - Encodes the problem as a SAT/SMT formula.
  - Efficient for bug finding; incomplete for full correctness.

Bounded Model Checking (BMC): Encoding

- BMC checks if a counterexample exists within k steps.
- Construct a logical formula and query a SAT/SMT solver for satisfiability.

Formula for BMC:

$$I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg\varphi(s_i)$$

- $I(s_0)$: Initial state predicate
- $T(s_i, s_{i+1})$: Transition relation unrolled up to step k
- $\neg\varphi(s_i)$: Violation of the desired property at any step $i \in [0, k]$

- If satisfiable, a counterexample trace exists.
- If unsatisfiable, no violation exists within bound k.
- Cannot conclude correctness beyond the bound.

Normal Induction vs K-Induction

Goal: Prove property $P(s)$ holds in all reachable states of a system.

Standard Induction:

- **Base Case:** $P(s_0)$
- **Inductive Step:** $P(s) \wedge T(s, s') \Rightarrow P(s')$

K-Induction:

- **Base Cases:** $P(s_0), \ldots, P(s_k)$
- **Inductive Step:**

$$P(s_0), \ldots, P(s_{k-1}),$$
$$T(s_0, s_1), \ldots, T(s_{k-1}, s_k) \Rightarrow P(s_k)$$

Over-Approximation in K-Induction

- Inductive step assumes:

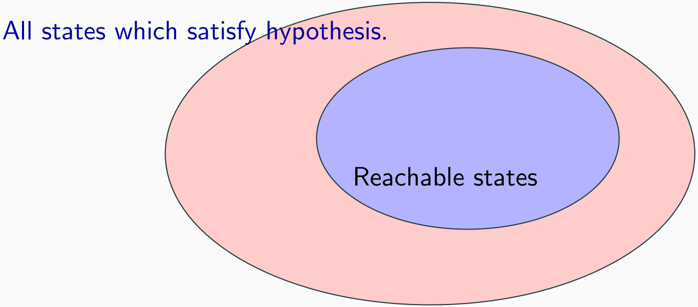
$$P(s_0), \ldots, P(s_{k-1}) \wedge T(s_0, s_1), \ldots, T(s_{k-1}, s_k)$$

- But no requirement that s_0 is reachable!
- So it proves the property over a **superset** of reachable traces.

Implication

If the property holds on this over-approximation,
then it must hold on the actual reachable states.

Overapproximation



If P holds in pink, it holds in reachable! but if its fails, we might need to strengthen the invariant i.e. make it the over-approximation tighter.

Formula Sent to SAT/SMT Solver

For a safety property P, initial predicate I, and transition T:

Base Case (Unrolling to depth k):

$$I(s_0) \wedge T(s_0, s_1) \wedge \cdots \wedge T(s_{k-1}, s_k) \Rightarrow P(s_0) \wedge \cdots \wedge P(s_k)$$

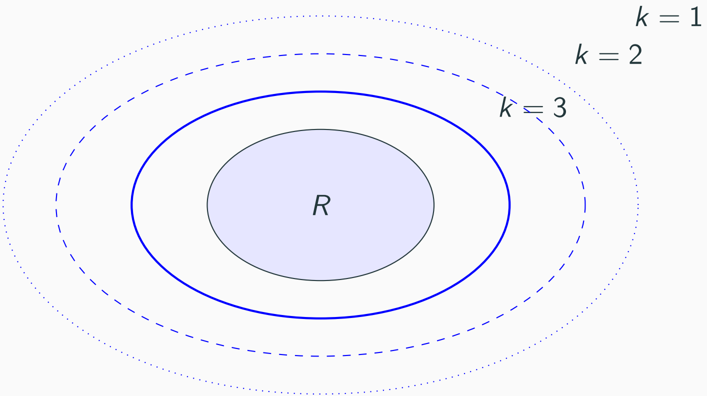
Inductive Step:

$$\begin{aligned} & P(s_0) \wedge \cdots \wedge P(s_{k-1}) \wedge \\ & T(s_0, s_1) \wedge \cdots \wedge T(s_{k-1}, s_k) \Rightarrow P(s_k) \end{aligned}$$

Effect of Increasing k

- K-induction checks property $P(s)$ over all k-step execution paths.
- Larger k leads to:
  - Stronger inductive hypothesis
  - Tighter approximation of reachable states
  - More spurious counterexamples eliminated

Effect of Increasing k (Cont.)



Over-approximation gets smaller

Summary

Larger k stronger assumption smaller over-approximation more accurate proof.

What is Raft

What is Distributed Consensus?

- Distributed consensus ensures that multiple nodes in a distributed system agree on a common value despite failures.
- **Challenges:**
  - Network partitions and asynchrony.
  - Node failures and leader crashes.
  - Ensuring consistency while allowing availability.
- **Common use cases:**
  - Replicated state machines (e.g., distributed databases, key-value stores).
  - Coordination services (e.g., Zookeeper, etcd, Consul).

Introduction to Raft

Raft is a consensus algorithm designed to be:

- **Understandable** – clarity over complexity.
- **Consistent** – all nodes agree on log contents.
- **Fault-tolerant** – tolerates failures of minority nodes.

Its goal is to ensure distributed state machines apply the same sequence of commands, even in the presence of failures.
Doesn't prevent against Byzantine failures. Only server death, network timeouts, message drops etc.

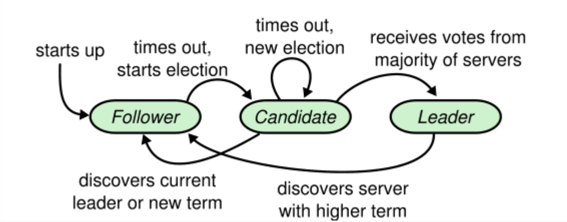
Introduction to Raft

- Raft decomposes the consensus problem into three subproblems:
  1. **Leader Election**: Choosing a single leader to manage the log.
  2. **Log Replication**: Ensuring logs on all servers are consistent.
  3. **Safety**: Maintaining the correctness of the log.
- Every change to the system state goes through the leader.

Raft Server States and Persistent State

- A Raft server can be in one of three states:
  - **Leader**: Handles all client interactions.
  - **Follower**: Passive state, responds to requests.
  - **Candidate**: Temporary state to initiate elections.

Overview of Leader Election Process



- Process:
  - Timeout → Candidate → RequestVote → Majority → Leader → Leader sends heartbeats
- Guarantees:
  - **Election Safety : Unique leader per term**

What is a Term?

- Raft divides time into periods called **terms**.
- Each term begins with a new election.
- Terms are numbered with monotonically increasing integers.
- `currentTerm` is the highest term a server has seen.
    - Used to detect stale leaders and RPCs.
- If a server sees a term higher than its current term, it updates its `currentTerm` and becomes a follower.

Triggering an Election

- Followers start an election after election timeout.
- Transition steps:
  - Increment `currentTerm`
  - Become Candidate
  - Vote for self: `votedFor = self`
  - Send `RequestVote` RPCs to all servers

The RequestVote RPC Explained

- RPC format: `RequestVote(term, candidateId, lastLogIndex, lastLogTerm)`
- Vote granted if:
  1. `term` $\geq$ `currentTerm`
  2. Not voted yet in this term
  3. Candidate's log is at least as up-to-date:
     - `lastLogTerm > receiverLastLogTerm` or
     - `lastLogTerm == receiverLastLogTerm` and `lastLogIndex >= receiverLastLogIndex`
- On majority votes (> `N/2`), candidate becomes Leader.

Election Outcomes and Split Votes

- **Win**: Candidate gets majority and becomes Leader.
- **Lose**: Candidate receives any message with term $\geq$ `currentTerm`.
  - Reverts to Follower if term $\geq$ `currentTerm`.
- **Split Vote**: No majority due to simultaneous candidates.
  - Candidate times out and retries with incremented term.
  - **Randomized election timeouts** reduce split votes:
    - Each follower waits a random time before becoming a candidate.
    - Reduces collisions and increases chance of a clear winner.
  - Failed candidates restart election with a new timeout.

Election Safety Properties

- **Election Safety**: Only one leader elected per term
- Enforced by:
  - One vote per server per term
  - Majority requirement guarantees uniqueness
- No two leaders in same term possible

Election Timeline (Leader Crash and Recovery)

A is Leader (term = 3) → A crashes → B times out → Candidate (term = 4) → B gets votes from C, D → B becomes Leader

↓

Sends heartbeats to C, D, E → E updates to term 4 → A recovers → Receives heartbeat (term 4) → A steps down to Follower

Objectives

- Come up with a model for Leader Election Protocol and prove the correctness using K-Induction

- Inject the D-duplication (recounting of votes from a server) bug in the model and try to catch it efficiently using Bounded Model Checking.

Models with Different Levels of Abstraction

What is a reasonable abstract model to use that allows invariants proved for an abstract model to expedite BMC at a concrete implementation level?

Took about 50% of the time.

Trivial Model: Level 1 Abstraction

Key Idea: Starting from a very trivial highly non deterministic model, we can keep making it more concrete until we reach desired level of abstraction.

Model Components:

$$
\begin{aligned}
\text{Role} &= \{F, C, L\} \quad \text{(Follower, Candidate, Leader)} \\
\text{Procid} &= \{0, 1, \ldots, \text{Max}\} \\
\text{Term} &= \text{Positive Integers} \\
\text{State} &= \text{Node: [Procid]} \;\to \langle R : \text{Role}, t : \text{Term} \rangle
\end{aligned}
$$

Initial State:

$$\forall i \in \text{Procid}, \quad s[i].R = F \text{ and } s[i].t = 1$$

Transition Relation and LEP

Transition Relation $T(s, s')$:

$$\big(\forall i, j \in \text{Procid}, \quad (s'[i].R = L \text{ and } s[j].R = L) \Rightarrow i = j\big) \wedge \big(\forall i \in \text{Procid}, s[i].t \leq s'[i].t\big)$$

Leader Election Property (LEP):

$$LEP(s) = \forall i \neq j, \quad \neg(s[i].R = L \wedge s[j].R = L)$$

- The transition relation explicitly **enforces at most one leader** per state.
- Thus, **every reachable state satisfies LEP by construction**.
- Terms (time) are non-decreasing, ensuring monotonic progression.
- The model is **correct by construction**: it *only explores valid states and paths* that obey LEP.

Refinement and Existential Abstraction

Existential Abstraction:

- A concrete system CS refines an abstract system AS if every concrete transition maps to an abstract transition:

$$\forall cs_i, cs_{i+1}, \quad T^\wedge(cs_i, cs_{i+1}) \implies T(A(cs_i), A(cs_{i+1}))$$

- Meaning: every concrete trace is representable as a trace in the abstract model.

Refinement Conditions:

- **Initialization:** $\forall cs,\ \mathrm{Init}(A(cs))$
- **Simulation:** $\forall i,\ T^\wedge(cs_i, cs_{i+1}) \implies T(A(cs_i), A(cs_{i+1}))$
- **Conformance:** $\forall cs, cs',\ T^\wedge(cs, cs') \implies T(A(cs), A(cs'))$

Criteria for Abstract Models

1. **Sufficient State Information:** Captures enough detail to exhibit safety violations and inductive invariants.
2. **Existential Abstraction:** Does not exclude any correct implementation behavior; uses nondeterminism for abstraction.
3. **Non-Inductive Safety Property:** Preferably, the safety property itself is not inductive, reducing invariant complexity.
4. **Simplicity:** Small and abstract enough to ease verification effort.

Level 2 : Less Abstract Model

- Abstracts away network and messages
- Retains roles, term, votes to enable safety proofs
- Goal: Prove Leader Election Property (LEP)

Model Components:

- `Role = {F, C, L}` (Follower, Candidate, Leader)
- `Procid = {0,1,..,Max}`
- `Term = PosInt`, `Quorum = bool`
- `Votes: [Procid]` $\rightarrow$ `Nat` tracks votes received from each node
- `State: Node = [Procid]` $\rightarrow$ $\langle$`Role, Term, ms: Votes, q: Quorum`$\rangle$

Defining LEP and Invariants

Leader Election Property (LEP):

$$\forall i \neq j : \quad \neg(s[i].R = L \land s[j].R = L)$$

$$s[i].R = L \Rightarrow \mathtt{ucard}(\{j \mid s[i].ms[j] > 0\}) \geq \mathtt{Max}/2 + 1$$

Supporting Invariants:

- **UniqueQ**: At most one node has quorum

$$\forall i \neq j : \neg(s[i].q \land s[j].q)$$

- **MajQ**: Quorum only if node received majority

$$s[i].q \Leftrightarrow \mathtt{ucard}(\{j \mid s[i].ms[j] > 0\}) \geq \mathtt{Max}/2 + 1$$

- **UniqueVote**: No double voting in a term

$$s[i].ms[p] > 0 \land s[j].ms[p] > 0 \Rightarrow i = j$$

Transition System: T(s, s')

Initial State:

$$\forall i : \quad s[i].R = F,\ s[i].t = 1,\ \forall j :\ s[i].ms[j] = 0$$

Transitions:

$$\forall i :\ s[i].t \leq s'[i].t \qquad \text{(Term never decreases)}$$

$$s[i].R = C \land \neg s[i].q \land s'[i].q \Rightarrow s'[i].R = L \qquad \text{(Promotion to Leader)}$$

$$s'[i].q \iff \left|\{j \mid s'[i].ms[j] > 0\}\right| \geq \left\lfloor \frac{\text{Max}}{2} \right\rfloor + 1 \qquad \text{(Quorum rule)}$$

$$\forall j \in \text{Procid} :\ s[i].ms[j] \leq s'[i].ms[j] \qquad \text{(Vote monotonicity)}$$

$$\forall j, k \in \text{Procid} :\ (s'[j].ms[i] > 0 \land\ s'[k].ms[i] > 0) \Rightarrow j = k \qquad \text{(Unique Vote)}$$

Invariants built into transition:

- $\texttt{UniqueQ}(s')$
- $\texttt{MajQ}(s')$
- $\texttt{UniqueVote}(s')$

LEP-Safe Abstraction for Verification

- Abstraction function `A(cs)` maps concrete state to abstract state
- Concrete transition: $T_{cs}(cs_0, cs_1)$
- BMC Check:

$$T_{cs}(cs_0, cs_1) \wedge \cdots \wedge \neg(LEP(A(cs)) \wedge UniqueQ \wedge MajQ \wedge \ldots)$$

- Ensures abstraction preserves all allowed behaviors
- Guarantees no two leaders in one term with quorum

Using Abstraction to Verify LEP

Verification via BMC:

$$T_{cs}(cs_0, cs_1) \wedge \cdots \wedge T_{cs}(cs_k, cs_{k+1})$$
$$\Rightarrow \neg[LEP(A(cs)) \wedge UniqueQ(A(cs)) \wedge MajQ(A(cs)) \wedge UniqueVote(A(cs))]$$

Advantage: Catch violations of LEP early in BMC using abstract inductive invariants.

Alternate Strategy: Use more refined models (closer to implementation) to discover deeper inductive invariants, enabling better debugging.

Abstraction Function: Level 2 to Level 1

State Mapping:

$$A(\text{State\_L2}) = \langle R, t \rangle \quad \text{(Role, Term)}$$

where ‘Votes’ and ‘Quorum’ are abstracted away.

Transition Mapping:

$$A(T_{\text{L2}}) = T_{\text{L1}} \quad \text{(Only Role and Term transitions considered)}$$

Key Property Preserved:

$$A(\text{LEP}) \quad \text{(LEP is preserved)}$$

Level 3 : Concrete Model

- We model Raft as a distributed system with:
  - Nodes: $\mathcal{N} = \{n_1, \ldots, n_k\}$
  - Message queue (network module): $\mathcal{M}$
- Messages: $m = \langle \text{type}, \text{payload}, s, t_s, r \rangle$
  - s = sender, t_s = sender's term, r = receiver
- All message sending and delivery are nondeterministic
- Network may delay messages arbitrarily but is fair

Node Local State

Each node $n \in \mathcal{N}$ maintains:

$$
\begin{aligned}
\text{state}_n = \langle & \text{term}_n \in \mathbb{N}, \\
& \text{role}_n \in \{\text{Follower}, \text{Candidate}, \text{Leader}\}, \\
& \text{log}_n : \text{List of } (\text{index}, \text{term}, \text{command}), \\
& \text{votedFor}_n \in \mathcal{N} \cup \{\perp\}, \\
& \text{timeout}_n \in \mathbb{N}, \\
& \text{Votes Received}_n \in \mathbb{N}, \\
& \text{inbox}_n \subseteq \mathcal{M} \qquad \text{outbox}_n \subseteq \mathcal{M} \rangle
\end{aligned}
$$

- Node behavior is defined by transition rules reacting to inbox messages and timeouts

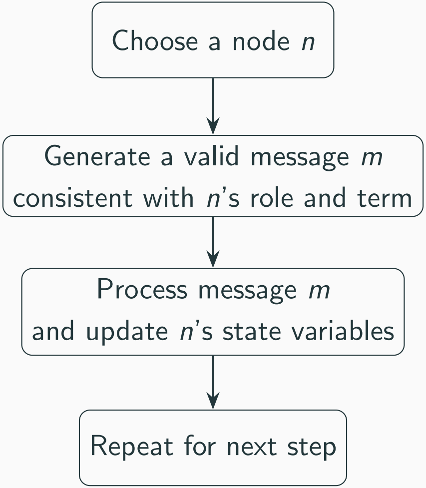
Level 2.5: Working model

- The explicit network module $\mathcal{M}$ is removed.
- Message queues are no longer modeled explicitly or globally.
- Instead, **messages are generated nondeterministically** and directly placed in node inboxes.

Execution semantics:

- At each transition:
  - A node n' is selected nondeterministically.
  - A valid message m (e.g., `RequestVote`, `AppendEntries`) is nondeterministically generated based on the current system state.
  - The message m is assumed to appear in the inbox of n'.
  - Node n' processes m and updates its local state accordingly.

Message Generation and Processing Flow



Node State

Each node n has:

- $n.\text{term} \in \mathbb{N}$: Current term
- $n.\text{role} \in \{\text{Follower}, \text{Candidate}, \text{Leader}\}$
- $n.\text{votedFor} \in \mathbb{N} \cup \{\perp\}$
- $n.\text{votesReceived} = [\text{votes}]$ (Number of votes received by each other node)

Global set: $\text{Terms} = \{n.\text{term} \mid n \in \text{Nodes}\}$

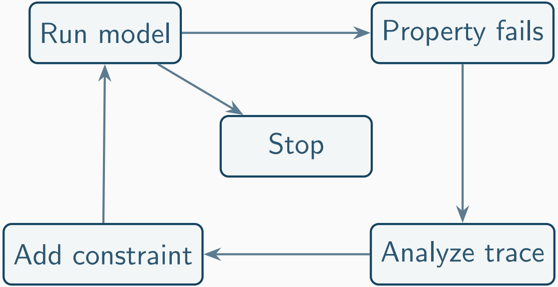
Message Schema

Messages are tuples (M, sender, t) where:

- $M \in \{\text{Hb}, \text{Vr}, \text{Vote}\}$ (Heartbeat, VoteRequest, VoteGrant)
- t is the sender's term

No queueing or delivery delays – messages exist virtually and must be justified by current state.

Inbox Constraint Refinement Process



- Start with minimal constraints (arbitrary state)
- Analyze counterexample trace and add constraints to rule out unrealistic behaviors
- Repeat until non-buggy model passes via K-Induction and buggy model fails

Transition Rules (1): Election

Election Timeout:

- If $\text{role}_n = \text{Follower}$ and $\text{timeout}_n = 0$
- Then:
  - $\text{term}_n := \text{term}_n + 1$, $\text{role}_n := \text{Candidate}$
  - $\text{votedFor}_n := n$
  - For all $r \neq n$, enqueue:

$$\langle \text{RequestVote}, \text{payload}, n, \text{term}_n, r \rangle \in \mathcal{M}$$

Receive RequestVote:

- If $t_s > \text{term}_r$, update term and become Follower
- Grant vote if log is up-to-date and not already voted

Transition Rules (2): Leader Election

Become Leader:

- Candidate c receives majority of VoteResponse(granted=True)
- Then: $\text{role}_c := \text{Leader}$
- Broadcast empty AppendEntries to all $r \neq c$

Receiving AppendEntries:

- If $t_s \geq \text{term}_r$, accept, update term and become Follower
- Else, reject

Abstraction Function from this model to Level 2 model

Maps concrete state to abstract state:

- `R`: `FOLLOWER` $\rightarrow$ `F`, `CANDIDATE` $\rightarrow$ `C`, `LEADER` $\rightarrow$ `L`
- `t`: `currentTerm + 1`
- `ms[i][j]`: 1 if node *i* received a vote from *j*, else 0
- `q`: True if majority votes received

Preserving Invariants

AbstractionPreservesInvariants checks:

- Term Monotonicity: $t_i \leq t'_i$
- Vote Monotonicity: $ms[i][j] \leq ms'[i][j]$
- Leader Promotion: Candidate with quorum becomes leader
- Quorum Validity: $q \Leftrightarrow$ majority received
- Unique Vote: A node can't vote twice in a term

Abstract Properties from Concrete

- **LEP**: No two leaders simultaneously
- **UniqueQ**: Only one node has quorum
- **MajQ**: Quorum $\Leftrightarrow$ majority of TRUE votes
- **UniqueVote**: No node receives multiple votes from same peer

Targeted Bugs

Common Bugs in Raft Implementations

- Despite Raft's design for understandability, implementations still have subtle bugs
- Several categories of bugs identified by Colin Scott et al.:
  - Vote counting issues
  - Term handling inconsistencies
  - Log indexing problems
  - Recovery management
- We focus on the duplicate vote bug (raft-45)

Duplicate Vote Bug (raft-45)

"Candidates accept duplicate votes from the same follower in the same election term. (A follower might resend votes because it believed that an earlier vote was dropped by the network). Upon receiving the duplicate vote, the candidate counts it as a new vote and steps up to leader before it actually achieved a quorum of votes."

- Results in violation of 'Leader Safety'
- Two leaders can be elected in the same term
- Can lead to data inconsistency and linearizability violations

Modeling Duplicate Votes

- We implemented two vote counters in our model:
  - `true_votes`: Counts unique voters (correct behavior)
  - `fake_votes`: Counts total votes including duplicates (buggy behavior)
- Toggle with compilation flag: `-DINJECT_DUPLICATE_VOTE_BUG`
- With bug flag, candidate becomes leader based on `fake_votes`
- Without bug flag, uses `true_votes` for leader election

Verification Properties

Safety Properties for Raft

- **check_safety_property():** No two leaders in the same term
  - $\forall i, j \in \text{Nodes}, i \neq j :$
    $\neg(s[i].\text{role} = \text{LEADER} \wedge s[j].\text{role} = \text{LEADER} \wedge s[i].\text{term} = s[j].\text{term})$
- **check_leader():** Leaders must have quorum of votes
  - $\forall i \in \text{Nodes} : s[i].\text{role} = \text{LEADER} \Rightarrow s[i].\text{true\_votes} \geq \frac{N}{2} + 1$
- **check_unique_vote():** No node votes twice in the same term
  - $\forall i, j \in \text{Nodes}, i \neq j, s[i].\text{term} = s[j].\text{term}, \forall k \in \text{Nodes} :$
    $\neg(s[i].\text{votes\_received}[k] > 0 \wedge s[j].\text{votes\_received}[k] > 0)$
- **check_unique_quorum():** Only one node has majority in same term
  - $\forall i, j \in \text{Nodes}, i \neq j, s[i].\text{term} = s[j].\text{term} :$
    $\neg(s[i].\text{true\_votes} \geq \frac{N}{2} + 1 \wedge s[j].\text{true\_votes} \geq \frac{N}{2} + 1)$

Checking Model with CBMC

- CBMC: C Bounded Model Checker
- Takes C program and unrolls loops to bounded depth
- Converts to SAT formula and checks for satisfiability
- We use it with following flags:
  - `--property main.assertion.X`: Specify property to check
  - `--trace-hex`: Output trace when property violated
  - `-DINJECT_DUPLICATE_VOTE_BUG`: Enable bug
  - `-DUSE_FIXED_INIT`: Use fixed initialization

Experimentation and Results

Impossible to Find Violation with N=3

- With N=3, quorum size is 2 nodes
- For violation, we need:
  - Node A becomes leader with duplicate votes
  - At least one vote must be duplicate (from B)
  - This means B already voted for A
  - Quorum is 2, so A has votes from itself and B
  - This is a legitimate quorum of unique votes!
- Duplicate vote bug cannot cause safety violation when N=3
- Requires at least N=4 to demonstrate the bug

Minimum Steps to Failure Analysis

For N=2x or N=2x+1, we need at least 2x+4 steps to find a safety violation:

1. Node A times out, becomes candidate, votes for itself.
2. Node B receives a vote request in its inbox and changes it's votedFor value.
3. Node A receives vote grants from Node B, x times, reaching x+1 votes and becoming the leader

Node A takes (x+2) steps to do it and so does Node C, requiring a total of 2x+4 steps to reach

Total steps: $(x+2) + (x+2) = 2x+4$

Experiments with N = 4 Nodes

Assumptions at Each Step	Asserted Property	Violation at K	Time (s)
`P`	`P`	8	210.75
`UQ, UV, L`	`P`	8	181.05
`P, UQ`	`P`	8	174.72
`P, UV`	`P`	8	162.49
`P, L`	`P`	8	190.97

Legend

`P` = safety property

`UQ` = unique quorum

`UV` = unique votes

`L` = leader uniqueness

Invariant Violations

Assumptions at Each Step	Asserted Invariant	Violation at K	Time (s)
`UQ`	`UQ`	8	222.06
`UV`	`UV`	4	80.83
`L`	`L`	5	83.25

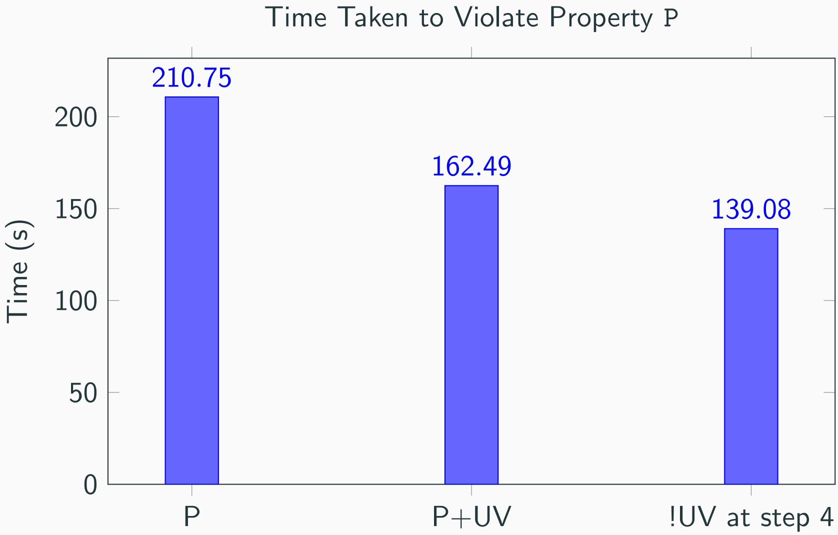
Violate Invariant Mid-Run

Injected Violation at Step	Asserted Property	Violation at K	Time (s)
`!UV at step 4`	`P`	8	139.08
`!L at step 5`	`P`	8	171.33
`!UQ at step 8`	`P`	8	165.11

Improvement over:

Assumptions at Each Step	Asserted Property	Violation at K	Time (s)
`P`	`P`	8	210.75
`P, UV`	`P`	8	162.49

Performance Comparison



Experiments with $N = 6$ Nodes

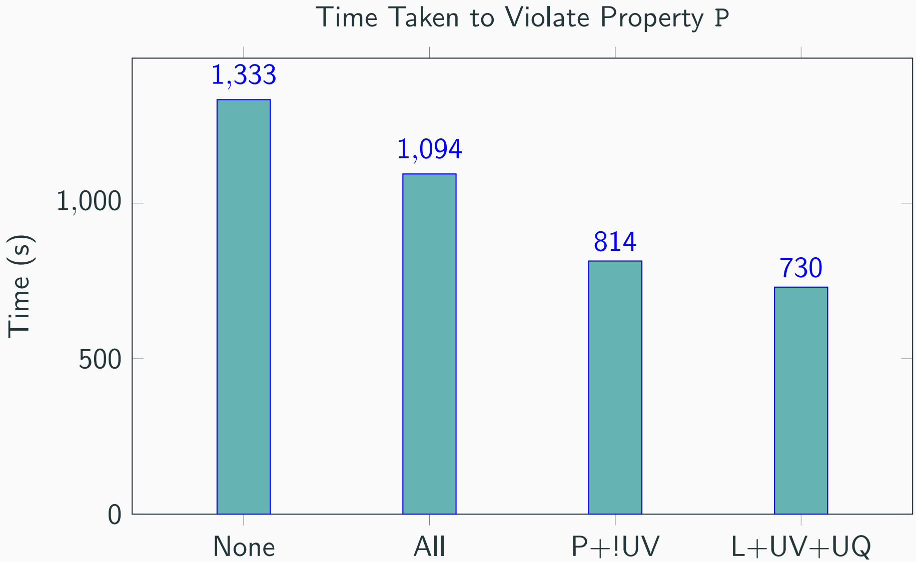
Assumptions at Step 9	Asserted Property	Violation at K	Time (s)
None	`P`	10	1333.11
All	`P`	10	1094.26
`P+!UV`	`P`	10	814.00
`L, UV, UQ`	`P`	10	730.69

Legend

`!UV` = vote conflict injected

All = all helper invariants assumed at step 9

Performance Comparison: All Assumptions



Key Insights

- Violations to the safety property are often preceded by violations to helper invariants.
- Selectively assuming helper properties can guide the model checker more effectively.
- Injecting violations at specific points can accelerate counterexample discovery.
- Larger models ($N = 6$) are more sensitive to which invariants are assumed and when.

Key Findings

- The duplicate vote bug can be successfully detected with BMC
- Minimum cluster size to demonstrate bug is N=4
- Verification time grows significantly with cluster size
- Judicious selection of invariants can improve performance:
    - Unique quorum check helps most for N=4
    - For N=6, overconstraining can increase verification time
- Trade-off: Tighter constraints reduce state space but increase formula complexity

Conclusion

- Demonstrated formal verification of Raft leader election
- Successfully modeled and detected duplicate vote bug
- Established minimum bounds for:
  - Cluster size required to exhibit bug (N=4)
  - Steps required to reach violation (2x+4 for N=2x)
- Identified critical invariants for efficient verification
- Iterative constraint refinement process effective for debugging

Q & A

Questions?

Email: {shobhits, akhoury}@cmi.ac.in

Acknowledgements

Thank You!