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

A case study applied to Raft Consensus Protocol

by

Shobhit

Advisor: **M Praveen and M.K. Srivas**

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science

Chennai Mathematical Institute, India

February 25, 2026

Acknowledgments

This project was carried out in collaboration with Akroury Shauryam, whose partnership and unwavering support were invaluable at every stage of this work. His technical insights and constant encouragement made this project possible.

I express my deepest gratitude to my supervisors, Prof. M.K. Srivas and Prof. M. Praveen, for their invaluable guidance, support, and insightful feedback throughout this research. Their expertise in Logic and Formal Methods has been instrumental in shaping this thesis. The courses and discussions under their mentorship have significantly deepened my understanding and interest in the field, laying the foundation for this work.

I am also grateful to Aditi, Abhinav, Ishaan, Monsey, Sampriti, Katha, Ketaki, Jessica, Sayan and other friends I probably forgot to mention for providing much-needed moments of relief during stressful times—especially through countless games of Cabo and Geoguessr. I owe special thanks to Shauryam for ensuring we never missed a meal during long nights of work and my parents for supporting me.

Special thanks to Alok Laddha for keeping me sane throughout my time in the institute.

I am sincerely thankful to Chennai Mathematical Institute for fostering an intellectually vibrant environment that allowed me to pursue my academic interests freely. The stimulating interactions with faculty and peers have profoundly enriched my learning experience.

Abstract

This thesis investigates the application of inductive invariants to enhance the efficiency of Bounded Model Checking (BMC) for distributed consensus protocols, with particular focus on the Raft leader election algorithm.

The research addresses the fundamental challenge of state space explosion in formal verification of distributed systems. While BMC effectively identifies bugs in shallow execution traces, it encounters significant limitations when verifying deeper properties due to exponential growth in formula complexity. We propose that strategically crafted inductive invariants can serve as "trip-wires" to detect safety violations at earlier stages, thereby reducing verification time and improving overall performance.

We developed abstract models of Raft leader election with varying levels of abstraction and implemented the duplicate vote bug (raft-45) to evaluate detection capabilities. Our systematic experimental evaluation using CBMC demonstrates that strategic invariant selection yields substantial improvements in BMC performance: 23% improvement for 4-node clusters and up to 45% improvement for 6-node clusters.

Key Contributions:

- A systematic methodology for creating verification-friendly abstractions of consensus protocols
- Identification of critical invariants (unique vote, unique quorum, leader validity) that accelerate bug detection
- Empirical evidence that invariant-guided BMC effectively handles realistic distributed system models
- Establishment of minimum bounds: duplicate vote bug requires at least N=4 nodes and $2\lceil N/2 \rceil + 2$ execution steps

The work provides valuable insights into the trade-offs between constraint tightness and verification complexity, demonstrating that violations to safety properties are often preceded by violations to helper invariants, enabling earlier and more efficient bug detection.

Keywords: Bounded Model Checking, Inductive Invariants, Distributed Consensus, Raft Protocol, Formal Verification, State Space Explosion

Contents

Chapter 1

Motivation

1.1 Model Checking Approaches

Model checking is a widely used technique to automatically verify whether a system satisfies a given property. Among the most popular approaches are **Symbolic Model Checking (SMC)**, **k-Induction**, and **Bounded Model Checking (BMC)**—each with its trade-offs.

- **Symbolic Model Checking (SMC)** explores the *entire state space* using symbolic encodings (typically BDDs or SAT/SMT formulas). It guarantees full coverage and is capable of proving safety properties completely. However, this comes at the cost of *state explosion*, making it infeasible for large or infinite-state systems.
- **k-Induction** generalizes induction to transition systems: it attempts to prove that a property holds in all reachable states by checking that it holds in the first k steps (base case) and is preserved from steps k to $k + 1$ (inductive step). If successful for small k, it gives a full proof. Unfortunately, many real-world systems require large values of k, making k-induction expensive or infeasible.
- **Bounded Model Checking (BMC)** unrolls the system for a fixed number of steps k and checks for a counterexample to the safety property within those steps. It is highly effective in finding *shallow bugs*, especially with SAT/SMT solvers. However, BMC is incomplete unless k exceeds the depth of the bug, which is often unknown in advance.

1.2 Bounded Model Checking in Practice

Bounded Model Checking is widely used in industry because of its ability to catch bugs early with minimal user guidance. Tools like `CBMC`, `ESBMC`, and hardware verification frameworks often rely on BMC engines for their core checking. BMC's appeal lies in its **automation**, **speed**, and **low setup cost**. Engineers do not need to write inductive invariants manually—unrolling the system and checking safety assertions is often enough.

However, BMC's efficiency drops when bugs lie deep in the execution or when we attempt to *prove* properties (i.e., show the *absence* of bugs). In such cases, the tool may need to unroll hundreds or thousands of steps, significantly increasing solver time and memory use.

1.3 A Motivating Example

Consider the following toy system with 8 registers `v[0..7]`, a step counter `i`, and a cycle constant `N`:

```
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)
```

This program performs a left shift of the register values every iteration. In the correct version, `v[7]` receives `v[0]`, maintaining the cycle. In the buggy version, however, `v[7]` is overwritten non-deterministically using `choose(v[0], v[7])`.

We want to check the safety invariant:

> **Whenever** $i \bmod 8 = 0$, **then** the registers contain exactly the values 0 through 7 (in the original order).

To detect this bug using standard BMC, we would need to **unroll at least 8 steps**—one full cycle—before the corruption becomes visible. This highlights a key limitation: **deep bugs require deep unrolling**, which in turn increases verification time.

1.4 Our Idea: Faster BMC via Invariants

We propose a technique to make BMC faster and more scalable by adding **auxiliary inductive invariants** that act as early "trip-wires."

Let:

- $Inv_1, \ldots, Inv_m$ be invariants such that each $Inv_j \Rightarrow Safety$.
- If any Inv_j is violated early, it implies that the main safety property will eventually fail.

Why this helps

- Each invariant provides an **additional check** that may fail earlier than the main property.
- These act as *trip-wires*—failing one gives us early evidence of a violation.
- More trip-wires $\Rightarrow$ smaller required unwind bound k $\Rightarrow$ **reduced solver runtime**.
- Even if not all invariants are inductive, failing them often correlates strongly with system errors.

Applying the idea to our example

We add the auxiliary invariant:

Distinct(`v[0]`, . . . , `v[7]`)

This property holds in the initial state (values are 0 to 7) and is **cheap to check**. In the correct version of the code, it is preserved throughout. But in the buggy version, the non-deterministic overwrite (`choose(v[0], v[7])`) breaks this immediately, possibly even in the **first unrolling step**.

Result

- We do **not** need to unroll 8 steps.
- The auxiliary invariant fails early, letting the solver terminate faster.

- We save time and avoid deeper unrollings without losing soundness.

This simple idea—injecting inductive invariants that imply safety but fail earlier—can significantly **improve the performance of BMC**, especially for properties with delayed violations.

Chapter 2

Problem Overview and Approach

2.1 Problem Statement

The verification of distributed protocols remains one of the most challenging tasks in the domain of formal methods. While Bounded Model Checking (BMC) has emerged as a promising technique for automatically finding counterexamples to safety properties, its applicability to realistic distributed systems is fundamentally limited by the problem of *state explosion*. The intricate interplay of concurrency, non-deterministic message delivery, and timing behavior leads to a rapidly growing state space, making exhaustive exploration computationally infeasible.

These challenges are particularly evident in the formal verification of consensus protocols, which form the backbone of modern fault-tolerant systems. In this project, we focus on the Leader Election Phase (LEP) of the Raft consensus protocol—a widely adopted algorithm known for its relative simplicity and strong safety guarantees.

Our central hypothesis is that the introduction of carefully crafted *auxiliary inductive invariants* can mitigate the scalability limitations of BMC. These invariants are properties that hold throughout all reachable states of the system and logically imply the target safety property. Crucially, they often fail *earlier* than the global safety condition, guiding the model checker toward counterexamples more efficiently and reducing the unrolling bound k required in k-induction–based verification.

2.2 Proposed Approach

To evaluate this hypothesis, we apply our methodology to the Raft LEP through the following systematic approach:

2.2.1 Abstract Modeling of Raft Leader Election

The first step involves constructing an *abstract yet faithful* model of Raft's Leader Election Phase. A good abstract model, in this context, satisfies the following criteria:

- It captures the essential semantics of Raft's leader election process, including node roles, term progression, and voting behavior.
- It abstracts away low-level implementation details and irrelevant system behaviors that do not influence the property under verification.
- It remains conservative, meaning that it over-approximates the set of possible behaviors rather than excluding valid executions, thus preserving soundness.
- It is executable and compatible with automated model checkers such as NuSMV or CBMC.

Our model strikes a careful balance between precision and tractability, preserving the protocol's voting logic and state transitions while avoiding state-space blow-up.

2.2.2 Model Correctness and Baseline Verification

Once the abstract model is established, we first verify its correctness under ideal conditions, i.e., in the absence of protocol violations. The global safety property of interest is:

Election Safety: At most one leader is elected per term.

This ensures that the fundamental safety guarantee of Raft is respected, preventing split-brain scenarios and inconsistent system state.

The successful verification of the model in this baseline case serves two purposes: it validates the fidelity of our abstraction and establishes a performance baseline for subsequent experiments.

2.2.3 Bug Injection: Deliberate Violation of Voting Behavior

To realistically assess the ability of auxiliary invariants to aid in bug detection, we introduce a known subtle flaw into the model—the *duplicate vote bug*. This bug, documented in real-world Raft implementations, allows a follower to cast multiple votes in the same election term, typically due to message retransmissions in unreliable networks.

The presence of this bug violates the election safety property by enabling two nodes to simultaneously believe they have obtained majority support, potentially resulting in multiple leaders within the same term.

2.2.4 Verification Strategy and Invariant Design

With the buggy model in place, we conduct two sets of verification experiments:

1. **Baseline BMC:** We attempt to discover the violation using only the global election safety property as the target invariant.
2. **Invariant-Aided BMC:** We incorporate a suite of auxiliary inductive invariants designed to logically imply the safety property and fail earlier in erroneous executions.

The auxiliary invariants considered include:

- **Unique Vote Invariant:** No node casts more than one vote per term.
- **Unique Quorum Invariant:** At most one node can obtain a majority of votes in any term.
- **Leader Uniqueness Invariant:** If two nodes are in the leader state, their terms must differ.

These invariants are carefully crafted to be inductive under the system's transition relation, ensuring their soundness in guiding the verification process.

2.2.5 Empirical Comparison

For both verification strategies, we collect and compare the following quantitative metrics:

- The **maximum unrolling bound** k required to detect a violation.
- The total number of **SAT solver calls**, reflecting computational effort.
- The overall **verification runtime**, providing a practical measure of efficiency.

This empirical evaluation allows us to assess the extent to which auxiliary invariants improve the scalability and bug-detection capabilities of BMC in realistic distributed protocol models.

Chapter 3

What is Raft

3.1 Distributed Consensus: The Foundational Problem

Distributed consensus refers to the problem of ensuring that multiple nodes in a distributed system reliably agree on a common value or sequence of operations, despite failures and unreliable communication. This problem is foundational to the correctness of databases, coordination services, distributed filesystems, and blockchain platforms.

In the absence of consensus, nodes may diverge in state, leading to inconsistencies, loss of data, or systemic failure.

A correct consensus protocol must satisfy the following properties:

- **Agreement**: All non-faulty nodes eventually decide on the same value.
- **Validity**: The chosen value was actually proposed by a node.
- **Termination**: All non-faulty nodes eventually reach a decision.

In realistic environments, consensus protocols must operate correctly under the following adverse conditions:

- Network partitions, message delays, and message loss.
- Crash-recovery failures, where nodes stop and subsequently restart.
- Asynchronous communication without global clocks.

Achieving consensus under these conditions, without compromising system liveness or safety, represents a fundamental challenge in distributed computing.

3.2 Introduction to Raft

Raft is a leader-based consensus protocol introduced by Diego Ongaro and John Ousterhout in 2013. It was explicitly designed to provide the same fault-tolerance and safety guarantees as Paxos, while significantly improving understandability and implementability.

Raft operates in the *crash-recovery model*, tolerating node failures and unreliable communication, but does not address Byzantine failures, where nodes behave arbitrarily or maliciously.

The protocol decomposes the consensus problem into three logically distinct subcomponents:

1. **Leader Election**: Dynamically selecting a unique leader to coordinate client requests and log replication.
2. **Log Replication**: Ensuring all nodes maintain identical logs of commands to be executed by their state machines.
3. **Safety Mechanisms**: Enforcing protocol invariants that guarantee consistency and correctness, even in the presence of failures.

These components collectively guarantee that all non-faulty nodes maintain a consistent state, progress is made in the presence of failures, and previously agreed-upon operations are never lost.

3.3 Raft Server Roles and State

Each node (or server) in a Raft cluster assumes one of three possible roles at any given time:

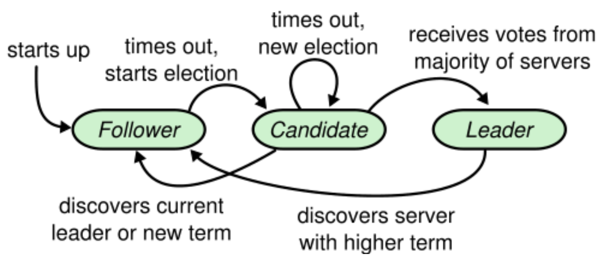
- **Follower**: A passive node that responds to RPCs from other nodes but does not initiate actions.
- **Candidate**: A node that initiates an election upon suspecting leader failure.
- **Leader**: The node responsible for handling client requests, managing log replication, and maintaining cluster consistency.

These roles are transient, with nodes transitioning between them based on timeouts, term comparisons, and received messages.

Each node maintains persistent state variables critical for correctness:

- `currentTerm`: The latest term observed by the node.
- `votedFor`: The candidate that received the node's vote in the current term.
- `log[]`: The sequence of commands to be applied to the replicated state machine.

3.4 Leader Election in Raft


starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term

Leader election is the process by which a unique node is selected to coordinate the cluster. This process is triggered when a follower does not receive heartbeats from a leader within its election timeout period.

Raft employs randomized election timeouts to reduce the probability of simultaneous candidacies. When the timeout elapses, a follower becomes a candidate, increments its term, votes for itself, and sends `RequestVote` RPCs to all other nodes.

A node becomes leader upon receiving votes from a majority of the nodes. If no candidate achieves a majority (e.g., due to a split vote), the process repeats with new randomized timeouts.

The election process satisfies the following critical safety conditions:

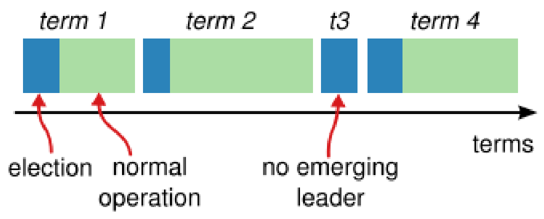
- Each node grants at most one vote per term.
- A candidate must obtain votes from a strict majority ($\lceil N/2 \rceil$) of nodes.

The requirement for intersecting majorities ensures that no two nodes can simultaneously become leaders in the same term, thus preserving *Election Safety*:

> *At most one leader may be elected in a given term.*

This property is foundational for the consistency of Raft, as it guarantees a single authoritative source of new log entries.

3.5 Terms and Time in Raft



Raft structures time into logical epochs known as **terms**. Each term begins with an election and may conclude with a stable leader successfully replicating commands.

Terms are essential for distinguishing newer information from obsolete state. All RPCs include the sender's current term, and nodes update their own term upon observing a higher term in received messages.

This mechanism ensures that outdated leaders and candidates relinquish authority in favor of more recent, valid terms.

3.6 Voting and the RequestVote RPC

To solicit votes, a candidate sends `RequestVote` RPCs containing:

- The candidate's current term.
- The candidate's unique identifier.
- The index and term of the candidate's last log entry.

A follower grants its vote if the following conditions are met:

- The candidate's term is at least as large as the follower's current term.
- The follower has not yet voted in the current term.
- The candidate's log is at least as up-to-date as the follower's log.

The log up-to-dateness check prevents the election of candidates with stale logs, thereby preserving log consistency.

3.7 Election Outcomes and Failover

An election concludes with one of the following outcomes:

- **Successful Election**: A candidate receives a majority of votes and becomes the leader.
- **Election Failure**: A candidate observes a higher term or is outvoted.
- **Split Vote**: No candidate secures a majority, triggering a new election.

Randomized election timeouts significantly reduce the likelihood of persistent split votes. Upon leader failure, followers independently detect the absence of heartbeats and initiate new elections, ensuring liveness under failure conditions.

3.8 Log Replication and the Log Matching Property

Once elected, the leader accepts client commands and appends them as log entries. These entries are replicated to followers via `AppendEntries` RPCs, which include consistency checks based on the preceding log index and term.

Followers reject RPCs that violate log consistency, ensuring the **Log Matching Property**:

> *If two logs contain an entry with the same index and term, all preceding entries in both logs are identical.*

This invariant guarantees that committed log entries are consistent across all nodes, even in the presence of failures and leader changes.

3.9 Raft's Safety Guarantees

In addition to election safety and log matching, Raft enforces the following formal safety properties:

- **Leader Completeness**: If a log entry is committed in a term, it will appear in the logs of all future leaders.
- **State Machine Safety**: If a server applies a log entry at a given index to its state machine, no other server applies a different entry at that index.

These guarantees ensure that once a command is committed, it is never lost, reordered, or overwritten, providing strong consistency for the replicated state machine.

Chapter 4

Model Development and Abstraction Levels

This chapter presents our approach to modeling the Raft Leader Election Protocol at varying levels of abstraction. We developed a hierarchy of models, beginning with highly abstract nondeterministic representations and progressively refining them to more concrete models that incorporate message-passing and network behaviors.

4.1 Abstraction Strategy and Criteria

A core challenge in the formal verification of distributed systems is choosing the right abstraction level. A useful abstraction must retain enough structure to encode meaningful properties while simplifying details that impede tractability.

4.1.1 Design Criteria for Abstract Models

Our abstract models were designed with the following criteria in mind:

1. **Sufficient State Information**: The model must include enough detail to support meaningful invariants and reveal potential safety violations.

2. **Existential Abstraction**: Use nondeterminism to abstract away low-level details while preserving all potential execution behaviors.

3. **Non-Trivial Safety Property**: The main safety property should not be trivially inductive, allowing the discovery of useful inductive invariants.

4. **Simplicity**: The model should remain small and easy to analyze, without sacrificing essential complexity.

4.1.2 Refinement and Simulation Relations

We use existential abstraction to relate models. A concrete system `CS` refines an abstract system `AS` under abstraction function `A` if:

$$\forall cs\_i, cs\_i+1 \in CS, \quad T\_CS(cs\_i, cs\_i+1) \Rightarrow T\_AS(A(cs\_i), A(cs\_i+1)) \tag{4.1}$$

This ensures that every concrete execution trace corresponds to a valid abstract trace, preserving properties verified on the abstract system in the concrete implementation.

4.2 Level 1: Toy Model

This model captures only high-level concepts such as node roles and terms. All implementation details are abstracted away.

4.2.1 Model Components

$$\text{Role} = F, C, L \tag{4.2}$$

$$\text{Procid} = 0, 1, \ldots, Max \tag{4.3}$$

$$\text{Term} = \mathbb{N}^{+} \tag{4.4}$$

$$\text{State} = \text{Procid} \rightarrow \langle R : \text{Role}, t : \text{Term} \rangle]$$

4.2.2 Initial State

All nodes begin as followers in term 1: $\forall i \in \text{Procid}, \quad s_0[i].R = F$ and $s_0[i].t = 1$

4.2.3 Transition Relation

- **At most one leader per state:** $\forall i, j, \ (s'[i].R = L \land s'[j].R = L) \Rightarrow i = j$
- **Term monotonicity:** $\forall i, \ s[i].t \leq s'[i].t$

4.2.4 Leader Election Property

$\text{LEP}(s) = \forall i \neq j, \ \neg(s[i].R = L \land s[j].R = L)$

4.2.5 Analysis

This model trivially satisfies the safety property due to hardcoded constraints. While it guarantees correctness, it provides limited insight into the mechanisms enforcing it and does not support discovery of helpful inductive invariants.

4.3 Level 2: Less Abstract Model

This model adds vote tracking mechanisms to make the safety property non-trivial and support invariant generation.

4.3.1 Extended Model Components

$$\text{Votes} = [\text{Procid} \to \mathbb{N}] \tag{4.5}$$
$$\text{Quorum} = \{\text{true}, \text{false}\} \tag{4.6}$$
$$\text{State} = [\text{Procid} \to \langle R, t, ms : \text{Votes}, q : \text{Quorum}\rangle] \tag{4.7}$$

4.3.2 Enhanced Leader Election Property

$$\forall i \neq j : \neg(s[i].R = L \wedge s[j].R = L)\wedge \tag{4.8}$$
$$\forall i : s[i].R = L \Rightarrow ucard(\{j \mid s[i].ms[j] > 0\}) \geq \lfloor \text{Max}/2 \rfloor + 1 \tag{4.9}$$

4.3.3 Supporting Invariants

- **Unique Quorum**: $\forall i \neq j : \neg(s[i].q \wedge s[j].q)$
- **Majority Quorum**: $s[i].q \Leftrightarrow |\{j \mid s[i].ms[j] > 0\}| \geq \lfloor Max/2 \rfloor + 1$
- **Unique Vote**: $\forall i \neq j, \forall p : s[i].ms[p] > 0 \wedge s[j].ms[p] > 0 \Rightarrow i = j$

4.3.4 Transition Constraints

- Term monotonicity: $\forall i,\ s[i].t \leq s'[i].t$
- Leader promotion: $s[i].R = C \wedge \neg s[i].q \wedge s'[i].q \Rightarrow s'[i].R = L$
- Quorum definition and vote monotonicity enforced explicitly

4.4 Level 3: Concrete Model with Explicit Network

This model explicitly models message queues and network behavior.

4.4.1 System Structure

- **Nodes:** $N = \{n_1, \ldots, n_k\}$
- **Messages:** $m = \langle \text{type}, \text{payload}, \text{sender}, \text{receiver}, \text{timestamp} \rangle$

4.4.2 Node State

Each node maintains:

$$term, role, votedFor, timeout, votesReceived, inbox, outbox \tag{4.10}$$

4.4.3 Message Types and Semantics

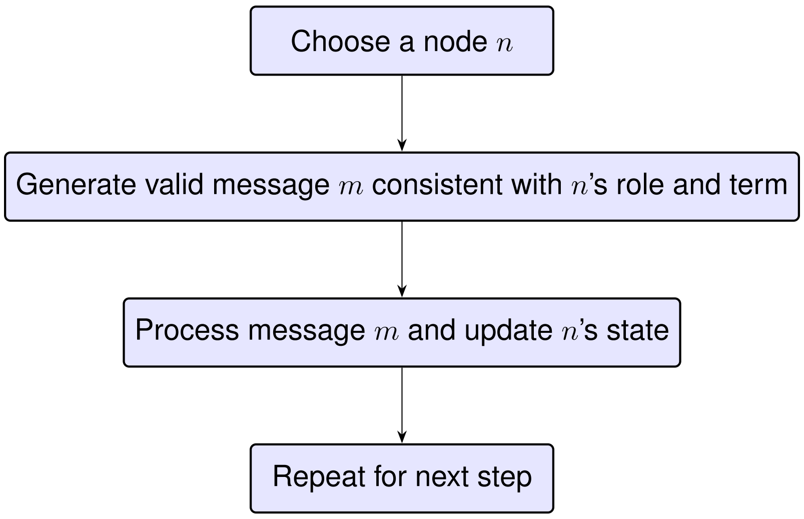
- **Heartbeat (Hb)**: Maintains leader authority.
- **Vote Request (Vr)**: Broadcast during elections.
- **Vote Grant**: Sent in response to vote requests.

4.5 Level 2.5: Nondeterministic Inbox Model : The final model

To reduce complexity, this model retains message-passing behavior while removing explicit queues.

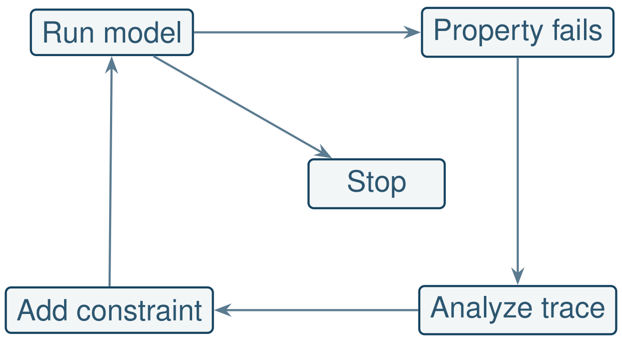
4.5.1 Key Simplification

Messages are placed nondeterministically in inboxes based on current global state.


Choose a node n
Generate valid message m consistent with n's role and term
Process message m and update n's state
Repeat for next step

4.5.2 Constraint Refinement

Starting with loose constraints, we refined message generation rules iteratively based on counterexamples.



4.5.3 Examples of constraints on inbox

Here are the valid message formats and their conditions:

Message	Condition
HB(t, s)	role$_s$ $=$ Leader $\wedge\, t <$ max_term
VR(t, s)	role$_s$ $=$ Candidate $\wedge\, t <$ max_term
VG(t, s)	role$_s$ $=$ Follower $\wedge\, t <$ max_term
HB(t, s) received by r	role$_s$ $=$ Leader $\wedge\, t <$ max_term $\wedge\, r \neq s$
VR(t, s) received by r	role$_s$ $=$ Candidate $\wedge\, t <$ max_term $\wedge\, r \neq s$
VG(t, s) received by r	role$_s$ $=$ Follower $\wedge\, t <$ max_term $\wedge\, r \neq s$

Table 4.1: Valid messages and their conditions in the protocol.

4.5.4 Abstraction Function to Level 2

$$A(\text{role}) \to \{\mathsf{F}, \mathsf{C}, \mathsf{L}\} \qquad \text{// Follower, Candidate, Leader} \tag{4.11}$$

$$A(\text{term}) \to currentTerm \tag{4.12}$$

$$A(\text{votes}[i][j]) \to \begin{cases} 1, & \text{if node } i \text{ received a vote from } j \\ 0, & \text{otherwise} \end{cases} \tag{4.13}$$

$$A(\text{quorum}) \to true \quad \text{if majority votes received} \tag{4.14}$$

4.6 Model Validation and Invariant Discovery

4.6.1 Property Preservation

We verified that the abstraction preserves:

- Term monotonicity
- Vote monotonicity
- Unique voting
- Leader promotion

4.6.2 Inductive Invariant Discovery

Through experimentation, we identified inductive invariants that improve verification:

- **Unique Vote**: Prevents multiple vote grants per term.

- **Unique Quorum**: Ensures a single majority per term.
- **Leader Validity**: Leader must have a valid quorum.

These invariants act as early warning indicators and significantly reduce the bound required in Bounded Model Checking.

Chapter 5

NuSMV Raft Consensus Protocol Model

This chapter presents a comprehensive analysis of a NuSMV model implementing the Raft consensus protocol. The model formalizes the distributed consensus algorithm using temporal logic and state transition systems, enabling formal verification of critical safety and liveness properties.

5.1 Model Architecture Overview

The NuSMV model implements a 3-node Raft consensus protocol with the following key architectural components:

- **Main Module**: Orchestrates the entire system with 3 nodes (node0, node1, node2).
- **Node Module**: Individual node behavior with state variables and message handling.
- **Active Node Selection**: Nondeterministic selection of which node processes messages.
- **Message Generation**: Constrained message creation based on node roles and system state.

5.2 Node State Variables

Each node maintains the following critical state variables that capture the essential Raft protocol state:

```
current_term : 0..5;
role : {Leader, Follower, Candidate};
voted_for : -1..2;
votes_received : array 0..2 of boolean;
timeout : boolean;
```

```
voted_in_term : 0..5;
inbox_type : {None, HB, VR, VG};
inbox_term : 0..5;
inbox_sender : 0..2;
```

5.3 Message Types and Semantics

The model uses three message types to simulate Raft communication:

- **HB (Heartbeat)**: Sent by leaders to maintain authority and reset follower timeouts.
- **VR (Vote Request)**: Sent by candidates requesting votes from followers.
- **VG (Vote Grant)**: Sent by followers granting votes to candidates.

5.4 Workflow Execution Steps

5.4.1 Active Node Selection

```
next(active_node) := {0, 1, 2};
```

5.4.2 Message Generation with Constraints

For Followers

```
TRANS
  (active_node = 0 & node0.role = Follower) ->
    ((next(node0.inbox_type) = None) |
     (next(node0.inbox_type) = HB & next(node0.inbox_term) <=
   max_term_in_system &
      ((next(node0.inbox_sender) = 1 -> node1.role = Leader) &
       (next(node0.inbox_sender) = 2 -> node2.role = Leader)))
    |
     ...
```

For Candidates

```
TRANS
  (active_node = 0 & node0.role = Candidate) ->
    ((next(node0.inbox_type) = None) |
     (next(node0.inbox_type) = HB & next(node0.inbox_term) <=
   max_term_in_system &
       ((next(node0.inbox_sender) = 1 -> node1.role = Leader) &
        (next(node0.inbox_sender) = 2 -> node2.role = Leader)))
    |
     ...
```

5.5 State Transition Processing

5.5.1 Timeout-Based Transitions

```
TRANS
  node0.timeout & node0.role = Follower ->
    next(node0.role) = Candidate &
    next(node0.current_term) = node0.current_term + 1 &
    next(node0.voted_for) = 0 &
    ...
```

5.5.2 Message-Based Transitions

```
TRANS
  node0.inbox_type != None ->
  case
    node0.inbox_term > node0.current_term:
      next(node0.role) = Follower &
      next(node0.current_term) = node0.inbox_term &
      ...
```

5.6 Key State Update Mechanisms

5.6.1 Term Management

```
TRANS
  (next(node0.current_term) > node0.current_term) ->
    (next(node0.current_term) = node0.current_term + 1 &
     ...
```

5.6.2 Voting Logic

```
DEFINE
  votes_count := toint(votes_received[0]) +
                 toint(votes_received[1]) +
                 toint(votes_received[2]);
```

5.7 Verification Properties

5.7.1 Safety Properties

```
SPEC AG !(node0.role = Leader & node1.role = Leader &
          node0.current_term = node1.current_term)
...
```

Thank you to Alberto Griggio who replied to my mails and gave an idea about using TRANS.

Chapter 6

Bug Injection and Safety Properties

This chapter systematically presents the types of faults targeted within our verification framework and delineates the associated safety properties employed to evaluate system correctness. Our primary focus is on the duplicate vote anomaly, a well-documented yet non-trivial error that compromises the election safety guarantees in Raft implementations.

6.1 Common Faults in Raft Implementations

Despite Raft's deliberate emphasis on conceptual clarity and simplicity, empirical studies of production-grade implementations reveal the presence of nuanced errors. Notably, Scott et al. have catalogued a variety of failure modes, which we briefly summarize below:

- **Vote Counting Defects:** Miscalculations arising during vote aggregation or validation.
- **Term Management Errors:** Faults stemming from improper comparison, update, or propagation of term values.
- **Log Indexing Anomalies:** Off-by-one and other indexing discrepancies during log comparison and replication.
- **Recovery Inconsistencies:** Inadequate restoration of internal state following node crashes.
- **Reconfiguration Faults:** Failures occurring during dynamic membership updates.

6.2 The Duplicate Vote Fault (raft-45)

Among the catalogued errors, the duplicate vote anomaly (identified as `raft-45`) represents a significant violation of Raft's safety criteria.

6.2.1 Formal Fault Description

As per the authoritative bug database:

"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."

6.2.2 Systemic Consequences

This fault, when manifest, can induce critical violations of protocol safety:

6.2.3 Minimal Configuration for Fault Manifestation

Through analytical modeling, we establish that a minimal configuration for triggering this anomaly includes:

- **Cluster Size:** A minimum of $N=4$ nodes is required.
- **Execution Depth:** At least $2x + 4$ transitions for $N = 2x$.
- **Voting Pattern:** At least one instance of a duplicate vote.

A configuration with $N=3$ nodes is provably insufficient, as any pair of votes necessarily constitutes a valid quorum.

6.3 Safety Specifications and Inductive Invariants

6.3.1 Primary Safety Condition: Election Uniqueness

$$\forall i, j \in \text{Nodes}, ; i \neq j : \neg(\text{role\_}i = \text{LEADER} \wedge \text{role\_}j = \text{LEADER} \wedge \text{term\_}i = \text{term\_}j) \quad (6.1)$$

This condition guarantees the existence of at most one leader in any given election term.

6.3.2 Auxiliary Inductive Invariants

The following invariants are introduced to bolster the primary safety check and assist bounded model checking in early bug detection:

- **Leader Validity:** $\forall i \in \text{Nodes} : \text{role}_i = \text{LEADER} \Rightarrow trueVotes_i \geq \lceil N/2 \rceil$
- **Unique Vote:**

$$\forall i \neq j,\ \text{term}_i = \text{term}_j,\ \forall k : \neg(\text{votesReceived}_i[k] > 0 \wedge \text{votesReceived}_j[k] > 0)$$

- **Unique Quorum:**

$$\forall i \neq j,\ \text{term}_i = \text{term}_j : \neg(trueVotes_i \geq \lceil N/2 \rceil \wedge trueVotes_j \geq \lceil N/2 \rceil)$$

- **Vote Consistency:**

$$\forall i, j \in \text{Nodes} : votesReceived_i[j] > 0 \Rightarrow votedFor_j = i$$

These invariants serve as logical constraints that help isolate latent errors in protocol behavior with greater efficiency.

6.4 Verification Methodology Using CBMC

6.4.1 Configuration Parameters

We leverage CBMC (C Bounded Model Checker) to evaluate both the baseline and faulty models. The tool is configured with the following parameters:

- `-property main.assertion.X` : Specifies the target property for analysis.
- `-trace-hex` : Enables trace outputs in hexadecimal format.
- `-DINJECT_DUPLICATE_VOTE_BUG` : Activates fault injection logic.
- `-DUSE_FIXED_INIT` : Applies deterministic initialization for repeatability.

6.4.2 Strategies for Invariant Application

To examine the efficacy of invariants, we experimented with multiple application strategies:

1. **Persistent Enforcement:** Enforcing invariants at every transition.
2. **Targeted Violation:** Intentionally breaking specific invariants under controlled conditions.
3. **Selective Injection:** Applying only a minimal subset of invariants deemed essential.
4. **Phase-Wise Staging:** Introducing invariants incrementally over distinct execution phases.

6.5 Analytical Bounds and Implications

6.5.1 Lower Bound on Execution Depth for Safety Violation

We derive a conservative lower bound on the number of execution steps required to manifest a duplicate vote-induced safety breach:

$$MinSteps(N) = 2 \times \lceil N/2 \rceil + 2 \tag{6.2}$$

This derivation is based on the following sequential behavior:

- Node A experiences a timeout, transitions to candidate state, and accumulates duplicate votes sufficient to claim leadership.
- Node B undergoes a similar process, using duplicate vote counts to also assert leadership in the same term.
- The resulting concurrent leadership violates election safety.

This theoretical result serves as a benchmark for assessing model checker efficacy in detecting safety violations under bounded conditions.

Chapter 7

Experimental Results and Analysis

This chapter presents a rigorous evaluation of our proposed invariant-augmented bounded model checking (BMC) methodology, applied to detecting the duplicate vote fault within Raft's leader election protocol. We detail the experimental configuration, systematically report performance metrics across varied model instances, and critically analyze the observed results.

7.1 Experimental Configuration

7.1.1 Verification Infrastructure

Our experiments employ the CBMC model checker (version 5.95.1), executed on an Apple Mac Pro M4 workstation equipped with 32GB of RAM. Each verification task was assigned a maximum timeout of 60 minutes.

The Raft leader election models under test are implemented in C, with configurations spanning cluster sizes of 4 to 6 nodes.

7.1.2 Evaluation Metrics

For each experimental run, the following metrics were recorded:

- **Wall-Clock Time:** Total duration of the verification process.
- **Violation Depth (k):** Minimum unrolling bound at which a property violation is detected.

7.2 Results: Four-Node Cluster ($N = 4$)

7.2.1 Baseline Verification Performance

Table 7.1 summarizes BMC performance under varying invariant assumptions for a four-node cluster.

Table 7.1: BMC Performance with Invariant Configurations ($N = 4$)

Assumed Invariants	Property Checked	Violation Depth k	Time (s)
None (Safety Only)	Election Safety	8	210.75
UV, UQ, L	Election Safety	8	181.05
UQ Only	Election Safety	8	174.72
UV Only	Election Safety	8	162.49
L Only	Election Safety	8	190.97

Observations

- The *Unique Vote* (UV) invariant yields the most significant improvement, reducing verification time by approximately 23%.
- All safety violations consistently occur at $k = 8$, corroborating the theoretically derived lower bound.
- Combining multiple invariants produces diminishing returns beyond UV.

7.2.2 Invariant-Focused Early Violation Detection

Table 7.2 demonstrates that direct verification of invariants expedites bug discovery.

Table 7.2: Direct Invariant Violation Detection ($N = 4$)

Assumed Property	Violation Depth k	Time (s)
Unique Quorum (UQ)	8	222.06
Unique Vote (UV)	4	80.83
Leader Validity (L)	5	83.25

Insights

- Violations of *Unique Vote* occur at $k = 4$, representing a 50% reduction in required unrolling depth.
- *Leader Validity* violations are detectable at $k = 5$.
- Targeted invariant checking significantly enhances verification efficiency.

7.2.3 Effect of Strategic Violation Injection

Table 7.3 examines the efficacy of deliberately injecting invariant violations during specific execution steps.

Table 7.3: Impact of Mid-Execution Invariant Violation Injection ($N = 4$)

Injected Fault	Property Checked	Violation Depth k	Time (s)
!UV at step 4	Election Safety	8	139.08
!L at step 5	Election Safety	8	171.33
!UQ at step 8	Election Safety	8	165.11

Key Findings

- Injecting *Unique Vote* violations at step 4 reduces verification time by 34% relative to baseline.
- Mid-execution violation strategies effectively guide BMC towards counterexample discovery.

7.3 Results: Six-Node Cluster ($N = 6$)

7.3.1 Scalability Assessment

Table 7.3.1 reports BMC performance for a six-node configuration.

Invariant Strategy	Property Checked	Violation Depth k	Time (s)
None	Election Safety	10	1333.11
All Invariants	Election Safety	10	1094.26
!UV Injection	Election Safety	10	814.00
L + UV + UQ	Election Safety	10	730.69

Scalability Insights

- Verification complexity increases non-linearly with cluster size.
- The most effective invariant combination (L + UV + UQ) achieves a 45% time reduction compared to baseline.
- The theoretical lower bound for safety violation depth ($k = 10$) holds in practice.

7.4 Synthesis and Interpretation

7.4.1 Invariant Efficacy

- **Unique Vote (UV):** Most effective for both performance improvement and early bug detection.

- **Unique Quorum (UQ):** Provides moderate search space reduction.
- **Leader Validity (L):** Highly effective when applied selectively; risk of over-constraining otherwise.

7.4.2 Strategic Recommendations

- Excessive invariant combinations may counteract intended efficiency gains.
- Temporal staging and targeted fault injection significantly expedite counterexample discovery.
- Invariant design should prioritize alignment with known bug mechanisms.

The results substantiate our central hypothesis that carefully constructed inductive invariants substantially enhance BMC efficiency for realistic distributed protocol models.

Chapter 8

Future Work and Limitations

The current NuSMV model focuses only on the leader election phase of the Raft protocol. Extending this model to cover the full protocol, including log replication and leader commitment, is a natural next step. A complete model would allow formal verification of Raft's safety and liveness properties beyond leader election.

The present model also has technical limitations. The number of nodes is hard-coded, and constructs like procid, which are deprecated in NuSMV, have been used. The state update logic is verbose and repetitive, with each state variable updated separately. A more structured and general approach to state updates would improve readability and maintainability.

Future work should focus on building a scalable, parameterized model of Raft that avoids hard-coding and deprecated features. This would allow systematic verification across different system sizes and simplify the implementation. A complete, clean model of Raft in NuSMV remains an important goal.