Ph.D. Dissertation

Automated and Interpretable Verification of Distributed Protocols

William Schultz

Khoury College of Computer Science

Northeastern University

Ph.D. Committee

Advisor	Stavros Tripakis	
	Pete Manolios	
	Ji-Yong Shin	
Ext. member	Stephan Merz	**Inria**

November 30, 2024

Northeastern University
Khoury College of
Computer Sciences

PhD Thesis Approval

Thesis Title: Automated and Interpretable Verification of Distributed Protocols

Author: William Schultz

PhD Program: [✓ Computer Science] ____ Cybersecurity ____ Personal Health Informatics

PhD Thesis Approval to complete all degree requirements for the above PhD program.

STAVROS TRIPAKIS [illegible]	11/20/2024
Thesis Advisor	*Date*
Stephan Merz Stephan Merz	11/20/2024
Thesis Reader	*Date*
Ji-Yong Shin Ji Yong Shin	11/21/2024
Thesis Reader	*Date*
Pete Manolios [illegible]	11/21/2024
Thesis Reader	*Date*
Thesis Reader	*Date*

KHOURY COLLEGE APPROVAL:

[illegible]	**11/27/2024**
Associate Dean for Graduate Programs	*Date*

COPY RECEIVED BY GRADUATE STUDENT SERVICES:

Pete Morency	11/27/2024
Recipient's Signature	*Date*

Distribution: Once completed, this form should be attached as page 2, immediately following the title page of the dissertation document. An electronic version of the document can then be uploaded to the Northeastern University-UMI Website.

Acknowledgements

First and foremost, I would like to thank my advisor, Prof. Stavros Tripakis, for his guidance, support, and mentorship throughout my PhD. His insights, feedback, and encouragement have been invaluable in shaping both this work and my development as a rigorous and thoughtful independent researcher.

I am additionally grateful to my committee members, Prof. Pete Manolios, Prof. Ji-Yong Shin, and Stephan Merz, for their thoughtful feedback and discussions that helped strengthen my work. Their expertise and perspectives have greatly enriched my research.

I am also thankful to Heidi Howard and Eddy Ashton at Microsoft Research Cambridge, who provided wonderful guidance and mentorship and were especially helpful in developing my views on and approaches to distributed systems research. Likewise, Jesse Bingham and Joonwon Choi were excellent mentors to me during my time working on chip verification at Apple, providing me with the opportunity to experience and apply formal verification in a practical industry setting.

I would also like to thank my colleagues and collaborators at MongoDB, particularly Siyuan Zhou and Tess Avitabile, who helped shape my initial views of the research world, and were perhaps most instrumental in inspiring me to pursue a PhD in the first place. The opportunity to work on designing and building real-world distributed systems with them has thoroughly informed my perspectives on engineering and research.

I would like to express sincere gratitude to my parents for their unwavering support and encouragement throughout this journey. They have supported and nourished my interests and goals for as long as I can remember, and I am deeply grateful.

Most importantly, I would like to thank my wife, Ryann, for her never-ending love, support, and encouragement throughout this journey, enduring absences and internships across coasts and continents, but always being a source of joy, lightness, and inspiration. There is no one else I would have rather shared this journey with.

Abstract

Distributed protocols serve as the foundation of modern fault tolerant systems, making the correctness of these protocols critical to the reliability of large scale database, cloud computing, and other decentralized systems. Formally modeling and automatically verifying the safety of distributed systems, however, remains an important and difficult challenge, and remains a non-trivial task that has traditionally required a large amount of human effort. A fundamental approach for reasoning about the correctness of these protocols involves specifying system *invariants*, which are assertions that must hold in every reachable system state. The classical technique for proving that such a system satisfies a given invariant is to discover an *inductive invariant*, which is an invariant that is typically stronger than the desired system invariant, and is preserved by all protocol transitions. Discovering inductive invariants is typically one of the most challenging aspects of verification. In this dissertation, we attempt to advance the state of the art of inductive safety verification of distributed protocols, both by manual and automated techniques, and by improving the interpretability of formal inductive proof artifacts as they are being developed and evolved.

We first present the design and formal specification and verification of a novel, logless dynamic reconfiguration protocol for Raft-based replication systems. Ours is the first formal specification and verification of the safety of a reconfigurable, Raft-based consensus protocol. We focus on a Raft-based dynamic reconfiguration protocol employed in the widely used MongoDB distributed database system. We present a formally stated inductive invariant for the protocol, which we formally prove and utilize to establish high level safety properties of the protocol.

We next present a new technique for automatically inferring inductive invariants of parameterized distributed protocols specified in TLA$^+$. We present a new algorithm for inductive invariant inference that is based around a core procedure for generating *plain*, potentially non-inductive *lemma invariants* that are used as candidate conjuncts of an overall inductive invariant. We couple this with a greedy lemma invariant selection procedure that selects lemmas that eliminate the largest number of counterexamples to induction at each round of our inference procedure.

Finally, aiming to improve the *interpretability* of automated verification methods for these protocols, we present *inductive proof decomposition*, a compositional technique for inductive invariant inference that scales to large distributed protocol verification tasks and provides an interpretable proof artifact in case of failure and during development. Our technique is built on a core data structure, the *inductive proof graph*, which explicitly represents the relative induction dependencies of an inductive invariant and is built incrementally during the inference procedure. We present an inductive invariant synthesis algorithm that integrates localized syntax-guided lemma synthesis routines at nodes of this graph, accelerated by computation of localized grammar and state variable *slices*. In the case of failure to produce a complete inductive invariant, this proof graph structure allows failures to be localized to small sub-components of this graph, enabling fine-grained failure diagnosis and repair by a user. We evaluate our technique on several complex distributed and concurrent protocols, including a large, asynchronous specification of the Raft distributed replication protocol.

Contents

Chapter 1

Introduction

High complexity increases the probability of human error in design, code, and operations. Errors in the core of the system could cause loss or corruption of data, or violate other interface contracts on which our customers depend. So, before launching such a service, we need to reach extremely high confidence that the core of the system is correct. We have found that the standard verification techniques in industry are necessary but not sufficient. We use deep design reviews, code reviews, static code analysis, stress testing, fault-injection testing, and many other techniques, but we still find that subtle bugs can hide in complex concurrent fault-tolerant systems.

Chris Newcombe, *Use of Formal Methods at Amazon Web Services* [1]

Modern computational systems and infrastructure have become fundamentally distributed in nature. Databases [2–4], blockchains [5, 6], and cloud applications all run across in distributed environments and across devices with highly unpredictable failure modes and latency characteristics. Underlying these critical systems and applications are *protocols* that must work correctly in these highly distributed, asynchronous environments. These distributed protocols serve as the foundation of many modern fault-tolerant systems, making their correctness critical to the reliability of modern, large scale database and cloud systems [2, 4, 7].

Designing these protocols correctly, however, is a notoriously difficult challenge, and their correctness is crucial as they serve as core components in complex systems and applications with many software layers. It has often been the case that protocols are designed and even formally published, but later discovered to have subtle and significant correctness bugs [8–10]. Thus, the design, modeling, and verification of these protocols has necessitated a level of mathematical rigor and precision afforded by the techniques of formal methods [11]. That is, approaches for formally *specifying* these protocols and their correctness properties, and having a means to formally *verify* their correctness. At its core, formal methods provides techniques for describing algorithms (e.g. distributed protocols) in a formal, mathematical language, making them amenable to mechanized and automated reasoning about their behavior and precise definition of their correctness (e.g., safety and liveness) properties. This approach to formally modeling systems has gained significant traction in industrial settings, where the cost of errors in system and protocol design is high [1, 12, 13].

In this dissertation, we aim to advance the state of the art in safety verification of distributed protocols, both by manual and novel automated techniques. Our aim is to improve the efficiency and applicability of automated verification techniques, and also the interpretability of formal proof artifacts as they are being developed. We introduce and outline the core problems we focus on in this dissertation below.

1.1 Verified Dynamic Reconfiguration

Distributed replication systems based on the replicated state machine model [14] have become ubiquitous as the foundation of modern, fault-tolerant data storage systems. In order for these systems to ensure availability in the presence of faults, they must be able to dynamically replace failed nodes with healthy ones, a process known as *dynamic reconfiguration*. The protocols for building distributed replication systems have been well studied and implemented in a variety of systems [2, 4, 15, 16]. The Raft consensus protocol, originally published in 2014, provided a dynamic reconfiguration algorithm in its initial publication, but did not include a precise discussion of its correctness or include a formal specification or proof. A critical safety bug [9] in one of its reconfiguration protocols was found after initial publication, demonstrating that the design and verification of reconfiguration protocols for these systems is a challenging task.

The first problem we focus on in this dissertation is the design of a novel, logless reconfiguration algorithm for Raft-based consensus systems, along with a formal specification of its behavior and a formally verified safety proof. Our work constitutes the first formal specification and safety proof for a Raft-based reconfiguration protocol.

1.2 Automated Inductive Invariant Inference

A fundamental approach for reasoning about the correctness of distributed protocols involves specifying system *invariants*, which are assertions that must hold in every reachable system state. For adequately small, finite state systems, symbolic or explicit state model checking techniques [17–19] can be sufficient to automatically prove invariants. For verification of infinite state or *parameterized* protocols, however, model checking techniques may, in general, be incomplete [20]. Thus, the standard technique for proving that such a system satisfies a given invariant is to discover an *inductive invariant*, which is an invariant that is typically stronger than the desired system invariant, and is preserved by all protocol transitions. Discovering inductive invariants, however, is one of the most challenging aspects of verification and remains a non-trivial task with a large amount of human effort required [21–24]. Thus, automating the inference of these invariants is a desirable goal.

In general, the problem of inferring inductive invariants for infinite state protocols is undecidable [25]. Even the verification of inductive invariants may require checking the validity of arbitrary first order formulas, which is undecidable [26]. Thus, this places fundamental limits on the development of fully general algorithmic techniques for discovering inductive invariants. Significant progress towards automation of inductive invariant discovery for infinite state protocols has been made with the Ivy framework [27], which utilizes a restricted system modeling language that allows for efficient checking of verification goals via an SMT solver such as Z3 [28]. In particular, the EPR and extended EPR subsets of Ivy are decidable. Ivy also provides an interface for an interactive, counterexample guided invariant discovery process. The Ivy language, however, may place an additional burden on users when protocols or their invariants don't fall naturally into one of the decidable fragments of Ivy. Transforming a protocol into such a fragment is a manual and nontrivial task [26].

Subsequent work has attempted to fully automate the discovery of inductive invariants for distributed protocols. State of the art tools for inductive invariant inference for distributed protocols include I4 [29], fol-ic3 [30], IC3PO [31], SWISS [32], and DistAI [33]. All of these tools, however, accept only Ivy or an Ivy-like language [34] as input. Moreover, several of these tools work only within the restricted decidable fragments of Ivy.

In Chapter 4, we present a new technique for automatic discovery of inductive invariants for protocols specified in TLA$^+$, a high level, expressive specification language [35]. To our knowledge, this is the first inductive invariant discovery tool for distributed protocols in a language other than Ivy. Our technique is built around a core procedure for generating small, *plain* (potentially non-inductive) invariants. We search for these invariants on finite protocol instances, employing the so-called *small scope* hypothesis [29, 36, 37], circumventing undecidability concerns when reasoning over unbounded domains. We couple this invariant generation procedure with an invariant selection procedure based on a greedy counterexample elimination heuristic in order to incrementally construct an overall inductive invariant.

1.3 Interpretable Inductive Protocol Verification

In the domain of distributed protocol verification, there have been several recent efforts to develop more automated inductive invariant development techniques [33, 38–40]. Many of these tools are based on modern model checking algorithms like IC3/PDR [30, 38, 40–42], and others based on syntax-guided or enumerative invariant synthesis methods [43–45]. These techniques have made significant progress on solving various classes of distributed protocols, including some variants of real world protocols like the Paxos consensus protocol [41, 46]. The theoretical complexity limits facing these techniques, however, limit their ability to be fully general [47] and, even in practice, the performance of these tools on complex protocols is still unpredictable, and their failure modes can be opaque.

In particular, one key drawback of these methods is that, in their current form, they are very much "all or nothing". That is, a given problem can either be automatically solved with no manual proof effort, or the problem falls outside the method's scope and a failure is reported. In the latter case, little assistance is provided in terms of how to develop a manual proof or how a human can offer guidance to the tool. We believe there is significant utility in providing a smoother transition between these possible outcomes. In practice, real world, large-scale verification efforts often benefit from some amount of human interpretability and interaction i.e., a human provides guidance when an automated engine is unable to automatically prove certain properties about a design or protocol. This may involve simplifying the problem statement given to the tool, or completing some part of the tool's proof process by hand. Recent verification efforts of industrial scale protocols often note the high amount of human effort in developing inductive invariants. Some leave human integration as future goals [24, 48], while others have adopted a paradigm of integrating human assistance to accelerate proofs for larger verification problems e.g., in the form of a manually developed refinement hierarchy [41, 49].

In Chapter 5 we present *inductive proof decomposition*, a new technique for inductive invariant development that aims to address these limitations of existing approaches. Our technique utilizes the underlying compositional structure of an inductive invariant to guide its development, based on the insight that a standard inductive invariant can be decomposed into an *inductive proof graph*. This graph structure makes explicit the induction dependencies between lemmas of an inductive invariant, and their relationship to the logical actions of a concurrent or distributed protocol. It serves as a core guidance mechanism for inductive invariant development, by making the global dependency structure apparent. In addition, the structure of the proof graph allows for localized reasoning about proof obligations, enabling a user or automated tool to focus on small sub-problems of the inductive proof rather than a large, monolithic inductive invariant.

We build a technique for automatically and efficiently synthesizing these proof graphs, thus enabling automation while also preserving amenability to human interaction and interpretability. We demonstrate that these proof graphs can be presented to and interpreted directly by a human user, facilitating a concrete and effective diagnosis and interaction process, enhancing interpretability of both the final inductive proof and the intermediate results. In addition, our automated synthesis technique also takes advantage of the proof graph structure to accelerate its local synthesis tasks.

1.4 Thesis Outline

The rest of this dissertation is structured as follows.

Chapter 2 begins by providing background on the formalisms used throughout this dissertation for formally modeling and specifying properties about concurrent and distributed protocols.

In Chapter 3 we examine the design and verification of *MongoRaftReconfig*, a logless dynamic reconfiguration protocol used in the Raft-based database replication system of MongoDB. We discuss the design and formal specification of this reconfiguration protocol [50], followed by work on development of a formal inductive invariant and proof in the TLA$^+$ proof system for establishing high level protocol safety [24].

In Chapter 4 we move on to the problem of *automatically* inferring inductive invariants for establishing the safety of distributed protocols. We consider protocols specified in the TLA$^+$ specification language [35], and propose an invariant inference algorithm that combines a lemma invariant generation prodcedure with a greedy, counterexample-guided lemma invariant selection heuristic [45]. To our knowledge, this is the first inductive invariant inference technique that works directly on protocols specified in TLA$^+$, a widely used and high level specification language that is widely used in industrial settings for modeling and verification [51]. We implement our technique in a tool and evaluate it in a diverse set of distributed protocol benchmarks, and compare against several other state of the art inductive invariant inference tools.

Finally, in Chapter 5, we focus on how to both scale inductive invariant inference techniques further while also making them more *interpretable* [52]. We propose a new data structure, the *inductive proof graph*, for managing the compositional structure of a monolithic inductive invariant. We then propose an inductive invariant development technique based on this data structure, *inductive proof decomposition*, that scales to large scale distributed protocols effectively, and also provides a high degreee of *interpretability* of the generated inductive proofs. We implement our technique and evaluate it on several large scale protocol benchmarks, demonstrating its ability to synthesize inductive invariants for large scale, asynchronous specifications of the Raft consensus protocol.

Chapter 2

Preliminaries

Throughout this dissertation, we adopt the notation of TLA$^+$ [35] for formally specifying distributed protocols and their correctness properties. TLA$^+$ is an expressive, high level specification language for specifying distributed and concurrent protocols. It has also been used effectively in industry for specifying and verifying correctness of protocol designs [1, 12].

2.1 Symbolic Transition Systems and TLA$^+$

The distributed and concurrent protocols we consider in this proposal can be modeled as parameterized *symbolic transition systems*. A symbolic transition system consists of a set of *state variables*, where a *state* is an assignment of values to all state variables. We use the notation $s \models P$ to denote that state s *satisfies* state predicate P, i.e., that P evaluates to true once we replace all state variables in P by their values as given by s. A state predicate *Init* specifies the possible values of the state variables at an *initial state* of the system. A predicate *Next* defines the *transition relation* of a symbolic transition system. In TLA$^+$, *Next* is typically written as a disjunction of *actions* i.e., possible symbolic transitions.

A simple example of a small distributed protocol specified in TLA$^+$ is shown in Figure 2.1. This system is parameterized by two sorts, *Server* and *Client*, where each sort represents an uninterpreted constant that can be interpreted as any set of values. *Instantiating* a sort means fixing it to a *finite* domain of distinct elements. For example, we can instantiate *Server* to be the set $\{a_1, a_2\}$ (meaning there are only two servers, denoted a_1 and a_2), and *Client* to be the set $\{c_1, c_2\}$ (meaning there are only two clients, denoted c_1 and c_2).

Given two states, s and s', we use the notation $s \rightarrow s'$ to denote that there exists a transition from s to s', i.e., that the pair (s, s') satisfies the transition relation predicate $Next$. A *behavior* is an infinite sequence of states $s_0, s_1, \dots$, such that $s_0 \models Init$ and $s_i \rightarrow s_{i+1}$ (i.e., $(s_i, s_{i+1}) \models Next$) for all $i \geq 0$. A state s is *reachable* if there exists a behavior $s_0, s_1, \dots$, such that $s = s_i$ for some i. We use $Reach(M)$ to denote the reachable

```
CONSTANT Server, Client
VARIABLE locked, held

Init ≜
    ∧ locked = [i ∈ Server ↦ TRUE]
    ∧ held = [i ∈ Client ↦ {}]

Connect(c, s) ≜
    ∧ locked[s] = TRUE
    ∧ held′ = [held EXCEPT ![c] = held[c] ∪ {s}]
    ∧ locked′ = [locked EXCEPT ![s] = FALSE]

Disconnect(c, s) ≜
    ∧ s ∈ held[c]
    ∧ held′ = [held EXCEPT ![c] = held[c] \ {s}]
    ∧ locked′ = [locked EXCEPT ![s] = TRUE]

Next ≜
    ∨ ∃ c ∈ Client, s ∈ Server : Connect(c, s)
    ∨ ∃ c ∈ Client, s ∈ Server : Disconnect(c, s)

Spec ≜ Init ∧ □[Next]_⟨locked, held⟩

Safe ≜
    ∀ c_i, c_j ∈ Client :
      (held[c_i] ∩ held[c_j] ≠ {}) ⟹ (c_i = c_j)
```

Figure 2.1: A simple parameterized protocol defined in TLA⁺.

states of a transition system M.

The entire set of behaviors of the system is defined as a single temporal logic formula *Spec*. In TLA⁺, *Spec* is typically defined as the TLA⁺ formula $Init \wedge \Box[Next]_{Vars}$, where $\Box$ is the "always" operator of linear temporal logic, and $[Next]_{Vars}$ represents a transition which either satisfies $Next$ or is a *stuttering* step, i.e., where all state variables in $Vars$ remain unchanged.

2.2 Safety Verification and Inductive Invariants

In this proposal we are concerned with the verification of safety properties, and in particular *invariants*, which are state predicates that hold at all reachable states. Formally, a state predicate P is an *invariant* if $s \models P$ holds for every reachable state s. The verification problem consists in checking that a system satisfies its specification. In TLA⁺, both the system and the specification are written as temporal logic formulas. Therefore, expressed in TLA⁺, the safety verification problem we consider consists of checking that the temporal logic formula

$$Spec \Rightarrow \Box Safe \tag{2.1}$$

is *valid* (i.e., true under all assignments). That is, establishing that Safe is an invariant of the system defined by Spec.

A standard technique for solving the safety verification problem (2.1) is to come up with an *inductive invariant* [53]. That is, a state predicate Ind which satisfies the following conditions:

$$\mathit{Init} \Rightarrow \mathit{Ind} \tag{2.2}$$
$$\mathit{Ind} \land \mathit{Next} \Rightarrow \mathit{Ind}' \tag{2.3}$$
$$\mathit{Ind} \Rightarrow \mathit{Safe} \tag{2.4}$$

where Ind' denotes the predicate Ind where state variables are replaced by their primed, next-state versions. Conditions (2.2) and (2.3) are, respectively, referred to as *initiation* and *consecution*. Condition (2.2) states that Ind holds at all initial states. Condition (2.3) states that Ind is *inductive*, i.e., if it holds at some state s then it also holds at any successor of s. Together these two conditions imply that Ind is also an invariant, i.e., that it holds at all reachable states. Condition (2.4) states that Ind is stronger than the invariant Safe that we are trying to prove. Therefore, if all reachable states satisfy Ind, they also satisfy Safe, which establishes (2.1). A core problem considered in this proposal is how to infer such an inductive invariant automatically.

Note also that for a given system $M = (I, T)$, a state predicate may be inductive only under the assumption of some other predicate. For given state predicates Ind and L, if the formula $L \land \mathit{Ind} \land T \Rightarrow \mathit{Ind}'$ is valid, we say that Ind is *inductive relative to* L.

2.3 Lemma Invariants and Counterexamples to Induction

An inductive invariant Ind typically has the form $\mathit{Ind} \triangleq \mathit{Safe} \land A_1 \land \cdots \land A_k$, where the conjuncts $A_1, ..., A_k$ are state predicates and we refer to them as *lemma invariants*. Observe that each A_i must itself be an invariant. The reason is that Ind must be an invariant, i.e., must contain all reachable states, and since Ind is stronger than (i.e., contained in) each A_i, each A_i must itself contain all reachable states. Although all lemma invariants must be invariants, they need not be individually inductive. However, the conjunction of all lemma invariants together with the safety property Safe must be inductive.

Given a state predicate P (which is typically a candidate inductive invariant), a *counterexample to induction (CTI)* is a state s such that: (1) $s \models P$; and (2) s can reach a state satisfying $\neg P$ in k steps, i.e. there exist transitions $s \rightarrow s_1 \rightarrow s_2 \rightarrow \cdots \rightarrow s_k$ and $s_k \models \neg P$. That is, a CTI is a state s which proves that P is not inductive i.e., not "closed" under the transition relation. We denote the set of all CTIs of predicate P by $\mathit{CTIs}(P)$. Note that for any inductive invariant Ind, the set $\mathit{CTIs}(\mathit{Ind})$ is empty. Given another state predicate Q and a state $s \in \mathit{CTIs}(P)$, we say that Q *eliminates* s if $s \not\models Q$, i.e., if $s \models \neg Q$.

Chapter 3

Design and Verification of a Logless Dynamic Reconfiguration Protocol

Practical systems must be able to handle changes in the set of replicas. This is referred to as the group membership problem in the literature...While group membership with the core Paxos algorithm is straightforward, the exact details are non-trivial when we introduce Multi-Paxos, disk corruptions, etc. Unfortunately the literature does not spell this out, nor does it contain a proof of correctness for algorithms related to group membership changes using Paxos. We had to fill in these gaps to make group membership work in our system.

Chandra et al., *Paxos Made Live - An Engineering Perspective* [15]

This chapter includes work originally presented in [50] and [24], and the supplementary materials for this work, including TLA+ specifications and TLAPS proofs can be found at [54]. Specifically, Sections 3.1 through Section 3.6 are based on the work in [50], while Section 3.7 draws from [24]. The formal verification work presented in Section 3.7 relies on the TLA+ specifications and TLAPS proofs available in [54].

3.1 Introduction

Distributed replication systems based on the replicated state machine model [14] have become ubiquitous as the foundation of modern, fault-tolerant data storage systems. In order for these systems to ensure availability in the presence of faults, they must be able to dynamically replace failed nodes with healthy ones, a process known as *dynamic reconfiguration*. The protocols for building distributed replication systems have been well studied and implemented in a variety of systems [2, 4, 15, 16]. Paxos [55] and, more recently, Raft [56], have served as the logical basis for building provably correct distributed replication systems. Dynamic reconfiguration, however, is an additionally challenging and subtle problem [57] that has not been explored as extensively as the foundational consensus protocols underlying these systems. Variants of Paxos have examined the problem of dynamic reconfiguration but these reconfiguration techniques may require changes to a running system that impact availability [58] or require the use of an external configuration master [59]. The Raft consensus protocol, originally published in 2014, provided a dynamic reconfiguration algorithm in its initial publication, but did not include a precise discussion of its correctness or include a formal specification or proof. A critical safety bug [9] in one of its reconfiguration protocols was found after initial publication, demonstrating that the design and verification of reconfiguration protocols for these systems is a challenging task.

MongoDB [60] is a general purpose, document oriented database which implements a distributed replication system [3] for providing high availability and fault tolerance. MongoDB's replication system uses a novel consensus protocol that derives from Raft [61]. Since its inception, the MongoDB replication system has provided a custom, legacy protocol for dynamic reconfiguration of replica members that was not based on a published algorithm. This legacy protocol managed configurations in a *logless* fashion i.e. each server only stored its latest configuration. In addition, it decoupled reconfiguration processing from the main database operation log. These features made for a simple and appealing protocol design, and it was sufficient to provide basic reconfiguration functionality to clients. The legacy protocol, however, was known to be unsafe in certain cases. In recent versions of MongoDB, reconfiguration has become a more common operation, necessitating the need for a redesigned, safe reconfiguration protocol with rigorous safety guarantees. From a system engineering perspective, a primary goal was to keep design and implementation complexity low. Thus, it was desirable that the new reconfiguration protocol minimize changes to the legacy protocol to the extent possible. In this chapter, we present *MongoRaftReconfig*, a novel dynamic reconfiguration protocol that achieves the above design goals.

MongoRaftReconfig provides safe, dynamic reconfiguration, utilizes a logless approach to managing configuration state, and decouples reconfiguration processing from the main database operation log. Thus, it bears a high degree of architectural and conceptual similarity to the legacy MongoDB protocol, satisfying our original design goal of minimizing changes to the legacy protocol. We provide rigorous safety guarantees of *MongoRaftReconfig*, including a proof of the protocol's main safety properties along with a formal specification in TLA+ [62], a specification language for describing distributed and concurrent systems. To our knowledge, this is the first published safety proof and formal specification of a reconfiguration protocol for a Raft-based system. We also verified the safety properties of finite instances of *MongoRaftReconfig* using the TLC model checker [63], which provides additional confidence in its correctness. Finally, we discuss the conceptual novelties of *MongoRaftReconfig*, related to its logless design and decoupling of reconfiguration processing. In particular, we discuss how it can be understood as an optimized and generalized variant of the single server Raft reconfiguration protocol.

We also include a preliminary experimental evaluation of how these optimizations can provide performance benefits over standard Raft, by allowing reconfigurations to bypass the main operation log.

To summarize, in this chapter we make the following contributions:

- We present *MongoRaftReconfig*, a novel, logless dynamic reconfiguration protocol for the MongoDB replication system, along with a proof of the protocol's key safety properties. To our knowledge, this is the first published safety proof of a reconfiguration protocol for a Raft-based system.

- We present a formal specification of *MongoRaftReconfig* in TLA+, along with results of model checking the safety properties of *MongoRaftReconfig* on finite protocol instances using the TLC model checker. To our knowledge, this is the first published formal specification of a reconfiguration protocol for a Raft-based system.

- We discuss the conceptual novelties of *MongoRaftReconfig*, and how it can be understood as an optimized and generalized variant of the single server Raft reconfiguration protocol.

- We provide a preliminary experimental evaluation of *MongoRaftReconfig*'s performance benefits, demonstrating how it improves upon reconfiguration in standard Raft.

- We present a formally stated inductive invariant for the safety of the *MongoRaftReconfig* protocol, along with formally verified TLAPS proof that *MongoRaftReconfig* satisfies its main high level safety property, *StateMachineSafety*. To our knowledge, this is both the first formally stated inductive invariant and machine checked safety proof for a Raft-based reconfiguration protocol.

3.2 Background

3.2.1 System Model

Throughout this chapter, we consider a set of *server* processes $Server = \{s_1, s_2, ..., s_n\}$ that communicate by sending messages. We assume an asynchronous network model in which messages can be arbitrarily dropped or delayed. We assume servers can fail by stopping but do not act maliciously i.e. we assume a "fail-stop" model with no Byzantine failures. We define both a *member set* and a *quorum* as elements of 2^{Server}. Member sets and quorums have the same type but refer to different conceptual entities. For any member set m, and any two non-empty member sets m_i, m_j, we define the following:

$$Quorums(m) \triangleq \{s \in 2^m : |s| \cdot 2 > |m|\} \tag{3.1}$$

$$QuorumsOverlap(m_i, m_j) \triangleq \forall q_i \in Quorums(m_i), q_j \in Quorums(m_j) : q_i \cap q_j \neq \emptyset \tag{3.2}$$

where $|S|$ denotes the cardinality of a set S. We refer to Definition 3.2 as the *quorum overlap* condition.

3.2.2 Raft

Raft [64] is a consensus protocol for implementing a replicated log in a system of distributed servers. It has been implemented in a variety of systems across the industry [65]. Throughout this chapter, we refer to the original Raft protocol as described and specified in [64] as *standard Raft*.

The core Raft protocol implements a replicated state machine using a static set of servers. In the protocol, time is divided into *terms* of arbitrary length, where terms are numbered with consecutive integers. Each term has at most one leader, which is selected via an *election* that occurs at the beginning of a term. To dynamically change the set of servers operating the protocol, Raft includes two, alternate algorithms: *single server membership change* and *joint consensus*. In this chapter we are only concerned with *single server membership change*. The single server change approach aims to simplify reconfiguration by allowing only reconfigurations that add or remove a single server. Reconfiguration is accomplished by writing a special reconfiguration entry into the main Raft operation log that alters the local configuration of a server. In this chapter, when referring to reconfiguration in standard Raft, we assume it to mean the single server change protocol.

3.2.3 Replication in MongoDB

MongoDB is a general purpose, document oriented database that stores data in JSON-like objects. A MongoDB database consists of a set of collections, where a collection is a set of unique documents. To provide high availability, MongoDB provides the ability to run a database as a *replica set*, which is a set of MongoDB servers that act as a consensus group, where each server maintains a logical copy of the database state.

MongoDB replica sets utilize a replication protocol that is derived from Raft, with some extensions. We refer to MongoDB's abstract replication protocol, without dynamic reconfiguration, as *MongoStaticRaft*. This protocol can be viewed as a modified version of standard Raft that satisfies the same underlying correctness properties. A more in depth description of *MongoStaticRaft* is given in [3, 61], but we provide a high level overview here, since the *MongoRaftReconfig* reconfiguration protocol is built on top of *MongoStaticRaft*. In a replica set running *MongoStaticRaft* there exists a single *primary* server and a set of *secondary* servers. As in standard Raft, there is a single primary elected per term. The primary server accepts client writes and inserts them into an ordered operation log known as the *oplog*. The oplog is a logical log where each entry contains information about how to apply a single database operation. Each entry is assigned a monotonically increasing timestamp, and these timestamps are unique and totally ordered within a server log. These log entries are then replicated to secondaries which apply them in order leading to a consistent database state on all servers. When the primary learns that enough servers have replicated a log entry in its term, the primary will mark it as *committed*, guaranteeing that the entry is permanently durable in the replica set. Clients of the replica set can issue writes with a specified *write concern* level, which indicates the durability guarantee that must be satisfied before the write can be acknowledged to the client. Providing a write concern level of *majority* ensures that a write will not be acknowledged until it has been marked as committed in the replica set. A key, high level safety requirement of the replication protocol is that if a write is acknowledged as committed to a client, it should be durable in the replica set.

3.3 MongoRaftReconfig: A Logless Dynamic Reconfiguration Protocol

In this section we present the *MongoRaftReconfig* dynamic reconfiguration protocol. First, we provide an overview and some intuition on the protocol design in Section 3.3.1. Section 3.3.2 provides a high level, informal description of the protocol along with a condensed pseudocode description in Algorithm 2. Sections 3.3.3 and 3.3.4 provide additional detail on the mechanisms required for the protocol to operate safely, and the TLA+ formal specification of *MongoRaftReconfig* is discussed briefly in Section 3.3.5.

The complete description of *MongoRaftReconfig* is left to Appendix A.1. The pseudocode presented in Algorithm 2 describes the reconfiguration specific behaviors of *MongoRaftReconfig*, which are the novel aspects of the protocol and the contributions of this work.

3.3.1 Overview and Intuition

Dynamic reconfiguration allows the set of servers operating as part of a replica set to be modified while maintaining the core safety guarantees of the replication protocol. Many consensus based replication protocols [56, 58, 66] utilize the main operation log (the *oplog*, in MongoDB) to manage configuration changes by writing special reconfiguration log entries. The *MongoRaftReconfig* protocol instead decouples configuration updates from the main operation log by managing the configuration state of a replica set in a separate, logless replicated state machine, which we refer to as the *config state machine*. The config state machine is maintained alongside the oplog, and manages the configuration state used by the overall protocol.

In order to ensure safe reconfiguration, *MongoRaftReconfig* imposes specific restrictions on how reconfiguration operations are allowed to update the configuration state of the replica set. First, it imposes a *quorum overlap* condition on any reconfiguration from C to C', which is an approach adopted from the Raft single server reconfiguration algorithm. This ensures that all quorums of two adjacent configurations overlap with each other, and so can safely operate concurrently. In order to allow the system to pass through many configurations over time, though, *MongoRaftReconfig* imposes additional restrictions which address two essential aspects required for safe dynamic reconfiguration: (1) *deactivation* of old configurations and (2) *state transfer* from old configurations to new configurations. Essentially, it must ensure that old configurations, which may not overlap with newer configurations, are appropriately prevented from executing disruptive operations (e.g. electing a primary or committing a write), and it must also ensure that relevant protocol state from old configurations is properly transferred to newer configurations before they become active. The details of these restrictions and their safety implications are discussed further in Section 3.3.3.

In the remainder of this section we give an overview of the behaviors of *MongoRaftReconfig*, along with a pseudocode description of the protocol. We discuss its correctness in more depth in Section 3.4.

3.3.2 High Level Protocol Behavior

At a high level, dynamic reconfiguration in *MongoRaftReconfig* consists of two main aspects: (1) updating the current configuration and (2) propagating new configurations between servers. Configurations also have an impact on election behavior which we discuss below, in Section 3.3.4. Formally, a *configuration* is defined as a tuple (m, v, t), where $m \in 2^{Server}$ is a member set, $v \in \mathbb{N}$ is a numeric configuration *version*, and $t \in \mathbb{N}$ is the numeric *term* of the configuration. For convenience, we refer to the elements of a configuration tuple $C = (m, v, t)$ as, respectively, $C.m$, $C.v$ and $C.t$. Each server of a replica set maintains a single, durable configuration, and it is assumed that, initially, all nodes begin with a common configuration, $(m_{init}, 1, 0)$, where $m_{init} \in (2^{Server} \setminus \emptyset)$.

Definitions 1 Definitions used in *MongoRaftReconfig* pseudocode.

```
$C_{(i)} \triangleq (config[i], configVersion[i], configTerm[i])$
$C_i > C_j \triangleq (C_i.t > C_j.t) \lor (C_i.t = C_j.t \land C_i.v > C_j.v)$
$C_i \geq C_j \triangleq (C_i > C_j) \lor ((C_i.v, C_i.t) = (C_j.v, C_j.t))$
$Q1(i) \triangleq \exists Q \in Quorums(config[i]) : \forall j \in Q : (C_{(j)}.v, C_{(j)}.t) = (C_{(i)}.v, C_{(i)}.t)$  ▷ Config Quorum Check
$Q2(i) \triangleq \exists Q \in Quorums(config[i]) : \forall j \in Q : term[j] = term[i]$  ▷ Term Quorum Check
$P1(i) \triangleq \exists Q \in Quorums(config[i])$ : all entries committed in terms $\leq term[i]$ are committed in $Q$
```

To update the current configuration of a replica set, a client issues a *reconfiguration* command to a primary server with a new, desired configuration, C'. Reconfigurations can only be executed on primary servers, and they update the primary's current local configuration C to the specified configuration C'. The version of the new configuration, $C'.v$, must be greater than the version of the primary's current configuration, $C.v$, and the term of C' is set equal to the current term of the primary processing the reconfiguration. After a reconfiguration has occurred on a primary, the updated configuration needs to be communicated to other servers in the replica set. This is achieved in a simple, gossip

Algorithm 2 Pseudocode for *MongoRaftReconfig* reconfiguration behavior. Auxiliary definitions shown in Definitions 1.

```
1: State and Initialization
2: Let $m_{init} \in 2^{Server} \setminus \emptyset$
3: $\forall i \in Server$ :
4:   $term[i] \in \mathbb{N}$, initially 0
5:   $state[i] \in \{Pri., Sec.\}$, initially $Secondary$
6:   $config[i] \in 2^{Server}$, initially $m_{init}$
7:   $configVersion[i] \in \mathbb{N}$, initially 1
8:   $configTerm[i] \in \mathbb{N}$, initially 0
9:
10: Actions
11: action: Reconfig($i$, $m_{new}$)
12:     require $state[i] = Primary$
13:     require $Q1(i) \wedge Q2(i) \wedge P1(i)$
14:     require $QuorumsOverlap(config[i], m_{new})$
15:     $config[i] \leftarrow m_{new}$
16:     $configVersion[i] \leftarrow configVersion[i] + 1$
17: end action:
18:
19: action: SendConfig($i, j$)
20:     require $state[j] = Secondary$
21:     require $C_{(i)} > C_{(j)}$
22:     $C_{(j)} \leftarrow C_{(i)}$
23: end action:
24:
25: action: BecomeLeader($i$, $Q$)
26:     require $Q \in Quorums(config[i])$
27:     require $i \in Q$
28:     require $\forall v \in Q : C_{(i)} \geq C_{(v)}$
29:     require $\forall v \in Q : term[i] + 1 > term[v]$
30:     $state[i] \leftarrow Primary$
31:     $state[j] \leftarrow Secondary$, $\forall j \in (Q \setminus \{i\})$
32:     $term[j] \leftarrow term[i] + 1$, $\forall j \in Q$
33:     $configTerm[i] \leftarrow term[i] + 1$
34: end action:
35:
36: action: UpdateTerms($i, j$)
37:     require $term[i] > term[j]$
38:     $state[j] \leftarrow Secondary$
39:     $term[j] \leftarrow term[i]$
40: end action:
```

like manner. Secondaries receive information about the configurations of other servers via periodic heartbeats. They need to have some mechanism, however, for determining whether one configuration is newer than another. This is achieved by totally ordering configurations by their $(version, term)$ pair, where term is compared first, followed by version. If configuration C_j compares as greater than configuration C_i based on this ordering, we say that C_j is *newer* than C_i. A secondary can update its configuration to any that is newer than its current configuration. If it learns that another server has a newer configuration, it will fetch that server's configuration, verify that it is still newer than its own upon receipt, and install it locally.

The above provides a basic outline of how reconfigurations occur and how configurations are propagated between servers in *MongoRaftReconfig*. The pseudocode given in Algorithm 2 gives a more abstract and precise description of these behaviors. Note that, in order for the protocol to operate safely, there are several additional restrictions that are imposed on both reconfigurations and elections, which we discuss in more detail below, in Sections 3.3.3 and 3.3.4.

3.3.3 Safety Restrictions on Reconfigurations

In *MongoStaticRaft*, which does not allow reconfiguration, the safety of the protocol depends on the fact that the *quorum overlap* condition is satisfied for the member sets of any two configurations. This holds since there is a single, uniform configuration that is never modified. For any pair of arbitrary configurations, however, their member sets may not satisfy this property. So, in order for *MongoRaftReconfig* to operate safely, extra restrictions are needed on how nodes are allowed to move between configurations. First, any reconfiguration that moves from C to C' is required to satisfy the quorum overlap condition i.e. $QuorumsOverlap(C.m, C'.m)$. This restriction is discussed in Raft's approach to reconfiguration [64], and is adopted by *MongoRaftReconfig*. Even if quorum overlap is ensured between any two adjacent configurations, it may not be ensured between all configurations that the system passes through over time. So, there are additional preconditions that must be satisfied before a primary server in term T can execute a reconfiguration out of its current configuration C:

Q1. *Config Quorum Check*: There must be a quorum of servers in $C.m$ that are currently in configuration C.

Q2. *Term Quorum Check*: There must be a quorum of servers in $C.m$ that are currently in term T.

P1. *Oplog Commitment*: All oplog entries committed in terms $\leq T$ must be committed on some quorum of servers in $C.m$.

The above preconditions are stated in Algorithm 2 as $Q1(i)$, $Q2(i)$, and $P1(i)$, and they collectively enforce two fundamental requirements needed for safe reconfiguration: *deactivation* of old configurations and *state transfer* from old configurations to new configurations. Q1, when coupled with the election restrictions discussed in Section 3.3.4, achieves deactivation by ensuring that configurations earlier than C can no longer elect a primary. Q2 ensures that term information from older configurations is correctly propagated to newer configurations, while P1 ensures that previously committed oplog entries are properly transferred to the current configuration, ensuring that any primary in a current or later configuration will contain these entries.

3.3.4 Configurations and Elections

When a node runs for election in *MongoStaticRaft*, it must ensure its log is appropriately up to date and that it can garner a quorum of votes in its term. In *MongoRaftReconfig*, there is an additional restriction on voting behavior that depends on configuration ordering. If a replica set server is a candidate for election in configuration C_i, then a prospective voter in configuration C_j may only cast a vote for the candidate if C_i is newer than or equal to C_j. Furthermore, when a node wins an election, it must update its current configuration with its new term before it is allowed to execute subsequent reconfigurations. That is, if a node with current configuration (m, v, t) wins election in term t', it will update its configuration to (m, v, t') before allowing any reconfigurations to be processed. This behavior is necessary to appropriately deactivate concurrent reconfigurations that may occur on primaries in a different term. This configuration re-writing behavior is analogous to the write in Raft's corrected membership change protocol proposed in [9].

3.3.5 Formal Specification

The complete, formal description of *MongoRaftReconfig* is given in the TLA+ specification in the supplementary materials [54]. Note that TLA+ does not impose an underlying system or communication model (e.g. message passing, shared memory), which allows one to write specifications at a wide range of abstraction levels. Our specifications are written at a deliberately high level of abstraction, ignoring some lower level details of the protocol and system model. In practice, we have found the abstraction level of our specifications most useful for understanding and communicating the essential behaviors and safety characteristics of the protocol, while also serving to make automated verification via model checking more feasible, which is discussed further in Section 3.4.4.

3.4 Correctness

In this section we present a brief outline of our safety proof for *MongoRaftReconfig*. We do not address liveness properties in this work. The full proof is left to Appendix A.1.

The key, high level safety property of *MongoRaftReconfig* that we establish in this chapter is *LeaderCompleteness*, which is a fundamental safety property of both standard Raft and *MongoStaticRaft*, and is stated below as Theorem 2. This property states that if a log entry has been committed in term T, then it must be present in the logs of all primary servers in terms $> T$. Essentially, it ensures that writes committed by some primary will be permanently durable in the replica set. Below we give a high level, intuitive outline of the proof.

3.4.1 Overview

Conceptually, *MongoRaftReconfig* can be viewed as an extension of the *MongoStaticRaft* replication protocol that allows for dynamic reconfiguration. *MongoRaftReconfig*, however, violates the property that all quorums of any two configurations overlap, which *Mongo-StaticRaft* relies on for safety. It is therefore necessary to examine how *MongoRaftReconfig* operates safely even though it cannot rely on the quorum overlap property. In *MongoStat-icRaft*, there are two key aspects of protocol behavior that depend on quorum overlap: (1) *elections* of primary servers and (2) *commitment* of log entries. Elections must ensure that there is at most one unique primary per term, referred to as the *ElectionSafety* property. Additionally, if a log entry is committed in a given term, it must be present in the logs of all primary servers in higher terms, referred to as the *LeaderCompleteness* property. Both of these safety properties must be upheld in *MongoRaftReconfig*.

LeaderCompleteness is the essential, high level safety property that we must establish for *MongoRaftReconfig*. *ElectionSafety* is a key, auxiliary lemma that is required in order to show *LeaderCompleteness*. So, this guides the general structure of our proof. Section 3.4.2 presents an intuitive outline of the *ElectionSafety* proof, followed by a similar discussion of *LeaderCompleteness* in Section 3.4.3. The full proofs are left to [50].

3.4.2 Election Safety

In *MongoStaticRaft*, if an election has occurred in term T it ensures that some quorum of servers have terms $\geq T$. This prevents any future candidate from being elected in term T, since the quorum required for any future election will contain at least one of these servers, preventing a successful election in term T. This property, referred to as *ElectionSafety*, is stated below as Lemma 1.

Lemma 1 (Election Safety)**.** *For all $s, t \in Server$ such that $s \neq t$, it is not the case that both s and t are primary and have the same term.*

$$\begin{aligned}&\forall s, t \in Server :\\ &\qquad (state[s] = Primary \land state[t] = Primary \land term[s] = term[t]) \Rightarrow (s = t)\end{aligned}$$

In *MongoRaftReconfig*, ensuring that a quorum of nodes have terms $\geq T$ after an election in term T is not sufficient to ensure that *ElectionSafety* holds, since there is no guarantee that all quorums of future configurations will overlap with those of past configurations. To address this, *MongoRaftReconfig* must appropriately *deactivate* past configurations before creating new configurations. Conceptually, configurations in the protocol can be considered as either or *active* or *deactivated*, the former being any configuration that is not deactivated. Deactivated configurations cannot elect a new leader or execute a reconfiguration. *MongoRaftReconfig* ensures proper deactivation of configurations by upholding an invariant that the quorums of all active configurations overlap with each other. In addition to deactivation of configurations, *MongoRaftReconfig* must also ensure that term information from one configuration is properly transferred to subsequent configurations, so that later configurations know about elections that occurred in earlier configurations. For example, if an election occurred in term T in configuration C, even if C is deactivated by the time C' is created, the protocol must also ensure that C' is "aware" of the fact that an election in T occurred in C. *MongoRaftReconfig* ensures this by upholding an additional invariant stating that the quorums of all active configurations overlap with some server in term $\geq T$, for any past election that occurred in term T.

Collectively, the two above invariants are the essential properties for understanding how the *ElectionSafety* property is upheld in *MongoRaftReconfig*. The formal statement of these invariants and the complete proof is left to [50]. In the following section, we briefly discuss the *LeaderCompleteness* property and its proof, which relies on the *ElectionSafety* property.

3.4.3 Leader Completeness

LeaderCompleteness is the key high level safety property of *MongoRaftReconfig*. It ensures that if a log entry is committed in term T, then it is present in the logs of all leaders in terms $> T$. Essentially, it ensures that committed log entries are durable in a replica set. It is stated below as Theorem 2, where $committed \in \mathbb{N} \times \mathbb{N}$ refers to the set of committed log entries as $(index, term)$ pairs, and $InLog(i, t, s)$ is a predicate determining whether a log entry (i, t) is contained in the log of server s.

Theorem 2 (Leader Completeness)**.** *If a log entry is committed in term T, then it is present in the log of any leader in term $T' > T$.*

$$\begin{aligned} \forall s \in & Server : \forall(cindex, cterm) \in committed : \\ & (state[s] = Primary \land cterm < term[s]) \Rightarrow InLog(cindex, cterm, s) \end{aligned} \tag{3.3}$$

In *MongoStaticRaft*, *LeaderCompleteness* is ensured due to the overlap between quorums used for commitment of a write and quorums used for the election of a primary. In *MongoRaftReconfig*, this does not hold, so the protocol instead upholds a more general invariant, stating that, for all committed entries E, the quorums of all active configurations overlap with some server that contains E in its log. *MongoRaftReconfig* also ensures that newer configurations appropriately disable commitment of log entries in older terms. We defer the statement of these invariants and the complete proof of Theorem 2 and its supporting lemmas to [50].

3.4.4 Model Checking

In addition to the safety proof outlined above, we used TLC [63], an explicit state model checker for TLA+ specifications, to gain additional confidence in the safety of the protocol. We consider it important to augment the human reasoning process for protocols like this with some type of machine based verification, even if the verification is incomplete, since it is easy for humans to make subtle errors in reasoning when considering distributed protocols of this nature.

We verified fixed, finite instances of *MongoRaftReconfig* to provide a sound guarantee of protocol correctness for given parameters. *MongoRaftReconfig* is an infinite state protocol, so verification via explicit state model checking is, necessarily, incomplete. That is, it does not establish correctness of the protocol for an unbounded number of servers or system parameters. It does, however, provide a strong initial level of confidence that the protocol is safe. Further details on these model checking efforts can also be found in Appendix A.2.

Methodology and Results

Formally, *MongoRaftReconfig* behaves as an extension of *MongoStaticRaft* that allows for dynamic reconfiguration. Thus, it can be viewed as a composition of two distinct subprotocols: one for managing the oplog, and one for managing configuration state. The oplog is maintained by *MongoStaticRaft*, and configurations are maintained by a protocol we refer to below as *MongoLoglessDynamicRaft*, which implements the logless replicated state machine that manages the configuration state of the replica set. Algorithm 2 summarizes the behaviors of *MongoLoglessDynamicRaft*. This compositional approach to describing *MongoRaftReconfig* is formalized in our TLA+ specification which can be found in the supplementary materials [54]. Our verification efforts centered on checking the two key safety properties discussed in the above sections, *ElectionSafety* and *LeaderCompleteness*.

We were able to successfully verify the *LeaderCompleteness* property on a finite instance of *MongoRaftReconfig* with 4 servers, logs of maximum length 2, maximum configuration versions of 3, and maximum server terms of 3. That is, we manually imposed a constraint preventing the model checker from exploring any states exceeding these finite bounds. Model checking this instance generated approximately 345 million distinct protocol states and took approximately 8 hours to complete with 20 TLC worker threads on a 48-core, 2.30GHz Intel Xeon Gold 5118 CPU.

As evidenced by the above metrics, it was difficult to scale verification of the *LeaderCompleteness* property to much larger system parameters. So, to provide additional confidence, we checked the *ElectionSafety* property on the *MongoLoglessDynamicRaft* protocol in isolation, which allowed us to verify instances with significantly larger parameters. Due to the compositional structure of *MongoRaftReconfig*, verifying that the *ElectionSafety* property holds on *MongoLoglessDynamicRaft* is sufficient to ensure that it holds in *MongoRaftReconfig*. Intuitively, the additional preconditions imposed by *MongoRaftReconfig* only *restrict* the behaviors of *MongoLoglessDynamicRaft*, but do not augment them. We formalize and prove this fact via a refinement based argument, whose details are left to Appendix A.3. This allows us to assume our verification efforts for *MongoLoglessDynamicRaft* hold in *MongoRaftReconfig*, providing stronger confidence in the correctness of the overall protocol.

We successfully verified the *ElectionSafety* property on a finite instance of *MongoLoglessDynamicRaft* with 5 servers, maximum configuration versions of 4, and maximum terms of 4. Model checking this instance generated approximately 812 million distinct states and took around 19.5 hours to complete with 20 TLC worker threads on a 48-core, 2.30GHz Intel Xeon Gold 5118 CPU. The ability to check these considerably larger parameter values in only several extra hours of wall clock time demonstrates the effectiveness of this compositional model checking approach, helping us mitigate state space explosion [67].

3.5 Conceptual Insights

MongoRaftReconfig can be viewed as a generalization and optimization of the standard Raft reconfiguration protocol. To explain the conceptual novelties of our protocol and how it relates to standard Raft, we discuss below the two primary aspects of the protocol which set it apart from Raft: (1) decoupling of the oplog and config state machine and (2) logless optimization of the config state machine. These are covered in Sections 3.5.1 and 3.5.2, respectively. Section 3.6 provides an experimental evaluation of how these novel aspects can provide performance benefits for reconfiguration, by allowing reconfigurations to bypass the main operation log in cases where it has become slow or stalled.

3.5.1 Decoupling Reconfigurations

In standard Raft, the main operation log is used for both normal operations and reconfiguration operations. This coupling between logs has the benefit of providing a single, unified data structure to manage system state, but it also imposes fundamental restrictions on the operation of the two logs. Most importantly, in order for a write to commit in one log, it must commit all previous writes in the other. For example, if a reconfiguration log entry C_j has been written at log index j on primary s, and there is a sequence of uncommitted log entries $U = \langle i, i+1, ..., j-1 \rangle$ in the log of s, in order for a reconfiguration from C_j to C_k to occur, all entries of U must become committed. This behavior, however, is stronger than necessary for safety i.e. it is not strictly necessary to commit these log entries before executing a reconfiguration. The only fundamental requirements are that previously committed log entries are committed by the rules of the current configuration, and that the current configuration has satisfied the necessary safety preconditions. Raft achieves this goal implicitly, but more conservatively than necessary, by committing the entry C_j and all entries behind it. This ensures that all previously committed log entries, in addition to the uncommitted operations U, are now committed in C_j, but it is not strictly necessary to pipeline a reconfiguration behind commitment of U. *MongoRaftReconfig* avoids this by separating the oplog and config state machine and their rules for commitment and reconfiguration, allowing reconfigurations to bypass the oplog if necessary. Section 3.6 examines this aspect of the protocol experimentally.

3.5.2 Logless Optimization

Decoupling the config state machine from the main operation log allows for an optimization that is enabled by the fact that reconfigurations are "update-only" operations on the replicated state machine. This means that it is sufficient to store only the latest version of the replicated state, since the latest version can be viewed as a "rolled-up" version of the entire (infinite) log. This logless optimization allows the configuration state machine to avoid complexities related to garbage collection of old log entries and it simplifies the mechanism for state propagation between servers. Normally, log entries are replicated incrementally, either one at a time, or in batches from one server to another. Additionally, servers may need to have an explicit procedure for deleting (i.e. rolling back) log entries that will never become committed. In the logless replicated state machine, all of these mechanisms can be combined into a single conceptual action, that atomically transfers the entire log of server s to another server t, if the log of s is newer, based on the index and term of its last entry. In *MongoRaftReconfig*, this is implemented by the *SendConfig* action, which transfers configuration state from one server to another.

3.6 Experimental Evaluation

In a healthy replica set, it is possible that a failure event causes some subset of replica set servers to degrade in performance, causing the main oplog replication channel to become lagged or stall entirely. If this occurs on a majority of nodes, then the replica set will be prevented from committing new writes until the performance degradation is resolved. For example, consider a 3 node replica set consisting of nodes $\{n0, n1, n2\}$, where nodes $n1$ and $n2$ suddenly become slow or stall replication. An operator or failure detection module may want to reconfigure these nodes out of the set and add in two new, healthy nodes, $n3$ and $n4$, so that the system can return to a healthy operational state. This requires a series of two reconfigurations, one to add $n3$ and one to add $n4$. In standard Raft, this would require the ability to commit at least one reconfiguration oplog entry with one of the degraded nodes ($n1$ or $n2$). This prevents such a reconfiguration until the degradation is resolved. In *MongoRaftReconfig*, reconfigurations bypass the oplog replication channel, committing without the need to commit writes in the oplog. This allows *MongoRaftReconfig* to successfully reconfigure the system in such a degraded state, restoring oplog write availability by removing the failed nodes and adding in new, healthy nodes.

Note that if a replica set server experiences a period of degradation (e.g. a slow disk), both the oplog and reconfiguration channels will be affected, which would seem to nullify the benefits of decoupling the reconfiguration and oplog replication channels. In practice, however, the operations handled by the oplog are likely orders of magnitude more resource intensive than reconfigurations, which typically involve writing a negligible

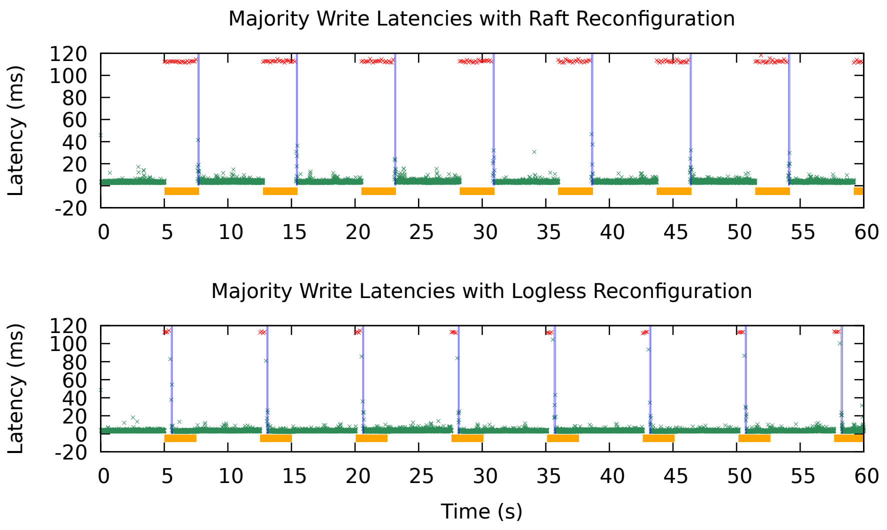


Figure 3.1: Latency of majority writes in the face of node degradation and reconfiguration to recover. Red points indicate writes that timed out i.e. failed to commit. Orange horizontal bars indicate intervals of time where system entered a *degraded* mode. Thin, vertical blue bars indicate successful completion of reconfiguration events.

amount of data. So, even on a degraded server, reconfigurations should be able to complete successfully when more intensive oplog operations become prohibitively slow, since the resource requirements of reconfigurations are extremely lightweight.

3.6.1 Experiment Setup and Operation

To demonstrate the benefits of *MongoRaftReconfig* in this type of scenario, we designed an experiment to measure how quickly a replica set can reconfigure in new nodes to restore majority write availability when it faces periodic phases of degradation. For comparison, we implemented a simulated version of the Raft reconfiguration algorithm in MongoDB by having reconfigurations write a no-op oplog entry and requiring it to become committed before the reconfiguration can complete [68]. Our experiment initiates a 5 node replica set with servers we refer to as $\{n0, n1, n2, n3, n4\}$. We run the server processes co-located on a single Amazon EC2 t2.xlarge instance with 4 vCPU cores, 16GB memory, and a 100GB EBS disk volume, running Ubuntu 20.04. Co-location of the server processes is acceptable since the workload of the experiment does not saturate any resource (e.g. CPU, disk) of the machine. The servers run MongoDB version v4.4-39f10d with a patch to fix a minor bug [69] that prevents optimal configuration propagation speed in some cases.

Initially, $\{n0, n1, n2\}$ are voting servers and $\{n3, n4\}$ are non voting. In a MongoDB replica set, a server can be assigned either 0 or 1 votes. A non-voting server has zero votes and it does not contribute to a commit majority i.e. it is not considered as a member of the consensus group. Our experiment has a single writer thread that continuously inserts small documents into a collection with write concern *majority*, with a write concern timeout of 100 milliseconds. There is a concurrent fault injector thread that periodically simulates a degradation of performance on two secondary nodes by temporarily pausing oplog replication on those nodes. This thread alternates between *steady* periods and *degraded* periods of time, starting out in *steady* mode, where all nodes are operating normally. It runs for 5 seconds in *steady* mode, then transitions to *degraded* mode for 2.5 seconds, before transitioning back to *steady* mode and repeating this cycle. When the fault injector enters *degraded* mode, the main test thread simulates a "fault detection" scenario (assuming some external module detected the performance degradation) by sleeping for 500 milliseconds, and then starting a series of reconfigurations to add two new, healthy secondaries and remove the two degraded secondaries. Over the course of the experiment, which has a 1 minute duration, we measure the latency of each operation executed by the writer thread. These latencies are depicted in the graphs of Figure 3.1. Red points indicate writes that failed to commit i.e. that timed out at 100 milliseconds. The successful completion of reconfigurations are depicted with vertical blue bars. It can be seen how, when a period of degradation begins, the logless reconfiguration protocol is able to complete a series of reconfigurations quickly to get the system back to a healthy state, where writes are able to commit again and latencies drop back to their normal levels. In the case of Raft reconfiguration, writes continue failing until the period of degradation ends, since the reconfigurations to add in new healthy nodes cannot complete.

3.7 Formal Safety Verification

In this section, we present the formal verification of the safety properties of *MongoRaftReconfig*. We present a formally stated inductive invariant for the protocol, which we prove and then utilize to establish two high level safety properties of the protocol. In particular, we prove (1) *LeaderCompleteness*, which, intuitively, states that if a log entry is committed it is durable, and (2) *StateMachineSafety*, which says that log entries committed at a particular index must be consistent across all nodes in the system. We carry out our verification efforts using TLAPS, the TLA+ proof system [70]. To our knowledge, this is the first machine checked inductive invariant and safety proof of a reconfiguration protocol for a Raft based replication system.

All of our TLA+ specifications, TLAPS proof code, and instructions for checking our proofs are included in the supplementary material [71] for this chapter. Where appropriate throughout this chapter, we cite the relevant files located in this material.

```
ASSUME $Conditions$
PROVE $Implication$
PROOF
  $\langle 1 \rangle 1.$ $Statement1.1$ BY DEF $Conditions$
  $\langle 1 \rangle 2.$ $Statement1.2$
    $\langle 2 \rangle 1.$ $Statement2.1$ BY $UsefulTheorem$
    $\langle 2 \rangle 2.$ $Statement2.2$ BY $\langle 1 \rangle 1$, $\langle 2 \rangle 1$
    $\langle 2 \rangle .$ QED BY $\langle 2 \rangle 2$
  $\langle 1 \rangle .$ QED BY $\langle 1 \rangle 1$, $\langle 1 \rangle 2$
```

Figure 3.2: Example of a hierarchically structured TLAPS proof.

3.7.1 The TLA+ Proof System

The TLA+ proof system [70], abbreviated as TLAPS, is an accompanying tool for the TLA+ language that allows one to write and mechanically check hierarchically structured proofs [72] in TLA+. Proofs consist of a series of statements that support the proof goal, which is the top level statement that must be proved. Each statement, in turn, must be proved either compositely using a nested structural proof, or as a leaf proof via a backend solver. TLAPS is independent of any particular SMT solver or theorem prover, and includes support for various backends e.g. Z3 [28], Isabelle [73], and Zenon [74].

Figure 3.2 shows an example of a lemma and its proof in TLAPS. The ASSUME-PROVE idiom treats the lemma as an implication. That is, if *Conditions* hold, then *Implication* must follow. Leaf statements are proved using the BY statement, and can reference theorems and lemmas by name, operator definitions, and previous statements by label. Each structural proof must end with a QED statement, closing the goal of either a nested or overall proof.

3.7.2 The MongoRaftReconfig TLA+ Specification

A formal TLA+ specification of *MongoRaftReconfig* was originally included in [50], but was not discussed in detail. This same specification serves as the basis for the TLAPS proofs presented in this chapter, so we give a brief overview of the specification here. The complete specification can be found in the *MongoRaftReconfig.tla* file of the supplementary material provided with this chapter [71].

The state variables of the specification and their types are shown in Figure 3.3. The initial states, next state relation, and specification definition of *MongoRaftReconfig* are summarized in Figure 3.4. The operator $Quorums(m)$ is defined as the set of all majority quorums [75] for a given set of servers m. Reconfigurations are modeled by the $Reconfig(s, m)$ action, which represents a reconfiguration that occurs on primary server s to a new configuration with member set $m \in 2^{Server}$. Configuration propagation is modeled by the $SendConfig(s, t)$ action, which represents the propagation of a configuration from server s to server t. Elections are modeled by the action $BecomeLeader(s, Q)$, which

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{TypeOK} \triangleq} \\
\quad \mathit{log} & \in [\mathit{Server} \rightarrow \mathit{Seq}(\mathbb{N})] \\
\quad \mathit{committed} & \in 2^{\mathbb{N} \times \mathbb{N}} \\
\quad \mathit{term} & \in [\mathit{Server} \rightarrow \mathbb{N}] \\
\quad \mathit{state} & \in [\mathit{Server} \rightarrow \{\mathit{Primary}, \mathit{Secondary}\}] \\
\quad \mathit{config} & \in [\mathit{Server} \rightarrow 2^{\mathit{Server}}] \\
\quad \mathit{configVersion} & \in [\mathit{Server} \rightarrow \mathbb{N}] \\
\quad \mathit{configTerm} & \in [\mathit{Server} \rightarrow \mathbb{N}]
\end{array}
$$

Figure 3.3: The state variables of the *MongoRaftReconfig* protocol and their corresponding types stated as a type correctness predicate in TLA+. The notation $[A \rightarrow B]$ represents the set of all functions from set A to set B and $\mathit{Seq}(S)$ represents the set of all sequences containing elements from the set S.

represents the election of server s by a set of voters Q. The action $\mathit{UpdateTerms}(s, t)$ propagates the term of a server s to server t, if the term of s is newer than t. The actions $\mathit{ClientRequest}(s)$, $\mathit{GetEntries}(s, t)$, $\mathit{RollbackEntries}(s, t)$, and $\mathit{CommitEntry}(s, Q)$ are responsible for log related actions that are conceptually unrelated to reconfiguration, so we do not discuss their details here. Their full definitions can be found in the specifications provided in the supplementary material [71].

Note that our specifications are written at a deliberately high level of abstraction, ignoring some lower level details of the protocol. In practice, we have found the abstraction level of our specifications most useful for understanding and communicating the essential behaviors and safety characteristics of the protocol, while also serving to make our automated verification and proof efforts more feasible. In the future, however, we believe it would be valuable to explore techniques for formally relating our abstract specifications to real world protocol implementations, with an aim of verifying whether a system implementation faithfully reflects our high level specifications [76–78].

3.7.3 Verification Problem Statement

We establish that the *MongoRaftReconfig* protocol satisfies *LeaderCompleteness* and *StateMachineSafety*, which are two key, high level safety properties of both the MongoDB replication system and standard Raft. Informally, the *LeaderCompleteness* property states that if a log entry is committed in term T, then it is present in the log of any leader in term $T' > T$. It is stated more precisely in Definition 1, where $\mathit{committed} \in \mathbb{N} \times \mathbb{N}$ refers to the set of committed log entries as $(\mathit{index}, \mathit{term})$ pairs, and $\mathit{InLog}(i, t, s)$ is a predicate determining whether a log entry (i, t) is contained in the log of server s. *StateMachineSafety* states that if two log entries are committed at the same log index, these entries must be the same, and is stated formally as Definition 2.

```
MODULE MongoRaftReconfig
$MRRInit \triangleq$
    $\land\ log = [i \in Server \mapsto \langle\rangle]$
    $\land\ committed = \{\}$
    $\land\ currentTerm = [i \in Server \mapsto 0]$
    $\land\ state = [i \in Server \mapsto Secondary]$
    $\land\ configVersion = [i \in Server \mapsto 1]$
    $\land\ configTerm = [i \in Server \mapsto 0]$
    $\land\ \exists\, initConfig \in 2^{Server} :$
      $\land\ initConfig \neq \emptyset$
      $\land\ config = [i \in Server \mapsto initConfig]$

$MRRNext \triangleq$
   $\exists\, s,\ t \in Server :$
   $\exists\, Q \in Quorums(config[s]) :$
      $\lor\ ClientRequest(s)$
      $\lor\ GetEntries(s,\ t)$
      $\lor\ RollbackEntries(s,\ t)$
      $\lor\ CommitEntry(s,\ Q)$
      $\lor\ SendConfig(s,\ t)$
      $\lor\ Reconfig(s, m)$
      $\lor\ BecomeLeader(s,\ Q)$
      $\lor\ UpdateTerms(s,\ t)$

$MRRSpec \triangleq MRRInit \land \Box[MRRNext]_{vars}$
```

Figure 3.4: Summary of the *MongoRaftReconfig* TLA+ specification. The full specification consists of 359 lines of TLA+ code, excluding comments, and can be found in the *MongoRaftReconfig.tla* file of the supplementary material [71].

Definition 1 (Leader Completeness)**.**

$$\begin{aligned}&\forall s \in Server : \forall (cindex, cterm) \in committed : \\ &\quad (state[s] = Primary \land cterm < term[s]) \Rightarrow \\ &\quad InLog(cindex, cterm, s)\end{aligned}$$

Definition 2 (State Machine Safety)**.**

$$\begin{aligned}&\forall (ind_i, t_i),(ind_j, t_j) \in committed : \\ &\qquad (ind_i = ind_j) \Rightarrow (t_i = t_j)\end{aligned}$$

Both *LeaderCompleteness* and *StateMachineSafety* are safety properties. More specifically, they are both invariants, meaning that they must hold in all reachable states of *MongoRaftReconfig*. Thus, our verification goals can be stated formally as Theorems 3 and 4, where $MRRSpec$ refers to the specification of *MongoRaftReconfig* as given in Figure 3.4.

Theorem 3 (MRRImpliesLeaderCompleteness)**.**

$$MRRSpec \Rightarrow \Box LeaderCompleteness$$

Theorem 4 (MRRImpliesStateMachineSafety)**.**

$$MRRSpec \Rightarrow \Box StateMachineSafety$$

Theorems 3 and 4 are the safety results established and formally verified in this chapter, and they can be found in the *MongoRaftReconfigProofs.tla* file of our supplementary material [71]. The proofs of Theorems 3 and 4 are discussed in Section 3.7.5. Both of these theorems are proved using the help of an inductive invariant, which we discuss next, in Section 3.7.4.

3.7.4 Our Inductive Invariant

To prove that a given system specification, $Spec = Init \land \Box[Next]_{vars}$, satisfies an invariant, Inv, one must find an *inductive invariant* that is sufficient to imply Inv[79]. Formally, a state predicate Inv is an invariant of a system $Spec$ if the following holds:

$$Spec \Rightarrow \Box Inv \tag{3.4}$$

Suppose that $Spec$ is of the form $Spec = Init \land \Box[Next]_{vars}$, as in the case of *MongoRaftReconfig*. Then, in order to establish Formula 3.4, it is sufficient to find a state predicate Ind such that the following conditions hold:

$$Init \Rightarrow Ind \tag{3.5}$$

$$Ind \land Next \Rightarrow Ind' \tag{3.6}$$

$$Ind \Rightarrow Inv \tag{3.7}$$

$$
\begin{array}{ll}
 & \mathit{MRRInd} \triangleq \\
T & \left\{ \begin{array}{l} \land\ \mathit{TypeOK} \end{array} \right. \\
E_1 & \left\{ \begin{array}{l} \land\ \mathit{ElectionSafety} \\ \land\ \mathit{PrimaryConfigTermEqualToCurrentTerm} \\ \land\ \mathit{ConfigVersionAndTermUnique} \\ \land\ \mathit{PrimaryInTermContainsNewestConfigOfTerm} \\ \land\ \mathit{ActiveConfigsOverlap} \\ \land\ \mathit{ActiveConfigsSafeAtTerms} \end{array} \right. \\
L_1 & \left\{ \begin{array}{l} \land\ \mathit{LogEntryInTermImpliesConfigInTerm} \\ \land\ \mathit{PrimaryHasEntriesItCreated} \\ \land\ \mathit{LogMatching} \end{array} \right. \\
L_2 & \left\{ \begin{array}{l} \land\ \mathit{PrimaryTermAtLeastAsLargeAsLogTerms} \\ \land\ \mathit{TermsOfEntriesGrowMonotonically} \\ \land\ \mathit{UniformLogEntriesInTerm} \end{array} \right. \\
C_1 & \left\{ \begin{array}{l} \land\ \mathit{CommittedEntryIndexesAreNonZero} \\ \land\ \mathit{CommittedTermMatchesEntry} \end{array} \right. \\
C_2 & \left\{ \begin{array}{l} \land\ \mathit{LeaderCompleteness} \\ \land\ \mathit{LogsLaterThanCommittedMustHaveCommitted} \\ \land\ \mathit{ActiveConfigsOverlapWithCommittedEntry} \\ \land\ \mathit{NewerConfigsDisableCommitsInOlderTerm} \end{array} \right. \\
N & \left\{ \begin{array}{l} \land\ \mathit{ConfigsNonEmpty} \end{array} \right.
\end{array}
$$

Figure 3.5: Our inductive invariant for *MongoRaftReconfig*.

The inductive invariant that we developed for *MongoRaftReconfig* is referred to as MRRInd and consists of 20 high level conjuncts, shown in Figure 3.5. Its full definition is given in 140 lines of TLA+ code, and is provided in the *MongoRaftReconfigIndInv.tla* file of our supplementary material [71].

The inductive invariant, shown in Figure 3.5, is composed of several conceptually distinct subcomponents. The first conjunct, TypeOK, establishes basic type-correctness constraints on the state variables of *MongoRaftReconfig*. This is necessary in most cases when stating inductive invariants in TLA+, since it is an untyped formalism [80]. The full definition of TypeOK is shown in Figure 3.3. The initial set of 6 conjuncts, labeled as E_1 in Figure 3.5, along with TypeOK, is itself an inductive invariant, and it establishes the $\mathit{ElectionSafety}$ property, a key auxiliary invariant of the protocol that is needed to establish $\mathit{LeaderCompleteness}$. These lemmas, some of which are shown in more detail in Figure 3.6, collectively ensure properties about how configurations should be *deactivated*

$$CV(i) \triangleq \langle \mathit{configVersion}[i], \mathit{configTerm}[i] \rangle$$

$$\mathit{NewerConfig}(ci, cj) \triangleq (ci[2] > cj[2]) \lor (ci[2] = cj[2] \land ci[1] > cj[1])$$

$$\mathit{ConfigDisabled}(i) \triangleq \forall Q \in \mathit{Quorums}(\text{config}[i]) : \exists n \in Q : \mathit{NewerConfig}(CV(n), \text{CV}(i))$$

$$\mathit{ActiveConfigSet} \triangleq \{s \in \text{Server} : \neg \mathit{ConfigDisabled}(s)\}$$

$$\begin{aligned}
&\mathit{ActiveConfigsOverlap} \triangleq \\
&\quad \forall s, t \in \mathit{ActiveConfigSet} : \mathit{QuorumsOverlap}(\mathit{config}[s], \mathit{config}[t])
\end{aligned}$$

$$\begin{aligned}
&\mathit{ActiveConfigsSafeAtTerms} \triangleq \\
&\quad \forall s \in \mathit{Server} : \\
&\quad \forall t \in \mathit{ActiveConfigSet} : \\
&\quad \forall Q \in \mathit{Quorums}(\mathit{config}[t]) : \\
&\qquad \exists n \in Q : \mathit{currentTerm}[n] \geq \mathit{configTerm}[s]
\end{aligned}$$

Figure 3.6: Excerpted definitions of key lemmas in the inductive invariant for *MongoRaftReconfig*, related to the concept of configuration deactivation. Intuitively, a configuration is *active* if it can conduct a new election or commit new log entries. To ensure safety, reconfigurations must ensure that the appropriate configurations are deactivated (e.g. disabled) before proceeding.

before new configurations can be created. Essentially, the *ActiveConfigsOverlap* property serves as a generalization of the static *quorum overlap* property in standard Raft, in the context of dynamic reconfigurations. Similarly, the *ActiveConfigsSafeAtTerms* property ensures that any active configurations have properly recorded the necessary quorum term state from prior elections. For example, if an election has occurred in term T, then a quorum of servers must be at term T or greater.

The conjuncts in group L_1 are a set of invariants related to logs of servers in the system, and they collectively establish $\mathit{LogMatching}$, another important auxiliary invariant. The L_2 group establishes a few additional log related invariants, which rely on previous conjuncts. In general, these log related conjuncts are not fundamentally related to dynamic reconfiguration, but are necessary to state precisely for a protocol that manages logs in a Raft like fashion. Group C_1 establishes some required, trivial aspects of the set of committed log entries. The conjunct group C_2 establishes the high level $\mathit{LeaderCompleteness}$ property, by relating how configurations interact with the set of committed log entries present in the system. Finally, the last conjunct, labeled as N, asserts that every configuration is non empty i.e. it contains some servers. This is an auxiliary invariant that is helpful for proving other facts.

Discovering an Inductive Invariant Discovering such an inductive invariant for a protocol of this complexity is non-trivial. To our knowledge, this is the first inductive invariant proposed for a dynamic reconfiguration protocol that is built on a Raft based replication system. The discovery of $MRRInd$ took approximately 1-2 human months of work and it involved repeated efforts of iteration and refinement. To aid in this discovery process, we leveraged a technique proposed in [81] that utilizes the TLC explicit state model checker [82] to probabilistically verify candidate inductive invariants. If a candidate Inv is not inductive, the TLC model checker can, with some probability, report a *counterexample to induction* (CTI). A *counterexample to induction* is a state transition $s \rightarrow t$ satisfying $MRRNext$, where s satisfies Inv and t violates Inv. We also considered using the Apalache symbolic model checker for TLA+ [83], which converts TLA+ into an SMT-based symbolic encoding. We found, however, working with its type inference and checking routines tedious and this was ultimately a significant usability barrier for our purposes.

These CTIs are helpful to understand why a candidate invariant fails to be inductive, and how it may need to be modified or strengthened further. This probabilistic method can only be used on finite protocol instances, and it does not provide a proof that an invariant is inductive. Nevertheless, the technique proved to be highly effective, as it helped us to discover an inductive invariant that we eventually proved formally correct using TLAPS, as discussed more in Section 3.7.5. Furthermore, we did not discover any errors in our inductive invariant during the TLAPS proof process.

Note that, although having a tool for finding counterexamples to induction is helpful for finding errors in candidate inductive invariants, it still does not provide much guidance in developing an inductive invariant from scratch. That is, it does not necessarily provide a systematic methodology for converging to a correct inductive invariant. Rather, development of our inductive invariant still required a large amount of creativity and human reasoning, largely driven by strong prior intuitions about the correctness of the protocol. For example, rather than aiming to develop the entire inductive invariant at once, we were able to develop it in components, based partially on human intuition about certain auxiliary lemmas that we knew must hold true of the overall protocol. For example, the $ElectionSafety$ and the $LogMatching$ invariants (shown in Figure 3.5) are two such lemmas that we worked on establishing first, before moving on to discover the additional conjuncts needed to establish the $LeaderCompleteness$ property.

3.7.5 Our TLAPS Proofs

To establish that $MRRInd$ is an inductive invariant, we must prove that $MRRInd$ satisfies both the initiation (3.5) and consecution (3.6) conditions for *MongoRaftReconfig*, as described in Section 3.7.4. This is captured in Lemma 5.

Lemma 5 (*MRRInd* is an inductive invariant)**.**

$$MRRInit \Rightarrow MRRInd \tag{a}$$

$$MRRInd \land MRRNext \Rightarrow MRRInd' \tag{b}$$

Cases (a) and (b) of Lemma 5 represent, respectively, initiation and consecution. The initiation case of Lemma 5 follows in a straightforward manner from the definitions of $MRRInit$ and $MRRInd$. Proving the consecution case of Lemma 5, however, is the most difficult and time consuming aspect of the verification efforts presented in this chapter. At a high level, this proof consists of showing that, assuming $MRRInd$ holds in a current state, every transition of the protocol upholds $MRRInd$ in the next state. To break this verification problem into smaller steps, we decompose the proof first by each conjunct of $MRRInd$, and then we decompose by each protocol transition.

Specifically, consider the definition of $MRRInd$, which is composed of 20 conjuncts (as shown in Figure 3.5):

$$MRRInd \triangleq I_1 \land I_2 \land \ldots \land I_{20}$$

Our first decomposition step breaks down case (b) of Lemma 5 into the following, independent proof goals, one for each conjunct of $MRRInd$:

$$\begin{aligned} MRRInd \land MRRNext &\Rightarrow I_1' \\ MRRInd \land MRRNext &\Rightarrow I_2' \\ &\vdots \\ MRRInd \land MRRNext &\Rightarrow I_{20}' \end{aligned} \tag{3.8}$$

Furthermore, $MRRNext$ is the disjunction of eight protocol actions (as shown in Figure 3.4):

$$MRRNext \triangleq A_1 \lor A_2 \lor \cdots \lor A_8 \tag{3.9}$$

So, we further decompose each goal of Statement 3.8 into one case for each protocol action. That is, we decompose each goal $MRRInd \land MRRNext \Rightarrow I_j'$ into the following proof goals:

$$\begin{aligned} MRRInd \land A_1 &\Rightarrow I_j' \\ MRRInd \land A_2 &\Rightarrow I_j' \\ &\vdots \\ MRRInd \land A_8 &\Rightarrow I_j' \end{aligned} \tag{3.10}$$

Our proof follows this methodology for every conjunct of $MRRInd$ and every action of $MRRNext$. This produces a set of proof goals whose size is the product of the number of protocol actions (8) and the number of invariant conjuncts (20), totaling $8 * 20 = 160$ proof goals. This decomposition allowed us to focus on proving one, small goal at a time, while incrementally building a library of reusable lemmas. The TLAPS proof of Lemma 5 can be found in the *MongoRaftReconfigProofs.tla* file of our supplementary material [71], while our library of lemmas can be found in the *Lib.tla*, *BasicQuorumsLib.tla*, and *LeaderCompletenessLib.tla* files.

Lemma 5 establishes that *MRRInd* is an inductive invariant of *MongoRaftReconfig*. In this section we provide an overview of our proofs for establishing that *MongoRaftReconfig* satisfies *LeaderCompletness* and *StateMachineSafety* (Theorems 3 and 4), which utilize $MRRInd$. We do this by establishing lemmas 6 and 7, which, together with Lemma 5, are sufficient to establish Theorems 3 and 4.

Lemma 6.

$$MRRInd \Rightarrow LeaderCompleteness$$

Lemma 7 (IndImpliesStateMachineSafety)**.**

$$MRRInd \Rightarrow StateMachineSafety$$

LeaderCompleteness is a conjunct of $MRRInd$ so the implication of Lemma 6 follows trivially. The proof of Lemma 7, which can be found in the *StateMachineSafetyLemmas.tla* file of our supplementary material [71], is not trivial and we present it in the following section as a concrete example of a TLAPS proof.

We present the proof of Lemma 7 to serve as an example of TLAPS. The proof relies on one additional lemma, stated as Lemma 8. The proof of Lemma 8 is contained in the *StateMachineSafetyLemmas.tla* file of the supplementary material [71].

Lemma 8 (CommitsAreLogEntries)**.**

$$\begin{aligned} &MRRInd \Rightarrow \\ &\quad \forall c \in committed : \exists s \in Server : \\ &\qquad InLog(c.entry, s) \end{aligned}$$

The TLAPS proof of Lemma 7 is shown in Figure 3.7. The proof uses the ASSUME-PROVE idiom to show that *MRRInd* implies *StateMachineSafety*. By the definition of the *StateMachineSafety* property, we can establish the proof goal given in $\langle 1\rangle 1$. Steps $\langle 1\rangle 2$ and $\langle 1\rangle 3$ assume that $c1$ and $c2$ are arbitrary committed entries that share the same index but are not identical, and PROVE FALSE OBVIOUS establishes that these assumptions will lead to a contradiction. $\langle 1\rangle 4$ is a composite proof that shows that $c1$ and $c2$ cannot share the same term. Steps $\langle 1\rangle 5$ through $\langle 1\rangle 8$ use Lemma 8 to show that there exist servers $s1$ and $s2$ that respectively contain the committed entries $c1$ and $c2$ in their logs. The two cases $\langle 1\rangle 9$ and $\langle 1\rangle 10$ show that if either $c1$ or $c2$ has a larger term than the other, then we derive a contradiction as expected. Finally, it suffices to only consider cases $\langle 1\rangle 9$ and $\langle 1\rangle 10$ because of step $\langle 1\rangle 4$, and hence the proof is complete.

3.7.6 TLAPS Proof Statistics and Discussion

We now present some summary statistics about our TLAPS proof and its development to give a better sense of its scope, size, and difficulty. The entire TLAPS proof, including the statement of the inductive invariant and the protocol specification, consists of 3189 lines of TLA+ code, excluding comments. 140 of these lines are used for defining the inductive invariant and 359 of these lines are used for specifying the *MongoRaftReconfig* protocol. There are a total of 3 top level theorems and 78 formally stated lemmas. In terms of proof effort, we spent approximately 4 human-months on development of the TLAPS proof, which does not include the time to develop the inductive invariant described in Section 3.7.4. Development of the inductive invariant took approximately an additional 1-2 human-months of work. For the TLAPS proof system to check the correctness of the completed proof from scratch it takes approximately 38 minutes on a 2020 Macbook Air using 8 Apple M1 CPU Cores. This computation time consists mostly of queries to an underlying backend solver e.g. Isabelle or an SMT solver.

We note that the hierarchical structure enforced by TLAPS led to well organized and generally readable proofs in our experience. Despite our overall positive experience, there two main shortcomings of TLAPS that we highlight below. First, TLAPS does not offer much guidance when a backend solver fails on a leaf proof. In general, TLAPS does not distinguish between obligations that fail because they are false, versus obligations that are too difficult for the backend solvers. Second, we found that the TLAPS library did not always cater to our needs as conveniently as we hoped. For example, the *MongoRaftReconfig* specification includes state variables that are represented as TLA+ sequences, which are indexed using $\mathbb{N} \setminus \{0\}$. While the TLAPS standard library has theorems for induction on $\mathbb{N}$ (*NaturalsInduction.tla*), we were not able to find direct support for induction over the domain of sequences. Support for induction over the domain of sequences was not seamless, yet we were able to prove the desired theorem by tailoring parts of the library to our needs.

Formally verifying safety properties for a large, real world distributed protocol is, in our experience, a very labor intensive task. Even if one has built up strong intuitions about correctness of a protocol, verification may take several months. Nevertheless, we believe that formal verification is of great value since, even for protocols that have been formally specified or model checked, design errors are still possible. For example, a safety bug in EPaxos [84], a well known variant of the original Paxos protocol, was discovered several years after its initial publication [10], even though EPaxos was accompanied by a TLA+ specification and manual safety proof in its original publication. Similarly, a bug in one of Raft's original reconfiguration protocols was also discovered after initial publication [9].

Furthermore, developing a formal inductive invariant and safety proof often provides deeper insights into why a protocol is correct, which fully automated techniques like model checking, on their own, are often unable to provide. Gaining deeper, formalized understanding of why a protocol is correct is valuable both from a theoretical perspective and also for system designers and engineers who may implement these protocols with extensions, modifications, or optimizations.

3.8 Related Work

Dynamic reconfiguration in consensus based systems has been explored from a variety of perspectives for Paxos based systems. In Lamport's presentation of Paxos [85], he suggests using a fixed parameter α such that the configuration for a consensus instance i is governed by the configuration at instance $i - \alpha$. This restricts the number of commands that can be executed until the new configuration becomes committed, since the system cannot execute instance i until it knows what configuration to use, potentially causing availability issues if reconfigurations are slow to commit. Stoppable Paxos [58] was an alternative method later proposed where a Paxos system can be reconfigured by stopping the current state machine and starting up a new instance of the state machine with a potentially different configuration. This "stop-the-world" approach can hurt availability of the system while a reconfiguration is being processed. Vertical Paxos allows a Paxos state machine to be reconfigured in the middle of reaching agreement, but it assumes the existence of an external configuration master [86]. In [15], the authors describe the Paxos implementation underlying Google's Chubby lock service, but do not include details of their approach to dynamic reconfiguration, stating that "While group membership with the core Paxos algorithm is straightforward, the exact details are non-trivial when we introduce Multi-Paxos...". They remark that the details, though minor, are "...subtle and beyond the scope of this paper".

The Raft consensus protocol, published in 2014 by Ongaro and Ousterhout [56], presented two methods for dynamic membership changes: single server membership change and joint consensus. A correctness proof of the core Raft protocol, excluding dynamic reconfiguration, was included in Ongaro's PhD dissertation [64]. Formal verification of Raft's linearizability guarantees was later completed in Verdi [87], a framework for verifying distributed systems in the Coq proof assistant [88], but formalization of dynamic reconfiguration was not included. In 2015, after Raft's initial publication, a safety bug in the single server reconfiguration approach was found by Amos and Zhang [89], at the time PhD students working on a project to formalize parts of Raft's original reconfiguration algorithm. A fix was proposed shortly after by Ongaro [9], but the project was never extended to include the fixed version of the protocol. The Zab replication protocol, implemented in Apache Zookeeper [66], also includes a dynamic reconfiguration approach for primary-backup clusters that is similar in nature to Raft's joint consensus approach.

The concept of decoupling reconfiguration from the main data replication channel

```
LEMMA $IndImpliesStateMachineSafety \triangleq$
ASSUME $MRRInd$
PROVE $StateMachineSafety$
  $\langle 1 \rangle 0$. $TypeOK$ BY DEF $MRRInd$
  $\langle 1 \rangle 1$. SUFFICES $\forall\, c1, c2 \in committed$ :
      $(c1.entry[1] = c2.entry[1]) \implies (c1 = c2)$
      BY DEF $StateMachineSafety$
  $\langle 1 \rangle 2$. TAKE $c1, c2 \in committed$
  $\langle 1 \rangle 3$. SUFFICES ASSUME $c1.entry[1] = c2.entry[1], c1 \neq c2$
       PROVE FALSE OBVIOUS
  $\langle 1 \rangle 4$. $c1.term \neq c2.term$
      $\langle 2 \rangle 1$. SUFFICES ASSUME $c1.term = c2.term$
          PROVE FALSE OBVIOUS
      $\langle 2 \rangle 2$. $c1.entry[2] = c2.entry[2]$
          BY $\langle 2 \rangle 1$ DEF $MRRInd$,
          $CommittedTermMatchesEntry$
      $\langle 2 \rangle 3$. $c1.entry[1] = c2.entry[1]$ BY $\langle 1 \rangle 3$
      $\langle 2 \rangle 4$. $c1 = c2$
          BY $\langle 1 \rangle 0$, $\langle 2 \rangle 1$, $\langle 2 \rangle 2$, $\langle 2 \rangle 3$, $Z3$ DEF $TypeOK$
      $\langle 2 \rangle$. QED BY $\langle 1 \rangle 3$, $\langle 2 \rangle 4$
  $\langle 1 \rangle 5$. PICK $s1 \in Server : InLog(c1.entry, s1)$
      BY $CommitsAreLogEntries$
  $\langle 1 \rangle 6$. PICK $s2 \in Server : InLog(c2.entry, s2)$
      BY $CommitsAreLogEntries$
  $\langle 1 \rangle 7$. $log[s1][c1.entry[1]] = c1.term$
      BY $\langle 1 \rangle 5$ DEF $MRRInd$, $CommittedTermMatchesEntry$,
      $InLog$, $TypeOK$
  $\langle 1 \rangle 8$. $log[s2][c2.entry[1]] = c2.term$
      BY $\langle 1 \rangle 6$ DEF $MRRInd$, $CommittedTermMatchesEntry$,
      $InLog$, $TypeOK$
  $\langle 1 \rangle 9$. CASE $c1.term > c2.term$
      $\langle 2 \rangle 1$. $\exists\, i \in$ DOMAIN $log[s1] : log[s1][i] = c1.term$
          BY $\langle 1 \rangle 5$ DEF $MRRInd$,
          $CommittedTermMatchesEntry$, $InLog$, $TypeOK$
      $\langle 2 \rangle 2$. $\exists\, i \in$ DOMAIN $log[s1] : log[s1][i] > c2.term$
          BY $\langle 1 \rangle 9$, $\langle 2 \rangle 1$ DEF $TypeOK$
      $\langle 2 \rangle 3$. $Len(log[s1]) \geq c2.entry[1]$
                $\land\ log[s1][c2.entry[1]] = c2.term$
          $\langle 3 \rangle 1$. $c2.term \leq c2.term$ BY DEF $MRRInd$, $TypeOK$
          $\langle 3 \rangle$. QED BY $\langle 1 \rangle 5$, $\langle 2 \rangle 2$, $\langle 3 \rangle 1$ DEF $MRRInd$, $TypeOK$,
             $LogsLaterThanCommittedMustHaveCommitted$
      $\langle 2 \rangle 4$. $log[s1][c1.entry[1]] = c2.term$
          BY $\langle 1 \rangle 3$, $\langle 2 \rangle 3$ DEF $MRRInd$, $TypeOK$,
          $CommittedEntryIndexesAreNonZero$
      $\langle 2 \rangle$. QED BY $\langle 1 \rangle 4$, $\langle 1 \rangle 7$, $\langle 2 \rangle 4$ DEF $TypeOK$
  $\langle 1 \rangle 10$. CASE $c1.term < c2.term$
   ...
```

Figure 3.7: Excerpt of the TLAPS proof of Lemma 7.

has previously appeared in other replication systems, but none that integrate with a Raft-based system. RAMBO [90], an algorithm for implementing a distributed shared memory service, implements a dynamic reconfiguration module that is loosely coupled with the main read-write functionality. Additionally, Matchmaker Paxos [91] is a more recent approach for reconfiguration in Paxos based protocols that adds dedicated nodes for managing reconfigurations, which decouples reconfiguration from the main processing path, preventing performance degradation during configuration changes. There has also been prior work on reconfiguration using weaker models than consensus [92], and approaches to logless implementations of Paxos based replicated state machine protocols [93], which bear conceptual similarities to our logless protocol for managing configuration state. Similarly, [6] presents an approach to asynchronous reconfiguration under a Byzantine fault model that avoids reaching consensus on configurations by utilizing a lattice agreement abstraction.

Previously, there have been a variety of distributed protocols formalized using TLAPS, including Classic Paxos [22], Byzantine Paxos [94], and the Pastry distributed hash table protocol [95]. The Raft protocol, upon initial publication, included a TLA+ formal specification of its static protocol, without dynamic reconfiguration [64]. Later, a formal verification of the safety properties of the static Raft protocol was completed using the Verdi framework for distributed systems verification [21]. The formal verification of static Raft in Verdi consisted of approximately 50,000 lines of Coq [88], took around 18 months to develop, and consisted of 90 total invariants. In comparison, our proof consists of 3189 lines of TLA+ code. Note, however, that it is difficult to directly compare our work with [21] because (1) our TLA+ specifications are written at a higher level of abstraction, and (2) part of the work in [21] was aimed at producing a verified, runnable Raft implementation, which was not our goal. The work of [21] did not include a verification of Raft's dynamic reconfiguration protocols. To our knowledge, our work is the first formally verified safety proof for a reconfiguration protocol that integrates with a Raft based system.

In general, developing formally verified proofs and inductive invariants for real world distributed protocols remains a challenging and non-trivial problem. In recent years, tools like Ivy [27] have attempted to ease the burden of inductive invariant discovery and verification by taking an interactive approach to invariant development, and constraining the specification language for describing these systems so it falls into a decidable fragment of first order logic [96]. These restrictions, however, can place additional burden on the user in cases where a protocol or its invariants do not naturally fall into this decidable fragment [97].

Recent work has built on top of the Ivy system in an attempt to automatically infer inductive invariants for distributed protocols, with varying degrees of success. Tools like IC3PO [31, 98], SWISS [32], and DistAI [33] represent the state of the art in automated inductive invariant discovery for distributed protocols. With some human guidance, they have recently been able to scale to larger protocols like Paxos, but have not yet been applied to protocols like Raft.

Apalache [83] is a symbolic model checker for TLA+ specifications that has been developed in recent years and can check inductiveness of protocol invariants for bounded parameters. It does not, however, currently have any procedures for automatic discovery of inductive invariants. In future it would be interesting to compare the effectiveness of using Apalache versus the probabilistic, TLC-based method for finding counterexamples to induction when debugging a candidate inductive invariant.

3.9 Conclusions

In this chapter we presented *MongoRaftReconfig*, a novel, logless dynamic reconfiguration protocol that improves upon and generalizes the single server reconfiguration protocol of standard Raft by decoupling the main operation and reconfiguration logs. Although *MongoRaftReconfig* was developed for and presented in the context of the MongoDB system, the ideas and underlying protocol generalize to other Raft-based replication protocols that require dynamic reconfiguration. We also presented, to our knowledge, the first formal verification of a reconfiguration protocol for a Raft based replication system. We used TLA+ and TLAPS, the TLA+ proof system, to formalize and mechanically verify our inductive invariant and safety proofs.

Chapter 4

Automatic Inductive Invariant Inference for Distributed Protocols

A predicate that can be proved to be an invariant by proving...an initial predicate and...a step predicate is called an inductive invariant. Model checkers can check whether a state predicate is an invariant of small instances of an abstract program. But the only way to prove it is an invariant is to prove that it either is or is implied by an inductive invariant. For any invariant P, there is an inductive invariant that implies P. However, writing an inductive invariant...is a skill that can be acquired only with practice.

Leslie Lamport, *A Science of Concurrent Programs* [99]

This chapter presents a novel technique for automatically discovering inductive invariants for distributed protocols specified in TLA+, based on the work originally presented in [45]. An artifact containing all of our source code and evaluation results can be found at [100], and a public, open-source version of our tool is also available at [101].

4.1 Introduction

Automatically verifying the safety of distributed systems remains an important and difficult challenge. Distributed protocols such as Paxos [55] and Raft [56] serve as the foundation of modern fault tolerant systems, making the correctness of these protocols critical to the reliability of large scale database, cloud computing, and other decentralized systems [1, 2, 5, 102]. An effective approach for reasoning about the correctness of these protocols involves specifying system *invariants*, which are assertions that must hold in every reachable system state. Thus, a primary task of verification is proving that a candidate invariant holds in every reachable state of a given system. For adequately small, finite state systems, symbolic or explicit state model checking techniques [17–19] can be sufficient to automatically prove invariants. For verification of infinite state or *parameterized* protocols, however, model checking techniques may, in general, be incomplete [20]. Thus, the standard technique for proving that such a system satisfies a given invariant is to discover an *inductive invariant*, which is an invariant that is typically stronger than the desired system invariant, and is preserved by all protocol transitions. Discovering inductive invariants, however, as discussed in Chapter 3, is one of the most challenging aspects of verification and remains a non-trivial task with a large amount of human effort required [21–24]. Thus, automating the inference of these invariants is a desirable goal.

In general, the problem of inferring inductive invariants for infinite state protocols is undecidable [25]. Even the verification of inductive invariants may require checking the validity of arbitrary first order formulas, which is undecidable [26]. Thus, this places fundamental limits on the development of fully general algorithmic techniques for discovering inductive invariants.

Significant progress towards automation of inductive invariant discovery for infinite state protocols has been made with the Ivy framework [27]. Ivy utilizes a restricted system modeling language that allows for efficient checking of verification goals via an SMT solver such as Z3 [28]. In particular, the EPR and extended EPR subsets of Ivy are decidable. Ivy also provides an interface for an interactive, counterexample guided invariant discovery process. The Ivy language, however, may place an additional burden on users when protocols or their invariants don't fall naturally into one of the decidable fragments of Ivy. Transforming a protocol into such a fragment is a manual and nontrivial task [26].

Subsequent work has attempted to fully automate the discovery of inductive invariants for distributed protocols. State of the art tools for inductive invariant inference for distributed protocols include I4 [29], fol-ic3 [30], IC3PO [31], SWISS [32], and DistAI [33]. All of these tools, however, accept only Ivy or an Ivy-like language [34] as input. Moreover, several of these tools work only within the restricted decidable fragments of Ivy.

In this chapter, we present a new technique for automatic discovery of inductive invariants for protocols specified in TLA$^+$, a high level, expressive specification language [35]. To our knowledge, this is the first inductive invariant discovery tool for distributed protocols in a language other than Ivy. Our technique is built around a core procedure for generating small, *plain* (potentially non-inductive) invariants. We search for these invariants on finite protocol instances, employing the so-called *small scope* hypothesis [29, 36, 37], circumventing undecidability concerns when reasoning over unbounded domains. We couple this invariant generation procedure with an invariant selection procedure based on a greedy counterexample elimination heuristic in order to incrementally construct an overall inductive invariant. By restricting our inference reasoning to finite instances, we avoid restrictions imposed by modeling approaches that try to maintain decidability of SMT queries.

Our technique is partially inspired by prior observations [22, 24, 32, 103] that, for many practical protocols, an inductive invariant I is typically of the form $I = P \land A_1 \land \ldots \land A_n$, where P is the main invariant (i.e. safety property) we are trying to establish, and $A_1, \ldots, A_n$ are a list of *lemma invariants*. Each lemma invariant A_i may not necessarily be inductive, but it is necessarily an invariant, and it is typically much smaller than I. These lemma invariants serve to strengthen P so as to make it inductive. Many prior approaches to inductive invariant inference have focused on searching for lemma invariants that are inductive, or inductive relative to previously discovered information [30–32, 103]. In contrast, our inference procedure searches for *plain* lemma invariants and uses them as candidates for conjuncts of an overall inductive invariant. To search for lemma invariants, we sample candidates using a syntax-guided approach [104], and verify the candidates using an off the shelf model checker.

We have implemented our invariant inference procedure in a tool, *endive*, and we evaluate its performance on a set of diverse protocol benchmarks, including 29 of the benchmarks reported in [31]. Our tool solves nearly all of these benchmarks, and compares favorably with other state of the art tools, despite the fact that all of these tools accept Ivy or decidable Ivy fragments as inputs. We also evaluate our tool and other state of the art tools on a more complex, industrial scale protocol, *MongoLoglessDynamicRaft (MLDR)* [24], which performs dynamic reconfiguration in a Raft based replication system. Our tool is the only one which manages to find a correct inductive invariant for MLDR.

To summarize, in this chapter we make the following contributions:

- A new technique for inductive invariant inference that works for distributed protocols specified in TLA$^+$.
- A tool, *endive*, which implements our inductive invariant inference algorithm. To our knowledge, this is the only existing tool that works directly on TLA$^+$.
- An experimental evaluation of our tool on a diverse set of distributed protocol benchmarks.
- The first, to our knowledge, automatic inference of an inductive invariant for an industrial scale Raft-based reconfiguration protocol.

The rest of this chapter is organized as follows. Section 4.2 describes our algorithm for inductive invariant inference, along with more details on our technique. Section 4.3 provides an experimental evaluation of our algorithm, as implemented in our tool, *endive*. Section 4.4 examines related work, and Section 4.5 presents conclusions and goals for future work.

4.2 Our Approach

At a high level, our inductive invariant inference method consists of the following steps:

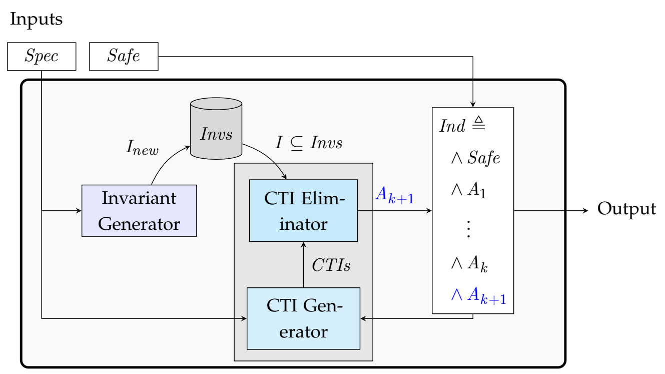


Figure 4.1: Components of our technique for inductive invariant inference.

1. Generate many candidate lemma invariants, and store them in a repository that we call $Invs$.
2. Generate counterexamples to induction for a current candidate inductive invariant, Ind. If we cannot find any such CTIs, return Ind.
3. Select lemma invariants from $Invs$ so that all CTIs are eliminated. If we cannot eliminate all CTIs, either give up, or go to Step 1 and populate the repository with more lemma invariants. Otherwise, add the selected lemma invariants to Ind and repeat from Step 2.

The conceptual approach is illustrated in Figure 4.1. Our detailed algorithm is described in Section 4.2.1. Section 4.2.2 provides details on our lemma invariant generation procedure, Section 4.2.3 provides details on CTI generation, and Section 4.2.4 describes the selection of lemma invariants.

4.2.1 Inductive Invariant Inference Algorithm

Our inductive invariant inference algorithm is given in pseudocode in Algorithm 3. The algorithm takes as input: (1) a finite instance of a symbolic transition system M, (2) a candidate invariant (safety property) $Safe$, (3) a lemma invariant repository $Invs$, and (4) a grammar G for generating lemma invariant candidates. The use of the grammar is discussed further in Section 4.2.2. $Invs$ may initially be empty, or be pre-populated from previous runs of the algorithm. The algorithm aims to discover an inductive invariant, Ind, of the form $Ind = Safe \wedge A_1 \wedge \ldots \wedge A_n$.

The algorithm maintains a current inductive invariant candidate, Ind, which it initializes to $Safe$, the safety property that we are trying to prove (Line 3). It then generates a set X of CTIs of Ind (Line 6). The algorithm may also initialize the repository of lemma invariants, $Invs$, or add more lemma invariants to $Invs$ if it is initially non-empty

Algorithm 3 Our inductive invariant inference algorithm.

```
1: Inputs:
   $M$: Finite instance of a parameterized STS
   $Safe$: Candidate invariant
   $Invs$: Lemma invariant repository (typically empty initially)
   $G$: Grammar for invariant generation
2: action: InferInductiveInvariant($M$, $Safe$, $G$, $Invs$)
3:     $Ind \leftarrow Safe$
4:     $X \leftarrow GenerateCTIs(M, Ind)$
5:     $Invs \leftarrow GenerateLemmaInvariants(M, Invs, G)$
6:     while $X \neq \emptyset$ do
7:         if $\exists A \in Invs : A$ eliminates at least one CTI in $X$ then
8:             pick $A_{max} \in Invs$ that eliminates the most CTIs from $X$
9:             $Ind \leftarrow Ind \wedge A_{max}$
10:            $X \leftarrow X \setminus \{s \in X : s \not\models A_{max}\}$
11:        else
12:            either goto Line 5
13:            or return ($Ind$, "Fail: couldn't eliminate all CTIs.")
14:        end if
15:        $X \leftarrow GenerateCTIs(M, Ind)$
16:    end while
17:    return ($Ind$, "Success: managed to eliminate all CTIs.")
18: end action:
```

(Line 5). The procedures *GenerateLemmaInvariants* and *GenerateCTIs* are described in more detail below, in Sections 4.2.2 and 4.2.3, respectively.

In its main loop, the algorithm tries to eliminate all currently known CTIs. As long as the set X of currently known CTIs is non-empty, the algorithm tries to find a lemma invariant in the *Invs* repository that eliminates the maximal number of remaining CTIs possible. If such a lemma invariant exists, the algorithm adds it as a new conjunct to *Ind* (Line 9), removes from X the CTIs that were eliminated by the new conjunct (Line 10), and proceeds by attempting to generate more CTIs, since the updated *Ind* is not necessarily inductive (Line 15).

If no lemma invariant exists in the current repository *Invs* that can eliminate any of the currently known CTIs (Line 11), then we may either (1) generate more lemma invariants in the repository, or (2) give up. The first choice is implemented by the **goto** statement in Line 12. The second choice represents a failure of the algorithm to find an inductive invariant (Line 13). However, in this case we still return *Ind* since, even though it is not inductive, it may contain several useful lemma invariants. These lemma invariants are useful in the sense that they might be part of an ultimate inductive invariant.

If all known CTIs have been eliminated, the algorithm terminates successfully and returns *Ind* (Line 17). Successful termination of the algorithm indicates that the returned

Ind is likely to be inductive. However our method does not provide a formal inductiveness guarantee. Ind might not be inductive for a number of reasons. First, as we discuss further in Section 4.2.3, our CTI generation procedure is probabilistic in nature, and therefore $GenerateCTIs$ might miss some CTIs. Second, even if the finite-state instance M explored by the algorithm has no remaining CTIs, there might still exist CTIs in other instances of the STS , for larger parameter values.

Even though a candidate invariant returned by a successful termination of Algorithm 3 is not formally guaranteed to be inductive, we ensure soundness of our overall procedure by doing a final check that the discovered candidate inductive invariant is correct using the TLA$^+$ proof system (TLAPS) [70]. Validation of invariants in TLAPS is discussed further in Section 4.2.5. In practice we found that all of the invariants generated in our evaluation (Section 5.5) are correct inductive invariants.

We also remark that in the current version of our algorithm and in the current implementation of our tool, we only explore the single finite-state instance of the STS provided by the user, and we do not attempt to automatically increase the bounds of the parameters within the algorithm, as is done for example in the approach described in [31]. This is, however, a relatively straightforward extension to our algorithm, and would like to explore this option in future work.

4.2.2 Lemma Invariant Generation

For a given finite instance M of a parameterized transition system, the goal of lemma invariant generation is to produce a set of state predicates that are invariants of M. To search for these invariants, we adopt an approach similar to other, *syntax-guided synthesis* based techniques [104, 105] for invariant discovery. We randomly sample invariant candidates from a defined *grammar*, which is generated from a given set of *seed* predicates. Each seed predicate is an atomic boolean predicate over the state variables of the system. Note that the parameterized distributed protocols that we consider in this chapter typically have inductive invariants that are universally or existentially quantified over the parameters of the protocol or other values of the system state. So, our invariant generation technique assumes a fixed quantifier template that is provided as input. The provided seed predicates are unquantified predicates that can contain bound variables that appear in the given quantifier template. An example of a simple grammar for the protocol of Figure 2.1 is shown in Figure 4.2.

Candidate invariants are produced by generating random predicates over the space of seed predicates. Specifically, a candidate predicate is formed as a random disjunction of seed predicates, where each disjunct may be negated with probability $\frac{1}{2}$. The logical connectives $\{\vee, \neg\}$ are functionally complete [106], so they serve as a simple basis for generating candidate invariants, which we chose to reduce the invariant search space.

For a given set of candidate invariants, C, we check which of the predicates in C are

$$\begin{aligned}
\langle seed \rangle &::= \ locked[s] \mid s \in held[c] \mid held[c] = \emptyset \\
\langle quant \rangle &::= \forall s \in Server : \forall c \in Client \\
\langle expr \rangle &::= \langle seed \rangle \mid \neg \langle expr \rangle \mid \langle expr \rangle \vee \langle expr \rangle \\
\langle pred \rangle &::= \langle quant \rangle : \langle expr \rangle
\end{aligned}$$

Figure 4.2: Example of a grammar for lemma invariant generation for the *lockserver* protocol shown in Figure 2.1. The list of unquantified *seed* predicates and the quantifier template, *quant*, are provided as user inputs.

invariants using an explicit state model checker. This can be done effectively due to our use of the small scope hypothesis i.e. the fact that we reason only about a finite instance M of a parameterized transition system. This largely reduces the invariant checking problem to a data processing task. Namely:

(1) Generate $Reach(M)$, the set of reachable states of M.

(2) Check that $s \models P$ for each predicate $P \in C$ and each $s \in Reach(M)$.

Note that after (1) has been completed once, the set of reachable states can be cached and only step (2) must be re-executed when searching for additional invariants.

In theory, the worst case cost of step (2) is proportional to $|C| \cdot |Reach(M)|$. In practice, however, it can often be much less costly than this, since once a state violates a predicate P, P need not be checked further. Furthermore, both of the above computation steps are highly parallelizable, a fact we make use of in our implementation, as discussed further in Section 4.3.1.

We also remark that, in practice, the *GenerateLemmaInvariants* procedure is configured to search for candidate invariants of a fixed term size i.e. with a fixed or maximal number of disjuncts. In our implementation, presented in Section 4.3.1, we utilize this to search for smaller invariants (fewer terms) first, before searching for larger ones. That is, we prefer to eliminate CTIs if possible with smaller invariants before searching for larger ones. This aims to bias our procedure towards discovery of compact inductive invariant lemmas.

Furthermore, since *GenerateLemmaInvariants* does not employ an exhaustive search for invariants over a given space of predicates, it accepts a numeric parameter, N_{lemmas}, which determines how many candidate predicates to sample. More details of how the concrete values of this parameter are configured are discussed in our evaluation, in Section 5.5.

4.2.3 CTI Generation

Each round of our algorithm relies on access to a set of multiple CTIs, as a means to prioritize between different choices of new lemma invariants. To generate these CTIs, we use a probabilistic technique proposed in [107] that utilizes the TLC explicit state model checker [82]. Given a finite instance of a STS M with system states S, transition relation predicate $Next$, and given candidate inductive invariant Ind, the procedure $GenerateCTIs(M, Ind)$ works by calling the TLC model checker. TLC attempts to randomly sample states $s_0 \in S$ for which there exists a sequence of states $s_1, s_2, \ldots, s_{k-1}, s_k \in S$, such that both of the following hold:

- $\forall i = 0, 1, \ldots, k-1 : (s_i, s_{i+1}) \models Next \wedge s_i \models Ind$
- $s_k \not\models Ind$.

The model checker will report this behavior, and all states $s_0, s_1, s_2, \ldots, s_{k-1}$ are recorded as counterexamples to induction.

Due to the randomized nature of this technique, the CTI generation procedure requires a given parameter, N_{ctis}, that effectively determines how many possible states TLC will attempt to sample before terminating the CTI generation procedure. This is required, since, for systems with sufficiently large state spaces, even if finite, sampling all possible states is infeasible. Generally, this parameter can be tuned based on the amount of compute power available to the tool, or a latency tolerance of the user. We discuss more details of this parameter and how it is tuned in our experiments in Section 5.5.

In practice, during our evaluation we found that TLC was able to effectively generate many thousands of CTIs at each round of the inference algorithm using the above technique. This provided an adequately diverse distribution of CTIs for effectively guiding our counterexample elimination procedure, which we describe in more detail in Section 4.2.4. Section 5.5 presents more detailed metrics on CTI generation as measured when testing our implementation on a variety of protocol benchmarks. In future we feel it would be valuable to explore and compare with other, SMT/SAT based techniques for this type of counterexample generation task [108, 109].

4.2.4 Lemma Invariant Selection by CTI Elimination

The task of selecting lemma invariants for use as inductive invariant conjuncts is based on a process of CTI elimination, as described briefly in Section 4.2.1. That is, CTIs are used as guidance for which invariants to choose for new lemma invariants to append to the current inductive invariant candidate. Once a sufficiently large set of CTIs has been generated, as discussed in Section 4.2.3, we select lemma invariants using a greedy heuristic of CTI elimination, which we describe below.

CTI Elimination

Recall that a CTI s is *eliminated* by a state predicate A if $s \not\models A$. When examining a current set of CTIs, X, our algorithm looks for the next lemma invariant $A \in \mathit{Invs}$ that eliminates the most CTIs in X. The algorithm will continue choosing additional lemma invariants according to this strategy until all counterexamples are eliminated, or until it cannot eliminate any further counterexamples. Each selected invariant $A_i \in \mathit{Invs}$ will be appended as a new conjunct to the current inductive invariant candidate i.e. $\mathit{Ind} \leftarrow \mathit{Ind} \land A_i$. Once all counterexamples have been eliminated, the tool will terminate and return a final candidate inductive invariant. This is a simple heuristic for choosing new invariant conjuncts that aims to bias the overall inductive invariant towards being relatively concise. That is, if we have a choice between two alternate lemma conjuncts to choose from, we prefer the conjunct that eliminates more CTIs.

More generally, lemma selection at each round of the algorithm can be viewed as a version of the set covering problem [110]. Ideally, we would like to find the smallest set of lemma invariants that eliminate (i.e. cover) the set of CTIs X. Solving this problem optimally is known to be NP-complete [111], but we have found a greedy heuristic [112] to work sufficiently well in our experiments, the results of which are presented in Section 5.5. In future we would like to explore more sophisticated heuristics for lemma selection that take into account additional metrics, like syntactic invariant size, quantifier depth, etc.

4.2.5 Validation of Inductive Invariant Candidates

If our inference algorithm terminates successfully, it will return a candidate inductive invariant. Since we look for invariants on finite protocol instances, though, this candidate may not be an inductive invariant for general (e.g. unbounded) protocol instances. So, upon termination, we check to see if the returned candidate invariant is truly inductive for all protocol instances by passing it to an SMT solver. Currently, we use the TLA$^+$ proof system (TLAPS) [70] for this step, which generates an SMT encoding for TLA$^+$ [113].

For many of the protocols we tested and the invariants discovered by our tool, we found that this step was fully automated (see Section 5.5 and Table 4.3 in the Appendix). That is, no user assistance was required to establish validity of the discovered invariant. In cases where the underlying solver cannot automatically prove the candidate inductive invariant, some amount of human guidance can be provided by decomposing the proof into smaller SMT queries. We have completed this validation step for all of the inductive invariant candidates discovered in our experiments, and we confirmed that all candidate invariants produced by our tool were indeed correct inductive invariants (see Section 5.5).

4.3 Evaluation

4.3.1 Implementation and Experimental Setup

Our invariant inference algorithm is implemented in a tool, *endive*, whose main implementation consists of approximately 2200 lines of Python code. There are also some optimized subroutines which consist of an additional few hundred lines of C++ code. Internally, *endive* makes use of version 2.15 of the TLC model checker [82], with some minor modifications to improve the efficiency of checking many invariants simultaneously. TLC is used by *endive* for most of the algorithm's compute intensive verification tasks, like checking candidate lemma invariants (Section 4.2.2) and CTI elimination checking (Section 4.2.4).

For all of the experiments discussed below, *endive* is configured to use 24 parallel TLC worker threads for invariant checking, 4 parallel threads for CTI generation, and 4 threads for CTI elimination. CTI generation and CTI elimination can be parallelized further in a straightforward manner, but we limit these procedures to 4 parallel threads to simplify certain aspects of our current implementation.

For each benchmark run, we initialize $Invs$ (as explained in Algorithm 3) as an empty set and configure the lemma invariant generation procedure discussed in Section 4.2.2 with a parameter value of $N_{lemmas} = 15000$. The grammars used for invariant generation were mined from predicates appearing in each protocol specification.

We configure our CTI generation procedure with a parameter value of $N_{ctis} = 50000$. N_{ctis} does not directly correspond to how many concrete CTI states will be generated, but a higher value indicates TLC will sample more states when searching for CTIs. We also limit the maximum number of CTIs returned by each call to the $GenerateCTIs$ procedure to 10000 states. In theory, generating more CTIs provides better counterexample diversity, and is therefore better for our CTI elimination heuristics. We impose an upper limit, however, to avoid scalability issues in our tool's current implementation. In practice we found this limit sufficient to provide effective guidance for lemma invariant selection.

All of our experiments were run on a 48-core Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz machine with 196GB of RAM.

4.3.2 Benchmarks

To evaluate *endive*, we measured its performance on 29 protocols selected from an existing benchmark set published in [31]. We also evaluate endive on an additional, industrial scale protocol, *MongoLoglessDynamicRaft (MLDR)*, which is a recent protocol for distributed dynamic reconfiguration in a Raft based replication system [24, 50].

Protocol Conversion

The 29 benchmarks we used from [31] were originally specified in Ivy [27], but *endive* accepts protocols in TLA$^+$, so it was necessary to manually translate the protocols from Ivy to TLA$^+$. There are significant differences in how protocols are specified in Ivy and TLA$^+$. The underlying approach to modeling systems as discrete transition systems, however, by specifying initial states and a transition relation, are common between them. In our manual translation, we aimed to emulate the original Ivy model as close as possible.

The formal specification for the *MongoLoglessDynamicRaft* protocol (*MLDR*) was originally written in TLA$^+$ [50]. Thus, in order to compare with other invariant inference tools which accept Ivy as their input language, we had to translate MLDR from TLA$^+$ into Ivy. This conversion process was highly nontrivial due to the significant differences between the Ivy and TLA$^+$ languages. TLA$^+$ is a very expressive language that includes integers, strings, sets, functions, records, and sequences as primitive data types along with their standard semantics. In contrast, the Ivy modeling language, RML [27], includes only basic, first order relations and functions. For more complex datatypes (e.g. arrays or sequences), their semantics must be defined and axiomatized manually.

An artifact containing all of our source code and instructions for reproducing our evaluation results can be found at [100]. A public, open-source version of our tool is also available at [101].

4.3.3 Results

Our overall results are shown in Table 4.1. We compared *endive* with four recent, state of the art techniques for inferring invariants of distributed protocols: IC3PO [31], fol-ic3 [30], SWISS [32], and DistAI [33]. Note that *endive* accepts protocols in TLA$^+$, whereas all other tools accept protocols in Ivy or mypyvy.

The numbers shown for both IC3PO and fol-ic3 in Table 4.1 are as reported in the evaluation presented in [31], with timeouts indicated by a *TO* entry. For the SWISS results in Table 4.1, where possible, we show the runtime numbers reported in [32], indicated with a † mark. For the benchmarks in Table 4.1 that were not tested in [32], we present the results from our own runs of the tool, all using default SWISS configuration parameters. We ran SWISS both with an invariant template matching our own template for *endive* and also in automatic mode, and report the better of the two results. The results for DistAI are reported from our runs using the tool in its default configuration. For DistAI and SWISS, we report an *err* result in cases where the tool returned an error without producing a result. We report a *fail* result in cases where DistAI or SWISS terminated without error but did not discover an inductive invariant. In all cases where a benchmark protocol was not available in the required input language for the corresponding tool, we mark this with an *n/a* entry.

		endive		IC3PO		fol-ic3		SWISS		DistAI	
No.	Protocol	Time	Inv	Time	Inv	Time	Inv	Time	Inv	Time	Inv
1	tla-consensus	1	1	0	1	1	1	1	2	2	1
2	tla-tcommit	2	1	1	2	2	3	2	8	2	7
3	i4-lock-server	7	2	1	2	1	2	†1	2	err	
4	ex-quorum-leader-election	11	2	3	5	24	8	11	5	3	8
5	pyv-toy-consensus-forall	19	3	3	5	11	5	†3	7	err	
6	tla-simple	8	2	6	3	TO		28	8	err	
7	ex-lockserv-automaton	23	9	7	12	10	12	fail		2	13
8	tla-simpleregular	10	4	8	4	57	9	65	21	err	
9	pyv-sharded-kv	312	6	10	8	22	10	†4024		2	16
10	pyv-lockserv	35	9	11	12	8	11	†3684		2	13
11	tla-twophase	43	10	14	9	9	12	33	24	29	306
12	i4-learning-switch	TO		14	10	TO		TO		21	32
13	ex-simple-decentralized-lock	44	4	19	15	4	8	1	2	26	17
14	i4-two-phase-commit	69	11	27	11	8	9	†6	15	17	67
15	pyv-consensus-wo-decide	127	8	50	9	168	26	†18	8	err	
16	pyv-consensus-forall	175	8	99	10	2461	27	†29	9	err	
17	pyv-learning-switch	TO		127	13	TO		†959		79	70
18	i4-chord-ring-maintenance	n/a		229	12	TO		†TO		53	164
19	pyv-sharded-kv-no-lost-keys	13	2	3	2	3	2	†1	4	fail	
20	ex-naive-consensus	40	4	6	4	73	18	18	5	fail	
21	pyv-client-server-ae	46	2	2	2	877	15	†3	5	err	
22	ex-simple-election	24	4	7	4	32	10	9	5	err	
23	pyv-toy-consensus-epr	19	4	9	4	70	14	†2	4	err	
24	ex-toy-consensus	7	2	10	3	21	8	6	4	err	
25	pyv-client-server-db-ae	4941	8	17	6	TO		†24	13	err	
26	pyv-hybrid-reliable-broadcast	n/a		587	4	1360	23	†TO		err	
27	pyv-firewall	38	5	2	3	7	8	75	5	err	
28	ex-majorityset-leader-election	53	4	72	7	TO		28	10	err	
29	pyv-consensus-epr	247	8	1300	9	1468	30	72	10	err	
30	mldr	2025	6	TO		n/a		err		err	

Table 4.1: Distributed protocol benchmark results.

For each benchmark result in Table 4.1, we report the total wall clock time to discover an inductive invariant in the *Time* column, along with the number of total lemma invariants contained in the discovered invariant, including the safety property, in the *Inv* column. Note that the number of total lemmas in the invariants discovered by SWISS was not reported in [32]. Thus, we report the number of lemmas discovered by SWISS in our own runs, for the cases where we were able to run SWISS successfully to produce an invariant.

More detailed statistics on the *endive* benchmark results are provided in Appendix 4.3.6, specifically: the number of eliminated CTIs, runtime profiling information, finite instance sizes used, and automation level of the TLAPS proofs.

4.3.4 Comparison with Other Tools

Although Table 4.1 relates our approach to several others, we note that our tool is not directly comparable to other tools. The most fundamental difference is that our tool accepts TLA$^+$ whereas all other tools in Table 4.1 accept Ivy or mypyvy. Furthermore, some tools work only with the restricted decidable EPR or extended EPR fragments of Ivy. To our knowledge, this is the case with SWISS and DistAI. As a result, our tool is a-priori less automated than other tools, following a standard tradeoff between expressivity and automation. In practice, however, and despite this theoretical limitation, our tool produces a result in most cases, while some of the a-priori more automated tools time out or fail.

Another important difference between the tools of Table 4.1 is what kind of inductive invariants can be produced by each tool. In our case, the user provides the grammar of possible lemma invariants as an input to the tool, allowing both universal and existentially quantified invariants ($\forall$ and $\exists$). DistAI is limited to only universally quantified ($\forall$) invariants, and SWISS is limited to invariants that fall into the extended EPR fragment, though it can learn both universal and existentially quantified invariants. Both fol-ic3 and IC3PO attempt to learn the quantifier structure itself during counterexample generalization, and can infer both universal and existentially quantified invariants. These tools do not always guarantee, however, that the discovered invariants will fall into a decidable logic fragment. Thus, they provide no explicit guarantee that the overall inference procedure will, in general, be fully automated.

4.3.5 Discussion

Our tool, *endive*, was able to successfully discover an inductive invariant for 25 of the 29 protocol benchmarks from [31], and all of the invariants it discovered were proven correct using TLAPS. For the two protocols out of these 29 that our tool did not solve, pyv-learning-switch and i4-learning-switch, this was due to scalability limitations of CTI generation, which we believe could be improved with a smarter CTI generation algorithm or by incorporating a symbolic model checker [109] for this task.

endive was also able to automatically discover an inductive invariant for a key safety property of *MLDR*, a Raft-based distributed dynamic reconfiguration protocol [50]. This protocol, reported in Table 4.1 as *mldr*, is a significantly more complex, industrial scale protocol [24]. IC3PO was not able to discover an invariant for our Ivy model of the MLDR protocol after a 1 hour timeout when given the same instance size used in the TLA$^+$ model given to *endive*. SWISS and DistAI both produced an error when run on our Ivy model of MLDR.

Generally, the wall clock time taken for *endive* to discover an inductive invariant is of a similar order of magnitude to IC3PO. *endive* even outperforms IC3PO in some cases, despite the fact that *endive* works with TLA$^+$ and IC3PO works with Ivy. Moreover, in several cases where *endive*'s runtime exceeds that of IC3PO, *endive* is able to discover a smaller inductive invariant (e.g. pyv-lockserv, ex-simple-decentralized-lock, pyv-consensus-forall). Additionally, *endive* is often able to discover a considerably smaller invariant than tools like DistAI and SWISS. For example, on tla-twophase, *endive* learns an invariant with 10 overall conjuncts, whereas SWISS learns a 24 conjunct invariant, and DistAI learns a much larger invariant, with over 300 conjuncts. *endive* performs similarly well for the tla-simpleregular and i4-two-phase-commit benchmarks. This demonstrates that *endive* compares favorably against other enumerative approaches for inductive invariant inference, both in terms of efficiency and compactness of invariants, while also working over TLA$^+$, a much more expressive input language.

It is additionally worth noting that our current *endive* implementation is not highly optimized. In particular, the TLC model checker, used internally by endive, is implemented in Java and interprets TLA$^+$ specifications dynamically [114], rather than compiling models to a low level, native representation as done by tools like SPIN [18]. As a result, TLC may not be the most efficient for our inference procedure, and could likely be optimized further.

4.3.6 Detailed Runtime and Proof Metrics

Table 4.2 gives a more detailed breakdown of the results presented in Table 4.1 for our *endive* invariant inference tool. The *Check*, *Elim*, and *CTIGen* columns of Table 4.2 indicate, respectively, the wall clock time in seconds for (1) checking candidate lemma invariants, (2) eliminating CTIs, and (3) generating CTIs. The *CTIs* column indicates the total number of eliminated CTIs.

Recall that we limit the maximum number of generated CTIs to 10000 per round, as mentioned in Section 4.3.1. This explains why some protocol results for the *endive* tool report elimination of exactly 10000 CTIs. For example, for the tla-twophase benchmark, an inductive invariant was discovered in a single round of the algorithm loop (starting at Line 7 of Algorithm 3), so no more than 10000 CTIs were generated in the entire run. If the benchmark run eliminated greater than 10000 CTIs, this indicates that it ran for more more than 1 round.

Also, for protocols that eliminated 0 CTIs (e.g. tla-consensus, tla-tcommit), this indicates that the starting safety property was already inductive. Thus, no CTIs were ever generated and no lemma invariants were needed. Similarly, some protocols eliminated a nonzero amount of CTIs less than 10000 (e.g. ex-quorum-leader-election). This may be the case when no more than a single round of the algorithm was needed to discover an inductive invariant, or that the number of generated counterexamples at each round did not exceed 10000. Recall that, even within a single round of the algorithm, as shown in Algorithm 3, it is possible to discover multiple new lemma invariants.

Additional statistics on the instance sizes used during invariant inference and the degree of automation required for TLAPS proofs are shown in Table 4.3. The *TLAPS Auto* column indicates whether the TLAPS proof of the inductive invariant discovered by *endive* was completely automatic (indicated with a ✓), or required some user assistance (indicated with a ✗).

To provide more fine-grained detail on the level of automation for each TLAPS proof, the *TLAPS Auto* column also includes the number of verification conditions in the induction check that were proved fully automatically. For a protocol with a transition relation of the form $Next = T_1 \vee \cdots \vee T_k$ and an inductive invariant candidate $Ind = A_1 \wedge \ldots \wedge A_n$, the consecution check $Ind \wedge Next \Rightarrow Ind'$ is typically the most significant verification burden, and can be trivially decomposed into $k \cdot n$ verification conditions (VCs). That is, a verification condition $Ind \wedge T_j \Rightarrow A_i'$ is generated for each $j \in \{1, \ldots, k\}$ and $i \in \{1, \ldots, n\}$, giving $k \cdot n$ total VCs. We notate these statistics in the *TLAPS Auto* column as (# VCs proved automatically / $k \cdot n$ total VCs). Protocols that were proved fully automatically are shown as $(k \cdot n / k \cdot n)$. The *Check (s)* column also shows the total time in seconds needed to check each proof, as measured on a 2020 M1 Macbook using version 1.4.5 of the TLA+ proof manager.

4.4 Related Work

There are several recently published techniques that attempt to solve the problem of inductive invariant inference for distributed protocols. The IC3PO tool [31], which extended the earlier I4 tool [29], uses a technique based on IC3 [103] with a novel *symmetry boosting* technique that serves to accelerate IC3/PDR and also to infer the quantifier structure of lemma invariants. The fol-ic3 algorithm presented in [30] presents another IC3 based algorithm which uses a novel *separators* technique for discovering quantified formulas to separate positive and negative examples during invariant inference. SWISS [32] is another recent approach that uses an enumerative search for quantified invariants while using the Ivy tool to validate possible inductive candidates. It relies on SMT based reasoning over an unbounded domain, and does not reason directly about finite instances of distributed protocols. DistAI [33] uses a similar approach but additionally utilizes a technique of sampling reachable protocol states to filter invariants, which is similar to our approach of executing explicit state model checking as a means to quickly discover invariants. DistAI is limited, however, to learning only universally quantified invariants.

In addition to these inductive invariant inference techniques, there also exists prior work on alternative techniques for parameterized protocol verification. These include approaches based on cutoff detection [115], regular model checking [116], and symbolic backward reachability analysis [117].

More broadly, there exist many prior techniques for the automatic generation of program and protocol invariants that rely on data driven or grammar based approaches.

Table 4.2: Detailed profiling results for the *endive* results from Table 4.1.

No.	Protocol	Time	CTIs	Check	Elim	CTIGen
1	tla-consensus	1	0	0	0	1
2	tla-tcommit	2	0	0	0	2
3	i4-lock-server	7	12	2	2	4
4	ex-quorum-leader-election	11	204	2	2	7
5	pyv-toy-consensus-forall	19	412	2	2	15
6	tla-simple	8	15	2	2	5
7	ex-lockserv-automaton	23	3624	6	8	9
8	tla-simpleregular	10	1972	3	3	5
9	pyv-sharded-kv	312	11715	17	46	249
10	pyv-lockserv	35	3654	11	11	13
11	tla-twophase	43	10000	10	22	12
12	i4-learning-switch	TO				
13	ex-simple-decentralized-lock	44	2035	13	18	14
14	i4-two-phase-commit	69	10408	18	19	33
15	pyv-consensus-wo-decide	127	12995	56	39	32
16	pyv-consensus-forall	175	10609	63	25	88
17	pyv-learning-switch	TO				
18	i4-chord-ring-maintenance	n/a				
19	pyv-sharded-kv-no-lost-keys	13	404	2	2	9
20	ex-naive-consensus	40	10000	10	15	16
21	pyv-client-server-ae	46	10000	2	4	40
22	ex-simple-election	24	551	10	7	8
23	pyv-toy-consensus-epr	19	384	8	6	6
24	ex-toy-consensus	7	14	2	2	4
25	pyv-client-server-db-ae	4941	12546	4657	46	239
26	pyv-hybrid-reliable-broadcast	n/a				
27	pyv-firewall	38	1740	11	22	7
28	ex-majorityset-leader-election	53	10000	12	15	26
29	pyv-consensus-epr	247	16269	80	38	129
30	mldr	2025	7751	1272	651	102

Houdini [118] and Daikon [119] both use enumerative checking approaches to discover program invariants. FreqHorn [104] tries to discover quantified program invariants about arrays using an enumerative approach that discovers invariants in stages and also makes use of the program syntax. Other techniques have also tried to make invariant discovery more efficient by using improved search strategies based on MCMC sampling [120].

4.5 Conclusions and Future Work

We presented a new technique for inferring inductive invariants for distributed protocols specified in TLA$^+$ and evaluated it on a diverse set of protocol benchmarks. Our approach is novel in that: (1) it is the first, to our knowledge, to infer inductive invariants directly for protocols specified in TLA$^+$ and (2) it is based around a core procedure for generating plain, not necessarily inductive, lemma invariants. Our results show that our approach performs strongly on a diverse set of distributed protocol benchmarks. In addition, it is

Table 4.3: Additional statistics for *endive* results reported in Table 4.1.

No.	Protocol	Instance Size	TLAPS Auto	Check (s)
1	tla-consensus	Value={v1,v2,v3}	✓ (1/1)	13
2	tla-tcommit	RM={rm1,rm2,rm3}	✓ (2/2)	1
3	i4-lock-server	Server={s1,s2} Client={c1,c2}	✓ (4/4)	1
4	ex-quorum-leader-election	Node={n1,n2,n3,n4}	✓ (4/4)	1
5	pyv-toy-consensus-forall	Node={n1,n2,n3} Value={v1,v2}	✓ (6/6)	1
6	tla-simple	N=4	✓ (4/4)	1
7	ex-lockserv-automaton	Node={n1,n2,n3}	✓ (45/45)	6
8	tla-simpleregular	N=3	✓ (12/12)	1
9	pyv-sharded-kv	Node={n1,n2,n3} Key={k1,k2} Value={v1,v2}	✓ (18/18)	15
10	pyv-lockserv	Node={n1,n2,n3}	✓ (45/45)	6
11	tla-twophase	RM={rm1,rm2,rm3}	✗ (68/70)	18
12	i4-learning-switch	TO		
13	ex-simple-decentralized-lock	Node={n1,n2,n3}	✓ (8/8)	17
14	i4-two-phase-commit	Node={n1,n2,n3}	✓ (77/77)	6
15	pyv-consensus-wo-decide	Node={n1,n2,n3}	✗ (35/40)	20
16	pyv-consensus-forall	Node={n1,n2,n3}	✗ (46/48)	25
17	pyv-learning-switch	TO		
18	i4-chord-ring-maintenance	n/a		
19	pyv-sharded-kv-no-lost-keys	Node={n1,n2} Key={k1,k2} Value={v1,v2}	✓ (6/6)	12
20	ex-naive-consensus	Node={n1,n2,n3} Value={v1,v2}	✗ (11/12)	6
21	pyv-client-server-ae	Node={n1,n2,n3} Request={r1,r2} Response={p1,p2}	✓ (6/6)	2
22	ex-simple-election	Acceptor={a1,a2,a3} Proposer={p1,p2}	✗ (11/12)	5
23	pyv-toy-consensus-epr	Node={n1,n2,n3} Value={v1,v2}	✗ (6/8)	9
24	ex-toy-consensus	Node={n1,n2,n3} Value={v1,v2}	✗ (1/4)	1
25	pyv-client-server-db-ae	Node={n1,n2,n3} Request = {r1,r2,r3} Response={p1,p2,p3} DbRequestId={i1,i2}	✓ (40/40)	20
26	pyv-hybrid-reliable-broadcast	n/a		
27	pyv-firewall	Node={n1,n2,n3}	✗ (4/10)	23
28	ex-majorityset-leader-election	Node={n1,n2,n3}	✗ (9/12)	9
29	pyv-consensus-epr	Node={n1,n2,n3} Value={v1,v2}	✗ (39/40)	21
30	mldr	MaxTerm=3 MaxConfigVersion=3 Server={n1,n2,n3,n4}	✗ (15/24)	226

able to discover an inductive invariant for an industrial scale dynamic reconfiguration protocol.

In future, our tool can be extended to allow for automatic quantifier template search, and further optimizations can be made to the lemma invariant generation and selection procedures. It would be interesting to explore ways in which the invariant generation procedure can be guided more directly by the generated counterexamples to induction, as a means to prune the search space of candidate invariants more efficiently, perhaps using techniques similar to those presented in [120]. We would also be interested to see if quantifier structures can be inferred from the protocol syntax itself. Improving the performance of TLC, or experimenting with other, more efficient model checkers [18] would be another avenue, since model checking performance is a main bottleneck of our current approach. Some related ideas are explored in Chapter 5.

Chapter 5

Interpretable Protocol Verification by Inductive Proof Decomposition

The deductive method for verifying safety properties attempts to establish that a predicate G is invariant by showing that it is inductive. This process can be made systematic, but is always tedious. Typically, this is done by conjoining additional predicates in an incremental fashion...until an inductive $G^m_\wedge$ is found. In one well-known example, 57 such strengthenings were required...and the process consumed several weeks...the transformation of a desired safety property into a provably inductive invariant remains the most difficult and costly element in deductive verification, and systematic methods are sorely needed.

John Rushby, *Verification Diagrams Revisited* [121]

The content in this chapter is based on the work presented in [52], and our associated verification tool and source code can be found at [122].

5.1 Introduction

Over the past several years, particularly in the domain of distributed protocol verification, there have been several recent efforts to develop more automated inductive invariant development techniques [33, 38–40]. Many of these tools are based on modern model checking algorithms like IC3/PDR [30, 38, 40–42], and others based on syntax-guided or enumerative invariant synthesis methods [43–45]. These techniques have made significant progress on solving various classes of distributed protocols, including some variants of real world protocols like the Paxos consensus protocol [41, 46]. The theoretical complexity limits facing these techniques, however, limit their ability to be fully general [47] and, even in practice, the performance of these tools on complex protocols is still unpredictable, and their failure modes can be opaque.

In particular, one key drawback of these methods is that, in their current form, they are very much "all or nothing". That is, a given problem can either be automatically solved with no manual proof effort, or the problem falls outside the method's scope and a failure is reported. In the latter case, little assistance is provided in terms of how to develop a manual proof or how a human can offer guidance to the tool. We believe there is significant utility in providing a smoother transition between these possible outcomes. In practice, real world, large-scale verification efforts often benefit from some amount of human interpretability and interaction i.e., a human provides guidance when an automated engine is unable to automatically prove certain properties about a design or protocol. This may involve simplifying the problem statement given to the tool, or completing some part of the tool's proof process by hand. Recent verification efforts of industrial scale protocols often note the high amount of human effort in developing inductive invariants. Some leave human integration as future goals [24, 48], while others have adopted a paradigm of integrating human assistance to accelerate proofs for larger verification problems e.g., in the form of a manually developed refinement hierarchy [41, 49].

In this chapter we present *inductive proof decomposition*, a new technique for inductive invariant development that aims to address these limitations of existing approaches. Our technique utilizes the underlying compositional structure of an inductive invariant to guide its development, based on the insight that a standard inductive invariant can be decomposed into an *inductive proof graph*. This graph structure makes explicit the induction dependencies between lemmas of an inductive invariant, and their relationship to the logical transitions of a concurrent or distributed protocol. It serves as a core guidance mechanism for inductive invariant development, by making the global dependency structure apparent. In addition, the structure of the proof graph allows for localized reasoning about proof obligations, enabling a user to focus on small sub-problems of the inductive proof rather than a large, monolithic inductive invariant.

We build a technique for automatically and efficiently synthesizing these proof graphs, thus enabling automation while preserving amenability to human interaction and interpretability. We demonstrate that these proof graphs can be presented to and interpreted directly by a human user, facilitating a concrete and effective diagnosis and interaction process, enhancing interpretability of both the final inductive proof and the intermediate results. In addition, our automated synthesis technique also takes advantage of the proof graph structure to accelerate its local synthesis tasks by computing local variable *slices* at nodes of the graph, via localized static analyses. That is, we are able to project away state variables that are irrelevant to proving a local proof obligation, allowing for both improved efficiency and interpretability.

We apply our technique to develop inductive invariants of several large-scale distributed and concurrent protocol specifications, including an industrial-scale specification of the Raft [56] consensus protocol, demonstrating the effectiveness of our technique. We also provide an empirical evaluation of the interpretability and interaction features of our method.

In summary, our contributions are as follows:

- Definition and formalization of *inductive proof graphs*, a formal structure representing the logical dependencies between conjuncts of an inductive invariant and actions of a distributed protocol.
- *Inductive proof decomposition*, a new compositional inductive invariant development technique that is amenable both to efficient automated synthesis and fine-grained human interaction and interpretability.
- Implementation of our technique in a verification tool, Scimitar, and an empirical evaluation on several distributed protocols, including a large-scale specification of the Raft [56] consensus protocol.

5.2 Overview

To illustrate the core ideas of *inductive proof decomposition*, our inductive invariant development technique, we walk through it on a small example protocol.

Figure 5.1 shows a formal specification of a simple consensus protocol, defined as a symbolic transition system. This protocol utilizes a simple leader election mechanism to select values, and is parameterized on a set of nodes, *Node*, a set of values to be chosen, *Value*, and *Quorum*, a set of intersecting subsets of *Node*. Nodes can vote at most once for another node to become leader, and once a node garners a quorum of votes it may become leader and decide a value. The top level safety property, *NoConflictingValues*, shown in Figure 5.1, states that no two differing values can be chosen. The protocol's specification consists of 6 state variables and 5 distinct protocol *actions*, expressed in a *guarded action* style i.e., actions are of the form $A = Pre \land Post$, where *Pre* is a predicate over current state variables and *Post* is a conjunction of update formulas where x_i' refers to the value of x_i in the next state of a transition.

Our overall goal is to verify that a given protocol like the one in Figure 5.1 satisfies its specified safety property. We can do this by discovering an *inductive invariant*, which is an invariant that (1) holds in all initial states of the system, is (2) closed under transitions of the protocol, and (3) implies our safety property. For example, given the protocol of Figure 5.1 as input, we may discover an inductive invariant such as *Ind* shown in Figure 5.2. *Ind* is the conjunction of the original safety property, plus 7 more *lemmas*, which strengthen this *NoConflictingValues* safety property (thus ensuring that *Ind* logically implies *NoConflictingValues*). Even for such a relatively simple protocol, the inductive invariant is non-trivial in both size and logical complexity of its predicates.

Our technique, *inductive proof decomposition*, utilizes the underlying compositional structure of any inductive invariant to guide the development of an invariant such as

CONSTANTS $Node, Value, Quorum$

VARIABLES $voteRequestMsg, voted, voteMsg, votes, leader, decided$

Protocol actions.

$\textbf{SendRequestVote}(src, dst) \triangleq$
$\quad \land\ voteRequestMsg' = voteRequestMsg \cup \{\langle src, dst \rangle\}$

$\textbf{SendVote}(src, dst) \triangleq$
$\quad \land\ \neg voted[src]$
$\quad \land\ \langle dst, src \rangle \in voteRequestMsg$
$\quad \land\ voteMsg' = voteMsg \cup \{\langle src, dst \rangle\}$
$\quad \land\ voted'[src] := \text{True}$
$\quad \land\ voteRequestMsg' = voteRequestMsg \setminus \{\langle src, dst \rangle\}$

$\textbf{RecvVote}(n, sender) \triangleq$
$\quad \land\ \langle sender, n \rangle \in voteMsg$
$\quad \land\ votes'[n] := votes[n] \cup \{sender\}$

$\textbf{BecomeLeader}(n, Q) \triangleq$
$\quad \land\ Q \subseteq votes[n]$
$\quad \land\ leader'[n] := \text{True}$

$\textbf{Decide}(n, v) \triangleq$
$\quad \land\ leader[n]$
$\quad \land\ decided[n] = \{\}$
$\quad \land\ decided'[n] := \{v\}$

Safety property.

$NoConflictingValues \triangleq$
$\quad \forall n_1, n_2 \in Node, v_1, v_2 \in Value :$
$\qquad (v_1 \in decided[n_1] \land v_2 \in decided[n_2]) \Rightarrow (v_1 = v_2)$

Figure 5.1: State variables, protocol actions, and safety property (*NoConflictingValues*) for the *SimpleConsensus* protocol. Initial conditions omitted for brevity.

the one in Figure 5.2. Specifically, our technique is centered around a data structure called an *inductive proof graph*, which we use to develop inductive invariants *incrementally* and *compositionally*. This data structure is amenable to automated synthesis while also facilitating fine-grained human interaction and interpretability due to its explicit compositional structure.

A complete inductive proof graph corresponding to the inductive invariant *Ind* of Figure 5.2 is shown in Figure 5.3. The main nodes of an inductive proof graph, *lemma nodes*, correspond to lemmas of a system (so can be mapped to lemmas of a traditional inductive invariant), and the edges represent *induction dependencies* between these lemmas. This dependency structure is also decomposed by protocol actions, represented in the graph via *action nodes*, which are associated with each lemma node, and map to distinct protocol actions e.g., the actions of *SimpleConsensus* listed in Figure 5.1. Each action node

$UniqueLeaders \triangleq$
$\forall n_1, n_2 \in Node : \; leader[n1] \land leader[n_2] \Rightarrow (n_1 = n_2)$
$LeaderHasQuorum \triangleq$
$\forall n \in Node : leader[n] \Rightarrow (\exists Q \in Quorum : votes[n] = Q)$
$LeadersDecide \triangleq$
$\forall n \in Node : (decided[n] \neq \{\}) \Rightarrow leader[n]$
$NodesVoteOnce \triangleq$
$\forall n, n_i, n_j \in Node :$
$\neg(n_i \neq n_j \land n \in votes[n_i] \land n \in votes[n_j])$

Ind $\triangleq$ Inductive invariant.
$\land$ *NoConflictingValues* (Safety)
$\land$ *UniqueLeaders*
$\land$ *LeaderHasQuorum*
$\land$ *LeadersDecide*
$\land$ *NodesVoteOnce*
$\land$ *VoteRecvdImpliesVoteMsg*
$\land$ *VoteMsgsUnique*
$\land$ *VoteMsgImpliesVoted*

Figure 5.2: Complete inductive invariant, *Ind*, for proving *NoConflictingValues* for the *SimpleConsensus* protocol from Figure 5.1, with selected lemma definitions.

of this graph is then associated with a corresponding *inductive proof obligation*. That is, each action node A with source lemmas $L_1, ..., L_k$ and target lemma L is associated with the corresponding proof obligation

$$(L \land L_1 \land \cdots \land L_k \land A) \Rightarrow L' \tag{5.1}$$

where L' denotes lemma L applied to the next-state (primed) variables. An additional, key feature of the proof graph is that each local node is associated with a *variable slice*, a subset of protocol variables sufficient to consider for discharging that node. Slices are computed from a static analysis of that node's lemma-action pair, and can be seen illustrated in Figure 5.3, which annotates each proof node with its variable slice.

At a high level, our approach to inductive invariant development is to incrementally construct an inductive proof graph, working backwards from a specified safety property, and with specific guidance along the way. This is illustrated more concretely in Figure 5.4, which shows a sample of possible steps in construction of the inductive proof graph for the *NoConflictingValues* safety property of *SimpleConsensus*. Nodes that are unproven (shown in orange and marked with ✗), means that there are outstanding counterexamples for those inductive proof obligations. At a high level, the goal of the invariant development process is to, at each unproven node, discover support lemmas that make the lemma node inductive relative to this set of lemmas (e.g. satisfying Formula

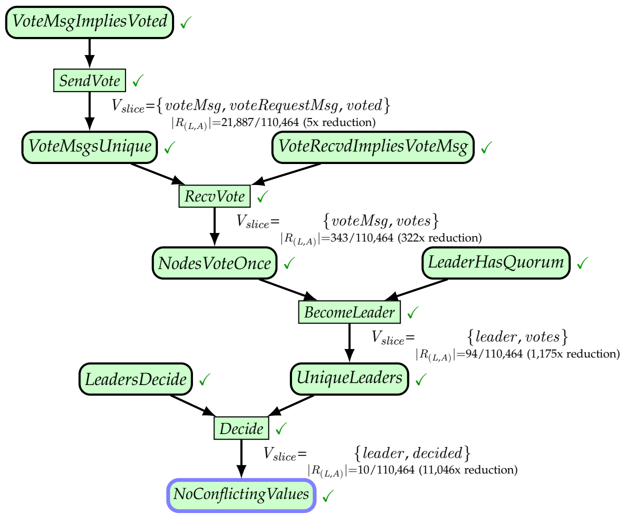
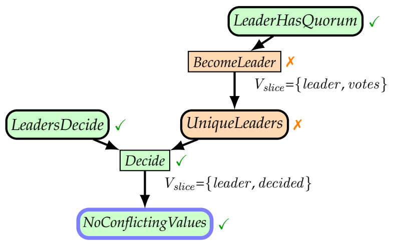

VoteMsgImpliesVoted ✓
SendVote ✓
V_{slice}={voteMsg, voteRequestMsg, voted}
$|R_{(L,A)}|$=21,887/110,464 (5x reduction)
VoteMsgsUnique ✓
VoteRecvdImpliesVoteMsg ✓
RecvVote ✓
V_{slice}={voteMsg, votes}
$|R_{(L,A)}|$=343/110,464 (322x reduction)
NodesVoteOnce ✓
LeaderHasQuorum ✓
BecomeLeader ✓
V_{slice}={leader, votes}
$|R_{(L,A)}|$=94/110,464 (1,175x reduction)
LeadersDecide ✓
UniqueLeaders ✓
Decide ✓
V_{slice}={leader, decided}
$|R_{(L,A)}|$=10/110,464 (11,046x reduction)
NoConflictingValues ✓

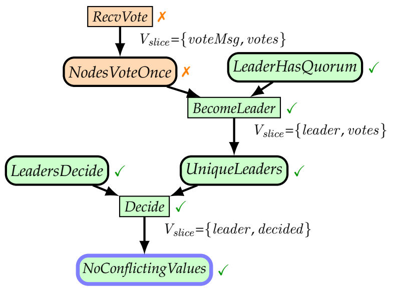
Figure 5.3: A complete inductive proof graph for *SimpleConsensus* protocol corresponding to the inductive invariant in Figure 5.2. Local variable slices are shown as V_{slice}, along with the size of the reachable state set slice at that node, indicated as $|R_{(L,A)}|$, along with the reduction factor over the full set of reachable states (of size 110,464) computed during synthesis.

5.1), discharging that proof obligation.

For example, in Figure 5.4a, the current focus is on discharging the unproven *Unique-Leaders* node. Note also that the variable slice associated with this node is shown below as $\{leader, votes\}$ (2 of 6 total state variables), indicating that only those state variables must be considered when developing a support lemma, focusing the reasoning task. In addition, counterexamples to the inductive proof obligation at that node (*counterexamples to induction*) can be examined to guide development of a support lemma. So, a new lemma, *NodesVoteOnce*, may then be synthesized to discharge *UniqueLeaders* and added to the graph, as shown in Figure 5.4b. As shown there, newly synthesized support lemmas



(a) In-progress proof graph (Step 1).



(b) In-progress proof graph (Step 2).

Figure 5.4: Example progression of inductive proof graph development for *SimpleConsensus*. Nodes in orange with ✗ are those with remaining inductive proof obligations to be discharged, and those in green with ✓ represent those with all obligations discharged.

create new proof obligations to consider (e.g. via *NodesVoteOnce*), with different variable slices. The process continues until all nodes are discharged e.g., leading to a complete inductive proof graph as shown in Figure 5.3.

A main feature of the approach to inductive invariant development as outlined above is that it is amenable both to efficient automation and interaction from a human user. With this in mind, we build an automated technique for synthesizing inductive proof graphs using a syntax-guided invariant synthesis technique [45, 104, 123]. The structure of the inductive proof graph and localized nature of these synthesis tasks also enables several *slicing* based optimizations, accelerating our automated synthesis routine. Crucially, due to the incremental maintenance of the proof graph during this overall synthesis procedure, we can allow fine-grained feedback and interaction from a human in the case of failure to produce a complete proof.

In the remainder of this chapter, we formalize the above ideas and techniques in more detail, and present an evaluation applying our techniques to several complex distributed and concurrent protocols.

5.3 Inductive Proof Graphs

Our inductive invariant development technique is based around a core logical data structure, the *inductive proof graph*, which we discuss and formalize in this section. This graph encodes the structure of an inductive invariant in a way that is amenable to efficient automated synthesis, and also to localized reasoning and human interpretability, as we discuss further in Sections 5.4 and 5.5.

5.3.1 Decomposing Inductive Invariants

A *monolithic* approach to inductive invariant development, where one searches for a single inductive invariant that is a conjunction of smaller lemmas, is a general proof methodology for safety verification [53]. Any monolithic inductive invariant, however, can alternatively be viewed in terms of its *relative induction* dependency structure, which is the initial basis for our formalization of inductive proof graphs, and which decomposes an inductive invariant based on this structure.

Namely, for a transition system $M = (I, T)$ and associated invariant S, given an inductive invariant

$$Ind = S \wedge L_1 \wedge \ldots \wedge L_k$$

each lemma in this overall invariant may only depend inductively on some other subset of lemmas in Ind. More formally, proving the consecution step of such an invariant requires establishing validity of the following formula

$$(S \wedge L_1 \wedge \ldots \wedge L_k) \wedge T \Rightarrow (S \wedge L_1 \wedge \ldots \wedge L_k)' \quad (5.2)$$

which can be decomposed into the following set of independent proof obligations:

$$\begin{aligned}
&(S \wedge L_1 \wedge \ldots \wedge L_k) \wedge T \Rightarrow S' \\
&(S \wedge L_1 \wedge \ldots \wedge L_k) \wedge T \Rightarrow L_1' \\
&\qquad\qquad\qquad \vdots \\
&(S \wedge L_1 \wedge \ldots \wedge L_k) \wedge T \Rightarrow L_k'
\end{aligned} \tag{5.3}$$

If the overall invariant Ind is inductive, then each of the proof obligations in Formula 5.3 must be valid. That is, we say that each lemma in Ind is inductive *relative* to the conjunction of lemmas in $\{S, L_1, \ldots, L_k\}$.

With this in mind, if we define $\mathcal{L} = \{S, L_1, \ldots, L_k\}$ as the lemma set of Ind, we can consider the notion of a *support set* for a lemma in $\mathcal{L}$ as any subset $U \subseteq \mathcal{L}$ such that L is inductive relative to the conjunction of lemmas in U i.e., $(\bigwedge_{\ell \in U} \ell) \wedge L \wedge T \Rightarrow L'$. As shown above in Formula 5.3, $\mathcal{L}$ is always a support set for any lemma in $\mathcal{L}$, but it may not be the smallest support set. This support set notion gives rise a structure we refer to as the *lemma support graph*, which is induced by each lemma's mapping to a given support set, each of which may be much smaller than $\mathcal{L}$. We note that a minimal support set should be computable for any monolithic inductive invariant e.g., by computing a minimal support set for each lemma via computation of minimum unsatisfiable cores [124] of each proof obligation above in Formula 5.3. Computing a minimum unsatisfiable core is computationally hard, though in practice there are effective methods for computing approximations e.g. for enumerating minimal unsatisfiable cores [125].

For distributed and concurrent protocols, the transition relation of a system $M = (I, T)$ is typically a disjunction of several distinct actions i.e., $T = A_1 \vee \cdots \vee A_n$, as shown in the example of Figure 5.1. So, each node of a lemma support graph can be augmented with sub-nodes, one for each action of the overall transition relation. Lemma support edges in the graph then run from a lemma to a specific action node, rather than directly to a target lemma. Incorporation of this action-based decomposition now lets us define the full inductive proof graph structure.

Definition 3. *For a system $M = (I, T)$ with $T = A_1 \vee \cdots \vee A_n$, an inductive proof graph is a directed graph (V, E) where*

- *$V = V_L \cup V_A$ consists of a set of lemma nodes V_L and action nodes V_A, where*
  - *V_L is a set of state predicates over M.*
  - *$V_A = V_L \times \{A_1, \ldots, A_n\}$ is a set of action nodes, associated with each lemma node in V_L.*
- *$E \subseteq V_L \times V_A$ is a set of lemma support edges.*

Figure 5.5 shows an example of an inductive proof graph along with its corresponding inductive proof obligations annotating each action node. Note that, for simplicity, when

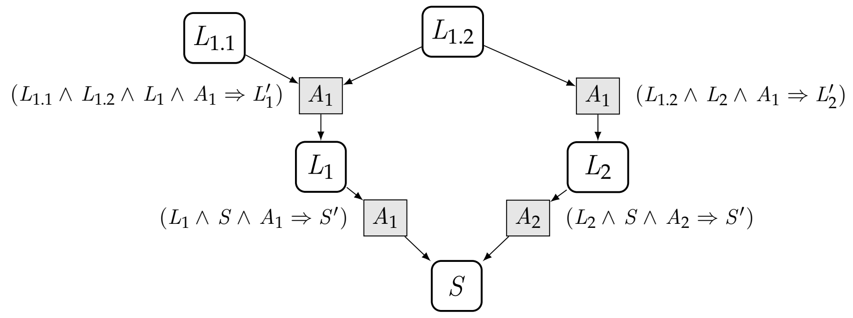

$L_{1.1}$
$L_{1.2}$
$(L_{1.1} \wedge L_{1.2} \wedge L_1 \wedge A_1 \Rightarrow L_1')$
A_1
A_1
$(L_{1.2} \wedge L_2 \wedge A_1 \Rightarrow L_2')$
L_1
L_2
$(L_1 \wedge S \wedge A_1 \Rightarrow S')$
A_1
A_2
$(L_2 \wedge S \wedge A_2 \Rightarrow S')$
S

Figure 5.5: Abstract inductive proof graph example, with lemma and action nodes (in gray), and associated inductive proof obligations next to each action node. Self-inductive obligations are omitted for brevity, and action to lemma node relationships are shown as incoming lemma edges.

depicting inductive proof graphs, if an action node is self-inductive, we omit it. Also, action nodes are, by default, always associated with a particular lemma, so when depicting these graphs, we show edges that connect action nodes to their parent lemma node, even though these edges do not appear in the formal definition.

5.3.2 Inductive Proof Graph Validity

We now define a notion of *validity* for an inductive proof graph. That is, we define conditions on when a proof graph can be seen as corresponding to a complete inductive invariant and, correspondingly, when the lemmas of the graph can be determined to be invariants of the underlying system.

Definition 4 (Local Action Validity)**.** *For an inductive proof graph $(V_L \cup V_A, E)$, let the inductive support set of an action node $(L, A) \in V_A$ be defined as $Supp_{(L,A)} = \{\ell \in V_L : (\ell, (L, A)) \in E\}$. We then say that an action node (L, A) is* locally valid *if the following holds:*

$$\left(\bigwedge_{\ell \in Supp_{(L,A)}} \ell \right) \wedge L \wedge A \Rightarrow L' \tag{5.4}$$

Definition 5 (Local Lemma Validity)**.** *For an inductive proof graph $(V_L \cup V_A, E)$, a lemma node $L \in V_L$ is* locally valid *if all of its associated action nodes, $\{L\} \times \{A_1, \ldots, A_n\}$, are locally valid. We alternately refer to a lemma node that is locally valid as being discharged.*

Based on the above local validity definitions, the notion of validity for a full inductive proof graph is then straightforward to define.

Definition 6 (Inductive Proof Graph Validity)**.** *An inductive proof graph is valid whenever all lemma nodes of the graph are* locally valid.

As an example, Figure 5.3 shows an example of a complete inductive proof graph satisfying the validity condition, whereas Figure 5.4 illustrates partial proof graphs, neither of which satisfy validity.

The validity notion for an inductive proof graph establishes lemmas of such a graph as invariants of the underlying system M, since a valid inductive proof graph can be seen to correspond with a complete inductive invariant. We formalize this as follows.

Lemma 9. *For a system $M = (I, T)$, if an inductive proof graph $(V_L \cup V_A, E)$ for M is valid, and $I \Rightarrow L$ for every $L \in V_L$, then the conjunction of all lemmas in V_L is an inductive invariant.*

Proof. The conjunction of all lemmas in a valid graph must be an inductive invariant, since every lemma's support set exists as a subset of all lemmas in the proof graph, and all lemmas hold on the initial states. □

Theorem 10. *For a system $M = (I, T)$, if a corresponding inductive proof graph $(V_L \cup V_A, E)$ for M is valid, and $I \Rightarrow L$ for every $L \in V_L$, then every $L \in V_L$ is an invariant of M.*

Proof. By Lemma 9, the conjunction of all lemmas in a valid proof graph is an inductive invariant, and for any set of predicates, if their conjunction is an invariant of M, then each conjunct must be an invariant of M. □

5.3.3 Cycles and Subgraphs

Note that the definition of proof graph validity does not imply any restriction on cycles in a valid inductive proof graph. For example, a proof graph that is a pure k-cycle can be valid. For example, a simple *ring counter* system with 3 state variables, a, b, and c, where a single value gets passed from a to b to c and exactly one variable holds the value at any time. An inductive invariant establishing the property that a always has a well-formed value will consist of 3 properties that form a 3-cycle, each stating that a, b and c's state are, respectively, always well-formed. Figure 5.6 presents a small example specification and associated proof graph for such a system.

Also note that based on the above validity definition, any subgraph of an inductive proof graph can also be considered valid, if it meets the necessary conditions. Thus, in combination with Theorem 10 this implies that, even if a particular proof graph is not valid, there may be subgraphs that are valid and, therefore, can be used to infer that a subset of lemmas in the overall graph are valid invariants.

5.3.4 Local Variable Slices

A benefit of the inductive proof graph is that its structure provides a way to focus, at each graph node, on a potentially small subset of state variables that are relevant for discharging that proof node. That is, when considering an action node (L, A), any support lemmas for this node must, to a first approximation, refer only to state variables that appear in either L or A. We make use of this general idea to compute a *variable slice*

VARIABLES a, b, c

Init $\triangleq a = 1 \land b = 0 \land c = 0$

$A \triangleq a > 0 \land b' = a \land a' = 0 \land c' = c$
$B \triangleq b > 0 \land c' = b \land b' = 0 \land a' = a$
$C \triangleq c > 0 \land a' = c \land c' = 0 \land b' = b$

Next $\triangleq A \lor B \lor C$

Inv $\triangleq a \in \{0, 1\}$ (* top-level invariant. *)

$L_1 \triangleq b \in \{0, 1\}$
$L_2 \triangleq c \in \{0, 1\}$

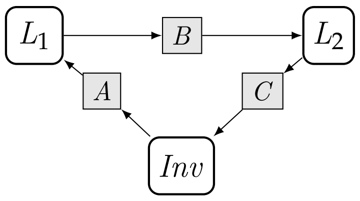


Figure 5.6: Example protocol that exhibits a cyclic inductive proof graph for establishing invariant *Inv*.

at each node, allowing us to project away any protocol state variables that are irrelevant for establishing a valid support set for that node.

Intuitively, the variable slice of an action node (L, A) can be understood as the union of: (1) the set of all variables appearing in the precondition of A, (2) the set of all variables appearing in the definition of lemma L, (3) for any variables in L, the set of all variables upon which the update expressions of those variables depend. Figure 5.3 shows an example of a proof graph annotated with its variable slices at each node. More precisely, our slicing computation at each action node is based on the following static analysis of a lemma and action pair (L, A). First, let $\mathcal{V}$ be the set of all state variables in our system, and let $\mathcal{V}'$ refer to the primed, next-state copy of these variables. For an action node (L, A), we have $L \land A \Rightarrow L'$ as its initial inductive proof obligation. Like the example protocol from Figure 5.1, we consider actions to be written in *guarded action* form, so they can be expressed as $A = Pre \land Post$, where Pre is a predicate over a set of current state variables, denoted $Vars(Pre) \subseteq \mathcal{V}$, and $Post$ is a conjunction of update expressions of the form $x_i' = f_i(\mathcal{D}_i)$, where $x_i' \in \mathcal{V}'$ and $f_i(\mathcal{D}_i)$ is an expression over a subset of current state variables $\mathcal{D}_i \subseteq \mathcal{V}$.

Definition 7. *For an action $A = Pre \land Post$ and variable $x_i' \in \mathcal{V}'$ with update expression $f_i(\mathcal{D}_i)$ in Post, we define the cone of influence of x_i', denoted $COI(x_i')$, as the variable set $\mathcal{D}_i$. For a set of primed state variables $\mathcal{X} = \{x_1', \ldots, x_n'\}$, we define $COI(\mathcal{X})$ simply as $COI(x_1') \cup \cdots \cup COI(x_n')$*

Now, if we let $\mathit{Vars}(\mathit{Pre}) \subseteq \mathcal{V}$ and $\mathit{Vars}(L') \subseteq \mathcal{V}'$ be the sets of state variables that appear in the expressions of L' and Pre, respectively, then we can formally define the notion of a slice as follows.

Definition 8. *For an action node (L, A), its variable slice is the set of state variables*

$$\mathit{Slice}(L, A) = \mathit{Vars}(\mathit{Pre}) \cup \mathit{Vars}(L) \cup \mathit{COI}(\mathit{Vars}(L'))$$

Based on this definition, we can now show that a variable slice is a strictly sufficient set of variables to consider when developing a support set for an action node.

Theorem 11. *For an action node (L, A), if a valid support set exists, there must exist one whose expressions refer only to variables in $\mathit{Slice}(L, A)$.*

Proof. Without loss of generality, the existence of a support set for (L, A) can be defined as the existence of a predicate Supp such that the formula

$$\mathit{Supp} \land L \land A \land \neg L' \tag{5.5}$$

is unsatisfiable. As above, actions are of the form $A = \mathit{Pre} \land \mathit{Post}$, where Post is a conjunction of update expressions, $x_i' = f_i(\mathcal{D}_i)$, so Formula 5.5 can be re-written as

$$\mathit{Supp} \land L \land \mathit{Pre} \land \neg L'[\mathit{Post}] \tag{5.6}$$

where $L'[\mathit{Post}]$ represents the expression L' with every $x_i' \in \mathit{Vars}(L')$ substituted with the update expression given by $f_i(\mathcal{D}_i)$. From this, it is straightforward to show our original goal. If $L \land \mathit{Pre} \land \neg L'[\mathit{Post}]$ is satisfiable, and there exists a Supp that makes Formula 5.6 unsatisfiable, then clearly Supp must only refer to variables that appear in $L \land \mathit{Pre} \land \neg L'[\mathit{Post}]$, which are exactly the set of variables in $\mathit{Slice}(L, A)$.

$\square$

5.4 Synthesizing Inductive Proof Graphs

Our overall technique for developing inductive invariants uses the inductive proof graph as its guiding data structure. We build an algorithm for automatically synthesizing inductive proof graphs, allowing for a smooth transition between both (1) automation and (2) human interaction and interpretability.

Our proof graph synthesis algorithm extends ideas from previously explored inductive invariant synthesis techniques [45, 104], applying them in our context to incrementally synthesize proof graphs efficiently, by running localized synthesis tasks that take advantage of various slicing-based optimizations. As discussed previously, incremental maintenance of the proof graph provides an effective, fine-grained interpretability and diagnosis mechanism when our automated technique does not synthesize a complete proof graph, or has made partial progress.

Algorithm 4 Inductive proof graph synthesis.

```
1: Inputs:
2: Transition system $M = (I, T)$, safety property $S$.
3: Grammar $Preds$, reachable state set $R$.
4: action: SynthIndProofGraph($M$, $S$, $Preds$, $R$)
5:     $(V_L, V_A, E) \leftarrow (\{S\}, \{S\} \times \{A_1, \dots, A_n\}, \emptyset)$
6:     $G \leftarrow (V_L \cup V_A, E)$                    ▷ Initialize proof graph.
7:     $failed \leftarrow \emptyset$
8:     if $\forall a \in V_A : (a \text{ is locally valid}) \vee (a \in failed)$ then
9:         return $(G, failed)$.        ▷ Returned graph $G$ is valid if $failed = \emptyset$
10:    else
11:        Pick $(L, A) \in (V_A \setminus failed)$ where $(L, A)$ is not locally valid.
12:        $(Supp_{(L,A)}, success) \leftarrow$ SynthLocal($M$, $Preds$, $R$, $L$, $A$)
13:        if $\neg success$ then
14:            $failed \leftarrow failed \cup \{(L, A)\}$
15:            goto Line 8
16:        end if
17:        $V_L \leftarrow V_L \cup Supp$                  ▷ Update the proof graph.
18:        $V_A \leftarrow V_A \cup (Supp \times \{A_1, \dots, A_k\})$
19:        $E \leftarrow E \cup (Supp \times \{(L, A)\})$
20:        goto Line 8.
21:    end if
22: end action:
```

5.4.1 Our Synthesis Algorithm

At a high level, our inductive invariant inference algorithm constructs an inductive proof graph incrementally, starting from a given safety property S as its initial lemma node. It works backwards from the safety property by synthesizing support lemmas for remaining, un-discharged proof nodes.

To synthesize these support lemmas at local graph nodes, we perform a local, syntax-guided invariant synthesis routine that is based on an extension of a prior technique [45], adapted to this compositional, graph-based setting.

Once all nodes of the proof graph have been discharged, the algorithm terminates, returning a complete, valid inductive proof graph. If it cannot discharge all nodes successfully, either due to a timeout or other specified resource bounds, it may return a partial, incomplete proof graph, containing some nodes that have not been discharged and are instead marked as *failed*. The overall algorithm is described formally in Algorithm 4, which we walk through and discuss in more detail below.

Formally, our algorithm takes as input a safety property S, a transition system $M = (I, T)$, and tries to prove that S is an invariant of M by synthesizing an inductive proof graph sufficient for proving S. It starts by initializing an inductive proof graph $(V_L \cup V_A, E)$ where $V_L = \{S\}$, $V_A = \{S\} \times \{A_1, \dots, A_n\}$, and $E = \emptyset$, as shown on Line 5 of Algorithm 4. From here, the graph is incrementally extended by synthesizing support lemmas and adding support edges from these lemmas to action nodes that are not yet discharged.

As shown in the main loop of Algorithm 4 at Line 11, the algorithm repeatedly selects some node of the graph that is not discharged, and runs a local inference task at that node (Line 12 of Algorithm 4). Our local inference routine for synthesizing support lemmas $Supp_{(L,A)}$, is a subroutine, SYNTHLOCAL, of the overall algorithm, and is shown separately as Algorithm 5, and described in more detail below in Section 5.4.2. Once the local synthesis call SYNTHLOCAL completes successfully, the generated set of support lemmas, $Supp_{(L,A)}$, is added to the current proof graph (Line 17 of Algorithm 4), and if there are remaining nodes that are not discharged, the algorithm continues. Otherwise, it terminates with a complete, valid proof graph (Line 9 of Algorithm 4).

It is also possible that, throughout execution, some local synthesis tasks fail, due to various reasons e.g., exceeding a local timeout, exhausting a grammar, or reaching some other specified execution or resource bound. In this case, we mark a node as *failed* (Line 14 of Algorithm 4), and continue as before, excluding failed nodes from future consideration for local inference. Due to our marking of nodes as locally failed, it is possible for the algorithm to terminate with some nodes that are not discharged (i.e. are marked in *failed*). We discuss this aspect further in our evaluation section where we discuss the interpretability and diagnosis capabilities of our approach.

The above outlines the execution of our algorithm at a high level. To accelerate it, however, we rely on several key optimizations that are enabled by the variable slicing computations we perform during local inference. We discuss these in more detail below and how they accelerate our overall inference procedure.

5.4.2 Local Lemma Synthesis with Slicing

As described above and shown in Algorithm 5, our local synthesis routine, SYNTHLOCAL, consists of a main loop that searches for candidate protocol invariants to serve as a valid set of support lemmas. Prior syntax-guided approaches [33, 45] for synthesizing inductive invariants utilize a set of reachable protocol states, R, to look for these invariants, and *counterexamples to induction* (CTIs) to guide selection from among these invariants those that are relevant to the current inductive proof obligation. We adopt a similar approach in the context of each local proof node synthesis task.

The search space for these candidate invariants is defined by a grammar of state predicates given as input to our overall algorithm, *Preds*, and the GENLEMMAINVS routine uses a set of reachable system states R to validate candidate invariants sampled from this grammar. In general, we search for invariant candidates in increasing size order (e.g. in max number of syntactic terms) until reaching a specified search bound. Thus, the number of candidates generated by *Preds* and the size of R are the main factors impacting the performance of these local synthesis tasks, which make up the main computational

Algorithm 5 Local support lemma synthesis.

```
1: action: SynthLocal($M$, $Preds$, $R$, $L$, $A$)
2:     $Vars_{(L,A)} \leftarrow$ Slice($L$, $A$)                                  ▷ See Definition 8.
3:     $R_{(L,A)} \leftarrow \{\pi_{Vars_{(L,A)}}(r) : r \in R\}$                  ▷ Project $R$ to the slice.
4:     $Preds_{(L,A)} \leftarrow \{p \in Preds : Vars(p) \subseteq Vars_{(L,A)}\}$   ▷ Grammar slice.
5:     $Supp_{(L,A)} \leftarrow \emptyset$
6:     $CTIs \leftarrow$ CTIs($M, L, A$)         ▷ Find states s.t. $\neg(Supp_{(L,A)} \wedge L \wedge A \wedge L')$.
7:     while $CTIs \neq \emptyset$ do
8:         $Invs \leftarrow$ GenLemmaInvs($M, Vars_{(L,A)}, Preds_{(L,A)}, R_{(L,A)}$)
9:         if $\exists A \in Invs : A$ eliminates some CTI in $CTIs$ then
10:            Pick $L_{max} \in Invs$ that eliminates the most CTIs from $CTIs$.
11:            $Supp_{(L,A)} \leftarrow Supp_{(L,A)} \cup \{L_{max}\}$
12:            $CTIs \leftarrow CTIs \setminus \{s \in CTIs : s \not\models L_{max}\}$
13:        else
14:            either goto Line 8
15:            or return ($Supp_{(L,A)}$, $False$)                        ▷ Couldn't eliminate CTIs.
16:        end if
17:    end while
18:    return ($Supp_{(L,A)}$, $True$)                         ▷ Success: eliminated all CTIs
19: end action:
```

work of our overall algorithm. We accelerate these tasks by making use of the local variable slices at each node to apply both *grammar slicing* and *state slicing* optimizations.

At the beginning of local synthesis at a node (L, A), we use the local variable slice, $Vars_{(L,A)}$, to prune the set of predicates in the global set *Preds*. That is, we simply filter out any predicates in *Preds* that do not refer to a subset of variables in $Vars_{(L,A)}$ (Line 4 of Algorithm 5). We then also compute a local projection, $R_{(L,A)}$, of the reachable state set R (Line 3 of Algorithm 5), projecting out any variables absent from the local variable slice $Vars_{(L,A)}$ (Line 3 of Algorithm 5). We assume these projections can be computed efficiently, and could in theory be done upfront when *R* is first generated. Both the local grammar slice and state projection are then passed as inputs to our invariant enumeration routine, GenLemmaInvs.

We show in our evaluation how these optimizations can have a significant impact on the efficiency of the synthesis procedure, since we can often prune out large portions of the state and grammar space.

5.5 Evaluation

We evaluated our technique to understand (1) how our automated synthesis technique performs, (2) to examine the interpretability of the generated proof graphs, and also (3) how our approach allowed us to develop an inductive proof for a large, complex distributed protocol when some amount of interaction was needed.

To evaluate (1) and (2), we test our technique on a set of distributed and concurrent protocols in a range of complexities, up to distributed protocol specifications considerably larger than those previously solved by other existing tools. For testing (3) we test a large, asynchronous, message-passing specification of the Raft consensus algorithm [56], which allowed us to evaluate (2) as well. To our knowledge, ours is the first automated inductive invariant synthesis effort for a specification of Raft of this complexity.

Implementation and Setup Our technique is implemented in our verification tool, Scimitar, which consists of approximately 6100 lines of Python code, and accepts as input protocols specified in the TLA+ specification language [35]. Internally, Scimitar uses the TLC model checker [82] for most of its compute-intensive inference sub-routines, like checking candidate lemma invariants and CTI generation and elimination checking. Specifically, it uses TLC to generate counterexamples to induction for finite protocol instances using a randomized search technique [107]. Our current implementation uses TLC version 2.15 with some modifications to enable the optimizations employed in our technique. We also use the TLA+ proof system (TLAPS) [70] to validate the correctness of the inductive invariants inferred by our tool. Code for our tool and benchmarks can be found at [122].

All of our experiments below were carried out on a 56-core Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz machine with 64GB of allocated RAM. We configured our tool to use a maximum of 24 worker threads for TLC model checking and other parallelizable inference tasks.

Benchmarks We used our tool to develop inductive invariants for establishing core safety properties of 6 protocol benchmarks. These protocols are summarized in Table 5.1, along with various statistics about the specifications and invariants. All formal specifications of these protocols are defined in TLA+, some of which existed from prior work and some of which we developed or modified based on existing specifications. Our aim in this benchmark was to evaluate our tool on a range of protocol complexities, to understand how it performs against existing techniques for smaller protocols, and then to examine both its performance gains on larger protocols, to examine the scale of protocols it could solve that existing tools could not.

The *TwoPhase* benchmark is a high level specification of the two-phase commit protocol [126], and *SimpleConsensus* is the consensus protocol presented in Section 5.2. The *Bakery* benchmark is a specification of Lamport's Bakery algorithm for mutual exclusion [127], which has only recently had attempts at automated verification [128]. We also test a specification of a concurrent cache coherence protocol, *GermanCache*, which has been used as a complex and challenging verification benchmark in past work [129].

The largest of our benchmarks is an industrial scale specification of the Raft consensus protocol [56]. The specification we use is based on a model similar to the original Raft formal specification [64, 130], and models asynchronous message passing between all

						Scimitar				endive
Benchmark	**LoC**	$\|R\|$	*Vars*	$\|A\|$	$\|Preds\|$	# Lemmas	Time	$\|R\|^{\sim}_{(L,A)}$	$\|Preds\|^{\sim}_{(L,A)}$	Time
SimpleConsensus	108	110,464	6	5	25	10	748	0.1%	56%	416
TwoPhase	195	288	6	7	18	14	348	5.9 %	66 %	173
GermanCache [129]	210	1,663,875	11	12	38	46	17165	0.3%	47 %	timeout
Bakery [127]	234	6,016,610	6	7	88	27	15854	1.2 %	50 %	timeout
AsyncRaft$_{ES}$	583	2,594,148	12	9	127	13	7088	7.1 %	34 %	timeout
AsyncRaft$_{PO}$	583	2,598,265	12	9	127	30	33735	0.7%	31 %	timeout

Table 5.1: Protocols used in evaluation and metrics on their specifications and inductive proof graphs. The $|R|$ column reports the size of the set of explored reachable states used during inference, $|Preds|$ is the number of predicates in the base grammar, and *Vars* and A, respectively, show the number of state variables and actions in the specification. *Time* shows the time in seconds to synthesize an inductive invariant. The (endive) column represents the baseline approach based on the technique of [45], when run with the same relevant parameters. A *timeout* entry indicates no invariant found after a 16 hour timeout. $|R|^{\sim}_{(L,A)}$ shows the median of state slice sizes computed during synthesis as a percentage of $|R|$, similarly for $|Preds|^{\sim}_{(L,A)}$.

nodes and fine-grained local state. We verify two core safety properties of the protocol. Namely, *ElectionSafety*, stating that no two leaders can be in the same term, and another higher level lemma, *PrimaryOwnsEntries*, stating that leader servers in Raft should always contain log entries they created in their term.

We note that our largest benchmark specification, for *AsyncRaft*, is of a complexity significantly greater than those tested in recent automated invariant inference techniques, so we consider them as the most relevant benchmarks for evaluating our automation and interpretability features. For example, even in a recent approach, DuoAI [44] reports the LoC of the largest protocol tested as 123 lines of code in the Ivy language [39], which is of a similar abstraction level to TLA+. Our largest specification of Raft is over 500 lines of TLA+. Thus, we view our benchmarks as examining scalability of our technique on protocols that are notably more complex than those tested by existing tools.

Note that all protocols tested are parameterized, meaning that they are typically infinite-state, but have some fixed set of parameters that can be instantiated with finite parameters e.g. the set of processes or servers. Our synthesis algorithm runs using finite instantiations of protocol parameters, but our grammar templates are general enough to infer invariants that are valid for all instantiations of the protocol. Once synthesized by our tool, we validate the correctness of the inductive invariants using the TLC model checker and the TLA+ proof system [70].

Results Summary Table 5.1 shows various statistics about the protocols we tested, including the number of state variables, number of actions, lines of code (LoC) in the TLA+ protocol specifications, number of lemmas in each proof graph, etc. More detailed results on the specifications and proof graphs for these benchmarks can also be found in Appendix B. We compare against another state of the art inductive invariant inference tool, *endive*, which was presented in [45], since it both accepts specifications in TLA+ and is also based on similar syntax-guided synthesis technique with similar input parameters. Thus, it makes a good candidate for comparisons since it has both performed strongly on modern distributed protocol inference benchmarks and is also the most analogous to our tool in terms of input and approach. To our knowledge, we are not aware of any other existing tool that can solve the largest benchmarks we test.

At a high level, the results can be understood as falling into roughly three distinct qualitative classes of performance. At the smallest protocol level, for benchmarks *TwoPhase* and *SimpleConsensus*, our approach successfully finds an inductive invariant, but its runtime is of comparable, albeit slower, performance than the *endive* tool, the baseline approach from [45]. We view this an expected artifact of our technique and implementation, which is optimized for scalability, at the cost of some upfront overhead when caching state slices, etc. The main goal for these smaller protocols, though, was to simply verify that our tool performs within a similar class of performance as existing approaches, and also to examine the generated proof graphs.

In the next larger class of protocols, including *Bakery* and *GermanCache*, our tool is able to solve both of these benchmarks whereas *endive* fails to solve any within a 16 hour timeout. These protocols are significantly larger than the previous benchmarks, as can be seen, for example, by their reachable state sets $|R|$, and the number of lemmas in each synthesized proof i.e., both with ≥ 20 lemmas, with *GermanCache* having 46 lemmas in its inductive proof graph.

We observed that our approach is able to leverage the slicing optimizations effectively to achieve these performance improvements on these difficult benchmarks. Note that in our implementation we in some cases compute slices at an even finer grained level than the full variable slice computed at a node, allowing for even greater acceleration. For example, if we check a set of properties that only refers to subset of variables in a local slice, we can use an even smaller state slice projection for those checks. For the *GermanCache* model, though $|R|$ is 1,663,875, the median state set size is under 1% of this full state set. We observe these slicing reductions similarly for *Bakery*, whose median state set sizes is under 2% of the reachable state set. Both of their median grammar slices are also near half of the full grammar size, which has a significant impact when searching over all candidate invariants generated by the predicates in the grammar.

The *AsyncRaft* protocol is the most complex benchmark we test, and we were able to synthesize an inductive proof graph for two high level safety properties, *ElectionSafety* (AsyncRaft$_{ES}$) and *PrimaryOwnsEntries* (AsyncRaft$_{PO}$), with both having median state sets under a few percent of the total reachable set, and similarly reduced grammar slices. We discuss the structure of those developed proof graphs and qualitative aspects of our experience developing them in more detail below in Section 5.5.1.

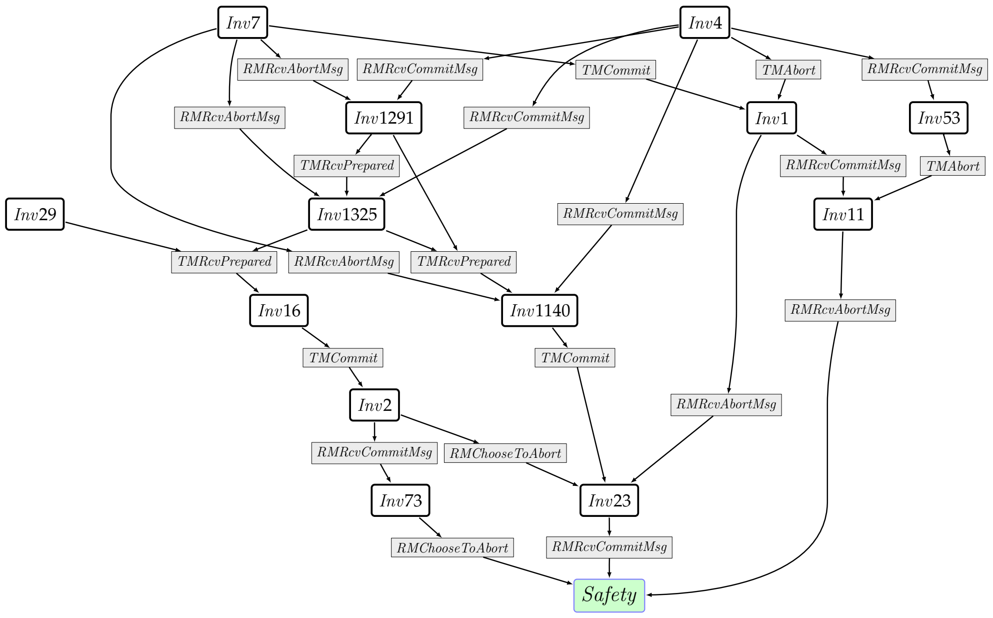


Figure 5.7: *TwoPhase* inductive proof graph for establishing safety property that decisions must be consistent across nodes. Excerpt of lemmas shown in Figure 5.8.

Examining Interpretability As a concrete examination of the interpretability of our method, Figure 5.7 shows a complete synthesized proof graph for the *TwoPhase* benchmark. This benchmark is a specification of the classic two-phase commit protocol [126], where a transaction manager aims to achieve agreement from a set of resource managers on whether to *commit* or *abort* a transaction. The proof graph shows action labels and omits full definitions of each lemma node, to illustrate the overall structure more clearly. This proof graph establishes the core safety property of two-phase commit which states that no two resource managers can come to conflicting commit and abort decisions, and provides an intuitive way to understand the structure of this inductive proof.

As seen in Figure 5.7, the root safety node, *Safety*, has 3 supporting action nodes, *RMRcvCommitMsg*, *RMRcvAbortMsg* and *RMChooseToAbort*, which represent, respectively, the actions that can directly falsify the target safety property of two-phase commit, via some resource manager committing or aborting. The support lemmas of *RMRcvCommitMsg* and *RMRcvAbortMsg* respectively, $Inv23$ and $Inv11$, for example, stipulate that presence of a commit or abort message must imply that no other resource manager has made a conflicting commit (abort) decision (definitions shown in Figure 5.8). This reasoning can also be seen tracing back to lower level support lemmas in the graph e.g., $Inv7$ which establishes invariants on the initial state of the transaction manager.

Overall, this proof graph admits a relatively tree-like structure, and we can naturally focus on small sub-components on the graph. We found that this case study generally supports our interpretability hypothesis i.e., that the inductive proof graph structure can provide insight into the reasoning structure of the proof, and can help guide its

$$Inv23 \triangleq \forall rm \in RM : (\text{"}Commit\text{"} \in msgsCommit) \Rightarrow (rmState[rm] \neq \text{"}aborted\text{"})$$
$$Inv11 \triangleq \forall rm \in RM : (\text{"}Abort\text{"} \in msgsAbort) \Rightarrow (rmState[rm] \neq \text{"}committed\text{"})$$
$$Inv7 \triangleq (\text{"}Abort\text{"} \in msgsAbort) \Rightarrow (tmState \neq \text{"}init\text{"})$$
$$Safety \triangleq \forall rm_1, rm_2 \in RM : \neg(rmState[rm_1] = \text{"}aborted\text{"} \land rmState[rm_2] = \text{"}committed\text{"})$$

Figure 5.8: Definitions of some lemma nodes from proof graph for *TwoPhase* in Figure 5.7, including the top-level safety property.

development and analysis.

5.5.1 Detailed Case Study: Raft Consensus Protocol

As a more in-depth evaluation of our technique for verifying a large scale protocol, we developed an inductive proof for a large-scale, asynchronous specification of Raft, to explore how our automated techniques and interpretability features are effective at facilitating this process. We verified a high level lemma of Raft, *PrimaryOwnsEntries*, which states that if a log entry in Raft exists in term T, then a leader in term T must have this entry in its own log.

Figure 5.9 shows the complete inductive proof graph that we synthesized for establishing this property of our Raft specification, which consists of 30 lemma nodes. The main actions of this proof graph correspond to those of standard Raft e.g., dealing with election of a leader (*RequestVote*, *HandleReqVoteReq*, *HandleReqVoteResp*, *BecomeLeader*) and replicating log entries between servers (*AppendEntries*, *AcceptAppendEntries*, *ClientRequest*). The core safety property is shown in green, and we show several important core synthesized support lemmas annotated in blue, which serve as helpful waypoints for understanding the structural components of this proof graph.

Specifically, along the *ClientRequest* support ancestry of the safety node, this leads to a first main support lemma, *ElectionSafety*, which is a key property of standard Raft stating that two leaders cannot be elected in the same term. Separately, the *BecomeLeader* parent subgraph of the safety property is supported by the *CandidateElectImpliesNoLogsInTerm* lemma, stating that if a candidate has gathered a quorum of votes, then there mustn't exist any logs in its term. The supporting subgraph for the *ElectionSafety* property, highlighted in orange, is largely independent of the subgraph supporting *CandidateElectImpliesNoLogsInTerm*, highlighting the compositional substructure of this graph.

Another key feature made apparent in this graph is where so-called *message induction cycles* occur. That is, cases where certain lemmas must hold both on a local server and also when its state is sent into the network via a message. For example, *Safety* and *Inv0* form one of these cycles, where *Inv0* states a similar property to *PrimaryOwnsEntries* but refers to the log state in an *AppendEntries* message rather than the state of a local server. Similarly, *CandidateElectImpliesNoLogsInTerm* and its support lemma *Inv12* form such a cycle. These local cycles form due to the message-passing nature of the protocol, and we note that these patterns bear similarities to recent observations that inductive invariant lemmas for distributed protocols often fall into a standard taxonomy [131], and some invariants can be automatically derived from others. More details on the lemmas appearing in this graph can also be found in Appendix B.

In general, we found the proof graph structure a significant aid to developing this inductive proof with both automation and guidance. During development, we encountered several cases where our synthesis algorithm was able to discharge significant portions of the graph but failed on key local nodes. We found a highly localized manner of inspection and grammar repair to be very effective in ultimately guiding the tool towards full convergence. Without the automation and interpretability features of our technique, we do not think it would have been possible to efficiently easily get a proof of this scale to go through at this level of automation.

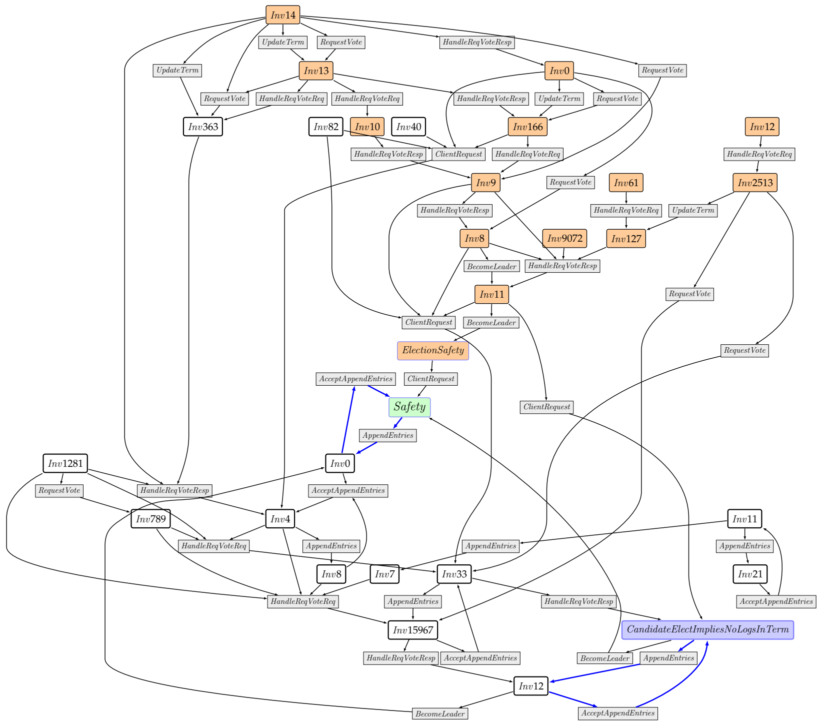


Figure 5.9: *AsyncRaft* inductive proof graph for *PrimaryOwnsEntries* safety property (green *Safety* node). Key lemmas shown in blue, support subgraph for *ElectionSafety* in orange, and induction cycles in blue.

5.6 Related Work

Automated Inductive Invariant Inference There are several recently published techniques that attempt to solve the problem of fully automated inductive invariant inference for distributed protocols, including IC3PO [38], SWISS [43] DuoAI [33, 44], and others [45]. These tools, however, provide little feedback when they fail on a given problem, and the large scale protocols we presented in this chapter are of a complexity considerably higher than what existing modern tools in this area can solve.

In the related domain of general model checking algorithms, we believe that our inductive proof graph structure for managing large scale inductive invariants bears similarities with approaches developed for managing proof obligation queues in IC3/PDR [132, 133]. In some sense, our approach revolves around making the set of proof obligations and their dependencies explicit (and also incorporating action-based decomposition), which can be a key factor in tuning of IC3/PDR, which often has many non-deterministic choices throughout execution.

More broadly, there exist many prior techniques for the automatic generation of program and protocol invariants that rely on data driven or grammar based approaches. Houdini [118] and Daikon [119] both use enumerative checking approaches to discover program invariants. FreqHorn [104] tries to discover quantified program invariants about arrays using an enumerative approach that discovers invariants in stages and also makes use of the program syntax. Other techniques have also tried to make invariant discovery more efficient by using improved search strategies based on MCMC sampling [120].

Interactive and Compositional Verification There is other prior work that attempts to employ compositional and interactive techniques for safety verification of distributed protocols, but these typically did not focus on presenting a fully automated and interpretable inference technique. For example, the Ivy system [39] and additional related work on exploiting modularity for decidability [134].

In the Ivy system [39] one main focus is on the modeling language, with a goal of making it easy to represent systems in a decidable fragment of first order logic, so as to ensure verification conditions always provide some concrete feedback in the form of counterexamples. They also discuss an interactive approach for generalization from counterexamples, that has similarities to the UPDR approach used in extensions of IC3/PDR [42]. In contrast, our work is primarily focused on different concerns e.g., we focus on compositionality as a means to provide an efficient and scalable automated inference technique, and as a means to produce a more interpretable proof artifact, in addition to allowing for localized counterexample reasoning and slicing. They also do not present a fully automated inference technique, as we do. Additionally, we view decidable modeling as an orthogonal component of the verification process that could be complementary to our approach.

More generally, compositional verification has a long history and has been employed as a key technique for addressing complexity of large scale systems verification. For example, previous work has tried to decompose proofs into decidable sub-problems [134]. The notion of learning assumptions for compositional assume-guarantee reasoning has also been explored thoroughly and bears similarities to our approach of learning support lemmas while working backwards from a target proof goal [135]. Compositional model checking techniques have also been explored in various other domains [136, 137].

Concurrent Program Analysis Our techniques presented in this chapter bear similarities to prior approaches used in the analysis and proofs of concurrent programs. Our notion of inductive proof graphs is similar to the *inductive data flow graph* concept presented in [138, 139]. That work, however, is focused specifically on the verification of multi-process concurrent programs, and d id not generalize the notions to a distributed setting. Our procedures for inductive invariant inference and our slicing optimizations are also novel to our approach. Our proof graphs also bear similarities to the *verification diagrams* concept presented in [121], as well as the *program unwinding* concepts of [140].

Our slicing techniques are similar to cone-of-influence reductions [141], as well as other *program slicing* techniques [142]. It also shares some concepts with other path-based program analysis techniques that incorporate slicing techniques [143, 144]. In our case, however, we apply it at the level of a single protocol action and target lemma, particularly for the purpose of accelerating syntax-guided invariant synthesis tasks.

5.7 Conclusions and Future Work

In this chapter, we presented *inductive proof decomposition*, a new technique for inductive invariant development of large scale distributed protocols. Our technique both improves on the scalability of existing approaches by building an inference routine around the inductive proof graph, and this structure also makes the approach amenable to interpretability and failure diagnosis. In future, we are interested in exploring new approaches and further optimizations enabled by our technique and compositional proof structure. For example, we would be interested in seeing how the compositional structure of the inductive proof graph can be used to further tune and optimize local inference tasks e.g. by taking advantage of more local properties that can accelerate inference, like specialized quantifier prefix templates, action-specific grammars, etc. We would also like to explore and understand the empirical structure of these proof graphs on a wider range of larger and more complex real world protocols, and to understand the structure of inductive proof graphs with respect to protocol refinement.

Chapter 6

Conclusions and Future Work

In this dissertation we have studied the problem of formal safety verification of real world distributed protocols, both by manual, interactive, and automated approaches. Our work centered around the development and automated inference of *inductive invariants*, which serve as the core, formal proof certificate required to prove safety of these protocols in general. Furthermore, we have focused on improving the interpretability of these methods to allow effective human guidance and interaction when needed.

Chapter 3 discussed our work on the design and formal verification of a new dynamic reconfiguration protocol that decouples reconfiguation from the main operation log, enabling a *logless* design. We presented a formally stated inductive invariant for establishing the high level safety property of this protocol, along with a formal TLAPS proof for proving its correctness. Our work constituted the first formal verification effort for a Raft-based reconfiguration protocol.

Chapter 4 presented a novel technique for automatically inferring inductive invariants for distributed protocols specified in TLA+. We presented an algorithm that combines lemma invariant generation with a counterexample-guided selection heuristic. The chapter details the algorithm's design, implementation, and evaluation on a diverse set of protocol benchmarks. Our results demonstrate the effectiveness of this approach in automating a crucial step of formal verification for distributed systems, reducing the manual effort required in proving protocol correctness.

Chapter 5 presented an additional technique for developing inductive invariants that bridges the automation of modern model checking and inference approaches with the interactivity and intepretability needed for human guided, interactive proofs. Our technique, *inductive proof decomposition*, takes advantage of the underlying compositional structure of an inductive invariant to aid in its development, both for automation and human guidance, making the inductive proof artifacts interpretable by human verifiers.

6.1 Future Work

Although significant progress towards automated and interpretable protocol verification has been made in the work presented in this dissertation, we view several possible directions for future work.

Compositional Verification of Reconfiguration Our formal specification of reconfiguration is described in a partially compositional structure, where the reconfiguration sub-protocol is loosely coupled with the main log replication protocol. Both protocols share and interact via some behaviors (e.g. leader election), but also contain portions of logic independent to each subprotocol. We partially take advantage of this for model checking modularity (e.g. as discussed in Appendix A.3), but ultimately we are limited to checking only the reconfiguration protocol, *MongoLoglessDynamicRaft*, in isolation. We are interested in the possibility of applying additional compositional verification techniques to reconfiguration protocols that can be decomposed like ours, perhaps using techniques like *interaction preserving abstraction* [145]. We believe that, in general, reconfiguration related sub-protocols may be amenable to abstraction within the larger context of an overall replication protocol, potentially allowing for significant model checking performance gains.

Automated Synthesis of Reconfigurable Protocol Variants Our design efforts for a dynamic reconfiguration protocol as discussed in Chapter 3 were considerable, and required a large amount of tedious design and verification effort. The original reconfiguration protocol designed for Raft and its subsequently discovered safety bug [9] is an illustrative example of the difficulty of designing these types of protocol extensions. It is compelling to consider whether these types of "reconfigurable" protocol extensions could benefit from automated program synthesis or repair techniques [146–148]. That is, exploring whether a reconfigurable consensus protocol variant be automatically synthesized from a non-reconfigurable one. Similar ideas have also been explored in work that attempts to automatically synthesize fault tolerant versions of existing protocols [149, 150], and also in the protocol synthesis community [151, 152].

Additional Compositional Features of Inductive Proof Graphs There are several potential avenues for extending the compositional features of inductive proof graphs outlined in Chapter 5. First, we could explore decomposing the graphs based on the quantifier structure at local subgraphs or also use more fine-grained, action-specific grammars to generate more targeted lemma candidates. Finally, we could experiment with dynamically weighting predicates in the grammar according to particular graph nodes. These extensions could potentially improve both the efficiency and interpretability of our techniques.

We are also interested to pursue a deeper and more thorough understanding of both the theoretical characteristics of these proof graphs and their empirical characteristics for a broader range of protocols. This may also open avenues related to how to repair these proof graphs efficiently when a breaking protocol change is made, for instance. Or, for example, how these graphs could be used to understand a broken protocol that is "almost" correct, modulo small repairs.

In addition, our formulation of inductive proof graphs does not explicitly take into account parameters of a distributed protocol in a formal way. We would be interested to explore whether incorporating this might shed further insight into the structure of protocol correctness proof. For example, a given lemma invariant may be inductive at certain parameter bounds but not others, so, in theory, we could extend our notion of action nodes to also be decomposed by parameter instance sizes.

Native Symbolic Reasoning in TLC From a tooling perspective, we found the TLC explicit state model checker [82] very effective for exhaustive reachability analyses when building various implementations of our verification and inference techniques, but we found it lacking for symbolic verification tasks like inductive proof obligation checking. Tools like TLAPS [70] and the more recently developed Apalache symbolic model checker [109] partially fill this tooling gap, but each come with their own limitations and tradeoffs that we found diffcult to work with in practice. We are interested in whether TLC could be expanded in future to have smarter heuristics for evaluating what are essentially satisfiability queries (e.g. perhaps modeled after classical SAT/SMT techniques [28, 153]). The KodKod constraint solver [154] and Alloy [155] are similar tools in this type of domain, and ideas from the constraint satisfaction (CSP) community [156] may also be applicable to enhance TLC in this regard.

Appendix A

Detailed MongoRaftReconfig Safety Proof

A.1 Detailed Safety Proof of *MongoRaftReconfig*

In this section we provide the detailed proof of Theorem 2, the *LeaderCompleteness* safety property of *MongoRaftReconfig*. We first provide some preliminary definitions in Section A.1.1 that are used throughout the proof. Section A.1.2 covers the proof of the *ElectionSafety* property, Section A.1.3 proves some auxiliary properties about logs of servers in the system, and Section A.1.4 presents the proof of Theorem 2, which relies on the lemmas established in the preceding sections.

Along with the proof, we provide a complete pseudocode description of *Mongo-RaftReconfig* in Algorithm 7, which is necessary to fully understand and verify the proof. Algorithm 7 includes the reconfiguration specific behaviors shown in Algorithm 2 along with the behaviors of *MongoStaticRaft*, which are mostly orthogonal to reconfiguration, but are necessary to describe for completeness.

A.1.1 Definitions and Notation

Recall that, in our system model, we consider a set of processes $Server = \{s_1, s_2, ..., s_n\}$ that communicate by sending messages. Also, recall that a *configuration* is defined as a tuple (m, v, t), where $m \in 2^{Server}$ is a member set, $v \in \mathbb{N}$ is a numeric configuration *version*, and $t \in \mathbb{N}$ is the numeric *term* of the configuration. We refer to the elements of a configuration tuple $C = (m, v, t)$ as, respectively, $C.m$, $C.v$ and $C.t$, and, for a configuration C, we informally refer to the elements of $Quorums(C.m)$ as the "quorums of C". For a sequence of elements L, we use $L[i]$ to refer to the i-th element of L (1-indexed), and $L[..i]$ to refer to the sequence containing the first i elements of L. We denote the concatenation of two sequences, L and M, as $L \circ M$, a concrete sequence of values as $\langle v_1, v_2, \ldots, v_n \rangle$, and the empty sequence as $\langle\rangle$. We also use the notation $1..n$ to refer to the set of natural numbers $\{1, 2, \ldots, n\}$.

When referring to state variables of the protocol in inductive proof arguments below, we follow a convention of referring to the value of a state variable X in the current state as X and its value in the next state as X' (i.e. its value after some state transition). When referring to log entries on a server, we sometimes use the pair notation $(index, term)$ to refer to a log entry at position $index$ with a term of $term$. For example, if we say that the log of server s contains entry $(index, term)$, this means that $Len(log[s]) \geq index$ and $log[s][index] = term$. Below we provide a few basic definitions utilized in the proof and in the pseudocode description given in Algorithm 7. Additional notation and definitions are also given in Algorithm 6 and relied upon throughout the proof.

Algorithm 6 Definitions used in *MongoRaftReconfig* pseudocode (Algorithm 7).

```
$Seq(S) \triangleq$ the set of all sequences with elements from the set $S$
$Len(s) \triangleq$ the length of a sequence $s$
$InLog(ind, t, s) \triangleq \exists k \in 1..Len(log[s]) : (k = ind \land log[s][k] = t)$
$IsPrefix(l_i, l_j) \triangleq Len(l_i) \leq Len(l_j) \land l_i = l_j[..Len(l_i)]$

$LogTerm(i) \triangleq$ if $log[i] = \langle\rangle$ then $-1$ else $log[i][Len(log[i])]$
$LogGeq(i,j) \triangleq (LogTerm(i) > LogTerm(j)) \lor (LogTerm(i) = LogTerm(j) \land Len(log[i]) \geq Len(log[j]))$
$LogCheck(i,j) \triangleq (Len(log[j]) > Len(log[i])) \land (log[i] = \langle\rangle \lor (log[i][Len(log[i])] = log[j][Len(log[i])]))$
$CanRollback(i,j) \triangleq (LogTerm(i) < LogTerm(j)) \land \neg IsPrefix(log[i], log[j])$
$IsCommitted(ind, t, Q) \triangleq \forall j \in Q : InLog(ind, t, j) \land term[j] = t$
$CommittedAt(t) \triangleq \{(index, term) \in committed : term = t\}$

$QuorumsAt(i) \triangleq Quorums(config[i])$
$C_{(i)} \triangleq (config[i], configVersion[i], configTerm[i])$

$Q1(i) \triangleq \exists Q \in QuorumsAt(i) : \forall j \in Q : (C_{(j)}.v, C_{(j)}.t) = (C_{(i)}.v, C_{(i)}.t)$      ▷ Config Quorum Check
$Q2(i) \triangleq \exists Q \in QuorumsAt(i) : \forall j \in Q : term[j] = term[i]$      ▷ Term Quorum Check

$P1a(i) \triangleq (committed = \emptyset) \lor CommittedAt(term[i]) \neq \emptyset$
$P1b(i, Q) \triangleq \forall c \in CommittedAt(term[i]) : IsCommitted(c[1], term[i], Q)$
$P1(i) \triangleq \exists Q \in QuorumsAt(i) : P1a(i) \land P1b(i, Q)$      ▷ Oplog Commitment
```

Definition 9 (Config Ordering). *For configurations C_i and C_j, we define the following*

$$C_i < C_j \triangleq (C_i.t < C_j.t) \lor (C_i.t = C_j.t \land C_i.v < C_j.v)$$
$$C_i > C_j \triangleq C_j < C_i$$
$$C_i \leq C_j \triangleq C_i < C_j \lor ((C_i.v, C_i.t) = (C_j.v, C_j.t))$$
$$C_i \geq C_j \triangleq C_j \leq C_i$$

Definition 10 (Deactivated Config). *A configuration C is deactivated if, for all $Q \in Quorums(C.m)$, there exists some server $n \in Q$ such that $C_n > C$, where C_n is the configuration of server n.*

$$Deactivated(C) \triangleq \forall Q \in Quorums(C.m) : \exists n \in Q : C_{(n)} > C$$

Definition 11 (Active Config). *A configuration C is active if it is not deactivated.*

$$Active(C) \triangleq \neg Deactivated(C)$$

Algorithm 7 Complete pseudocode description of *MongoRaftReconfig* behavior. State, actions, and behavior specific to *MongoStaticRaft* is highlighted in blue.

```
1: State and Initialization
2:
3: Let $m_{init} \in 2^{Server} \setminus \emptyset$
4: $\forall i \in Server$ :
5:   $term[i] \in \mathbb{N}$, initially 0
6:   $state[i] \in \{Pri., Sec.\}$, initially $Secondary$
7:   $config[i] \in 2^{Server}$, init $m_{init}$
8:   $configVersion[i] \in \mathbb{N}$, initially 1
9:   $configTerm[i] \in \mathbb{N}$, initially 0
10:  $log[i] \in Seq(\mathbb{N})$, initially $\langle\rangle$
11:  $committed \subseteq \mathbb{N} \times \mathbb{N}$, initially $\emptyset$
12:
13: Actions
14:
15: action: Reconfig($i$, $m_{new}$)
16:     require $m_{new} \in 2^{Server}$
17:     require $state[i] = Primary$
18:     require $Q1(i) \wedge Q2(i) \wedge P1(i)$
19:     require $QuorumsOverlap(config[i], m_{new})$
20:     $config[i] \leftarrow m_{new}$
21:     $configVersion[i] \leftarrow configVersion[i] + 1$
22: end action:
23:
24: action: SendConfig($i, j$)
25:     require $state[j] = Secondary$
26:     require $C_{(i)} > C_{(j)}$
27:     $C_{(j)} \leftarrow C_{(i)}$
28: end action:
29:
30: action: BecomeLeader($i$, $Q$)
31:     require $Q \in Quorums(config[i])$
32:     require $i \in Q$
33:     require $\forall v \in Q : C_{(i)} \geq C_{(v)}$
34:     require $\forall v \in Q : term[i] + 1 > term[v]$
35:     require $\forall v \in Q : LogGeq(i, v)$
36:     $state[i] \leftarrow Primary$
37:     $state[j] \leftarrow Secondary, \forall j \in (Q \setminus \{i\})$
38:     $term[j] \leftarrow term[i] + 1, \forall j \in Q$
39:     $configTerm[i] \leftarrow term[i] + 1$
40: end action:
41:
42: action: UpdateTerms($i, j$)
43:     require $term[i] > term[j]$
44:     $state[j] \leftarrow Secondary$
45:     $term[j] \leftarrow term[i]$
46: end action:
47:
48: action: ClientRequest($i$)
49:     require $state[i] = Primary$
50:     $log[i] \leftarrow log[i] \circ \langle term[i] \rangle$
51: end action:
52:
53: action: GetEntries($i, j$)
54:     require $state[i] = Secondary$
55:     require $LogCheck(i, j)$
56:     $log[i] \leftarrow log[i] \circ \langle log[j][Len(log[i]) + 1] \rangle$
57: end action:
58:
59: action: RollbackEntries($i, j$)
60:     require $state[i] = Secondary$
61:     require $CanRollback(i, j)$
62:     $log[i] \leftarrow log[i][..(Len(log[i]) - 1)]$
63: end action:
64:
65: action: CommitEntry($i$, $Q$)
66:     require $Q \in Quorums(config[i])$
67:     require $state[i] = Primary$
68:     require $IsCommitted(Len(log[i]), term[i], Q)$
69:     $committed \leftarrow committed \cup \{(Len(log[i]), term[i])\}$
70: end action:
71:
```

Definition 12 (Active Config Set). *The active config set is the set of servers with a configuration that is active.*

$$\mathit{ActiveConfigSet} \triangleq \{s \in \mathit{Server} : \mathit{Active}(C_{(s)})\}$$

A.1.2 Election Safety

In this section we present the proof of Lemma 1 along with auxiliary invariants required for the proof. We prove it inductively, by assuming that Lemmas 15, 16, 17, 18, 19, and 1 of this section act as strengthening assumptions for the inductive hypothesis needed to prove Lemma 1. That is, for the proof of each lemma L in this group, we show that L holds in the initial protocol states, and then show that, if all lemmas of this group hold in the current state, then L holds in the next state, for any possible protocol transition. Lemmas 12, 13, and 14 are some additional, helpful facts that we establish first, and are useful for proving the lemmas in this and later sections.

Lemma 12 (Deactivated configs cannot reconfig or elect primary). *If server i is in a deactivated configuration, C_i, then it cannot execute a $\mathit{Reconfig}(i)$ or $\mathit{BecomeLeader}(i)$ action.*

Proof. We must consider *Reconfig* and *BecomeLeader* actions.

- $\mathit{Reconfig}(i)$ requires the *Config Quorum Check* (Algorithm 7, Line 18) to be satisfied for C_i, the current configuration of primary server i. This requires that, for some quorum $Q \in \mathit{Quorums}(C_i.m)$, all servers in Q are in configuration C_i. If C_i is deactivated, though, all quorums of C_i contain some server in configuration $> C_i$, violating this precondition.

- $\mathit{BecomeLeader}(i)$ elects a primary server i in configuration C_i. It requires that a quorum of servers in C_i have configurations $\leq C_i$ (Algorithm 7, Line 33). If C_i is deactivated, though, all quorums of C_i contain some server in configuration $> C_i$, violating this precondition.

□

Lemma 13 (Configs increase monotonically). *If Lemma 19 holds in the current state, then for all $s \in \mathit{Server}$, if C_s is the configuration of server s in the current state and C'_s is the configuration of s after any state transition, then $C'_s \geq C_s$.*

Proof. We must consider any actions that modify server configurations: *Reconfig*, *Become-Leader*, *SendConfig*.

- $\mathit{Reconfig}(i)$ updates the configuration on a primary server from C_i to C'_i, where $(C_i.m, C_i.t) = (C'_i.m, C'_i.t)$ and $C'_i.v > C_i.v$. So, monotonicity is upheld, by the definition of configuration ordering (Definition 9).

- $BecomeLeader(i)$ elects a primary server i in $term'[i]$ and updates the configuration of i from C_i to C'_i, where $(C_i.m, C_i.v) = (C'_i.m, C'_i.v)$ and $C'_i.t = term'[i]$. So, it is sufficient to show that $term'[i] \geq C_i.t$. Assume this were not the case i.e. that $C_i.t > term'[i]$. Since C_i must be active in order for the $BecomeLeader(i)$ action to occur, by Lemma 12, this would imply that all quorums of C_i contain some server in term $\geq C_i.t$, by Lemma 19. This would prevent the $BecomeLeader(i)$ action from electing a primary in $term'[i]$, though, since $C_i.t > term'[i]$. So, it must be that $term'[i] \geq C_i.t$, implying that $C'_i.t \geq C_i.t$.

- $SendConfig(i, j)$ updates a configuration on server j to the configuration of server i. It follows directly from the precondition of *SendConfig* (Algorithm 7, Line 26) that $C_{(i)} > C_{(j)}$, so monotonicity is upheld, by the definition of configuration ordering (Definition 9).

□

Lemma 14 (Config deactivation stability). *If Lemma 13 holds in the current state, then if a configuration C is deactivated in the current state, C cannot be active in the next state.*

Proof. A configuration C is deactivated if, for all $Q \in Quorums(C.m)$, there exists some server $n \in Q$ such that $C_n > C$, where C_n is the configuration of server n. Since configurations increase monotonically on servers (Lemma 13), if $C_n > C$ holds currently for some server n, it must hold in the next state. □

Lemma 15 (Primary term equals config term). *For all $i \in Server$, if i is currently primary in $term[i]$ in configuration C, then $C.t = term[i]$.*

Proof. In all initial states, no server is primary, so it holds. The only actions that could falsify the lemma are those that change the term of a server's local configuration, its current term, or its primary status: *SendConfig*, *UpdateTerms*, *BecomeLeader*.

- The *SendConfig* action can only update the configuration of a server that is not currently primary, so such a transition could not falsify this lemma in the next state.

- *UpdateTerms* cannot change the term of a primary server, so it upholds the lemma.

- $BecomeLeader(i)$ elects server i as primary in $term'[i]$, and it sets $configTerm[i] \leftarrow term'[i]$, so it upholds the lemma.

□

Lemma 16 (Config version and term unique). *For all servers i and j, in configurations C_i and C_j, respectively, if $(C_i.v, C_i.t) = (C_j.v, C_j.t)$ then $C_i.m = C_j.m$.*

$$\forall i, j \in Server :$$
$$((configVersion[i], configTerm[i]) = (configVersion[j], configTerm[j])) \Rightarrow$$
$$(config[i] = config[j])$$

Proof. In all initial states, every server configuration is identical, so the lemma holds. The only actions that could falsify Lemma 16 in the next state are those that modify configurations on a server: *BecomeLeader*, *SendConfig*, and *Reconfig*.

- A $BecomeLeader(i)$ action elects a server i as primary in $term'[i]$ and updates its configuration from C_i to C'_i, where $(C'_i.m, C'_i.v) = (C_i.m, C_i.v)$ and $C'_i.t > C_i.t$ (by Lemma 13). Since this action only modifies the configuration of server i, the only way Lemma 16 could be falsified in the next state is if, in the current state, there was a server $j \neq i$ in configuration C_j such that $(C_j.v, C_j.t) = (C'_i.v, C'_i.t)$ and $C_j.m \neq C'_i.m$. If $C_j.t = C'_i.t$, though, this implies, by Lemma 19, that all quorums of active configurations in the current state must contain some server in term $\geq C'_i.t$. If this were the case, though, the $BecomeLeader(i)$ action could not have occurred to elect i as primary in term $C'_i.t$, since C_i must be active, by Lemma 12, and the voting precondition on terms (Algorithm 7, Line 34) would have prevented it.

- $SendConfig(i, j)$ updates the configuration on server j to that of server i, and doesn't modify the state of any other server. So, the set of unique configurations in the system can only be reduced or left the same by this action. So, if Lemma 16 held currently, it must hold in the next state.

- $Reconfig(i)$ updates the configuration on a primary server i from C_i to C'_i, where $C'_i.t = term[i]$ and $C'_i.v > C_i.v$. By Lemma 1, we know that there is a unique primary per term in the current state, and, since $Reconfig(i)$ doesn't modify the primary status or term of any server, there must still be a unique primary per term in the next state. So, server i is the only primary in $term[i]$, and by Lemma 17, i contains the newest configuration in $term[i]$. So, the version of i's new configuration, $C'_i.v$, will be greater than all other configurations in $term[i]$, implying that the configuration C'_i will be unique among all existing configurations, upholding the lemma.

□

Lemma 17 (Primary contains newest config of term)**.** *For all $i \in Server$, if i is currently a primary in $term[i]$, then it contains the newest configuration in $term[i]$.*

$$\begin{aligned}&\forall i, j \in Server :\\ &\quad (state[i] = Primary \land configTerm[j] = term[i]) \Rightarrow\\ &\quad (configVersion[j] \leq configVersion[i])\end{aligned}$$

Proof. The only actions that could falsify this lemma are those that affect the primary status of a server or modify configurations or terms on a server: *BecomeLeader*, *Reconfig*, *SendConfig*, and *UpdateTerms*.

- $BecomeLeader(i)$ elects a primary server i in $term'[i]$. It updates the configuration of server i from configuration C_i to C'_i, where $C'_i.t = term'[i]$ and $C_i.v = C'_i.v$. Since this action doesn't modify the state of any other server, in order for Lemma 17 to be falsified in the next state, it must be that, in the current state, there exists some server $j \neq i$, with configuration C_j, such that $C_j.t = term'[i]$ and $C_j.v > C'_i.v$. That is, server j contains a configuration in the term of primary i after its election but j's configuration is newer than i's. If $C_j.t = term'[i]$ in the current state, though, this implies, by the assumption of Lemma 19, that the quorums of all active configurations contain some server in term $\geq C_j.t$. If this were the case, then the $BecomeLeader(i)$ action could not have occurred to elect i as primary in $term'[i]$, since C_i must be active in the current state, by Lemma 12, and the voting precondition on terms (Algorithm 7, Line 34) would have prevented it.

- $Reconfig(i)$ updates the configuration of a primary server i from C_i to C'_i, where $C'_i.v > C_i.v$ and $C'_i.t = C_i.t$. By Lemma 1, we know that there is a unique primary in a given term, and since $Reconfig(i)$ doesn't modify the terms or primary status of any server, this will continue to hold in the next state. And, by assumption of Lemma 17 in the current state, a primary has the newest configuration in its term. So, the new configuration created by i must be the newest configuration in $term[i]$, and it will be contained on server i, the unique primary of $term[i]$.

- $SendConfig(i, j)$ updates the configuration on a secondary server j to that of server i, and doesn't modify the state of any other server. So, no primary configuration is modified and the set of unique configurations in the system can only be reduced or left the same by this action, so if Lemma 17 held currently, it must hold in the next state.

- *UpdateTerms* only updates server terms, and if it updates the term of a server s it sets $state[s] \leftarrow Secondary$, so it could not falsify the lemma.

□

Lemma 18 (Active configs overlap). *All quorums of any two active configurations overlap.*

$$\forall s, t \in ActiveConfigSet : QuorumsOverlap(config[s], config[t])$$

Proof. In all initial states, there is only a single, unique configuration, so the lemma holds. If Lemma 18 holds in the current state, then the only actions that could possibly falsify it in the next state are those that affect server configurations: *BecomeLeader*, *SendConfig*, and *Reconfig*. Let $Active$ refer to the set of active configurations that exist in the current state, and $Active'$ to the set of active configurations that exist in the next state.

- $BecomeLeader(i)$ updates the configuration of a server i from C_i to C'_i, where $(C'_i.m, C'_i.v) = (C_i.m, C_i.v)$. This does not change the member set of any existing configuration, and, due to Lemma 14, cannot create any new active configuration other than C_i', so if the quorums of all configurations in *Active* overlapped, all of those in $Active'$ should still overlap.

- The $SendConfig(i, j)$ action updates the configuration of server j to C_i, the configuration of server i. This action does not create any new configurations, so the only way it could falsify Lemma 18 in the next state is if it made C_i active in the next state, and the quorums of C_i did not overlap with some other, active configuration. By Lemma 14, though, C_i cannot become active in the next state if it was not already active, so the lemma must hold.

- $Reconfig(i)$ updates the configuration of a primary server i from C_i to C_i'. To falsify the lemma in the next state, it must be that $C_i' \in Active'$ and there exists a server j, in configuration C_j, such that $C_j \in Active'$ and $\neg QuorumsOverlap(C_i'.m, C_j.m)$. We know, by the precondition enforced by the $Reconfig(i)$ action (Algorithm 7, Line 19), that $QuorumsOverlap(C_i'.m, C_i.m)$, so it must be that $C_j.m \neq C_i.m$, implying, due to Lemma 16, that $(C_j.v, C_j.t) \neq (C_i.v, C_i.t)$. In addition, we know that C_i must have been active in order for the $Reconfig(i)$ to occur (Lemma 12). So, there are now two cases to consider:

  – $C_j > C_i$.
    We know that $C_i.t = C_i'.t$, by the definition of the *Reconfig* action. It must also be the case that $C_j.t \neq C_i.t$. Otherwise, it would imply that $C_j.t = C_i.t \wedge C_j.v > C_i.v$, which cannot hold since there is a unique primary per term (Lemma 1) and a primary contains the newest configuration of its term (Lemma 17). So, it must be that $C_j.t > C_i.t$. If C_j exists, though, by Lemma 19, all quorums of C_i must contain some server in term $\geq C_j.t$, since we know that C_i is active. But, since $C_j.t > C_i.t$, this would imply the *Term Quorum Check* precondition, $Q2(i)$ (Algorithm 7, Line 18) could not have been satisfied in the current state, preventing the $Reconfig(i)$ action from occurring.

  – $C_j < C_i$
    By the assumption of Lemma 18 in the current state, we know that all quorums of C_j and C_i overlap, since both configurations are active. In order for the $Reconfig(i)$ action to occur, though, there must have been some quorum $Q \in Quorums(C_i.m)$ such that, for all servers $n \in Q$, in configuration C_n, $(C_n.v, C_n.t) = (C_i.v, C_i.t)$. This is ensured by $Q1(i)$, the *Config Quorum Check* precondition (Algorithm 7, Line 18). If $C_i > C_j$, though, and $QuorumsOverlap(C_i.m, C_j.m)$, this would imply that all quorums of C_j contain some server $m \in Q$, in configuration C_m. Since we know that $(C_m.v, C_m.t) = (C_i.v, C_i.t)$ and $C_i > C_j$, this implies that $C_j \notin Active$, which additionally implies that $C_j \notin Active'$, by Lemma 14, contradicting our as-

sumption that $C_j \in \mathit{Active}'$.

□

Lemma 19 (Active configs safe from past terms). *For any existing configurations C_a and C, where C_a is active, all quorums of C_a contain some server in term $\geq C.t$.*

$$\begin{aligned}
&\forall s \in \mathit{Server} : \\
&\forall t \in \mathit{ActiveConfigSet} : \\
&\forall Q \in \mathit{Quorums}(\mathit{config}[t]) : \exists n \in Q : \mathit{term}[n] \geq \mathit{configTerm}[s]
\end{aligned}$$

Proof. In all initial states, all servers have the same, unique configuration, and all servers have the same terms, so the lemma holds. We must then consider actions that change configurations or terms on servers: *BecomeLeader*, *Reconfig*, *SendConfig*, and *UpdateTerms*. Again, in the below arguments we refer to Active as the set of active configurations in the current state, and Active' as the set of active configurations in the next state.

- $\mathit{BecomeLeader}(i)$ updates the configuration on server i from C_i to C_i', where $(C_i'.m, C_i'.v) = (C_i.m, C_i.v)$ and $C_i'.t > C_i.t$ (by Lemma 13). We also know, by Lemma 12, that C_i must be active in order for the $\mathit{BecomeLeader}(i)$ action to occur. Lemma 19 could be falsified in two cases, which we examine below:
  - If C_i' is active, then we must show that, for any other server j, with configuration C_j, all quorums of C_i' contain some server in term $\geq C_j.t$. By the assumption of Lemma 19 in the current state, we know that all quorums of configuration C_i contain some server in term $\geq C_j.t$. Since $C_i'.m = C_i.m$, this should also hold for C_i'.
  - If C_j is the configuration of some server j such that $C_j \in \mathit{Active}'$, we must show that all quorums of C_j intersect with some server in term $\geq C_i'.t$. By Lemma 14, we know that if $C_j \in \mathit{Active}'$ then $C_j \in \mathit{Active}$. So, by the assumption of Lemma 19 in the current state, we know that all quorums of C_j intersect with some server in term $\geq C_i.t$. After the $\mathit{BecomeLeader}(i)$ action occurs, due to its postcondition (Algorithm 7, Line 38), there must be some quorum $Q \in \mathit{Quorums}(C_i.m)$ such that $\mathit{term}'[n] = C_i'.t$ for all $n \in Q$, since $C_i.m = C_i'.m$. Since C_j and C_i are both active in the current state, it must be that $\mathit{QuorumsOverlap}(C_i.m, C_j.m)$, by Lemma 18. So, all quorums of C_j must contain some server $m \in Q$, where $\mathit{term}[m] \geq C_i'.t$, upholding Lemma 19 in the next state.

- $\mathit{Reconfig}(i)$ updates the configuration of server i from C_i to C_i', where $C_i.t = C_i'.t$. We know that C_i must have been active in order for the $\mathit{Reconfig}(i)$ to occur, by Lemma 12. We consider the two cases in which this action could falsify the lemma:

  - If C_i' is active, then we must show that, for any other server j, in configuration C_j, all quorums of C_i' overlap with some server in term $\geq C_j.t$. By the assumption of Lemma 19, we know that, in the current state, all quorums of C_i contain some server in term $\geq C_j.t$, since $C_i \in \mathit{Active}$. In order for the $\mathit{Reconfig}(i)$ action to have occurred, the *Term Quorum Check* precondition, $Q2(i)$ (Algorithm 7, Line 18) must have been satisfied, meaning that there exists some quorum $Q \in \mathit{Quorums}(C_i.m)$ such that, for all $n \in Q$, $\mathit{term}[n] = C_i.t$. If all quorums of C_i currently contain some server in term $\geq C_j.t$ and $Q2(i)$ was satisfied, though, then this must imply that there is some $m \in Q$ such that

$$\mathit{term}[m] = C_i.t \land \mathit{term}[m] \geq C_j.t$$

    implying that $C_i.t \geq C_j.t$. So, since $\mathit{QuorumsOverlap}(C_i.m, C_i'.m)$, we know that all quorums of C_i' will contain some server v such that $\mathit{term}[v] \geq C_j.t$, ensuring Lemma 19 is upheld.
  - If C_j is the configuration of some server j such that $C_j \in \mathit{Active}'$, we must show that all quorums of C_j contain some server in term $\geq C_i'.t$. By assumption of Lemma 19 in the current state, all quorums of C_j contain some server in term $\geq C_i.t$, since C_i is active. Since $C_i.t = C_i'.t$ and $\mathit{Reconfig}(i)$ doesn't modify the terms of any servers, Lemma 19 must be upheld.

- $\mathit{SendConfig}(i,j)$ updates the configuration of server j to C_i, the configuration of server i, and does not modify the terms of any servers. The set of unique configurations in the system can only be reduced or left the same by this action. So, it does not create any new active configurations (by Lemma 14) and therefore cannot falsify Lemma 19 in the next state.

- *UpdateTerms* does not modify configurations and can only increase the term of a server, so it must uphold the property.

$\square$

Proof of Lemma 1. In all initial states, there are no primary servers, so the lemma holds. The only possible actions that could falsify Lemma 1 in the next state are $\mathit{BecomeLeader}$ or $\mathit{UpdateTerms}$ actions.

- $\mathit{BecomeLeader}(i)$ elects a primary server i in configuration C_i in $\mathit{term}'[i]$, and does not modify the state of any other server. Assume there exists another server $j \neq i$, in configuration C_j, that is also primary in $\mathit{term}'[i]$ in the current state. If server j is currently primary in $\mathit{term}'[i]$, this implies, by Lemma 15, that $C_j.t = \mathit{term}'[i]$. But, in order for $\mathit{BecomeLeader}(i)$ to occur, configuration C_i must be active in the current state, by Lemma 12. So, if C_i is active, Lemma 19 implies that all quorums of C_i must contain some server in a term $\geq C_j.t = \mathit{term}'[i]$. This would, however, prevent the election of i in $\mathit{term}'[i]$ due to the voting precondition of $\mathit{BecomeLeader}(i)$ (Algorithm 7, Line 34) that requires some quorum of servers in C_i to have terms $< \mathit{term}'[i]$.

- An $\mathit{UpdateTerms}(i, j)$ action only changes the term of server j, and sets $\mathit{state}[j] \leftarrow \mathit{Secondary}$, so it cannot falsify the lemma if it held in the current state.

□

A.1.3 Log Properties

In this section we establish several auxiliary lemmas related to properties of logs in the system. These lemmas are, for the most part, conceptually unrelated to reconfiguration, but are required for a Raft-based system that replicates logs like *MongoRaftReconfig* and are required for completeness of the proof. Many of the arguments are similar to those in the original Raft dissertation [64]. In Section A.1.3 we establish the *LogMatching* property (Lemma 23) with the help of a few auxiliary lemmas, and in Section A.1.3 we establish some additional, higher level lemmas about server logs. All lemmas of the preceding sections (A.1.2) hold in all reachable states of the protocol, so we can utilize them below.

Log Matching

In this section we assume that Lemmas 21, 22, and 23 act as strengthening assumptions for the inductive hypothesis needed to prove Lemma 23, the *LogMatching* property. That is, we assume all of these lemmas hold in the current state, and show each is upheld by any protocol transition, as we did in Section A.1.2. Lemma 20 is an additional, helpful fact that we establish first.

Lemma 20 (GetEntries ensures prefix)**.** *If Lemma 23 holds in the current state, then after a* $\mathit{GetEntries}(i, j)$ *action,* $\mathit{log}'[i]$ *is a prefix of* $\mathit{log}[j]$.

Proof. $\mathit{GetEntries}(i, j)$ can only occur if $\mathit{Len}(\mathit{log}[j]) > \mathit{Len}(\mathit{log}[i])$ and $\mathit{log}[i][\mathit{Len}(\mathit{log}[i])] = \mathit{log}[j][\mathit{Len}(\mathit{log}[i])]$. If Lemma 23 holds currently, then we know that $\mathit{log}[i] = \mathit{log}[j][..\mathit{Len}(\mathit{log}[i])]$. After the $\mathit{GetEntries}(i, j)$ action occurs, $\mathit{log}'[i] = \mathit{log}[i] \circ \langle \mathit{log}[j][\mathit{Len}(\mathit{log}[i]) + 1] \rangle$. So, we know that

$$\begin{aligned}
\mathit{log}'[i] &= \mathit{log}[i] \circ \langle \mathit{log}[j][\mathit{Len}(\mathit{log}[i]) + 1] \rangle \\
&= \mathit{log}[j][..\mathit{Len}(\mathit{log}[i])] \circ \langle \mathit{log}[j][\mathit{Len}(\mathit{log}[i]) + 1] \rangle \\
&= \mathit{log}[j][..(\mathit{Len}(\mathit{log}[i]) + 1)]
\end{aligned}$$

showing that $\mathit{log}'[i]$ is a prefix of $\mathit{log}[j]$. □

Lemma 21 (Log entry in term implies config in term)**.** *If a log entry $E = (ind, t)$ exists in the log of some server, then there exists some server in a configuration C such that $C.t \geq t$.*

Proof. In all initial states, the logs of all servers are empty, so the lemma holds. We must consider those actions which modify server logs, terms, or configurations: $ClientRequest$, $GetEntries$, $RollbackEntries$, $UpdateTerms$, $BecomeLeader$, $Reconfig$, and $SendConfig$.

- $ClientRequest(i)$ creates a log entry $(ind, term[i])$ on a primary i in configuration C_i. To show that Lemma 21 holds in the next state we must ensure that there exists some configuration C where $C.t \geq term[i]$. By Lemma 15, we know that $C_i.t = term[i]$. Since $C_i.t \geq term[i]$, and the configuration of i is unmodified by this action, Lemma 21 is upheld in the next state.

- After a $GetEntries(i, j)$ action occurs, the log of the receiving server, $log'[i]$, is a prefix of the sender's log, $log[j]$, by Lemma 20. So, if Lemma 21 was satisfied for $log[j]$, it will be satisfied for $log'[i]$, since $log'[i]$ will contain a subset of the entries present in $log[j]$, and no configurations are modified by this action.

- *RollbackEntries* only deletes log entries, so it cannot falsify the lemma in the next state.

- *UpdateTerms* only modifies server terms or primary status, so it cannot falsify the lemma in the next state.

- $BecomeLeader(i)$ updates the configuration of a primary server i from C_i to C_i', where $C_i' > C_i$ (by Lemma 13), and the action does not modify any server logs. Since it only modifies the configuration of server i, and $C_i' > C_i$, Lemma 21 must hold in the next state if it held currently.

- $Reconfig(i)$ updates the configuration of server i from C_i to C_i', and does not modify any server terms or logs. Since $C_i'.t = C_i.t$, it must uphold the lemma.

- $SendConfig(i, j)$ updates the current configuration of a server j from C_j to C_j', and does not modify any server logs, other configurations, or terms. Since $C_j' > C_j$, by Lemma 13, Lemma 21 must be upheld.

□

Lemma 22 (Primary has entries it created)**.** *For any log entry $E = (ind, t)$ that exists on some server, if a server s is primary in term t, then $log[s]$ must contain E.*

Proof. In all initial states, the logs of all servers are empty, so the lemma holds. The actions that could possibly falsify this lemma in the next state are those that modify server logs or primary status: *BecomeLeader*, *ClientRequest*, *GetEntries*, and *RollbackEntries*.

- $BecomeLeader(i)$ elects a primary i in $term'[i]$ in configuration C_i, and we know that i is the unique primary of $term'[i]$, by Lemma 1. We also know that C_i is active, otherwise the election could not have occurred (Lemma 12). The only way Lemma 22 could be falsified in the next state is if, in the current state, there exists some server j such that there is a log entry $E_j = (ind_j, term'[i])$ in $log[j]$ and E_j is not contained in $log[i]$. If E_j exists, though, by Lemma 21 this would imply that, in the current state, there exists some configuration in term $\geq term'[i]$. Then, by Lemma 19, this would imply that all quorums of active configurations in the current state contain some server in term $\geq term'[i]$. This means that all quorums of C_i would contain some server in term $\geq term'[i]$, preventing the $BecomeLeader(i)$ action from occurring, due to the voting precondition on terms (Algorithm 7, Line 34).

- $ClientRequest(i)$ appends a new log entry $E = (ind, term[i])$ to the log of a primary server i. Since there is a unique primary per term (Lemma 1), we are assured that E is present in the log of server i, the only primary in $term[i]$, since i is the server that created the entry.

- $GetEntries(i, j)$ sends a new log entry from server j to a secondary server i. We know that $log'[i]$ is a prefix of $log[j]$, by Lemma 20. So, the entries contained in $log'[i]$ are a subset of those in $log[j]$. Thus, if Lemma 22 held currently for all entries in $log[j]$, it will hold in the next state for all entries in $log'[i]$, since the logs of no primary servers are modified.

- *RollbackEntries* only modifies log entries on a secondary server, so it cannot falsify Lemma 22 in the next state.

□

Lemma 23 (Log Matching). *An* $(index, term)$ *pair uniquely identifies a log prefix.*

$$\forall s, t \in Server :$$
$$\forall ind \in (1..Len(log[s]) \cap 1..Len(log[t])) :$$
$$(log[s][ind] = log[t][ind]) \Rightarrow (log[s][..ind] = log[t][..ind])$$

Proof. In all initial states, the logs of all servers are empty, so the lemma holds. If Lemma 23 holds in the current state, the only possible actions that could falsify it in the next state are those that affect the state of server logs: *ClientRequest*, *GetEntries*, and *RollbackEntries*.

- $ClientRequest(i)$ appends a single entry to the log of a primary server s in $term[i]$. Let $ind_i = Len(log[i])$. So, we know that

$$log'[ind_i + 1] = term[i] \tag{A.1}$$

The only way for this action to violate Lemma 23 in the next state is if there exists some server $j \neq i$ such that both of the following hold:

$$log'[i][ind_i + 1] = log[j][ind_i + 1] \tag{A.2}$$
$$log'[i][..(ind_i + 1)] \neq log[j][..(ind_i + 1)] \tag{A.3}$$

That is, $log[j]$ contains an entry in $term[i]$ at index $ind_i + 1$, but it has a prefix that differs from $log'[i]$. This cannot be possible, though, since, from statements A.1 and A.2 above, we know that $log[j][ind_i + 1] = term[i]$. By Lemma 22, this implies that the entry $(ind_i + 1, term[i])$ must be contained in $log[i]$. So, it must be that $Len(log[i]) \geq ind_i + 1$, contradicting our assumption that $Len(log[i]) = ind_i$.

- $GetEntries(i, j)$ sends a single log entry from server i to server j, and, by Lemma 20, we know that $log'[i]$ is a prefix of $log[j]$ after the action occurs. So, if Lemma 23 held in the current state, it will hold in the next state.

- *RollbackEntries* truncates a single entry from the end of a server's log, so it cannot not violate Lemma 23.

□

Additional Log Lemmas

In this section we prove some additional log lemmas. Each is proved inductively. We assume that all lemmas established in the preceding sections (A.1.2, A.1.3) hold in all reachable states of the protocol, so we can utilize them below.

Lemma 24 (Primary term at least as large as log term)**.** *For any server that is currently primary, its current term must be $\geq$ the largest term of any entry in its log.*

$$\forall s \in Server : (state[s] = Primary) \Rightarrow \forall ind \in 1..Len(log[s]) : term[s] \geq log[s][ind]$$

Proof. In all initial states, the logs of all servers are empty, so the lemma holds. We must consider actions that modify the primary status of a server, terms of a server, or its logs: *BecomeLeader*, *ClientRequest*, *GetEntries*, *RollbackEntries*, *UpdateTerms*.

- $BecomeLeader(i)$ elects a new primary server i in $term'[i]$. It also updates the configuration of i from C_i to C_i', where $(C_i.m, C_i.v) = (C_i'.m, C_i'.v)$ and $C_i'.t > C_i.t$ (by Lemma 13). To uphold Lemma 24, we must be sure that $term'[i]$ is $\geq$ the largest term of any entry in $log[i]$. Assume there was an index ind such that $log[i][ind] > term'[i]$. By Lemma 21, this implies that there exists some server j, in configuration C_j, such that $C_j.t = log[i][ind]$. By Lemma 19, this implies that all quorums of C_i, which must have been active in the current state for the $BecomeLeader(i)$ to occur (Lemma 12), contain some server in term $\geq log[i][ind]$. Since $log[i][ind] > term'[i]$, this would have prevented the $BecomeLeader(i)$ action from occurring, due to its voting precondition on terms (Algorithm 7, Line 34).

- $ClientRequest(i)$ creates a new entry $(ind, term[i])$ on a primary in $term[i]$, and does not modify the state of any other server. If Lemma 24 holds currently, then, since the term of the new log entry is $term[i]$, it will continue to hold in the next state, since $term[i] \leq term[i]$.

- *GetEntries* only modifies the state of server logs on a secondary server, so could not falsify the lemma.
- *RollbackEntries* only removes log entries from a secondary server, so it could not falsify the lemma if it holds in the current state.
- *UpdateTerms* only increases the term of a server and does not modify any server logs, so it could not violate the lemma if it holds currently.

□

Lemma 25 (Log entry terms increase monotonically). *For all $s \in Server$, the terms of the log entries in $log[s]$ increase monotonically.*

$$\forall s \in Server : \forall ind_i, ind_j \in 1..Len(log[s]) : (ind_i < ind_j) \Rightarrow log[s][ind_i] \leq log[s][ind_j]$$

Proof. In all initial states, the logs of all servers are empty, so the lemma holds. We only need to consider actions that modify server logs: *ClientRequest*, *GetEntries*, and *RollbackEntries*.

- A $ClientRequest(i)$ appends a new log entry $(ind, term[i])$ on a primary i in term $term[i]$. By Lemma 24 we know that $term[i]$ is $\geq$ the largest term of any entry in $log[i]$. So, $log'[i][ind]$ must be $\geq$ the term of the largest entry in $log[i]$, ensuring monotonicity of log entry terms in $log'[i]$.
- $GetEntries(i, j)$ ensures that the log of the receiving server, i, log is a prefix of the sender, j, by Lemma 20. So, if the sender's log satisfies Lemma 25 then the receiver's also will.
- *RollbackEntries* only deletes log entries, so it must maintain Lemma 25.

□

Lemma 26 (Uniform log entries in term). *For all $i, j \in Server$, if $log[i]$ contains a log entry (ind_i, t) and $log[j]$ contains an entry (ind_j, t), where $ind_j < ind_i$, it must be that $log[i][ind_j] = t$.*

$$\begin{aligned}
&\forall i, j \in Server : \\
&\forall ind_i \in 1..Len(log[i]) : \\
&\forall ind_j \in 1..Len(log[j]) : \\
&\quad (ind_j < ind_i \land log[i][ind_i] = log[j][ind_j]) \Rightarrow (log[i][ind_j] = log[i][ind_i])
\end{aligned}$$

Proof. In all initial states, the logs of all servers are empty, so the lemma holds. The only actions that could falsify this lemma are those that modify logs of servers: *ClientRequest*, *GetEntries*, *RollbackEntries*.

- $\mathit{ClientRequest}(i)$ appends a new log entry $(\mathit{ind}_i, \mathit{term}[i])$ on primary server i in $\mathit{term}[i]$. The only way this action could falsify Lemma 26 in the next state is in the following two cases:

  – There is another server j that contains an entry $(\mathit{ind}_j, \mathit{term}[i])$ such that

  $$\begin{aligned} &\mathit{ind}_j < \mathit{ind}_i \\ \wedge\ &\mathit{log}'[i][\mathit{ind}_j] \neq \mathit{term}[i] \end{aligned}$$

  By Lemma 22, we know that a primary server has all log entries that exist in its own term, so if $\mathit{log}[j][\mathit{ind}_j] = \mathit{term}[i]$, it must be that $\mathit{log}'[i][\mathit{ind}_j] = \mathit{term}[i]$, since server i is primary in $\mathit{term}[i]$ in both the current and next state.

  – There is another server j that contains an entry $(\mathit{ind}_j, \mathit{term}[i])$ such that

  $$\begin{aligned} &\mathit{ind}_i < \mathit{ind}_j \\ \wedge\ &\mathit{log}[j][\mathit{ind}_i] \neq \mathit{term}[i] \end{aligned}$$

  By Lemma 22, we know that a primary server must have all entries in its term, so if $(\mathit{ind}_j, \mathit{term}[i])$ exists, it must be contained in $\mathit{log}'[i]$, which means that $\mathit{ind}_i \geq \mathit{ind}_j$, contradicting our assumption that $\mathit{ind}_i < \mathit{ind}_j$.

- For a $\mathit{GetEntries}(i, j)$ action, we know that the receiver's log after the action is a prefix of the sender's log, by Lemma 20, so this action could not falsify the lemma, since it held currrently for the sender's log.

- *RollbackEntries* only deletes log entries, so it could not falsify the lemma in the next state.

□

A.1.4 Leader Completeness

In this section we present the proof of Theorem 2, which relies on the auxiliary lemmas of this section and those proven in the previous sections. Similar to the proofs in the preceding sections, we assume that all of the lemmas in this section (Lemmas 27, 28, 29 and Theorem 2) act as strengthening assumptions for the inductive hypothesis needed to prove Theorem 2. All lemmas established in the preceding sections (A.1.2, A.1.3) hold in all reachable states of the protocol, so we can utilize them below.

Lemma 27 (Logs later than committed must have past committed entries). *If a log contains an entry (ind, t), then it also contains all entries committed in terms $< t$.*

$$\begin{aligned}
&\forall s \in \mathit{Server} : \\
&\forall (\mathit{index}, \mathit{term}) \in \mathit{committed} : \\
&\forall \mathit{ind}_s \in 1..\mathit{Len}(\mathit{log}[s]) : \\
&\quad \mathit{term} < \mathit{log}[s][\mathit{ind}_s] \Rightarrow \mathit{InLog}(\mathit{index}, \mathit{term}, s)
\end{aligned}$$

Proof. In all initial states, the logs of all servers are empty, so the lemma holds. We must consider the actions that modify server logs or the set of committed entries: *ClientRequest*, *GetEntries*, *RollbackEntries*, and *CommitEntry*.

- $\mathit{ClientRequest}(i)$ appends a new, uncommitted entry $(\mathit{ind}, \mathit{term}[i])$ to the log of primary server i in term $\mathit{term}[i]$. We must show that $\mathit{log}'[i]$ contains all log entries committed in terms $< \mathit{term}[i]$. By the assumption of Theorem 2 in the current state, we know that server i, which is primary in $\mathit{term}[i]$, contains all log entries committed in terms $< \mathit{term}[i]$. The newly appended entry of $(\mathit{ind}, \mathit{term}[i])$ is uncommitted, so the set of committed entries is not changed by this action and no other server logs are modified. So, if server i contained all entries committed in terms $< \mathit{term}[i]$ in the current state, it will in the next state.

- After a $\mathit{GetEntries}(i, j)$ action, due to Lemma 20, $\mathit{log}'[i]$ is a prefix of $\mathit{log}[j]$ and the set of committed entries is unmodified. So, if $\mathit{log}[j]$ satisfied Lemma 27 in the current state, then $\mathit{log}'[i]$ will also satisfy it.

- $\mathit{RollbackEntries}(i, j)$ removes a single log entry, E, from the end of a secondary server i's log. In order for this to falsify the lemma in the next state, it would have to be the case that E is a committed log entry. In order for a $\mathit{RollbackEntries}(i, j)$ action to occur, the $\mathit{CanRollback}(i, j)$ predicate must be satisfied for servers i and j. This predicate is satisfied if

$$(\mathit{LogTerm}(i) < \mathit{LogTerm}(j)) \land \neg \mathit{IsPrefix}(\mathit{log}[i], \mathit{log}[j])$$

  If $\mathit{LogTerm}(i) < \mathit{LogTerm}(j)$ and $E = (\mathit{ind}, \mathit{LogTerm}(i))$ were committed in the current state, it would imply that E_i is contained in $\mathit{log}[j]$, by assumption of Lemma 27. If this were the case, though, then

$$\mathit{log}[j][\mathit{Len}(\mathit{log}[i])] = \mathit{log}[i][\mathit{Len}(\mathit{log}[i])]$$

  implying, by Lemma 23, that $\mathit{log}[i]$ is a prefix of $\mathit{log}[j]$, contradicting the precondition of *RollbackEntries* requiring that $\mathit{log}[i]$ cannot be a prefix of $\mathit{log}[j]$.

- $\mathit{CommitEntry}(i)$ commits the latest log entry $E_i = (\mathit{ind}, \mathit{term}[i])$ of a primary server i in $\mathit{term}[i]$ in configuration C_i. In order for this action to falsify Lemma 27 in the next state, there must exist some server j such that $log[j]$ contains an entry $E_j = (ind_j, t_j)$ where $t_j > term[i]$ and E_i is not contained in $log[j]$. If E_j exists with term t_j, though, this implies, by Lemma 21, that there exists a configuration C_k, where $C_k.t \geq t_j$. By Lemma 29, this implies that all quorums of primary server i's configuration, C_i, must contain some server in term $> term[i]$. This would prevent the $CommitEntry(i)$ action from occurring, due to its precondition that requires a quorum of servers in C_i to be in $term[i]$ (Algorithm 7, Line 68).

□

Lemma 28 (Active configs overlap with committed entries)**.** *For any active configuration C and committed entry E, all quorums of C contain some server that has entry E in its log.*

$$\forall s \in ActiveConfigSet :$$
$$\forall (index, term) \in committed :$$
$$\forall Q \in Quorums(config[s]) : \exists n \in Q : InLog(index, term, n)$$

Proof. In all initial states, the set of committed entries is empty, so the lemma holds. The only actions that could possibly falsify this property in the next state are those that delete log entries, modify configurations, or update the set of committed entries: *RollbackEntries*, *Reconfig*, *SendConfig*, *BecomeLeader*, and *CommitEntry*.

- $RollbackEntries(i, j)$ truncates one entry from the log of a secondary server. So, in order for this action to falsify Lemma 28 in the next state, it must delete a committed log entry on some server. As argued in the *RollbackEntries* case of the proof of Lemma 27, though, *RollbackEntries* cannot delete a committed log entry.

- $Reconfig(i)$ updates the configuration of a primary server i in $term[i]$ from C_i to C_i'. We know that no new, active configurations on other servers could be created by this action since no configurations on other servers are changed, and deactivated configurations cannot become active (Lemma 14). Also, server logs and the set of committed entries are not modified. So, we only need to show that, if C_i' is active, then, for all committed entries E, all quorums of C_i' contain some server that has E in its log. Suppose there is some quorum $Q \in Quorums(C_i'.m)$ and committed entry $E_j = (ind_j, t_j)$ such that no server in Q contains entry E_j in its log. By the assumption of Lemma 28 in the current state, we know that all quorums of C_i contain some server with E_j in its log, since C_i is active (by Lemma 12). Furthermore, we know that the *Oplog Commitment* precondition, $P1(i)$, must have been satisfied in the current state in order for the $Reconfig(i)$ action to occur. This implies that, for all entries committed in $term[i]$, some quorum $Q_i \in Quorums(C_i.m)$ contains this log entry and all servers $n \in Q_i$ are in $term[i]$. Now, consider the following two cases:

  - $t_j \leq \mathit{term}[i]$
    We know that all servers in Q_i contain all entries committed in $\mathit{term}[i]$. So, by Lemma 27, we also know that the logs of all servers in Q_i contain all entries committed in terms $\leq \mathit{term}[i]$. So, the log of every server in Q_i must contain E_j. Since the quorums of C_i and C_i' overlap, all quorums of C_i' must contain some server that has entry E_j in its log, upholding Lemma 28.

  - $t_j > \mathit{term}[i]$
    By Lemma 21, this implies there exists a configuration C_j such that $C_j.t = t_j$. If C_j exists, though, then this implies that the $\mathit{Reconfig}(i)$ could not have occurred, due to Lemma 19 and the *Term Quorum Check* precondition, $Q2(i)$ (Algorithm 7, Line 18), which requires a quorum of servers in C_i to be in $\mathit{term}[i]$. Since $C_j.t = t_j$ and $t_j > \mathit{term}[i]$, the $\mathit{Reconfig}(i)$ would have been prevented.

- $\mathit{SendConfig}(i, j)$ cannot create new configurations, cannot activate any existing configurations (by Lemma 14), and does not modify the logs of any servers or the set of committed entries, so it must uphold Lemma 28.

- $\mathit{BecomeLeader}(i)$ updates the configuration on server i from C_i to C_i', where $(C_i'.m, C_i'.v) = (C_i.m, C_i.v)$. It does not change the member set of any existing configuration and does not modify the set of committed entries or server logs. So, if C_i' is active in the next state, we know that its quorums will overlap with some server containing a committed entry, for all committed entries, since we know this held for C_i, and $C_i.m = C_i'.m$.

- $\mathit{CommitEntry}(i)$ commits a log entry E_i on a primary server i in configuration C_i, and does not modify server logs or configurations. In order for this to falsify the lemma in the next state, there must exist some server j, in active configuration C_j, and some quorum $Q_j \in \mathit{Quorums}(C_j.m)$ such that no server in Q_j contains entry E_i. If a primary i can commit a log entry in C_i, this means that there is a quorum $Q_i \in \mathit{Quorums}(C_i.m)$ such that all servers in Q_i contain E_i and are in $\mathit{term}[i]$. This implies that $\neg \mathit{QuorumsOverlap}(C_i.m, C_j.m)$, since otherwise $Q_i \cap Q_j \neq \emptyset$, implying Q_j contains some server with entry E_i. So, this must imply that $C_i.m \neq C_j.m$, which, by Lemma 16, implies that $(C_i.v, C_i.t) \neq (C_j.v, C_j.t)$. So, we have the following two cases to consider:

  - $C_i < C_j$
    If $C_i.t = C_j.t$, this would imply that server j had a newer configuration, C_j, in term $C_i.t$, which, by Lemma 17, could not be possible since primary i contains the newest configuration of its term. So, it must be that $C_i.t < C_j.t$. By Lemma 29 this implies that C_i is prevented from executing a $\mathit{CommitEntry}(i)$ action

in $\mathit{term}[i]$, since all quorums of C_i must intersect with some server in term $> \mathit{term}[i]$.

– $C_i > C_j$

If primary i is able to commit an entry in $\mathit{term}[i]$, the term of its configuration, C_i, must be greater than all other configurations. If another configuration existed in a term $> \mathit{term}[i]$, it would prevent a $\mathit{CommitEntry}(i)$ action, by Lemma 29. This means that configuration C_i is the newest configuration in existence, since there are no configurations that exist in higher terms, and we know that primary i contains the newest configuration of its term, by Lemma 17. If, however, there are no configurations newer than C_i, this implies that C_i must be active, since the definition of deactivation requires the existence of at least some configuration $> C_i$. If C_i is active, then by Lemma 18, this means that all quorums of C_i and C_j overlap, implying that $Q_i \cap Q_j \neq \emptyset$. So, Q_j must contain some server that has entry E_i.

□

Lemma 29 (Newer configs disable commits in older terms). *For all servers $i, j \in \mathit{Server}$, in configurations C_i and C_j, respectively, if server i is primary and $C_j.t > \mathit{term}[i]$, then all quorums of C_i must contain some server in term $> \mathit{term}[i]$.*

$$\begin{aligned}
&\forall s, t \in \mathit{Server} : \\
&\quad (\mathit{state}[t] = \mathit{Primary} \land \mathit{term}[t] < \mathit{configTerm}[s]) \Rightarrow \\
&\quad \forall Q \in \mathit{Quorums}(\mathit{config}[t]) : \exists n \in Q : \mathit{term}[n] > \mathit{term}[t]
\end{aligned}$$

Proof. In all initial states, no servers are primary, so the lemma must hold. We must consider actions that modify configurations, primary status of a server, or terms of servers: *Reconfig*, *SendConfig*, *BecomeLeader*, and *UpdateTerms*.

- $\mathit{Reconfig}(i)$ updates the configuration on a primary server i from C_i to C_i'. For this action to falsify Lemma 29 in the next state, there are two cases:

  – There exists some server j, in configuration C_j, such that $C_j.t > \mathit{term}[i]$, and some quorum $Q \in \mathit{Quorums}(C_i'.m)$ that does not contain a server in term $> \mathit{term}[i]$. By assumption of Lemma 29 in the current state, though, we know that all quorums of C_i contain some server in term $> \mathit{term}[i]$. In order for the $\mathit{Reconfig}(i)$ to occur, the *Term Quorum Check* precondition, $Q2(i)$, must have been satisfied, which requires that there exists some quorum $Q_i \in \mathit{Quorums}(C_i.m)$ such that $\mathit{term}[n] = \mathit{term}[i]$, for all $n \in Q_i$. This, however, contradicts the assumption that all quorums of C_i contain some server in term $> \mathit{term}[i]$. So, such a $\mathit{Reconfig}(i)$ action could not have occurred.

  - There exists some primary server $j \neq i$, such that $C'_i.t > \mathit{term}[j]$, and there is some quorum $Q \in \mathit{Quorums}(C_j.m)$ such that Q does not contain a server in term $> \mathit{term}[j]$. Since we know that $C'_i.t = C_i.t$ and $C'_i.t > \mathit{term}[j]$, we know, by assumption of Lemma 29 in the current state, that all quorums of C_j contain some server in term $> \mathit{term}[j]$. So, the lemma must continue to hold in the next state.

- $\mathit{SendConfig}(i, j)$ transfers the configuration of server i to a secondary server j. This action does not create any new configurations, and it doesn't modify terms or the configuration of a primary server, so if the lemma held for all configurations in the current state, it will hold in the next state.

- $\mathit{BecomeLeader}(i, Q)$ elects a server i as primary with voters Q and updates its configuration from C_i to C'_i, where $C'_i.t > C_i.t$, by Lemma 13. In order to falsify this lemma there are two cases:

  - There exists another server j, in configuration C_j, such that $C_j.t > C'_i.t$ and there is some quorum $Q_i \in \mathit{Quorums}(C'_i.m)$ such that Q_i does not contain a server in term $> C'_i.t$. If C_j exists and $C_j.t > C'_i.t$, then Lemma 19 implies that all quorums of C_i must contain some server in term $\geq C_j.t$, since C_i must be active in order for a $\mathit{BecomeLeader}(i)$ action to occur (Lemma 12). Since $C_j.t > C'_i.t > C_i.t$, this would have prevented the $\mathit{BecomeLeader}(i)$ action from occurring, due to its voting precondition on terms.
  - There exists some primary server $j \neq i$, such that $C'_i.t > \mathit{term}[j]$, and there is some quorum $Q_j \in \mathit{Quorums}(C_j.m)$ such that Q_j does not contain a server in term $> \mathit{term}[j]$. If C_j, the configuration of server j, is deactivated, then this implies the existence of some configuration $> C_j$ in the current state which, by the assumption of Lemma 29 in the current state, would ensure that all quorums of C_j contain some server in term $> \mathit{term}[j]$, since j is primary, which ensures Lemma 29. If C_j is active, then both it and C_i must be active, since the $\mathit{BecomeLeader}(i)$ action was able to occur (Lemma 12). If both C_i and C_j are active, this implies that $\mathit{QuorumsOverlap}(C_i.m, C_j.m)$. After the $\mathit{BecomeLeader}(i)$ action occurs, due to its postcondition, all servers in Q will have a term of $C'_i.t > \mathit{term}[j]$. Since $C'_i.m = C_i.m$, this means that a quorum of servers in C_i will also have a term of $C'_i.t$. So, all quorums of C_j will contain some server in term $> \mathit{term}[j]$, upholding Lemma 29..

- An *UpdateTerms* action could not falsify the lemma in the next state since it only updates server terms, which increase monotonically on all servers, and if it updates the term of a server s it sets $\mathit{state}[s] \leftarrow \mathit{Secondary}$.

□

Proof of Theorem 2. In all initial states, the logs of all servers are empty, so the theorem holds. If we assume Theorem 2 holds currently, it could only be falsified in the next state via actions that elect a primary or commit a log entry: *BecomeLeader* and *CommitEntry*.

- $BecomeLeader(i, Q)$ elects a primary server i in a configuration C_i in term $term'[i]$ with a quorum of voters Q. For such an election to occur we know that C_i must be active, by Lemma 12. To falsify the lemma, there must be some committed entry $E_j = (ind_j, t_j)$, such that $t_j < term'[i]$. and $log[i]$ does not contain E_j. Since C_i is active, we know, by Lemma 28, that there must be some server $n \in Q$ such that E_j is in $log[n]$. If n voted for i to become primary, we know that $LogGeq(i, n)$ (defined in Algorithm 7) must have been satisfied in the current state. If we let

$$(ind_i, t_i) = (Len(log[i]), LogTerm(i))$$
$$(ind_n, t_n) = (Len(log[n]), LogTerm(n))$$

  there are two cases to consider:

  - $t_i = t_n \land ind_i \geq ind_n$
    If $ind_i = ind_n$, then we have $log[i][ind_i] = t_i = t_n = log[n][ind_i]$. By Lemma 23, this tells us that $log[i][..ind_i] = log[n][..ind_i]$, implying that $log[i]$ contains E_j, since $log[n]$ contains it. So, we consider the case where $ind_i > ind_n$. If $log[i][ind_i] = t_i$, and $log[n][ind_n] = t_n = t_i$, then, by Lemma 26, this implies that $log[i][ind_n] = t_i = log[n][ind_n]$. So, by Lemma 23, it must be that $log[i][..ind_n] = log[n][..ind_n]$. Since we know that $log[n]$ contains the committed entry E_j, this must mean that $log[i]$ contains it, contradicting our assumption that it did not contain E_j.
  - $t_i > t_n$
    First, it must be that $t_n \geq t_j$, since the last entry of $log[n]$ has term t_n, we know that $log[n]$ contains entry E_j in term t_j, and log entry terms increase monotonically (Lemma 25). So, we have the following:

$$t_j \leq t_n < t_i$$

    From Lemma 27, we know that $log[i]$ contains all entries committed in terms $< t_i$. So, it contains all entries committed in terms t_j, contradicting our assumption that it did not contain E_j in its log.

- $CommitEntry(i)$ commits a log entry E in $term[i]$ on a primary server i. Assume there is some other server $j \neq i$, in configuration C_j, that is currently primary in term $t_j > term[i]$ and E is not contained in $log[j]$. From Lemma 15, we know that $C_j.t = t_j$. And, by Lemma 29, we know that the existence of C_j prevents commits of log entries occurring in any terms $< t_j$. So, such a $CommitEntry(i)$ entry action could not occur.

□

A.2 Detailed Model Checking Results

In this section we provide additional details of our automated verification results using the TLC model checker. We first give a brief overview of TLC and its mode of operation, and then provide more detailed results from checking the safety properties discussed in Section 3.4.4.

A.2.1 The TLC Model Checker

TLC is an explicit state model checker that can check temporal properties of a given TLA+ specification. It is provided as a Java program that takes as input a TLA+ module file, a model checker configuration file, and a set of command line parameters. For checking safety properties, TLC assumes a TLA+ specification of the form $Init \land \Box[Next]_{vars}$. The configuration file tells TLC the name of the specification to check and of the properties to be checked. In addition, the configuration file defines a *model* of the specification, which is an assignment of values to any constant parameters of the specification. It is also possible to provide a *state constraint*, which is a state predicate that can be used to constrain the set of reachable states. If TLC discovers a reachable state that violates the state constraint predicate, it will not add the state to its current graph of reachable states. TLC also allows definition of a *symmetry set*, which causes the model checker to consider states that have the same constant value under some permutation as equivalent, which can significantly reduce the set of reachable states for certain models [157]. A more complete and in-depth explanation of TLC behavior and parameters can be found in [35]. For all model checking runs discussed below we used TLC version 2.15 (adc67eb) running on CentOS Linux 7, with a 48-core, 2.30GHz Intel Xeon Gold 5118 CPU.

A.2.2 Details and Results

For checking safety of *MongoRaftReconfig* we used a model we refer to as *MCMongoRaftReconfig*, which imposes finite bounds on the *MongoRaftReconfig* TLA+ specification. The complete, runnable TLC configuration for this model can be found in the supplementary materials [54]. The model sets $Server = \{n1, n2, n3, n4\}$, and imposes the following state constraint:

$$\begin{aligned} StateConstraint \triangleq\ & \forall s \in Server : \\ & \quad \land\ currentTerm[s] \leq MaxTerm \\ & \quad \land\ Len(log[s]) \leq MaxLogLen \\ & \quad \land\ configVersion[s] \leq MaxConfigVersion \end{aligned}$$

This constraint, along with a finite *Server* set, is sufficient to make the reachable state space of this model finite, since it limits the size of the three unbounded variables of the

MCMongoRaftReconfig	
Server	$\{n1, n2, n3, n4\}$
MaxLogLen	2
MaxTerm	3
MaxConfigVersion	3
Constraint	*StateConstraint*
Symmetry	*Permutation(Server)*
Invariant	*LeaderCompleteness*
States	345,587,274
Depth	45
TLC Workers	20
Duration	8h 06min

MCMongoLoglessDynamicRaft	
Server	$\{n1, n2, n3, n4, n5\}$
MaxTerm	4
MaxConfigVersion	4
Constraint	*StateConstraint*
Symmetry	*Permutation(Server)*
Invariant	*ElectionSafety*
States	812,587,401
Depth	30
TLC Workers	20
Duration	19h 28min

Figure A.1: Summary of TLC Model Checking Results. *States* is the number of reachable, distinct states discovered by TLC. *Depth* is the length of the longest behavior.

specification: *terms*, *logs*, and configuration *versions*. It restricts logs to be of a maximum finite length, and imposes a finite upper bound on terms and configuration versions. Figure A.1 shows the parameters and results for this model. *Permutation* is an operator in the *TLC.tla* standard module [158] defined as the set of all permutations of elements in a given set. Under our symmetry declaration, any two states that are equal up to permutation of server identifiers are treated as equivalent by the model checker.

As discussed in Section 3.4.4, the compositional structure of *MongoRaftReconfig* makes it possible to verify *MongoLoglessDynamicRaft* in isolation and assume that its safety properties hold in *MongoRaftReconfig*. The full model checking results for our model, *MC-MongoLoglessDynamicRaft*, whose definition is provided in the supplementary materials [54], are presented in Figure A.1.

A.3 Subprotocol Refinement Proof

In order to ensure that any safety property of *MongoLoglessDynamicRaft* holds for *Mongo-RaftReconfig*, we must demonstrate that the behaviors of *MongoLoglessDynamicRaft* are not augmented when operating as a subprotocol of *MongoRaftReconfig*. To formalize this, we adopt TLA+ notation. Correctness properties and system specifications in TLA+ are both written as temporal logic formulas. This allows one to express notions of property satisfaction and refinement in a concise and similar manner. We say that a specification S satisfies a property P iff the formula $S \Rightarrow P$ is valid (i.e. true under all assignments). We say that a specification S_1 refines (or is a refinement of) S_2 iff $S_1 \Rightarrow S_2$ is valid i.e. every behavior of S_1 is a valid behavior of S_2 [159]. So, if we view $MongoRaftReconfig$ and $MongoLoglessDynamicRaft$ as temporal logic formulas describing the set of possible system behaviors for each respective protocol, we can formally state our refinement theorem as follows:

Theorem 30. $MongoRaftReconfig \Rightarrow MongoLoglessDynamicRaft$

To prove this, we must show that, (1) for any behavior σ of *MongoRaftReconfig*, the initial state of σ is a valid initial state of *MongoLoglessDynamicRaft* and (2) every transition in σ is a valid transition of *MongoLoglessDynamicRaft*. This proof has been formalized and checked in the TLA+ proof system. The full proof can be found in the supplementary materials [54].

Appendix B

Detailed Inductive Proof Graphs

In this section we provide detailed inductive proof graphs for the protocols and safety properties proven from Table 5.1.

- The full specification and TLAPS proof for *SimpleConsensus* can be found at [160], and the inductive proof graph for the *SimpleConsensus* safety property is shown in Figure B.1, along with full lemma definitions in B.2.
- The full specification and TLAPS proof for *TwoPhase* can be found at [161], and the inductive proof graph for the *TwoPhase* safety property is shown in Figure B.3, along with full lemma definitions in B.4.
- The full specification and TLAPS proof for *GermanCache* can be found at [162], and the inductive proof graph for the *GermanCache* safety property is shown in Figure B.5, along with full lemma definitions in B.6.
- The full specification and TLAPS proof for *Bakery* can be found at [163], and the inductive proof graph for the safety property of *Bakery* is shown in Figure B.7, along with full lemma definitions in B.8.
- The full specifications and TLAPS proofs for *AsyncRaft* can be found can be found at [164, 165]. The inductive proof graph for the *ElectionSafety* safety property of *AsyncRaft* is shown in Figure B.9, along with full lemma definitions in B.10, and TLAPS proof at [164]. The inductive proof graph for the *PrimaryOwnsEntries* safety property is shown in Figure B.11, along with full lemma definitions in B.13, and TLAPS proof at [165].

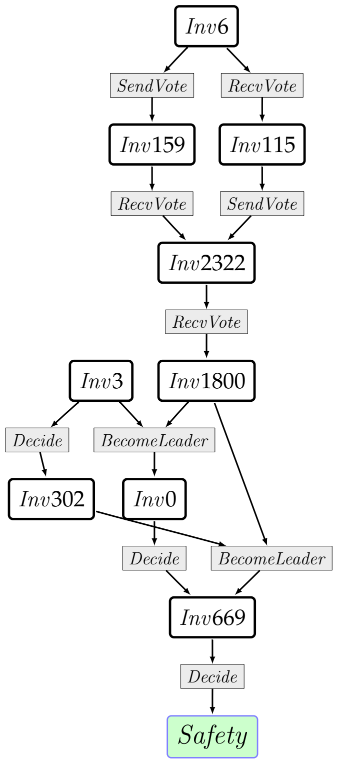


Figure B.1: *SimpleConsensus* inductive proof graph (alternate) for safety property (labeled as *Safety* in green).

$$Inv669 \triangleq \forall i, j \in Node : ((decided[j] = \{\}) \land leader[j]) \Rightarrow (decided[i] = \{\})$$
$$Inv1800 \triangleq \forall i, j, k \in Node : (i = k) \lor (\neg(j \in votes[k])) \lor (\neg(j \in votes[i]))$$
$$Inv302 \triangleq \forall i \in Node : \exists j \in Quorum : (j = votes[i]) \lor ((decided[i] = \{\}))$$
$$Inv0 \triangleq \forall i \in Node : \forall j \in Node : (i = j \land leader = leader) \lor (\neg(leader[i]) \lor (\neg(leader[j])))$$
$$Inv2322 \triangleq \forall i \in Node : \forall j \in Node : \forall k \in Node : (i = k) \lor (\neg(\langle j, i \rangle \in vote\_msg) \lor (\neg(j \in votes[k])))$$
$$Inv3 \triangleq \forall i \in Node : \exists j \in Quorum : (j = votes[i]) \lor (\neg(leader[i]))$$
$$Inv159 \triangleq \forall i, j, k \in Node : (j = k) \lor (\neg(\langle i, j \rangle \in vote\_msg) \lor (\neg(\langle i, k \rangle \in vote\_msg)))$$
$$Inv115 \triangleq \forall i, j \in Node : ((j \in votes[i])) \Rightarrow (voted[j])$$
$$Inv6 \triangleq \forall i, j \in Node : (voted[i]) \lor (\neg(\langle i, j \rangle \in vote\_msg))$$
$$\textbf{Safety} \triangleq \forall n_1, n_2 \in Node, v_1, v_2 \in Value : (v_1 \in decided[n_1] \land v_2 \in decided[n_2]) \Rightarrow (v_1 = v_2)$$

Figure B.2: Lemma definitions for *SimpleConsensus* inductive proof graph.

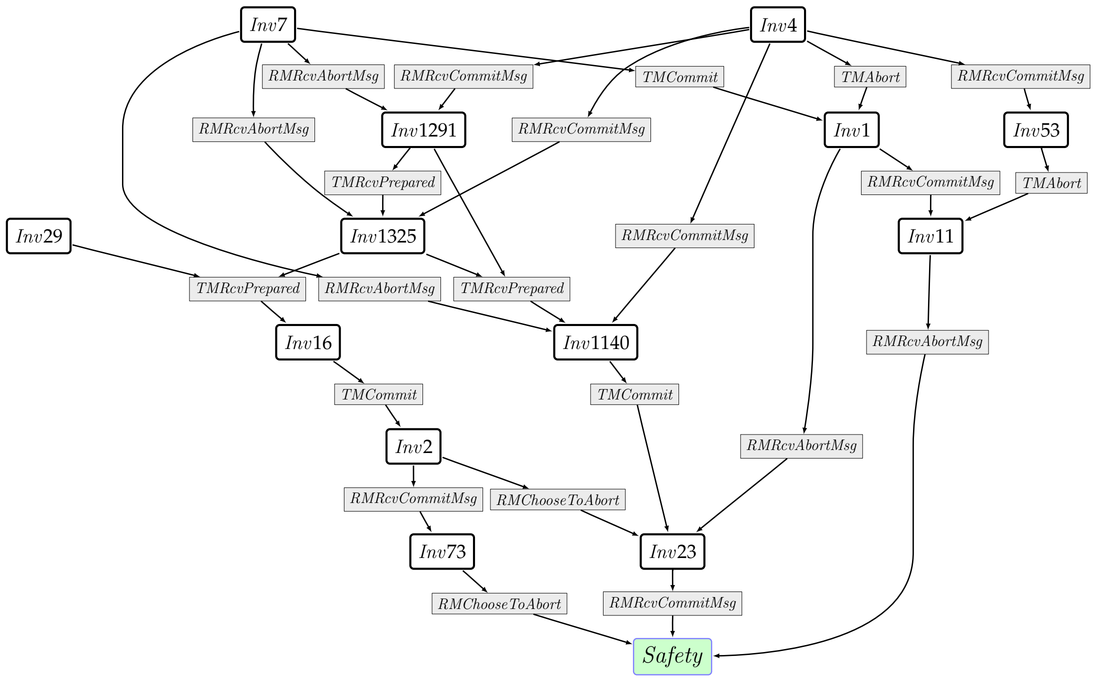


Figure B.3: *TwoPhase* inductive proof graph for safety property *Consistency* (labeled as *Safety* in green).

$Inv73 \triangleq \forall rm_i, rm_j \in RM : (rmState[rm_i] = "committed") \Rightarrow (rmState[rm_j] \neq "working")$

$Inv11 \triangleq \forall rm_j \in RM : ([type \mapsto "Abort"] \in msgsAbort) \Rightarrow (rmState[rm_j] \neq "committed")$

$Inv23 \triangleq \forall rm_i \in RM : \neg([type \mapsto "Commit"] \in msgsCommit) \vee (rmState[rm_i] \neq "aborted")$

$Inv2 \triangleq \forall rm_i \in RM : \neg([type \mapsto "Commit"] \in msgsCommit) \vee \neg(rmState[rm_i] = "working")$

$Inv1 \triangleq \neg([type \mapsto "Abort"] \in msgsAbort) \vee \neg([type \mapsto "Commit"] \in msgsCommit)$

$Inv53 \triangleq \forall rm_i \in RM : \neg(rmState[rm_i] = "committed") \vee \neg(tmState = "init")$

$Inv1140 \triangleq \forall rm_i \in RM : (rmState[rm_i] = "prepared") \vee \neg(tmPrepared = RM) \vee \neg(tmState = "init")$

$Inv16 \triangleq \forall rm_i \in RM : \neg(rmState[rm_i] = "working") \vee \neg(tmPrepared = RM)$

$Inv4 \triangleq \neg([type \mapsto "Commit"] \in msgsCommit) \vee \neg(tmState = "init")$

$Inv7 \triangleq \neg([type \mapsto "Abort"] \in msgsAbort) \vee \neg(tmState = "init")$

$Inv1325 \triangleq \forall rm_j \in RM : (rmState[rm_j] = "prepared") \vee \neg(tmPrepared = tmPrepared \cup \{rm_j\}) \vee \neg(tmState = "init")$

$Inv1291 \triangleq \forall rm_j \in RM : (rmState[rm_j] = "prepared") \vee ([type \mapsto "Prepared", rm \mapsto rm_j] \notin msgsPrepared) \vee (tmState \neq "init")$

$Inv29 \triangleq \forall rm_i \in RM : \neg([type \mapsto "Prepared", rm \mapsto rm_i] \in msgsPrepared) \vee (rmState[rm_i] \neq "working")$

Safety $\triangleq \forall rm_i, rm_j \in RM : \neg(rmState[rm_i] = "aborted" \wedge rmState[rm_j] = "committed")$

Figure B.4: Lemma definitions for *TwoPhase* inductive proof graph.

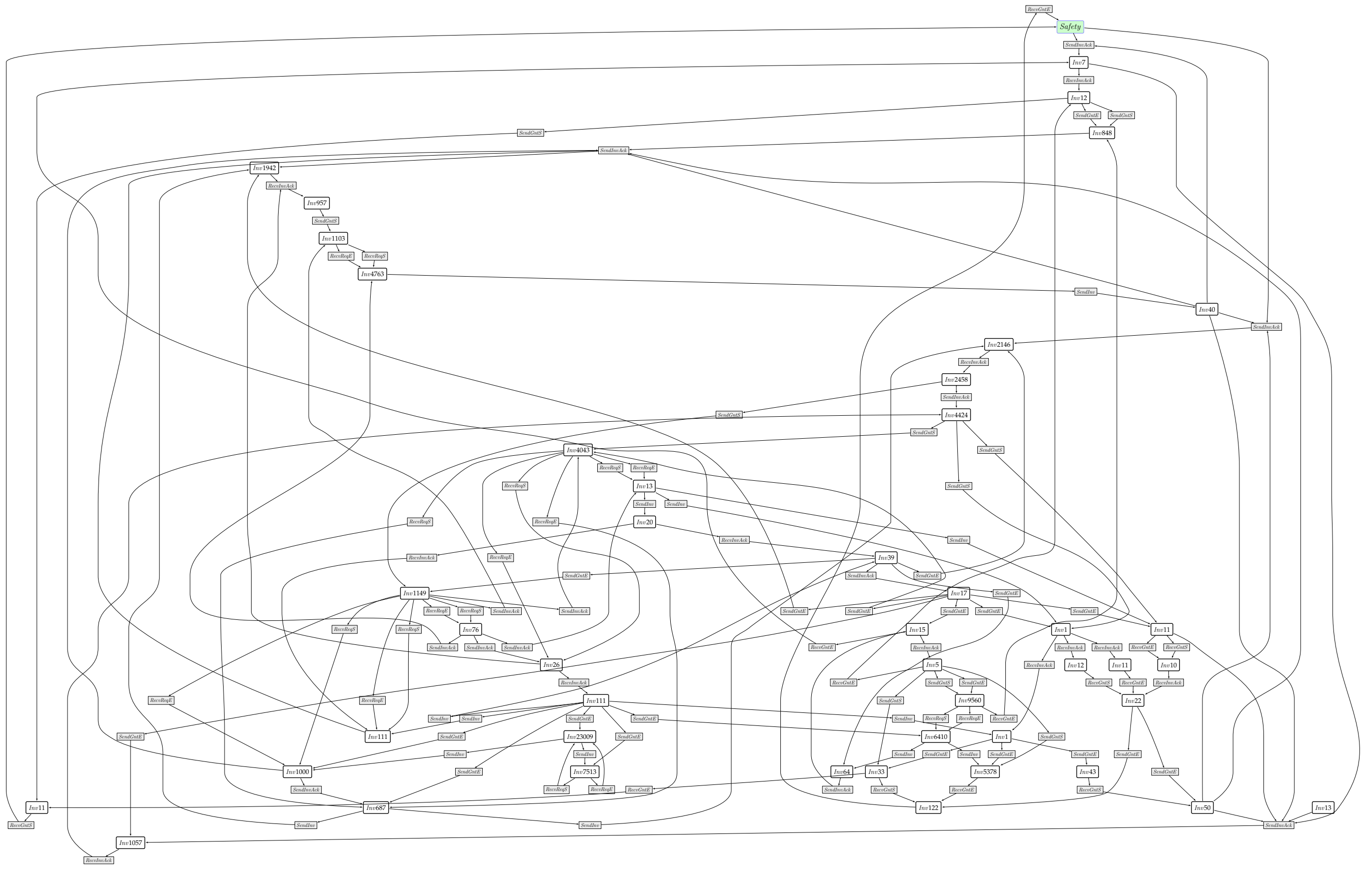


Figure B.5: *GermanCache* inductive proof graph for safety property (labeled as *Safety* in green).

$Inv122 \triangleq \forall i \in NODE : \forall j \in NODE : (Cache[j].State = ''I'') \lor \neg(Chan2[i].Cmd = ''GntE'')$

$Inv11 \triangleq \forall i \in NODE : \forall j \in NODE : \neg(Cache[i].State = ''E'') \lor \neg(Chan2[j].Cmd = ''GntS'')$

$Inv5378 \triangleq \forall i \in NODE : \forall j \in NODE : (Chan2[i].Cmd = ''Empty'') \lor (\neg(Chan2[j].Cmd = ''GntE'') \lor \neg(i \neq j))$

$Inv33 \triangleq \forall i \in NODE : \forall j \in NODE : \neg(Chan2[i].Cmd = ''GntE'') \lor \neg(Chan2[j].Cmd = ''GntS'')$

$Inv22 \triangleq \forall i \in NODE : (Cache[i].State = ''I'') \lor (ShrSet[i])$

$Inv12 \triangleq \forall i \in NODE : (ExGntd) \lor \neg(Cache[i].State = ''E'')$

$Inv1 \triangleq \forall i \in NODE : (Chan2[i].Cmd = ''Empty'') \lor (ShrSet[i])$

$Inv5 \triangleq \forall i \in NODE : (ExGntd) \lor \neg(Chan2[i].Cmd = ''GntE'')$

$Inv6410 \triangleq \forall i \in NODE : \forall j \in NODE : \neg(Chan2[i].Cmd = ''GntE'') \lor (\neg(InvSet[j]) \lor \neg(i \neq j))$

$Inv11 \triangleq \forall i \in NODE : (ShrSet[i]) \lor \neg(Chan2[i].Cmd = ''GntE'')$

$Inv12 \triangleq \forall i \in NODE : (ShrSet[i]) \lor \neg(Chan2[i].Cmd = ''GntS'')$

$Inv10 \triangleq \forall i \in NODE : (Cache[i].State = ''I'') \lor (Chan3[i].Cmd = ''Empty'')$

$Inv7 \triangleq \forall i \in NODE : \forall j \in NODE : (Chan3[i].Cmd = ''Empty'') \lor \neg(Cache[j].State = ''E'')$

$Inv1 \triangleq \forall i \in NODE : (Chan2[i].Cmd = ''Empty'') \lor (Chan3[i].Cmd = ''Empty'')$

$Inv111 \triangleq \forall i \in NODE : (ShrSet[i]) \lor \neg(InvSet[i])$

$Inv15 \triangleq \forall i \in NODE : \forall j \in NODE : (Chan3[i].Cmd = ''Empty'') \lor \neg(Chan2[j].Cmd = ''GntE'')$

$Inv9560 \triangleq \forall i \in NODE : \forall j \in NODE : \neg(Chan2[i].Cmd = ''GntE'') \lor \neg(i \neq j) \lor \neg(ShrSet[j])$

$Inv11 \triangleq \forall j \in NODE : (Chan2[j].Cmd = ''Empty'') \lor (Chan3[j].Cmd = ''Empty'')$

$Inv40 \triangleq \forall i \in NODE : \neg(Cache[i].State = ''I'') \lor \neg(Chan2[i].Cmd = ''Inv'')$

$Inv17 \triangleq \forall i \in NODE : (Chan3[i].Cmd = ''Empty'') \lor (ShrSet[i])$

$Inv4424 \triangleq \forall i \in NODE : (Chan3[i].Cmd = ''Empty'') \lor (CurCmd = ''ReqE'') \lor (ExGntd)$

$Inv13 \triangleq \forall i \in NODE : (Chan3[i].Cmd = ''Empty'') \lor \neg(InvSet[i])$

$Inv26 \triangleq \forall j \in NODE : (Chan3[j].Cmd = ''Empty'') \lor \neg(InvSet[j])$

$Inv64 \triangleq \forall i \in NODE : \forall j \in NODE : \neg(Chan2[i].Cmd = ''GntE'') \lor \neg(Chan2[j].Cmd = ''Inv'')$

$Inv4763 \triangleq \forall i \in NODE : \neg(Cache[i].State = ''I'') \lor (\neg(Chan2[i].Cmd = ''Empty'') \lor \neg(InvSet[i]))$

$Inv39 \triangleq \forall i \in NODE : (ShrSet[i]) \lor \neg(Chan2[i].Cmd = ''Inv'')$

$Inv1057 \triangleq \forall i \in NODE : \forall j \in NODE : (Chan3[i].Cmd = ''Empty'') \lor (\neg(ExGntd) \lor \neg(i \neq j)) \lor (Chan3[j].Cmd = ''Empty'')$

$Inv2458 \triangleq \forall i \in NODE : (CurCmd = ''ReqE'') \lor \neg(Chan2[i].Cmd = ''Inv'') \lor (ExGntd)$

$Inv4043 \triangleq \forall i \in NODE : (Chan3[i].Cmd = ''Empty'') \lor (CurCmd = ''ReqE'') \lor (CurCmd = ''ReqS'')$

$Inv76 \triangleq \forall i \in NODE : \neg(Chan2[i].Cmd = ''Inv'') \lor \neg(InvSet[i])$

$Inv1103 \triangleq \forall j \in NODE : (CurCmd = ''ReqE'') \lor (InvSet[j]) \lor \neg(Chan2[j].Cmd = ''Empty'')$

$Inv20 \triangleq \forall i \in NODE : (Chan3[i].Cmd = ''Empty'') \lor \neg(Chan2[i].Cmd = ''Inv'')$

$Inv13 \triangleq \forall i \in NODE : (Chan3[i].Cmd = ''Empty'') \lor (Chan3[i].Cmd = ''InvAck'')$

$Inv50 \triangleq \forall i \in NODE : \neg(Cache[i].State = ''S'') \lor \neg(ExGntd)$

$Inv2146 \triangleq \forall i \in NODE : \forall j \in NODE : (Chan3[i].Cmd = ''Empty'') \lor \neg(ExGntd) \lor \neg(Chan2[j].Cmd = ''Inv'')$

$Inv1149 \triangleq \forall j \in NODE : (CurCmd = ''ReqE'') \lor (CurCmd = ''ReqS'') \lor \neg(Chan2[j].Cmd = ''Inv'')$

$Inv957 \triangleq \forall i \in NODE : (ExGntd) \lor (InvSet[i] = ShrSet[i]) \lor \neg(CurCmd = ''ReqS'')$

$Inv43 \triangleq \forall i \in NODE : \neg(Chan2[i].Cmd = ''GntS'') \lor \neg(ExGntd)$

$Inv687 \triangleq \forall i \in NODE : \forall j \in NODE : (Chan3[i].Cmd = ''Empty'') \lor \neg(ExGntd) \lor \neg(InvSet[j])$

$Inv1942 \triangleq \forall i \in NODE : \forall j \in NODE : (Chan3[j].Cmd = ''Empty'') \lor (InvSet[i] = ShrSet[i]) \lor (\neg(ExGntd) \lor \neg(i \neq j))$

$Inv1000 \triangleq \forall i \in NODE : \forall j \in NODE : \neg(Chan2[j].Cmd = ''Inv'') \lor \neg(InvSet[i]) \lor \neg(ExGntd)$

$Inv111 \triangleq \forall i \in NODE : \neg(Chan2[i].Cmd = ''Inv'') \lor \neg(InvSet[i] = ShrSet[i])$

$Inv848 \triangleq \forall i \in NODE : \forall j \in NODE : \neg(Cache[i].State = ''E'') \lor \neg(ShrSet[j]) \lor \neg(i \neq j)$

$Inv23009 \triangleq \forall i \in NODE : \forall j \in NODE : \neg(ExGntd) \lor \neg(InvSet[i]) \lor \neg(InvSet[j]) \lor \neg(i \neq j)$

$Inv7513 \triangleq \forall i, j \in NODE : (InvSet[i] = ShrSet[i]) \lor \neg(ExGntd) \lor \neg(i \neq j) \lor \neg(ShrSet[j])$

Safety $\triangleq \forall i, j \in NODE : (i \neq j) \Rightarrow (Cache[i].State = ''E'' \Rightarrow Cache[j].State = ''I'')$

Figure B.6: Lemma definitions for *GermanCache* inductive proof graph.

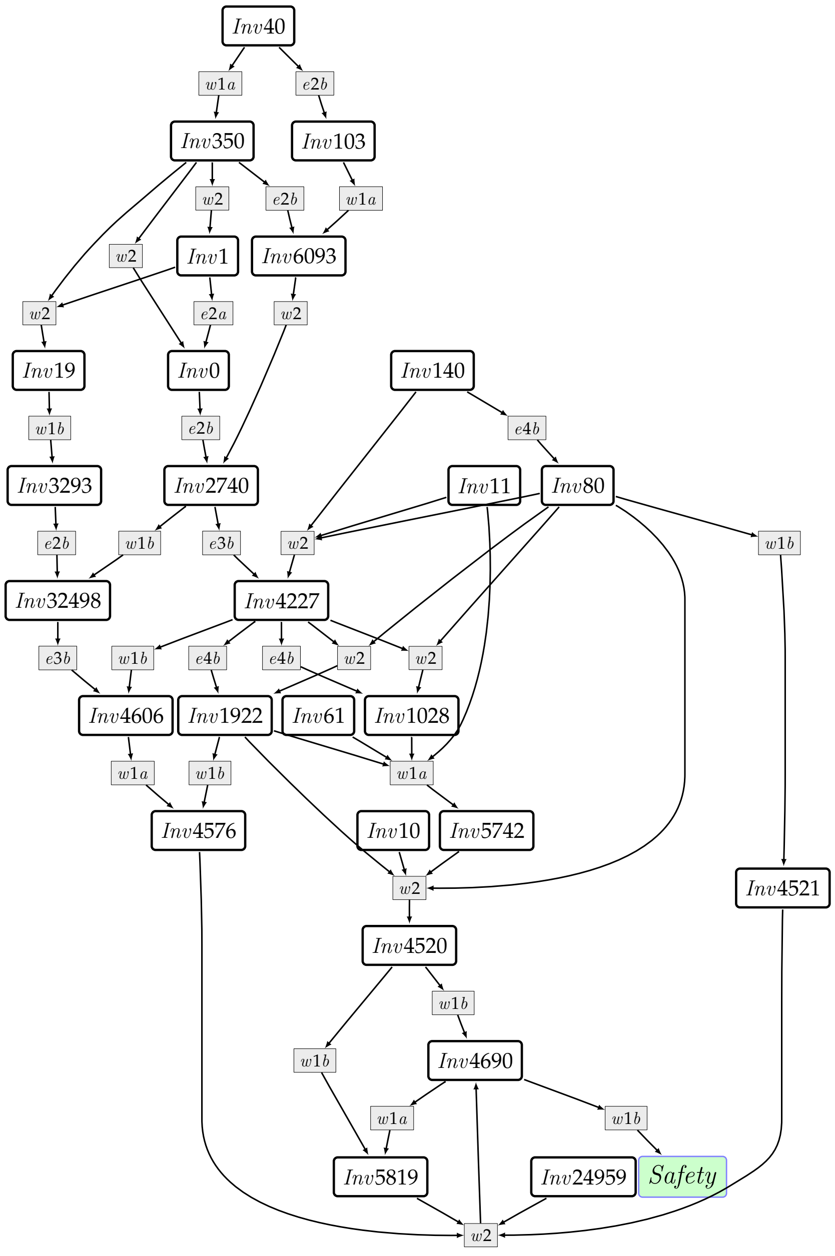


Figure B.7: *Bakery* inductive proof graph for safety property (labeled as *Safety* in green).

$$a \prec b \triangleq (a_1 < b_1) \lor ((a_1 = b_1) \land (a_2 < b_2))$$

$$\begin{aligned}
\mathit{Inv103} &\triangleq \forall j \in \mathit{Procs} : (\mathit{flag}[j]) \lor (\mathit{pc}[j] \neq e3) \\
\mathit{Inv40} &\triangleq \forall i \in \mathit{Procs} : (\mathit{flag}[i]) \lor (\mathit{pc}[i] \neq e2) \\
\mathit{Inv4690} &\triangleq \forall i \in \mathit{Procs} : \forall j \in \mathit{Procs} : (i \in \mathit{unchecked}[j]) \lor \neg(\mathit{pc}[i] = \mathit{cs}) \lor \neg(\mathit{pc}[j] = w1) \\
\mathit{Inv4520} &\triangleq \forall i \in \mathit{Procs} : \forall j \in \mathit{Procs} : (i \in \mathit{unchecked}[j]) \lor \neg\left((\mathit{pc}[j] \in \{w1, w2\} \land \mathit{pc}[i] = w1) \lor \neg(j \in (\mathit{Procs} \setminus \mathit{unchecked}[i]) \setminus \{i\})\right) \\
\mathit{Inv24959} &\triangleq \forall j \in \mathit{Procs} : (\mathit{pc}[j] = e2) \lor (\mathit{unchecked}[j] = \{\}) \lor (\mathit{pc}[j] \in \{w1, w2\}) \\
\mathit{Inv5819} &\triangleq \forall i \in \mathit{Procs} : \forall j \in \mathit{Procs} : (i \in \mathit{unchecked}[j]) \lor \neg(\mathit{pc}[i] = \mathit{cs}) \lor \neg(\mathit{pc}[j] = w2) \\
\mathit{Inv4576} &\triangleq \forall i \in \mathit{Procs} : \forall j \in \mathit{Procs} : (\langle \mathit{num}[i], i \rangle \prec \langle \mathit{num}[j], j \rangle) \lor \neg(\mathit{pc}[i] = \mathit{cs}) \lor \neg(\mathit{pc}[j] = w2) \\
\mathit{Inv4521} &\triangleq \forall i \in \mathit{Procs} : \neg(\mathit{num}[i] = 0) \lor \neg(\mathit{pc}[i] = \mathit{cs}) \\
\mathit{Inv10} &\triangleq \forall j \in \mathit{Procs} : (j \notin \mathit{unchecked}[j]) \\
\mathit{Inv5742} &\triangleq \forall i, j \in \mathit{Procs} : (i \in \mathit{unchecked}[j]) \lor \neg\left(\langle \mathit{num}[i], i \rangle \prec \langle \mathit{num}[j], j \rangle\right) \lor \neg(\mathit{pc}[i] = w2) \lor (\mathit{pc}[j] \neq w2) \\
\mathit{Inv80} &\triangleq \forall i \in \mathit{Procs} : \neg(\mathit{num}[i] = 0) \lor \neg(\mathit{pc}[i] \in \{w1, w2\}) \\
\mathit{Inv1922} &\triangleq \forall i, j \in \mathit{Procs} : (\langle \mathit{num}[i], i \rangle \prec \langle \mathit{num}[j], j \rangle) \lor \neg\left((\mathit{pc}[j] \in \{w1, w2\} \land \mathit{pc}[i] = w1)\right) \lor \neg(j \in (\mathit{Procs} \setminus \mathit{unchecked}[i]) \setminus \{i\}) \\
\mathit{Inv4606} &\triangleq \forall i, j \in \mathit{Procs} : \neg\left(\langle \mathit{num}[i], i \rangle \prec \langle \mathit{num}[j], j \rangle\right) \lor \neg(\mathit{pc}[i] \in \{e4, w1, w2\}) \lor \neg(\mathit{pc}[j] = \mathit{cs}) \\
\mathit{Inv11} &\triangleq \forall i \in \mathit{Procs} : \neg(i \in \mathit{unchecked}[i]) \\
\mathit{Inv61} &\triangleq \forall i \in \mathit{Procs} : (\mathit{nxt}[i] \in \mathit{unchecked}[i]) \lor \neg(\mathit{pc}[i] = w2) \\
\mathit{Inv1028} &\triangleq \forall i, j \in \mathit{Procs} : (i \in \mathit{unchecked}[j]) \lor \neg\left((\mathit{pc}[j] \in \{w1, w2\} \land \mathit{pc}[i] = w1)\right) \lor \neg\left(\langle \mathit{num}[i], i \rangle \prec \langle \mathit{num}[j], j \rangle\right) \\
\mathit{Inv140} &\triangleq \forall i \in \mathit{Procs} : \neg(\mathit{num}[i] = 0) \lor \neg(\mathit{pc}[i] = e4) \\
\mathit{Inv4227} &\triangleq \forall i, j \in \mathit{Procs} : (i \in \mathit{unchecked}[j]) \lor \neg\left(\langle \mathit{num}[i], i \rangle \prec \langle \mathit{num}[j], j \rangle\right) \lor \neg\left(\mathit{pc}[j] \in \{w1, w2\}\right) \lor \neg\left(\mathit{pc}[i] \in \{e4, w1, w2\}\right) \\
\mathit{Inv32498} &\triangleq \forall i, j \in \mathit{Procs} : \neg(\mathit{max}[i] < \mathit{num}[j]) \lor \neg(\mathit{pc}[i] = e3) \lor \neg(\mathit{pc}[j] = \mathit{cs}) \\
\mathit{Inv2740} &\triangleq \forall i, j \in \mathit{Procs} : (i \in \mathit{unchecked}[j]) \lor \neg(\mathit{max}[i] < \mathit{num}[j]) \lor \neg(\mathit{pc}[i] = e3) \lor \neg(\mathit{pc}[j] \in \{w1, w2\}) \\
\mathit{Inv3293} &\triangleq \forall i, j \in \mathit{Procs} : (j \in \mathit{unchecked}[i]) \lor \neg(\mathit{pc}[i] = e2) \lor \neg(\mathit{pc}[j] = \mathit{cs}) \lor \neg(\mathit{max}[i] < \mathit{num}[j]) \\
\mathit{Inv0} &\triangleq \forall i, j \in \mathit{Procs} : \\
&\qquad (\mathit{pc}[j] \in \{w1, w2\}) \Rightarrow \\
&\qquad\quad ((i \in \mathit{unchecked}[j]) \lor (\mathit{pc}[j] = e1 \lor \neg\left(\mathit{max}[i] < \mathit{num}[j] \lor (\mathit{pc}[i] = e2)\right) \quad \lor (\mathit{unchecked}[i] \neq \{\}))) \\
\mathit{Inv6093} &\triangleq \forall i \in \mathit{Procs} : (\mathit{max}[\mathit{nxt}[i]] \geq \mathit{num}[i]) \lor (\mathit{pc}[\mathit{nxt}[i]] \neq e3) \lor \neg(\mathit{pc}[i] = w2) \\
\mathit{Inv19} &\triangleq \forall i, j \in \mathit{Procs} : ((i \in \mathit{unchecked}[j]) \lor \neg(\mathit{pc}[j] = e2) \lor (\mathit{max}[j] \geq \mathit{num}[i]) \lor \neg(\mathit{pc}[i] \in \{w1, w2\})) \lor \neg(\mathit{unchecked}[i] = \{\}) \\
\mathit{Inv1} &\triangleq \forall i, j \in \mathit{Procs} : ((i \in \mathit{unchecked}[j]) \lor \neg(\mathit{pc}[j] = e2) \lor (\mathit{max}[j] \geq \mathit{num}[i]) \lor \neg(\mathit{pc}[i] \in \{w1, w2\})) \lor (j \in \mathit{unchecked}[i]) \\
\mathit{Inv350} &\triangleq \forall i, j \in \mathit{Procs} : ((i \in \mathit{unchecked}[j]) \lor \neg(\mathit{pc}[j] = e2) \lor (\mathit{max}[j] \geq \mathit{num}[i]) \lor \neg(\mathit{pc}[i] \in \{w1, w2\})) \lor \neg(\mathit{nxt}[i] = j) \lor \neg(\mathit{pc}[i] = w2) \\
\textbf{Safety} &\triangleq \forall i, j \in \mathit{Procs} : (i \neq j) \Rightarrow \neg(\mathit{pc}[i] = \text{"}cs\text{"} \land \mathit{pc}[j] = \text{"}cs\text{"})
\end{aligned}$$

Figure B.8: Lemma definitions for *Bakery* inductive proof graph.

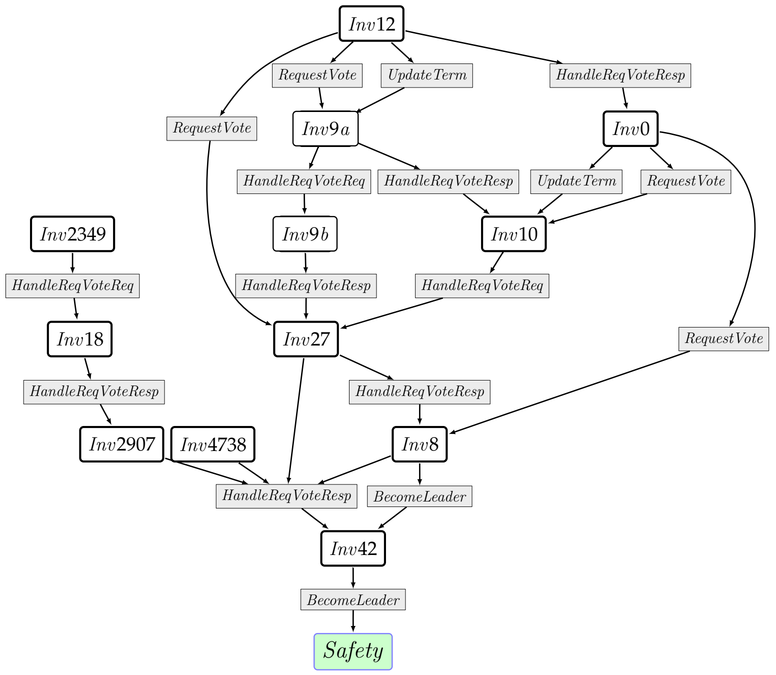


Figure B.9: *AsyncRaft* inductive proof graph for safety property *ElectionSafety* (labeled as *Safety* in green).

$Inv42 \triangleq \forall i, j \in Server :$
$\quad ((state[j] = Leader) \land (votesGranted[i] \in Quorum)) \Rightarrow$
$\quad\quad \neg(state[i] = Candidate \land i \neq j \land currentTerm[i] = currentTerm[j])$

$Inv8 \triangleq \forall s, t \in Server :$
$\quad (s \neq t \land state[s] \in \{Leader, Candidate\} \land state[t] \in \{Leader, Candidate\} \land currentTerm[s] = currentTerm[t]) \Rightarrow$
$\quad\quad (votesGranted[s] \cap votesGranted[t] = \{\})$

$Inv2907 \triangleq \forall i \in Server : (i \in votesGranted[i]) \lor (votesGranted[i] = \{\})$

$Inv4738 \triangleq \forall i \in Server : (state[i] = Leader) \Rightarrow (votesGranted[i] \in Quorum)$

$Inv27 \triangleq \forall s, t \in Server : \forall m \in requestVoteResponseMsgs :$
$\quad (state[s] \in \{Candidate, Leader\} \land t \in votesGranted[s]) \Rightarrow$
$\quad \neg(m.mterm = currentTerm[s] \land m.msource = t \land m.mdest \neq s \land m.mvoteGranted)$

$Inv0 \triangleq \forall i, j \in Server : (currentTerm[i] \leq currentTerm[j]) \lor \neg(state[i] \in \{Leader, Candidate\} \land j \in votesGranted[i])$

$Inv18 \triangleq \forall i \in Server : \forall r \in requestVoteResponseMsgs : \neg(r.mdest = i) \lor \neg(votesGranted[i] = \{\})$

$Inv10 \triangleq \forall i \in Server : (\forall t \in votesGranted[i] : currentTerm[t] = currentTerm[i] \Rightarrow votedFor[t] = i) \lor (state[i] = Follower)$

$Inv9b \triangleq \forall m_i, m_j \in requestVoteResponseMsgs :$
$\quad (m_i.mterm = m_j.mterm \land m_i.msource = m_j.msource \land m_i.mvoteGranted \land m_j.mvoteGranted) \Rightarrow$
$\quad\quad (m_i.mdest = m_j.mdest)$

$Inv12 \triangleq \forall r \in requestVoteResponseMsgs : currentTerm[r.msource] \geq r.mterm$

$Inv2349 \triangleq \forall i \in Server : \forall m \in requestVoteRequestMsgs : (m.msource = i) \Rightarrow (votesGranted[i] \neq \{\})$

$Inv9a \triangleq \forall m \in requestVoteResponseMsgs : m.mtype = RequestVoteResponse \Rightarrow$
$\quad (m.mvoteGranted \land (currentTerm[m.msource] = m.mterm)) \Rightarrow votedFor[m.msource] = m.mdest$

Safety $\triangleq \forall s, t \in Server : (s \neq t \land state[s] = Leader \land state[t] = Leader) \Rightarrow (currentTerm[s] \neq currentTerm[t])$

Figure B.10: Lemma definitions for *AsyncRaft* inductive proof graph for *ElectionSafety*.

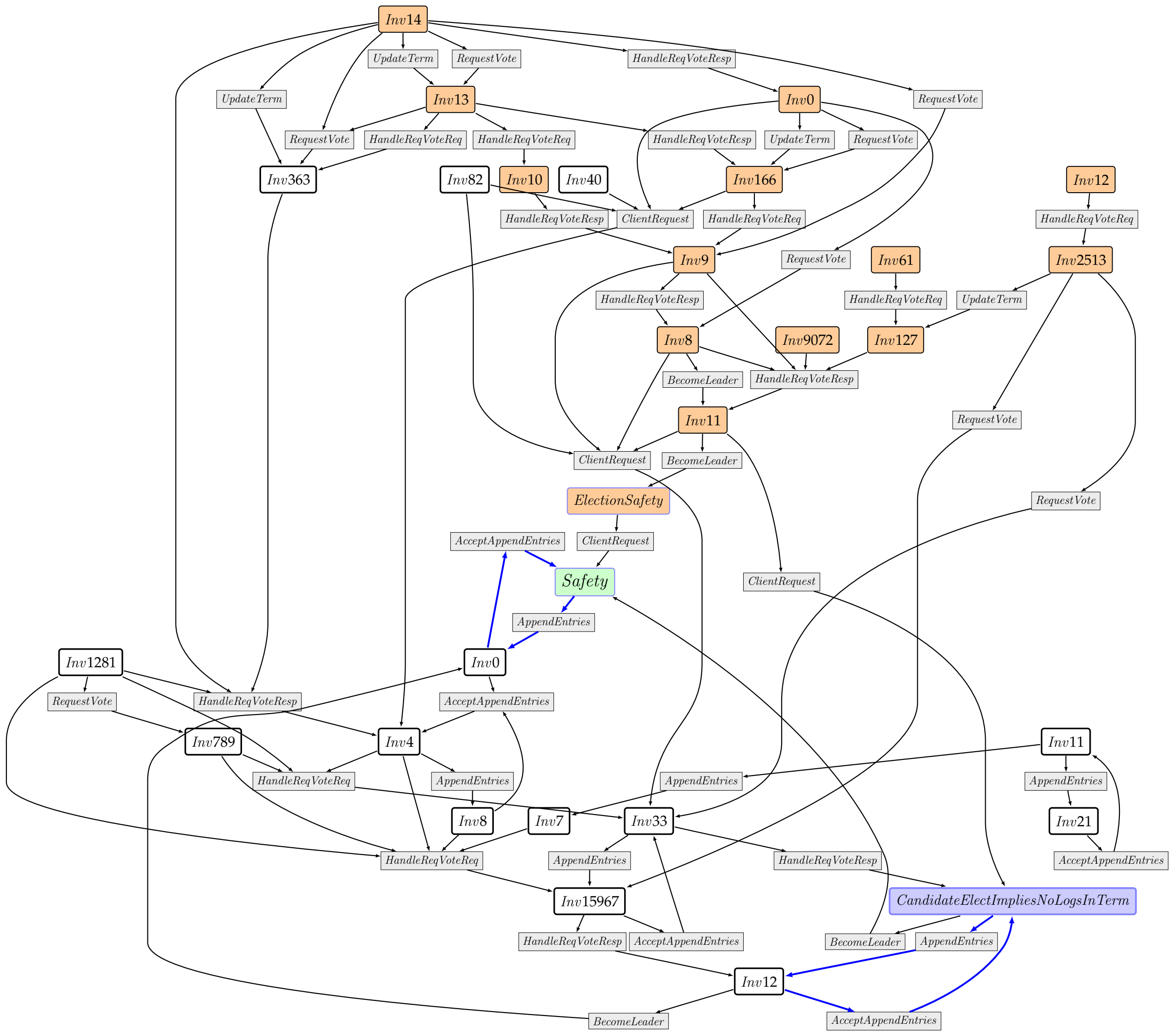


Figure B.11: *AsyncRaft* inductive proof graph for safety property *PrimaryOwnsEntries* (labeled as *Safety* in green).

$$
\begin{aligned}
&\mathit{GrantedVoteSet}(cand) \triangleq \\
&\quad votesGranted[cand] \cup \{s \in Server : \exists m \in requestVoteResponseMsgs : \\
&\quad m.mdest = cand \land m.mvoteGranted \land \\
&\quad m.msource = s \land m.mterm = currentTerm[cand]\}
\end{aligned}
$$

$$
\begin{aligned}
Inv0 \triangleq\ & \forall i \in Server : \forall m \in appendEntriesRequestMsgs : \forall logInd \in LogIndices : \\
& (\land logInd \in DOMAIN\ log[i] \land log[i][logInd] = currentTerm[i]) \\
& \lor \neg (m.mentries \neq \langle\rangle \land m.mentries[1] = currentTerm[i] \land state[i] = Leader) \\
& \lor \neg (logInd = m.mprevLogIndex + 1)
\end{aligned}
$$

$$
\begin{aligned}
&CandidateElectImpliesNoLogsInTerm \triangleq \\
&\quad \forall i, j \in Server : \\
&\qquad \lor \neg (state[i] = Candidate \land i \neq j) \\
&\qquad \lor \neg (votesGranted[i] \in Quorum) \\
&\qquad \lor \neg (\exists k \in DOMAIN\ log[j] : log[j][k] = currentTerm[i])
\end{aligned}
$$

$$
\begin{aligned}
ElectionSafety \triangleq\ & \forall i, j \in Server : \neg (state[i] = Leader \land i \neq j \land currentTerm[i] = currentTerm[j]) \\
& \lor \neg (state[j] = Leader)
\end{aligned}
$$

$$
\begin{aligned}
Inv12 \triangleq\ & \forall i, j \in Server : \forall mae \in appendEntriesRequestMsgs : \\
& \neg (state[i] = Candidate \land i \neq j) \\
& \lor \neg (mae.mentries \neq \langle\rangle \land mae.mentries[1] = currentTerm[i]) \\
& \lor \neg (votesGranted[i] \in Quorum)
\end{aligned}
$$

$$
\begin{aligned}
Inv11 \triangleq\ & \forall i, j \in Server : \neg (state[i] = Candidate \land i \neq j \land currentTerm[i] = currentTerm[j]) \\
& \lor \neg (state[j] = Leader) \\
& \lor \neg (votesGranted[i] \in Quorum)
\end{aligned}
$$

$$
\begin{aligned}
Inv33 \triangleq\ & \forall i, j \in Server : \\
& \neg (state[i] = Candidate \land i \neq j) \\
& \lor \neg (GrantedVoteSet(i) \in Quorum) \\
& \lor \neg (\exists k \in DOMAIN\ log[j] : log[j][k] = currentTerm[i])
\end{aligned}
$$

$$
\begin{aligned}
Inv15967 \triangleq\ & \forall i, j \in Server : \forall mae \in appendEntriesRequestMsgs : \\
& \neg (state[i] = Candidate \land i \neq j) \\
& \lor \neg (GrantedVoteSet(i) \in Quorum) \\
& \lor \neg (mae.mentries \neq \langle\rangle \land mae.mentries[1] = currentTerm[i])
\end{aligned}
$$

$$
\begin{aligned}
Inv8 \triangleq\ & \forall s, t \in Server : \\
& (s \neq t \land state[s] \in \{Leader, Candidate\} \land state[t] \in \{Leader, Candidate\} \land currentTerm[s] = currentTerm[t]) \\
& \Rightarrow (votesGranted[s] \cap votesGranted[t] = \{\})
\end{aligned}
$$

$$
\begin{aligned}
Inv127 \triangleq\ & \forall i, j \in Server : j \in votesGranted[j] \\
& \lor \neg (i \in GrantedVoteSet(j))
\end{aligned}
$$

$$
\begin{aligned}
Inv9072 \triangleq\ & \forall i, j \in Server : state[j] = Follower \\
& \lor votesGranted[i] \in Quorum \\
& \lor \neg (state[i] = Leader)
\end{aligned}
$$

$$
\begin{aligned}
Inv9 \triangleq\ & \forall s, t \in Server : \forall m \in requestVoteResponseMsgs : (\land state[s] \in \{Candidate, Leader\} \land t \in votesGranted[s]) \\
& \Rightarrow \neg (\land m.mterm = currentTerm[s] \land m.msource = t \land m.mdest \neq s \land m.mvoteGranted)
\end{aligned}
$$

$$
Inv82 \triangleq \forall i \in Server : (state[i] = Leader) \Rightarrow votesGranted[i] \in Quorum
$$

Figure B.12: Lemma definitions for *AsyncRaft* inductive proof graph for *PrimaryOwnsEntries*.

$$
\begin{array}{rl}
\mathit{Inv789} \triangleq & \forall i \in \mathit{Server} : \mathit{state}[i] = \mathit{Follower} \\
& \lor (\mathit{votedFor}[i] \neq \mathit{Nil} \land i \in \mathit{votesGranted}[\mathit{votedFor}[i]]) \\
\mathit{Inv1281} \triangleq & \forall i \in \mathit{Server} : (\mathit{state}[i] \neq \mathit{Follower}) \Rightarrow \mathit{votedFor}[i] = i \\
\mathit{Inv4} \triangleq & \\
& \forall s \in \mathit{Server} : \forall i \in \mathit{DOMAIN}\ \mathit{log}[s] : \\
& \exists Q \in \mathit{Quorum} : \exists u \in \mathit{Server} : \\
& \land\ \mathit{currentTerm}[u] \geq \mathit{log}[s][i] \\
& \land\ (\mathit{currentTerm}[u] = \mathit{log}[s][i] \Rightarrow (\mathit{state}[u] = \mathit{Leader} \land \mathit{votesGranted}[u] = Q)) \\
& \land\ \forall n \in Q : \\
& \quad \land\ \mathit{currentTerm}[n] \geq \mathit{log}[s][i] \\
& \quad \land\ (\mathit{currentTerm}[n] = \mathit{log}[s][i] \Rightarrow \mathit{votedFor}[n] = u) \\
\mathit{Inv2513} \triangleq & \forall m \in \mathit{requestVoteResponseMsgs} : \mathit{currentTerm}[m.\mathit{mdest}] \geq m.\mathit{mterm} \\
& \lor \neg (m.\mathit{mvoteGranted}) \\
\mathit{Inv7} \triangleq & \forall i, j \in \mathit{Server} : \forall \mathit{mae} \in \mathit{appendEntriesRequestMsgs} : \mathit{currentTerm}[j] > \mathit{currentTerm}[i] \\
& \lor \neg (\mathit{mae}.\mathit{mentries} \neq \langle\rangle \land \mathit{mae}.\mathit{mentries}[1] > \mathit{mae}.\mathit{mterm}) \\
\mathit{Inv0} \triangleq & \forall i, j \in \mathit{Server} : \mathit{currentTerm}[i] \leq \mathit{currentTerm}[j] \\
& \lor \neg (\mathit{state}[i] \in \{\mathit{Leader}, \mathit{Candidate}\} \land j \in \mathit{votesGranted}[i]) \\
\mathit{Inv61} \triangleq & \forall i \in \mathit{Server} : \forall m \in \mathit{requestVoteRequestMsgs} : \neg (m.\mathit{msource} = i) \lor (\mathit{votesGranted}[i] \neq \{\}) \\
\mathit{Inv166} \triangleq & \forall i \in \mathit{Server} : (\forall t \in \mathit{votesGranted}[i] : \mathit{currentTerm}[t] = \mathit{currentTerm}[i] \Rightarrow \mathit{votedFor}[t] = i) \\
& \lor\ \mathit{state}[i] = \mathit{Follower} \\
\mathit{Inv10} \triangleq & \forall \mathit{mi}, \mathit{mj} \in \mathit{requestVoteResponseMsgs} : \\
& (\land \mathit{mi}.\mathit{mterm} = \mathit{mj}.\mathit{mterm} \land \mathit{mi}.\mathit{msource} = \mathit{mj}.\mathit{msource} \land \mathit{mi}.\mathit{mvoteGranted} \land \mathit{mj}.\mathit{mvoteGranted}) \\
& \Rightarrow \mathit{mi}.\mathit{mdest} = \mathit{mj}.\mathit{mdest} \\
\mathit{Inv14} \triangleq & \forall m \in \mathit{requestVoteResponseMsgs} : \mathit{currentTerm}[m.\mathit{msource}] \geq m.\mathit{mterm} \\
\mathit{Inv8} \triangleq & \forall m \in \mathit{appendEntriesRequestMsgs} : (m.\mathit{mentries} \neq \langle\rangle) \Rightarrow \\
& \exists Q \in \mathit{Quorum} : \exists u \in \mathit{Server} : \\
& \land\ \mathit{currentTerm}[u] \geq m.\mathit{mentries}[1] \\
& \land\ (\mathit{currentTerm}[u] = m.\mathit{mentries}[1] \Rightarrow \mathit{state}[u] = \mathit{Leader}) \\
& \land\ \forall n \in Q : \mathit{currentTerm}[n] \geq m.\mathit{mentries}[1] \land (\mathit{currentTerm}[n] = m.\mathit{mentries}[1] \Rightarrow \mathit{votedFor}[n] = u) \\
\mathit{Inv40} \triangleq & \forall i \in \mathit{Server} : \mathit{state}[i] = \mathit{Follower} \\
& \lor\ i \in \mathit{votesGranted}[i] \\
\mathit{Inv363} \triangleq & \forall i \in \mathit{Server} : (\forall m \in \mathit{requestVoteResponseMsgs} : m.\mathit{mtype} = \mathit{RequestVoteResponse} \Rightarrow \\
& (m.\mathit{mvoteGranted} \land \mathit{currentTerm}[m.\mathit{msource}] = m.\mathit{mterm}) \Rightarrow \mathit{votedFor}[m.\mathit{msource}] = m.\mathit{mdest}) \\
& \lor \neg (\mathit{votedFor}[i] \neq \mathit{Nil}) \\
\mathit{Inv12} \triangleq & \forall m \in \mathit{requestVoteRequestMsgs} : \mathit{currentTerm}[m.\mathit{msource}] \geq m.\mathit{mterm} \\
\mathit{Inv11} \triangleq & \forall i \in \mathit{Server} : \forall k \in \mathit{DOMAIN}\ \mathit{log}[i] : \mathit{log}[i][k] \leq \mathit{currentTerm}[i] \\
\mathit{Inv13} \triangleq & \forall m \in \mathit{requestVoteResponseMsgs} : m.\mathit{mtype} = \mathit{RequestVoteResponse} \Rightarrow \\
& (m.\mathit{mvoteGranted} \land \mathit{currentTerm}[m.\mathit{msource}] = m.\mathit{mterm}) \Rightarrow \mathit{votedFor}[m.\mathit{msource}] = m.\mathit{mdest} \\
\mathit{Inv21} \triangleq & \forall \mathit{mae} \in \mathit{appendEntriesRequestMsgs} : \neg (\mathit{mae}.\mathit{mentries} \neq \langle\rangle \land \mathit{mae}.\mathit{mentries}[1] > \mathit{mae}.\mathit{mterm}) \\
\textbf{Safety} \triangleq & \forall i, j \in \mathit{Server} : \\
& (\mathit{state}[i] = \mathit{Leader}) \Rightarrow \\
& \quad \neg(\exists k \in \mathit{DOMAIN}\ \mathit{log}[j] : \\
& \qquad \land\ \mathit{log}[j][k] = \mathit{currentTerm}[i] \\
& \qquad \land\ \nexists \mathit{ind} \in \mathit{DOMAIN}\ \mathit{log}[i] : (\mathit{ind} = k \land \mathit{log}[i][k] = \mathit{log}[j][k]))
\end{array}
$$

Figure B.13: Additional lemma definitions for *AsyncRaft* inductive proof graph for *Primary-OwnsEntries*.

Bibliography

[1] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services Uses Formal Methods. *Commun. ACM*, 58(4):66–73, March 2015.

[2] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1493–1509. Association for Computing Machinery, 2020.

[3] William Schultz, Tess Avitabile, and Alyson Cabral. Tunable Consistency in MongoDB. *Proc. VLDB Endow.*, 12(12):2071–2081, August 2019.

[4] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Dale Woodford, Yasushi Saito, Christopher Taylor, Michal Szymaniak, and Ruth Wang. Spanner: Google's Globally-Distributed Database. In *OSDI*, 2012.

[5] Sean Braithwaite, Ethan Buchman, Igor Konnov, Zarko Milosevic, Ilina Stoilkovska, Josef Widder, and Anca Zamfir. Formal Specification and Model Checking of the Tendermint Blockchain Synchronization Protocol (Short Paper). In Bruno Bernardo and Diego Marmsoler, editors, *2nd Workshop on Formal Methods for Blockchains (FMBC 2020)*, volume 84 of *OpenAccess Series in Informatics (OASIcs)*, pages 10:1–10:8, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[6] Petr Kuznetsov and Andrei Tonkikh. Asynchronous Reconfiguration with Byzantine Failures. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing (DISC 2020)*, volume 179 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[7] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. TiDB: A Raft-Based HTAP Database. *Proc. VLDB Endow.*, 13(12):3072–3084, aug 2020.

[8] George Pîrlea. Protocol bugs list. https://github.com/dranov/protocol-bugs-list, 2020. Accessed: 2024-09-17.

[9] Diego Ongaro. Bug in single-server membership changes. https://groups.google.com/g/raft-dev/c/t4xj6dJTP6E/m/d2D9LrWRza8J, jul 2015.

[10] Pierre Sutra. On the correctness of Egalitarian Paxos. https://arxiv.org/abs/1906.10917, 2019.

[11] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag, Berlin, Heidelberg, 2007.

[12] Robert Beers. Pre-RTL formal verification: An Intel experience. In *2008 45th ACM/IEEE Design Automation Conference*, pages 806–811, 2008.

[13] Elastic. Formal models of core elasticsearch algorithms, 2024. Accessed: October 10, 2024.

[14] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys (CSUR)*, 1990.

[15] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live: An Engineering Perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, page 398–407, New York, NY, USA, 2007. Association for Computing Machinery.

[16] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment*, 2020.

[17] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98(2):142–170, jun 1992.

[18] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[19] Armin Biere, Alessandro Cimatti, Edmund Clarke, Ofer Strichman, and Yunshan Zhu. Bounded Model Checking. volume 58, pages 117–148, 12 2003.

[20] Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. Decidability of Parameterized Verification. *Synthesis Lectures on Distributed Computing Theory*, 6(1):1–170, 2015.

[21] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. Planning for Change in a Formal Verification of the Raft Consensus Protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, page 154–165. Association for Computing Machinery, 2016.

[22] Saksham Chand, Yanhong A Liu, and Scott D Stoller. Formal Verification of Multi-Paxos for Distributed Consensus. In *International Symposium on Formal Methods*, pages 119–136. Springer, 2016.

[23] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In David Grove and Stephen M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 357–368. ACM, 2015.

[24] William Schultz, Ian Dardik, and Stavros Tripakis. Formal Verification of a Distributed Dynamic Reconfiguration Protocol. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2022, page 143–152, Philadelphia, PA, USA, 2022. Association for Computing Machinery.

[25] Oded Padon, Neil Immerman, Sharon Shoham, Aleksandr Karbyshev, and Mooly Sagiv. Decidability of Inferring Inductive Invariants. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 217–231. ACM, 2016.

[26] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos Made EPR: Decidable Reasoning about Distributed Protocols. *Proc. ACM Program. Lang.*, 1(OOPSLA), Oct 2017.

[27] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 614–630, New York, NY, USA, 2016. Association for Computing Machinery.

[28] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[29] Haojun Ma, Aman Goel, Jean Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. I4: Incremental Inference of Inductive Invariants for Verification of Distributed Protocols. In *SOSP 2019 - Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019.

[30] Jason R. Koenig, Oded Padon, Neil Immerman, and Alex Aiken. First-Order Quantified Separators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 703–717. Association for Computing Machinery, 2020.

[31] Aman Goel and Karem Sakallah. On Symmetry and Quantification: A New Approach to Verify Distributed Protocols. In *NASA Formal Methods Symposium*, pages 131–150. Springer, 2021.

[32] Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. Finding Invariants of Distributed Systems: It's a Small (Enough) World After All. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 115–131. USENIX Association, April 2021.

[33] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 405–421. USENIX Association, July 2021.

[34] mypyvy tool, github repository. `https://github.com/wilcoxjay/mypyvy`, 2022.

[35] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Jun 2002.

[36] Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006.

[37] Tamarah Arons, Amir Pnueli, Sitvanit Ruah, Ying Xu, and Lenore Zuck. Parameterized verification with automatically computed inductive assertions? In *International Conference on Computer Aided Verification*, pages 221–234. Springer, 2001.

[38] Aman Goel and Karem Sakallah. On Symmetry and Quantification: A New Approach to Verify Distributed Protocols. In *NASA Formal Methods: 13th International Symposium, NFM 2021, Virtual Event, May 24–28, 2021, Proceedings*, page 131–150, Berlin, Heidelberg, 2021. Springer-Verlag.

[39] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety Verification by Interactive Generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 614–630, New York, NY, USA, 2016. Association for Computing Machinery.

[40] Jason R. Koenig, Oded Padon, Sharon Shoham, and Alex Aiken. Inferring Invariants with Quantifier Alternations: Taming the Search Space Explosion. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 338–356, Cham, 2022. Springer International Publishing.

[41] Aman Goel and Karem A. Sakallah. Towards an Automatic Proof of Lamport's Paxos. *2021 Formal Methods in Computer Aided Design (FMCAD)*, pages 112–122, 2021.

[42] Aleksandr Karbyshev, Nikolaj Bjørner, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. Property-Directed Inference of Universal Invariants or Proving Their Absence. *J. ACM*, 64(1), mar 2017.

[43] Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. Finding Invariants of Distributed Systems: It's a Small (Enough) World After All. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 115–131. USENIX Association, April 2021.

[44] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. DuoAI: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 485–501. USENIX Association, 2022.

[45] William Schultz, Ian Dardik, and Stavros Tripakis. Plain and Simple Inductive Invariant Inference for Distributed Protocols in TLA+. In *2022 Formal Methods in Computer-Aided Design (FMCAD)*, pages 273–283. IEEE, 2022.

[46] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

[47] Oded Padon, Neil Immerman, Sharon Shoham, Aleksandr Karbyshev, and Mooly Sagiv. Decidability of Inferring Inductive Invariants. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 217–231, New York, NY, USA, 2016. Association for Computing Machinery.

[48] Sean Braithwaite, Ethan Buchman, Igor Konnov, Zarko Milosevic, Ilina Stoilkovska, Josef Widder, and Anca Zamfir. Formal Specification and Model Checking of the Tendermint Blockchain Synchronization Protocol (Short Paper). In *2nd Workshop on Formal Methods for Blockchains (FMBC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[49] Haojun Ma, Hammad Ahmad, Aman Goel, Eli Goldweber, Jean-Baptiste Jeannin, Manos Kapritsos, and Baris Kasikci. Sift: Using Refinement-guided Automation to Verify Complex Distributed Systems. In *USENIX Annual Technical Conference*, 2022.

[50] William Schultz, Siyuan Zhou, Ian Dardik, and Stavros Tripakis. Design and Analysis of a Logless Dynamic Reconfiguration Protocol. In Quentin Bramas, Vincent Gramoli, and Alessia Milani, editors, *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*, volume 217 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:16, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[51] Chris Newcombe. Why Amazon Chose TLA+. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 25–39. Springer, 2014.

[52] William Schultz, Edward Ashton, Heidi Howard, and Stavros Tripakis. Scalable, Interpretable Distributed Protocol Verification by Inductive Proof Slicing. `https://arxiv.org/abs/2404.18048`, 2024.

[53] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, Berlin, Heidelberg, 1995.

[54] William Schultz. MongoRaftReconfig TLA+ Specifications. `https://doi.org/10.5281/zenodo.5715511`, November 2021.

[55] Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 1998.

[56] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, USA, 2014. USENIX Association.

[57] Marcos Aguilera, Idit Keidar, Dahlia Malkhi, Jean-Philippe Martin, and Alexander Shraer. Reconfiguring Replicated Atomic Storage: A Tutorial. *Bulletin of the European Association for Theoretical Computer Science EATCS*, 2010.

[58] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Stoppable paxos. *TechReport, Microsoft Research*, 2008.

[59] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 312–313. ACM, 2009.

[60] MongoDB Github Project, 2021.

[61] Siyuan Zhou and Shuai Mu. Fault-Tolerant replication with Pull-Based consensus in MongoDB. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 687–703. USENIX Association, April 2021.

[62] Stephan Merz. *The Specification Language TLA+*, pages 401–451. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[63] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA+ specifications. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 54–66. Springer, 1999.

[64] Diego Ongaro. Consensus: Bridging Theory and Practice. *Doctoral thesis*, 2014.

[65] Diego Ongaro. The Raft Consensus Algorithm, 2021.

[66] Alexander Shraer, Benjamin Reed, Dahlia Malkhi, and Flavio Junqueira. Dynamic reconfiguration of primary/backup clusters. In *Proceedings of the 2012 USENIX Annual Technical Conference, USENIX ATC 2012*, 2019.

[67] Edmund M. Clarke, Orna Grumberg, Doron A. Peled, and Helmut Veith. Model checking. In *Handbook of Model Checking*, pages 353–415. Springer, 2011.

[68] William Schultz. MongoDB Experiment Source Code. `https://github.com/will62794/mongo/tree/2bb9f30da`, September 2021.

[69] MongoDB JIRA Ticket SERVER-46907, 2020.

[70] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernan Vanzetto. TLA+ Proofs. *Proceedings of the 18th International Symposium on Formal Methods (FM 2012), Dimitra Giannakopoulou and Dominique Mery, editors. Springer-Verlag Lecture Notes in Computer Science*, 7436:147–154, January 2012.

[71] William Schultz and Ian Dardik. TLAPS Safety Proof of MongoRaftReconfig. `https://doi.org/10.5281/zenodo.5768484`, December 2021.

[72] Leslie Lamport. How to Write a Proof. *The American Mathematical Monthly*, 102(7):600–608, 1995.

[73] Makarius Wenzel, Lawrence C Paulson, and Tobias Nipkow. The isabelle framework. In *International Conference on Theorem Proving in Higher Order Logics*, pages 33–38. Springer, 2008.

[74] Richard Bonichon, David Delahaye, and Damien Doligez. Zenon: An extensible automated theorem prover producing checkable proofs. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 151–165. Springer, 2007.

[75] Marko Vukolić et al. The origin of quorum systems. *Bulletin of EATCS*, 2(101), 2013.

[76] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. *Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3*, page 836–850. Association for Computing Machinery, New York, NY, USA, 2021.

[77] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17, 2015.

[78] A. Jesse Jiryu Davis, Max Hirschhorn, and Judah Schvimer. Extreme Modelling in Practice. *Proc. VLDB Endow.*, 13(9):1346–1358, may 2020.

[79] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. Springer Science & Business Media, 2012.

[80] Leslie Lamport and Lawrence C Paulson. Should your specification language be typed. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3):502–526, 1999.

[81] Leslie Lamport. Using TLC to Check Inductive Invariance. https://lamport.azurewebsites.net/tla/inductive-invariant.pdf, 2018.

[82] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model Checking TLA+ Specifications. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, pages 54–66, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[83] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. TLA+ Model Checking Made Symbolic. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019.

[84] Iulian Moraru, David G Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 358–372, 2013.

[85] Leslie Lamport. Paxos Made Simple. *ACM SIGACT News*, 2001.

[86] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical Paxos and Primary-Backup Replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC '09, pages 312–313, New York, NY, USA, 2009. Association for Computing Machinery.

[87] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. Planning for change in a formal verification of the raft consensus protocol. In *CPP 2016 - Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, co-located with POPL 2016*, 2016.

[88] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[89] Brandon Amos and Huanchen Zhang. Specifying and proving cluster membership for the Raft distributed consensus algorithm, 2015.

[90] Seth Gilbert, Nancy A. Lynch, and Alexander A. Shvartsman. Rambo: A robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 2010.

[91] Michael Whittaker, Neil Giridharan, Adriana Szekeres, Joseph M Hellerstein, Heidi Howard, Faisal Nawab, and Ion Stoica. Matchmaker Paxos: A Reconfigurable Consensus Protocol [Technical Report], 2020.

[92] Leander Jehl and Hein Meling. Asynchronous Reconfiguration for Paxos State Machines. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2014.

[93] Denis Rystsov. CASPaxos: Replicated State Machines without logs, 2018.

[94] Leslie Lamport. Byzantizing Paxos by refinement. In *International Symposium on Distributed Computing*, pages 211–224. Springer, 2011.

[95] Noran Azmy, Stephan Merz, and Christoph Weidenbach. A Rigorous Correctness Proof for Pastry. In Michael Butler, Klaus-Dieter Schewe, Atif Mashkoor, and Miklós Biró, editors, *5th Intl. Conf. Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ 2016)*, volume 9675 of *LNCS*, pages 86–101. Springer, 2016.

[96] Ruzica Piskac, Leonardo de Moura, and Nikolaj Bjørner. Deciding effectively propositional logic using DPLL and substitution sets. *Journal of Automated Reasoning*, 44(4):401–424, 2010.

[97] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos Made EPR: Decidable Reasoning about Distributed Protocols. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017.

[98] Aman Goel and Karem A. Sakallah. Towards an Automatic Proof of Lamport's Paxos. In Ruzica Piskac and Michael W Whalen, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, pages 112–122, New Haven, Connecticut, October 2021.

[99] Leslie Lamport. A Science of Concurrent Programs. https://lamport.azurewebsites.net/tla/science-book.html, 2024.

[100] William Schultz, Ian Dardik, and Stavros Tripakis. Artifact for FMCAD 2022 paper: Plain and Simple Inductive Invariant Inference for Distributed Protocols in TLA+. https://doi.org/10.5281/zenodo.6994922, August 2022.

[101] William Schultz. endive: Tool for automatically inferring inductive invariants of distributed protocols. https://github.com/will62794/endive, 2022.

[102] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 836–850. Association for Computing Machinery, 2021.

[103] Aaron R Bradley. SAT-based model checking without unrolling. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011.

[104] Grigory Fedyukovich and Rastislav Bodík. Accelerating Syntax-Guided Invariant Synthesis. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 251–269, Cham, 2018. Springer International Publishing.

[105] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. Quantified Invariants via Syntax-Guided Synthesis. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 259–277, Cham, 2019. Springer International Publishing.

[106] William Wernick. Complete sets of logical functions. *Transactions of the American Mathematical Society*, 51:117–132, 1942.

[107] Leslie Lamport. Using TLC to Check Inductive Invariance. http://lamport.azurewebsites.net/tla/inductive-invariant.pdf, 2018.

[108] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. Efficient Sampling of SAT Solutions for Testing. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 549–559. Association for Computing Machinery, 2018.

[109] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. TLA+ Model Checking Made Symbolic. *Proc. ACM Program. Lang.*, 3(OOPSLA), Oct 2019.

[110] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[111] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.

[112] V. Chvatal. A greedy heuristic for the set-covering problem. *Math. Oper. Res.*, 4(3):233–235, aug 1979.

[113] Stephan Merz and Hernán Vanzetto. Encoding TLA+ into Many-Sorted First-Order Logic. In Michael Butler, Klaus-Dieter Schewe, Atif Mashkoor, and Miklos Biro, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 54–69, Cham, 2016. Springer International Publishing.

[114] Markus A Kuppe. A Verified and Scalable Hash Table for the TLC Model Checker: Towards an Order of Magnitude Speedup. Master's thesis, University of Hamburg., 2017.

[115] Parosh Abdulla, Frédéric Haziza, and Lukáš Holík. Parameterized verification through view abstraction. *Int. J. Softw. Tools Technol. Transf.*, 18(5):495–516, Oct 2016.

[116] Ahmed Bouajjani, Peter Habermehl, and Tomáš Vojnar. Abstract Regular Model Checking. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification*, pages 372–386, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[117] Silvio Ghilardi and Silvio Ranise. MCMT: A Model Checker modulo Theories. In *Proceedings of the 5th International Conference on Automated Reasoning*, IJCAR'10, page 22–29, Berlin, Heidelberg, 2010. Springer-Verlag.

[118] Cormac Flanagan and K. Rustan M. Leino. Houdini, an Annotation Assistant for ESC/Java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, FME '01, page 500–517, Berlin, Heidelberg, 2001. Springer-Verlag.

[119] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007.

[120] Rahul Sharma and Alex Aiken. From Invariant Checking to Invariant Inference Using Randomized Search. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 88–105. Springer, 2014.

[121] John Rushby. Verification Diagrams Revisited: Disjunctive Invariants for Easy Verification. In E. Allen Emerson and Aravinda Prasad Sistla, editors, *Computer Aided Verification*, pages 508–520, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[122] William Schultz. Scimitar: Verification tool for distributed protocols based on inductive proof decomposition. https://github.com/will62794/scimitar, 2024. GitHub repository.

[123] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8, 2013.

[124] Mark H. Liffiton and Karem A. Sakallah. Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints. *J. Autom. Reason.*, 40(1):1–33, jan 2008.

[125] Mark H. Liffiton, Alessandro Previti, Ammar Malik, and Joao Marques-Silva. Fast, Flexible MUS Enumeration. *Constraints*, 21(2):223–250, apr 2016.

[126] Jim Gray. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course*, page 393–481, Berlin, Heidelberg, 1978. Springer-Verlag.

[127] Leslie Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Commun. ACM*, 17(8):453–455, aug 1974.

[128] Aman Goel, Stephan Merz, and Karem A. Sakallah. Towards an Automatic Proof of the Bakery Algorithm. In Marieke Huisman and António Ravara, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 21–28, Cham, 2023. Springer Nature Switzerland.

[129] Ching-Tsun Chou, Phanindra K Mannava, and Seungjoon Park. A simple method for parameterized verification of cache coherence protocols. In *Formal Methods in Computer-Aided Design: 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004. Proceedings 5*, pages 382–398. Springer, 2004.

[130] Jack Vanlightly. raft-tlaplus: A TLA+ specification of the Raft distributed consensus algorithm. https://github.com/Vanlightly/raft-tlaplus/blob/main/specifications/standard-raft/Raft.tla, 2023. GitHub repository.

[131] Tony Nuda Zhang, Travis Hance, Manos Kapritsos, Tej Chajed, and Bryan Parno. Inductive Invariants That Spark Joy: Using Invariant Taxonomies to Streamline Distributed Protocol Proofs. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 837–853, Santa Clara, CA, July 2024. USENIX Association.

[132] A. Gurfinkel and Alexander Ivrii. Pushing to the top. *2015 Formal Methods in Computer-Aided Design (FMCAD)*, pages 65–72, 2015.

[133] Ryan Berryhill, Alexander Ivrii, Neil Veira, and Andreas Veneris. Learning support sets in IC3 and Quip: The good, the bad, and the ugly. In *2017 Formal Methods in Computer Aided Design (FMCAD)*, pages 140–147. IEEE, 2017.

[134] Marcelo Taube, Giuliano Losa, Kenneth L McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R Wilcox, and Doug Woos. Modularity for decidability of deductive verification with applications to distributed systems. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 662–677, 2018.

[135] Jamieson M Cobleigh, Dimitra Giannakopoulou, and Corina S Păsăreanu. Learning assumptions for compositional verification. In *Tools and Algorithms for the Construction and Analysis of Systems: 9th International Conference, TACAS 2003 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003 Warsaw, Poland, April 7–11, 2003 Proceedings 9*, pages 331–346. Springer, 2003.

[136] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification*, pages 521–525, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[137] K.L. McMillan. A methodology for hardware verification using compositional model checking. *Science of Computer Programming*, 37(1):279–309, 2000.

[138] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Inductive Data Flow Graphs. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 129–142. ACM, 2013.

[139] Zachary Kincaid. *Parallel Proofs for Parallel Programs*. PhD thesis, University of Toronto, November 2016.

[140] Kenneth L McMillan. Lazy Abstraction with Interpolants. In *Computer Aided Verification: 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings 18*, pages 123–136. Springer, 2006.

[141] Michael J. Gordon, Matt Kaufmann, and Sandip Ray. The Right Tools for the Job: Correctness of Cone of Influence Reduction Proved Using ACL2 and HOL4. *J. Autom. Reason.*, 47(1):1–16, jun 2011.

[142] Frank Tip. A survey of program slicing techniques. *J. Program. Lang.*, 3, 1994.

[143] Joxan Jaffar, Vijayaraghavan Murali, Jorge A Navas, and Andrew E Santosa. Path-sensitive backward slicing. In *Static Analysis: 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings 19*, pages 231–247. Springer, 2012.

[144] Ranjit Jhala and Rupak Majumdar. Path slicing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming language Design and Implementation*, pages 38–47, 2005.

[145] Xiaosong Gu, Wei Cao, Yicong Zhu, Xuan Song, Yu Huang, and Xiaoxing Ma. Compositional Model Checking of Consensus Protocols via Interaction-Preserving Abstraction. In *2022 41st International Symposium on Reliable Distributed Systems (SRDS)*, pages 82–93. IEEE, 2022.

[146] George Chatzieleftheriou, Borzoo Bonakdarpour, Scott A. Smolka, and Panagiotis Katsaros. Abstract Model Repair. In Alwyn E. Goodloe and Suzette Person, editors, *NASA Formal Methods*, pages 341–355, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[147] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE, 2013.

[148] Yalin Ke, Kathryn T Stolee, Claire Le Goues, and Yuriy Brun. Repairing Programs with Semantic Code Search (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 295–306. IEEE, 2015.

[149] Ali Ebnenasir. Synthesizing self-stabilizing parameterized protocols with unbounded variables. In *2022 Formal Methods in Computer-Aided Design (FMCAD)*, pages 245–254, 2022.

[150] Ali Ebnenasir, Sandeep S Kulkarni, and Anish Arora. Ftsyn: A framework for automatic synthesis of fault-tolerance. *International Journal on Software Tools for Technology Transfer*, 10:455–471, 2008.

[151] Rajeev Alur and Stavros Tripakis. Automatic synthesis of distributed protocols. *Acm Sigact News*, 48(1):55–90, 2017.

[152] Daniel Egolf, William Schultz, and Stavros Tripakis. Efficient Synthesis of Symbolic Distributed Protocols by Sketching. In Nina Narodytska and Philipp Rümmer, editors, *Proceedings of the 24th Conference on Formal Methods in Computer-Aided Design (FMCAD 2024)*, pages 281–291. TU Wien Academic Press, 2024.

[153] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Adv. Comput.*, 58:117–148, 2003.

[154] Emina Torlak and Daniel Jackson. Kodkod: A Relational Model Finder. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647. Springer, 2007.

[155] Aleksandar Milicevic, Joseph P Near, Eunsuk Kang, and Daniel Jackson. Alloy*: A general-purpose higher-order relational constraint solver. *Formal Methods in System Design*, 55:1–32, 2019.

[156] E. Lamma, P. Mello, M. Milano, R. Cucchiara, M. Gavanelli, and M. Piccardi. Constraint propagation and value acquisition: why we should do it interactively. In *Proceedings of the 16th International Joint Conference on Artifical Intelligence - Volume 1*, IJCAI'99, page 468–473, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[157] E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. Symmetry Reductions in Model Checking. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1998.

[158] TLC.tla Module. https://github.com/tlaplus/tlaplus/blob/master/tlatools/org.lamport.tlatools/src/tla2sany/StandardModules/TLC.tla, 2020. Accessed: 2024-07-29.

[159] Martín Abadi and Leslie Lamport. The Existence of Refinement Mappings. *Theoretical Computer Science*, 1991.

[160] William Schultz. *TwoPhase* Inductive Proofs. https://github.com/will62794/scimitar/blob/main/benchmarks/consensus_epr_IndProofs_2.tla, 2024. Accessed: 2024-10-02.

[161] William Schultz. *TwoPhase* Inductive Proofs. https://github.com/will62794/scimitar/blob/main/benchmarks/TwoPhase_IndProofs.tla, 2024. Accessed: 2024-10-02.

[162] William Schultz. *GermanCache* Inductive Proofs. https://github.com/will62794/scimitar/blob/main/benchmarks/GermanCache_IndProofs.tla, 2024. Accessed: 2024-10-02.

[163] William Schultz. *Bakery* Inductive Proofs. https://github.com/will62794/scimitar/blob/main/benchmarks/Bakery_IndProofs.tla, 2024. Accessed: 2024-10-02.

[164] William Schultz. *AsyncRaft ElectionSafety* Inductive Proofs. https://github.com/will62794/scimitar/blob/main/benchmarks/AsyncRaft_IndProofs_OnePrimaryPerTerm_1.tla, 2024. Accessed: 2024-10-02.

[165] William Schultz. *AsyncRaft PrimaryOwnsEntries* Inductive Proofs. https://github.com/will62794/scimitar/blob/main/benchmarks/AsyncRaft_IndProofs_PrimaryHasEntriesItCreated_4.tla, 2024. Accessed: 2024-10-02.