Automated Verification of Safety and Liveness Properties for Distributed Protocols

Jianan Yao

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
under the Executive Committee
of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2024

Abstract

Automated Verification of Safety and Liveness Properties for Distributed Protocols

Jianan Yao

The world relies on distributed systems, but these systems are increasingly complex and hard to design and implement correctly. This is due to the intrinsic non-determinism from asynchronous node communications, various failure scenarios, and potentially adversarial participants. To address this problem, developers are starting to turn to formal verification techniques to prove the correctness of distributed systems. This involves formally verifying that desired safety and liveness properties hold for the distributed protocol. A safety property is an invariant that should hold true at any point in a system's execution. It ensures the protocol does not reach invalid or dangerous states. A liveness property, on the contrary, describes that some desired good event will eventually happen. There have long been efforts to formally verify safety and liveness of distributed protocols. However, the proof burden is usually prohibitively high for broad real-world adoption. Although there has been a growing list of methods that try to automate the verification of distributed protocols, in particular their safety properties, none of these tools scale to real-world complex protocols with theoretical guarantee.

In this dissertation, I introduce our verification methods and tools for verifying distributed protocols with little to no human effort. The thesis consists of two parts. In the first part, I present our two inductive invariant inference tools, DistAI and DuoAI, which automatically verify safety properties of distributed protocols. In DistAI, I introduce a simulation-enumeration-refinement framework for invariant reasoning, and DuoAI extends it to more complex protocols and

existential quantifiers. The evaluation shows that DuoAI outperforms alternative methods in both the number of protocols verified and the speed to verify them, including solving Paxos more than two orders of magnitude faster than any alternative method.

In the second part, I introduce LVR, our liveness verification tool for distributed protocols. The key theoretical insight is that liveness verification can be soundly reduced to the verification of a list of simpler safety properties, which can often be proved automatically utilizing an arsenal of invariant inference tools. The reduction leaves one remaining task—to synthesize a ranking function to prove termination, for which I present a new and effective pipeline. LVR is successfully applied to eight distributed protocols and is the first to demonstrate that liveness properties of distributed protocols can be proved with limited human input.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would express my sincere gratitude to everyone who has provided knowledge, guidance, encouragement, and support throughout the five years of my Ph.D. journey.

First and foremost, I want to thank my advisor, Ronghui Gu, for his years of guidance and support in my Ph.D. study. Ronghui Gu led me through the treacherous roads into the palace of systems verification, provided invaluable advice on pursuing my research direction, and guided me to become an independent researcher. I also want to thank Jason Nieh and Suman Jana, my mentors and close collaborators at Columbia. Jason Nieh provided instrumental supervision from a systems perspective on my protocol verification works, sharing intellectual insights from experimental design to paper storytelling. Suman Jana provided inspiration and essential supervision from a machine learning perspective on my earlier works on invariant inference. I strongly appreciate my three faculty mentors for their intellectual acumen and their profound patience in countless meetings and emails over the years, without which this dissertation would never come to light.

I would also like to extend my gratitude to my mentors and collaborators during my internships in industry. I thank Xinyuan Sun and Zhaozhong Ni for their mentorship at CertiK. I thank Junkil Park, David Dill, and Shaz Qadeer for their mentorship at Meta. I thank Ziqiao Zhou, Weiteng Chen, and Weidong Cui for their mentorship and guidance at Microsoft Research and beyond. From these talented researchers, I gained fresh knowledge, fresh skills, as well as fresh mindset on the dynamics between academia research and industry product, which has helped shape my research trajectory and will have a positive and lasting impact on my future career.

I want to thank Junfeng Yang for his valuable feedback on the dissertation as a committee member.

I want to thank Eden Frenkel for pointing out a mistake in Lemma 6 of the DuoAI paper, which is corrected to Lemmas 6 and 7 in this dissertation.

I would like to extend my gratitude to all my collaborators, including Runzhou Tao, Gabriel Ryan, Justin Wong, John Hui, Shaokai Lin, Xupeng Li, Yunong Shi, Shih-Wei Li, and many others. Their hard work and contributions are indispensable in our joint research achievements. I am also grateful to all my friends and colleagues who have exchanged ideas and lent support on research and life throughout my Ph.D. years. In particular, I want to thank Chengzhi Mao, Binghui Peng, Shunhua Jiang, Raphael J. Sofaer, Kexin Pei, Jay Lorch, Chris Hawblitzel, Xuheng Li, Kele Huang, Fanqi Yu, Wei Hao, Yuhong Zhong, Haoyu Li, Haoran Ding, João Carlos Pereira, Chengyue He, Wenxin Cao, Zhanpeng He, Chenghao Yang, and many others.

Lastly, I want to thank my parents, grandparents, and other family members. I have not met my family for over four years due to first travel restrictions and then visa concerns. However, their support on my Ph.D. study has been unwavering, and their encouragement has been a great source of strength in my life, especially at the most difficult times.

Chapter 1: Introduction

The computational world we live in today relies on distributed systems. By their classical definition, distributed systems are "systems consist of a number of physically distributed components that work independently using their private storage, but also communicate from time to time by explicit message passing" [1]. In the past two decades, distributed systems have been growing in variety and prevalence. Ride sharing services like Uber and Lyft use distributed real-time systems to match drivers and passengers, and autonomous vehicles further use them to communicate with other vehicles and infrastructure to make time-sensitive driving decisions. Built on top of distributed computing systems, cloud service providers such as Amazon Web Service (AWS), Microsoft Azure, and Google Cloud, are utilized by an estimated 94% of U.S. enterprises in 2023 [2], enabling countless companies to serve billions of users. Distributed database systems across geographic regions are used by applications to ensure fault tolerance and reduce latency. Blockchain systems, a special form of unforgeable distributed ledgers, are now applied for decentralized finance, gaming, auctions, health care, and others.

Along with the broadening application of distributed systems, these systems become increasingly large and complicated, and concerns and incidents about their correctness and security have been growing over time. A failure of a cloud service can disrupt millions of users. The outage of Amazon Web Service in December 2021 not only halted Amazon's own retail and delivery service, but also left colleges delaying online exams and web-connected cat feeders at home disfunctioning [3]. A security flaw in a distributed database or cloud infrastructure can compromise sensitive data from a large number of users. Attacks on blockchain systems can cause huge financial loss to companies and individuals.

Given the intrinsic nondeterminism from asynchronous node communications and various failure scenarios, the possible execution traces of a distributed system are generally huge. Therefore,

it is very difficult to statically analyze, or sufficiently test the system without leaving corner cases behind. Formal verification provides a way to fundamentally ensure the correctness of distributed systems, using mathematical proof to demonstrate that the system consistently satisfies its specifications [4, 5, 6]. However, existing methods require skilled developers familiar with verification methods and tools to write thousands of lines of proof, in Coq, Dafny, or another verification language for a prototype system. The huge proof burden, often many times of the system itself, has long limited the wider adoption of verification in real-world software development.

To formally verify a distributed system, IronFleet [5] introduced a two-stage framework. The developers first abstract the system into a high-level protocol (i.e., a state machine), upon which the developers formally prove the desired properties hold. Then the developers prove that the low-level implementation of each node in the system is consistent with the behavior specified in the high-level state machine. Later work on verifying distributed systems have followed and extended this paradigm [7, 8], in which verifying a high-level distributed protocol is a necessary first stage. Besides, verifying distributed protocols has merits in itself. Although classical protocols such as Paxos have had their correctness established for decades, many new and complicated consensus protocols are actively being proposed primarily for blockchain systems, aiming to ensure fault tolerance while maximizing throughput [9, 10]. Formal verification is useful to ensuring the correctness of these protocols, which is crucial for the blockchain systems they underpin.

When formally verifying distributed protocols, the desired properties to be verified can be categorized into safety properties and liveness properties. Informally speaking, a safety property describes that certain bad things can never happen. while a liveness property describes that certain good things must eventually happen. For example, in a consensus protocol, the safety property says that the nodes should never reach conflicting consensus. The protocol should never be in a state where node A believes a consensus is reached on value V_1, while node B believes a consensus is reached on a different value V_2. The liveness property, on the other hand, says that the nodes will eventually reach a consensus, rather than exchange messages infinitely or terminate without a consensus value determined.

Much effort has been dedicated to verifying safety properties of distributed protocols. The Ivy tool [11] lets the developer provide a set of invariants and protocol specification that defines its safety property, which Ivy automatically checks using an SMT solver. Each invariant can be expressed as a logical formula, which consists of a prefix with quantifiers ($\forall$ or $\exists$) and a certain number of variables, and a set of logical relations among the variables. Ivy checks if adding the invariants to the safety property makes it inductive, meaning that the conjunction of all invariants with the safety property is inductive. Conjunction requires each invariant to hold, so Ivy reports whether any invariant fails, at which point the developer can try again with a different set of invariants. This requires substantial manual effort by the developer.

Recent work has focused on automating invariant generation for distributed protocols, but with various limitations. I4 [12] observes that invariants for some distributed protocols do not depend on the size of the system, so I4 uses a specialized model checker to generate invariants for a small size system, then generalizes them and uses Ivy to check if the conjunction of the invariants with the safety property is inductive for the protocol specification. If not, Ivy indicates which invariants failed. I4 removes them and tries again, and if that fails, tries using a larger size system to generate invariants. However, I4 provides no guarantee that it can find the inductive invariant, as it may not be possible to verify a protocol based on invariants derived from a single instantiation of the protocol. For example, if the protocol involves the parity of nodes, then no single instance can capture all behaviors of the protocol. I4 still requires manual effort, as a developer must inspect a protocol to add additional constraints that reduce the state space to make model checking feasible.

FOL-IC3 [13] infers invariants by searching for logical separators between reachable and invalid states in the protocol. It searches for separators by checking if a separator exists for a fixed number of variables and logical constraints, iteratively increasing the number of possible variables and constraints it considers if it fails. FOL-IC3 does not scale to more complex protocols due to the large space of possible separators that it enumerates and its heavy and repeated use of expensive SMT queries.

On the separate front of verifying liveness properties of distributed protocols, the progress has

been more limited compared with verifying safety. This originates from the inherent difficulty in liveness reasoning, when the developer has to reason about possibly infinite sequences of states on a time frame. Previous approaches require extensive manual effort. IronFleet [5] was the first to verify the liveness properties of distributed protocols with SMT solvers, but required thousands of lines of user inputs in Dafny for not just the protocol specification but its implementation as well to complete the proof. Liveness properties were expressed in temporal logic, which was problematic to use with SMT solvers without causing them to time out. To avoid timeouts, IronFleet employed various workarounds that required extensive human effort to tune triggers to tell the underlying SMT solver what to do and what not to do. Dynamic abstraction [14, 15] reduces a liveness property of a distributed protocol to a safety property of another protocol that extends the original one, but the extended protocol is more complex and involves a much larger number of states. Unfortunately, proving the safety property of the extended protocol can require finding dozens of additional invariants, which must be done manually since it is well beyond the scope of existing automated techniques. The resulting proof structure with many invariants unrelated to the original protocol is also not human-readable.

My thesis is that data-driven methods, combined with new algorithmic and system design, can automatically infer inductive invariants thus formally verify safety properties of distributed protocols. Furthermore, my thesis is that by leveraging a reduction from liveness to safety combined with ranking functions, liveness properties of distributed protocols can also be formally verified in a mostly automated manner.

In Chapter 2, I will present DistAI [16], a data-driven invariant inference tool for distributed protocols. I introduce the simulation-enumeration-refinement framework for invariant inference. DistAI first uses protocol simulation to obtain random execution traces, then filter out incorrect invariants based on the fact that an invariant must hold on any valid protocol state. DistAI enumerates possible invariants in a bounded search space and check them against the traces. Then DistAI iteracts with Ivy to monotonically refine the invariants until a correct inductive invariant is reached. I prove that DistAI is guaranteed to succeed if the protocol is verifiable with a $\exists$-free

inductive invariant. The evaluation demonstrates that DistAI verifies more protocols automatically and runs faster than alternative tools.

DistAI marks a significant step forward in verifying protocol safety, but it still cannot infer invariants with $\exists$ quantifiers and struggle to scale to more complicated protocols. In Chapter 3, I will introduce DuoAI [17], a faster and more effective invariant inference tool. DuoAI inherits the simulation-enumeration-refinement pipeline from DistAI but employs new algorithms for both enumeration and refinement. To improve efficiency and avoid redundancy, I introduce the minimum implication graph, which encodes implication relations between logical formulas. Invariant enumeration is done following the order defined in the graph, starting from the strongest formulas and iteratively moving to weaker ones. After invariant enumeration, DuoAI runs two procedures in parallel, top-down refinement and bottom-up refinement, that utilizes the minimum implication graph and interacts with Ivy to refine the invariants until a correct inductive invariant is reached. I prove that DuoAI is guaranteed to find a correct inductive invariant if one exists in the search space. Our evaluation on 27 distributed protocols shows that DuoAI is able to automatically infer inductive invariants for a wide range of distributed protocols from database maintenance to consensus, outperforming alternative methods both in the number of protocols verified and the time to verify them, achieving performance improvements up to over two orders of magnitude.

Then In Chapter 4, I will present our liveness verification framework LVR [18]. The theoretical breakthrough underpinning LVR is that liveness proof can be soundly reduced to the proof of a set of simpler safety properties, without augmenting the protocol as in previous acyclicity-based methods. The simpler safety properties can then be mostly proved automatically by utilizing an arsenal of invariant inference tools. The only challenge is that the safety properties require the identification of an appropriate ranking function—one that decreases after each transition and remains nonnegative. Contrary to conventional wisdom, my empirical research reveals that ranking functions for a broad range of distributed protocols can be constructed as polynomial expressions over integer variables, making their synthesis easier than previously believed. LVR first employs static analysis to determine certain properties of the integer variables, such as upper and lower bounds.

Subsequently, LVR formulates the synthesis of the ranking function as a constraint solving problem, which is efficiently resolved using an SMT solver. The effectiveness of LVR is underscored by its ability to automatically verify liveness for eight distributed protocols for the first time, including three versions of Paxos, with minimal human intervention.

The contributions of this dissertation include:

1. The simulation-enumeration-refinement pipeline for inductive invariant inference for distributed protocols.

2. DistAI, a data-driven invariant inference tool targeting $\exists$-free inductive invariants for distributed protocols.

3. The minimum implication graph that captures the implication relation between formulas, and the efficient invariant enumeration and refinement algorithms based on the graph.

4. The observation that practical distributed protocols usually require few invariants with quantifier alternation, and an invariant refinement algorithm taking advantage of it.

5. DuoAI, a faster and more effective invariant inference tool with a theoretical guarantee of success and strong performance across various classes of distributed protocols.

6. A sound reduction from verifying liveness to verifying a set of safety properties without augmenting the protocol.

7. A pipeline to synthesize a ranking function of a state-transition system by analyzing integer variables, upper and lower bounds, then how they change in each transition.

8. LVR, a new verification framework that is the first verify liveness properties of distributed protocols in a mostly automated manner.

Chapter 2: DistAI: Data-Driven Inductive Invariant Inference for Verifying Protocol Safety

In this chapter, I present my research on automated invariant inference for verifying safety properties of distributed protocols. Specifically, I present DistAI [16], the first data-driven automated invariant inference tool for distributed protocols. DistAI introduces the new pipeline for invariant synthesis consisting of three phases, protocol simulation (§2.2), invariant enumeration (§2.3), and invariant refinement (§2.4). The evaluation results are shown in §2.5.

2.1 DistAI Workflow

Figure 2.1 illustrates how DistAI works. Starting with a distributed protocol specification for IVy, first, DistAI does two-stage sampling, as discussed in Section 2.2. It simulates the protocol on different sizes of systems, which we refer to as different size protocol instances, and records the system state as it changes as a sequence of data samples. It then projects the samples into subsamples based on the formula size currently being considered. We express formulas in prenex normal form (PNF), so that the prefix, which we refer to as *an invariant template*, has a maximum number of quantified variables and the matrix has a maximum number of literals. Second, DistAI does enumeration, as discussed in Section 2.3. It enumerates all strongest candidate invariants that satisfy the subsamples. Third, DistAI feeds the candidate invariants to IVy, which either succeeds with the conjunction of the invariants and the desired safety property as the inductive invariant, or fails and indicates each invariant that does not hold. Fourth, DistAI performs monotonic refinement, as discussed in Section 2.4. For each candidate invariant that does not hold, DistAI weakens the invariant by adding literals, then feeds the new set of candidate invariants to IVy, repeatedly weakening failed invariants until either it finds the inductive invariant or the safety property it-

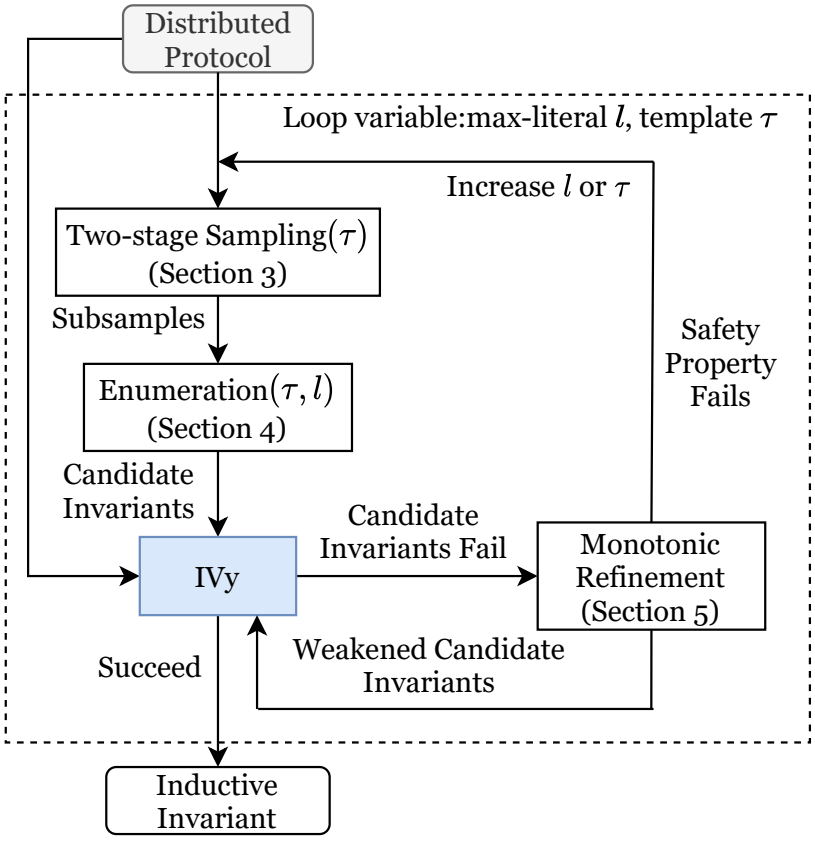


Figure 2.1: DistAI workflow.

self fails. In the latter case, DistAI increases the formula size by increasing either the maximum number of variables or the maximum number of literals allowed, and repeats the whole process of sampling, enumeration, and refinement.

We use the Ricart-Agrawala protocol [19] as an example of how DistAI works. Figure 2.2 shows the IVy specification of this classic distributed mutual exclusion protocol, which has five key pieces we use for learning invariants:

1. **Types.** (line 2) Types define different domains of the protocol (e.g., nodes, packets, epochs). The Ricart-Agrawala protocol only has one type, `node`.

2. **Relations.** (lines 4-6) Relations define the state of the protocol, with variables that represent types used as arguments. The Ricart-Agrawala protocol has three relations. For example, relation `holds(N)` has one variable `N` of type node, and indicates if `N` is in the critical section. If the current instance has three nodes N_1, N_2, N_3, then there are three concrete predicates

$holds(N_1), holds(N_2), holds(N_3)$ associated with relation $holds$. Each predicate is either `true` or `false` at a certain system state.

3. **Initialization.** (lines 8-12) Initialization defines the initial state of the protocol in terms of its relations. For the Ricart-Agrawala protocol, all relations are initially false.

4. **Actions.** (lines 13-35) Actions define how the protocol may transition from one state to another, modifying the state by setting relations to `true` or `false`. Actions are defined with preconditions using the `require` keyword, which must be satisfied for the protocol to take the action.

5. **Safety Property.** (line 43) The target invariant, defined with logical constraints on the types and relations.

As shown in the specification, a node can send a request for the critical section to another node and can only enter the critical section after it has received replies from all other nodes. When receiving a request, a node delays its reply if it is currently holding the critical section, or if it has requested the critical section and already received a reply from the requester, which indicates an earlier timestamp and a higher priority. The node then sends the reply after it leaves the critical section. The safety property at line 43 asserts that at any time, there is no more than one node holding the critical section. For simplicity, Figure 2.2 specifies the protocol without explicit timestamps and only shows one `requested` relation as opposed to separate `request_send` and `request_received` relations which would be part of the real protocol. The safety property of Ricart-Agrawala is not an inductive invariant itself. One needs to add the following two invariants to the safety property so that the resulting conjunction forms an inductive invariant:

$$\forall N_1\ N_2.\ \neg(replied(N_1, N_2) \wedge replied(N_2, N_1)) \tag{2.1}$$

$$\forall N_1\ N_2.\ holds(N_1) \wedge N_1 \neq N_2 \rightarrow replied(N_1, N_2). \tag{2.2}$$

The first invariant asserts the absence of bidirectional reply, meaning that any two nodes cannot

```
1  #lang ivy1.7
2  type node
3
4  relation requested(N1:node, N2:node)
5  relation replied(N1:node, N2:node)
6  relation holds(N:node)
7
8  after init {
9      requested(N1, N2) := false;
10     replied(N1, N2) := false;
11     holds(N) := false;
12 }
13 action request(requester: node, responder: node) = {
14     require ~requested(requester, responder);
15     require requester ~= responder;
16     requested(requester, responder) := true;
17 }
18 action reply(requester: node, responder: node) = {
19     require ~replied(requester, responder);
20     require ~holds(responder);
21     require ~replied(responder, requester);
22     require requested(requester, responder);
23     require requester ~= responder;
24     requested(requester, responder) := false;
25     replied(requester, responder) := true;
26 }
27 action enter(requester: node) = {
28     require N ~= requester -> replied(requester, N);
29     holds(requester) := true;
30 }
31 action leave(requester: node) = {
32     require holds(requester);
33     holds(requester) := false;
34     replied(requester, N) := false;
35 }
36
37 export request
38 export reply
39 export enter
40 export leave
41
42 # safety property
43 invariant [safety] holds(N1) & holds(N2) -> N1 = N2
```

Figure 2.2: Ricart-Agrawala protocol written in IVy. "~" stands for negation. Capitalized variables are implicitly quantified. For example, line 9 means $\forall N_1\ N_2 \in node,\ requested(N_1, N_2) := false$.

both give the other one a higher priority. The second invariant says that any node holding the critical section must have received replies from all other nodes. DistAI automatically finds the inductive invariant by learning the additional invariants.

Two-stage sampling. To automatically learning the inductive invariant and prove the correctness of the Ricart-Agrawala protocol, DistAI first does two-stage sampling, as shown in Figure 2.1. It simulates the protocol at different instance sizes and records the system state as a sequence of data samples, each of which presents the values of all the relations. For example, a data sample for an instance size of five nodes (i.e., n_1, n_2, $\cdots$, n_5) using the Ricart-Agrawala protocol consists of 55 boolean values denoting if the following 55 predicates hold or not at the current state:

$$
\begin{aligned}
&requested(n_1, n_1),\ requested(n_1, n_2), \cdots,\ requested(n_5, n_5)\\
&replied(n_1, n_1),\ replied(n_1, n_2), \cdots,\ replied(n_5, n_5)\\
&holds(n_1),\ holds(n_2), \cdots,\ holds(n_5).
\end{aligned}
$$

DistAI chooses a maximum formula size for a candidate invariant, which defines the maximum number of quantified variables that can be used per domain and the maximum number of literals (a predicate or its negation) in the formula. DistAI projects data samples to subsamples, which only contain values of predicates that match the formula size. For example, given a formula with two variables $\{\forall N_1\ N_2\}$, indicating that candidate invariants start with $\forall N_1\ N_2 \cdots$, each subsample only contains the value of predicates related to two assigned nodes.

Enumeration. DistAI then enumerates all strongest candidate invariants that satisfy the subsamples, up to the maximum formula size. Invariants are expressed as formulas in first-order logic. For example, given a maximum formula size of at most two variables and two literals, the following

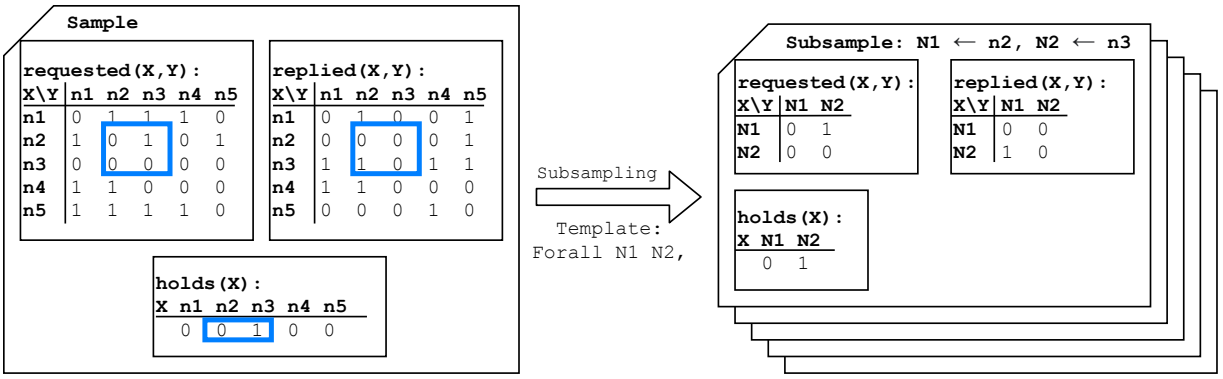


Figure 2.3: The subsampling process. The frame on the left shows a single sample state of a finite instance of the Ricart-Agrawala protocol with five nodes. A single subsample with two quantified variables $\{\forall N_1\ N_2\}$ is generated by mapping the quantified variables to concrete nodes in the finite instance, **n2** and **n3**, and extracting their associated values (shown in blue boxes in the sample frame). $0/1$ stand for false/true.

three formulas could be enumerated, assuming they all satisfy the subsamples:

$$\forall N_1 \neq N_2.\ replied(N_1, N_2) \tag{2.3}$$

$$\forall N_1 \neq N_2.\ replied(N_1, N_2) \vee replied(N_2, N_1) \tag{2.4}$$

$$\forall N_1 \neq N_2.\ replied(N_1, N_2) \vee \neg holds(N_1). \tag{2.5}$$

However, DistAI would only generate the first one as a candidate invariant because the first one implies the other two, so the latter two formulas can be skipped. The first formula is the strongest candidate invariant among the three formulas.

Monotonic refinement. DistAI feeds the candidate invariants and the protocol specification to IVy, which runs its SMT solver to check if the conjunction of the invariants with the safety property is an inductive invariant. If the solver passes, DistAI has succeeded. Succeeding means that if the conjunction of the invariants with the safety property holds before a protocol action is taken, each invariant still holds after the action is taken. Otherwise, IVy outputs which candidate invariants failed, and DistAI weakens each failed invariant and tries again with IVy with the candidate invariants, each failed invariant being replaced by weakened invariants with no more variables and literals than the maximum formula size. For the Ricart-Agrawala protocol, IVy shows that the in-

variant in Equation (2.3) is incorrect. The invariant is then weakened into Equations (2.4) and (2.5), among others, which will then be checked by IVy. Later, IVy will also invalidate Equation (2.4), and since it has reached the maximum formula size, it will be simply discarded. Equation (2.5) is never invalidated by IVy and will be part of the inductive invariant in the end. If IVy indicates that the safety property failed, it means that the formula size is not sufficient. DistAI will then increase the formula size by either increasing the maximum number of variables or the maximum number of literals, then re-run the process.

2.2 Two-Stage Sampling

Obtaining data samples for a distributed protocol requires simulating a finite instance of the protocol and recording the system state on each action. However, invariants are usually composed of quantified variables that impose constraints on all domains of the protocol, not just the specific domains of a finite instance. Therefore, DistAI projects the collected finite state samples into abstract subsamples on quantified variables that also apply to all domains of the protocol and represent potential predicates in the invariant. We refer to these two procedures as *sampling* and *subsampling* respectively, since many abstract subsamples can be generated from a single concrete data sample.

The two-stage sampling has four parameters: the absolute maximum number of instances to consider before terminating (MI), the maximum number of instances to consider before terminating if no further subsamples are generated (MIS), the number of actions to take when simulating a finite instance (MA), and the number of subsamples to generate from one data sample (SD). As we show in Section 2.5, DistAI's ability to find an inductive invariant is not sensitive to the specific values of these parameters, which are always set to their defaults of 1000, 20, 50, and 3, respectively.

Sampling. DistAI first translates the protocol from IVy into Python, with relations simulated by multidimensional arrays, and actions simulated by Python functions. This allows DistAI to

efficiently simulate the protocol. The translation is not in the trusted computing base since learned invariants are eventually validated by IVy.

DistAI then simulates the protocol in Python from different valid initial states on randomly chosen finite instances of the protocol. DistAI randomly chooses an instance size from some range of sizes using a simple discrete uniform distribution. For each domain T (e.g., node), a protocol typically has some minimum instance size to function well, which we refer to as N^T_{min}. In practice, the minimum instance size N^T_{min} is determined as the maximum number of variables of type T in any relation; a protocol will not function well if its relations have variables that cannot be mapped to the instance size. For example, for a protocol with two relations $p(n_1 : T_1, n_2 : T_1, m_1 : T_2)$ and $q(m_1 : T_2)$, we have $N^{T_1}_{min} = \max(2, 0) = 2$ and $N^{T_2}_{min} = \max(1, 1) = 1$. The probability for choosing a given domain size N^T is then:

$$Pr[N^T] = 1/w \;\; (N^T_{min} \leq N^T < N^T_{min} + w)$$

DistAI uses $w = 4$ by default. This allows sampling from multiple instance sizes, but limits the instance sizes to within w of the minimum instance size for performance, as larger instances take more time to simulate.

For each valid initial state, DistAI simulates the protocol by performing MA number of actions. Since the distributed protocols are nondeterministic with regard to the next action taken (e.g., we do not know which node will send the next request or reply), multiple runs from the same initial state will result in different samples. Given an initial state s_0, DistAI uses the simple method formalized in Algorithm 1, to simulate a protocol, which randomly chooses an action from an action pool (line 6) with randomly chosen arguments from an argument pool (line 9) that satisfy the precondition (line 10). It then performs the action, records the new system state, and repeats the process (line 16-17). An action is removed from the action pool once its argument pool is exhausted, and the protocol terminates if the action pool is exhausted. Since some protocols may never terminate, MA defines an upper bound on the number of actions performed. Once the simulation completes,

Algorithm 1 Stochastic Sampling Algorithm.

Input: Protocol $\mathcal{P}$ with actions $\mathcal{A}$. Initial state s_0
Output: A simulation trace, represented by a set of states S

```
1:  S := {s_0}, s := s_0
2:  for iter := 1 to MA do
3:      action_pool := A
4:      action_found := false
5:      while ¬action_found ∧ |action_pool| > 0 do
6:          action := select_random(action_pool)
7:          args_pool := enum_arguments(s, a)
8:          while |args_pool| > 0 do
9:              args := select_random(args_pool)
10:             if precondition_holds(P, s, action, args) then
11:                 action_found := true
12:                 break
13:             args_pool := args_pool \ {args}
14:         action_pool := action_pool \ {action}
15:     if action_found then
16:         s:=execute_protocol(P, s, a, args)
17:         S:=S ∪ {s}
18:     else
19:         break
20: return S
```

the set of reached states S is returned.

For example, for the Ricart-Agrawala protocol, during each iteration of the algorithm, DistAI first randomly selects one of the four possible actions: `request`, `reply`, `enter`, and `leave`. If `request` is selected, DistAI then randomly chooses the nodes for its two arguments, `requester` and `responder`. However, not every pair of nodes are valid arguments as the two nodes must satisfy lines 14-15, the two preconditions to legitimately trigger the `request` action under the protocol. If the current ⟨`requester`, `responder`⟩ pair violates the precondition, DistAI removes it from the argument pool and randomly selects another one. This repeats until a valid pair of arguments is found or the pool is exhausted, in which case DistAI removes `request` from the action pool and selects another action.

After each iteration, DistAI logs the current system state, represented by the value of all the

relations. In Figure 2.2, that is the value of predicates $requested(N_1, N_2)$, $replied(N_1, N_2)$, and $holds(N_1)$ for all N_1 and N_2. Figure 2.3 shows a sample of the Ricart-Agrawala algorithm for an instance size of five nodes in the left frame. The $requested$ and $replied$ relations each take two nodes as arguments, so their samples record the relations for all possible pairs of nodes, resulting in 25 boolean values each. The $holds$ relation only takes a single node as argument, so five boolean values are recorded, one for each of the five nodes in the protocol instantiation.

DistAI can also simulate protocols calling other protocols, even when the protocol being called is a blackbox. When a protocol calls another protocol through a blackbox interface, described by a specification without a concrete implementation, DistAI treats it as an action with nondeterministic behavior. If the action is selected with arguments that satisfy its preconditions, DistAI selects randomly updated states that satisfy its postconditions as the execution result.

Our simple stochastic sampling procedure, while very efficient, may not achieve high coverage and can leave corner cases uncovered. More sophisticated techniques [20, 21, 22] can be applied to improve coverage for complex protocols with sparse inputs and difficult to reach states. However, the correctness of learned invariants is guarded by the Z3 SMT solver used by IVy. If the samples are incomplete and the invariants fail the SMT check, DistAI will iteratively refine the invariants until they are correct, as discussed in Section 2.4.

Subsampling. The data samples from protocol simulation may be of all different lengths depending on the instance size used. We want to map the concrete samples from simulation to an invariant template, the small set of quantified variables that may appear in the invariant, denoted by τ. Given a set of data samples and an invariant template, DistAI applies a subsampling procedure translating the variable length data samples to fixed length vectors, which we call *subsamples*. Formally, a subsample corresponds to an assignment α of the variables in τ and contains the values of relations given the assignment to the template. Subsamples taken with an invariant template with

variables $V_1, ..., V_n$ can then be used to learn invariants (denoted I) on those variables:

$$\tau = \{\forall V_1 ... V_n\} \qquad\qquad \forall V_1 ... V_n.\ I(V_1, ..., V_n).$$

For example, in the Ricart-Agrawala protocol, the relations $requested$ and $replied$ each operate on two nodes, so the initial template is $\tau = \{\forall N_1\ N_2\}$. Under this template, there are only 10 predicates that may appear in an invariant formula, namely:

$$requested(N_1, N_1),\ requested(N_1, N_2),\ requested(N_2, N_1),$$
$$requested(N_2, N_2),\ replied(N_1, N_1),\ replied(N_1, N_2),$$
$$replied(N_2, N_1),\ replied(N_2, N_2),\ holds(N_1),\ holds(N_2).$$

For this template, a 5-node data sample can induce $5 * 4 = 20$ subsamples, by first assigning one node X_1 for N_1 and then another node X_2 for N_2 , as illustrated in Figure 2.3. Since there are 10 predicates, each subsample has 10 possible boolean values, so one data sample results in $20 \times 10 = 200$ boolean values.

Enumerating all valid subsamples from each sample can be computationally undesirable, especially for multi-domain protocols. If we add a new domain `msg`, and let the template be $\{\forall\ N_1\ N_2 \in$ `node`, $M_1\ M_2 \in$ `msg`$\}$, then a sample with five nodes and 10 messages will induce 1,800 subsamples. Therefore, DistAI randomly chooses SD valid subsamples from each sample. Two-stage sampling terminates when MI instances have been simulated, or no new subsample is found after simulating MIS consecutive instances of the protocol. The subsamples are then deduplicated and passed on to invariant enumeration.

Although DistAI's sampling has several parameters, they do not need to be manually tuned to find inductive invariants. We use the default values for all protocols. For example, when MA or SD become larger, each simulation round will take longer, but fewer rounds will be required to converge. Similarly, a small MI/MIS may stop two-stage sampling prematurely, but the missing states will be resolved later by monotonic refinement. The parameters do not affect whether

DistAI finds inductive invariants, only how fast it finds them. Sampling is useful simply as a performance optimization that reduces the number of SMT queries required during refinement.

2.3 Candidate Invariant Enumeration

Algorithm 2 shows DistAI's enumeration-based algorithm to generate candidate invariants from the subsamples obtained in Section 2.2. To reduce the number of candidate invariants required for covering the invariant space and reduce the maximum number of literals needed for finding the inductive invariant, we partition the invariant space into multiple regions, each represented by a constrained template called a *subtemplate*. We then enumerate all possible invariants in each region (i.e., under each subtemplate), and retain candidate invariants that hold for the collected subsamples.

Template decomposition. Before enumerating candidates invariants, we decompose templates into subtemplates that incorporate additional constraints (line 1). A template with N variables in the same domain will be split into N subtemplates which have from 1 to N variables. A subtemplate with more variables is said to be larger than a subtemplate with fewer variables. For example, a template $\tau = \{\forall N_1\ N_2\}$ will be split into two subtemplates, $\tau_1 = \{\forall N_1\ N_2.\ N_1 = N_2\} = \{\forall N_1\}$ and $\tau_2 = \{\forall N_1\ N_2.\ N_1 \neq N_2\}$, abbreviated as $\{\forall N_1 \neq N_2\}$, with τ_2 being larger than τ_1. All operations that use subtemplates in Algorithm 2 traverse them from smallest to largest (lines 2 & 5).

This subtemplate optimization reduces the cost of enumeration in two ways. First, subtemplates reduce the number of candidates that need to be enumerated due to symmetry. For example, for the Ricart-Agrawala protocol, when using template τ, both of the following invariants will be enumerated:

$$\forall N_1\ N_2.\ \neg replied(N_1, N_1) \qquad \forall N_1\ N_2.\ \neg replied(N_2, N_2).$$

On the other hand, when using the subtemplate τ_1, the equivalent enumeration would only result in one candidate:

$$\forall N_1.\ \neg replied(N_1, N_1). \tag{2.6}$$

Furthermore, DistAI will project Equation (2.6) using τ_2 to the following candidate invariants:

$$\forall N_1 \neq N_2.\ \neg replied(N_1, N_1) \quad \forall N_1 \neq N_2.\ \neg replied(N_2, N_2),$$

which are then marked as validated using τ_1, avoiding further redundant validations. We refer to this as invariant projection.

Second, subtemplates can reduce the maximum number of literals in the invariant formula. For example, one invariant of the Ricart-Agrawala protocol (Equation 2.2) can be rewritten as:

$$\forall N_1\ N_2.\ \neg(N_1 \neq N_2) \vee \neg holds(N_1) \vee replied(N_1, N_2) \tag{2.7}$$

This is a disjunction of three literals under the full template τ. However, using subtemplate $\tau_2 = \{\forall N_1 \neq N_2\}$, an equivalent form of this invariant can be learned using a formula size with a maximum of two literals:

$$\forall N_1 \neq N_2.\ \neg holds(N_1) \vee replied(N_1, N_2)$$

We can denote an invariant as $\tau : inv$, where τ is the subtemplate under which the invariant formula inv is found and inv is expressed as a disjunction of literals. In this example, we effectively can denote the same invariant using subtemplate τ_2 as $\tau_2 : inv'$, where one literal that was previously part of inv is no longer part of inv' because it is now a part of τ_2. Because DistAI operates in formula space and the time complexity of enumeration is exponential in the maximum number of literals, such a small reduction in the number of literals can have a significant impact on the overall cost of enumeration.

Subtemplates can also reduce the maximum number of literals by exploiting another form of symmetry. If there is a total order on a domain, such as with node identifiers, we will further assign an order on the variables in the subtemplate and strengthen $\{\forall N_1 \neq N_2\}$ into $\{\forall N_1 < N_2\}$ Because of symmetry, invariant formulas with $\{\forall N_1 < N_2\}$ are equivalent to those with $\{\forall N_2 < N_1\}$,

Algorithm 2 Invariant Enumeration Algorithm.

Input: Template τ, subsample table ST, max-literal l
Output: A set of invariants I^*

```
1:  subtemplates := decompose_templates(τ)
2:  for τ' ∈ traverse(subtemplates) do
3:      proj_table[τ'] := ST_|τ'
4:      I[τ'] := ∅
5:  for τ' ∈ traverse(subtemplates) do
6:      predicates := proj_table[τ'].header
7:      P_τ := predicates ∪{¬p|p ∈ predicates}
8:      for n := 1 to l do
9:          for inv ∈ combinations(P_τ, n) do
10:             if check_subset_exists(inv, I[τ']) then
11:                 continue
12:             if check_inv_holds(inv, proj_table[τ']) then
13:                 I[τ'] := I[τ'] ∪ {inv}
14:     for τ'_succ ∈ successors(τ') do
15:         for inv ∈ I[τ']) do
16:             I[τ'_succ] := I[τ'_succ]∪ proj_inv(inv, τ', τ'_succ)
17: I* := {(τ' : inv) | τ' ∈ subtemplates, inv ∈ I[τ'], inv was checked against subsamples}
```

so we do not need to enumerate the latter once we have done the former. This is useful since inductive invariants often contain comparisons between variables in a domain with a total order. For example, this optimization helps reduce the maximum number of literals required from six to four for the database chain replication protocol evaluated in Section 2.5.

Subtemplates work with multiple domains as well. Consider a protocol with two domains, T_1 and T_2, where T_1 defines a total order, and the template τ is $\{\forall X_1\ X_2\ X_3 \in T_1,\ Y_1\ Y_2 \in T_2\}$. After template decomposition we get six subtemplates, as shown in Figure 2.4. For multiple domains, there may not exist a total ordering of all subtemplates from smallest to largest, so the only requirement for the order of traversal of subtemplates is that the quantified variables in each subtemplate are not a subset of a prior one, such that formulas can always be validated with the smallest possible subtemplate. In Figure 2.4, $A \to B \to C \to D \to E \to F$ is a valid traversal order, while $A \to D \to E \to B \to C \to F$ would be invalid because the quantified variables $\{X_1, X_2, Y_1\}$ for subtemplate B are a subset of $\{X_1, X_2, Y_1, Y_2\}$ of subtemplate E. We follow graph terminology and

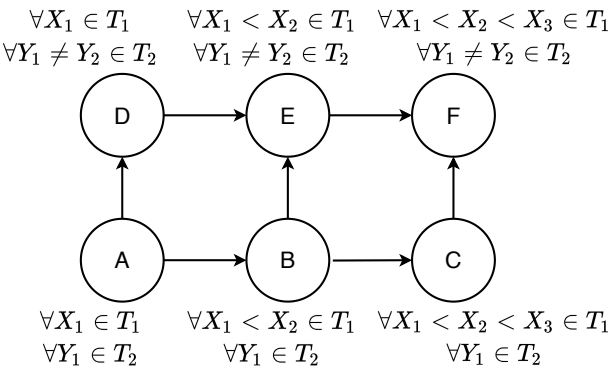


Figure 2.4: Dependency relations between the six subtemplates derived from the template $\{\forall X_1\ X_2\ X_3 \in T_1,\ Y_1\ Y_2 \in T_2\}$.

call subtemplates B and D the *successors* of A, and A the *predecessor* of B and D.

Subtemplate projection. Because DistAI uses subtemplates for candidate enumeration instead of templates, the full subsamples of the invariant template need to be projected onto each subtemplate (line 3). This is done similarly to how data samples are projected onto full subsamples using an invariant template, as discussed in Section 2.2, except in this case, full subsamples are projected onto subsamples using a subtemplate. For example, for this multi-domain protocol, when projecting full subsamples to subtemplate $\{\forall X_1 < X_2 \in T_1,\ \forall Y_1 \in T_2\}$ (node B in Figure 2.4), there are six possible variable mappings: $\{X_1 \rightarrow X_1, X_2 \rightarrow X_2, Y_1 \rightarrow Y_1\}, ..., \{X_1 \rightarrow X_2, X_2 \rightarrow X_3, Y_1 \rightarrow Y_2\}$. Note that the total order on T_1 needs to be preserved, otherwise there would be 12 possible mappings. Similarly, going to back the Ricart-Agrawala protocol example, when projecting full subsamples of the invariant template τ to subtemplate τ_1, there are two possible variable mappings: $\{N_1 \rightarrow N_1, N_1 \rightarrow N_2\}$.

Subtemplate candidate enumeration. DistAI enumerates and checks all possible candidates for each subtemplate τ' (lines 6-13). Each subtemplate τ' has a certain number of predicates m. For example, for the Ricart-Agrawala protocol using template τ_1, there are three predicates: $requested(N_1, N_1), replied(N_1, N_1), holds(N_1)$. DistAI adds the m predicates $p_1, p_2, ..., p_m$ and their negations $\neg p_1, \neg p_2, ..., \neg p_m$ to the literal pool $\mathcal{P}_\tau$ (line 7).

Given a formula size with the maximum number of literals l, DistAI enumerates all subsets

of $\mathcal{P}_\tau$ with size at most l as candidate invariants. For example, if $m = 3$ and $l = 1$, there would be six candidate invariants: $p_1, \neg p_1, p_2, \neg p_2, p_3, \neg p_3$. By default, DistAI initially sets $l = 3$, and iteratively increases it later in the refinement process (see Section 2.4). We only consider invariants in the form of disjunctions of literals since invariants with conjunctions can simply be split into multiple invariants. If a candidate invariant C includes both a predicate and its negation, it will be discarded. If not, DistAI checks the validity of C against the subsamples. If C is satisfied by all subsamples for the subtemplate, C is added to the set of generated invariants, which we refer to as the invariant set.

DistAI exploits symmetry to prune the candidate enumeration space. Whenever an invariant is learned, we permute the quantified variables with the same type and emit equivalent candidates without needing to check if they are satisfied by the subsamples. For example, under the subtemplate $\{\forall\ X \neq Y \in T_1,\ A \neq B \in T_2\}$, if $p(X, Y) \vee \neg q(Y) \vee r(X, A, B)$ is an invariant, then $p(Y, X) \vee \neg q(X) \vee r(Y, B, A)$, along with two other formulas, are also invariants.

Enumeration is ordered by the number of literals in the candidate invariants, and any candidate that is weaker than an invariant already added to the invariant set is skipped (lines 8-11). For example, if we already know $p \vee \neg q$ is an invariant, then for any predicate r, $p \vee \neg q \vee r$ is guaranteed to be a valid but weaker invariant, and can be skipped in the enumeration. Based on Figure 2.3, applying this enumeration procedure to the Ricart-Agrawala protocol with subtemplate $\{\forall N_1\}$ and $l = 2$ results in the following two generated invariants:

$$\neg requested(N_1, N_1) \qquad \neg replied(N_1, N_1)$$

Invariant projection. After finding all candidates on one subtemplate, DistAI calculates the projection of the candidates on each successor, then propagates the projection and moves on to enumerate the next subtemplate (line 14-16). This reduces the cost of validating candidates using larger subtemplates against their subsamples. For example, for the Ricart-Agrawala protocol, suppose we have learned two invariants $\neg requested(N_1, N_1)$ and $\neg replied(N_1, N_1)$ under sub-

template $\tau_1 = \{\forall N_1\}$. Before enumerating candidates under subtemplate $\tau_2 = \{\forall N_1\ N_2\}$, we know the following four candidates must hold under τ_2 because they are projections of the learned invariants under the simpler template τ_1:

$$\neg requested(N_1, N_1) \qquad \neg replied(N_1, N_1)$$
$$\neg requested(N_2, N_2) \qquad \neg replied(N_2, N_2)$$

As a result, these four candidates can simply be added to the invariant set under τ_2 without enumerating and validating them against any subsamples (line 16). Any weaker candidates will also be skipped, further reducing the cost of enumeration. For example, $\neg requested(N_1, N_1) \vee holds(N_1)$ can be skipped since it is weaker than $\neg requested(N_1, N_1)$.

Strongest possible invariant set. Finally, after traversing all subtemplates, DistAI unions together the subtemplate invariant sets to form the initial set of generated invariants (line 17) which will be fed to IVy. Since all possible candidate invariants have been considered for each subtemplate, we can prove that, for any invariant inv (in the form of disjunctions of literals) under template τ with a maximum number of literals of no more than l, there must exist an invariant inv' in the constructed invariant set such that $inv' \Rightarrow inv$. General invariants can be converted into conjunctive normal form (CNF) and then split into multiple invariants in the form of disjunctions of literals. Thus, the initial invariant set for a subtemplate is a *strongest possible* invariant set that is guaranteed to be at least as strong as the inductive invariant if there are no more than τ variables, also known as quantifiers, and l literals.

Intuitively, since each subtemplate provides the strongest possible invariant set, the invariant checked by IVy, which is constructed using the conjunction of all invariants across the union of subtemplate invariant sets, should also be the strongest with regard to the subsamples. In practice, when unioning the invariant sets, we can exclude invariants generated by projection from predecessors because they can be implied by their original counterparts. We formalize this in Theorem 1:

Theorem 1. *Let I^* be the output of Algorithm 2. For any invariant set I under template τ with a*

maximum number of literals of no more than l, if I is satisfied by every subsample, then $I^* \Rightarrow I$.

Proof. First we consider a variant of Algorithm 2 where Line 17 does not exclude invariants generated by projection. We prove this by contradiction. Suppose there exists I under template τ with a maximum number of literals of no more than l, I is satisfied by every subsample and $I^* \not\Rightarrow I$. Consider any invariant $\tau' : inv \in I$ but $\tau' : inv \notin I^*$. Recall each individual invariant is a disjunction of literals, assuming CNF. Since I is satisfied by every subsample, $\tau' : inv$ is also satisfied by every subsample. If we reach lines 12-13 in Algorithm 2, it will be added to the invariant set. The only possibility of $\tau' : inv \notin I^*$ is that the branch condition at line 10 evaluates to $true$. However, this indicates that a subset of inv is already in the invariant set. The subset of inv implies inv (e.g., $p \vee q \Rightarrow p \vee q \vee r$). So we still have $I^* \Rightarrow \tau' : inv$. To conclude, every $\tau' : inv \in I$ but $\tau' : inv \notin I^*$ can be implied by I^*, which is a contradiction to $I^* \not\Rightarrow I$.

Now we exclude invariants generated by projection and get a new I^*_{new}. From lines 15-16, every excluded invariant can be implied by another invariant in its predecessor subtemplate, so we can show $I^* \Leftrightarrow I^*_{new}$, thus completing the proof. □

Constants and function symbols. Although the discussion above assumes a literal can only be a predicate or its negation, DistAI also supports constants and function symbols as literals. For example, given a template $\{\forall X\ Y \in T\}$ and a constant $c \in T$, DistAI considers $X = c$ and $Y = c$ as two independent predicates and reasons about them like any other predicate. As another example, given a template $\{\forall X_1\ X_2 \in T_1,\ Y_1 \in T_2\}$ and a function $f : T_1 \rightarrow T_2$, DistAI can introduce $Y_2 = f(X_1), Y_3 = f(X_2)$ and treat Y_2, Y_3 as variables like Y_1.

2.4 Monotonic Invariant Refinement

When DistAI feeds the enumerated invariants to IVy, IVy may find that the conjunction of the invariants and the safety property are not inductive and return a list of invariants that failed. This is likely to happen at least for the initial invariants that DistAI enumerates as its sampling is not guaranteed to be complete. Because sampling is not complete and is primarily to improve

performance, DistAI may generate invariants that would not hold if sampling was done for more protocol instances. In general, when IVy indicates that an invariant fails, it is difficult to know whether the solution is to weaken or strengthen the invariant. Prior work uses different methods to evade this challenge but gives no fundamental solution [23, 24].

DistAI provides a simple and clean solution to this problem by starting with the strongest possible invariants and ensuring that the invariants remains the strongest possible ones throughout the refinement process. For each invariant that fails, which we refer to as a broken invariant, DistAI applies *mimimum weakening* to the invariant. The candidate invariant space becomes strictly smaller after each failure. DistAI ensures that the conjunction of the weakened invariants will remain stronger than the eventual invariants that must be added to the safety property to make it inductive, if it is expressible under the current template τ and maximum number of literals l. The overall process is guaranteed to converge to find the inductive invariant.

Algorithm 3 shows the minimum weakening algorithm used, given an initial invariant set and a broken invariant. We denote an invariant as $\tau' : inv$, where τ' is the subtemplate under which inv is found and inv is the invariant expressed as a disjunction of literals. The algorithm consists of three steps. First, DistAI removes the broken invariant from the initial invariant set. When IVy returns that $\tau'_0 : inv$ fails, DistAI removes $\tau' : inv$ from the invariant set that was initially passed to IVy (line 1).

Second, DistAI finds all weakened versions of the broken invariant and adds them back to the invariant set. A weakened version of $\tau' : inv$ is created by add one more literal via disjunction to inv (lines 2-7). For example, suppose $inv = p \vee \neg q$ is rejected by IVy. Recall that during the invariant enumeration, since $p \vee \neg q$ was considered as an invariant, for all literals r, the candidate $p \vee \neg q \vee r$ would be skipped. Now, for any literal r, $p \vee \neg q \vee r$ becomes a meaningful invariant. DistAI updates the invariant set by adding the weakened invariants back to the invariant set as long as they can not be implied by some other invariant that is already in the invariant set (e.g., $p \vee r$). If the broken invariant has reached the maximum number of literals, this second step will be skipped.

Third, DistAI projects the broken invariant to higher subtemplates, and adds all such projec-

Algorithm 3 Minimum Weakening Algorithm.

Input: Invariant set $I[\tau']$ for each subtemplate τ', and the broken invariant $\tau'_0 : inv$
Output: Updated invariant set $I[\tau']$ for each subtemplate τ'

```
1:  I[τ'_0] := I[τ'_0] \ {inv}
2:  if inv.length < l then
3:     for literal ∈ valid_literals(τ'_0) do
4:        if literal ∉ inv then
5:           new_inv := inv ∪ {literal}
6:           if ¬ check_subset_exists(new_inv, I[τ'_0]) then
7:              I[τ'_0] := I[τ'_0] ∪ {new_inv}
8:  for τ'_succ ∈ successors(τ'_0) do
9:     new_invs:=proj_inv(inv, τ'_0, τ'_succ)
10:    for new_inv ∈ new_invs do
11:       I[τ'_succ] := I[τ'_succ] ∪ {new_inv}
```

tions. For each successor τ'_{succ} of τ'_0, DistAI adds all the projections of inv on τ'_{succ} to the invariant set (line 8-11). To see why this is necessary, consider the following candidate invariant in some leader election protocol:

$$\forall X \in T.\ \neg leader(X). \tag{2.8}$$

This asserts no one can be a leader. This invariant may fail in IVy because the SMT solver observes a system state $\{leader(i_1),\ \neg leader(i_2),\ i_2 < i_1\}$ (suppose T has total order). Recall that DistAI uses a traversal order to enumerate invariants under different subtemplates, and the invariants from smaller subtemplates will be projected to larger subtemplates to avoid repeated enumeration. So previously, the following two candidate invariants, under a larger subtemplate $\{\forall X < Y \in T\}$, were skipped because they could be implied by Invariant (2.8).

$$\forall X < Y \in T.\ \neg leader(X) \tag{2.9}$$

$$\forall X < Y \in T.\ \neg leader(Y). \tag{2.10}$$

But now, after Invariant (2.8) is invalidated and removed, we need to reconsider Invariants (2.9)

and (2.10) and add them to the candidate invariant set. We validate the new invariants using IVy again. If successful, DistAI outputs the current invariant set as the inductive invariant, otherwise we will enter the next refinement round. In this case, if the distributed protocol has the property that only the greatest user can be the leader, then Invariant (2.10) will be invalidated in a later round, while Invariant (2.9) is likely to be correct and remain valid to the end.

The three-step minimum weakening procedure guarantees after any number of refinement rounds, the invariant set is always a strongest possible one that is satisfied by *all* the subsamples. This "strongest possible" property implies that throughout refinement, the invariant set is always stronger than the correct invariant set required for an inductive invariant, so whenever an invariant fails, we should always weaken the broken invariant. The guarantee can be formally stated as:

Theorem 2. *Let I^* be the invariant set after n refinement rounds, and $B_n = \{\tau_1' : inv_1, \tau_2' : inv_2, ..., \tau_n' : inv_n\}$ be the broken invariants in each round. For any invariant set I under template τ with no more than l literals, if I is satisfied by every subsample, and does not imply any broken invariant in B_n, then $I^* \Rightarrow I$.*

Proof. We prove this by induction on the number of rounds. The base case is simple. In Round 0, there is no broken invariant, and the statement degenerates to Theorem 1. Now we focus on the induction case. Suppose after k refinement rounds, we get invariant set I_k^*. For any invariant set I under template τ with no more than l literals, if I is satisfied by every subsample, and does not imply any broken invariant in B_k, then $I_k^* \Rightarrow I$.

Now we come to round $k+1$. We prove by contradiction. Suppose we have an invariant set I under template τ with no more than l literals such that 1) I is satisfied by every subsample, 2) I does not imply any broken invariant in B_{k+1}, and 3) $I_{k+1}^* \not\Rightarrow I$. Consider any invariant $\tau' : inv$ such that $I \Rightarrow \tau' : inv$ but $I_{k+1}^* \not\Rightarrow \tau' : inv$. From the induction hypothesis, we know $I_k^* \Rightarrow \tau' : inv$. Let $\tau_{k+1}' : inv_{k+1}$ be the invalidated invariant in round $k+1$. From the algorithm, $\tau_{k+1}' : inv_{k+1}$ is the only removed invariant in this round. Since each invariant is a disjunction of literals, we can show $\tau_{k+1}' : inv_{k+1} \Rightarrow \tau' : inv$. In other words, the hypothetical "missing" invariant must be implied by the removed invariant. We further know either inv includes more literals than inv, or τ' includes

more quantified variables not in τ'_{k+1}, or both. Otherwise we have $\tau' : inv \Rightarrow \tau'_{k+1} : inv_{k+1}$. Then $\tau' : inv = \tau'_{k+1} : inv_{k+1}$, a contradiction to $I \Rightarrow \tau' : inv$ and I does not imply the broken invariant $\tau'_{k+1} : inv_{k+1}$.

Now we consider the two cases separately. 1) inv includes a literal p not in inv_{k+1}. Consider the formula $F = \tau_{k+1} : inv_{k+1} \vee p$. From $\tau'_{k+1} : inv_{k+1} \Rightarrow \tau' : inv$, we can show $F \Rightarrow \tau' : inv$. However, F is added to the new invariant set I^*_{k+1} unless it can be implied by existing invariants (Line 3-7 in Algorithm 3). So we have $I^*_{k+1} \Rightarrow F \Rightarrow \tau' : inv$. 2) τ' includes a quantified variable X not in τ'_{k+1}. Consider the formula $G = \tau'' : inv_{k+1}$, where τ'' is τ' extended with X. Again, from $\tau'_{k+1} : inv_{k+1} \Rightarrow \tau' : inv$, we can show $G \Rightarrow \tau' : inv$. However, G is added to the new invariant set I^*_{k+1}(Line 8-11 in Algorithm 3). So we have $I^*_{k+1} \Rightarrow G \Rightarrow \tau' : inv$. In both 1) and 2), we reach $I^*_{k+1} \Rightarrow \tau' : inv$, which means the "missing" invariant is already implied by the existing invariant set output by the algorithm, a contradiction. □

Intuitively, Theorem 2 ensures that starting from a too strong invariant set, the minimum weakening steps never over-weaken the invariants and "bypass" the correct invariants in between. Combined with Theorem 1, which guarantees that monotonic refinement indeed starts from the strongest invariant set, we have the following corollary:

Corollary 1. *If there exists a correct invariant set expressible with template τ and maximum number of literals l, then the refinement procedure will terminate with one such invariant set within a finite number of rounds, otherwise the refinement procedure will terminate with a broken safety property.*

Sometimes, the weakened versions of a broken invariant are all discarded in the end. Then, minimum weakening provides no benefits versus just removing the broken invariants. In practice, DistAI first applies only the first step of minimum weakening — removing the broken invariants. Then if failed, DistAI applies refinement again with the first and second step. If failed again, DistAI applies the standard three-step minimum weakening in Algorithm 3. This practice optimizes performance when the weakened versions of a broken invariant are all discarded while maintaining Theorem 2 and Corollary 1.

If available, DistAI can use counterexamples to check weakened invariants, only adding them if they satisfy the counterexamples. However, DistAI's refinement procedure currently does not use them because obtaining counterexamples from IVy for an entire invariant set is extremely inefficient. When IVy is configured to return a counterexample, it halts early and returns the counterexample once it identifies the first broken invariant in a set. Using IVy in this configuration would force DistAI to weaken broken invariants one at a time and perform many redundant SMT checks of the invariant set through IVy, instead of weakening all failed invariants at once between each IVy call.

Convergence and Feedback loop. Since the invariant set is weakened after each refinement round, we can prove that the refinement procedure terminates in a finite number of rounds, resulting in an inductive invariant set.

If the safety property has never been violated during the refinement process, the resulting set is the correct inductive set and can derive the desired safety property. If, at any point, the safety property is violated and needs to be weakened (when all other candidates are weak enough), it means that the correct invariant set cannot be expressed under the current formula size, with its per-domain template size and maximum number of literals. DistAI will then increase the formula size, increasing the per-domain template size or the maximum number of literals, and relearn the invariants. By default, DistAI increases the formula size by alternating between increasing the maximum number of literals or increasing the template size, the latter by increasing the number of quantified variables for each domain in the template. For example, in Figure 2.4, i) we first increase the maximum number of literals by one, ii) if it fails, increase the template size by adding a new variable in type T_1, iii) if fails again, add another new variable in T_2, iv) and if still fails, increase the maximum number of literals by one again.

After increasing the formula size, we redo sampling, enumeration, and refinement. Since any invariant contains a finite number of quantified variables and a finite number of literals, the feedback loop will eventually reach a template and literal size large enough to express the correct

invariant set if one exists. Once a sufficient formula size is reached, Corollary 1 guarantees that a correct invariant set will be generated. Therefore, DistAI provides the following end-to-end convergence guarantee:

Theorem 3. *If the safety property of a protocol is provable with a $\exists$-free invariant set, then DistAI will terminate with one such invariant set in finite time.*

Theorem 3 guarantees conditional convergence of DistAI. However, if the safety property does not hold for the protocol or existential quantifiers are necessary to prove it correct, DistAI may continue in the feedback loop forever.

2.5 Evaluation, Results, and Discussions

To demonstrate its effectiveness at determining inductive invariants, we implemented and evaluate DistAI on a collection of 14 distributed protocols, including all 7 protocols previously evaluated with I4 [12]. The implementation consists of 1.6K lines of Python code for protocol simulation and sampling and 1.6K lines of C++ code for enumeration and refinement. For comparison, we also evaluated I4 and FOL-IC3, in both cases using the implementations created by the original authors. All experiments were performed on a Dell Precision 5829 workstation with a 4.3GHz 28-core Intel Xeon W-2175, 62GB RAM, and a 512GB Intel SSD Pro 600p. Table 2.1 shows the results for each protocol, including the number of domains and relations for each protocol as indicators of protocol complexity.

DistAI outperforms both I4 and FOL-IC3 in terms of the number of protocols for which it infers the correct invariants. DistAI automatically infers the correct invariants for 13 out of the 14 protocols, only failing for Paxos, on which both I4 and FOL-IC3 also fail. I4 only solves 9 protocols, and FOL-IC3 only solves 3 protocols using its default setting, which searches over all first-order logic formulas, but improves to solving 9 protocols if an option is enabled that limits the search space to only $\forall$ quantifiers. Each approach was allowed to run for an entire week, 168 hours, per protocol before timing out, more than two orders of magnitude longer than the worst runtime reported in Table 2.1.

Distributed Protocol	Domains	Relations	Variables	Literals	Refinements	Invariants
asynchronous lock server [25]	2	5	3	2	0	12
chord ring maintenance [11]	1	8	3	4	48	163
database chain replication [11]	4	13	7	4	158	66
decentralized lock [26]	2	2	4	2	150	16
distributed lock [11]	2	4	4	3	82	45
hashed sharding [13]	3	3	5	2	0	15
leader election [11]	2	3	6	3	0	17
learning switch [11]	2	4	4	3	8	71
lock server [11]	2	2	2	2	0	1
Paxos [27, 28, 29]	4	9	-	-	-	-
permissioned blockchain [30]	4	10	6	3	2	13
Ricart-Agrawala [19]	1	3	2	2	0	6
simple consensus [13]	3	8	5	3	19	50
two-phase commit [12]	1	7	2	3	3	30

Table 2.1: Evaluation results on 14 distributed protocols from multiple sources.

Distributed Protocol	DistAI time(s)	I4 time(s) final	I4 time(s) total	FOL-IC3 time(s) ∀	FOL-IC3 time(s) default
asynchronous lock server[25]	1.1	generalize fail[s]		6.9	-[*]
chord ring maintenance[11]	52.8	586.1[‡]	594.4	-[*]	-[*]
database chain replication[11]	58.8	20.2[‡]	63.1	-[*]	Z3 fail
decentralized lock[26]	9.4	generalize fail[s]		37.1[d]	Z3 fail
distributed lock[11]	12.6	152.1[‡]	204.7	1451.3	Z3 fail
hashed sharding[13]	1.1	nondet fail[s]		9.2	-[*]
leader election[11]	1.9	4.9[‡]	4.9	26.3	-[*]
learning switch[11]	27.6	10.5[‡]	12.4	-[*]	-[*]
lock server[11]	0.8	0.5[‡]	0.8	0.5	2.1
Paxos[27, 28, 29]	-[*]	-[*]	-[*]	-[*]	-[*]
permissioned blockchain[30]	4.9	blackbox fail[s]		21.2	-[*]
Ricart-Agrawala[19]	0.9	0.8	0.8	0.7	3.2
simple consensus[13]	23.3	41.8	68.7	-[*]	-[*]
two-phase commit[12]	1.9	3.1[‡]	8.0	3.4	7.9

[*] Time out after 1 week.

[‡] I4 runtimes on our machine are similar (6 out of 7 protocols slightly faster) to those previously reported for I4 [12].

[s] "generalize fail" means I4's implementation fails to convert invariants from the AVR model checker to generalized universally quantified invariants. "nondet fail" means failed on nondeterministic initialization. "blackbox fail" means error triggered on reasoning of blackbox functions.

[d] FOL-IC3 initially completed in less than a second, but this turned out to be incorrect due to a bug in the mypyvy protocol specification used by FOL-IC3, which does not exist in the Ivy protocol specification used by DistAI and I4.

Table 2.1: Evaluation results on 14 distributed protocols from multiple sources (continued).

DistAI and I4 only time out trying to solve Paxos, but FOL-IC3 times out on many protocols. This is because DistAI only uses the SMT solver for validating rather than generating invariants, I4 uses a model checker to generate invariants only for a specific, small instance, while FOL-IC3 invokes the SMT solver to generate invariants for the general protocol, multiple times for each invariant, which is undecidable in general and very expensive in practice. FOL-IC3 performs worse with the default setting since the formula search space is larger and the SMT solver performs worse for formulas with existential quantifiers. In fact, FOL-IC3 fails for database chain replication, decentralized lock, and distributed lock with the default setting because Z3, the underlying SMT solver, fails and reports `unknown`, indicating that the formula generated by FOL-IC3 does not fall in the supported decidable fragment of first-order logic. In contrast, DistAI never generates an undecidable formula.

Although both DistAI and I4 fail to solve Paxos, a complex and realistic consensus protocol, the reasons for the failures are different. I4 fails because its model checker is unable to produce any candidate invariants. Model checking is complex and quite resource intensive, and I4's authors report its model checker runs out of memory trying to solve Paxos [12]. In contrast, DistAI produces candidate invariants, but it fails because it does not support invariants with existential quantifers, which Paxos requires; I4 also has this limitation. Upon failed refinement, DistAI keeps increasing the formula size until it times out or exhausts memory. By manual inspection, we find that DistAI infers all $\exists$-free invariants for Paxos. FOL-IC3 supports finding invariants with existential quantifiers, but it also fails to solve Paxos, the one protocol in our evaluation with existential quantifiers.

The most common reason overall why I4 fails to solve protocols is its dependency on modeling checking a small size implementation of the protocol to generate candidate invariants. I4 also fails to infer the correct invariants for decentralized lock and asynchronous lock server because it cannot generalize the candidate invariants generated by the model checker for a small size implementation to universally quantified invariants. Although I4 succeeds on lock server, it fails on asynchronous lock server because the latter explicitly models packet loss in the network, resulting

in more complex invariants.

DistAI takes a fundamentally different approach that does not require model checking a finite instance. DistAI operates in formula space, allowing it to enumerate invariants that hold for any instance size. It optimizes the enumeration by running protocol simulations across different size systems, but does not rely on the simulations to find candidate invariants, only to reduce the number of invariants it needs to enumerate. By taking this data-driven approach, it is able to produce better initial invariants to achieve greater success with more protocols and guarantee success if there are no invariants with existential quantifiers. Unlike I4, DistAI is simple and self-contained, avoiding the need for, and dependence on, a complex external model checker that, like all complex software, may have bugs.

Permissioned blockchain is another example that demonstrates the effectiveness of DistAI. It has a blackbox Byzantine broadcast protocol as a subprocedure. In permissioned blockchain, n users have a synchronized clock. At epoch E, only one user n_E, the round-robin leader of the epoch, can (optionally) propose a block, if it has found a valid one extending its longest chain. The epoch leader uses the Byzantine broadcast protocol to broadcast the block in the P2P network. An honest user always adds all outstanding transactions in the block it proposed and follows the Byzantine broadcast protocol, while an adversary can neglect certain transactions, delay block proposal, and send conflicting block messages to any node at any epoch, regardless of who the leader is. A Byzantine broadcast protocol satisfies *agreement*, if all honest users always share the same eventual result regardless of the leader is honest or not. A Byzantine broadcast protocol satisfies *validity*, if when the leader is honest, all honest users will eventually decide on the message of the leader. A blockchain satisfies *consistency*, if at any epoch, all honest users have the same view of the blockchain (i.e., no forks or orphaned blocks). That is, for any two honest users at any time, a block is either confirmed by both, or confirmed by none. A blockchain satisfies *liveness*, if all transaction will be confirmed within a finite number of epochs.

DistAI successfully proves that for any Byzantine broadcast procedure that satisfies agree-

ment and validity, the resulting permissioned blockchain satisfies consistency and liveness[1]. The Byzantine broadcast procedure is described by pre-conditions and post-conditions in IVy without the need for an executable implementation. When simulating the blackbox Byzantine procedure, DistAI simply picks a random state that satisfies the post-condition as the execution result. This random selection may leave corner cases uncovered, but the eventual correctness is guarded by the SMT solver, and we monotonically weaken the invariant set to reach a correct solution. In contrast, the use of a blackbox procedure poses difficulty and triggers errors in I4. We should note DistAI solves permissioned blockchain for not one, but any valid implementation of the Byzantine broadcast protocol because it does not depend on or require its implementation, a key benefit of our approach.

DistAI also outperforms both I4 and FOL-IC3 in terms of the time required to infer the correct invariants. For I4, we report both the runtime for the final instance size on which the correct invariant is generated, as reported in [12], as well as the total runtime, which includes trying increasingly larger instance sizes that fail until the final instance size succeeds. Except for learning switch, DistAI is about the same or faster than I4 for all of the protocols solved by I4, up to an order of magnitude faster. The runtime comparisons between DistAI and I4 are conservative as they do not include the time required for concretization [12], a step required by I4 to manually introduce additional constraints to the protocol to limit the search space of the model checker. DistAI is also faster than FOL-IC3 for all but the two simplest protocols solved by FOL-IC3, in many cases by more than one to two orders of magnitude. This is because SMT queries are expensive and FOL-IC3 uses them extensively.

Table 2.1 also shows for DistAI the number of invariants identified for the correct invariant set, the total number of refinement steps required (i.e., the number of times Algorithm 3 is called), the total number of quantified variables of all domains in the final template used, and the maximum number of literals used for each protocol. Most protocols have a maximum number of literals of 2

[1]We prove a variant of the liveness property — if the leader of epoch T is honest, then all transactions before T will be confirmed at T. The original liveness property cannot be encoded as a safety property, thus falling out of the scope of DistAI, I4, and FOL-IC3.

Distributed Protocol	Sample	Enumerate	Refine	Total
asynchronous lock server	0.6	0.0	0.5	1.1
chord ring maintenance	11.1	1.2	40.5	52.8
database chain replication	36.3	0.1	24.4	58.8
decentralized lock	2.1	0.1	7.2	9.4
distributed lock	0.8	0.1	11.7	12.6
hashed sharding	0.6	0.1	0.4	1.1
leader election	0.8	0.1	1.0	1.9
learning switch	19.4	0.3	7.9	27.6
lock server	0.5	0.0	0.3	0.8
permissioned blockchain	3.5	0.1	1.3	4.9
Ricart-Agrawala	0.5	0.0	0.4	0.9
simple consensus	18.8	0.4	4.1	23.3
two-phase commit	0.5	0.1	1.3	1.9

Table 2.2: DistAI runtime breakdown in seconds for each protocol.

or 3, and in two cases 4. This validates our key assumption that inductive invariants of distributed protocols should be human-readable and concise. DistAI uses invariant refinement to address missing cases during sampling for all but the five simplest protocols, Ricart-Agrawala, hashed sharding, leader election, lock server, and asynchronous lock server, in which no refinement is needed as the subsample set is complete and the correct invariant is learned without refinement.

Runtime breakdown. Table 2.2 provides a breakdown of the total runtime using DistAI for each protocol. One can see the bottleneck is either sampling or refinement, but not enumeration. Sampling is expensive when the argument space for actions is sparse because DistAI randomly selects arguments so it can end up trying many arguments that are invalid for each set of valid arguments, increasing the simulation runtime. This is the reason why sampling is most expensive for database chain replication, a protocol that guarantees serializability and atomicity for distributed databases. A transaction is split into subtransactions that operate sequentially on data that is sharded across multiple nodes. For one subtransaction to commit, it must operate on the correct node and satisfy a set of constraints (e.g., no uncommited previous writes). Most randomly selected subtransactions will not satisfy these constraints. As a result, sampling spends significant time finding valid arguments because most arguments are invalid.

Sampling is also expensive for learning switch, the only protocol for which DistAI is slower

than I4. One reason is the argument space for actions is sparse, so it takes a while to find a data sample, but the other is because the subsample space is too large to explore. With learning switch, each node maintains a routing table that matches destination addresses to outbound ports (i.e., neighbors), and updates the table upon receiving new packets. It has a 3-ary single-domain relation $route\_tc(N_1, N_2, N_3)$, which means the routing trace from N_2 to N_1 includes N_3. Under template $\forall N_1\, N_2\, N_3$, this single relation yields 27 predicates ($route\_tc(N_1, N_1, N_1), route(N_1, N_1, N_2), ...$). There are 60 predicates across all relations, meaning that each subsample is a 60-bit vector, so the candidate subsample space has size 2^{60}. Although valid subsamples are sparse, DistAI generates 33K subsamples before it cannot find anymore and terminates. This takes a while.

Refinement can be the dominant factor in performance, as is the case for chord ring maintenance, database chain replication, and distributed lock, but Table 2.2 shows that DistAI is successful overall at avoiding substantial SMT query costs as refinement runtime, which includes the cost of IVy checking the initial candidate invariants, is modest in most cases.

Figure 2.5 shows how sampling helps reduce the cost of refinement for the simple consensus protocol. DistAI has provable guarantees to find the correct invariants for any sample sizes, but if the number of samples is too small (e.g., 100 in Figure 2.5), it takes much longer due to many more SMT queries on refinement. Increasing the number of samples increases sampling time roughly linearly but decreases refinement time roughly exponentially. However, once a minimum threshold of samples is met, it becomes more of an even tradeoff. Sampling can be faster by obtaining fewer samples, but because more corner cases are missing, the refinement process takes longer to "fix" the invariants through monotonic weakening. Conversely, more samples require more time to simulate the protocol, while the refinement process will be faster. The default sampling parameters, discussed in Section 2.2 and used for all experiments, resulted in 50K samples for the simple consensus protocol.

We also reran the protocol experiments with DistAI for other sampling parameters, ranging from $MA = 25$ to $MA = 100$, and $SD = 2$ to $SD = 5$. In all cases, DistAI was able to solve the same 13 protocols with mostly similar runtimes and in the worst case, three times slower runtimes

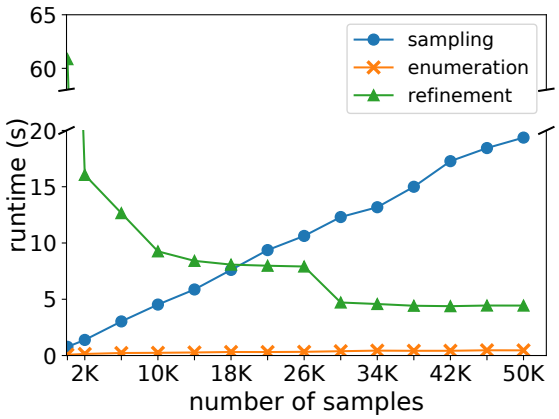


Figure 2.5: Runtime breakdown of DistAI on simple consensus.

than the defaults. Detailed runtimes are omitted due to space constraints.

Chapter 3: DuoAI: Faster and More Effective Inductive Invariant Inference

In the last chapter, I presented DistAI, the first data-driven invariant inference tool for distributed protocols. DistAI outperforms pre-existing alternatives. However, it still has two significant limitations. It does not support inferring invariants with existential ($\exists$) quantifiers, and when reasoning about complicated protocols, the number of candidate invariants can reach hundreds or even thousands, leaving SMT solvers timed out when validating the invariants. We further observe that timeouts become more frequent, even with only dozens of invariants, when a significant number of them involve both $\forall$ and $\exists$. To address these problems, I propose and develop DuoAI, a faster and more effective invariant inference tool. DuoAI inherits the simulation-enumeration-refinement pipeline of DistAI, but more effectively capture the structure of the invariant search space via the minimum implication graph (§3.2), upon which the invariant enumeration is based to achieve minimum redundancy and maximum efficiency (§3.3). The top-down refinement procedure (§3.4) largely mirrors and extends the monotonic refinement procedure in DistAI, but to alleviate the SMT solvers from expensive queries with too many invariants, DuoAI introduces a novel complimentary bottom-up refinement procedure that runs in parallel and significantly reduces the complexity of SMT queries (§3.5). §3.6 introduces further optimizations and §3.7 shows evaluation results.

3.1 DuoAI workflow

We use a simplified consensus protocol as an example to show how DuoAI works. Figure 3.1 shows the protocol written in Ivy [11], a language and tool for specifying, modeling, and verifying distributed protocols built on top of the Z3 SMT solver. Each node can vote for another node to be the leader, and when a node receives votes from a quorum of nodes, it can become the leader

```
1  type value
2  type quorum
3  type node
4
5  relation vote(N1:node, N2:node)
6  relation voted(N:node)
7  relation leader(N:node)
8  relation decided(N:node, V:value)
9  relation member(N:node, Q:quorum)
10 axiom forall Q1, Q2. exists N. member(N, Q1) & member(N, Q2)
11
12 after init {
13     voted(N) := false;
14     vote(N1, N2) := false;
15     leader(N) := false;
16     decided(N, V) := false;
17 }
18
19 action cast_vote(n1: node, n2: node) = {
20     require ~voted(n1);
21     vote(n1, n2) := true;
22     voted(n1) := true;
23 }
24
25 action become_leader(n: node, q: quorum ) = {
26     require forall N. member(N, q) -> vote(N, n);
27     leader(n) := true;
28 }
29
30 action decide(n:node, v: value) = {
31     require leader(n);
32     require forall V. ~decided(n, V);
33     decided(n, v) := true;
34 }
35
36 invariant decided(N1,V1) & decided(N2,V2) -> V1=V2
```

Figure 3.1: The simplified consensus protocol written in Ivy. Capitalized variables are implicitly quantified. For example, Line 16 means $\forall\, N : node, V : value.\ decided(N, V) := false$. "~" stands for negation.

and decide on a value. The protocol state at any moment is represented by five relations (Lines 5-9). $vote(n_1, n_2)$ indicates whether node n_1 has voted for node n_2. $voted(n)$ indicates whether node n has ever casted a vote. $leader(n)$ indicates if n is the leader among nodes. $decided(n, v)$ indicates whether node n has decided on value v. $member(n, q)$ indicates if node n belongs to quorum q, where each quorum is a set of nodes. The axiom (Line 10) dictates a property of the member relation: any two quorums of nodes must have at least one node in common. After initialization (Lines 13-16), the protocol can non-deterministically transition from one state to another as described by the three actions $cast\_vote$, $become\_leader$, and $decide$ (Lines 19-34).

For example, $cast\_vote(n_1, n_2)$ lets a node n_1 vote for another node n_2, under the precondition that n_1 has not voted before (Line 20). Then the protocol will transition to a new state where $vote(n_1, n_2) = true$ and $voted(n_1) = true$. Finally, the safety property (Line 36) encodes the desired property of correctness of the protocol that the system cannot decide on two different values.

The safety property is an invariant of the protocol, but is not inductive as taking an action from a state satisfying the safety property may result in a new state that breaks the safety property. To verify the protocol, we need four additional invariants:

$$\forall N_1, N_2 : node.\ vote(N_1, N_2) \rightarrow voted(N_1) \tag{3.1}$$

$$\forall N_1, N_2, N_3 : node.\ vote(N_1, N_2) \wedge vote(N_1, N_3) \rightarrow N_2 = N_3 \tag{3.2}$$

$$\exists Q : quorum.\ \forall N_1, N_2 : node.$$

$$leader(N_1) \wedge member(N_2, Q) \rightarrow vote(N_2, N_1) \tag{3.3}$$

$$\forall N : node.\ V : value.\ decided(N, V) \rightarrow leader(N). \tag{3.4}$$

The first invariant says that if a node has voted for another node, then it must be recorded as $voted$ in the protocol. The second says that one node cannot vote for two different nodes. The third says that a leader must be endorsed by a quorum of nodes. More specifically, we can find a quorum Q that every node N_2 in the quorum must have voted for the leader N_1. The fourth says that only a leader can decide on a value. The conjunction of the four invariants and the safety property is inductive.

To find this inductive invariant, DuoAI simulates the protocol using different instance sizes and logs the samples. It then builds a minimum implication graph, a small fragment of which is shown in Figure 3.2. The full graph for simplified consensus has over 35K nodes and 170K edges. Nodes represent formulas and edges represent implication between formulas. A stronger formula will have a directed edge to an implied weaker formula. DuoAI enumerates possible candidate invariants following the graph and adds it to the candidate invariant set if it holds on the samples. For example, DuoAI checks the root node in Figure 3.2 and it does not hold on the samples. DuoAI

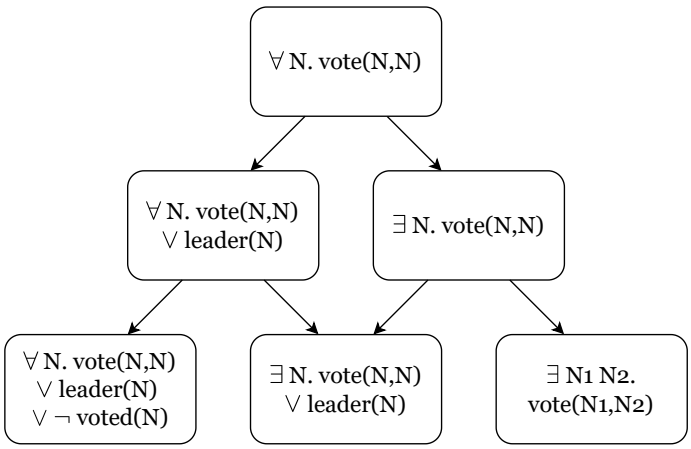


Figure 3.2: Fragment of the minimum implication graph for the simplified consensus protocol.

then checks its implied weaker formulas, the two nodes in the second layer, iteratively going down the graph. For the simplified consensus protocol, enumeration ends with 19 candidate invariants, including equivalent forms of Eq. (3.1), (3.2), (3.3), and (3.4).

After enumeration, DuoAI runs top-down and bottom-up refinement in parallel. Top-down refinement feeds all candidate invariants and the safety property to Ivy to see if their conjunction is inductive. For simplified consensus, the conjunction is inductive, so no further weakening is required. Bottom-up refinement feeds all ∀-only invariants from the initial candidate set to Ivy then weakens them until the set of invariants is itself inductive, but may not imply the safety property. For simplified consensus, this universal core includes three invariants Eq. (3.1), (3.2), and (3.4). DuoAI then tries to search a small number of ∃-included invariants to add to the universal core along with the safety property so that the resulting set is inductive. DuoAI uses counterexamples from Ivy to guide the search for additional invariants and eventually identifies invariant (3.3) for the simplified consensus protocol, forming an inductive invariant set. For simplified consensus, top-down refinement succeeds more quickly than bottom-up refinement.

3.2 Minimum Implication Graph

The backbone of DuoAI is the minimum implication graph, which encodes implication relations among formulas. The graph is used to determine the order of formulas to be enumerated,

and how invariants are weakened. We present formulas in prenex normal form, where the quantified variables, called the prefix, appear at the beginning of the formula followed by quantifier-free relations, called the matrix. The matrix is required to be in disjunctive normal form (DNF). For simplicity, here we only consider predicate symbols with equality. The methods can be extended to uninterpreted functions in the same manner as DistAI [16].

A formula P is strictly stronger than Q if $P \Rightarrow Q$ and $Q \nRightarrow P$. For two formulas $P, Q \in \mathcal{S}$, where $\mathcal{S}$ is a finite formula search space, there is a directed edge from P to Q in the minimum implication graph if and only if P is strictly stronger than Q and there is no formula R which is strictly weaker than P while strictly stronger than Q. For example, the fragment of the minimum implication graph in Figure 3.2 includes three formulas:

$$\forall N.\ vote(N, N) \tag{3.5}$$

$$\exists N.\ vote(N, N) \tag{3.6}$$

$$\exists N_1, N_2.\ vote(N_1, N_2) \tag{3.7}$$

Eq. (3.5) $\Rightarrow$ (3.6) since if $vote(N, N)$ is true for all N, there must exist some N for which it is true. Eq. (3.6) $\Rightarrow$ (3.7) since if $vote(N, N)$ is true for some N, there must exist some $N_1 = N_2$ for which $vote(N_1, N_2)$ is true. Because Eq. (3.5) $\Rightarrow$ (3.6) $\Rightarrow$ (3.7), there is an edge from Eq. (3.5) to (3.6), an edge from Eq. (3.6) to (3.7), but no edge from Eq. (3.5) to (3.7), because Eq. (3.6) is between them.

DuoAI defines the search space $\mathcal{S}$ as all formulas in disjunctive normal form for a given set of quantified variables and formula size. The formula size is defined by four parameters: max_exists sets the maximum number of existentially quantified variables, $max\_literal$ sets the upper bound of the total number of literals in the formula, max_and sets the maximum number of literals connected by AND, and max_or sets the maximum number of conjunctions connected by OR.

The minimum implication graph has two important properties as stated in Lemmas 1 and 2:

Lemma 1. *The minimum implication graph is a directed acyclic graph (DAG).*

Proof. Suppose there is a cycle $P_1 \to P_2 \to \ ... \ \to P_k \to P_1$. The edges $P_1 \to P_2, \ ... \ , \ P_{k-1} \to P_k$ imply that $P_1 \Rightarrow P_2, \ ... \ , \ P_{k-1} \Rightarrow P_k$. From the transitivity of $\Rightarrow$ we know $P_1 \Rightarrow P_k$. Since there is an edge from P_k to P_1, we know $P_1 \not\Rightarrow P_k$, a contradiction. □

Lemma 2. *For any $P, Q \in \mathcal{S}$, there is a path from P to Q in the minimum implication graph if and only if $P \Rightarrow Q \wedge Q \not\Rightarrow P$.*

Proof. We first prove the "if" direction by induction on the number of formulas in S that are strictly weaker than P while strictly stronger than Q. For the base case, if there are zero such formulas, then by definition there is an edge from P to Q. Next we prove the induction step. Suppose for any $P, Q \in S$, if $P \Rightarrow Q \wedge Q \not\Rightarrow P$, and there is no more than n formulas that are strictly weaker than P while strictly stronger than Q, then there is a path from P to Q. Now consider the case that there are $n+1$ formulas that are strictly weaker than P while strictly stronger than Q. Let R be one of the $n+1$ formulas. We know $P \Rightarrow R \wedge R \not\Rightarrow P$, and there can be no more than n formulas that are strictly weaker than P while strictly stronger than R. By the induction hypothesis, there is a path from P to R. In the same manner, we can show there is a path from R to Q. Then we concatenate the two paths and get a path from P to Q.

Next we prove the "only if" direction. If there is a path from P to Q, Let $P, F_1, F_2, ..., F_k, Q$ be the path. We know $P \Rightarrow F_1, \ ... \ , \ F_k \Rightarrow Q$, so $P \Rightarrow Q$. We prove $Q \not\Rightarrow P$ by contradiction. Suppose $Q \Rightarrow P$, then $P \Leftrightarrow Q$, so there must be an edge from F_k to P. This forms a cycle $P, F_1, ..., F_k, P$, a contradiction to Lemma 1. □

To build the minimum implication graph, we need to determine the "root" nodes in the graph, that is, formulas with no predecessors since they cannot be implied by any other formula, and how to find their successors. In DuoAI, a formula $P \in S$ is added to the set of root nodes if it falls into one of two cases:

1. P has no $\exists$-quantified variable and no logical OR. For example:

$$\forall N : node.\ vote(N, N) \wedge leader(N). \tag{3.8}$$

2. P has unique $\exists$-quantified variables and no logical OR. For example:

$$\exists N_1, N_2 : node.\ N_1 \neq N_2 \wedge vote(N_1, N_2). \tag{3.9}$$

Intuitively, if a formula has an $\exists$, then by changing it to a $\forall$, we can get a stronger formula. If a formula has a logical OR, then by removing the OR and any literals followed by it, we can get a stronger formula. So in general, a root formula should have no $\exists$ and no OR, such as Eq. (3.8). There is one exception, represented by Eq. (3.9). At first sight Eq. (3.9) has a predecessor $\forall N_1, N_2 : node.\ N_1 \neq N_2 \wedge vote(N_1, N_2)$. However, this formula is a contradiction because $\forall N_1, N_2 : node.\ N_1 \neq N_2$ cannot be true. The minimum implication graph does not include tautologies and contradictions, so Eq. (3.9) itself is a root formula.

Starting from the root nodes, DuoAI incrementally builds the minimum implication graph. For formulas $P, Q \in S$, DuoAI adds an edge from P to Q if the shapes of P and Q fall into one of five cases:

1. P and Q share the same matrix. Q replaces the $\forall$-quantified variables of one type with $\exists$-quantified variables. For example:

$$P = \forall N : node,\ V : value.\ \neg decided(N, V)$$
$$Q = \exists N : node.\ \forall V : value.\ \neg decided(N, V).$$

2. P and Q share the same prefix. Q has one less ANDed literal than P. For example:

$$P = \text{Eq.(3.8)} \qquad\qquad Q = \text{Eq.(3.5)}.$$

3. P and Q share the same prefix. Q has one more ORed conjunction than P. For example:

$$P = \forall N : node.\ vote(N, N)$$
$$Q = \forall N : node.\ vote(N, N) \vee (voted(N) \wedge leader(N)).$$

DuoAI requires that the ORed conjunction be maximal, which means it contains the maximum number of literals for the search space. The conjunction $voted(N) \wedge leader(N)$ in Q is maximal if $max\_and = 2$ or $max\_literal = 3$. For example, $Q' = \forall N : node.\ vote(N, N) \vee voted(N)$ also adds one more ORed conjunction from P, but DuoAI does not add an edge from P to Q', because Q is strictly stronger than Q'.

4. Starting from P, Q projects two $\forall$-quantified variables of the same type into one variable. For example:

$$P = \forall N_1, N_2 : node.\ vote(N_1, N_2) \vee leader(N_1)$$
$$Q = \forall N : node.\ vote(N, N) \vee leader(N).$$

5. Starting from Q, P projects two $\exists$-quantified variables of the same type into one variable. For example:

$$P = \text{Eq.(3.6)} \qquad\qquad Q = \text{Eq.(3.7)}.$$

The graph constructed in this way may differ slightly from the exact minimum implication graph due to equivalent formulas. For example, formulas $\forall X.\ p(X) \vee (\neg p(X) \wedge q(X))$ and $\forall X.\ p(X) \vee q(X)$ fall into the second case, so there is an edge in the constructed graph. However, the two formulas are equivalent so there is no edge in the exact graph. We call the graph constructed by DuoAI an *approximate minimum implication graph*, whose properties are formalized in Lemmas 3, 4, and 5:

Lemma 3. *The approximate minimum implication graph is a directed acyclic graph (DAG).*

Proof. For all of the five cases, we can show that for formulas along any path in the approximate minimum implication graph, there exists one function that is strictly increasing, so there can be no cycle. For example, this is true for the function $(\#\ \exists\text{-}variables) - (\#\ \forall\ variables) + (max\_and *$

(# $\vee$)) − (# $\wedge$), where # denotes "the number of" (e.g., (# $\vee$) is the number of logical OR in a formula). □

Lemma 4. *For any $P, Q \in \mathcal{S}$, there is a path from P to Q in the approximate minimum implication graph only if $P \Rightarrow Q$.*

Proof. From the transitivity of $\Rightarrow$, we only need to show that if there is an edge from P to Q in the approximate minimum implication graph, then $P \Rightarrow Q$. This can be proved by showing $P \Rightarrow Q$ holds in each of the five cases. The first three cases are trivial. For the fourth case, in general $P = ... \ \forall X_1 \, X_2 \ ... \ matrix(X_1, X_2)$ and $Q = ... \ \forall X_1 \ ... \ matrix(X_1, X_1)$. Let $P' = ... \ \forall X_1 \, X_2 \ ... \ X_1 = X_2 \rightarrow matrix(X_1, X_2)$, then $P \Rightarrow P' \Leftrightarrow Q$. Similarly, for the fifth case, $P = ... \ \exists X_1 \ ... \ matrix(X_1, X_1)$ and $Q = ... \ \exists X_1 \, X_2 \ ... \ matrix(X_1, X_2)$. Let $Q' = ... \ \exists X_1 \, X_2 \ ... \ X_1 = X_2 \wedge matrix(X_1, X_2)$, then $P \Leftrightarrow Q' \Rightarrow Q$. □

Lemma 5. *For any formula $P \in S$ that is not a tautology or a contradiction, there exists a directed path from a root node $Q \in S$ to P in the approximate minimum implication graph.*

Proof. We prove this by construction. For a $\exists$-free formula P, if it includes no logical OR, then it is a root formula itself. Otherwise, we find the root formula Q by removing all but one ORed conjunctions. Starting from Q, we can iteratively apply the second and third cases to add conjunctions and remove literals until we reach P. For a $\exists$-included formula P, if it includes unique $\exists$-quantified variables then it is a root formula itself. Otherwise, we iteratively find a predecessor by replacing $\exists$ with $\forall$ for quantified variables of each type, until the formula becomes the $\exists$-free P'. The first case guarantees that there is a path from P' to P, and we have already shown for the $\exists$-free P', there exists a path from a root node Q. Putting it together, we have a path from Q to P in the approximate minimum implication graph. □

In other words, the approximate minimum implication graph is as useful and complete as the exact graph. DuoAI uses the approximate minimum implication graph, which, for simplicity, we will continue to refer to as the minimum implication graph unless otherwise specified.

DuoAI requires that formulas in $\mathcal{S}$ must be in a decidable fragment of first-order logic. In general, satisfiability in first-order logic is undecidable [31], so an SMT solver can get stuck in infinite instantiations and never give the $sat/unsat$ answer. DuoAI ensures that the formulas are decidable by enforcing a fixed order of types if there is quantifier alternation (i.e., alternating $\forall$ and $\exists$) [32]. If type A is ordered before type B, then for any formula, if there exists a quantified variable V of type A, any quantified variable of type B can only occur after V if there is quantifier alternation. For example, if type $node$ is ordered before type $packet$, then $\forall N : node.\ \exists P : packet$ and $\exists N : node.\ \forall P : packet$ are allowed while $\forall P : packet.\ \exists N : node$ and $\exists P : packet.\ \forall N : node$ are not. DuoAI tries to infer the order of types from the protocol specification and obtains input from the user when necessary. For example, for the simplified consensus protocol, DuoAI can infer from Line 10 that type `quorum` must be ordered before type `node`, then ask the user to place type `value` in the order. Absent user input, DuoAI will try different possible orders in parallel.

3.3 Candidate Invariant Enumeration

Similar to DistAI [16], DuoAI first repeatedly simulates the distributed protocol using various instance sizes, and records the reached states as samples. For example, DuoAI simulates the simplified consensus protocol on concrete instances of different numbers of values, quorums, and nodes. The simulations of different instance sizes are done in parallel and yield samples of different lengths. DuoAI follows the minimum implication graph to enumerate candidate invariants, but rather than feeding all of them to an inefficient SMT solver, it checks them directly on the samples first. A correct invariant must hold on every reachable protocol state and thus on every sample. A key difference between DuoAI and DistAI is that DuoAI keeps the original variable-length samples and uses them in invariant enumeration, while DistAI projects all samples to fixed-length vectors that it calls subsamples. The problem is that DistAI is not exhaustive in its subsampling, so that a formula with existential quantifiers that holds for DistAI's subsamples may not actually hold for the original samples. DuoAI avoids this problem by effectively considering all possible subsamples that can be derived from the original samples.

Algorithm 4 Invariant Enumeration Algorithm

Input: Distributed protocol $\mathcal{P}$, invariant search space $\mathcal{S}$, a set of samples from protocol simulation $samples$
Output: Candidate invariants

```
1:  graph := build_minimum_implication_graph(P, S)
2:  candidates, invalidated := ∅, ∅
3:  pending := graph.rootNodes
4:  while pending.notEmpty() do
5:      f := pending.dequeue()
6:      if graph.ancestors(f) ∩ candidates ≠ ∅ then
7:          continue
8:      if check_inv_holds(f, samples) then
9:          candidates := candidates ∪ {f}
10:     else
11:         invalidated := invalidated ∪ {f}
12:         for next_f ∈ graph.successors(f) do
13:             if next_f ∉ candidates and next_f ∉ invalidated and next_f ∉ pending then
14:                 pending.enqueue(next_f)
15: return candidates
```

Algorithm 4 shows the enumeration algorithm, in which $pending$ is a queue whose elements are formulas that will be checked on the samples, $candidates$ is the set of formulas that hold on all the samples and $invalidated$ is the set of formulas that do not hold on at least one of the samples. Both $candidates$ and $invalidated$ are initially empty (Lines 2-3), and $pending$ initially consists of the root nodes of the minimum implication graph, that is, formulas that cannot be implied by any other formula. In each iteration, a formula f is popped from the $pending$ queue (Line 5). If one of f's ancestors in the graph has already been added to $candidates$, DuoAI will not check f on the samples or add f to the $candidates$ invariants (Lines 6-7). Otherwise, DuoAI will check f on the samples and if it holds, add it to $candidates$ (Lines 8-9). If f does not hold on at least one sample, DuoAI will add it to $invalidated$ (Line 11), and add its successors, which are formulas weaker than f, to the $pending$ queue if they have not already been added (Lines 12-14).

Figure 3.3 shows an example of invariant enumeration using the graph in Figure 3.2. DuoAI starts from the root nodes, iteratively goes down the minimum implication graph, and checks formulas against the samples. Because of this design, formulas D and E are never checked against

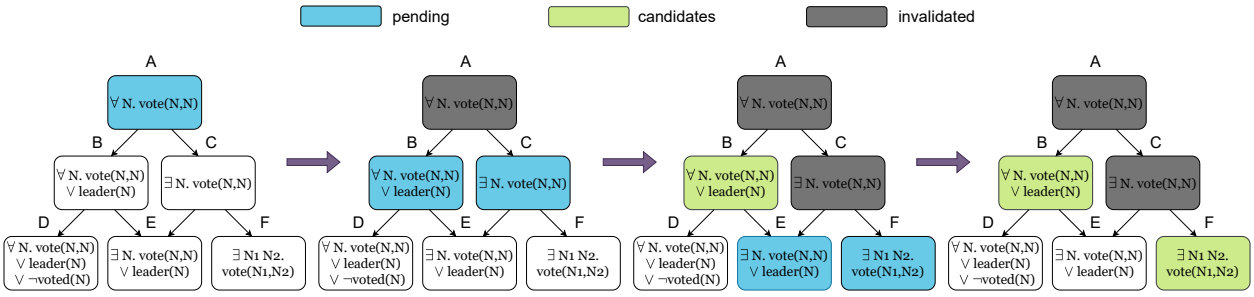


Figure 3.3: Invariant enumeration procedure based on the minimum implication graph. Suppose formula A and formula C do not hold on all the samples, while the other four formulas hold. Step 1: Only the root node A is in the *pending* queue. Step 2: The root node A is invalidated by the samples. We add its two successors B and C to the *pending* queue. Step 3: Formula B holds on the samples thus being added to *candidates*, while formula C is invalidated and its two successors E and F are added to the *pending* queue. Step 4: Formula E has an ancestor B which is already in *candidates*, so E is simply skipped instead of being checked on the samples. Formula F holds on the samples and is added to *candidates*.

the samples and are not added to the candidates, because their predecessor B, a formula stronger than both D and E, is already a candidate invariant. This design not only saves time checking formulas on samples, but also avoids burdening the SMT solver later with checking the inductiveness of redundant invariants. More importantly, this procedure guarantees that the resulting invariants, formulas B and F in this example, are the strongest candidate invariants that hold on the samples, which is formally stated in the following theorem:

Theorem 4. *For any correct invariant $I \in \mathcal{S}$ held by the protocol $\mathcal{P}$, at the end of invariant enumeration, either 1) $I \in candidates$, or 2) one of I's ancestors $I_{anc} \in candidates$.*

Proof. Consider three cases: 1) I has been checked on the samples, 2) I has been added to the *pending* queue but was not checked on samples, and 3) I has been never added to the *pending* queue. In the first case, since I is a correct invariant held by the protocol, it must hold on all the samples and will be added to *candidates* (Lines 8-9), so $I \in candidates$. In the second case, after I is popped from the *pending* queue, there must be an ancestor I_{anc} of I already in *candidates* (Line 6), otherwise I will be checked on the samples, so $I_{anc} \in candidates$. In the third case, we show that an ancestor $I_{anc} \in candidates$ exists. From Lemma 5, there must be a path from a root node I_0 to I, namely $I_0, I_1, ..., I$. On Line 3 the root node I_0 is added to the *pending* queue. Since I_0 is added to the *pending* queue and I is not, let I_k be the last formula on the path $I_0, I_1, ..., I$

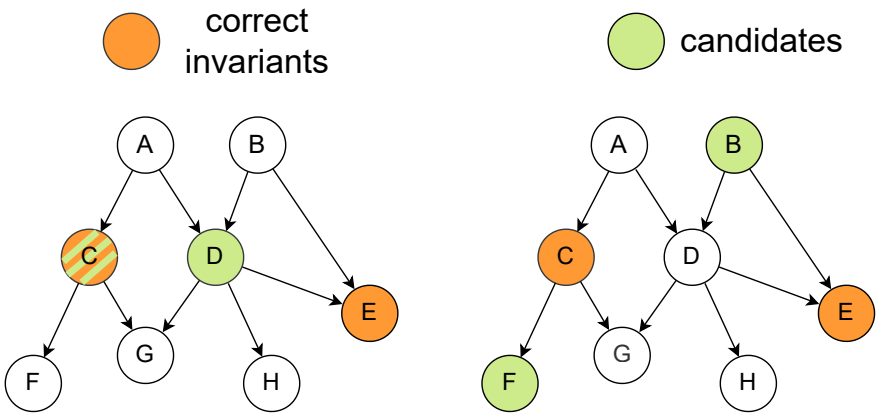


Figure 3.4: Possible (left) and impossible (right) candidate invariants after enumeration. Formulas C and E are correct invariants held by the protocol. It is possible that after enumeration, $candidates = \{C, D\}$ (left). Correct invariant C is in the candidate invariants itself. For correct invariant E, its ancestor D is in the candidate invariants. Theorem 4 guarantees that $candidates = \{B, F\}$ is not a possibility after enumeration, because for correct invariant C, neither itself nor its lone ancestor A is in the candidate invariants.

that is ever added to the *pending* queue. After I_k is dequeued, there are three possible branches to take: Lines 6-7, Lines 8-9, or Lines 10-14. If it takes Lines 6-7, then there is an ancestor I_{anc} of I_k such that $I_{anc} \in candidates$. If it takes Lines 8-9, then I_k will be added to $candidates$ so I_k can be the ancestor I_{anc} of I such that $I_{anc} \in candidates$. If it takes Lines 10-14, its successors will be added to the *pending* queue unless the branch condition at Line 13 evaluates to false. From our hypothesis, I_k is last formula on path $I_0, ..., I_k, I_{k+1}, ..., I$ that is ever added to the *pending* queue. Thus, the branch condition for I_{k+1} must evaluate to false, so either $I_{k+1} \in candidates$ or $I_{k+1} \in invalidated$. However, I_{k+1} must be added to the *pending* queue before it can be added to either $candidates$ or $invalidated$, a contradiction. □

Theorem 4 says that any correct invariant has either itself or its ancestor (a stronger formula) in the candidate invariants. Figure 3.4 gives an illustration. A direct corollary is that, the set of candidate invariants after the enumeration is at least as strong as the correct invariant in $\mathcal{S}$.

3.4 Top-down Invariant Refinement

Based on Theorem 4, the candidate invariants can only be *too strong*, so DuoAI can monotonically weaken the candidate invariants until a correct inductive invariant is reached, which we refer

Algorithm 5 Top-down Invariant Refinement Algorithm

Input: Distributed protocol $\mathcal{P}$, minimum implication graph $graph$, candidate invariants from enumeration CI
Output: Either an inductive invariant II, or NotProvable

```
1:  candidates, invalidated := CI, ∅
2:  while candidates.notEmpty() do
3:      failed_inv := Ivy_check(P, candidates)
4:      if failed_inv is None then
5:          return candidates
6:      else if failed_inv = safety_property then
7:          return NotProvable
8:      else
9:          candidates := candidates \ {failed_inv}
10:         invalidated := invalidated ∪ {failed_inv}
11:         for next_inv ∈ graph.successors(failed_inv) do
12:             if graph.reachable_ancestors(next_inv) ∩ candidates = ∅ and next_inv ∉ invalidated
                then
13:                 candidates := candidates ∪ {next_inv}
14: return NotProvable
```

to as top-down invariant refinement. Algorithm 5 shows the top-down refinement algorithm. In each iteration, DuoAI feeds the current candidate invariants to Ivy. Ivy invokes the Z3 SMT solver to check the inductiveness of each candidate invariant and the safety property. Ivy will return which invariants fail the check; if there are none, the correct inductive invariant has been found (Lines 4-5). If the safety property fails, there is no point to weaken it, and the system returns Not-Provable (Lines 6-7). If one of the candidate invariants fails, DuoAI moves it from *candidates* to *invalidated* (Lines 9-10), then adds its successors (i.e., formulas that can be implied by the failed invariant) to *candidates* so long as the successor does not have a reachable ancestor in *candidates* and has not already been invalidated (Lines 12-13). An ancestor of a node is reachable if there is a path from the ancestor to the node along which no node is invalidated.

Figure 3.5 shows an example of top-down refinement. Suppose the current candidate invariants include formulas B and F, and by invoking the Z3 SMT solver, Ivy indicates that B is not inductive. Formulas D and E are not in *candidates*, because they can be implied by formula B which is already in *candidates*. After B is invalidated, both D and E will be added to *candidates* to let

Ivy decide their inductiveness in future iterations. Alternatively, if formula F is invalidated by Ivy, no formula will be added to $candidates$ because F has no successor in the minimum implication graph of search space $\mathcal{S}$.

By weakening failed invariants based on the minimum implication graph rather than discarding them, DuoAI can guarantee that it never overweakens invariants to bypass the correct invariants in between. In other words, top-down refinement has a theoretical guarantee to eventually find an inductive invariant if one exists in the search space, as stated in the following theorem:

Theorem 5. *For any protocol $\mathcal{P}$ and finite search space $\mathcal{S}$, if there exists an inductive invariant $II^* \subset \mathcal{S}$ that can prove the safety property, then Algorithm 4 followed by Algorithm 5 will output such an inductive invariant II in finite time.*

Proof. The key is to prove that the while loop (Lines 2-13) maintains the following loop invariant: For any invariant $I \in II^*$, either 1) $I \in candidates$, or 2) there exists a reachable ancestor I_{anc} of I such that $I_{anc} \in candidates$. The loop invariant says that after any rounds of invariant weakening, the candidate invariants must be still at least as strong as the correct invariants. If Algorithm 5 terminates, it is impossible to have the safety property fail (Line 7). The only possibility is that a correct inductive invariant is returned (Line 5).

Theorem 4 guarantees that the loop invariant holds before entering the loop. We only need to prove that if this loop invariant holds at the beginning of round k of invariant weakening, it must still hold at the beginning of round $k + 1$. This proof is done by construction for each $I \in II^*$. From the induction hypothesis, at the beginning of round k, either 1) $I \in candidates$, or 2) a reachable ancestor $I_{anc} \in candidates$. In the first case, I cannot have been invalidated during round k because $I \in II^*$, so $I \in candidates$ still holds at the beginning of iteration $k + 1$. In the second case, the invalidated invariant must either be on or not on the path from I_{anc} to I. If it is not on the path, I_{anc} remains a reachable ancestor of I and $I_{anc} \in candidates$ still holds at the beginning of iteration $k + 1$. If it is on the path, let I_d be the successor of the invalidated invariant on the path. From Lines 11-13, either I_d is added to $candidates$, in which case I_d can be the new I_{anc} for iteration $k + 1$, or I_d has a reachable ancestor $I_e \in candidates$, in which case we choose

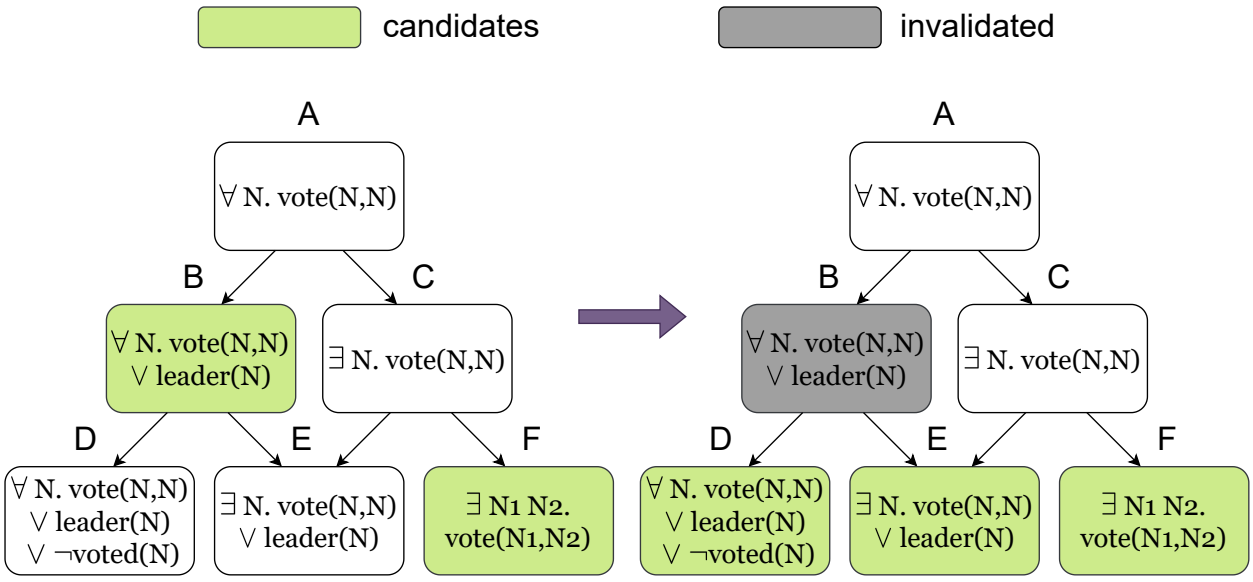


Figure 3.5: One round of top-down refinement. Suppose candidate invariant B fails the Ivy check. DuoAI removes B from *candidates* and adds its successors D and E to *candidates*.

I_e as the new I_{anc} for iteration $k+1$. In all cases, we can find either I or a reachable ancestor I_{anc} in the candidate set, therefore the loop invariant holds.

Now we only need to prove that Algorithm 5 terminates, which follows from three observations: 1) In each loop iteration a formula is removed from *candidates* (Line 9); 2) each formula can only be added to *candidates* once (Lines 10 & 12); and 3) the formula search space $\mathcal{S}$ is finite. □

3.5 Bottom-up Invariant Refinement

Although top-down refinement provides a strong theoretical guarantee of finding an inductive invariant, it may take too long or run out of memory given limited computing resources if there are too many unnecessary invariants to consider. For the simplified consensus protocol in Figure 3.1, besides the four invariants (3.1)(3.2)(3.3)(3.4), many other invariants hold for the protocol but are

unnecessary to prove the inductiveness of the safety property, for example,

$$\forall V : value,\ Q : quorum.\ \exists N : node.$$
$$member(N, Q) \wedge (leader(N) \vee \neg decided(N, V)). \tag{3.10}$$

Invariants such as Eq. (3.10) do not affect the soundness of DuoAI, but they will significantly slow down the validation of candidate invariants by the SMT solver. If there are m candidate invariants, validating each invariant takes $O(m)$ time in the worst case, since the inductiveness of one invariant can depend on any other invariant, so checking all candidate invariants can take $O(m^2)$ time. Adding unnecessary invariants can increase validation time quadratically.

The key issue though is not just how many unnecessary invariants there are, but whether they have quantifier alternation (i.e., alternating $\forall$ and $\exists$), which we observe causes SMT solvers to struggle. For the Paxos protocol, a correct inductive invariant set of size 14 can be validated in less than a second. If we add 10 correct but unnecessary invariants with quantifier alternation, the validation will take 5 minutes. If we add 20 such invariants, the validation will take over 3 hours.In contrast, the chord ring maintenance protocol [12] with 149 $\forall$-only invariants only takes 8 seconds to validate.

However, a correct distributed protocol typically has a clear and human-understandable intuition, which leads to concise invariants [26]. This motivates our bottom-up invariant refinement algorithm shown in Algorithm 6. In essence, the algorithm tries to identify a small set of correct *and* helpful invariants that can eventually prove the safety property. §3.7 shows that the combination of bottom-up with top-down refinement provides fast performance for finding inductive invariants across a wide-range of protocols.

Algorithm 6 consists of two procedures. In Procedure 1, DuoAI first extracts all the $\forall$-only invariants from the candidate invariants (Line 1), which are guaranteed to be the strongest $\forall$-only invariants that hold on the samples. Then, DuoAI runs the top-down refinement algorithm (Line 2) using only the universal invariants and the universal portion of the minimum implication

Algorithm 6 Bottom-up Invariant Refinement Algorithm

Input: Distributed protocol $\mathcal{P}$, minimum implication graph $graph$, candidate invariants from enumeration CI
Output: Either an inductive invariant II, or NotProvable

Procedure 1
1: $CI_\forall := \{I \mid I \in CI \wedge I \text{ is } \exists\text{-free}\}$
2: $core :=$ Algorithm2($\mathcal{P}$, $graph_\forall$, $CI_\forall$)
3: $noncore := CI \setminus core$
Procedure 2
4: $CE := \emptyset$
5: **for** sub **in** $powerset(noncore)$ **do**
6: **if** $\exists s \in CE$. invs_hold_on_state(sub, s) **then**
7: **continue**
8: result := Algorithm2($\mathcal{P}$, $graph$, $core \cup sub$)
9: **if** result = NotProvable **then**
10: $s \xrightarrow{a} s' :=$ get_counterexample()
11: $CE := CE \cup \{s\}$
12: **else**
13: **return** result
14: **return** NotProvable

graph by removing all nodes representing existentially quantified formulas. The safety property is neglected in this top-down refinement. In this way, the ∀-only invariants are monotonically weakened until they become inductive, regardless of whether the safety property can be proved (it probably cannot). Recall that we call the now inductive ∀-only invariants the universal inductive core. DuoAI then puts every enumerated candidate invariant that is not in the universal inductive core into $noncore$ (Line 3). $noncore$ mainly consists of formulas with existential quantifiers, but also includes ∀-only formulas that are not in the core, whose inductiveness may depend on ∃-included invariants. For example, for the simplified consensus protocol, the universal inductive core includes five candidate invariants, which are exactly the equivalent forms of Eq. (3.1), (3.2), and (3.4). There are 14 non-core candidate invariants, 13 of which have quantifier alternation, including Eq. (3.3) and (3.10).

Based on our observation that SMT solvers struggle with quantifier alternation, we expect $noncore$ formulas will have a much higher cost of checking. Procedure 2 aims to identify a *small*

subset of $noncore$ to strengthen the candidate invariants, such that the conjunction of the universal inductive core and the subset (denoted as $core \cup sub$), or their weaker forms, can prove the safety property. Procedure 2 enumerates each subset sub of $noncore$ (Line 5), and runs the monotonic weakening algorithm (Algorithm 5) on $core \cup sub$ (Line 8). If Algorithm 5 returns NotProvable (Line 9), DuoAI moves on to consider the next subset. Otherwise, Algorithm 5 outputs a correct inductive invariant (Line 13). The enumeration of subsets is conducted in increasing order of size, starting from the $\emptyset$, followed by all single formulas from $noncore$, then pairs, triples, and so on.

Whenever Algorithm 5 finds the safety property failed and reports NotProvable, Ivy returns a counterexample of inductiveness $s \xrightarrow{a} s'$ (Line 10), which means starting from a protocol state s satisfying the safety property and the candidate invariants, and taking an action a, the system reaches a new state s' where the safety property is violated.[1] If we view the samples from protocol simulation as positive samples on which the invariants must hold, then we can view these counterexample states s as negative samples which the invariants must exclude. DuoAI needs to identify and include another invariant I that does *not* hold on s, so that the counterexample $s \xrightarrow{a} s'$ can be excluded. When enumerating a subset of $noncore$, Procedure 2 first checks if the subset can exclude all counterexamples seen so far (Line 6). If there exists one counterexample state s on which all invariants in the subset hold, or in other words, the counterexample cannot be excluded, the monotonic weakening algorithm is bound to fail, because if a stronger invariant cannot exclude the counterexample, then its weaker forms cannot either. So Procedure 2 simply moves on to enumerate the next subset (Line 7).

For the simplified consensus protocol, when $sub = \emptyset$, the safety property fails and Ivy gives the counterexample $s = \{vote(n_1, n_1) = vote(n_1, n_2) = vote(n_2, n_1) = vote(n_2, n_2) = false,$ $voted(n_1) = voted(n_2) = false,\ leader(n_1) = leader(n_2) = true,\ member(n_1, q) =$ $member(n_2, q) = true,\ decided(n_2, v_1) = true,\ decided(n_1, v_1) = decided(n_1, v_2) =$ $decided(n_2, v_2) = false\}$. Eq. (3.3) does not hold on s, so it can exclude this counterexample. In contrast, Eq. (3.10) holds on s, so the counterexample will persist even if Eq. (3.10) is added to

[1] In general, other than showing an invariant is not inductive, a counterexample may also show an invariant does not hold at the protocol initial state. But this cannot happen to the safety property, unless the protocol is wrong.

the candidate set. Therefore, DuoAI will skip Eq. (3.10) and try Eq. (3.3), and run Algorithm 5 on its conjunction with the universal core, which gives a correct inductive invariant set consisting of Eq. (3.1)(3.2)(3.3)(3.4).

Although counterexamples can be used for top-down refinement, DuoAI currently does not because Ivy cannot return counterexamples in batch. When Ivy is configured to return a counterexample, it terminates once it identifies the first broken invariant. This is inefficient for top-down refinement, but for bottom-up refinement, counterexamples are only needed for the safety property, so DuoAI puts the safety property on top of other invariants and Ivy will give the desired counterexample.

Like top-down refinement, bottom-up refinement has a theoretical guarantee to eventually find an inductive invariant if one exists in the search space, as stated in the following theorem:

Theorem 6. *For any protocol $\mathcal{P}$ and finite search space $\mathcal{S}$, if there exists an inductive invariant $II^* \subset \mathcal{S}$ that can prove the safety property, then Algorithm 4 followed by Algorithm 6 will output such an inductive invariant II in finite time.*

Proof. We first prove that Algorithm 6 terminates in finite time. This directly follows from three facts: 1) $powerset(noncore)$ is a finite set so the for loop (Line 5) has a finite number of iterations; 2) In each loop iteration, there is at most one invocation of Algorithm 5 (Line 8); and 3) From Theorem 5, Algorithm 5 terminates in finite time.

To prove the soundness of Algorithm 6, we first observe that if Algorithm 6 outputs an invariant, it must be a correct inductive invariant, because the output must come from Algorithm 5, in which the output can only occur when the safety property is proved.

Now we prove that there will be an output invariant eventually. Observe that $noncore \in powerset(noncore)$ (Line 5). When $sub = noncore$, we have $CI \subset core \cup sub$, then Line 8 degenerates to Algorithm 5 in §3.4. From Theorem 5, we know a correct inductive invariant will be outputted. □

For both the top-down and bottom-up refinement, if NotProvable is returned, we know the

protocol cannot be verified using invariants in the search space $\mathcal{S}$. DuoAI will try a larger search space by increasing either $max\_literal$, max_or, max_and, or max_exists, or the per-domain number of quantified variables. By default, DuoAI alternates among the five in a round-robin manner. DuoAI sets the initial $max\_literal = 4$, $max\_or = max\_and = 3$, and $max\_exists = 1$ unless the safety property already involves $k \geq 2$ existentially quantified variables, in which case DuoAI sets $max\_exists = k$. DuoAI sets the initial number of quantified variables for domain T as the maximum number of variables of type T in any relation. For example, the relation `vote(N1:node,N2:node)` guarantees type `node` has at least two variables.

Because SMT solvers are much less efficient at checking invariants with existential quantifiers, and many distributed protocols are provable by ∀-only invariants [12], DuoAI runs a ∀-only instance (i.e., $max\_exists = 0$) in parallel. The ∀-only instance only runs top-down refinement, as bottom-up refinement degenerates to the same top-down refinement (Line 2).

3.6 Optimizations Based on Mutual Implication

In using the minimum implication graph, DuoAI introduces several optimizations based on mutual implication relations among formulas. These relations further prune the search space and avoid redundant candidate invariants. DuoAI considers two kinds of mutual implication relations, 1) $P_1 \wedge P_2 \wedge ... \wedge P_k \Rightarrow Q$, and 2) $P_1 \wedge P_2 \wedge ... \wedge P_k \Leftrightarrow Q$. Although the latter is a special case of the former, DuoAI treats them differently. We refer to Q as a *conjunction implied formula* in the former and an *equivalently decomposable formula* in the latter. Since checking inductiveness has a quadratic complexity with the number of invariants, these optimizations have a significant improvement on efficiency.

Conjunction implied formulas. DuoAI identifies conjunction implied formulas to avoid redundant candidate invariants. Given a mutual implication relation $P_1 \wedge P_2 \wedge ... \wedge P_k \Rightarrow Q$, if all $P_1, P_2, ...P_k$ are already in the candidate invariants, DuoAI will mark Q as a conjunction implied formula and not add Q to the candidate invariants. Later during refinement, if one of $P_1, P_2, ...P_k$

is invalidated by Ivy, then the conjunction implied invariant Q is no longer redundant and will be added to the candidate invariants.

For example, suppose we have a disk replication protocol with the following three invariants:

$$\forall E : epoch,\ R : replica.\ crashed(E, R) \rightarrow \neg readable(E, R) \tag{3.11}$$

$$\forall E : epoch.\ \exists R : replica.\ readable(E, R) \tag{3.12}$$

$$\forall E : epoch.\ \exists R : replica.\ \neg crashed(E, R). \tag{3.13}$$

One can check that among Eq. (3.11)(3.12)(3.13), no formula can imply another. But the conjunction of Eq. (3.11) and (3.12) can imply Eq. (3.13). This is because Eq. (3.12) says that for every epoch E, there must be a readable replica R. Then from Eq. (3.11), the readable replica R cannot be crashed. Therefore, for every epoch E, there must be a replica R that does not crash, which is expressed by Eq. (3.13). If Eq. (3.11) and (3.12) are already candidate invariants, DuoAI will mark Eq. (3.13) as a conjunction implied formula and not add it to the candidate invariants.

There are many classes of mutual implication relations in first-order logic. DuoAI identifies three classes of conjunction implied formulas to prune candidate invariants; in each class, the first two formulas mutually imply the third:

1. Replace a literal with a weaker literal, as discussed in the example Eq. (3.11)(3.12)(3.13):

$$P_1 = \forall X.\ r(X) \rightarrow s(X)$$

$$P_2 = prefix.\ (r(X) \wedge ...) \vee ...$$

$$Q = prefix.\ (s(X) \wedge ...) \vee ...$$

2. Conjunct a literal with a weaker literal:

$$P_1 = \forall X.\ r(X) \rightarrow s(X)$$

$$P_2 = prefix.\ (r(X) \wedge ...) \vee ...$$

$$Q = prefix.\ (r(X) \wedge s(X) \wedge ...) \vee ...$$

3. "Merge" a $\forall$ formula and an $\exists$ formula:

$$P_1 = \exists X.\ r(X)$$

$$P_2 = \forall X.\ s(X) \vee ...$$

$$Q = \exists X.\ (r(X) \wedge s(X)) \vee ...$$

In all three classes, r and s can be generalized to conjunctions (e.g., $r_1(X) \wedge \neg r_2(X)$, $\neg s_1(X) \wedge s_2(X) \wedge s_3(X)$). A key advantage of this optimization is that given a finite search space, DuoAI can identify conjunction implied formulas based on invariants within that search space, even though the conjunction of invariants is not in that search space.

Equivalently decomposable formulas. DuoAI also identifies equivalently decomposable formulas to avoid redundant candidate invariants. Given a mutual implication relation $P_1 \wedge P_2 \wedge ... \wedge P_k \Leftrightarrow Q$, DuoAI will mark Q as an equivalently decomposable formula and never add Q to the candidate invariants. Later during refinement, if one of $P_1, P_2, ...P_k$ is invalidated by Ivy, Q will also be invalidated and therefore there is never any reason to consider Q further as a candidate invariant.

For example, suppose the disk replication protocol has invariant:

$$\forall E : epoch.\ \exists R_1, R_2 : replica.$$
$$readable(E, R_1) \wedge writable(E, R_2). \tag{3.14}$$

There is no need to ever include such an invariant in the candidate set, because it is equivalently decomposable to invariants (3.15) and (3.16).

$$\forall E : epoch.\ \exists R : replica.\ readable(E, R) \tag{3.15}$$

$$\forall E : epoch.\ \exists R : replica.\ writable(E, R). \tag{3.16}$$

However, suppose we slightly modify invariant (3.14) to one of the following two formulas:

$$\forall E : epoch.\ \exists R_1 : replica.$$
$$readable(E, R_1) \wedge writable(E, R_1) \tag{3.17}$$
$$\forall E : epoch.\ \exists R_1, R_2 : replica.$$
$$R_1 \neq R_2 \wedge readable(E, R_1) \wedge writable(E, R_2). \tag{3.18}$$

These invariants are not equivalently decomposable to invariants (3.15) and (3.16). Take Eq. (3.17) as an example. One can verify that (3.17) $\Rightarrow$ (3.15) $\wedge$ (3.16), but (3.15) $\wedge$ (3.16) $\not\Rightarrow$ (3.17), because Eq. (3.17) requires the same replica to be both readable and writable, while for Eq. (3.15) and (3.16), it is possible to have one readable replica and a different writable replica.

DuoAI identifies if a formula is equivalently decomposable based on the structure of the formula itself by considering three classes of equivalently decomposable formulas. The first two classes are embodied by the following two lemmas. Let $\vec{X}$ denote a list of variables $X_1, X_2, ..., X_n$, and let $Q\vec{X}$ denote a quantifier prefix on variables $\vec{X}$ with an arbitrary combination of $\forall$ and $\exists$.

Lemma 6. *Any formula of shape*

$$\forall \vec{X}\ \exists \vec{Y}\ Q\vec{W}.\ (f(\vec{X}, \vec{W}) \wedge g(\vec{X}, \vec{Y})) \vee h(\vec{X}, \vec{W}). \tag{3.19}$$

is equivalently decomposable into

$$\forall \vec{X}\, Q\vec{W}.\ f(\vec{X}, \vec{W}) \vee h(\vec{X}, \vec{W}) \tag{3.20}$$

$$\forall \vec{X}\, \exists \vec{Y}\, Q\vec{W}.\ g(\vec{X}, \vec{Y}) \vee h(\vec{X}, \vec{W}). \tag{3.21}$$

Proof. It is trivial that Eq. (3.19) implies both Eq. (3.20) and (3.21). We now show the interesting direction — the conjunction of Eq. (3.20) and (3.21) implies Eq. (3.19). Suppose both Eq. (3.20) and (3.21) hold. For any $\vec{X}$, consider two cases: 1) $Q\vec{W}.\ h(\vec{X}, \vec{W})$. In this case Eq. (3.19) directly holds. 2) $\neg Q\vec{W}.\ h(\vec{X}, \vec{W})$. Then according to Eq. (3.21), $\exists \vec{Y_1}.\ g(\vec{X}, \vec{Y_1})$. Then if we instantiate $\exists \vec{Y}$ with $\vec{Y_1}$ in Eq. (3.19), $Q\vec{W}.\ (f(\vec{X}, \vec{W}) \wedge g(\vec{X}, \vec{Y_1})) \vee h(\vec{X}, \vec{W})$ is simplified to $Q\vec{W}.\ f(\vec{X}, \vec{W}) \vee h(\vec{X}, \vec{W})$ which is assumed correct from Eq. (3.20). Putting the two cases together, when both Eq. (3.20) and (3.21) are true, Eq. (3.19) must be true. □

Lemma 7. *Any formula of shape*

$$\forall \vec{X}\, \exists \vec{Y}\, \vec{Z}\, Q\vec{W}.\ (f(\vec{X}, \vec{Y}) \wedge g(\vec{X}, \vec{Z})) \vee h(\vec{X}, \vec{Y}, \vec{Z}, \vec{W}). \tag{3.22}$$

is equivalently decomposable into

$$\forall \vec{X}\, \exists \vec{Y}\, \vec{Z}\, Q\vec{W}.\ f(\vec{X}, \vec{Y}) \vee h(\vec{X}, \vec{Y}, \vec{Z}, \vec{W}) \tag{3.23}$$

$$\forall \vec{X}\, \exists \vec{Y}\, \vec{Z}\, Q\vec{W}.\ g(\vec{X}, \vec{Z}) \vee h(\vec{X}, \vec{Y}, \vec{Z}, \vec{W}). \tag{3.24}$$

Proof. It is trivial that Eq. (3.22) implies both Eq. (3.23) and (3.24). We now show the interesting direction — the conjunction of Eq. (3.23) and (3.24) implies Eq. (3.22). Suppose both Eq. (3.23) and (3.24) hold. For any $\vec{X}$, consider two cases: 1) $\exists \vec{Y}\, \vec{Z}\, Q\vec{W}.\ h(\vec{X}, \vec{Y}, \vec{Z}, \vec{W})$. In this case Eq. (3.22) directly holds. 2) $\forall \vec{Y}\, \vec{Z}.\ \neg Q\vec{W}.\ h(\vec{X}, \vec{Y}, \vec{Z}, \vec{W})$. Then according to Eq. (3.23), $\exists \vec{Y_1}\, \vec{Z_1}.\ f(\vec{X}, \vec{Y_1})$. Similarly, from Eq. (3.24), $\exists \vec{Y_2}\, \vec{Z_2}.\ g(\vec{X}, \vec{Z_2})$. If we select $\vec{Y_1}$ and $\vec{Z_2}$, then we have $f(\vec{X}, \vec{Y_1}) \wedge g(\vec{X}, \vec{Z_2})$, so Eq. (3.22) still holds. Putting the two cases together, when both Eq. (3.23) and (3.24) are true, Eq. (3.22) must be true. □

We note a corollary of both Lemma 6 and Lemma 7. For an ∃-free formula, it is equivalently decomposable if it has any conjunction. For example, $\forall X.\ p(X) \wedge q(X)$ is equivalent with the pair $\forall X.\ p(X)$ and $\forall X.\ q(X)$. This indicates that we do not need to consider any conjunction when enumerating ∀-only formulas, a significant reduction in search space.

The third class of equivalently decomposable formulas that DuoAI identifies is embodied in the following:

$$\forall X_1\ X_2.\ matrix(X_1, X_2) \tag{3.25}$$

$$\forall X_1.\ matrix(X_1, X_1) \tag{3.26}$$

$$\forall X_1\ X_2.\ X_1 \neq X_2 \rightarrow matrix(X_1, X_2) \tag{3.27}$$

One can check that Eq. (3.25) is equivalently decomposable to Eq. (3.26) and (3.27), and will therefore not be added as a candidate invariant. In general, DuoAI only considers formulas whose leading ∀-quantified variables are unique. Similar optimizations have been used in DistAI [16].

3.7 Evaluation, Results, and Discussions

Experimental setup. To demonstrate the performance of DuoAI, we implemented and evaluated DuoAI on 27 distributed protocols from multiple sources [11, 33, 12, 29, 13], including those that can only be proved by inductive invariants with ∃-quantifiers. The DuoAI implementation consists of 6.1K lines of C++ code for invariant enumeration and refinement, compiled by gcc 7.5.0, and 2.3K lines of Python code running with Python 3.8.10 for protocol simulation. For comparison, we also ran 6 other invariant inference tools: SWISS [26], IC3PO [34, 35], FOL-IC3 [13], DistAI [16], UPDR [33], and I4 [12]. All experiments were performed on a Dell Precision 5829 workstation with a 4.3GHz 28-core Intel Xeon W-2175, 62GB RAM, and a 512GB Intel SSD Pro 600p, running Ubuntu 18.04.

We configured the alternative invariant inference tools following their best practices. SWISS requires the user to bound the search space by specifying 4 parameters, including the number of

existentially quantified variables and the number of literals in a formula. For every protocol, we use the same parameter settings as in SWISS's own evaluation [26]. IC3PO and I4 require the user to specify a finite instance size for their model checkers to work on. For IC3PO, we only specified the minimum size and the tool itself could determine how to increase the instance size. For I4, we started from the minimum size where the protocol can function and iteratively increased the instance size upon failure (e.g., $node = 2, node = 3, ...$). FOL-IC3 provides a $\forall$-only mode and a default mode. We ran both and report the runtime of whichever succeeded first.

Results summary. Table 3.1 shows the running time in seconds for each tool on each protocol. For each protocol, we also report the number of relations and lines of code in its Ivy specification; for example, Figure 3.1 is a simplified version of consensus epr. The top portion of the table shows protocols provable with a $\forall$-only inductive invariant, while the bottom portion shows protocols that can only be proved with a $\exists$-included inductive invariant. We allowed each tool to spend up to an entire week trying to solve each protocol. For protocols that a tool fails to solve, we report "fail" if the tool terminated without an inductive invariant, "error" if the tool itself returned an error, "Z3 error" if the underlying SMT solver used returned an error, "memout" if the tool ran out of memory and terminated, and "timeout" if the tool did not complete within a week.

DuoAI dominates all other tools in the number of protocols it solves, solving all but 1 of the 27 protocols. SWISS cannot solve 8 protocols, FOL-IC3 and IC3PO cannot solve 9 protocols, DistAI and UPDR cannot solve 13 protocols, while I4 cannot solve 15 protocols. DuoAI is the only tool that solves all $\forall$-only protocols, is the only tool that solves Paxos as well as all other non-Paxos protocols with $\exists$ quantifiers, and is the only tool that solves 3 of the more complex Paxos variants, including multi-Paxos, stoppable Paxos, and fast Paxos. There were no protocols solvable by another tool that were not solved by DuoAI.

DuoAI also dominates all other tools in how fast it solves the protocols, solving 15 of the protocols faster than any other tool. DuoAI is faster than SWISS on all but 3 of the protocols solved by SWISS, is faster than IC3PO on all but 3 of the protocols solved by IC3PO, is faster

Distributed protocol	Relations	LoC	DuoAI	SWISS	IC3PO	FOL-IC3
chord ring maintenance	8	123	194.7	timeout	17.1	timeout[c]
consensus forall	7	55	11.1	40.3	457	1500
consensus wo decide	6	46	3.7	26.1	160	24.8
database chain replication	13	96	9.1	108951	4.5	559
decentralized lock	2	21	9.5	5.8	24.3	69.0
distributed lock	4	43	5.8	timeout	12856	1660
ring leader election	3	45	3.4	14.3	memout	10.8
learning switch ternary	4	45	13.1	308	23.8	timeout
learning switch quad	2	21	48.8	1322	63.6	timeout
lock server async	5	45	1.7	2625	5.6	4.8
lock server sync	2	20.7	1.3	1.0	3.2	1.0
sharded key-value store	3	31	1.9	7662	5.4	9.7
ticket lock	5	49	20.7	fail	56.2	58.1
toy consensus forall	4	27	2.0	5.9	3.0	5.4
two-phase commit	7	70	1.4	9.1	4.7	4.8
client server ae	4	28	1.5	5.2	2.3	355
client server db ae	7	48	2.8	33.7	memout[i]	4822
consensus epr	7	52	4.5	28.8	1118	471
hybrid reliable broadcast	12	120	1277	fail	memout[i]	931
sharded kv no lost keys	3	32	2.1	1.8	4.8	3.7
toy consensus epr	4	25	2.5	4.3	2.4	32.9
Paxos	9	75	58.6	16665	fail[i]	timeout
flexible Paxos	10	77	75.7	28337	memout[i]	timeout
multi-Paxos	10	91	1565	timeout	fail	timeout
stoppable Paxos	11	118	4128	error	fail	timeout
fast Paxos	12	102	26249	timeout	memout	memout
vertical Paxos	12	120	memout	timeout	memout	memout

c The SWISS authors reported that FOL-IC3 solved chord ring maintenance [26], but we found that the `chord.pyv` file they used has 3 bugs.

i The IC3PO authors [34, 35] reported that IC3PO succeeded on client server db ae (17 s), hybrid reliable broadcast (587 s), Paxos (568 s), and flexible Paxos (561 s). However, they retrofitted the protocols and manually provided clauses with quantifier alternation that could appear in the invariants, which is difficult to do without first knowing the ground-truth invariants. The 4 protocols have much simpler inductive invariants when expressed on top of these clauses, with all except the simplest, client server db ae, becoming $\exists$-free. Ivy fails when checking the invariants generated by IC3PO for Paxos and flexible Paxos. The IC3PO authors [35] imply that the invariants had to be manually checked against the human-expert invariants.

Table 3.1: Comparison of different tools for finding inductive invariants for 27 distributed protocols (running time in seconds).

Distributed protocol	DistAI	UPDR	I4
chord ring maintenance	58.0	Z3 error	673
consensus forall	15.6	59.0	122
consensus wo decide	8.5	24.8	27.5
database chain replication	90.3	57.6	66.6
decentralized lock	10.2	51.0	20.7
distributed lock	15.3	63568	195
ring leader election	2.9	103	5.3
learning switch ternary	24.7	1334	12.9
learning switch quad	372	273	memout
lock server async	1.1	4.4	8.7
lock server sync	0.9	3.3	0.6
sharded key-value store	1.2	3.5	error
ticket lock	timeout	143	fail
toy consensus forall	3.1	3.4	9.4
two-phase commit	2.0	9.4	10.2
client server ae	timeout	fail	fail
client server db ae	timeout	fail	fail
consensus epr	timeout	fail	memout
hybrid reliable broadcast	error	fail	error
sharded kv no lost keys	timeout	fail	error
toy consensus epr	timeout	fail	fail
Paxos	timeout	timeout	memout
flexible Paxos	timeout	fail	memout
multi-Paxos	timeout	timeout	memout
stoppable Paxos	error	timeout	error
fast Paxos	timeout	fail	error
vertical Paxos	error	fail	error

Table 3.1: Comparison of different tools for finding inductive invariants for 27 distributed protocols (running time in seconds) (continued).

than FOL-IC3 on all but 2 of the protocols solved by FOL-IC3, and is faster than UPDR on all of the protocols solved by UPDR. DuoAI is up to 3 orders of magnitude faster than each of these protocols. DuoAI is faster than DistAI on all but 5 of the protocols solved by DistAI, and is faster than I4 on all but 2 of the protocols solved by I4. DuoAI is up to an order of magnitude faster than either DistAI or I4. The speed differences versus DistAI and I4 appear less in part because neither could solve any of the protocols with existential quantifiers. In most cases in which DuoAI is slower than other protocols, it is by at most a few seconds.

Detailed comparison and discussion. For the protocols provable with ∀-only invariants, DuoAI is the only tool that solves all 15 protocols. On ∀-only protocols, DuoAI's ∀-only instance is similar to DistAI, without subsampling and with mutual implication optimization and parallelism in simulation. DuoAI beats DistAI on 10 protocols. Unlike DuoAI, DistAI times out on ticket lock, which we discovered is due to a bug in the implementation of its protocol simulation. Chord ring maintenance is the only protocol on which DuoAI is much slower than DistAI. DistAI only allows invariants as disjunction of literals, and implements an invariant as a `vector<int>`. In contrast, DuoAI considers invariants in disjunctive normal form, so an invariant is a disjunction of conjunction of literals, implemented as a `set<vector<int>>`. This makes invariant operation slower in DuoAI. Chord ring maintenance is the only ∀-provable protocol that takes significant time on candidate invariant enumeration so the overhead is exacerbated.

For the protocols that require invariants with ∃-quantifiers to prove, DuoAI solves 11 out of 12 protocols, more than any other tool. DuoAI only fails on vertical paxos, which other tools also fail on. DistAI, I4, and UPDR fail on all of these protocols because they can only generate ∀-only invariants. IC3PO solves 4 protocols and fails on 3 protocols, but runs out of memory on 6 protocols, because it requires model checking to infer invariants on a finite instance. For more complex protocols like fast Paxos, the model checker requires too large of an instance size. In contrast, DuoAI searches in formula space and its performance does not depend (exponentially) on instance size.

For the complex Paxos-family protocols, only SWISS also verified Paxos and flexible Paxos, though it required several hours to do so. All other tools failed on all Paxos-family protocols. In contrast, DuoAI verified Paxos and flexible Paxos in less than 2 minutes. Only DuoAI verified multi-Paxos, stoppable Paxos, and fast Paxos.

As the only other tool that solves Paxos, it is instructive to compare SWISS with DuoAI. Similar to DuoAI, SWISS also enumerates candidate invariants given a bounded search space and checks their inductiveness using the SMT solver. However, it has two fundamental differences compared with DuoAI. First, SWISS relies exclusively on the SMT solver to tell the correctness of invariants, while DuoAI also uses the samples from protocol simulation to filter out invalid invariants. As we demonstrated in §3.5, SMT calls can be expensive with quantifier alternation and will negatively affect performance. Second, SWISS struggles to find mutually inductive invariants, i.e., a bundle of invariants that are inductive together but none are inductive individually. This is because SWISS can only build the invariant set by adding one and only one invariant each time and keep the set inductive. In the lock server async and the sharded key-value store protocols, where mutually inductive invariants are required to prove the safety property, SWISS has to manually increase the maximum number of literals from 6 to 9. This allows the mutually inductive invariants to be conjuncted into one big invariant, but results in a much larger search space and long runtimes of 44 and 128 minutes, respectively. DuoAI enumerates candidate invariants following the minimum implication graph and generates the strongest candidate invariants. The mutually inductive invariants (or their stronger forms) are guaranteed to be in the candidate invariants together. DuoAI solved both protocols within 2 seconds.

For the vertical paxos protocol, the human-expert inductive invariants include an invariant with 8 literals. Even after the optimization based on mutual implication, it still has 7 literals. Under the minimum per-domain number of quantified variables that can encode the human-expert invariants, there are 60 predicates that can appear in the invariants. Considering their negations, the size of the invariant search space is at the magnitude of $120^7 \approx 4e14$, well exceeding the computational power of a normal workstation. In comparison, for fast paxos, the largest invariant includes 5

literals, and there are 38 predicates. The size of the search space is at the magnitude of $3e9$. For vertical Paxos, DuoAI ends with a universal core and a set of checked non-core invariants when exhausting memory. These invariants are inductive and can be utilized as hints, although they cannot imply the safety property.

As explained in §3.5, DuoAI runs top-down refinement, bottom-up refinement, and a ∀-only instance in parallel. Not surprisingly, the ∀-only instance generates the inductive invariants first for all 15 protocols that do not require existential quantifiers. Among the 11 protocols solved by DuoAI that require existential quantifiers, the top-down refinement gives the inductive invariants first for the 5 simpler protocols — client server ae, client server db ae, toy consensus epr, consensus epr, and sharded kv no lost keys. The bottom-up refinement also succeeded but took longer. For example, for client server db ae, there are 8 candidate invariants in $noncore$. A subset of size 3 was sufficient to prove the safety property. However, the bottom-up refinement would first enumerate and fail on all single invariants and pairs before enumerating the correct triple. This takes more than 3 times longer than top-down refinement, in which after a single round of weakening, DuoAI found an inductive invariant.

For the 6 more complicated protocols with existential quantifiers, including hybrid reliable broadcast and the 5 Paxos-family protocols solved, only the bottom-up refinement generated the inductive invariants. The top-down refinement got stuck at checking the inductiveness of the invariants. For example, for multi-Paxos, after enumeration, DuoAI has a candidate invariant set of size 615, and 581 of them have quantifier alternation. Checking inductiveness of this many formulas is a hopeless task for the Z3 SMT solver. However, to prove the safety property, only 2 of the 581 candidate invariants are needed. In the bottom-up refinement, each time only a small subset of $noncore$ invariants conjuncted with the ∀-only core are fed to Ivy, so the Z3 SMT solver could handle the candidate invariants. For Paxos, flexible Paxos, multi-Paxos, and stoppable Paxos, a subset of size 2 were sufficient, while a subset of size 6 was needed for fast Paxos. This also validates our assumption that real-world distributed protocols should have concise invariants, and should not require too many invariants with quantifier alternation to verify.

Limitations By requiring quantifier alternation to conform to a fixed order of types, DuoAI ensures that the verification condition is in a decidable fragment of first-order logic. However, without decidability, an SMT solver may still succeed. For client server db ae and hybrid reliable broadcast, the invariants written by human experts are not in a decidable fragment, yet they can be efficiently verified by the SMT solver. For both protocols, DuoAI found alternative inductive invariants within the decidable fragment. If a protocol cannot be verified in decidable logic, DuoAI will fail to prove it.

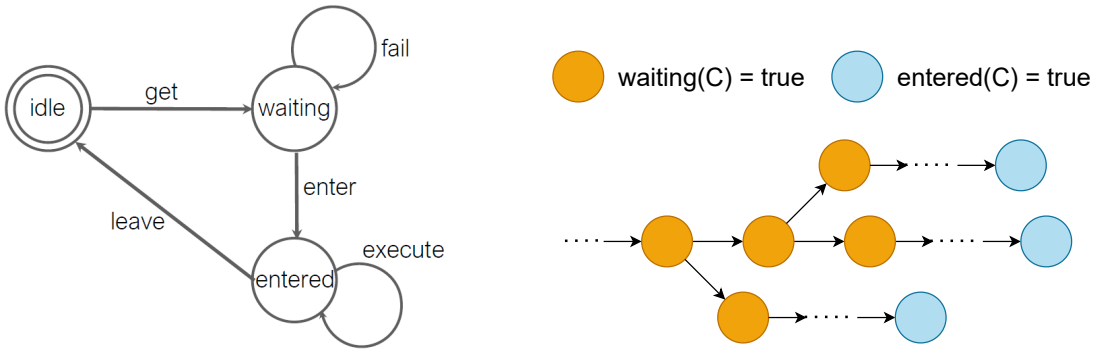
Chapter 4: LVR: Verifying Protocol Liveness via Reduction to Safety with Ranking Functions

DistAI and DuoAI mark significant steps forward on formally verifying distributed protocols. However, same as most existing tools, they only target safety properties. Liveness properties that describe what should and should not happen on a timeframe are crucial for distributed protocols to function correctly yet are out of scope for most verification methods. In this chapter, I present LVR, the first framework to verify liveness of distributed protocols in a mostly automated manner.

4.1 LVR Overview

We use the ticket lock protocol, taken from the Ivy repository [36], as a running example to show how LVR works. The ticket lock protocol uses tickets to control access to a shared resource on a server, where each client takes a ticket with a number, and only the client holding the current number being served can enter the critical section. To acquire the lock, a client requests a new ticket and waits until the ticket being served reaches its own, at which point it can enter the critical section. The ticket being served increments when a client releases the lock.

Figure 4.1a visualizes the ticket lock protocol as a state transition system specified by a state space and a set of transitions. Executing the protocol involves applying a series of transitions on an initial state to update the system state. The state of each client is either `idle` (not having a ticket), `waiting` (having a ticket and waiting for the lock), or `entered` (in the critical section). The protocol defines five transitions (`get`, `enter`, `execute`, `leave`, and `fail`) to update a client's state. The protocol also maintains two global variables `now` (current ticket being served) and `next` (next ticket to allocate) shared among clients, and a local variable `myt` (a client's own ticket) for each client. For simplicity, we assume `now` and `next` can be accessed atomically by each client.



(a) **(b)**

Figure 4.1: (a) State transition diagram for a single client in ticket lock protocol; (b) Liveness property of ticket lock. A circle represents a protocol state and an arrow represents a transition. For any client C, given a state where `waiting(C)` = *true*, no matter how the system nondeterministically evolves, there will be a future state where `entered(C)` = *true*.

Liveness property The liveness property for the ticket lock protocol ensures that all clients waiting for the lock will eventually enter the critical section, which can be expressed in first-order linear temporal logic as follows

$$\forall \texttt{C: client}.\ \Box(\texttt{waiting(C)} \rightarrow \Diamond \texttt{entered(C)}), \tag{4.1}$$

where $\Box$ denotes *globally* and $\Diamond$ denotes *finally*. Figure 4.1b gives an illustration of this property through the lens of execution traces.

Fairness assumptions The liveness property does not hold unconditionally; it requires *fairness assumptions*. For the ticket lock protocol, if the server never schedules some client, or the client in the critical section does not exit, the entire system will be starved. To prove liveness, the protocol makes two fairness assumptions:

Every client is scheduled infinitely often. (4.2)

Any client in the critical section can only execute a finite number of steps before leaving. (4.3)

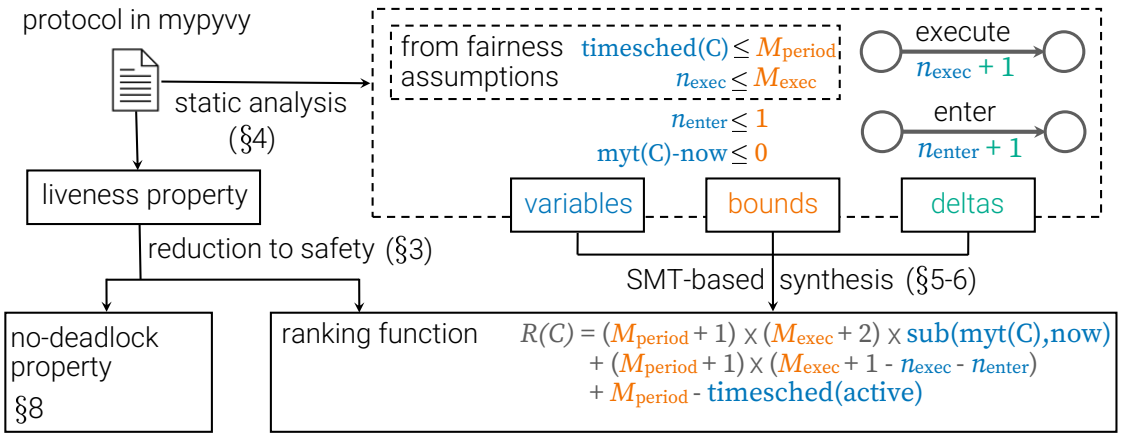


Figure 4.2: LVR workflow to verify the liveness property for ticket lock protocol.

LVR workflow Figure 4.2 shows the workflow of using LVR to verify the liveness property of the ticket lock protocol. Verifying the liveness of the ticket lock protocol means proving the liveness property Eq. (4.1) using the fairness assumptions Eq. (4.2) and (4.3). The second fairness assumption Eq. (4.3) is straightforward and typically embedded in the protocol specification [36] by counting and limiting execution steps. We focus on the first fairness assumption Eq. (4.2), which is usually formalized as a linear temporal logic formula:

$$\forall \texttt{C: client}.\ \Box(\Diamond \texttt{scheduled(C)}). \tag{4.4}$$

SMT solvers do not directly support linear temporal logic notation, but Eq. (4.4) can be translated into an equivalent first-order logic formula usable with SMT solvers [5]:

$$\forall \texttt{C: client}, \forall \texttt{T}_1\texttt{:}\ \ \texttt{time}, \exists \texttt{T}_2\texttt{:}\ \ \texttt{time}.\ \texttt{T}_1 \leq \texttt{T}_2 \wedge \texttt{scheduled(C)}|_{\texttt{T}_2}, \tag{4.5}$$

where `time` is logical time counting steps rather than wall-clock time. Eq. (4.5) states that for any client, at any time $\texttt{T}_1$, there exists a future time $\texttt{T}_2$ at which the client is scheduled. Although Eq. (4.5) is not hard for humans to understand, such a formula poses two key challenges for automating liveness proofs. First, the formula has quantifier alternation, meaning that $\forall$ and $\exists$ quantifiers are mixed together, which is known to frequently cause SMT solvers to timeout [17].

Second, although the formula implies a bound on the time, it is not explicitly stated in the formula. However, synthesizing a ranking function to prove liveness properties can be complex and difficult with only implicit bounds in fairness assumptions, as then the bounds cannot be used explicitly in the ranking function.

Encoding fairness We make two key insights to enable LVR to use SMT solvers and ranking functions to prove liveness properties. First, we can eliminate quantifier alternation from fairness assumptions if the time implied by such assumptions can be captured in the protocol state itself. In this way, fairness assumptions no longer need to be formulated as whether or not some time exists because it necessarily exists as part of the protocol state. Second, since fairness assumptions necessarily imply some bound on time, we can make explicit its existence by introducing an abstract bound in the formula. The abstract bound is not fixed to any specific value, but simply provides a concrete declaration that a bound exists. Based on these insights, we can encode Eq. (4.5) by introducing and tracking a time `timesched(C)`, the time since client `C` was last scheduled, and an abstract bound M_{period}. `timesched(C)` is reset to 0 when `C` is scheduled. Then the fairness assumption is rewritten as:

$$\forall \texttt{C: client}, \forall \texttt{T: time}.\, \texttt{timesched(C)}|_{\texttt{T}} \leq M_{\text{period}}. \tag{4.6}$$

Eq. (4.6) is essentially equivalent to and expresses the same fairness assumption as Eq. (4.5). The only minor difference is the abstract bound is global instead of for each scheduling decision, but otherwise the intuition and expression of the fairness assumption remain the same. Section 4.6 provides further details. LVR can synthesize ranking functions based on Eq. (4.6).

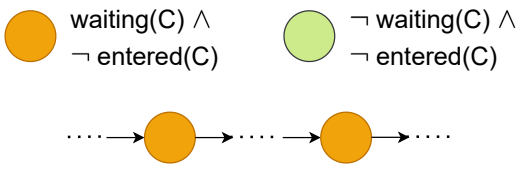
```
1  sort client, ticket
2  mutable relation idle(client), waiting(client), entered(client)
3  mutable constant now: ticket, next: ticket
4  mutable function myt(client): ticket
5  mutable constant n_exec: int
6  mutable function timesched(client): int
7  immutable constant M_exec: int, M_period: int
8  axiom M_exec > 0 & M_period > 0
9
10 init forall C. idle(C) & !waiting(C) & !entered(C)
11 init now = 0 & next = 0
12 init forall C. myt(C) = 0
13 init n_exec = 0
14 init forall C. timesched(C) = 0
15
16 procedure set_timesched(c:client)
17   timesched(c) = 0;
18   forall C. C != c -> timesched(c) = timesched(c) + 1;
19
20 transition get(c:client)
21   precondition idle(c);
22   idle(c) = false;
23   waiting(c) = true;
24   myt(c) = next;
25   next = next + 1;
26   set_timesched(c);
27
28 transition fail(c:client)
29   precondition waiting(c) & myt(c) != now;
30   set_timesched(c);
31
32 transition enter(c:client)
33   precondition waiting(c) & myt(c) = now;
34   waiting(c) = false;
35   entered(c) = true;
36   set_timesched(c);
37
38 transition execute(c:client)
39   precondition entered(c);
40   n_exec = n_exec + 1;
41   set_timesched(c);
42
43 transition leave(c:client)
44   precondition entered(c);
45   entered(c) = false;
46   idle(c) = true;
47   now = now + 1;
48   n_exec = 0;
49   set_timesched(c);
50
51 trusted invariant forall C. timesched(C) <= M_period
52 trusted invariant n_exec <= M_exec
```

Figure 4.3: Ticket lock protocol specification. In mypyvy, "constant" represents a nullary relation or function, instead of describing immutability, and "&" stands for logical AND. For illustrative purposes, we show transitions in an imperative style, use an arithmetic "+" instead of a successor relation at Lines 25 and 47, and omit an ordering relation for tickets for brevity.

Figure 4.3 shows the ticket lock protocol specification, using Eq. (4.6) at Line 51. Lines 2-7 specify the variables representing the protocol state. Lines 20-49 specify the five transitions the protocol can take. `get(c)` requires that client `c` is idle. `fail(c)` happens when client `c` does not hold the ticket being served `now` so that the client fails to enter the critical section. `enter(c)` requires client `c` is waiting and holds the ticket being served so the client can enter the critical section. `exec(c)` happens when client `c` executes an instruction in the critical section. `leave(c)` happens when client `c` leaves the critical section, incrementing the ticket number being served. Each transition executes atomically and can only happen if its preconditions are satisfied. Transitions from different clients can interleave arbitrarily.

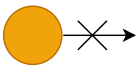
Encoding liveness Having reduced the fairness assumptions out of temporal logic, we consider how we can do the same with the liveness property, even though it may seem more entrenched in temporal logic. Instead of directly proving Eq. (4.1), it suffices to prove three simpler properties. If we call `waiting(C)` $\wedge$ `¬entered(C)` a "bad" state where a symbolic client `C` is waiting for the lock and has not entered the critical section, the three properties are: 1) the "bad" state will terminate after a finite number of transitions; 2) the system will not encounter deadlock and can always make transitions in the "bad" state; and 3) when the "bad" state ends, it must be in a "good" state where `C` enters the critical section, rather than ceasing to wait. Figure 4.4 gives an illustration of what executions are ruled out by each of the three properties.

The latter two are safety properties that do not involve infinite sequences of states and can be solved using automated invariant inference tools. The first property still involves infinite sequences of states but becomes a termination problem, which can be proved if we successfully find a *ranking function* to limit the number of steps before the termination, analogous to proving termination of loops and recursive functions [37, 38, 39, 40]. For a distributed protocol, a ranking function decreases after each protocol transition while remaining nonnegative. Both monotonically decreasing and nonnegative properties involve only two consecutive states and can be proved as safety properties. Section 4.2 formalizes this liveness-to-safety reduction.



(a) An infinite sequence in which every state satisfies `waiting(C)` ∧ ¬`entered(C)`.

(b) A transition out of `waiting(C)` without `entered(C)`.



(c) A deadlock state where no transition can be taken.

Figure 4.4: The three cases that the liveness proof needs to rule out.

Synthesizing ranking functions Although finding a valid ranking function is generally hard, we observe that ranking functions for distributed protocols are usually not that complex. They can be expressed in polynomial form with integer-valued variables, making it possible to infer and verify them automatically. Based on this observation, LVR introduces an approach to synthesizing ranking functions by first enumerating what variables may appear in the ranking function, then solving for the coefficients for each variable.

It is natural to include variables in the ranking function from the original protocol specification, such as `myt(C)`, and the ones in the encoded fairness assumptions, such as n_{exec} and `timesched(C)`, as listed at Lines 4-6 in Figure 4.3 for the ticket lock protocol. Beyond these, LVR also introduces other counting variables related to the protocol's configuration, such as n_{enter} for the number of clients that are in the entered state for the ticket lock protocol.

If LVR can infer each variable's bounds and deltas, the task can be reduced to solving for coefficients to make the formula monotonically decreasing and nonnegative with the constraints of bounds and deltas, which can be efficiently solved by SMT solvers. The bounds can be constant integers or some constants that depend on the protocol's configuration. For example, the ticket lock protocol has $n_{\text{enter}} \leq 1$, which means that only at most one client can be in the critical section.

As shown in Figure 4.2, LVR infers bounds based on fairness assumptions, such as $n_{\text{exec}} \leq M_{\text{exec}}$ at Line 52 in Figure 4.3 for the ticket lock protocol, and reasoning regarding how protocol

transitions affect variables, such as `timesched(C)` ≥ 0 for the ticket lock protocol. After finding bounds, LVR infers deltas of each integer variable by statically analyzing each transition. For example, LVR can infer that the delta of n_{enter} is 1 after each `enter` transition. Section 4.3 details the static analysis LVR employs to infer integer variables, bounds, and deltas. LVR will automatically verify the inferred bounds and deltas as safety properties.

Some limited user guidance may be needed, in terms of including additional variables or tightening the inferred bounds or deltas. For the ticket lock protocol, anyone who understands the protocol will recognize that there is an intuitive yet implicit notion of an `active` client that holds the ticket currently being served. This notion is needed for liveness reasoning, but is not explicitly declared as a variable in the protocol specification. As a result, user guidance is needed to provide an explicit variable denoting the `active` client. Section 4.8.2 quantifies the user guidance required to verify eight distributed protocols.

LVR then synthesizes a ranking function over integer variables that monotonically decreases on each transition and remains nonnegative. Nonnegativity defines constraints on variables' bounds, and monotonicity defines constraints on variables' deltas. These constraints are translated to inequalities over coefficients in the ranking function, and solved by an SMT solver. For ticket lock protocol, the ranking function inferred by LVR is:

$$\begin{aligned} R(\texttt{C}) =& (M_{\text{period}} + 1) * (M_{\text{exec}} + 2) * sub(\texttt{myt(C)}, \texttt{now}) + \\ & (M_{\text{period}} + 1) * (M_{\text{exec}} + 1 - n_{\text{exec}} - n_{\text{enter}}) + \\ & M_{\text{period}} - \texttt{timesched(active)}. \end{aligned} \tag{4.7}$$

After finding the ranking function, LVR uses the bounds and deltas to prove that $R(\texttt{C})$ is nonnegative and monotonically decreasing after each transition as safety properties. Section 4.4 shows how LVR utilizes an SMT solver to synthesize ranking functions. Section 4.5 shows how LVR can scale ranking functions by tiering to handle complex protocols.

Finally, as shown in Figure 4.2, LVR proves the absence of deadlock in the protocol. Sec-

tion 4.7 shows how LVR rules out deadlocks in the protocol using an automated invariant inference tool.

4.2 General Reduction of Liveness Properties

We prove that our approach can be generalized to any protocol's liveness property, which states that something good will eventually happen after certain conditions are triggered. LVR considers any liveness property in the following form:

> For any value `V` in the given set of variables, if at any time the `trigger(V)` condition is satisfied, the `good(V)` condition will eventually be satisfied.

This can be expressed formally in first-order linear temporal logic as:

$$\forall \texttt{V: var}.\ \Box(\texttt{trigger(V)} \rightarrow \Diamond \texttt{good(V)}). \tag{4.8}$$

Compared with the liveness property for ticket lock in Section 4.1, we can see that the `trigger` condition corresponds to `waiting(C)`, while the `good` condition corresponds to `entered(C)`.

Liveness properties of distributed protocols can naturally be expressed using Eq. (4.8). A network transmission protocol ensures that once a message is sent (the `trigger` condition), the message will eventually be received (the `good` condition). A consensus protocol ensures that a common value is eventually reached (the `good` condition), where the `trigger` condition can be a degenerate `True` in this case. We prove that verifying any liveness property in the form of Eq. (4.8) can be reduced to verifying four safety properties, as stated in the following theorem:

Theorem 7. *Let* **`old`**(e) *denote the expression e's value in the previous state. If the following four*

safety properties hold for some ranking function `rank`*,*

$$\forall \texttt{V}, \textbf{old}(\texttt{trigger(V)} \wedge \neg \texttt{good(V)}) \\ \rightarrow \texttt{rank(V)} < \textbf{old}(\texttt{rank(V)}) \tag{4.9}$$

$$\forall \texttt{V}, (\texttt{trigger(V)} \wedge \neg \texttt{good(V)}) \\ \rightarrow \texttt{rank(V)} \geq 0 \tag{4.10}$$

$$\forall \texttt{V}, (\texttt{trigger(V)} \wedge \neg \texttt{good(V)}) \\ \rightarrow \exists \texttt{tx:transition.} \quad \texttt{preconditions(tx)} \tag{4.11}$$

$$\forall \texttt{V}, \textbf{old}(\texttt{trigger(V)} \wedge \neg \texttt{good(V)}) \\ \rightarrow \texttt{trigger(V)} \vee \texttt{good(V)}, \tag{4.12}$$

then the liveness property in Eq. (4.8) *holds.*

Proof. Suppose Eq.(4.8) does not hold. Then either there exists an infinite sequence of transitions starting with a `trigger(V)` state and none of the states has `good(V)`, or there exists a sequence of transitions starting from a `trigger(V)` state and resulting in a dead state where no transitions can be taken before reaching a `good(V)` state. By Eq.(4.12), a state with `trigger(V)` but not `good(V)` will always be transitioned into a `trigger(V)` $\vee$ `good(V)` state, which can then be reduced to `trigger(V)` $\wedge \neg$`good(V)`, since none of the states in either sequence have `good(V)`. Thus, all the states in either sequence must always stay at `trigger(V)` $\wedge \neg$`good(V)`. By Eq. (4.11), such states cannot be dead. Thus the sequence must be infinite. We can then construct an integer sequence by calculating the ranking function of each state. By Eq. (4.9), this sequence is decreasing, and by Eq. (4.10), this sequence is nonnegative. Thus, we construct an infinite, decreasing sequence of nonnegative integers, which does not exist. This is a contradiction, so Eq.(4.8) must hold. □

The liveness-to-safety reduction in Theorem 7 works for liveness properties expressible using

Eq. (4.8). Compared with the three properties for the ticket lock protocol in Section 4.1, Eq. (4.9) and (4.10) correspond to the first "termination" property but are defined using a ranking function, Eq. (4.11) corresponds to the second "no deadlock" property, and Eq. (4.12) corresponds to the third "ending in good state" property. We refer to the condition `trigger(V)` $\wedge \neg$`good(V)`, which appears in all Eq. (4.9-4.12), as the *liveness prerequisite*. When the liveness prerequisite holds, the desired "good" thing is yet to happen, and the ranking function must decrease and remain nonnegative. There is no constraint on the ranking function when the liveness prerequisite does not hold. For the ticket lock example, the liveness prerequisite is `waiting(C)` $\wedge \neg$`entered(C)`.

Unlike earlier liveness-to-safety reductions based on acyclicity [41, 14], LVR does not introduce any auxiliary variables or transitions into the protocol. This protocol-preserving character brings two significant benefits. First, it makes LVR easily usable by anyone that understands a distributed protocol, without needing to additionally study and understand some reduction mechanism. Second, it allows LVR to utilize off-the-shelf inference tools to prove safety properties automatically. In contrast, acyclicity-based reduction leads to an augmented protocol that is more complex than the original protocol and beyond the power of automated invariant inference tools, requiring manual proofs for the reduced safety properties [14].

Eq. (4.12) says that the trigger condition keeps holding before entering a "good" state, which is common for many protocols including all of those we evaluated in Section 4.8. However, LVR can also be used with a protocol whose trigger condition holds for just one moment, without entering a "good" state. To verify such a protocol with LVR, its mypyvy specification can be written to include an auxiliary state `trigger'(V)`. `trigger'(V)` is initialized to false, becomes and remains true when `trigger(V)` holds at just one moment, and becomes false when `good(V)` happens. Then `trigger'(V)` can be used in place of `trigger(V)` when proving Eq. (4.9-4.12).

4.3 Inference of Bounds and Deltas

Using ranking functions reduces proving liveness to proving safety which is much easier, but the key challenge is finding such a ranking function. LVR takes a new approach to synthesizing

ranking functions by analyzing integer variables of a distributed protocol. It analyzes their upper and lower bounds and deltas. Since the ranking functions usually return integers, it is reasonable to limit the functions we consider to be polynomials of integer variables. We first show how LVR infers the set of integer variables to use when synthesizing a ranking function along with their bounds and deltas, with limited user guidance. We do not attempt completeness throughout our analysis. Instead, we aim to make LVR simple and effective on practical distributed protocols, so it can be used by protocol developers with limited expertise in formal verification.

4.3.1 Integer Variables

LVR considers three sources of integer variables. The first source is all integer variables directly defined in the protocol specification, including those used in the fairness assumptions. For example, n_{exec} is an integer variable defined in the ticket lock protocol specification.

The second source is the functions of variables that return integers, even if the variables themselves may not be integers, such as `timesched(C)` at Line 6 in Figure 4.3 for the ticket lock protocol. In general, the functions are directly defined in the protocol specification. However, if an ordering relation is defined in the specification, LVR automatically derives a subtraction function that returns an integer representing the distance between two ordered objects. For example, the ticket lock specification necessarily defines an ordering relation for tickets, so LVR introduces a subtraction function sub : `ticket` $\times$ `ticket` $\rightarrow$ *int* resulting in integer variables such as $sub($`myt(C)`$,$`now`$)$. LVR enumerates all function-induced integer variables from the protocol specification then removes redundant ones. For example, after $sub($`myt(C)`$,$`now`$)$ and $sub($`myt(C)`$,$`next`$)$ are enumerated, $sub($`now`$,$`next`$)$ is redundant because $sub($`now`$,$`next`$) = sub($`myt(C)`$,$`next`$) - sub($`myt(C)`$,$`now`$)$. Any ranking function with $sub($`now`$,$`next`$)$ can be rewritten using $sub($`myt(C)`$,$`next`$)$ and $sub($`myt(C)`$,$`now`$)$.

The third source is integer variables derived from first-order relations defined in the protocol specification. For example, n_{wait} and n_{enter} are derived from first-order relations in ticket lock. There can be many such variables derived from the relations, and it is hard to tell which ones are

relevant to liveness reasoning. LVR uses a simple heuristic to identify any variables which are counts corresponding to the first-order relations that appear in the liveness prerequisite or fairness assumptions. For example, the ticket lock protocol has a liveness prerequisite of `waiting(C)` $\wedge$ $\neg$`entered(C)`, so LVR automatically identifies the corresponding counts n_{wait} and n_{enter}.

Besides the automatically inferred variables, an LVR user, such as the protocol developer, can add additional variables based on human intuition. For the ticket lock protocol, the liveness proof intuitively involves the client holding the serving ticket, but the client holding the serving ticket is not explicitly tracked in the protocol specification in Figure 4.3. Therefore, it makes sense for the user to introduce a variable `active` denoting the client that holds the serving ticket by declaring it via the following proposition: `myt(active)` = `now` $\wedge$ $\neg$`idle(active)`. With `active` declared, `timesched(active)` is automatically introduced as an integer variable. Section 4.8 shows that only limited user guidance is generally needed in finding variables. For any variable declared via a proposition, LVR automatically adds two proof targets, namely the existence and uniqueness of the variable under the given liveness prerequisite, to ensure the variable is well-defined. For `active` in ticket lock, the two proof targets are:

$\forall$`C:client.waiting(C)` $\wedge$ $\neg$`entered(C)` $\rightarrow$

$\exists$`C': client.myt(C')` = `now` $\wedge$ $\neg$`idle(C')`

$\forall$`C:client.waiting(C)` $\wedge$ $\neg$`entered(C)` $\rightarrow$

$\forall C_1$: `client`, C_2: `client.myt(`C_1`)` = `now` $\wedge$ $\neg$`idle(`C_1`)` $\wedge$ `myt(`C_2`)` = `now` $\wedge$

$\neg$`idle(`C_2`)` $\rightarrow$ $C_1 = C_2$.

The two proof targets are both safety properties and can be proved using an automated invariant inference tool such as P-FOL-IC3.

4.3.2 Bounds

Given a set of integer variables, LVR determines their coefficients to synthesize a ranking function. Whether a ranking function decreases after a transition eventually comes down to how the mutable integer variables in the ranking function change. A variable is mutable, and declared as such in mypyvy, if it can change during protocol transitions. For example, Figure 4.3 shows that n_{exec} is mutable but M_{exec} is not. To analyze how mutable integer variables change, LVR first needs to determine the upper and lower bounds of these variables. For example, Table 4.1 shows the bounds for the mutable integer variables for the ticket lock protocol. Variables n_{exec}, `timesched(C)`, and `timesched(active)` are introduced from fairness assumptions and defined with upper bounds M_{exec} and M_{period}. From the liveness prerequisite `waiting(C)` $\wedge \neg$ `entered(C)`, the serving ticket `now` has not reached the ticket of a waiting client `C`, so $sub($`myt(C)`, `now`$) \geq 0$. Since `next` is the next free ticket to allocate, $sub($`myt(C)`, `next`$) \leq 0$. Since n_{wait} and n_{enter} count subsets of clients, they are bound to be nonnegative. Additionally, the ticket lock protocol ensures mutual exclusion, so n_{enter} cannot exceed 1.

LVR automatically infers these bounds using simple rule-based static analysis, which is formalized in Algorithm 7. LVR first infers conservative variable bounds in the first phase of the algorithm, then further refines the bounds by considering constraints from protocol transitions in the second phase of the algorithm.

In Phase 1, LVR retrieves bounds defined directly from trusted invariants or axioms. For the ticket lock protocol in Figure 4.3, the trusted invariants at Lines 51 and 52 give upper bounds `timesched(C)` $\leq M_{\text{period}}$, `timesched(active)` $\leq M_{\text{period}}$, and $n_{\text{exec}} \leq M_{\text{exec}}$. LVR then uses information from variable definitions. If the variable is a count over first-order relations, its lower bound is at least zero and its upper bound is constrained by the product of the maximum number of each type of object (*MaxObj*) in the relation; *MaxObj* is ∞ by default. For the ticket lock protocol, the number of waiting clients n_{wait} is defined as `count (C:client) where waiting(C)`, and the number of entered clients n_{enter} is defined as `count (C:client) where entered(C)`, so this count rule gives lower bounds $n_{\text{wait}} \geq 0$ and $n_{\text{enter}} \geq 0$. If the variable is an indicator for some

boolean expression, its lower bound is at least zero and its upper bound is at most one. The ticket lock protocol has no indicator variables, but other protocols may use them, such as alternating bit and sliding window, which are discussed in Section 4.8.

In Phase 2, LVR infers more accurate bounds using the transition information. Phase 2 guarantees that the final bounds define a subrange of the conservative bounds from Phase 1. For each variable v, LVR first initializes its bound according to the protocol initialization, then relaxes the bound for each transition based on how the transition updates the variable. If the transition does not change variable v, the bound remains the same. If the transition sets v to some constant $Const$, the bound is relaxed to include $Const$. If the transition increases v, its upper bound is relaxed to the conservative upper bound. If the transition decreases v, its lower bound is relaxed to the conservative lower bound. If the transition can either increase or decrease v, both its lower and upper bounds will be relaxed. For example, for the ticket lock protocol, the conservative bound from Phase 1 for n_{exec} is $(-\infty, M_{\text{exec}}]$, which is too loose. In Phase 2, n_{exec} starts with $[0, 0]$ after initialization, and is changed in only two transitions `execute` and `leave`. `execute` increments n_{exec} and falls into the case at Line 23, so the bound is relaxed to $[0, M_{\text{exec}}]$. `leave` sets n_{exec} to 0 and falls into the case at Line 19, so the bound is relaxed to include 0, which means it remains $[0, M_{\text{exec}}]$.

Algorithm 7 Static Inference of Integer Variable Bounds

Input: Distributed protocol $\mathcal{P}$ with transitions *Txs* and integer variables *Vars*
Output: Lower bound $lower[v]$ and upper bound $upper[v]$ for each $v \in Vars$

// Phase 1: infer conservative "hard" bounds that are independent of transitions
for $v \in Vars$ **do**
  $hardLower[v], hardUpper[v]$:= from_trusted($\mathcal{P}$, v)
  if v is count $(X_1 : sort_1, ..., X_n : sort_n)$ where $f(X_1, ..., X_n)$ **then**
    $hardLower[v] := \max\{0, hardLower[v]\}$
    $hardUpper[v] := \min\{\Pi_{i\in[1,n]} MaxObj(sort_i), hardUpper[v]\}$
  else if v is indicator *boolExpr* **then**
    $hardLower[v] := \max\{0, hardLower[v]\}$
    $hardUpper[v] := \min\{1, hardUpper[v]\}$
// Phase 2: infer more accurate bounds according to transitions
for $v \in Vars$ **do**
  $lower[v], upper[v]$:= from_inits($\mathcal{P}$, v)
  $lower[v] := \max\{lower[v], hardLower[v]\}$
  $upper[v] := \min\{upper[v], hardUpper[v]\}$
  for $tx \in Txs$ **do**
    // $new(v)|_{tx}$ denotes the value of v after transition tx
    if $new(v)|_{tx} = v$ **then**
      pass
    else if $new(v)|_{tx} = Const$ **then**
      assert $hardLower[v] \le Const \le hardUpper[v]$
      $lower[v] := \min\{lower[v], Const\}$
      $upper[v] := \max\{upper[v], Const\}$
    else if $new(v)|_{tx} \ge v$ **then**
      $upper[v] := hardUpper[v]$
    else if $new(v)|_{tx} \le v$ **then**
      $lower[v] := hardLower[v]$
    else
      $lower[v] := hardLower[v]$
      $upper[v] := hardUpper[v]$

The rule-based static analysis is incomplete, yet our experience is that it works reasonably well for most protocols. The bounds inferred may be too loose but are never too tight. The user should look through the inferred bounds, and tighten them if needed. For the ticket lock protocol, LVR does not determine the tight upper bound of 1 on n_{enter}. This tight bound is unlikely to be achieved via any static analysis of the protocol specification, even though it is obvious to a human since mutual exclusion means that no more than one client can enter the critical section. This is because

Table 4.1: Bounds of integer variables in the ticket lock protocol.

variable	lower bound	upper bound
n_{exec}	0	M_{exec}
`timesched(C)`	0	M_{period}
`timesched(active)`	0	M_{period}
$sub($`myt(C),now`$)$	0	∞
$sub($`myt(C),next`$)$	$-\infty$	0
n_{wait}	0	∞
n_{enter}	0	1

there is no mutual exclusion property explicitly in the specification itself. The same applies to $sub($`myt(C),now`$) \geq 0$, the other tight bound LVR does not infer. LVR will rely on the user to tighten these two bounds.

LVR verifies the bounds are correct using mypyvy. Each bound is a safety property on the protocol (e.g., `timesched(C)` $\geq 0 \wedge$ `timesched(C)` $\leq M_{\text{period}}$), so verifying it involves finding an inductive invariant. An invariant is inductive if it holds on initial protocol states, is preserved after each transition, and implies the bound. Most bounds are inductive and can be directly verified by mypyvy. For example, all the bounds in Table 4.1 are inductive except for two, $n_{\text{enter}} \leq 1$ and $sub($`myt(C),now`$) \geq 0$. For bounds that are not inductive, LVR uses an off-the-shelf invariant inference tool to find an inductive invariant that implies the bound, specifically P-FOL-IC3 because it supports the most recent version of mypyvy.

4.3.3 Deltas

Once LVR determines the bounds, it needs to analyze further how mutable integer variables in the ranking function change on each transition. These deltas help further constrain the coefficients of the ranking function. Deltas are determined as either actual integer values or in terms of their respective lower and upper bounds.

For each transition, it is important to account for how its arguments can affect those deltas if the arguments overlap with the quantified variables `V` of interest from Theorem 7 or existing variables in the specification. For example, for the ticket lock protocol, we are interested in proving that

a client `C` that is waiting will eventually be able to acquire the lock, specified in the liveness prerequisite `waiting(C)` $\wedge$ $\neg$`entered(C)`. Since each transition has a client `c` as an argument, LVR separately considers deltas for each transition when the argument `c = C` or `c` $\neq$ `C`. Since there is a special client `active` that holds the serving ticket, LVR also distinguishes between `c = active` or `c` $\neq$ `active`. If a transition has a branch, LVR decomposes the transition into two, one corresponding to the *true* branch and the other for the *false* branch. This allows fine-grained analysis of deltas in each case.

Table 4.2 shows the deltas for each transition of the ticket lock protocol. For example, for transition `enter(c)`, its precondition `myt(c) = now` implies that only client `active` can take this transition, so `c`$\neq$`active` is a contradiction. Therefore, this transition can be ignored for any client `c`$\neq$`active` since it will not affect those deltas. If `c` is our client of interest `C`, four variables will change. n_{wait} decreases by 1. n_{enter} increases by 1. Since `C` is scheduled in this transition, `timesched(C)` and `timesched(active)` are reset to 0. Since `timesched(C)` and `timesched(active)` have lower bound 0 and upper bound M_{period}, their deltas must be in range $[-M_{\text{period}}, 0]$. This also shows why LVR first determines the bounds for each variable, as they are used in the delta analysis. If `c` is not our client of interest `C`, `timesched(C)` increases by 1, while the deltas of the other three variables remain the same.

LVR first automatically infers contradictions to eliminate transitions that do not need to be considered in computing deltas. Let L be the liveness prerequisite and P be the precondition of some transition. If $L \rightarrow \neg P$ holds for the protocol, then the transition is contradictory. For example, `get(c)` in Figure 4.3 has a precondition P = `idle(c)`, while the liveness prerequisite L = `waiting(C)` $\wedge$ $\neg$`entered(C)` says that our client of interest `C` is already waiting, so intuitively the transition `get(c)` where `c = C` is not allowed. Formally, LVR checks if

$$\forall \texttt{C: client}, \texttt{c: client}.\ \texttt{waiting(C)} \wedge \neg\texttt{entered(C)} \wedge \texttt{c} = \texttt{C} \rightarrow \neg\texttt{idle(c)}$$

holds for the protocol by calling the invariant inference tool P-FOL-IC3, which returns an inductive

Table 4.2: Deltas of integer variables after each transition in ticket lock.

transition	variable	delta
`get(c≠C∧c≠active)`	n_{wait}	1
	$sub(\texttt{myt(C)}, \texttt{next})$	$(-\infty, \text{-}1]$
	`timesched(C)`	1
	`timesched(active)`	1
`fail(c=C∧c≠active)`	`timesched(C)`	$[-M_{\text{period}}, 0]$
	`timesched(active)`	1
`fail(c≠C∧c≠active)`	`timesched(C)`	1
	`timesched(active)`	1
`enter(c=C∧c=active)`	n_{wait}	-1
	n_{enter}	1
	`timesched(C)`	$[-M_{\text{period}}, 0]$
	`timesched(active)`	$[-M_{\text{period}}, 0]$
`enter(c≠C∧c=active)`	n_{wait}	-1
	n_{enter}	1
	`timesched(C)`	1
	`timesched(active)`	$[-M_{\text{period}}, 0]$
`execute(c≠C∧c=active)`	n_{exec}	1
	`timesched(C)`	1
	`timesched(active)`	$[-M_{\text{period}}, 0]$
`leave(c≠C∧c=active)`	n_{enter}	-1
	n_{exec}	$[-M_{\text{exec}}, 0]$
	$sub(\texttt{myt(C)}, \texttt{now})$	$(-\infty, \text{-}1]$
	`timesched(C)`	1
	`timesched(active)`	$[-M_{\text{period}}, 0]$
others	contradiction	contradiction

invariant. This means $L \rightarrow \neg P$ holds for the protocol so the transition `get(c)` where `c = C` is contradictory. If P-FOL-IC3 instead returns `safety violation`, there is no contradiction. Whether each transition is contradictory is independent, so LVR checks their respective $L \rightarrow \neg P$ in parallel.

For transitions that are not contradictory, LVR then automatically infers deltas using simple rule-based static analysis, which is formalized in Algorithm 8. LVR iteratively checks if the update of variable v matches one of five patterns in Lines 2-16, and calculates the delta accordingly. If none of the patterns is matched, LVR will conservatively assume the variable can change arbitrarily and give the loosest delta $[lower[v] - upper[v], upper[v] - lower[v]]$ at Lines 18-19. Similar to bounds, the deltas inferred via static analysis can only be too loose, and the user can manually

Algorithm 8 Static Inference of Integer Variable Deltas

Input: Distributed protocol $\mathcal{P}$, transition tx, integer variables *Vars* with their bounds *upper* and *lower*
Output: Upper and lower bound of delta $deltaUpper[v]|_{tx}$ and $deltaLower[v]|_{tx}$ for each $v \in Vars$ on transition tx

1: **for** $v \in Vars$ **do**
2: **if** $new(v)|_{tx} = v + Const$ **then**
3: $deltaLower[v]|_{tx} := Const$
4: $deltaUpper[v]|_{tx} := Const$
5: **else if** $new(v)|_{tx} = v + v'$ **then**
6: $deltaLower[v]|_{tx} := lower[v']$
7: $deltaUpper[v]|_{tx} := upper[v']$
8: **else if** $new(v)|_{tx} = Const$ **then**
9: $deltaLower[v]|_{tx} := Const - upper[v]$
10: $deltaUpper[v]|_{tx} := Const - lower[v]$
11: **else if** $new(v)|_{tx} \geq v + Const$ **then**
12: $deltaLower[v]|_{tx} := Const$
13: $deltaUpper[v]|_{tx} := upper[v] - lower[v]$
14: **else if** $new(v)|_{tx} \leq v + Const$ **then**
15: $deltaLower[v]|_{tx} := lower[v] - upper[v]$
16: $deltaUpper[v]|_{tx} := Const$
17: **else**
18: $deltaLower[v]|_{tx} := lower[v] - upper[v]$
19: $deltaUpper[v]|_{tx} := upper[v] - lower[v]$

tighten them if needed. For the ticket lock protocol, all deltas are accurately inferred by LVR.

To verify the deltas are correct, LVR proves that the deltas hold in the distributed protocol using mypyvy. To do this, LVR temporarily adds auxiliary variables to the protocol that record old values of integer variables before a transition (e.g., old_n_{wait}). Then the delta of an integer variable is encoded as a safety property (e.g., $n_{wait} = old\_n_{wait} + 1$ for transition `get(c ≠ C ∧ c ≠ active)`). LVR verifies deltas for different transitions in parallel. Bounds and contradictions that have been verified earlier are encoded as trusted invariants, as the inductiveness of the delta properties is dependent on them.

4.3.4 Protocol Refinement for Automatically Proving Safety Properties

LVR is designed to leverage invariant inference tools such as P-FOL-IC3, but even the ticket lock protocol specification in Figure 4.3 has integer variables. Unfortunately, invariant inference tools currently can only find inductive invariants to prove safety properties in first-order logic. They do not find invariants with integer arithmetic [33, 12, 26, 34, 42]. To address this problem, a key observation is that the state transitions generally only involve first-order relations while integer variables are used just to encode fairness assumptions. For example, the ticket lock protocol only uses integer variables n_{exec}, `timesched`, M_{exec}, and M_{period} for its fairness assumptions in Figure 4.3. We refer to the variables used for encoding fairness assumptions as fairness variables, the protocol with fairness variables as the *fairness-included protocol*, and the protocol with all fairness variables removed as the *fairness-free protocol*.

To use an invariant inference tool, LVR automatically removes all fairness variables, including all initializations, preconditions, and assignments that include them, to transform the *fairness-included protocol* to its *fairness-free protocol*. LVR also automatically transforms the safety property when needed. For example, the bound $n_{\text{enter}} \leq 1$ for the ticket lock protocol is translated into:

$$\forall \mathtt{C}_1\mathtt{:}\ \ \mathtt{client}, \mathtt{C}_2\mathtt{:}\ \ \mathtt{client.entered(C}_1\mathtt{)} \wedge \mathtt{entered(C}_2\mathtt{)} \rightarrow \mathtt{C}_1 = \mathtt{C}_2.$$

We can then use the fairness-free protocol for proving safety properties based on:

Theorem 8. *For any fairness-included protocol $\mathcal{P}_I$ that never assigns a fairness variable to a non-fairness variable, let $\mathcal{P}_F$ be its induced fairness-free protocol. For any safety property $\mathcal{S}$, if $\mathcal{S}$ holds on $\mathcal{P}_F$, then $\mathcal{S}$ must also hold on $\mathcal{P}_I$.*

Proof. Let Σ_I and Σ_F be the protocol state space for $\mathcal{P}_I$ and $\mathcal{P}_F$. Let $f : \Sigma_I \rightarrow \Sigma_F$ be the projection function from a fairness-included protocol state to a fairness-free protocol state by removing all fairness variables. For any $\sigma_I \in \Sigma_I$, if σ_I satisfies the protocol initialization conditions of $\mathcal{P}_I$, $f(\sigma_I)$ is guaranteed to satisfy the initialization conditions of $\mathcal{P}_F$, because the initialization conditions for $\mathcal{P}_F$ is a subset of those for $\mathcal{P}_I$.

Next consider any transition $\sigma_{I_1} \xrightarrow{tx} \sigma_{I_2}$ in $\mathcal{P}_I$. The transition tx is represented by a transition name with parameters. Since the preconditions of tx in $\mathcal{P}_F$ are a subset of those in $\mathcal{P}_I$, tx must also satisfy its preconditions for $f(\sigma_{I_1})$ in $\mathcal{P}_F$. Since there is no assignment from fairness variables to non-fairness variables, $f(\sigma_{I_1}) \xrightarrow{tx} f(\sigma_{I_2})$ is a valid transition in $\mathcal{P}_F$.

The two facts above establish a refinement relation between state machines of $\mathcal{P}_I$ and $\mathcal{P}_F$. For any state $\sigma_I^* \in \Sigma_I$ reachable via $\sigma_{I_0} \rightsquigarrow \sigma_I^*$ in $\mathcal{P}_I$, we know $f(\sigma_I^*)$ is reachable via $f(\sigma_{I_0}) \rightsquigarrow f(\sigma_I^*)$ in $\mathcal{P}_F$. So any safety property $\mathcal{S}$ that holds on $\mathcal{P}_F$ also holds on $\mathcal{P}_I$. □

Theorem 8 allows us to prove safety properties on the fairness-free protocol, where invariant inference tools are more likely to be applicable, then automatically extend the result to the fairness-included protocol. Section 4.8 shows that all the protocols we evaluated except for one have fairness-free counterparts that contain no arithmetic so that an invariant inference tool can use them to verify bounds and deltas. The only exception is the sliding window protocol, where integer arithmetic is embedded in the protocol itself to reason about the window size `width`. As a result, the user has to manually write an inductive invariant to verify bounds and contradictions for this protocol, as shown in Table 4.5.

Theorem 8 does not require fairness variables to be completely independent of protocol variables, although they usually are in practice. Instead, it only requires that the fairness variables are never assigned to protocol variables. If some precondition involves both fairness and protocol variables, it will be removed in the fairness-free protocol. This may lead to weaker preconditions in the fairness-free protocol, and thus additional safety violations. Consequently, Theorem 8 does not guarantee that a safety violation in the fairness-free protocol implies a safety violation in the fairness-included protocol, so theoretically, the invariant inference tool may fail on the fairness-free protocol when an inductive invariant does exist for the fairness-included protocol. However, LVR does not expect the invariant inference tool to always succeed in the first place. When the inference tool times out or throws an error, the problem is simply deferred to the user to manually check the property and write an inductive invariant if needed.

The user does not need to understand the state-machine refinement in Theorem 8, since both

the protocol retrofitting and the property translation are handled by LVR behind the scenes. The user is only involved when the protocol does not satisfy the prerequisite of Theorem 8. Specifically, LVR will alert the user if a fairness variable is assigned to a non-fairness variable, which indicates that the fairness assumption may have been written incorrectly.

After P-FOL-IC3 returns an inductive invariant for some safety property, LVR feeds that inductive invariant to the fairness-included protocol and lets mypyvy validate it. In this way, the user does not need to trust LVR's retrofitting and translation procedure.

4.4 SMT-based ranking function synthesis

Based on the proven bounds and deltas, LVR assembles a ranking function using the integer variables. LVR formulates the ranking function as a weighted linear combination of mutable integer variables; Section 4.5 discusses more complex functions. Inferring the ranking function reduces to determining how much weight to give to each variable, which can be expressed as a set of coefficients that can be solved by an SMT solver.

For example, for the ticket lock protocol, the ranking function has the shape:

$$\begin{aligned} R(\texttt{C}) = {} & W_1 \times sub(\texttt{myt(C)}, \texttt{now}) + W_2 \times sub(\texttt{myt(C)}, \texttt{next}) \\ & + W_3 \times n_{\text{exec}} + W_4 \times \texttt{timesched(C)} \\ & + W_5 \times \texttt{timesched(active)} + W_6 \times n_{\text{wait}} + W_7 \times n_{\text{enter}}, \end{aligned} \tag{4.13}$$

where $W_1, \ldots W_7$ are the weights. Each weight is further formulated as a sub-formula of immutable integer variables. If we allow up to second-degree polynomials of immutable integer variables, then a sub-formula for weight could be

$$\begin{aligned} W_1 = {} & x_1 \times M_{\text{period}}^2 + x_2 \times M_{\text{period}} \times M_{\text{exec}} + x_3 \times M_{\text{exec}}^2 \\ & + x_4 \times M_{\text{period}} + x_5 \times M_{\text{exec}} + x_6. \end{aligned} \tag{4.14}$$

For the ticket lock protocol, there are two immutable integer variables, so considering all possible combinations of those variables up to a second-degree term results in six terms with six coefficients $x_1, ..., x_6$. Expanding $W_2, ..., W_6$ in the same way defines coefficients $x_7, ..., x_{42}$, resulting in 42 coefficients.

The bounds and deltas on the mutable integer variables, together with the requirement that the ranking function decreases for each transition while remaining nonnegative, will impose a set of constraints on the weights and consequently the coefficients of the immutable integer variables as well. The constraint generation process is formalized in Algorithm 9.

LVR first considers nonnegativity of the ranking function which depends on variables' bounds. If a variable v has an open upper bound, then it cannot have a negative weight in the ranking function (Line 5). If a variable v has an open lower bound, then it cannot have a positive weight in the ranking function (Line 7).

Next for each transition tx, LVR considers the decreasing of the ranking function which depends on variables' deltas. This is a linear programming problem, where the delta of each $v \in Vars$ is a variable x_v with constraint $deltaLower[v]|_{tx} \leq x_v \leq deltaUpper[v]|_{tx}$, and the ranking function changes by $\sum_{v \in Vars} W_v \cdot x_v$ after a transition. The constraints define a feasible region of possible solutions, and $\max \sum_{v \in Vars} W_v \cdot x_v$ must occur at a vertex of the feasible region if it exists. Since $\max \sum_{v \in Vars} W_v \cdot x_v < 0$ must hold for the ranking function to decrease, LVR adds constraints on the weights to ensure a maximum ranking function change exists and it is less than zero. If a variable v can increase infinitely in a transition, then x_v has an open upper bound and cannot have a positive weight (Line 14). Similarly, if a variable v can decrease infinitely in a transition, then x_v has an open lower bound and cannot have a negative weight (Line 18). A variable x_v has specific values at each vertex, so LVR then enumerates all vertices and adds a constraint on the weights for each vertex so that the ranking function change is negative.

For simplicity, suppose no variable $v \in Vars$ can increase or decrease infinitely, and for each $v \in Vars$ we have $deltaUpper[v]|_{tx} \neq deltaLower[v]|_{tx}$. Then the feasible region is a $|Vars|$-dimensional hyperrectangle. For each $v \in Vars$, *deltaSet* will have two elements $deltaUpper[v]|_{tx}$

and $deltaLower[v]|_{tx}$, added at Lines 16 and 20. Then the Cartesian product over the $|Vars|$ deltaSets will give $2^{|Vars|}$ elements, each representing a vertex of the hyperrectangle (Line 24). For each vertex, LVR adds a constraint saying that the change of the ranking function is negative (Line 25). In practice, the number of generated constraints is usually much fewer than $2^{|Vars|}$, since a transition usually changes only a subset of integer variables, and among them many have their deltas as a number (i.e., $deltaUpper[v]|_{tx} = deltaLower[v]|_{tx}$) instead of a range.

Now we have a set of constraints over weights $W_1, W_2, ...$ and consequently the underlying coefficients $x_1, x_2, ...$ Solving for the coefficients across a set of inequalities can be extremely complicated, especially for multivariable and higher-order functions. We make two observations regarding practical distributed protocols that can simplify this analysis. First, we observe that the immutable variables are generally nonnegative integers, such as the one used for bounds in the case of the ticket lock protocol. Second, we conjecture that there should exist a simple ranking function.

Based on these observations, LVR makes a simplifying assumption that for any inequality, the weight and the coefficients of all of its monomials have the same sign. For example, if $W_1 \geq 0$, then all $x_1, ...x_{10} \geq 0$. Similarly if $W_1 \leq 0$, then all $x_1, ...x_{10} \leq 0$. By assuming the weights and coefficients have the same sign, complex minima/maxima analysis of multivariate high-order functions can be reduced to simple linear inequalities. For example, suppose $W_1 = x_1 \times M_{\text{exec}}^2 + x_2 \times M_{\text{exec}} + x3$, and some constraint requires $W_1 \geq 0$ for any $M_{\text{exec}} \geq 0$. This constraint can be satisfied if $(x_1 \geq 0 \wedge x_2 \geq 0 \wedge x_3 \geq 0)$ or $(x_1 > 0 \wedge x_2 < 0 \wedge 4x_1x_3 - x_2^2 \geq 0)$. LVR only considers the former and not the latter, simplifying its analysis.

In other words, the constraints on all of the coefficients can be expressed as simple linear inequalities. For example, $sub(\texttt{myt(C)}, \texttt{now})$ has an open upper bound ∞, so $W_1 \geq 0$ otherwise the ranking function $R(\texttt{C})$ cannot be nonnegative. For transition `execute(c≠C∧c=active)`, the delta in Table 4.2 shows that n_{exec} and `timesched(C)` increases by 1, and `timesched(active)` changes by $[-M_{\text{period}},0]$. To make the ranking function decrease after this transition, LVR gets $W_3 + W_4 < 0$ and $W_3 + W_4 - M_{\text{period}} \times W_5 < 0$. These constraints are then decomposed into

Algorithm 9 Constraint Generation for Ranking Function

Input: Integer variables *Vars*, with their bounds *upper* and *lower*, and their deltas $deltaUpper|_{tx}$ and $deltaLower|_{tx}$ for each transition $tx \in Txs$
Output: A set of linear inequality constraints $\mathcal{I}$ over weights

$\mathcal{I} := \emptyset$
for $v \in Vars$ **do**
  declare weight W_v
  if $upper[v]$ is ∞ **then**
    $\mathcal{I} := \mathcal{I} \cup \{W_v \geq 0\}$
  if $lower[v]$ is $-\infty$ **then**
    $\mathcal{I} := \mathcal{I} \cup \{W_v \leq 0\}$
for $tx \in Txs$ **do**
  $varList := []$
  $deltaSetList := []$
  for $v \in Vars$ **do**
    $deltaSet := \emptyset$
    if $deltaUpper[v]|_{tx}$ is ∞ **then**
      $\mathcal{I} := \mathcal{I} \cup \{W_v \leq 0\}$
    else
      $deltaSet := deltaSet \cup \{deltaUpper[v]|_{tx}\}$
    if $deltaLower[v]|_{tx}$ is $-\infty$ **then**
      $\mathcal{I} := \mathcal{I} \cup \{W_v \geq 0\}$
    else
      $deltaSet := deltaSet \cup \{deltaLower[v]|_{tx}\}$
    if $deltaSet \neq \emptyset$ **then**
      Append($varList$, v)
      Append($deltaSetList$, $deltaSet$)
  for $deltaEachVar \in$ CartesianProduct($deltaSetList[0], ..., deltaSetList[|varList| - 1]$) **do**
    $\mathcal{I} := \mathcal{I} \cup \{\sum_{i \in [0, |varList|-1]} deltaEachVar[i] \cdot W_{varList[i]} < 0\}$

inequalities of coefficients $x_1, ..., x_{42}$.

Eventually 94 linear inequalities are generated. Any assignment of the 42 coefficients that satisfies the 94 inequalities corresponds to a valid ranking function. LVR feeds the inequalities to the Z3 SMT solver, which returns a solution $x_2 = 1, x_4 = 2, x_5 = 1, x_6 = 2, x_{16} = -1, x_{18} = -1, x_{30} = -1, x_{40} = -1, x_{42} = -1$, with every other x_i being 0. Then LVR automatically adds terms involving only immutable integer variables to make the ranking function nonnegative, leading to the final ranking function Eq. (4.7) for the ticket lock protocol.

While the ranking function shape in Eq. (4.13) is linear with regard to mutable integer variables, LVR can also infer piecewise linear ranking functions. In the TLB shootdown protocol discussed in Section 4.8, the ranking function is piecewise linear and depends on the program counter pc:

$$R(\texttt{C}) = \begin{cases} R_1(\texttt{C}) & pc = i1 \\ R_2(\texttt{C}) & pc = i2 \\ \ldots \end{cases}$$

The translation to inequalities is mostly the same as above, with additional considerations to switch the ranking function. While ranking functions can theoretically be of any form, we observe that simpler linear and piecewise linear ranking functions of mutable integer variables can often be sufficient for practical distributed protocols, especially in combination with tiering as discussed in Section 4.5.

Validation of ranking functions Once the ranking function has been synthesized, LVR verifies it using mypyvy. Since the synthesis of the ranking function was itself accomplished using an SMT solver to verify different steps of the synthesis process, it is expected that the ranking function will be correct. However, we perform an additional validation step to ensure that the correctness of the ranking function does not need to trust LVR and is guarded by an SMT solver. LVR validates the ranking function by providing mypyvy with the prerequisite, protocol specification, bounds and deltas, inductive invariants for verifying bounds and deltas, as well as the ranking function itself. mypyvy feeds all the information into the underlying Z3 SMT solver to verify the ranking function is strictly decreasing and nonnegative for each protocol transition. Alternatively, if the bounds and deltas are trusted since they were previously verified using an SMT solver, they do not have to be reverified and the inductive invariants for verifying the bounds and deltas can be omitted in verifying the ranking function, reducing the constraints that need to be considered by the SMT solver. Since whether a ranking function decreases and remains nonnegative after one transition is independent of another transition, LVR validates the ranking function on different transitions in

parallel.

4.5 Tiered Ranking Functions for Scalability

4.5.1 Motivation and Example

Although the ranking functions previously discussed are linear combinations of mutable integer variables, they are higher-degree polynomials from the perspective of an SMT solver in terms of reasoning about a combination of both mutable and immutable integer variables. For example, the ranking function for ticket lock protocol in Eq. (4.7) is a third-degree polynomial. For more complex protocols, the degree can be higher with many more terms, which poses challenges to not only synthesizing the ranking function but also validating it. For example, for the sliding window network transmission protocol discussed in Section 4.8, the entire ranking function is actually a fourth-degree polynomial, on which mypyvy times out after 24 hours when trying to prove, all at once, that it decreases after a transition.

To address this problem, LVR introduces tiered ranking functions to modularize liveness reasoning and make proofs easier for SMT solvers. Rather than synthesizing a ranking function all at once, which is valid for all protocol transitions, we can decompose the transitions into tiers and synthesize a ranking function for each tier, then verify ranking functions incrementally tier by tier. Each tier encompasses a subset of the transitions. LVR synthesizes a tiered ranking function for the first tier, ignoring any transitions not included in that tier. The tiered ranking function is the same as a regular ranking function, except it only has to be strictly decreasing for transitions in the respective tier. For transitions not included in that tier but part of subsequent tiers, the tiered ranking function can remain unchanged, but it cannot increase. Once a tiered ranking function is found, that intuitively means that the liveness property can be verified for the distributed protocol assuming only transitions in that tier occur.

However, we also need to make sure that transitions occurring in other tiers do not repeatedly happen between transitions included in the first tier such that they prevent first-tier transitions from happening. LVR accomplishes this by next synthesizing a tiered ranking function for the next tier.

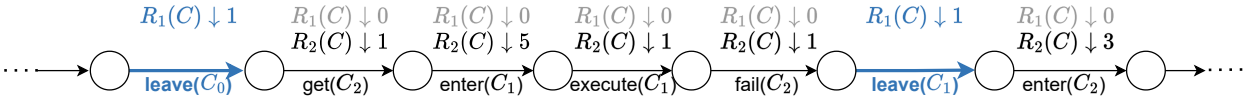


Figure 4.5: Tiered ranking function for ticket lock protocol. Both R_1 `(C)` and R_2 `(C)` are nonnegative. The top-tier ranking function R_1 `(C)` strictly decreases on transition `leave` and does not increase on others. So the system will terminate within finite steps of `leave`. The second tier ranking function R_2 `(C)` strictly decreases on every transition other than `leave`, so within finite steps of these transitions, the system will either take transition `leave` or terminate. Thus, the system will terminate in finite steps, regardless of what transitions are taken.

Once that tiered ranking function is found, that intuitively means that those transitions will not go on forever and prevent the first-tier transitions from occurring. This implies that the liveness property can be verified for the distributed protocol assuming only transitions in the first two tiers occur. The same process is repeated for each tier to verify the liveness property across all tiers, and therefore all transitions.

For example, we can use tiered ranking functions to prove the liveness of the ticket lock protocol, instead of constructing one ranking function all at once for all transitions. Figure 4.5 gives an illustration. At a high level, a waiting client `C` eventually will enter the critical section because the serving ticket increases whenever a client leaves the critical section, so it will sooner or later reach client `C`'s ticket so it can enter the critical section. This can be stated as "if transition `leave` keeps happening, then our desired `entered(C)` will happen." If `leave` is the only transition included in a first tier, its liveness property can be proved using a tiered ranking function that decreases in transition `leave` and does not increase in other transitions while remaining nonnegative. Such a ranking function can be automatically inferred in the same way as discussed in Section 4.4. The only difference is that, for transitions other than `leave`, the ranking function can remain unchanged (but not increase), so < 0 will be replaced by ≤ 0 in the respective constraints. LVR successfully finds a first-tier ranking function

$$R_1(\texttt{C}) = sub(\texttt{myt(C)}, \texttt{now}).$$

The liveness property will hold as long as no other transitions keep `leave` from happening.

If all other transitions are included in a second tier, then we want to prove that those transitions do not happen infinitely often to keep `leave` from happening, or the `enter` transition in the second tier directly occurs for `C`, so the desired `entered(C)` condition happens. In other words, we want to prove a liveness property for the remaining transitions, not including `leave`. This can be done using a tiered ranking function that decreases in all transitions except `leave` while remaining nonnegative. This can be automatically inferred as well in the same way as discussed in Section 4.4. LVR successfully finds a second-tier ranking function

$$R_{2\,\texttt{(C)}} = (M_{\text{period}} + 1) \times (M_{\text{exec}} + 1 - n_{\text{exec}} - n_{\text{enter}}) + M_{\text{period}} - \texttt{timesched(active)}.$$

In this way, the ranking function in Eq. (4.7), a third-degree polynomial with 14 terms, is decomposed into two smaller tiered ones of lower degree, the first having only one term, and the second being a second-degree polynomial with eight terms. Each tiered ranking function can be separately verified by an SMT solver, avoiding the need for the solver to consider higher-degree polynomials.

4.5.2 General Form and Proof of Correctness

More formally, a k-tier ranking function is a length-k sequence of pairs:

$$[(R_1(V), TX_1); ...; (R_k(V), TX_k)],$$

where $R_i(V)$ is the ranking function for the i-th tier ($i \in [1, ..., k]$), and TX_i is the set of transitions belonging to that tier. The union $\bigcup_{1 \le i \le k} TX_i$ must be the set of all transitions in the protocol.

For tiered ranking functions, it is necessary to prove that it decreases after each transition and remains nonnegative, similar to single ranking functions, but the proof needs to be done for each tier. Specifically, we need to prove nonnegativity for each tier:

$$\forall i \in [1, ..., k],\ \forall \texttt{V}.\ (\texttt{trigger(V)} \wedge \neg \texttt{good(V)}) \rightarrow R_{i\,\texttt{(V)}} \ge 0. \tag{4.15}$$

We also need to prove it decreases for each tier such that after any transition in tier i, the ranking function of tier i strictly decreases, as well as all higher-tier ranking functions do not increase:

$$\begin{aligned}\forall i \in [1, ..., k], \forall tx \in TX_i,\ \forall \mathtt{V}.\ \mathbf{old}(\mathtt{trigger(V)} \wedge \neg \mathtt{good(V)}) \rightarrow \\ R_i\,(\mathtt{V}) < \mathbf{old}(R_i\,(\mathtt{V}))\wedge \\ \wedge_{j\in[1,...,i-1]}\, R_j\,(\mathtt{V}) \leq \mathbf{old}(R_j\,(\mathtt{V})).\end{aligned} \tag{4.16}$$

The soundness of tiered ranking functions for proving liveness properties is established by:

Theorem 9. *If there exists a k-tier ranking function* $[(R_1(V), TX_1); ...; (R_k(V), TX_k)]$ *that satisfies Eq.* (4.15)(4.16)*, then the protocol cannot have an infinite sequence of states satisfying* `trigger(V)` $\wedge$ `¬good(V)`.

Proof. We prove this by induction. For the base case when $k = 1$, the ranking function has only one tier, and Eq. (4.15)(4.16) degenerate to Eq. (4.9)(4.10) in Section 4.2. Then the proof for Theorem 7 directly applies.

Now suppose the proposition holds for any n-tier ranking function. We will show it also holds any $(n + 1)$-tier ranking function by contradiction. Suppose we have an infinite sequence of states satisfying `trigger(V)` $\wedge$ `¬good(V)`. Let $i = 1$ in Eq. (4.15)(4.16). We get

$$\forall \mathtt{V}.\ (\mathtt{trigger(V)} \wedge \neg \mathtt{good(V)}) \rightarrow R_1\,(\mathtt{V}) \geq 0 \tag{4.17}$$

$$\forall tx \in TX_1,\ \forall \mathtt{V}.\ \mathbf{old}(\mathtt{trigger(V)} \wedge \neg \mathtt{good(V)}) \rightarrow R_1\,(\mathtt{V}) < \mathbf{old}(R_1\,(\mathtt{V})). \tag{4.18}$$

Let σ_1 be a state in the infinite sequence. From Eq. (4.17) we know $R_1(V)$ is a nonnegative integer at σ_1. Eq. (4.18) tells that $R_1(V)$ strictly decreases whenever a transition $tx \in TX_1$ is taken. So the number of transitions belonging to TX_1 after σ_1 must be finite. Let σ_2 be the state after the last transition $tx \in TX_1$ in the infinite sequence. The sequence after σ_2 is still an infinite sequence satisfying `trigger(V)` $\wedge$ `¬good(V)` but does not involve TX_1. We can construct a new protocol without TX_1 and a new n-tier ranking function $[(R_2(V), TX_2); ...; (R_k(V), TX_k)]$. The new ranking

function satisfies Eq. (4.15)(4.16) for the new protocol. From the induction hypothesis, it does not include any infinite sequence of states satisfying `trigger(V)` $\wedge \neg$ `good(V)`, a contradiction. □

The proof of Theorem 9 establishes the logical equivalence between a k-tier ranking function and a lexicographic ranking function of length k. The difference is that the former can be easily encoded and reasoned about in mypyvy, while the latter cannot be directly expressed in mypyvy due to its restrictions to first-order logic and integer arithmetic.

To use tiered ranking functions to prove liveness for a distributed protocol, LVR requires the user to identify which transitions belong to each tier and the number of tiers to consider. Tiering is not unique, so users can make different choices of tiers when verifying the liveness properties of a protocol. Our evaluation suggests that tiering is not required to verify liveness properties for most distributed protocols, but can be useful in some cases and requires only a modest amount of additional information from the user.

4.6 Theoretical Gap in Abstract Bound

When encoding the fairness assumptions of ticket lock in Section 4.1, we let variable `timesched`(C) track the time since a client `C` was last scheduled, and give an abstract declaration of M_{period} representing the upper bound of the time delta from one scheduled to the next. The fairness assumption that every client is scheduled infinitely often is then transformed to Eq. (4.6) which is

$$\forall \texttt{C: client}, \forall \texttt{T: time}, \texttt{timesched(C)}|_{\texttt{T}} \leq M_{\text{period}}.$$

Based on this fairness assumption, LVR proves liveness for *any* M_{period} instead of some specific value. We mentioned in Section 4.1 that there is a minor theoretical difference between Eq. (4.5) and Eq. (4.6). Specifically, there exist execution traces in which client `C` is scheduled infinitely often, but the scheduling interval has no upper bound—if a client is scheduled at time $1, 2, 4, ..., 2^n, ...$ and the sequence grows infinitely, then temporal logic formulation in Eq. (4.5) holds, but no M_{period}

exists that satisfies Eq. (4.6).

This does not mean the liveness proof is unsound. Instead, it relies on a slightly stronger fairness assumption. If some server's implementation satisfies the fairness assumption in Eq. (4.5) but not Eq. (4.6), then the liveness proof does not apply to such a server implementation.

LVR can also prove liveness without introducing abstract bounds. If one prefers to verify ticket lock using the original formulation in Eq. (4.5), instead of a global bound M_{period}, LVR will introduce a prophecy variable `period(C)` that predicts when client `C` will be scheduled next. `period(C)` is nondeterministically reset to any nonnegative integer when `C` is scheduled. The fairness assumption will then be encoded as

$$\forall \texttt{C: client}, \forall \texttt{T: time}, \texttt{timesched(C)}|_{\texttt{T}} \leq \texttt{period(C)}.$$

In other words, `period(C)` is a dynamic bound for each scheduling decision. However, without an upper bound on `period(C)`, when `C` is scheduled `period(C)` will change by $(-\infty, \infty)$. The simplest ranking function will be the three-tuple

$$(sub(\texttt{myt(C)}, \texttt{now}),\ M_{\text{exec}} + 1 - n_{\text{exec}} - n_{\text{enter}},\ \texttt{period(active)} - \texttt{timesched(active)})$$

in lexicographic order. This can also be written as a three-tier ranking function. $sub(\texttt{myt(C)}, \texttt{now})$ strictly decreases on transition `leave`. $M_{\text{exec}} + 1 - n_{\text{exec}} - n_{\text{enter}}$ strictly decreases on transitions `enter` and `execute`. `period(active)`$-$`timesched(active)` strictly decreases on transitions `get` and `fail`. As long as the user puts the five transitions in the aforementioned three tiers, LVR can automatically infer the ranking function of each tier.

From our perspective, the small theoretical gain of prophecy variables is outweighed by the additional complexity and human effort in tiering, so abstract bounds are used by default. A user can make their own choice when using LVR.

4.7 Proving Absence of Deadlocks

Under the liveness verification scheme formalized in Theorem 7, besides proving a ranking function, we need to prove two safety properties: the system has no deadlock (Eq. (4.11)), and can only end in a good state (Eq. (4.12)). The second property is usually easy to prove. For ticket lock, this means a waiting client cannot go back to idle without entering the critical section. The property is inductive and can be directly proved by mypyvy.

To prove the first "no deadlock" property, we need to show that at any state before the desired liveness condition is satisfied, there must be at least one transition that can be taken. A transition is takeable if its preconditions are satisfied. For example, for the ticket lock protocol, this means

$$
\begin{aligned}
&\forall \texttt{C: client}.\ \texttt{waiting(C)} \wedge \neg \texttt{entered(C)} \rightarrow \\
&\qquad (\exists \texttt{C': client}.\ \text{preconditions of } \texttt{get(C')}) \\
&\qquad \vee (\exists \texttt{C': client}.\ \text{preconditions of } \texttt{fail(C')}) \\
&\qquad \vee (\exists \texttt{C': client}.\ \text{preconditions of } \texttt{enter(C')}) \\
&\qquad \vee (\exists \texttt{C': client}.\ \text{preconditions of } \texttt{execute(C')}) \\
&\qquad \vee (\exists \texttt{C': client}.\ \text{preconditions of } \texttt{leave(C')}).
\end{aligned} \tag{4.19}
$$

In other words, at least one of the five transitions must be takeable for some client `C`. This is a safety property, so we can prove it by using an off-the-shelf tool to find the inductive invariant, same as proving bounds and deltas. P-FOL-IC3 successfully finds an inductive invariant to prove Eq. (4.19) in two minutes.

For some complex protocols, examining all transitions to prove the absence of deadlock may result in too many terms to consider to find the inductive invariant, causing the underlying SMT solver used by an invariant inference tool to time out. For example, if a distributed protocol has 10 transitions, each transition has three preconditions, and each precondition has two terms, there are 60 terms to resolve in a single formula, which is beyond the capabilities of existing automated

invariant inference tools. To avoid this, LVR allows the user to provide additional guidance to simplify invariant inference by specifying a subset of transitions that are sufficient to ensure the absence of deadlock and eliminating existentially quantified variables.

For example, for the ticket lock protocol, we can simplify Eq. (4.19) by specifying which transitions need to be considered. We know at any time before the desired `entered(C)` condition is satisfied, 1) if no client is holding the critical section, then entering is a valid transition for the client whose local ticket equal to `now`, or 2) if some client `c′` is holding the critical section, then leaving is a valid transition for `c′`. We can therefore narrow the set of transitions to consider from five to two, namely `enter` and `leave`, while excluding the other three transitions `get`, `fail`, and `execute`. If we further expand the exact preconditions, we can simplify Eq. (4.19) to:

$$\begin{aligned}\forall \texttt{C: client.}\ &\texttt{waiting(C)} \wedge \neg\texttt{entered(C)} \rightarrow \\ &(\exists \texttt{C′: client.}\ \texttt{waiting(C′)} \wedge \texttt{myt(C′)} = \texttt{now}) \\ &\vee (\exists \texttt{C′: client.}\ \texttt{entered(C′)}).\end{aligned} \tag{4.20}$$

We can further eliminate an existential quantifier because we know which `C′` can enter or leave. "The client whose local ticket equals to now" is exactly the declared variable `active`, whose existence and uniqueness have been verified earlier (see Section 4.3.1), so one can narrow the choice of `C′` to exactly `active`. This further simplifies Eq. (4.20) to:

$$\begin{aligned}\forall \texttt{C: client.}\ &\texttt{waiting(C)} \wedge \neg\texttt{entered(C)} \rightarrow \\ &(\texttt{waiting(active)} \wedge \texttt{myt(active)} = \texttt{now}) \\ &\vee (\texttt{entered(active)}).\end{aligned} \tag{4.21}$$

Note that we only use the ticket lock protocol as an example of how user guidance can simplify invariant inference for proving deadlock freedom. In fact, the ticket lock protocol is simple enough that user guidance is actually not necessary to prevent the underlying SMT solver from timing out

Table 4.3: Distributed protocols evaluated.

name	description
ticket lock	Shown in Figure 4.3. Liveness property: any waiting client will eventually enter the critical section.
HRB	Broadcasts messages in a network consisting of correct, omittable, and byzantine nodes. Liveness property: any correct node will eventually accept a message broadcast by all non-byzantine nodes.
alternating bit [43]	Involves a sender and receiver transmitting over a channel with packet loss. Liveness property: any data from the sender will eventually be received by the receiver.
TLB shootdown [44]	Ensures TLB consistency. Liveness property: after a shared memory write, each processor will eventually update its TLB correctly.
Paxos [27, 28]	Enables nodes to reach a consensus value in an unreliable network. Liveness property: eventually some consensus must be reached.
multi-Paxos [28]	An extension of Paxos, where nodes reach consensus on a growing log instead of a single value.
stoppable Paxos [45]	An extension of multi-Paxos that allows reconfiguration of the system (e.g., adding new nodes).
sliding window [43]	An extension of alternating bit where the sender can send consecutive pieces of data up to its window size, and the network can both drop and reorder packets.

when proving the absence of deadlocks.

4.8 Evaluation

4.8.1 Distributed Protocols and Their Liveness

We demonstrate the effectiveness of LVR by evaluating it on eight distributed protocols, including all protocols whose liveness has been manually verified by either Ironfleet [5] or Dynamic Abstraction [14, 15].

Table 4.3 describes each protocol and its liveness property. Each liveness property also requires fairness assumptions. Ticket lock requires fair scheduling and finite execution in the critical section. Hybrid reliable broadcast (HRB), alternating bit, and sliding window are network protocols that require some minimum guarantee on packet delivery. TLB shootdown requires scheduling and lock fairness. Paxos and its variants require some period of synchrony with some minimum guar-

antee on packet delivery, and limited contention between proposers across rounds. Specifically, Paxos requires that there exists a round r_0 which no later round contends with. The leader of round r_0 carries out its duty in initializing the round and proposing a value. For any packet that belongs to round r_0, if it is sent between the leader and a participant that belongs to a live quorum q_0, then it will eventually be delivered.

For protocols manually verified by Dynamic Abstraction, we translate their Ivy specifications [14, 15] to its twin language, mypyvy. We use mypyvy because it is written in Python 3 and supports both Z3 and CVC4 SMT solvers. We observe that for some protocols, the same ranking function can be validated by mypyvy but times out on Ivy. For the sliding window protocol which was manually verified by IronFleet, we reimplemented its Dafny specifications [46] in mypyvy, which involved some necessary rewriting due to language differences. The reimplementation is logically equivalent but not a line-by-line translation. IronFleet's sliding window protocol is used as a module to guarantee packet delivery between hosts in a key-value store, but how each host maintains its hashtable and generates new data to send is unrelated to the protocol. For IronFleet's sliding window, our reimplementation implements the network protocol. The rest of the system is unrelated to its liveness reasoning, so we abstract the host by simply introducing a `gen_data` transition that nondeterministically generates a new message to be sent, similar to the alternating bit protocol in Dynamic Abstraction. Algebraic datatypes in Dafny are encoded as relations or functions in mypyvy.

4.8.2 Results and Discussion

Tables 4.4 and 4.5 show the results for using LVR to verify the liveness for each protocol. In all cases, LVR automatically synthesizes a ranking function for each protocol and verifies all eight protocols, while only requiring limited user guidance. Table 4.4 shows various protocol characteristics, and characteristics of the verification process for each protocol. It shows the number of transitions and lines of code (LoC). It also shows the number of mutable integer variables from three sources: explicitly declared as `int` in the specification, generated from `int`-typed functions,

Table 4.4: Protocol statistics and user guidance for verifying liveness properties of distributed protocols.

			integer variables			user guidance				
protocol name	transitions	LoC	declare	function	derive	variable	bound	contradiction	delta	tier
ticket lock	5	68	1	4	2	1	2	0	0	-
HRB	8	137	0	0	2	2	0	0	0	-
alternating bit	9	127	3	2	5	3	2	0	0	2
TLB shootdown	27	361	0	0	25	1	0	3	0	4
Paxos	9	133	0	0	9	4	0	3	0	-
multi-Paxos	10	135	0	0	10	5	0	0	0	-
stoppable Paxos	11	160	0	0	10	5	0	0	0	-
sliding window	8	119	1	6	2	1	2	2	0	4

and derived from first-order relations.

Table 4.5 shows the runtime for each stage in LVR: 1) static analysis to determine integer variables, bounds, and deltas, 2) proving bounds, 3) determining which transitions are contradictory, 4) proving deltas for non-contradictory transitions, 5) synthesizing the ranking function with the Z3 SMT solver, 6) validating the ranking function using mypyvy, 7) proving the "ending in good state" property, and 8) proving the absence of deadlock. For steps 2, 3, 4, 7, and 8, P-FOL-IC3 is called when the property is not inductive. If P-FOL-IC3 times out determining whether a transition is contradictory, the transition is conservatively considered non-contradictory; a one hour timeout threshold is used. The user can add and prove a missing contradiction by rerunning P-FOL-IC3 or manually writing the inductive invariant. In our experiments, LVR keeps running until reaching the timeout threshold, but Table 4.5 shows the actual runtime required until the last contradiction is found. When tiered ranking functions are used, the results reported are the sum totals across all tiers. In the LVR workflow, static analysis, constraint solving, and ranking function validation are all deterministic and the runtime is stable across trials, while invariant inference using P-FOL-IC3 has highly variable runtimes due to "variations in seeds and the non-determinism of parallelism" [47]. For each P-FOL-IC3 query, we run it three times and report the average.

Table 4.4 shows user guidance needed for synthesizing ranking functions in terms of the number of: additional variables introduced, bounds adjusted, contradictions added, deltas adjusted, and tiers provided as input if tiered ranking functions were used. To provide guidance, users only need

Table 4.5: Runtime breakdown for verifying liveness properties of distributed protocols. Runtime is measured in seconds unless otherwise specified.

protocol name	static	bound	contradiction	delta	synthesis	validate	good-end	no-deadlock
ticket lock	0.02	48.3	127.5	0.57[s]	0.21	0.56	0.43[s]	117.4
HRB	0.06	0.64[s]	1.07[n]	0.65[s]	0.19	0.64	0.59[s]	86.2
alternating bit	0.05	560.8	1.46[n]	5.11	0.39	1.68	0.78[s]	0.83[s]
TLB shootdown	0.43	10.3[s]	6h[m]	24.0	1.75	17.3	0[d]	872.7
Paxos	0.11	0.91[s]	4929	0.67[s]	0.20	0.71	0[d]	533.3
multi-Paxos	0.13	1.02[s]	544.1	0.68[s]	0.21	0.71	0[d]	287.8
stoppable Paxos	0.18	1.29[s]	3113	0.79[s]	0.23	0.82	0[d]	1979.8
sliding window	0.14	2.5h[m]	1.5h[m]	12.6	1.54	4.55	0.57[s]	0.59[s]

s = Property is inductive by itself. Runtime is just for mypyvy to check it.
m = Inductive invariant found manually because P-FOL-IC3 does not support or times out. Time reported in hours.
n = No transition is contradictory. Runtime reported is just for preprocessing.
d = Degenerate case. Liveness property has trigger condition *true*, so "ending in good state" property of Eq. (4.12) becomes a tautology.

an understanding of the distributed protocol, not the internal workings of LVR or SMT solvers. Given an understanding of the protocol, providing hints is straightforward in all cases. Each protocol requires some user guidance in adding variables. The number of integer variables listed for each protocol in Table 4.4 includes those that require user guidance. In some cases, the explicit user guidance provided may not be the actual integer variable. For example, as discussed in Section 4.3.1, for the ticket lock protocol, an `active` variable is introduced via user guidance, from which `timesched(active)` is automatically introduced as an integer variable based on an integer function. Other than the ticket lock protocol, user guidance for all other protocols is required only for derived variables.

Most protocols also require some hints from the user to tighten bounds or deltas or specify tiers. For example, Table 4.4 shows that Paxos requires three corrections on deltas, all in the form of adding missing contradictions. One such contradiction is about transition `cast_vote`, which lets a node n receive a previously unseen proposed value v at round r, and cast a vote for it under some condition. P-FOL-IC3 does not recognize the transition is contradictory under condition $\exists V' : \texttt{value}.\ \texttt{proposal\_received}(n, r, V')$, but a human developer with knowledge of the Paxos protocol can quickly tell that this cannot happen — there cannot be two conflicting

proposed values v and V' in one round, because each round has only one proposer who can only propose one value. The other two missing contradictions are of similar nature. If the user identifies one, it is easy to correct all. If the user does not provide enough variables or tighten all necessary bounds or deltas, Z3 will output "no solution" when solving coefficients, indicating that something is missing to the user. For all protocols, all user guidance for deltas involved adding contradictions, rather than adjusting the upper or lower bound of deltas for a non-contradictory transition.

For all protocols, user guidance is not necessary to prove the absence of deadlocks. However, if desired, user guidance by excluding transitions or eliminating existential quantifiers can reduce the runtimes for protocols whose no-deadlock safety property is not itself already inductive. For example, including user guidance as discussed in Section 4.7, the runtime to prove the absence of deadlocks would be cut in half to only a minute for ticket lock and could be reduced by more than a factor of three for stoppable Paxos.

Table 4.5 shows that LVR reduces liveness to safety properties that can mostly be automatically verified. As discussed in Section 4.2, the liveness property is reduced to two safety properties—Eq. (4.12) (ending in good state) and Eq. (4.11) (no deadlock)—and a ranking function property. The ranking function property is decomposed into three more safety properties—proving bounds, proving contradictions, and proving deltas, before synthesizing the polynomial function. Therefore, there are a total of five safety properties per protocol, or 40 total for eight protocols. A dominant majority of safety properties (37 out of 40) are either always true, inductive by themselves thus automatically verified by mypyvy, or have an inductive invariant found by P-FOL-IC3 automatically. Only three safety properties could not be verified by P-FOL-IC3, so those inductive invariants were found manually in a few hours each.

Table 4.5 shows that the time to synthesize the ranking function, after verifying bounds and deltas, was a tiny fraction of the overall runtime. Ranking function synthesis just requires solving coefficients for a set of linear inequalities, which is where SMT solvers excel. Every Z3 query to synthesize a ranking function finished in less than a second. For sliding window and TLB shootdown, a ranking function is synthesized for each tier, so their total runtime slightly exceeds

one second. In most cases, Z3 synthesizes the exact same ranking function as a human would manually specify. Overall, the results and runtimes show that LVR was successful in significantly reducing the human effort to verify the liveness properties.

IronFleet previously manually verified liveness for two of the eight protocols, sliding window in IronKV and multi-Paxos in IronRSL, which required 2,093 and 7,869 LoC in Dafny, respectively. In comparison, using LVR, even if we include all lines in the protocol specification, then count each user-specified or corrected variable, bound, delta, and tier as one LoC, the manual effort to verify sliding window and multi-Paxos are 128 and 165 LoC, reducing the coding effort by 16 and 47 times, respectively. While part of the difference is due to Dafny versus mypyvy and the protocol encoding (IronFleet's multi-Paxos includes the local scheduler and IO queue used in its low-level implementation while our multi-Paxos is more abstracted), LVR still saves significant manual effort, with most reasoning and proofs automated using off-the-shelf tools and SMT solvers.

Dynamic Abstraction previously manually verified seven of the eight protocols, but required a large number of additional auxiliary relations and transitions to augment the protocols. For example, for Paxos, the user needs to introduce an additional 25 relations and 143 LoC in the augmented protocol, resulting in more than three times as many relations and more than twice the code as the original protocol. More importantly, Dynamic Abstraction requires 40 invariants for the augmented protocol to verify the liveness of Paxos. The sheer size of the augmented protocols and invariants required is beyond the reach of any existing invariant inference methods, so Dynamic Abstraction requires significant human effort to manually verify the safety property of the augmented protocol. In contrast, LVR leverages automated invariant inference and SMT solvers to verify liveness for Paxos with limited user guidance to add variables and adjust deltas. This is possible because its ranking function approach involves much fewer and shorter invariants, and it avoids requiring a more complex augmented protocol with many more relations.

Limitations LVR only targets liveness properties that say something eventually happens (see Eq. (4.8)), not that something happens within or after a specific time. For example, byzantine consensus protocols for blockchains may have liveness properties specifying that consensus is reached within a certain time [48]. LVR only synthesizes ranking functions as polynomials over integer variables, or combinations of such polynomials in lexicographic order, while theoretically, a ranking function can have any shape. There also exist polynomial ranking functions that cannot be synthesized from variables' deltas. For example, if a transition tx nondeterministically resets variables x and y ($x, y \in \mathbb{N}$), while only guaranteeing that $new(x)|_{tx}+new(y)|_{tx} = x+y-1$, then $x+y$ is a valid ranking function, although the deltas of both x and y are $(-\infty, \infty)$ from which no ranking function can be synthesized, unless the user explicitly specifies $x + y$ as an auxiliary variable. LVR inherits the limitations on expressiveness and theorem solving capabilities of mypyvy and the underlying Z3 and CVC4 solvers. As discussed in Section 4.5, SMT solvers may time out when reasoning about high-order polynomials, in which case LVR will not be able to validate the ranking function.

Chapter 5: Related Work

Manual verification of distributed protocols and systems. There has long been interest and effort in formally verifying distributed protocols and systems due to the importance of their correctness and security. Verdi [4] proves the correctness of a distributed system by first verifying the system under an ideal fault model, and then transferring the correctness to realistic model via a verified system transformer. Using Verdi, Wilcox et al. verify an implementation of Raft in around 5000 lines of proof using the Coq proof assistant. Ironfleet [5] verifies a multi-Paxos-based data replication system and a shared key-value store in Dafny using a combination of state-machine refinement and Hoare-logic-style deductive reasoning. Chapar [49] verifies the causal consistency for replicated key-value store implementations manually in Coq. Disel [50] propose a distributed Hoare-type logic that enables modular verification of the composition of protocols. All frameworks above rely on developers to write often thousands of lines of proofs, which limit their broader application and highlight the need for better automation techniques.

Automated safety verification of distributed protocols. To facilitate easier safety verification of distributed protocols, the Ivy language and tool [11] lets developers write a protocol as a state transition system, then provides an inductive invariant that can prove the desired safety property. Ivy then automatically checks the inductiveness of the invariants by calling the Z3 SMT solver [51] thus completes the safety proof.

However, finding an inductive invariant that proves the desired safety property is still a time-consuming task, which usually requires developers to write and fix their invariants based on counterexamples over many iterations. Various approaches have explored learning invariants automatically for distributed protocols. Dinv [52] identifies and tracks critical variables in distributed systems, and then infers likely correct invariants over these variables with Daikon [53], a data-

driven invariant learning tool. The invariants inferred using Dinv are not guaranteed to be valid and may not be inductive. Since the introduction of Ivy and its twin language mypyvy [54], a large number of tools have been developed to infer inductive invariants on top of Ivy or mypyvy, including UPDR [33], I4 [12], FOL-IC3 [13], SWISS [26], IC3PO [34], pdH [42], and P-FOL-IC3 [47]. Earlier systems, including UPDR, I4, and pdH, can only infer invariants with universal quantifiers ($\forall$) but not existential quantifiers ($\exists$).

Recent systems consider invariants with $\exists$-quantifiers. FOL-IC3 [13] generates an invariant candidate that can separate a positive and negative example set, and iteratively adds more examples until the invariant is correct. It has no theoretical guarantee of success. Its heavy use of SMT queries to validate as well as synthesize invariants makes it slow in practice, timing out on even protocols with $\forall$-only inductive invariants.

SWISS [26] iteratively strengthens an invariant by adding small inductive formulas until the invariant is strong enough to prove the safety property. It was the first tool to automatically verify Paxos. Its inefficiency in exploring the search space and inability to infer mutually inductive invariants make it fail on several protocols solved by alternative tools.

IC3PO [34, 35] applies model checking on a finite instance similar to I4, while adding support to generalize existentially quantified invariants. The model checker functions well on small instances, but frequently exhausts memory on complex protocols that require larger instances, as shown in Table 2.1.

P-FOL-IC3 [47] is concurrent work that extends FOL-IC3 by exploiting parallelism in formula search, introducing the invariant-friendly *k-Term Pseudo-DNF* to bound the search space, and randomly guessing some not-yet-inductive formulas to be eventually inductive, forcing their counterexamples to be excluded. P-FOL-IC3 has no theoretical guarantee and is less robust in practice due to its randomized nature; it failed in three out of five trials on stoppable Paxos, and two out of five trials on fast Paxos.

Automated liveness verification. Given both the theoretical undecidability and the practical struggle of SMT solvers on temporal reasoning, verifying liveness properties of distributed protocols is much more challenging than verifying safety. IronFleet verifies the liveness properties of its data replication system and key-value store, but with thousands of lines of manual proof and little automation.

Biere et al. introduces the classical liveness-to-safety reduction for finite-state systems, in which liveness is established by the acyclicity of system states in any execution [41]. Oded et al. proposes *Dynamic Abstraction* to generalize this reduction to infinite-state systems, which most distributed protocols can be written as [14, 15]. It reduces the liveness property to a fair cycle detection problem, which is representable as a safety property in an augmented protocol with many additional variables and transitions. The safety property can be proved with an inductive invariant, but the high complexity of the augmented protocol renders automated invariant inference tools useless. Manually writing invariants requires a thorough understanding of Dynamic Abstraction. LVR reduces liveness to safety, but on the original protocol using ranking functions, which is compatible with automated tools and is much easier to understand for users.

Ranking functions, also called measure functions [55], have long been used to prove the termination of loops and recursions in programs [37]. There is extensive work on automatically synthesizing linear [56, 57, 38, 58], polynomial [40], lexicographic [59], and multi-phase [39] ranking functions. There are also heuristic approaches to proving fair termination for multithreaded programs without explicit ranking functions [60, 61, 62]. However, these methods target only arithmetic, executable programs. The formulation and reasoning of distributed protocols usually involves richer structure, in particular the universal and/or existential quantification over nodes, messages, etc., in first-order logic, which is beyond the scope of previous work. LVR allows both first-order relations and nonlinear integer arithmetic in a protocol, although as discussed in Section 4.3.4, using integers in fairness assumptions only can best utilize the power of currently available automated invariant inference tools.

Automated invariant inference for other domains. Many automated invariant inference tools have been built for other domains, mostly on learning loop invariants to verify sequential programs. Traditional methods use symbolic reasoning to infer invariants [63, 64], while recently data-driven methods using execution traces and/or counterexamples have shown promise. Guess-and-check, Numinv, and G-CLN recast invariant inference as a curve-fitting task on execution traces, and learn loop invariants represented by polynomials of program variables [65, 66, 67, 68]. ICE-DT and LoopInvGen (PIE) apply decision tree learning and PAC learning on counterexamples and iteratively refine the invariants [69, 70, 71]. FreqHorn exploits both syntax and data in its inference tool and learns $\forall$-quantified array invariants [72, 73]. Recently, invariant inference has been used to prove properties on inductive algebraic data types [24, 74], integer linear dynamical systems [75], and deep neural networks [76]. None of these methods consider nondeterminism in concurrent or distributed settings, thus they cannot be directly applied to distributed protocols.

Conclusions and Future Work

In this thesis, I presented my research on automated verification of safety and liveness properties for distributed protocols. Specifically, I presented DistAI and DuoAI, invariant inference tools that automatically verify safety properties of distributed protocols, as well as LVR, a new framework and tool to verify liveness properties of distributed protocols in a mostly automated manner.

In DuoAI, I introduced the simulation-enumeration-refinement pipeline for invariant inference. I introduced the minimum implication graph, which captures the implication relations between possible invariants. Based on a key insight that if a stronger invariant is already in the candidate set, then a weaker (i.e., inferred) invariant must be redundant, DuoAI enumerates candidate invariants effectively by starting from the strongest formulas and iteratively try weaker forms following the minimum implication graph. Then DuoAI applies two refinement procedures in parallel, top-down refinement and bottom-up refinement, that interact with the SMT solver to refine the invariants until a correct inductive invariant is reached. The algorithm is guaranteed to find a correct inductive invariant if one exists in the search space. Our evaluation demonstrates that DuoAI is able to verify a large variety of disbributed protocols automatically and outperforms alternative methods both in the number of protocols verified and the time to verify them.

In LVR, I introduced a reduction from a liveness property to verifying four safety properties with the help of a ranking function. By reducing liveness to safety, LVR enables off-the-shelf invariant inference tools to prove the reduced safety properties automatically. To identify the ranking function, I introduced a new pipeline by statically analyzing the integer variables, their bounds, and

their deltas. The ranking function synthesis problem then becomes a coefficient solving problem which can be solved by an SMT solver. The developer only needs to provide limited guidance in the process. Our evaluation shows that using LVR, developers can verify liveness properties of disbributed protocols with little human effort.

In my future research career, I plan to continue my research on formally verifying distributed protocols and systems. I see three possible future directions starting from this thesis.

Modularization of protocol and proof. The methods presented in this thesis, along with most existing tools, treat the distributed protocol as a monolithic entity. However, a protocol intrinsically has structures such as phases or layers. By utilizing the structure of the protocol, one can decompose the safety and liveness proof into smaller, more manageable components. Furthermore, it is possible to develop and modularize the protocol and the proof simultaneously. This will also benefit the maintenance of the distributed protocol if it evolves over time.

Verifying blockchain consensus protocols. Blockchain consensus protocols are distributed protocols, but the probabilistic nature of many blockchain Byzantine Fault Tolerance (BFT) protocols make them difficult to verify with existing tools. For example, a blockchain transaction may satisfy some liveness constraint with $1 - \epsilon^k$ probability, rather than always be live like the current protocols verified with LVR. An extended reasoning frameworks and presumably new formal libraries for probability theory will be required to prove the probabilistic properties.

Verifying low-level distributed system implementation. This thesis focuses on formally verifying high-level distributed protocols. As I explained in Chapter 1, verifying the protocol is a necessary first step in verifying the implementation. But connecting the protocol with the implementation with a refinement proof is not an easy task as well. It generally requires huge engineering effort and becomes infeasible for large commercial distributed systems. I consider it very interesting to see how we can combine formal verification with more scalable techniques such as property-based testing and model checking to retain a high confidence of correctness without a

complete proof. This direction is partially inspired by the verification of the key-value storage in AWS S3 [8].

References

[1] K. R. Apt, F. S. De Boer, and E.-R. Olderog, *Verification of sequential and concurrent programs*. Springer Science & Business Media, 2009.

[2] *Cloud adoption statistics*, `https://www.zippia.com/advice/cloud-adoption-statistics/`.

[3] A. Palmer, "Dead roombas, stranded packages and delayed exams: How the aws outage wreaked havoc across the U.S.," *CNBC*, 2021 (https://www.cnbc.com/2021/12/09/how-the-aws-outage-wreaked-havoc-across-the-us.html).

[4] J. R. Wilcox *et al.*, "Verdi: A framework for implementing and formally verifying distributed systems," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*, Jun. 2015, pp. 357–368.

[5] C. Hawblitzel *et al.*, "IronFleet: Proving practical distributed systems correct," in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*, Oct. 2015, pp. 1–17.

[6] M. Taube *et al.*, "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 (PLDI '18)*, Jun. 2018, pp. 662–677.

[7] T. Hance, A. Lattuada, C. Hawblitzel, J. Howell, R. Johnson, and B. Parno, "Storage systems are distributed systems (so verify them that way!)" In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, 2020, pp. 99–115.

[8] J. Bornholt *et al.*, "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*, ser. SOSP '21, Virtual Event, Germany, 2021, 836–850.

[9] J. Chen, S. Gorbunov, S. Micali, and G. Vlachos, "Algorand agreement: Super fast and partition resilient byzantine agreement," *Cryptology ePrint Archive*, 2018.

[10] T. Rocket, M. Yin, K. Sekniqi, R. van Renesse, and E. G. Sirer, "Scalable and probabilistic leaderless bft consensus through metastability," *arXiv preprint arXiv:1906.08936*, 2019.

[11] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham, "Ivy: Safety verification by interactive generalization," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*, Jun. 2016, pp. 614–630.

[12] H. Ma, A. Goel, J.-B. Jeannin, M. Kapritsos, B. Kasikci, and K. A. Sakallah, "I4: Incremental inference of inductive invariants for verification of distributed protocols," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, Oct. 2019, pp. 370–384.

[13] J. R. Koenig, O. Padon, N. Immerman, and A. Aiken, "First-order quantified separators," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*, Sep. 2020, 703–717.

[14] O. Padon, J. Hoenicke, G. Losa, A. Podelski, M. Sagiv, and S. Shoham, "Reducing liveness to safety in first-order logic," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–33, 2017.

[15] O. Padon, J. Hoenicke, K. L. McMillan, A. Podelski, M. Sagiv, and S. Shoham, "Temporal prophecy for proving temporal properties of infinite-state systems," *Formal Methods in System Design*, vol. 57, no. 2, pp. 246–269, 2021.

[16] J. Yao, R. Tao, R. Gu, J. Nieh, S. Jana, and G. Ryan, "DistAI: Data-driven automated invariant learning for distributed protocols," in *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI '21)*, Jul. 2021, pp. 405–421.

[17] J. Yao, R. Tao, R. Gu, and J. Nieh, "DuoAI: Fast, automated inference of inductive invariants for verifying distributed protocols," in *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*, 2022, pp. 485–501.

[18] J. Yao, R. Tao, R. Gu, and J. Nieh, "Mostly automated verification of liveness properties for distributed protocols with ranking functions," *Proceedings of the ACM on Programming Languages*, vol. 8, no. POPL, pp. 1028–1059, 2024.

[19] G. Ricart and A. K. Agrawala, "An optimal algorithm for mutual exclusion in computer networks," *Communications of the ACM*, vol. 24, no. 1, pp. 9–17, 1981.

[20] V.-T. Pham, M. Böhme, and A. Roychoudhury, "AFLNET: A greybox fuzzer for network protocols," in *Proceedings of the IEEE 13th International Conference on Software Testing, Validation and Verification (ICST '20)*, Oct. 2020, pp. 460–465.

[21] J. de Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations," in *Proceedings of the 24th USENIX Security Symposium (USENIX Security '15)*, Aug. 2015, pp. 193–206, ISBN: 978-1-939133-11-3.

[22] S. Sivakorn, G. Argyros, K. Pei, A. D. Keromytis, and S. Jana, "HVLearn: Automated black-box analysis of hostname verification in SSL/TLS implementations," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (IEEE S&P '17)*, May 2017, pp. 521–538.

[23] H. Zhu, S. Magill, and S. Jagannathan, "A data-driven CHC solver," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*, Jun. 2018, pp. 707–721.

[24] A. Miltner, S. Padhi, T. Millstein, and D. Walker, "Data-driven inference of representation invariants," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*, Jun. 2020, pp. 1–15.

[25] Y. M. Feldman, J. R. Wilcox, S. Shoham, and M. Sagiv, "Inferring inductive invariants from phase structures," in *Proceedings of the 31st International Conference on Computer Aided Verification (CAV '19)*, Jul. 2019, pp. 405–425.

[26] T. Hance, M. Heule, R. Martins, and B. Parno, "Finding invariants of distributed systems: It's a small (enough) world after all," in *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI '21)*, Apr. 2021, pp. 115–131.

[27] L. LAMPORT, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

[28] L. Lamport *et al.*, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[29] O. Padon, G. Losa, M. Sagiv, and S. Shoham, "Paxos made EPR: Decidable reasoning about distributed protocols," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, Oct. 2017.

[30] K.-C. Li, X. Chen, H. Jiang, and E. Bertino, *Essentials of Blockchain Technology*. CRC Press, 2019.

[31] D. Monk, *Mathematical Logic*. Springer, Oct. 1976.

[32] *Decidability in Ivy*, `http://microsoft.github.io/ivy/decidability.html`.

[33] A. Karbyshev, N. Bjørner, S. Itzhaky, N. Rinetzky, and S. Shoham, "Property-directed inference of universal invariants or proving their absence," *Journal of the ACM*, vol. 64, no. 1, Mar. 2017.

[34] A. Goel and K. Sakallah, "On symmetry and quantification: A new approach to verify distributed protocols," in *Proceedings of the 13th NASA Formal Methods Symposium (NFM '21)*, May 2021, pp. 131–150.

[35] A. Goel and K. A. Sakallah, "Towards an automatic proof of Lamport's Paxos," in *Proceedings of the 21st Conference on Formal Methods in Computer Aided Design (FMCAD '21)*, Oct. 2021, pp. 112–122.

[36] O. Padon, *Source file of the ticket lock protocol in ivy*, `https://github.com/kenmcmil/ivy/blob/master/examples/liveness/ticket_nested.ivy`, 2021.

[37] B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Y. Vardi, "Proving that programs eventually do something good," in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*, 2007, 265–276.

[38] M. Heizmann and J. Leike, "Ranking templates for linear loops," *Logical Methods in Computer Science*, vol. 11, 2015.

[39] A. M. Ben-Amram and S. Genaim, "On multiphase-linear ranking functions," in *Proceedings of the 29th International Conference on Computer Aided Verification (CAV '17)*, 2017, pp. 601–620.

[40] E. Neumann, J. Ouaknine, and J. Worrell, "On ranking function synthesis and termination for polynomial programs," in *Proceedings of the 31st International Conference on Concurrency Theory (CONCUR '20)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[41] A. Biere, C. Artho, and V. Schuppan, "Liveness checking as safety checking," *Electronic Notes in Theoretical Computer Science*, vol. 66, no. 2, pp. 160–177, 2002.

[42] O. Padon, J. R. Wilcox, J. R. Koenig, K. L. McMillan, and A. Aiken, "Induction duality: Primal-dual search for invariants," *Proceedings of the ACM on Programming Languages*, vol. 6, no. POPL, Jan. 2022.

[43] G. Tel, *Introduction to distributed algorithms*. Cambridge university press, 2000.

[44] J. Hoenicke, R. Majumdar, and A. Podelski, "Thread modularity at many levels: A pearl in compositional verification," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*, 2017, pp. 473–485.

[45] D. Malkhi, L. Lamport, and L. Zhou, "Stoppable paxos," Tech. Rep. MSR-TR-2008-192, 2008.

[46] C. Hawblitzel *et al.*, *The ironfleet repository*, `https://github.com/microsoft/Ironclad/tree/main/ironfleet`, 2015.

[47] J. R. Koenig, O. Padon, S. Shoham, and A. Aiken, "Inferring invariants with quantifier alternations: Taming the search space explosion," in *Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '22)*, Apr. 2022, pp. 338–356.

[48] B. Y. Chan and E. Shi, "Streamlet: Textbook streamlined blockchains," in *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, ser. AFT '20, 2020, 1–11, ISBN: 9781450381390.

[49] M. Lesani, C. J. Bell, and A. Chlipala, "Chapar: Certified causally consistent distributed key-value stores," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*, Jan. 2016, pp. 357–370.

[50] I. Sergey, J. R. Wilcox, and Z. Tatlock, "Programming and proving with distributed protocols," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, Jan. 2018.

[51] *Z3 SMT solver*, `https://github.com/Z3Prover/z3`.

[52] S. Grant, H. Cech, and I. Beschastnikh, "Inferring and asserting distributed system invariants," in *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*, May 2018, pp. 1149–1159.

[53] M. D. Ernst *et al.*, "The Daikon system for dynamic detection of likely invariants," *Science of computer programming*, vol. 69, no. 1-3, pp. 35–45, Dec. 2007.

[54] J. Wilcox, O. Padon, Y. Feldman, and other contributors, *The mypyvy language*, `https://github.com/wilcoxjay/mypyvy`, 2018.

[55] M. Kaufmann, P. Manolios, and J. S. Moore, *Computer-aided reasoning: ACL2 case studies*. Springer Science & Business Media, 2013, vol. 4.

[56] M. A. Colón and H. B. Sipma, "Practical methods for proving program termination," in *Proceedings of 14th International Conference on Computer Aided Verification (CAV '02)*, 2002, pp. 442–454.

[57] A. Podelski and A. Rybalchenko, "A complete method for the synthesis of linear ranking functions," in *Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '04)*, 2004, pp. 239–251.

[58] L. Gonnord, D. Monniaux, and G. Radanne, "Synthesis of ranking functions using extremal counterexamples," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*, 2015, pp. 608–618.

[59] B. Cook, A. See, and F. Zuleger, "Ramsey vs. lexicographic termination proving," in *Proceedings of the 19th international conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '13)*, 2013, pp. 47–61.

[60] B. Cook, A. Podelski, and A. Rybalchenko, "Proving thread termination," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, 2007, 320–330.

[61] A. Farzan, Z. Kincaid, and A. Podelski, "Proving liveness of parameterized programs," in *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '16)*, 2016, 185–196.

[62] B. Kragl, C. Enea, T. A. Henzinger, S. O. Mutluergil, and S. Qadeer, "Inductive sequentialization of asynchronous programs," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*, 2020, pp. 227–242.

[63] S. Kong, Y. Jung, C. David, B.-Y. Wang, and K. Yi, "Automatically inferring quantified loop invariants by algorithmic learning from simple templates," in *Proceedings of the 8th Asian Symposium on Programming Languages and Systems (APLAS '10)*, Nov. 2010, pp. 328–343.

[64] P. Garg, C. Löding, P Madhusudan, and D. Neider, "Learning universally quantified invariants of linear data structures," in *Proceedings of the 25th International Conference on Computer Aided Verification (CAV '13)*, Jul. 2013, pp. 813–829.

[65] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori, "A data driven approach for algebraic loop invariants," in *Proceedings of the 22nd European Symposium on Programming (ESOP '13)*, Mar. 2013, pp. 574–592.

[66] T. Nguyen, T. Antonopoulos, A. Ruef, and M. Hicks, "Counterexample-guided approach to finding numerical invariants," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (FSE '17)*, Aug. 2017, pp. 605–615.

[67] G. Ryan, J. Wong, J. Yao, R. Gu, and S. Jana, "CLN2INV: Learning loop invariants with continuous logic networks," in *Proceedings of 8th International Conference on Learning Representations (ICLR '20)*, Mar. 2020.

[68] J. Yao, G. Ryan, J. Wong, S. Jana, and R. Gu, "Learning nonlinear loop invariants with gated continuous logic networks," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*, Jun. 2020, pp. 106–120.

[69] P. Garg, D. Neider, P. Madhusudan, and D. Roth, "Learning invariants using decision trees and implication counterexamples," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*, Jan. 2016, 499–512.

[70] S. Padhi, R. Sharma, and T. Millstein, "Data-driven precondition inference with learned features," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*, Jun. 2016, 42–56.

[71] S. Padhi, R. Sharma, and T. Millstein, "Loopinvgen: A loop invariant generator based on precondition inference," *arXiv preprint arXiv:1707.02029v4*, Oct. 2019.

[72] G. Fedyukovich, S. Prabhu, K. Madhukar, and A. Gupta, "Solving constrained Horn clauses using syntax and data," in *Proceedings of the 18th Conference on Formal Methods in Computer Aided Design (FMCAD '18)*, Oct. 2018, pp. 1–9.

[73] G. Fedyukovich, S. Prabhu, K. Madhukar, and A. Gupta, "Quantified invariants via syntax-guided synthesis," in *Proceedings of the 31st International Conference on Computer Aided Verification (CAV '19)*, Jul. 2019, pp. 259–277.

[74] Y. Kostyukov, D. Mordvinov, and G. Fedyukovich, "Beyond the elementary representations of program invariants over algebraic data types," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21)*, Virtual, Canada, 2021, 451–465, ISBN: 9781450383912.

[75] E. Lefaucheux, J. Ouaknine, D. Purser, and J. Worrell, "Porous invariants," in *Proceedings of 33rd International Conference on Computer Aided Verification (CAV '21)*, Jul. 2021, pp. 172–194.

[76] G. Amir, M. Schapira, and G. Katz, "Towards scalable verification of deep reinforcement learning," in *Proceedings of the 21st Conference on Formal Methods in Computer Aided Design (FMCAD '21)*, Oct. 2021, pp. 193–203.