Automating the Verification of Distributed Systems

by

Haojun Ma

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2022

Doctoral Committee:

Assistant Professor Manos Kapritsos, Chair
Assistant Professor Jean-Baptiste Jeannin
Dr. Jacob Lorch, Microsoft Research
Professor Westley Weimer

Haojun Ma

mahaojun@umich.edu

ORCID iD: 0000-0002-2155-4809

This dissertation is dedicated to my family and friends

who gave me all the support in the academic path.

ACKNOWLEDGMENTS

Academia is never an easy path. Only the most talented people and the most creative projects can be rewarded. During these five years, I have been through countless frustration, rejection, and self-doubt. There are many people who helped me during this journey.

My advisor, Manos Kapritsos, is the best mentor I have ever met. Manos not only showed his immense knowledge to help my research skills and philosophy, but also shared his experience with my life choices. Without his continuous support, I would never achieve this result within just five years. I'm grateful for my time working with him. Also, thanks for his *Harry Potter*.

I am extremely grateful to the rest of my thesis committee: Prof. Jean-Baptiste Jeannin, Dr. Jacob Lorch, and Prof. Westley Weimer. They provided valuable feedback not only on the writing of this final dissertation but also during my research work. Their service significantly improved this dissertation.

During these five years, I got much help, comfort, and knowledge from other students. I'd like to thank my collaborators, Aman Goel, Hammad Ahmad, and my labmate, Eli Goldweber. Their brilliant ideas and great work made our projects, I4 and Sift, excellent. Luoxi Meng and Xingran Shen helped on implementing systems in Cruiser. Also, my other labmates, Boyu Tian (although he decided to leave), Tony Zhang, and Armin Vakil are all great researchers and friends. I really enjoy every discussion and brainstorming we had.

I'd like to thank my friends, who have been a great treasure in my life. Jie You, Rui Liu, Yiwen Zhang, and Haizhong Zheng are all important people during this five-year journey. I'll always remember their flashes in our discussions, accompany before deadlines, and all the fun we had in the hard times.

My parents are the *unmoved mover* to drive me towards this path. I always remember how they tell me about trusting myself and aiming for a higher goal. Although this sometimes could be challenging and stressful, they made me the person I am.

Special thanks to my wife, Yike Liu, and our little dog, Paula. You really changed my life.

Thanks to myself, who picked the challenging path, and made it to the end.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE

LIST OF TABLES

TABLE

ABSTRACT

Designing and implementing distributed systems correctly is a very challenging task. Traditionally, people use tests to find and resolve bugs in their systems. But this approach doesn't scale to complex large systems, as there are too many possible interleaving of components. As a result, we still leave bugs in our systems.

To build systems with strong correctness guarantees, formal verification has been successfully used to prove the correctness of distributed systems. Although some approaches successfully applied formal verification to distributed systems and proved the implementation correct, writing a proof for a complex distributed system still requires an ultimate understanding of both the system and the formal method. And these approaches take years of manual effort to write a proof. Thus, formal verification is still not ready for real-world distributed applications.

In this dissertation, I aim to make formal verification more practical by reducing the manual effort required in writing a proof. I'll first show how I can leverage the power of model checking to automatically verify a distributed protocol in I4. After that, we propose the use of encapsulation in Sift to combine the automation from I4 with refinement to scale automation to verification of real distributed implementations. Finally, to make our verified system more practical, I move my focus to high-performance implementations and propose Cruiser to automatically generate such an implementation and its refinement proof to simplify the effort for developers.

CHAPTER I

Introduction

For more than 50 years, the systems community and industry have been relying on testing to increase their confidence in the correctness of software [CDE08, GKS05, SMA05, BEL75, Kin76]. As the availability demands start to increase, however, testing can fall short, since it is impractical to exhaustively test a program. Consequently, testing is bound to occasionally miss a bug, which may manifest during production, resulting in loss of availability, revenue, and company reputation [Tea08, The04, CVE17, YCSS12]. This has led many researchers and companies to look for alternative ways to develop software with strong correctness guarantees.

Thankfully, the increasing need for availability has been paralleled by an increase in the capabilities of formal verification techniques. Over the last decade, a number of techniques and tools have been built to formally verify the correctness of complex systems software [KEH+09, HHL+14, HHK+15, CZC+15, CCK+17, RUIMI16, NSZ+17]. The promise of formal verification—to eliminate all bugs by construction—is particularly attractive for distributed systems, which are notoriously hard to design and implement correctly.

Unfortunately, existing approaches to formally verifying complex systems have a major scalability bottleneck, rendering them not ready for real-world applications. Some techniques use interactive and automated theorem provers [dt, NWP02, Lei10, PMP+16, SCF+11] to dispatch a number of proof obligations, thereby simplifying the proof. But this process still heavily relies on human intuition and understanding of the system. As a result, the most powerful techniques, such as IronFleet [HHK+15] and Verdi [WWP+15], rely on *refinement* proofs [AL91, GL00, Lam94] to reason about complex systems and verify real implementations. Alas, the power of those techniques comes at a high cost: performing these refinement proofs manually requires large amounts of manual effort. In an attempt to reduce the manual verification effort, the Ivy tool [PMP+16] proposes to express distributed protocols using decidable—and thus simpler to verify—reasoning [PdMB10]. The Ivy tool achieves remarkable automation, but still requires significant human effort to complete the proof.

In this dissertation, we target the automation in the verification of distributed systems. We first present how to leverage the power of model checking to automatically infer the inductive

invariant for monolithic protocols with I4. After that, we combine the automation from monolithic provers like I4 with refinement to scale such automation to multi-layer systems, and even real implementation. Finally, we show how to push the boundary of automated verification to more realistic high-performance systems.

1.1 Finding Inductive Invariant

At the heart of all proofs of distributed systems, lies a critical process that existing approaches do not automate: finding an *inductive invariant*. An invariant of a state transition system is a predicate on the states that holds for all states that are reachable from the initial state(s); it is any set of states that *includes* all reachable states. An invariant is inductive if it is *closed* under the transition relation, i.e., the next state of every state in this set is also a member of this set. Proving a safety property P—i.e., showing that P *always* holds for any execution started from the initial state(s)—amounts to showing either a) that P is an inductive invariant, or b) that P is an invariant that can be *strengthened* to become inductive.

Inductive invariants are tightly linked to the correctness of distributed systems. Proving the correctness of such a system is typically split into two parts: proving the correctness of the distributed protocol by finding an inductive invariant; and showing that the implementation follows the protocol, which typically does not require inductive reasoning [HHK+15].

While there are simple centralized programs for which inductive invariants are not required, we have found that all but the most trivial distributed protocols require an inductive invariant in order to prove a reasonable safety property. In non-trivial verification cases, the required safety property P *is an invariant* but not an inductive one, and completing the verification proof entails the derivation of additional invariants that are used to constrain P until it becomes inductive. These additional invariants are viewed as *strengthening assertions* that remove those parts of P that are not closed under the system's transition relation. In most cases, finding these assertions is the hardest part of the proof. Most crucially, even for simple systems, these assertions can be very complicated, and as system complexity increases, these assertions—and the resulting inductive invariants—grow proportionally more complex.

As a result, finding an inductive invariant for a distributed protocol is usually the hardest part of the proof. During the IronFleet project [HHK+15], the authors spent about one week trying to identify the inductive invariant for a very simple distributed protocol, where nodes pass around a token in a ring. After careful thought and long discussions, they identified an invariant that included 5 separate clauses which, when combined, were sufficient to support an inductive proof. It is perhaps not surprising then, that when they attempted to find the inductive invariant of a real-world system—i.e., the Paxos protocol [Lam98]—the required effort was on the order of months.

Proving the correctness of protocols has a lot of practical value. Major companies are using formal verification to prove their designs and protocols correct, even when their implementations are unverified. For example, Amazon uses TLA+ to verify the correctness of its system designs [NRZ+14]. Similarly, Microsoft uses the IronFleet methodology to verify some of its protocols without going down to the implementation level. Even when the implementation must be verified, IronFleet showed that protocol-level verification is an integral part of the process.

In Chapter II, we introduce an automated way of finding inductive invariants for distributed protocols, one that does not rely on human intuition. The core insight that drives this new approach is that the basic elements of these invariants are independent of the size of the system, and that they can therefore be inferred from small, finite instances. For example, the inductive invariant of the token ring mentioned above states that only the last node in a sequence of token owners can hold the token. This must be true regardless of the number of nodes on the ring. Similarly, the inductive invariant of Paxos states that any two quorums of acceptors must overlap in at least one node; this must hold for any number of participating acceptors. This work has been published in HotOS'19 [MGJ+19b] and SOSP'19 [MGJ+19a].

1.2 Scaling Automation to Refinement

Approaches like I4 leverage model checking and SMT solvers to automate the most challenging part of proving the correctness of distributed protocols: finding an *inductive invariant*. Alas, this automation comes at the expense of expressiveness and applicability, because tools like I4 were designed to prove properties of *monolithic* protocols which consist of a single layer. As such, they cannot prove refinement.

Refinement [AL91, GL00, Lam94], however, is an essential concept in proving the correctness of real, complex systems. It allows us to prove the correctness of a system by showing that it is equivalent to a simpler, more abstract version of that system. The power of refinement comes in many forms:

Concise specification As Lamport has argued [Lam02] and as IronFleet demonstrated, specifications should be written as simple, abstract state machines. Consider the specification of a Paxos-based State Machine Replication in IronFleet, where the goal is to prove that the entire service is linearizable. Expressing linearizability as a set of properties on the requests and responses is daunting and will likely yield a complex specification. Using refinement, the task is simple: just show that the entire service is equivalent to a single machine executing requests one at a time. Similarly, the sharded key-value store in IronFleet was simply proven equivalent to an abstract, logically centralized key-value store; i.e., a map.

Scaling to complex systems As IronFleet and Verdi demonstrated, the key to dealing with the

complexity of a real system is to take a modular approach: split the proof into multiple layers and show that each layer refines the one above it. This is especially true when verifying actual implementations, as these tend to be much more complex than abstract protocols. In the absence of refinement, we are left with the task of reasoning about a single, monolithic system, whose complexity now becomes a limiting factor for both manual and automated approaches.

Dealing with undecidability Even when one only cares about proving the correctness of the protocol, and not of the implementation, being unable to split a monolithic system into multiple layers can be a showstopper for automation. As Padon et al. demonstrated [PLSS17], some protocols may be undecidable by construction and thus not amenable to the automation of I4 and IC3PO. In these cases, one can use refinement to split the protocol into two layers, each of which is separately decidable [TLM+18].

We aim to get the best of what are currently two distinct worlds: the *power of refinement* (i.e. IronFleet-style proofs) but with only *a fraction of the manual effort* (i.e. using the automation of monolithic provers like I4 [MGJ+19b, MGJ+19a], IC3PO [GS21], SWISS [HHMP21], and DistAI [YTG+21]). This combination allows us to not only achieve simple, concise specifications, but also to scale our proofs to more complicated distributed protocols, and even to distributed implementations.

In Chapter III, we introduce Sift to achieve this goal. Sift is a two-tier methodology that combines automated verification with a small amount of manual effort to push the boundary on the kinds of systems that can benefit from proof automation. Sift has been published in ATC'22 [MAG+22].

1.3 Generating Efficient Implementation

Even with the scalability of Sift, we found that it still raises many restrictions for the implementation. As a result, the developer can hardly optimize the executable code and achieve better performance.

However, in real-world systems, performance is critical. For Amazon, every 100ms latency penalty causes a 1% sales loss [LIN]. It is the same case for Google: experiments demonstrate that increasing web search latency 100 to 400 ms reduces the daily number of searches per user by 0.2% to 0.6% [Bru09]. Throughput, on the other hand, helps to serve more users within given resources.

From our study of the performance gap between Ivy and Dafny, we found that the decidability that Ivy pursues sometimes conflicts with performance optimizations. For example, Ivy models all variables static in the heap, which makes it impossible to implement some dynamic data structures such as a tree of a hash map.

For all previously existing approaches, unfortunately, the developer needs to choose either implementing their whole system within Ivy and decidable logic, with strict restrictions about how

the implementation can be generated; or adopt the freedom and incompleteness of theorem provers like Dafny, and implement all details of the proof manually.

In Chapter IV, we work on more practical systems. We found that if the developer can model their protocol in a specific way, even in undecidable theorem provers, it is possible to generate a high-performance implementation. We introduce Cruiser, a new approach to guide the developer to implement a function-style protocol layer proof, and then generate a heap-based efficient implementation along with its refinement proof.

CHAPTER II

Verifying Distributed Protocols

As we have seen in Chapter I, despite the complexity in a protocol, distributed protocols exhibit a lot of regularity. The behavior of a protocol doesn't fundamentally change from a small instance to a large scale. As a result, the inductive invariants can therefore be inferred from small, finite instances.

We leverage this insight to automate the process of identifying inductive invariants for distributed systems. We propose a new proof technique and tool called *Incremental Inference of Inductive Invariants* (I4). The key idea behind I4 is to first identify the inductive invariant of a small instance of the system and then to use that *instance-specific* invariant to infer a generalized invariant that holds for all instances.

The key requirement for I4 is automatically identifying the inductive invariant for a small instance of the system. To that end, I4 uses decades of progress made by the model checking community. Model checking is typically considered inadequate [HHK+15, BDH07, WWP+15] for proving the correctness of real-world distributed systems, as the state space it must explore grows exponentially. While this state space explosion certainly happens in generic distributed systems, model checking is powerful enough to prove the correctness of small, finite instances. In particular, the IC3 model checker [Bra11] has shown that it is possible, given a finite and moderately complex system instance, to either compute an inductive invariant which implies the desired safety property; or to produce a counterexample if the system is not correct. I4 harvests this power as a means to an end: not to prove the correctness of those small instances, but to infer an inductive invariant that holds for all instances.

Overall, we make the following contributions:

- We propose a new approach to the verification of distributed protocols. Instead of manually and painstakingly identifying an inductive invariant, we can draw inspiration from the inductive invariants of small, finite instances of these protocols.

- We propose I4, an algorithm that implements this new approach by combining the power of model checking and automated theorem provers. I4 creates a finite instance of a distributed

protocol and leverages model checking to find an inductive invariant specific to that instance. I4 then uses this invariant as a starting point to identify a *generalized* inductive invariant that holds for all instances of the protocol.

- We evaluate the effectiveness of I4 and find that it can prove the correctness of a number of interesting distributed protocols—e.g., Two-Phase Commit, Chord, Transaction Chains—with little to no manual effort, and without us providing any insight into the subtleties of these protocols (in fact, we didn't fully understand how some of these protocols work).

The details of this methodology are explained in this Chapter as follows: Section 2.1 shows a concrete example of how the inductive invariants of small, finite instances can help us discover a generalized inductive invariant that holds for all instances. Section 2.2 gives an overview of the I4 approach, while the next two sections present the details of the two main components of the approach: generating an inductive invariant for a finite instance (Section 2.3) and generalizing that invariant to one that holds for all instances (Section 2.4). Section 2.5 presents our experiences applying I4 to real distributed protocols and Section 2.6 discusses the limitations of our approach and some open research problems. Section 2.7 concludes this chapter.

2.1 A New Perspective

Verification of distributed systems has so far relied heavily on human intellect: to prove the correctness of a distributed system, one must first understand it well. One can then attempt to write a formal proof of correctness, by demonstrating the existence of an *inductive invariant*. Inductive invariants are typically much more complex than the desired safety property and finding them requires an intimate understanding of the internal mechanics of the protocol.

In an attempt to facilitate this process, Padon et al. recently developed Ivy [PMP+16]. Ivy takes as input a protocol description and a safety property, and guides the user, through a series of interactive steps and visual counterexamples-to-induction, to discover an inductive invariant. Ivy still fundamentally relies on human intellect for identifying such an inductive invariant; but once that invariant is found, Ivy automatically checks that it is indeed inductive.

In this chapter, we propose a new approach for finding such inductive invariants that does not rely on a deep understanding of the system. The key insight of our approach is that the behavior of most distributed protocols does not fundamentally change as their size increases. This initial insight led us to ask the question: *is it possible to infer the inductive invariant of a distributed protocol by observing a small instance of the protocol?*

Our experience so far indicates that the answer to this question is typically "yes". Distributed protocols exhibit a high degree of regularity: what is true for a small instance of four nodes is also

Algorithm 1 Lock Server

```
after init {
    semaphore(Y) := true;
    link(X, Y) := false;
}
action connect(c: client, s: server) = {
    require semaphore(s);
    link(c, s) := true;
    semaphore(s) := false;
}
action disconnect(c: client, s: server) = {
    require link(c, s);
    link(c, s) := false;
    semaphore(s) := true;
}
```

true for a large instance of 1000 nodes. We will demonstrate this regularity using a simple lock server protocol [WWP+15, PMP+16] with N servers and M clients.

Case study: lock server

Algorithm 1 shows the lock server protocol description, written in Ivy. In this protocol, every server S maintains a lock and the server's state is a boolean *semaphore(S)* indicating whether it currently holds its lock. Every client-server pair is associated with a boolean *link(C, S)* which denotes whether client C holds the lock of server S. Initially every server holds its own lock and all client-server links are set to false.

There are two possible actions in this protocol. A client may send a lock request to a server and acquire that server's lock if the server currently holds its lock. A client may also release a lock, handing it back to the server. The safety property of this protocol is simple: "no two clients can have a link to (i.e., hold the lock of) the same server at the same time":

$$\forall C_1, C_2, S.\ link(C_1, S) \wedge link(C_2, S) \implies (C_1 = C_2)$$

Let us now consider the inductive invariants of small instances of this protocol. Section 2.3 describes how we can automatically generate these instances and their inductive invariants. For now, we are only interested in what these inductive invariants are.

The smallest non-trivial instance consists of one server and two clients. The inductive invariant to prove the correctness of that instance is:

$$\begin{array}{|ll|}\hline \neg(semaphore(S0) \wedge link(C0, S0)) & \wedge \\ \neg(semaphore(S0) \wedge link(C1, S0)) & \wedge \\ \neg(link(C0, S0) \wedge link(C1, S0)) & \textit{(Safety Property)} \\ \hline \end{array}$$

If we consider a larger instance with more clients—say four—the inductive invariant becomes bigger, but remains essentially the same:

$$\begin{array}{|ll|}\hline \neg(semaphore(S0) \wedge link(C0, S0)) & \wedge \\ \neg(semaphore(S0) \wedge link(C1, S0)) & \wedge \\ \neg(semaphore(S0) \wedge link(C2, S0)) & \wedge \\ \neg(semaphore(S0) \wedge link(C3, S0)) & \wedge \\ \textit{Safety Property} & \\ \hline \end{array}$$

When we further consider instances with multiple servers, the invariant again increases in size *but not in complexity*. For example, the inductive invariant for an instance with two servers and four clients is:

$$\begin{array}{|ll|}\hline \neg(semaphore(S0) \wedge link(C0, S0)) & \wedge \\ \neg(semaphore(S0) \wedge link(C1, S0)) & \wedge \\ \neg(semaphore(S0) \wedge link(C2, S0)) & \wedge \\ \neg(semaphore(S0) \wedge link(C3, S0)) & \wedge \\ \neg(semaphore(S1) \wedge link(C0, S1)) & \wedge \\ \neg(semaphore(S1) \wedge link(C1, S1)) & \wedge \\ \neg(semaphore(S1) \wedge link(C2, S1)) & \wedge \\ \neg(semaphore(S1) \wedge link(C3, S1)) & \wedge \\ \textit{Safety Property} & \\ \hline \end{array}$$

Given the above instances, it does not take much ingenuity to manually come up with an inductive invariant that works for *all* instances of this protocol:

$$\begin{array}{|ll|}\hline \forall C, S. \neg(semaphore(S) \wedge link(C, S)) & \wedge \\ \textit{Safety Property} & \\ \hline \end{array}$$

Of course, this protocol is rather simple and its inductive invariant is quite small. But the principle still applies to more complicated protocols: we can use the inductive invariant of a small instance to infer a *generalized* inductive invariant that works for all instances of the protocol.

2.2 Overview of I4

The ultimate goal of I4 is simple: given a protocol description and a safety property, it tries to prove the correctness of the protocol by identifying an inductive invariant that implies the safety

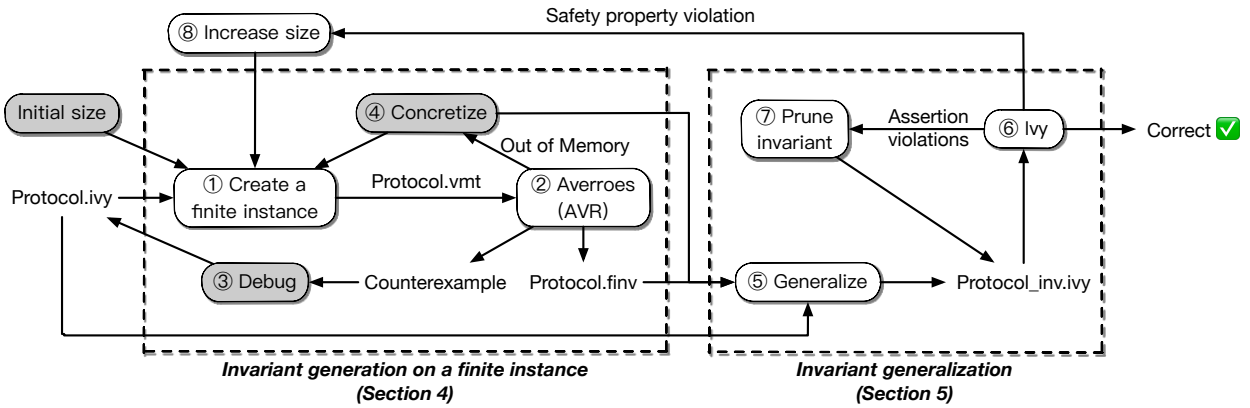

Figure II.1: Flow of I4. White boxes are fully automated, while gray boxes denote manual effort.

property. As we mentioned in Section 2.1, the hardest part of any correctness proof is *finding* such an invariant. Once we have a candidate invariant for a finite instance, we generalize it and use the Ivy tool [PMP+16] to check whether it is inductive for an arbitrarily-sized instance.

Figure II.1 shows an overview of the I4 flow. Given a protocol description—written in Ivy—and an initial size, I4 first generates a finite instance of that protocol with a given initial size. For example, given the lock server protocol of the previous section and an initial size of $(1, 2)$, I4 will generate a finite instance of the protocol with one server and two clients. It then uses the Averroes model checker [GS19b] to either generate an inductive invariant that proves the correctness of the protocol for that particular instance, or produce a counterexample demonstrating how the protocol can be violated and which can be used to debug the protocol [Eng12, GGK+18].

If the protocol is too complex, the model checker may fail to produce an answer within a reasonable amount of time or it may run out of memory. If this occurs, the finite encoding is simplified—using a concretization technique—to further constrain it and make it easier for the model checker to run to completion. This step is currently done manually but is easily automatable. Section 2.3 describes the above steps in more detail (steps ①, ②, ③ and ④ of Figure II.1).

Once an inductive invariant has been identified, I4 *generalizes* it to apply not only to the finite instance that produced it, but also to *all* instances of the protocol. This process (steps ⑤, ⑥ and ⑦) is described in detail in Section 2.4.

2.3 Invariant generation on a finite instance

The core idea of I4 is to leverage advances in model-checking techniques to generate an inductive invariant on a small finite instance of the protocol, then generalize this invariant to the full protocol. This section details the instantiation of the protocol to a finite instance to allow for automated model checking. Given a distributed protocol description in Ivy, I4 automatically creates a finite

instance of the protocol in the VMT format [CGT21], an extension of the SMT-LIB format [BFT16] to represent state-transition systems. I4 then calls the Averroes model-checking tool [GS19b] to generate an inductive invariant of this finite instance.

2.3.1 Leveraging the power of model checking

Model checking algorithms have had significant success on verification problems with bounded domains. Unbounded domains, however, continue to pose a serious challenge: an unbounded number of objects creates an unbounded number of interactions that are harder to reason about. As a result, automated reasoning on unbounded domain protocols generally employs quantifiers and performs complex, expensive—and often undecidable—quantifier-based reasoning. On the other hand, model checking for finite-state transition systems can be done quantifier-free, is much more mature, and has been successfully applied to many real-world finite systems [Bra11, EMB11, LS14, CGMT14, GS19a, GS19b]. I4 leverages the strength and maturity of model checking on a finite instance of the problem, before generalizing the invariant to the general protocol.

One of the key simplifications offered by reasoning on a finite instance is the ability to reason on *quantifier-free* formulas. For example, the assertion $\forall_E\ (zero \leq E)$ can be translated in the finite domain where $E \in \{e_0, e_1, e_2, e_3\}$ as

$$(zero \leq e_0) \wedge (zero \leq e_1) \wedge (zero \leq e_2) \wedge (zero \leq e_3).$$

This translation is always possible, even for complicated assertions involving two or more alternating quantifiers. For example, in the unbounded domain of nodes N, the assertion $\forall_{X \in N}\ \exists_{Y \in N}\ s(X, Y)$, can be simply expanded and translated to a finite 3-node domain $\{n_0, n_1, n_2\}$ as:

$$\begin{aligned}
&(s(n_0, n_0) \vee s(n_0, n_1) \vee s(n_0, n_2)) \wedge \\
&(s(n_1, n_0) \vee s(n_1, n_1) \vee s(n_1, n_2)) \wedge \\
&(s(n_2, n_0) \vee s(n_2, n_1) \vee s(n_2, n_2))
\end{aligned}$$

In general, in a finite domain any formula in first-order logic can be expanded into a quantifier-free formula, typically at the price of a growth in the size of the formula. In general, this does not scale but proves very useful for applying model checking on small, finite instances.

A typical definition of a distributed protocol involves different elements ranging over *domains*. Domains are typically *unbounded*. Examples of domains include the domain of nodes, clients, servers, epochs, rounds, transactions, etc. For example, the lock server protocol we described in Section 2.1 includes two unbounded domains: D_{server} representing the servers, and D_{client} representing the clients. Some domains such as epochs, rounds or transactions may have a natural

ordering. This ordering can be modelled in Ivy by adding axioms to the original protocol.

Creating a finite instance of a protocol consists of making each domain size *bounded* by an explicit value. The explicit bound enables the simplification of the expressions of the protocol, that can now be expressed without quantifiers. This, in turn, allows the finite protocol to be efficiently verified using model checking algorithms.

2.3.2 Picking a size for the finite instance

A crucial aspect of creating a finite instance is picking the right size for each domain of the finite instance. The instance must be large enough to exhibit some pattern that can then be generalized to the full protocol, but also small enough so that the finite instance of the problem is tractable for automatic model checking. Our current prototype relies on the user to provide an initial size of the finite instance. I4 will create an instance of that size; but we may later realize (see Section 2.4.2 for details) that this instance was too small. In that case, we pick one of the variables in the instance, increase its size by one, and repeat the process (step ⑧ in Figure II.1). This approach is akin to an iterative deepening search algorithm, starting with a small size or depth and growing it as needed.

The initial size of the problem is an educated guess based on a response to the question: *at least how many nodes, clients, servers or epochs are needed to exercise interesting actions and properties of the protocol?* For example, if clients are sharing a lock, at least two clients are needed to show possible violations (lock server protocol); if epochs are totally ordered, at least three epochs are needed for transitivity of the order to be interesting; if a special $zero$ epoch is further needed, at least four epochs are needed to exercise meaningful interactions of $zero$ and the order transitivity (distributed lock protocol). In the lock server protocol, for example, we set the initial size to one server and two clients, i.e., $|D_{server}| = 1$ and $|D_{client}| = 2$.

2.3.3 Translation from unbounded to finite domain

Once a size has been picked for each domain, I4 translates the Ivy implementation to the desired finite VMT instance (step ①). The first step is to define an explicit set of the right size for each domain. In the translation of the lock server presented in Figure II.2, we define $D_{server} = \{\texttt{S0}\}$ and $D_{client} = \{\texttt{C0}, \texttt{C1}\}$. Relations are then expanded to a number of boolean variables representing each possible instantiation. In this example, the relation $link : D_{client} \times D_{server} \rightarrow Bool$ is expanded into two boolean variables `link_C0_S0` and `link_C1_S0`. Function definitions are similarly expanded: a function $f : D_{client} \rightarrow D_{client}$ would be translated to two variables $\texttt{f\_C0}, \texttt{f\_C1} \in \{\texttt{C0}, \texttt{C1}\}$, respectively representing $f(\texttt{C0})$ and $f(\texttt{C1})$. Note that we only handle relations and functions with finite (or finitized) domains and codomains. From there on the translation is purely syntactic and

Figure II.2: Translation of lock server in Ivy to a finite instance (1 server / 2 clients) in VMT. Note that our VMT representation uses different vector sizes (1 for boolean, 2 for clients, 3 for servers) as an easy way to ensure that the model-checker does not try to compare values of different types.

straightforward (Figure II.2). For instance, the property expression

$$\forall_{C_1,C_2,S}\, link(C_1, S) \wedge link(C_2, S) \implies (C_1 = C_2)$$

is translated into a conjunction of 4 instances with $C_1 \in \{\texttt{C0}, \texttt{C1}\}$, $C_2 \in \{\texttt{C0}, \texttt{C1}\}$ and $S \in \{\texttt{S0}\}$. For example, one instance where $(C_1, C_2, S) = (\texttt{C0}, \texttt{C1}, \texttt{S0})$ becomes $\texttt{link\_C0\_S0} \wedge \texttt{link\_C1\_S0} \implies (\texttt{C0} = \texttt{C1})$.

2.3.4 Overcoming the limitations of model checking

In some cases, the finite instance may still be intractable for the model checker. In such cases, we manually *concretize* the values of certain variables to make the problem easier to solve, and to compute a more concise finite inductive invariant. For example, a finite, totally ordered domain of epochs $D_{epoch} = \{\texttt{E0}, \texttt{E1}, \texttt{E2}, \texttt{E3}\}$ can be concretized by explicitly imposing the domain axiom $(\texttt{E0} \leq \texttt{E1} \leq \texttt{E2} \leq \texttt{E3})$. This reduces the search space and allows the model checker to only consider states where epochs are following this specific order. Other examples of useful concretizations include fixing special elements as constants in the VMT instantiation, for example fixing the smallest epoch $zero$ to E0, explicitly specifying which node initially holds a lock, etc. Note that concretizing the finite instance results in limiting the scope of the finite inductive invariant we produce, i.e., the finite inductive invariant is only valid for the given concretized finite instance. But this does not hinder the invariant generalization presented in Section 2.4, as long as the concretizing information is included in the invariant generalization procedure.

There are, however, a few subtleties to pay attention to when solving finite instances instead of the original problem. First, the finite instance cannot capture interactions involving elements in a higher domain space. For example, with $|D_{client}| = 2$, the finite instance cannot capture interactions that involve at least 3 distinct clients. I4 handles this by increasing the size of the finite instance when it fails to generalize the invariant. Second, the finite instance may introduce special clauses that are not present in the original, unbounded protocol. For example, for the domain of epochs with $(\texttt{E0} \leq \texttt{E1} \leq \texttt{E2} \leq \texttt{E3})$, epoch E3 is inherently special in the finite space since there does not exist any epoch larger than E3. The finite inductive invariant may thus include special clauses involving E3, which may no longer be useful for the unbounded case. We call such clauses *instance-specific* clauses. Section 2.4 discusses how these are pruned during invariant generalization. Note, finally, that concretization is not a panacea. It is a *manual* and error-prone process and should therefore be used only when necessary to overcome the limitations of model-checking.

2.3.5 Invariant generation on the finite instance

After creating a finite VMT instance, I4 passes it on to the Averroes v2.0 tool [GS19b] (AVR) to perform model checking. If AVR finds that the finite instance does not uphold the safety property, it produces a counterexample, which can then be used to debug the protocol [Eng12, GGK+18]. If the safety property holds, AVR generates an inductive invariant for the finite instance; minimizes the invariant by removing redundant clauses; and then passes it on to the next step to be generalized.

2.4 Invariant generalization

Once AVR finds an inductive invariant for the finite instance, I4 will use that inductive invariant as a starting point to identify a *generalized* inductive invariant that works for all instances of the protocol. While performing this generalization, however, we must guard against the following two dangers:

- **Too-small finite instance** If the finite instance we have chosen is too small, its inductive invariant may not contain enough information to be generalizable to all instances. Consider, for example, what would happen if we started with an instance of the lock server protocol that had only one server and one client. Since the safety property of the protocol is trivially true in this case, the safety property is actually an inductive invariant for this particular instance. This means that the inductive invariant from this particular instance does not give us any information about the generalized inductive invariant that holds for all instances. In this case, the algorithm should eventually realize that this finite instance is not helpful and move on to a larger instance, eventually finding an instance that is large enough to be generalizable.

- **Instance-specific clauses** As we discussed in Section 2.3.4, even after we have identified an instance that is large enough to contain all the information required to put together a generalized inductive invariant, it is possible that the inductive invariant of the finite instance includes additional clauses that only hold for that specific instance and are thus not easily generalizable. Our generalization algorithm should therefore identify and prune such clauses.

The rest of this section describes the steps that I4 takes to identify this *generalized* inductive invariant (steps ⑤,⑥ and ⑦ in Figure II.1).

2.4.1 Initial generalization

The first step is to generalize the finite invariant to instances of arbitrary size in step ⑤ by universally quantifying the strengthening assertions (clauses) produced by AVR. Consider, for example, the clause $P(\texttt{N1})$, where P denotes an arbitrary predicate and `N1` is one of the nodes in

Algorithm 2 Generalization

```
1:  function GENERALIZATION(clause, relations)
2:      weakenings ← relations.conjunction()
3:      for var1 ∈ clause do
4:          for var2 ∈ clause do
5:              if var1 ≠ var2 & var1.type = var2.type then
6:                  weakenings.add(var1 ≠ var2)
7:              end if
8:          end for
9:      end for
10:     for const ∈ concrete_consts do
11:         for var ∈ clause do
12:             if var.type = const.type then
13:                 if var.value = const.value then
14:                     weakenings.add(var = const)
15:                 else
16:                     weakenings.add(var ≠ const)
17:                 end if
18:             end if
19:         end for
20:     end for
21:     return ∀var ∈ clause.  weakenings ⟹ clause
22: end function
```

the finite instance of the protocol. In step ⑤, I4 generalizes this clause to apply for all nodes; i.e., $\forall N_1.\ P(N_1)$ (line 21 of Algorithm 2). This is the simplest form of clause generalization, where the clause applies universally to all nodes.

There are two cases, however, where clauses do not apply universally and require a slightly more complex generalization. The first case (lines 3-9 in Algorithm 2) is when a clause involves different variables of the same type. In this case, we weaken the universally quantified clause to only apply to *distinct* elements of that type. For example, a clause such as $P(\texttt{N1}) \wedge Q(\texttt{N2})$ is generalized to $\forall N_1, N_2.\ (N_1 \neq N_2) \implies P(N_1) \wedge Q(N_2)$.

The second case (lines 2 and 10-20 in Algorithm 2) is a result of the (optional) concretization procedure we described in Section 2.3.4. During this procedure, we reduce the search space of the model checking problem by assigning concrete values to some of the state variables of the protocol. In our distributed lock protocol, for example, one of the nodes—called *first*—is special in that it is the one initially holding the lock. During concretization, we can assign $\texttt{N0} = first$, to limit the model checking problem to only consider the case where N0 is that node. Any such concrete assignments—in the form of generic relations or simple constant assignments—must be taken into account when performing generalization. For example, if we used $\texttt{N0} = first$ as our concretization, we would generalize clause $P(\texttt{N0})$ to $\forall N_0.\ (N_0 = first) \implies P(N_0)$. Similarly, we would generalize clause

$P(\texttt{N0}) \wedge Q(\texttt{N1})$ as $\forall N_0, N_1.\ (N_0 \neq N_1) \wedge (N_0 = first) \wedge (N_1 \neq first) \implies P(N_0) \wedge Q(N_1)$. In addition to such constant assignments, our concretization process may include additional information about this finite instance, in the form of generic relations defined in the original Ivy protocol. For example, our concretization can produce the clause $btw(\texttt{N0}, \texttt{N1}, \texttt{N2})$, denoting that node `N1` is *between* nodes `N0` and `N2` in a ring topology. The conjunction of all such concretizations is given as input to our generalization algorithm and is applied as a weakening to all invariant clauses (lines 1,2,21 of Algorithm 2).

2.4.2 Invariant pruning

After all clauses are generalized, they are added to the original protocol and are passed on to Ivy (step ⑥), which checks if they are sufficient to prove the correctness of the protocol. Ivy will check if the conjunction of all clauses is inductive. In particular, it tries to answer the following question (separately for each clause A, including the safety property). Given an arbitrary state s where the invariant holds, is there a valid transition to a state s' where A does not hold? There are three possible outcomes:

1. The generalized clauses are inductive and Ivy successfully proves the correctness of the protocol.

2. Ivy fails to prove the safety property on s'. This happens if the finite instance that led to the generalized clauses was too small to capture all behaviors of the distributed protocol. In this case, we need to create an instance with a larger size (e.g., more nodes) and repeat the process (step ⑧).

3. Ivy fails to prove one or more of the generalized clauses on s'. There are two possible reasons for this failure. First, it is possible that the finite instance we are considering is "large enough"—i.e., its inductive invariant covers all interesting behaviors of the unbounded protocol—but it additionally includes some instance-specific clauses. These clauses do not generalize to all instances and thus their universally quantified form is too strong and will fail Ivy's check. I4 removes these clauses from the inductive invariant (step ⑦) and retries the Ivy verification.

   The second reason why some assertion may not be inductive is that the entire invariant (i.e., the conjunction of the safety property and all strengthening clauses) is not inductive. This can happen when the finite instance is too small. Note that we do not have a way to distinguish this case from the case above—where the invariant was too strong, rather than too weak. We will therefore start pruning clauses one by one, until the invariant is too weak to support the

safety property (case (2) above), which will lead I4 to abandon the attempt on this finite instance and repeat the process with a larger one.

2.5 Evaluation

We evaluate the ability of I4 to infer inductive invariants by testing it on seven distributed protocols: a *lock server* (Section 2.5.1), a *leader election* algorithm (Section 2.5.2), a *distributed lock* protocol (Section 2.5.3), a *Chord ring [SMLN+03]* (Section 2.5.4), a *learning switch* (Section 2.5.5), a *database chain consistency* protocol (Section 2.5.6), and a *two-phase commit* protocol [Gra78] (Section 2.5.7). For the first six protocols, we verified the correctness of existing Ivy implementations using the safety properties specified in [PMP+16]; we cover all examples originally presented in [PMP+16], albeit with a much higher degree of automation. We implemented *two-phase commit* and specified its safety property based on its original description [Gra78]. Finally, Section 2.5.8 evaluates our I4 prototype in terms of how long it takes to verify each protocol. All our implementations and artifacts can be found in GitHub [MGJ+].

We also tried the I4 approach on Paxos, but we have not found an inductive invariant. This is because the Averroes model-checker runs out of memory even on small, finite instances of the Paxos protocol. This is not too surprising, since Averroes was originally designed for hardware model-checking and not for distributed systems. We therefore do not think that this is a fundamental roadblock and are currently exploring ways to leverage the inherent regularity of distributed systems to facilitate the model-checker's job.

We used the first three protocols (lock server, leader election and distributed lock) to develop and refine the I4 approach. We were then able to apply the I4 approach with no modification on the last four protocols (Chord ring, learning switch, database chain consistency and two-phase commit), and prove their safety property fully automatically—except for the simple manually-added concretization in Chord, which required a cursory inspection of about one minute—*without* even fully understanding each protocol.

Table II.1 presents some relevant parameters for each protocol and for its finite instantiation, such as the number of domains and variables it uses, the size of the finite instance and the complexity of the resulting invariant. Note that our concretization phase—where needed—is relatively low-effort, requiring at most one assignment in all cases.

2.5.1 Lock Server

Our first case study is a simple lock server, our running example from Section 2.1. Since the safety property of the lock server is trivially satisfied with only one client, we instantiate this protocol with one server and two clients. I4 generates the generalized inductive invariant for this protocol by

Protocol	Domains	Var	Finite instance size	Concreti-zations	SMT calls	Clauses in invariant	Clauses in minimized invariant	Pruning iterations
Lock server	2	2	$client = 2$ $server = 1$	$\emptyset$	40	3	2	0
Leader election in ring	2	5	$node = 3$ $id = 3$	$idn(N_i) = ID_i$	10527	61	19	0
Distributed lock protocol	2	5	$node = 2$ $epoch = 4$	$zero = E_0$	94713	629	241	2
Chord ring maintenance	1	9	$node = 4$	$org = N_0$	286818	1141	94	2
Learning switch	2	6	$node = 3$ $packet = 1$	$\emptyset$	4986	105	53	2
Database chain replication	4	13	$transaction = 3$ $operation = 3$ $key = 1$ $node = 2$	$\emptyset$	6552	154	31	2
Two-Phase Commit	1	7	$node = 6$	$\emptyset$	9619	88	46	0

Table II.1: Various parameters for creating finite instances for our seven distributed protocols. *Var* is the number of state variables and *SMT calls* is the number of SMT [BFT16] calls required to find the inductive invariant of the finite instance.

going through its generalization (step ④ in Fig. II.1), where it places universal quantifiers before every strengthening assertion. The generalized inductive invariant below passes Ivy's verification with no manual effort.

$$\boxed{\begin{array}{l}\forall S0, C0.\ \neg(semaphore(S0) \wedge link(C0, S0)) \quad \wedge \\ Safety\ Property\end{array}}$$

2.5.2 Leader Election

Our second case study is a leader election protocol on a ring [CR79, PMP+16]. Given a ring of an unbounded number of nodes, each with its own unique ID, the goal of the protocol is to elect the node with the highest ID to be the leader. A node can either (a) send its ID to the next node; or (b) forward a message from the previous node, if the ID in the message is larger than its own ID. When a node receives its own ID, the protocol determines that no other ID is larger than the node's own ID, and this node becomes the leader.

This protocol uses two domains $node$ and id, respectively for nodes and IDs, and the ID of node N is $idn(N)$. The ring structure is modelled using a relation $btw(N_1, N_2, N_3)$ indicating whether node N_2 is between nodes N_1 and N_3, and includes axioms to build a ring topology, where each node can only communicate with its two neighbors [PMP+16]. The protocol also instantiates a total order $le(ID_1, ID_2)$ to compare any two IDs. A relation $pending(ID, N)$ is used to indicate that there is a message ID to node N pending in the network. As the network may delay or duplicate any message, the protocol never discards any sent message.

The safety property of leader election is defined as "there cannot be two distinct leaders":

$$\forall N_1, N_2.\ leader(N_1) \wedge leader(N_2) \implies (N_1 = N_2)$$

Since a meaningful ring modeled with btw involves at least three nodes, we generate the inductive invariant on a three-node finite instance, with domain of nodes $\{\texttt{N0}, \texttt{N1}, \texttt{N2}\}$ and domain of IDs $\{\texttt{ID0}, \texttt{ID1}, \texttt{ID2}\}$. AVR finds an invariant on this finite instance, but we are not able to generalize it to the full protocol. We further find that if we increase the size to four nodes, AVR runs out of memory without finding an inductive invariant on the final instance. This is where our *concretization* technique proves its usefulness: by manually assigning concrete values to the three-node protocol—i.e., by adding axioms $idn(\texttt{N0}) = \texttt{ID0}$, $idn(\texttt{N1}) = \texttt{ID1}$ and $idn(\texttt{N2}) = \texttt{ID2}$—we facilitate AVR to find a finite-instance invariant (shown in Table II.2) that I4 can generalize to the full protocol (shown in Table II.3) and pass Ivy's verification. Since we make no assumption on the order of `ID0`, `ID1` and `ID2`, those axioms have no impact on the generality of the proof.

Note that, as part of concretization, a universally-quantified conjunction of the concretization

$\neg(\neg pending\_\texttt{ID0\_N0} \wedge leader\_\texttt{N0})$	$\wedge$
$\neg(\neg pending\_\texttt{ID1\_N1} \wedge leader\_\texttt{N1})$	$\wedge$
$\neg(\neg pending\_\texttt{ID1\_N2} \wedge pending\_\texttt{ID1\_N1})$	$\wedge$
$\neg(\neg pending\_\texttt{ID1\_N2} \wedge btw\_\texttt{N0\_N1\_N2} \wedge pending\_\texttt{ID1\_N0})$	$\wedge$
$\neg(le\_\texttt{ID0\_ID1} \wedge pending\_\texttt{ID0\_N0})$	$\wedge$
$\neg(le\_\texttt{ID0\_ID1} \wedge btw\_\texttt{N0\_N1\_N2} \wedge pending\_\texttt{ID0\_N2})$	$\wedge$
$\neg(le\_\texttt{ID0\_ID2} \wedge pending\_\texttt{ID0\_N0})$	$\wedge$
$\neg(le\_\texttt{ID0\_ID2} \wedge btw\_\texttt{N1\_N0\_N2} \wedge pending\_\texttt{ID0\_N1})$	$\wedge$
$\neg(le\_\texttt{ID1\_ID0} \wedge pending\_\texttt{ID1\_N1})$	$\wedge$
$\neg(le\_\texttt{ID1\_ID0} \wedge btw\_\texttt{N1\_N0\_N2} \wedge pending\_\texttt{ID1\_N2})$	$\wedge$
$\neg(le\_\texttt{ID1\_ID2} \wedge pending\_\texttt{ID1\_N1})$	$\wedge$
$\neg(le\_\texttt{ID1\_ID2} \wedge btw\_\texttt{N0\_N1\_N2} \wedge pending\_\texttt{ID1\_N0})$	$\wedge$
$\neg(le\_\texttt{ID2\_ID0} \wedge leader\_\texttt{N2})$	$\wedge$
$\neg(le\_\texttt{ID2\_ID0} \wedge pending\_\texttt{ID2\_N2})$	$\wedge$
$\neg(le\_\texttt{ID2\_ID0} \wedge btw\_\texttt{N0\_N1\_N2} \wedge pending\_\texttt{ID2\_N1})$	$\wedge$
$\neg(le\_\texttt{ID2\_ID1} \wedge leader\_\texttt{N2})$	$\wedge$
$\neg(le\_\texttt{ID2\_ID1} \wedge pending\_\texttt{ID2\_N2})$	$\wedge$
$\neg(le\_\texttt{ID2\_ID1} \wedge btw\_\texttt{N1\_N0\_N2} \wedge pending\_\texttt{ID2\_N0})$	$\wedge$
Safety Property	

Table II.2: Instance of invariant of Leader Election.

axiom

$$\forall N_0 \neq N_1 \neq N_2, ID_0 \neq ID_1 \neq ID_2.$$
$$(idn(N_0) = ID_0) \wedge (idn(N_1) = ID_1) \wedge (idn(N_2) = ID_2)$$

is passed on to the generalization algorithm, which adds it as a weakening to all clauses in the general invariant. For example, the clause $\neg(\neg pending\_\texttt{ID0\_N0} \wedge leader\_\texttt{N0})$ (highlighed in Table II.2) is generalized into (highlighted in Table II.3, slightly edited for readability):

$$\forall N_0, ID_0.\ (idn(N_0) = ID_0) \implies$$
$$\neg(\neg pending(ID_0, N_0) \wedge leader(N_0))$$

I4 applies a similar strategy to all clauses of the finite-instance invariant and obtains the inductive invariant shown (simplified for readability) in Table II.3. This generalized inductive invariant passes Ivy's verification, thus proving the correctness of the protocol.

$\forall N_0, ID_0.$	$(idn(N_0) = ID_0) \implies \neg(\neg(pending(ID_0, N_0)) \wedge leader(N_0))$	$\wedge$
$\forall N_0, N_1, N_2, ID_1.$	$(idn(N_1) = ID_1) \wedge (N_0 \neq N_1) \wedge (N_0 \neq N_2) \wedge (N_1 \neq N_2)$	
	$\implies \neg(\neg(pending(ID_1, N_2)) \wedge btw(N_0, N_1, N_2) \wedge pending(ID_1, N_0))$	$\wedge$
$\forall N_0, N_1, ID_0, ID_1.$	$(idn(N_0) = ID_0) \wedge (idn(N_1) = ID_1) \wedge (ID_0 \neq ID_1)$	
	$\implies \neg(le(ID_0, ID_1) \wedge pending(ID_0, N_0))$	$\wedge$
$\forall N_0, N_1, N_2, ID_0, ID_1.$	$(idn(N_0) = ID_0) \wedge (idn(N_1) = ID_1) \wedge (ID_0 \neq ID_1)$	
	$\wedge (N_0 \neq N_1) \wedge (N_0 \neq N_2) \wedge (N_1 \neq N_2)$	
	$\implies \neg(le(ID_0, ID_1) \wedge btw(N_0, N_1, N_2) \wedge pending(ID_0, N_2))$	$\wedge$
Safety Property		

Table II.3: Generalized invariant of Leader Election (slightly edited for readability).

2.5.3 Distributed lock

Our third case study is a distributed lock protocol [HHK+15, PMP+16]. This protocol models an unbounded number of nodes that transfer the ownership of a lock among themselves. Nodes transfer locks by sending and receiving messages in an unreliable network that can drop or duplicate messages. The ownership of a lock is associated with an ever increasing epoch, to allow detection of stale messages.

If a node N holds the lock at epoch $ep(N)$, N can pass the lock to any node N' in the system at epoch $E > ep(N)$ by sending it a *transfer*(E, N') message. When a node N' at epoch $ep(N')$ receives a *transfer*(E, N') message with epoch $E > ep(N')$, node N' accepts the lock at epoch E, and sends a message *locked*(E, N') to denote that N' holds the lock at epoch E', and update $ep(N') = E$. Otherwise, if $E' \leq ep(N')$, N' ignores this stale message. As the network may delay or duplicate any message, the protocol never discards any sent *transfer* message.

The safety property of the protocol is "no two distinct nodes can hold the lock at the same time":

$$\forall N_1, N_2, E.\ locked(E, N_1) \wedge locked(E, N_2) \implies (N_1 = N_2)$$

This protocol involves two sources of infinity: the number of nodes and the number of epochs. Therefore a finite instance of the protocol bounds not only the number of nodes, but also the number of epochs. Unlike previous protocols, even an instance with just two nodes is enough to generate an unbounded number of messages by passing the lock between them with ever-increasing epoch numbers. I4 is able to prove the correctness of the protocol based on a finite instance with just two nodes and four epochs.

During our experiments, we found that AVR runs out of memory due to the large search space. To simplify the problem, we manually concretize the special epoch $zero$ to `E0` (as discussed in Section 2.3.4), facilitating AVR's task.

Given the inductive invariant for this finite instance, I4 still needs two iterations of invariant

pruning to get rid of instance-specific clauses, after which it produces an inductive invariant which passes Ivy's verification.

2.5.4 Chord Ring

Our next case study is a Chord Ring [SMLN+03], a popular distributed hash table approach in peer-to-peer systems. In the Chord protocol, nodes are organized in a ring, and each node stores part of the hash table. Additionally, nodes may join or leave the ring at any time, prompting a re-arranging of the ring. The safety property of interest is that the ring remains connected under certain assumptions about failures.

We use the Chord protocol description in Ivy [PMP+16], which models the protocol in Ivy with each node maintaining two pointers: one to its successor and one to its successor's successor. Those two pointers are implemented as two relations over the nodes, $s1(N_1, N_2)$ when N_2 is N_1's successor, and $s2(N_1, N_3)$ when N_3 is N_1's successor's successor. Even if some nodes fail, the safety property remains true as long as every live node still has a pointer to at least one other live node. The failure of a node is modelled as a normal transition. A `test` transition is used to check whether a given node can access another given node, and sets an error flag to true if that is not the case. The safety property is simply expressed as the error flag never being true.

Part of the protocol was first proved by Zave [Zav17], including an informal, intuitive proof as well as a formal proof in Alloy [Jac02]. The protocol was later implemented in Ivy [PMP+16], formally proving (with manual effort) the primary safety property. We adopted a slightly modified version of the Ivy implementation, and were able to prove the same safety property based on a finite instance with 4 nodes, and by concretizing a special node, $org = N_0$.

2.5.5 Learning Switch

Learning switches maintain a table that maps MAC addresses to ports where the incoming frames will be forwarded. When a frame is received, a learning switch will check to see if the source MAC address is already in the table, and if not, it will insert a (Source MAC Address, Port Number) entry into the table. The switch will then check the destination MAC address of the incoming frame in the table. If there is an entry mapping the destination MAC address to port number, the switch will forward the frame to that port. Otherwise, the switch will send the packet to all its ports except the port where the frame was received from (otherwise known as flooding).

We use the existing implementation of the learning switch protocol in Ivy, by simply updating it to the latest Ivy syntax, and feeding it to I4. The safety property states that there does not exist any forwarding cycles, where an incoming frame would be forwarded to the same port that it arrived from. We instantiated this protocol with 4 nodes and 2 packets since forwarding cycles can possibly

form in such a small setup.

Given the inductive invariant for the finite instance with 4 nodes and 2 packets, I4 can infer the general inductive invariant and pass Ivy's verification with no manual effort. Later, we found that even 3 nodes with only a single packet is sufficient for I4 to infer the generalizable inductive invariant for this protocol.

2.5.6 Database Chain Consistency

Database chain consistency is the last protocol we evaluated from the protocol suite found in the Ivy paper. This protocol provides traditional database safety guarantees of atomicity, serializability, and isolation for distributed databases. In this distributed setting, a chain transaction is split into subtransactions that operate sequentially on data that is sharded across multiple nodes. In order for the chain transaction to commit, each subtransaction should also commit. If any subtransaction aborts, then the entire chain transaction is also aborted.

Using I4, we successfully verified the safety properties defined by the Ivy authors. We started with an instance of 4 transactions, 5 operations, 2 keys and 2 nodes. That initial instance was large enough to be generalizable. We later found that even a smaller instance with 3 transactions, 3 operations, 1 key and 2 nodes is generalizable.

2.5.7 Two-Phase Commit

We implemented two-phase commit in Ivy. We proved that our implementation satisfies the traditional Atomic Commit safety properties: (a) all processes that reach a decision reach the same one, (b) the Commit decision can only be reached if all processes vote Yes, and (c) if there are no failures and all processes vote Yes, then the decision must be Commit. We started with an initial size of 4 nodes, and kept increasing the instance size by one. We finally proved the correctness of the protocol with 6 nodes.

Protocol	F	M	G	total
Lock server	0.02	0.0	0.8	0.8
Leader election in ring	4.0	0.1	2.0	6.1
Distributed lock protocol	30.6	53.3	75.5	159.5
Chord ring maintenance	386.1	218.5	24.3	628.9
Learning switch	2.9	0.8	6.9	10.7
Database chain replication	4.2	2.3	6.2	12.6
Two-Phase Commit	2.6	0.1	1.6	4.3

Table II.4: Runtime results (in seconds). F is the time required to find the finite inductive invariant; M is the time is takes to minimize the finite inductive invariant; and G is the time to generalize the clauses and perform invariant pruning.

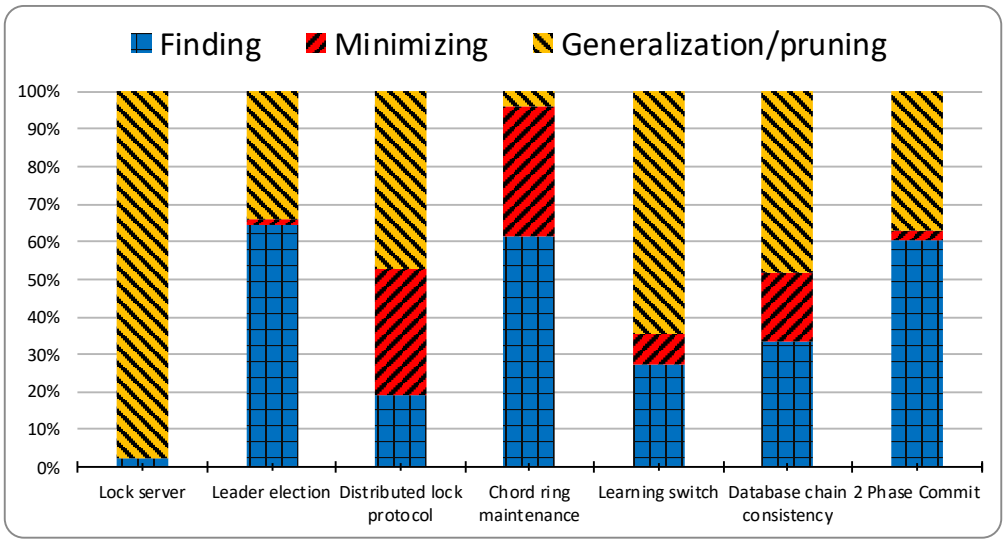

Figure II.3: Runtime break down for each of the seven protocols we verified using I4.

2.5.8 Runtime Breakdown of I4's Verification

Table II.4 and Figure II.3 show the runtime of various phases of the I4 algorithm. For clarity, we omit the time it takes to generate the finite instances, as it was negligible and almost identical among all protocols (∼0.3 seconds). In most cases, I4 can prove the correctness of these protocols within a few tens of seconds, with the worst case being that one has to wait for 10.5 minutes.

2.6 Limitations and future directions

This chapter takes the first step in a new direction—proving the correctness of distributed protocols based on the inductive invariants of small, finite instances. Our experience applying this approach to real protocols is thus far encouraging: we have been able to prove the correctness of a number of interesting protocols, with little to no manual effort. For all its successes heretofore, we believe that there are several more steps to be taken in this research direction. We list below a number of limitations of our current approach and prototype in the hope that they will serve as inspiration for future research.

- **Choosing an instance size** When instantiating a protocol, we need to use an instance that is large enough to exhibit all the interesting properties of an arbitrarily-sized instance. Currently, we incrementally increase the size of the instance to guarantee we will eventually consider an instance that is large enough. In our experiments, we manually select an initial size to speed this process up.

- **Existential quantifiers** Our generalization algorithm adds universal quantifiers to generalize finite inductive invariants. We currently do not support inductive invariants that include existential quantifiers. Our experience so far suggests that existential quantifiers are not very common, but supporting them would increase the generality of our approach. For example, we were careful to write the two-phase commit protocol so that it would not use any existential quantifiers, but doing so required human intervention, which we try to avoid as much as possible.

- **Verifying implementations** I4 focuses on verifying distributed protocols, rather than implementations. Automating the proof of a full distributed implementation is much harder, as it usually requires reasoning about undecidable fragments of logic, which are notoriously hard to verify automatically. Also, our current prototype does not support some of Ivy's features, such as objects, translation to concrete imperative code, arbitrary assumptions, etc.

- **Optimizing model checking for distributed systems** We are currently relying on existing, unmodified model checkers to find the inductive invariants of finite instances. While these tools have come a long way in recent years, they may still not scale even for small instances of some complex protocols. Our concretization technique helps mitigate this problem to a certain degree, but we believe this is only the first step in a line of optimizations that will customize model checking algorithms to deal with the particular requirements of distributed systems. Specifically, the high-level structure of a protocol can be used by the model checker to identify *compact* strengthening assertions that, in some sense, respect that structure and exhibit its inherent regularity.

2.7 Conclusion

This chapter presents I4, a new approach for verifying the correctness of distributed protocols, with little to no manual effort and without relying on human intuition. I4 is based on a simple intuition: an inductive invariant of a small, finite instance can be used to infer a generalized inductive invariant that holds for all instances of the protocol. I4 leverages the power of model checking to automatically find an inductive invariant for a small instance of the protocol and then generalizes that invariant to instances of arbitrary size. Our evaluation shows that I4 is successful in automatically proving the correctness of a number of interesting distributed protocols, even ones whose subtleties and internal workings were unknown to us.

CHAPTER III

Refinement-guided Automation

So far, we have seen how we can automatically identify an inductive invariant of a distributed protocol. Unfortunately, all such automation only works in a single layer, but not refinement proofs. In this Chapter, we aim to get the best of two worlds: the *power of refinement* but with only *a fraction of the manual effort*. To achieve this goal, we introduce Sift, a two-tier methodology that combines automated verification with a small amount of manual effort to push the boundary of the kinds of systems that can benefit from proof automation. Just like IronFleet before it, Sift is a *methodology*, not a tool. Its contribution is a way of structuring refinement proofs in order to leverage the automation of existing tools. Similar to how IronFleet guided developers to manually construct proofs based on the existing tools (TLA+ and Dafny), so does Sift show developers how to construct proofs that leverage the automation of more recent tools, like IC3PO and Ivy.

The first tier of Sift introduces a new technique, called *refinement-guided automation*, which leverages the automation of monolithic provers in the context of a refinement proof. At the high level, this technique enables the automation of refinement proofs between two layers by *encapsulating* the state of the upper, more abstract, layer into the state of the lower, more concrete layer. This encapsulation allows us to transform a two-layer refinement proof into a single-layer, monolithic proof that provers like I4 [MGJ+19b, MGJ+19a], IC3PO [GS21], SWISS [HHMP21], and DistAI [YTG+21] can perform.

Leveraging automation to prove refinement is not always enough, though. Monolithic provers have their limits and thus some refinement proofs are just too complex to prove automatically. When that happens, we provide developers with an escape hatch. The second tier of the Sift methodology describes a divide-and-conquer technique for introducing intermediate layers, thus splitting a complex proof into chunks that are small enough for the prover to handle.

The Sift methodology applies refinement-guided automation within each refinement step and uses our divide-and-conquer technique to split a refinement step into smaller, more manageable steps. As a result, Sift allows us to apply, for the first time, automation to refinement-based proofs and scale to much harder problems than was previously possible. We use Sift to automate the verification of four distributed implementations, whose proof required minimal manual effort (less

than five minutes, in most cases).

We further use our divide-and-conquer technique to prove the correctness of an implementation of Raft [OO14] and an implementation of MultiPaxos [Lam98, Lam01] — a feat that was only possible before by providing a fully manual proof of correctness. Using Sift, we were able to automate most of the proof for both Raft and MultiPaxos. The manual effort required to complete the proof with Sift is not only significantly less than that of previous approaches, it is also much less reliant on having expertise in formal verification.

Overall, this chapter makes the following contributions:

- We introduce *refinement-guided automation*, a technique that leverages the automation of monolithic-oriented tools to perform more complex, refinement-based proofs.
- We present a divide-and-conquer technique for splitting a complex refinement proof into smaller pieces, such that each piece is amenable to automated verification.
- We introduce Sift, a methodology that incorporates refinement-guided automation and our divide-and-conquer technique. We evaluate Sift on six distributed implementations and find that it allows us to prove their correctness in a mostly automated manner which drastically reduces the manual effort required compared to previous refinement-based approaches.

We present the specifics of this work in this chapter. Section 3.1 discusses the tradeoff between automation and refinement. Section 3.2 recaps some background material, while Section 3.3 gives an overview of Sift. Section 3.4 introduces refinement-guided automation and Section 3.5 shows how to introduce intermediate refinement layers when needed. Section 3.6 evaluates the effectiveness of using Sift to automate the verification of a number of distributed implementations. Section 3.7 presents the limitations of Sift and discusses future work, and Section 3.8 concludes.

3.1 The Price of Automation

As discussed earlier, there are currently two approaches for verifying the correctness of distributed systems. The first is the powerful but manual approach of IronFleet and Verdi [HHK+15, HHK+17, WWP+15], where the developer uses refinement to show that a complex implementation is equivalent—through a series of layers or transformations—to an abstract specification.

The second approach is that of I4 [MGJ+19a], IC3PO [GS21], SWISS [HHMP21] and DistAI [YTG+21] which leverage the power of model-checking and SMT solving [BFT16] to automatically prove the correctness of abstract system descriptions at the protocol level. These approaches aim to prove that a given safety property holds for the protocol at hand, by automatically identifying an *inductive invariant* that implies this safety property.

While such automation is undoubtedly a desirable property, it comes at a heavy price. In particular, I4, IC3PO, SWISS and DistAI can only perform *monolithic* proofs: they can prove that a

protocol—defined as a single layer—satisfies a given safety property. As we described, this not only limits the type of specifications we can use, but also severely limits the scalability of the approach.

Most importantly, the scalability limitation is not an artifact of the implementation of *monolithic provers*—like I4, IC3PO, SWISS and DistAI—but rather inherent in their design. By asking the underlying solver to find an inductive invariant that supports the desired safety property, they essentially adopt an *all-or-nothing* approach: either the solver is powerful enough to find an inductive invariant or it is not. If we consider more and more complex systems, we soon reach a point where the solver is simply not powerful enough to find an inductive invariant.

In fact, a similar dichotomy presents itself when the protocol description has elements outside the *decidable* fragment of logic [Lew80, PLSS17]. In several of these cases, the solver struggles considerably, even when it is trivial for a human to split the problem into decidable sub-problems. Without the ability to split this monolithic proof into multiple pieces, there is no middle ground. For example, I4 simply fails when the problem lies outside the decidable fragment, even though it is still possible to use refinement to split the protocol into two layers, each of which is separately decidable [TLM+18].

In this chapter, we show that there exists a middle ground between the fully manual approaches that support refinement, like IronFleet and Verdi; and the automated-but-monolithic approaches, like I4, IC3PO, SWISS and DistAI. This middle ground, enabled by our novel Sift methodology, allows for refinement-based reasoning—and thus allows us to prove the correctness of complex distributed implementations—while making heavy use of automation to drastically reduce the amount of manual effort required compared to IronFleet and Verdi.

3.2 Background

3.2.1 Multi-Layer Refinement

Sift is heavily based on the notion of refinement. We will therefore first recap the notion of refinement and how it can be used to prove the correctness of complex systems.

A system P *refines* another system Q if the observable outputs produced by any execution of P can also be produced by some execution of Q. In the case of distributed systems, the only outputs that are visible to external observers are the messages produced by these systems.

In the simplest application of refinement, the developer writes two layers: a specification and an implementation. The specification is written as a simple, logically centralized state machine. In the case of a sharded key-value store, for example, the specification is a simple map, where the only possible actions are to `put` something to the map, or to `get` something from the map [HHK+15, HHK+17]. The developer then shows that the implementation refines the specification, thus proving the correctness of the implementation.

In more complex systems, directly proving refinement from the implementation to the specification can be difficult [HHK+15, HHK+17, WWP+15]. In that case, the developer must insert one or more increasingly complex layers between the implementation and specification, thus creating a multi-layer structure, where each layer must be proven to refine the one above it. We explain how to design and insert intermediate layers in Section 3.5.

3.2.2 Automated Reasoning and Monolithic Provers

Traditional verification languages [Lei10, BBC+97] rely on the developer to write a full proof, including a large number of manual annotations. As a result, approaches like IronFleet [HHK+15, HHK+17] and Verdi [WWP+15] incur a high proof-to-code ratio. To reduce this manual effort, Ivy [PMP+16] uses decidable logic to guarantee completeness. With Ivy, the developer only needs to find an *inductive invariant*—an invariant which is closed (inductive) under the system transitions—and the prover can automatically identify if this inductive invariant is correct. Ivy significantly simplifies the effort of proving the correctness of distributed systems, but finding such inductive invariants is still a non-trivial task that relies on human intuition and an intimate understanding of the system at hand.

To push the automation a step further, I4 [MGJ+19a] leverages the regularity of distributed protocols, so that the inductive invariant can be automatically inferred from a small, finite instance. Unfortunately, such a strategy only applies to monolithic protocols, not refinement proofs. Thus, I4 doesn't scale well when the system has a large state space and complex transitions. More recent tools [GS21, HHMP21, YTG+21] have followed the direction of using finite instances to guide the verification of distributed protocols. All these tools, however, apply only to monolithic proofs and cannot support refinement. We call such tools *monolithic provers*.

3.2.3 IC3PO: Our Monolithic Prover of Choice

The design of Sift does not rely on the internals of the monolithic prover that it uses. The refinement-guided automation technique of Sift can leverage any tool designed for automating monolithic, single-layer proofs. In fact, we previously tried I4 as the monolithic prover in Sift, but later found that IC3PO performs better. Our experience so far shows that IC3PO also outperforms SWISS and DistAI. As new and more powerful monolithic provers become available, Sift can adopt them to perform even larger refinement steps to further reduce manual effort. The next paragraph gives a short overview of IC3PO.

IC3PO [GS21, GS] is a recently-developed prover that uses the synergistic relationship between symmetry and quantification to prove the safety of distributed protocols fully automatically, by inferring compact inductive invariants with both *universal* and *existential* quantifiers. At its core,

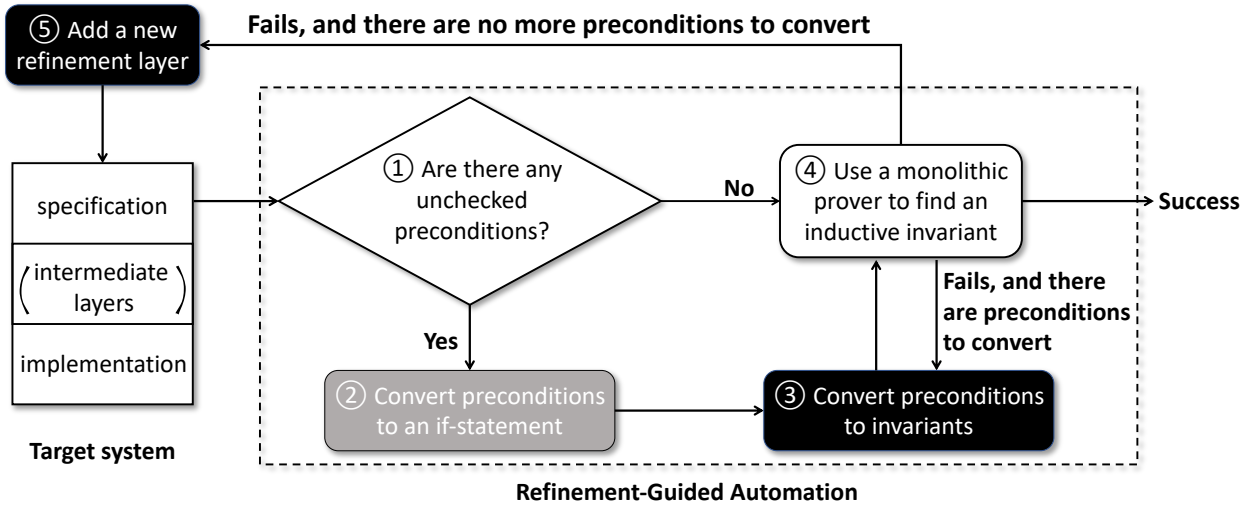

Figure III.1: Summary of the Sift methodology. White boxes are fully automated, gray boxes indicate a trivial syntax change, and black boxes denote manual effort.

IC3PO exploits the inherent regularity present in distributed protocols to significantly scale up IC3/PDR-style verification [Bra11, EMB11] over finite instances of the protocol. Starting with an initial instance size, IC3PO systematically computes quantified inductive invariants over protocol instances of increasing sizes, until protocol behaviors *saturate*, concluding with an inductive proof that works for all instances of the protocol.

3.3 Overview of Sift

This chapter introduces Sift, a methodology that allows reasoning about complex systems while still using a large degree of automation in proofs. Sift accomplishes this by employing a small amount of manual effort, when needed, to split the system into a number of layers, where each layer can be shown to refine the layer above it.

Figure III.1 shows an overview of the Sift methodology. Initially, the developer starts with an implementation of the system, along with a specification, both written in the Ivy language [PMP+16]. If one were to use the Ivy prover, they would have to provide a manual proof of refinement between the specification and implementation. Sift, instead, introduces our *encapsulation* technique to merge the two layers into a single proof that our monolithic prover can attempt to solve.

Indeed, the first step of the Sift methodology is to attempt to prove refinement directly between the implementation and specification layers. If this proof is too much for the prover to handle, the developer adds an additional layer of refinement and tries again. Each additional layer of refinement splits the proof into smaller pieces that are more amenable to automation; but, of course, this comes at the cost of some manual effort, as the developer must manually introduce the new layer.

Algorithm 3 Specification of the Sharded Hash Table (SHT)

```
1  function requests(R : request) : bool
2  function replies(R : reply) : bool
3  function map(K : key) : value
4  initialization {
5    ∀R. requests(R) ← false
6    ∀R. replies(R) ← false
7    ∀K. map(K) ← 0
8  }
9  action commit(req : request, rep : reply) = {
10   require rep.type = req.type
11   require rep.src = req.src
12   require rep.key = req.key
13   require req.type = read ⇒ rep.data = map(req.key)
14   if ¬requests(req) {                          ▷ require ¬requests(req)
15     if req.type = write {
16       map(req.key) ← req.data
17     };
18     requests(req) ← true;
19     replies(rep) ← true;
20   }
21 }
```

In the next two sections, we describe the Sift methodology in more detail. Section 3.4 describes how we can use the automation of a monolithic prover to perform a refinement proof between two layers (steps ①-④ in Figure III.1). Section 3.5 presents the methodology for adding intermediate layers to the refinement structure (step ⑤).

Case Study: Sharded Hash Table Throughout this chapter, we use the example of a Sharded Hash Table application (SHT) [HHK+15] to illustrate the Sift methodology. SHT implements a distributed key-value store, and consists of two layers, a specification layer and an implementation layer. As shown in Algorithm 3, the specification layer describes a key-value store as a simple *map* from keys to values. It maintains two local sets (modeled as boolean-valued functions, lines 1 and 2) to keep track of which messages (requests and replies) have been sent. Initially, all keys are mapped to 0, and no messages have been sent. Requests can either be read requests or write requests. The only transition allowed by this specification is to *commit* a request req and its reply rep: i.e., perform the update (if this is a write request) and mark the corresponding messages as sent by setting $requests(req)$ and $replies(rep)$ to $true$. The specification layer consists of 32 lines of Ivy code. In the implementation layer, every node contains a local hash table containing some subset of the total keys in the system and a delegation map. The node uses the delegation map to

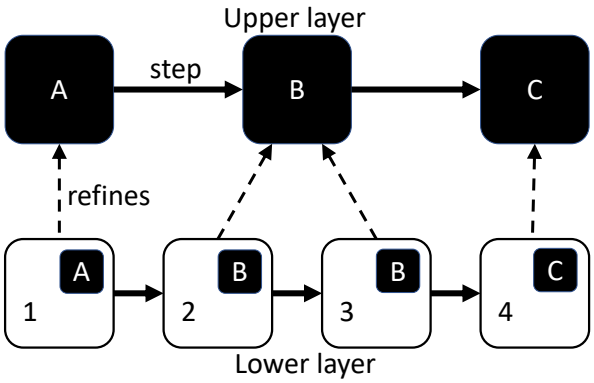

Figure III.2: Encapsulation: to enable automatic refinement proofs, the state of the upper layer (A, B, C) is encapsulated inside the state of the lower layer (1, 2, 3, 4) refining it.

maintain its knowledge of where keys are stored on remote nodes. Each node can service a request using `get` and `set` actions for the keys that are locally stored, or use the delegation map to look up and forward requests to the appropriate node in the system if the requested key is not local. Nodes at the implementation layer can dynamically exchange sets of keys they are responsible for, by exchanging `delegate` messages (each carrying a key-value pair) among themselves. The implementation layer consists of 127 lines of Ivy code.

We aim to show that the implementation layer *refines* the specification, i.e., that any observable output produced by any execution of the implementation layer can also be produced by some execution of the specification. A key property is that for every key owned by a node at the implementation layer, the data matches the value stored in the specification. Additionally, every key is either owned by exactly one node in the system, or part of an in-flight `delegate` message.

3.4 Refinement-Guided Automation

We first explain the key high-level idea behind automating refinement proofs (step ④ in Fig. III.1, Section 3.4.1). We then present what modifications Sift makes to the layers of a target system description to ensure that the correspondences between the layers are correctly represented before a proof is attempted (steps ①-③ in Fig. III.1, Section 3.4.2).

3.4.1 From Monolithic Proofs to Refinement

A key feature of Sift is that it uses the automation of monolithic provers to perform more complex, refinement-based proofs. As we explained in Section 3.1, monolithic (i.e. single-layer) proofs do not scale to complex systems, either due to complexity or undecidability. Yet, monolithic proofs are the only type of proof supported by these provers. The first innovation of Sift is that it

Algorithm 4 Example of encapsulation in SHT

```
1   action set(req : request) = {
2     require req.type = write
3     owner ← delegation.get_owner(req.key)
4     if owner = me {
5       hash(req.k) ← req.v
6       rep ← create_reply(req)
7       call spec.commit(req, rep)
8       call network.send_reply(rep)
9     } else {
10      call network.forward_request(req, owner)
11    }
12  }
```

converts a refinement proof between two layers into a monolithic proof which can be given as input to any monolithic prover.

We perform this transformation using our *encapsulation* technique, depicted in Figure III.2. The idea of encapsulation is simple: if we want to show that a lower layer L refines an upper layer U, then we augment the state of L with the state of U. Additionally, whenever the state machine L makes a transition, the *encapsulated* U state also makes an upper-layer transition. In practice, this is expressed as a function call in Ivy, where the lower layer invokes a transition on its encapsulated state. For example, in the SHT application, the lower layer includes an encapsulated $spec$ object (see Algorithm 3) and a lower-layer transition calls $spec.commit()$ if it refines the $commit$ transition of the upper layer.

Such calls require a little manual effort to be inserted into the lower layer. But such effort is reasonable because when implementing these layers, most developers have intuition on how the upper layer takes a transition. Also, there can be multiple places to place such a call, so the proof is not very sensitive to this choice, and the monolithic prover can provide a counter-example to guide the developer if this call is placed incorrectly.

Encapsulating the upper-layer state into the lower layer effectively creates a single, *augmented lower layer* that can be used to reason about the relation between the upper and lower layer. Most importantly, we can now leverage traditional single-layer provers to show whether a certain property—the refinement property—holds for this augmented lower layer.

Case study: refinement proof for the SHT Algorithm 4 shows a simple example of encapsulation at the implementation layer of SHT. To perform this encapsulation, the implementation layer imports (in Ivy) the specification layer. In this example of handling a `set` request, the program checks if this node (me) is the owner of the key in the request (line 4). If it is the owner, the implementation layer internally makes a call (line 7) to *spec.commit* (shown in Algorithm 3). This transition corresponds

to the transition from state 1 to state 2 in Figure III.2: the implementation layer transitions from state 1 to state 2, while each of these states encapsulates the corresponding upper layer state, indicating a transition from state A to state B at the upper layer.

If this node is not the owner, it simply redirects the request to the owner. Such an implementation layer transition does not entail a specification layer transition and so the code does not call $spec.commit$ or any specification-level function. This is usually called a "stutterring" step of the specification layer—essentially a *no-op*—and corresponds to the transition from state 2 to state 3 in Figure III.2.

To prove that the implementation refines the specification, we ask our monolithic prover to prove a simple property:

$$\forall R : reply, N : node. \quad net.replied(R, N) \implies spec.replies(R)$$

This property says that any reply R sent to any node N at the network (implementation level) can only be present if the same reply R is present at the specification level. Since replies are the only observable outputs of the system, it ensures that every output of the implementation is also an output of the specification, thus ensuring that the implementation is indeed a refinement of the specification. Note that the reply message at the implementation layer is part of the network and thus modeled as $net.replied(M, N)$.

3.4.2 Enforcing pre- and postconditions across layers

When calling functions from a lower layer to an upper layer, an upper-layer transition's precondition must be met. The preconditions of the callee (in the upper layer) become postconditions (assertions) for the caller (in the lower layer) to check. For example, on line 13 of SHT's specification (Algorithm 3), before committing a request, the precondition concerning the request, req, and the corresponding reply, rep, must be met:

$$req.type = read \implies rep.data = map(req.key)$$

This precondition ensures that every time a read request is committed, the data contained in the response must correspond to the data in the abstract map. Since it is the caller's responsibility to guarantee that this precondition is met before committing the request, this precondition is effectively an assertion that needs to be checked by the monolithic prover. Unfortunately, the current state-of-the-art monolithic provers do not support checking these kinds of assertions, and can only find an inductive invariant for a safety property.

If we attempt to ignore this assertion check and let the monolithic prover prove the refinement

property as is, the result could be unsound—i.e., the proof may go through even if the implementation is buggy. For example, let us consider again the refinement property for SHT:

$$\forall R : reply, D : node.\ \ net.replied(R, D) \implies spec.replies(R)$$

Without precondition checks, a buggy implementation can send a bogus reply message and call $commit$ at the encapsulated specification layer. This would make the refinement property trivially inductive—since the $commit$ call adds the message to the $replies$—without guaranteeing that contents of that message are correct.

To avoid this problem, Sift needs to consider the assertions in function calls to maintain soundness in automated refinement proofs. In the rest of this section, we explain how we transform the assertions to either conditionals (if/else) or invariants that the monolithic prover can reason about.

3.4.2.1 Converting Assertions to Conditionals

A straightforward approach to model assertions in function calls is to convert the callee to an *always-enabled action* using a conditional if/else block [HHK+17]. The developer can manually rewrite an assertion P as follows: if P holds, take the transition; otherwise, do nothing. In this context, the entire if/else block is always-enabled, in that it has no preconditions and can always be taken.

For example, the original SHT specification had a precondition $\neg requests(req)$ in the specification of the $commit$ action, which we convert to an if-statement (Algorithm 3, line 14). This precondition ensures that the specification can never execute the same request twice.

The benefit of this approach is that it does not rely on any understanding of the system, which makes it very easy to implement. It has, however, two downsides. First, adding an if/else block in place of a precondition makes the proof a little harder for monolithic provers, since it is harder to find an inductive invariant for a weaker problem. Second, if the if-statement refers to ghost state—i.e., proof-related state that is not compiled to an executable—such as the sets corresponding to network messages, these if-statements are not compiled directly to executable code. Therefore, if there are any assertions that refer to ghost states at the implementation layer, we cannot rely on the approach of converting assertions to conditionals. In these cases, we need to convert them to invariants, as we describe below.

3.4.2.2 Converting Assertions to Invariants

A second, more involved approach to the problem is to convert these assertions into invariants. Doing so requires human intuition but reduces the difficulty for the monolithic prover. For every

assertion that needs to be checked, there must be an invariant to support its proof. The key idea is simple: a programmer can trace backward through a function call from the upper layer (the callee) to the lower layer (the caller) to find the enabling precondition. For the SHT precondition example above, we observe that only the node who owns the key can commit the reply. Leveraging this observation, we can construct an invariant that if node N thinks it is the owner of key K, the local value for key K at node N, which forms the reply, must match the value in the spec:

$$\forall N : node, K : key.\ server(N).delmap(K, N) \implies server(N).hash(K) = spec.map(K)$$

where $server(N).delmap(K, N)$ indicates that from the perspective of server N, the owner of key K is N ($delmap$ stands for the delegation map). By maintaining this invariant, Sift can ensure the associated assertion will never be violated during the execution of the system. Note that the invariant is not necessarily inductive, but Sift leverages the automation of the monolithic prover to complete the proof.

Case study: converting assertions for the SHT The manual effort involved in the SHT proof requires converting seven assertions into two if-statements and five invariants. The assertion $\neg requests(r1)$ is converted from an assertion to an if-statement, as described in Section 3.4.2.1. On the other hand, the first three assertions (lines 10 to 12 in Algorithm 3) are already enforced by the implementation layer and do not need to be converted. We could further use the methodology described above in Section 3.4.2.2 to convert the fourth assertion (line 13) to an invariant, but it turns out that monolithic provers are powerful enough to complete the proof even if we simply convert it to a if-statement.

3.5 Introducing Intermediate Layers

We have discussed how to use automation to prove refinement between two layers. However, sometimes, the automation provided by the monolithic prover is not powerful enough to prove the desired refinement. This can happen either due to the complexity of the proof, or the presence of undecidable reasoning. When faced with such complex proofs, monolithic provers will either time out or run out of memory.

To perform such complex proofs, the solution is to introduce an intermediate layer (step ⑤ in Figure III.1), thereby splitting the proof into two simpler refinement proofs: one refinement proof from the original lower layer to the intermediate layer, and another refinement proof from the intermediate layer to the original upper layer. By repeatedly using this proof-splitting technique

until every refinement proof is automated, we effectively execute a *divide-and-conquer strategy* that allows us to tackle complicated refinements.

This idea is similar to IronFleet's methodology of introducing an intermediate protocol layer to simplify the proof. In IronFleet, however, the developer needed to both *write* an intermediate layer and *manually prove* it correct. By contrast, Sift uses the automation of monolithic provers to dispense with most of the latter manual effort of writing the proof, and only requires the user to write intermediate layers—a much smaller effort than coming up with manual proofs.

Thankfully for developers, introducing an additional layer is done incrementally. The new layer is essentially a variation of the layer above or below it: either a more detailed version of the layer above it or a more abstract version of the layer below it. This helps keep the manual effort needed to introduce such layers small.

In the rest of this section, we discuss the strategies that we have developed and used to introduce intermediate layers, and walk through the process on a MultiPaxos example.

3.5.1 Intermediate Layers for Complexity

In most cases, the biggest challenge for a monolithic prover to automatically prove a refinement is its complexity. If the system is too complex, the prover either times out or runs out of memory. When this happens, we can split the refinement proof into two simpler refinement proofs by introducing an intermediate layer. We list here a number of ways in which such a split can simplify the proof burden. This list is extracted from our experience adding intermediate layers to facilitate refinement, and is not meant to be a complete enumeration of all possible layer-splitting strategies.

Abstract Away Messages Not Needed for Safety. Some of the messages used in the implementation may only be needed for liveness or performance, but not for safety. When trying to prove safety, those messages can be abstracted away in an intermediate layer: they are removed from the intermediate layer but kept in the implementation layer—itself proven to be a refinement of the intermediate layer.

For example, in MultiPaxos the current leader needs to periodically broadcast a heartbeat message to indicate that it is still alive. This message is not needed for safety and can therefore be removed in an intermediate layer—though it is preserved in the implementation layer. The resulting intermediate layer is now simpler and thus easier to prove equivalent to the specification.

Merge Multiple Transitions into One Abstract Transition. Sometimes, the intermediate layer can take an abstract transition which is broken into multiple transitions in the low-level implementation.

For example, in MultiPaxos the learner can only receive one vote (*two_b* message) from an acceptor at a time. But what the learner really needs is a quorum of messages to learn a value. In this case we can merge multiple transitions of receiving each message separately into one abstract transition of receiving a quorum, and remove local variables for temporary results. This significantly simplifies the intermediate layer, with fewer state variables and simpler transitions.

Simplify Local State and Requirements for Transitions. Implementation layers have to take into account implementation constraints: for example, a node can only read its local state when taking a transition; and it cannot access messages sitting in the network. But intermediate layers are essentially proof constructs and thus do not need to respect such implementation constraints.

For example, in MultiPaxos, a node needs to maintain an explicit local history of previous *two_b* votes to construct its *one_b* promise to a new leader, since a promise message depends on previous votes. In an intermediate layer however, a node can directly access all sent messages in the network, thus eliminating the need for this local history. Moreover, in an implementation a node can only read its local history, thus requiring a proof that the local history is consistent with sent messages. In the intermediate layer, since the node has access to all sent messages, it can directly check that the *one_b* promise is consistent with the vote messages, thus eliminating the need for this proof. As a result, we can introduce a middle layer which reads the sent messages instead of its local history. This layer is more abstract than the implementation, but more concrete than the specification. Thus, the refinement from the implementation to the specification can be broken into two simpler refinements.

3.5.2 Intermediate Layers for Decidability

When the verifier returns an explicit decidability error, it means our refinement is not in the EPR decidable logic [Lew80] and may take forever to check. Such an issue is typically resolved by introducing an intermediate layer and a ghost state (also known as a derived relation [PLSS17]) to hide the existential quantifier creating the undecidability [PLSS17, TLM+18]. We apply a similar technique in Sift.

For example, in MultiPaxos an acceptor needs to send its last votes for different slots in a *one_b* message to a new leader to decide what value to propose. When a proposer becomes a leader, it needs to have a quorum of *one_b* messages, resulting in the following $\forall Round\, \exists Votes$ alternation:

$$\forall N : Node, R : Round.\ \ quorum\_of(R).contains(N) \\ \implies \exists V : votes.\ \ one\_b(N, R, V)$$

The alternation of the $\forall$ and $\exists$ quantifiers, along with the inductive invariant, means that this

proposition is outside the decidable logic of EPR. We leverage results from a followup work on Ivy [TLM+18], and introduce an intermediate layer to abstract away the payload (previous votes), thereby breaking the quantifier alternation. In this case, we only need an intermediate-layer state $joined\_round(N, R)$ to represent $\exists V.\ one\_b(N, R, V)$.

3.6 Evaluation

System	Proof Effort	Refinement	# of types	Solution to Preconditions	# of Clauses in Invariant	Time (sec)	Memory (MB)
Leader Election	$<$ 5 min	spec to impl	2	1 if-statement	6	196	1744
Distributed Lock	$<$ 5 min	spec to impl	2	1 if-statement	8	111	425
Two-Phase Commit	$<$ 5 min	spec to impl	4	3 if-statements	12	613	815
SHT	$<$ 30 min	spec to impl	7	5 invariants, 2 if-statements	13	1021	856
Raft	1 person-month	spec to layer 0	6	manual			
		layer 0 to layer 1	6	15 invariants	22	787	4178
		layer 1 to impl	10	15 invariants, 1 if-statement	17	1239	2981
MultiPaxos	Previously proved	spec to layer 0	9	manual			
	3 person-weeks	layer 0 to layer 1	9	7 invariants, 2 if-statements	12	49	249
		layer 1 to layer 2	11	8 invariants, 8 if-statements	21	258	719
		layer 2 to layer 3	11	19 invariants	28	841	1935
		layer 3 to impl	14	19 invariants	25	196	398

Table III.1: Summary of our six distributed systems; "spec" stands for specification, "impl" stands for implementation, and "layer i" represents intermediate layers. The number of different types that are needed to express the state transition illustrates the complexity of different system.

We evaluate Sift by using it to formally verify the correctness of six implementations of distributed systems: a *leader election* protocol (Section 3.6.1), a *distributed lock* protocol (Section 3.6.2), a *two-phase commit* protocol (Section 3.6.3), a *sharded hash table* (SHT, Section 3.6.4), and two consensus protocols: *Raft* (Section 3.6.5) and *MultiPaxos* (Section 3.6.6). We use Ivy to implement these systems, and extract the executable code to C++ using Ivy's built-in translator. For the more complex systems (*SHT*, *Raft* and *MultiPaxos*), we also perform a performance evaluation (Section 3.6.7) to demonstrate our automated approach does not impact the performance of implementations.

For all systems in our evaluation, we consider crash failures and an asynchronous network, which can arbitrarily delay, drop, or duplicate messages. Both of these can be easily implemented in Ivy. Note that since Sift (like all its predecessors that also target automation) does not support liveness proofs, it does not need to explicitly reason about crash failures—a crash results in a machine no longer taking any steps and thus has no effect on safety properties.

We find that we are able to prove these complex systems with little manual effort within a reasonable memory and time budget, using IC3PO [GS21] as our monolithic prover. Our verification results are in Table III.1. The complexity of different systems is illustrated by the number of different types that are needed to express state transitions for a given system. For example, for the leader election protocol, there are just two types: *node* and *id*. In contrast, MultiPaxos contains 14 different types, e.g., *round*, *inst*, *value*, *time*, *node*, etc.

We now give details about the proofs of the aforementioned systems, followed with a performance evaluation (Section 3.6.7) of three of the more complex resulting implementations (i.e., SHT, Raft and Paxos). We ran our performance experiments on a cluster where nodes have a 16-core Intel Xeon E5-2667 v4 @3.20 GHz processor and are connected with a 10 GB Ethernet connection running Ubuntu 16.04. All our implementations and artifacts can be found in GitHub [MAG+].

3.6.1 Leader Election

The *leader election* protocol aims to elect a unique leader from a ring with an unbounded number of nodes with unique integer IDs [CR79, PMP+16, MGJ+19a]. The specification layer dictates a single action the system can take: elect a node as the leader, under the condition that no other node is already the leader. This layer contains 13 lines of Ivy code.

In the implementation layer, the nodes are totally ordered in a ring so that every node has a next node. A node n has two valid actions: (a) periodically send its ID $idn(n)$ to the next node in the ring; or (b) forward an ID i received from its predecessor if $i > idn(n)$. Once n receives its $idn(n)$, it knows that no other node in the system has a larger ID, and can now safely become the leader. The implementation layer consists of 28 lines of Ivy code.

To prove refinement between the implementation and the specification layers, we ensure that when a message stating that a leader is elected is sent in the implementation, the destination of the message should correspond to the leader node in the specification.

We perform a manual, albeit trivial, syntactic change to the specification layer to convert one precondition into an if-statement, which takes less than 5 minutes. We then simply use Sift's encapsulation technique to convert the refinement between the implementation and specification layers into a monolithic proof that is proven automatically by IC3PO.

3.6.2 Distributed Lock

The *distributed lock* protocol [HHK+15, PMP+16, MGJ+19a] models an unbounded number of nodes that transfer the ownership of a single lock. In this system, the ownership of a lock is associated with an ever-increasing epoch: only one node can own the lock at each epoch. This makes for a concise specification layer—12 lines of Ivy code—that only contains a lock history to

indicate which node holds the lock at every epoch.

In the implementation layer, there are two possible transitions for a node: (a) transfer the lock if it holds the lock; or (b) accept the lock and jump to a higher epoch by sending a *locked* message to indicate ownership. This implementation has 35 lines of Ivy code.

The refinement property in this system is that all *locked* messages should have a corresponding node in the specification layer's lock history.

The only manual effort involved in this proof is converting one precondition to an if-statement in the specification layer, which takes less than 5 minutes. After this transformation, we can use the encapsulation technique from Sift to convert the refinement between the implementation and specification layers into a monolithic proof, and prove the $locked$ message is equivalent to the lock history.

3.6.3 Two-Phase Commit

The *two-phase commit* protocol [Gra78] is used by a group of nodes, known as resource managers (RMs), to coordinate the decision on whether to abort or commit a transaction. The RMs vote to either commit or abort the proposed transaction and a transaction manager (TM) node is in charge of coordinating the decision-making procedure.

The specification layer of this system uses the Transaction Commit protocol by Lamport [GL06, Sec. 2] translated from TLA+ [Lam02] to Ivy. The safety property does not allow a node to commit if another node aborts. The specification contains 54 lines of Ivy code.

The implementation of this system is an Ivy translation inspired by the TLA+ specification of Two-Phase Commit [GL06, Sec. 3]. This layer introduces a special TM node, which coordinates all RMs. An RM can send a *Prepared* message to the TM when transiting into the *prepared* state, or unilaterally decide to abort. Upon receiving a *Prepared* message from every RM, the TM can decide to commit, broadcasting a *Commit* message to every RM node. The receipt of a *Commit* message from the TM allows an RM to decide to commit the transaction. This implementation of two-phase commit has 110 lines of code.

The refinement property between the implementation and specification ensures that all RMs commit or abort at the same time between the implementation and the specification.

After a trivial syntactic change converting preconditions to three if-statements in the specification layer, this refinement property is proven automatically.

3.6.4 Sharded Hash Table (SHT)

The Sharded Hash Table protocol was previously introduced as a running example in Section 3.3. Its specification is a simple key-value map processing read and write requests. We can automatically

prove the refinement from the implementation to the specification, after converting preconditions to five invariants and two if-statements to guide IC3PO, as detailed in Section 3.4.2. Compared to IronKV (IronFleet's implementation of SHT), we simplify the delegate messages by transferring one key at a time. Transferring intervals of keys would require a loop iterating over keys and a loop invariant [YRW+20, RWY+20], which cannot be found automatically by IC3PO.

The network interface for SHT is more complex than that of other systems. In particular, SHT's network interface requires that messages are not delivered twice, so that requests can only be committed once and only one node at a time can own a key. As this is not part of refinement, we leverage an existing proof [McM] for these requirements.

3.6.5 Raft

Raft [OO14] implements a shared log among nodes, which can be used to implement a fault-tolerant distributed service. The log is maintained as a set of (index, value) pairs.

Raft is a $term$-based protocol. In each term, a node can be elected as the leader, append values to the log, and replicate its log to other nodes by sending an $append$ message. For safety, each node maintains its own log and only votes for a leader whose log is not earlier than its own. When the leader receives reply messages for its $append$ message from a majority of nodes , the leader can consider all previous log entries committed. This strategy ensures that all future leaders contain the committed log.

At the specification layer, Raft can commit a prefix to an index in the leader's log and ensure that only one value is committed at each index. The refinement property from the implementation to the specification ensures that they have the same log.

3.6.5.1 Intermediate Layers and Proof Effort

Our Raft implementation is similar to the previous Ivy implementation of Raft [TLM+18] with 212 lines of code. Due to undecidability, we could not refine the implementation to the specification directly. Instead, we build a first intermediate layer—layer 0—to separate the quantifier alternation (as outlined in Section 3.5.2). We tried to prove the refinement from specification to layer 0 automatically, but the inductive invariant contains complex quantifier alternations, which IC3PO was unable to handle. As a result, we manually prove the refinement from specification to layer 0. The refinement from spec to layer 0 took two person-weeks (including understanding the protocol). Layer 0 contains 143 lines of code.

From layer 0, the implementation is still too complex to refine directly using IC3PO. We introduce another intermediate layer, layer 1, to help IC3PO automatically prove the refinement. To write layer 1, we follow the strategies presented in Section 3.5, specifically by merging actions

into one abstract action. In the abstract action a node can receive a quorum of messages at once, rather than receiving each of them individually in separate transitions. Layer 1 changes 57 lines from layer 0. We spent another two person-weeks to identify this intermediate layer and debug our implementation.

Overall, we were able to complete the proof of Raft in one person month, which compares favorably to the three person months needed by the original proof [TLM+18] written in Ivy. This reduction was the result of using a much higher degree of automation, by splitting the proof into layers and leveraging the power of IC3PO to prove each refinement between consecutive layers.

3.6.6 MultiPaxos

MultiPaxos [Lam98, Lam01] is a common consensus protocol that is widely used in industry (e.g., Chubby [Bur06], Megastore [BBC+11], and Spanner [CDE+12]). However, MultiPaxos is notoriously complex and difficult to verify.

At the specification level, *MultiPaxos* maintains an array of values; some that have been decided (i.e., agreed upon and finalized) and some that are empty. The only possible transition in the specification is to add a new decided value to this array. Similar to Raft, our refinement ensures that the implementation maintains the same values as the array in the specification.

The implementation of MultiPaxos is very similar to that of Raft but uses different strategies to ensure safety. In Raft, the leader can only be a node with the most up-to-date logs, while MultiPaxos relies on the messages from other nodes to generate an up-to-date log for the new leader.

3.6.6.1 Intermediate Layers and Proof Effort

Our design of the MultiPaxos protocol is inspired by previous work on expressing Paxos and MultiPaxos in the EPR decidable logic [PLSS17, TLM+18]. Our evaluation uses the MultiPaxos implementation from [TLM+18], removing certain re-transmissions that are unnecessary for safety to simplify the refinement.

Since proving refinement directly between the implementation layer and the specification layer would introduce undecidability (see Section 3.5.2), we initially introduce a single intermediate layer, layer 0, to circumvent this undecidability. Moreover, as the refinement from the specification to layer 0 contains complex quantifier alternations that are too hard for IC3PO to prove automatically, we borrow the existing manual proof from Ivy. Layer 0 contains 88 lines of Ivy code.

After addressing undecidability concerns through layer 0, we found that a direct refinement from the implementation to layer 0 remains infeasible for IC3PO. Using our divide-and-conquer technique, we added three additional intermediate layers to simplify this refinement. Following the strategies outlined in Section 3.5.1, we first added a layer 1 that abstracts away liveness messages

and merges transitions to receive a quorum of messages. We augmented this by introducing a layer 2 that uses a local variable to track the current round, receives one *two_b* message, and keeps track of when a valid quorum can be formed. We then introduced a final intermediate layer that more closely resembles the implementation by using an array to track previous voted values for acceptors, and restricting a node to only receive one message during a transition.

With the addition of the four intermediate layers, Sift splits the complex refinement proof into manageable pieces, where each refinement between layers is amenable to automated verification. Producing the three intermediate layers (layers 1, 2, and 3) and converting the necessary preconditions to invariants is still a non-trivial task which takes about two person-weeks. About one third of the time is spent waiting for IC3PO to run out of time or memory, which indicates that another layer is needed (step ⑤ in Figure III.1). While non-negligible, this manual effort is significantly less than the original attempt in Ivy, which was two person-months to refine layer 0 to the implementation [TLM+18].

3.6.7 Performance Evaluation

3.6.7.1 SHT Performance

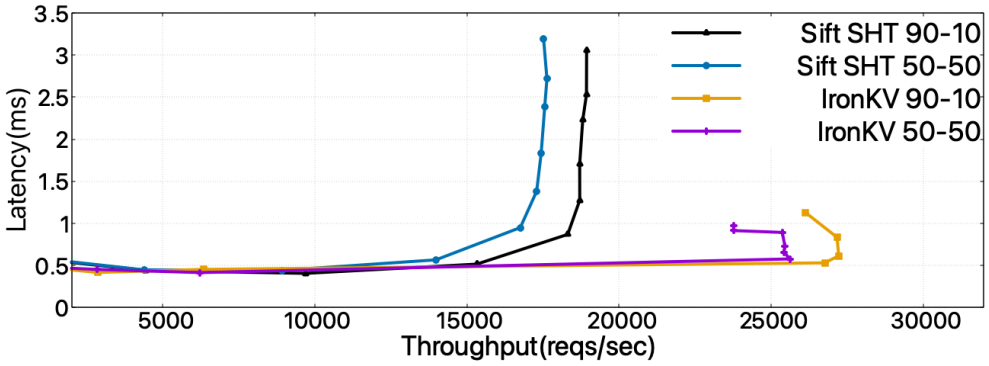

Figure III.3: Sift SHT performance evaluation

We compare the throughput and latency of our verified Sift implementation of SHT with IronKV [HHK+15], as shown in Figure III.3. IronKV is the closest verified implementation of a SHT that we could compare against. The SHT cluster was preloaded with 1,000 keys delegated evenly across the three nodes and serviced requests from an increasing number of clients in a closed loop. In one experiment, client processes send an even 50/50 mix of randomized GET and SET requests. We further increase the percentage of GET requests to 90%. IronKV scales about 25% better than our version of SHT. The disparity in performance between these two systems can be attributed to both unoptimized generated C++ from Ivy and design choices made in IronKV, which

added extra manual proof complexity for the sake of performance purposes, such as an efficient delegation map data structure that each node maintains and consists of 833 lines of Dafny code.

3.6.7.2 Raft and MultiPaxos Performance

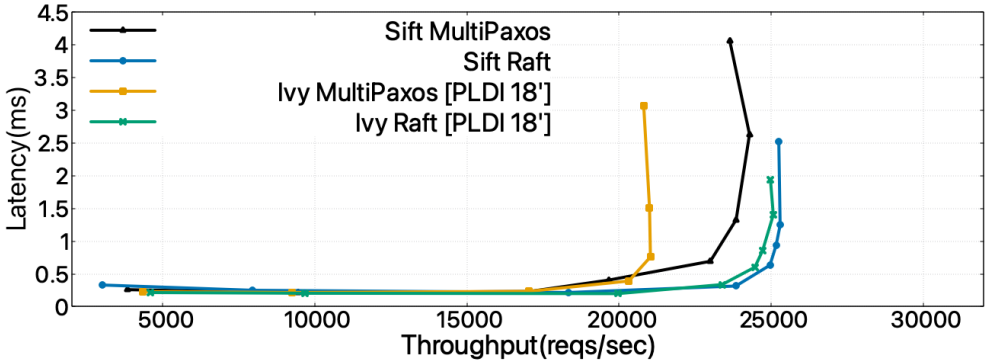

Figure III.4: Sift Raft and MultiPaxos performance evaluation

We evaluated the performance of the verified Sift implementations of Raft and MultiPaxos by varying the load of each system with an increasing number of clients submitting requests in a closed loop, as shown in Figure III.4. For both systems, the experimental setup consists of three replicas on separate machines, with a fourth machine containing the client processes.

We tried to compare the performance of these systems with IronRSL, IronFleet's verified Paxos-based replicated state machine library, but the performance results of IronRSL were not reproducible for a direct comparison. By re-running the original implementation from the IronFleet paper [HHK+15], we found the performance for IronRSL to be lower than originally reported [HHK+15, Sec. 7][1]. The performance of our implementation of MultiPaxos, which does not support batching, is comparable to the results reported for IronRSL in non-batch mode [HHK+15, Fig. 13].

We do compare both Raft and MultiPaxos with the Ivy-based manually-verified implementations [TLM+18]. The performance of Raft is almost identical to the version of Raft, but we find that our MultiPaxos system exceeds the performance of MultiPaxos from that work. We think the main reason is that we simplified some unnecessary transitions, such as periodical retransmit of previous *two_a* messages for undecided slots. These results show that the automation and reduced proof effort gained by using Sift has no negative impact on the performance of either system.

[1]Even after close discussions with two of the IronFleet authors, this discrepancy was not resolved. They attributed this to possible code changes between what was originally evaluated and the currently available code.

3.7 Limitations and Future Directions

Our experience with Sift suggests that it advances what is possible in the realm of automated verification of complex systems. For all of its successes, however, there are still more steps to be taken in this direction.

• **Automating simple transformations.** While Sift greatly increases the automation of complex refinement proofs, parts of the methodology still require manual effort that could potentially be automated, such as converting assertions to if-statements and transforming to invariants through automatic computation of weakest preconditions [Dij75].

• **Loop invariants.** Certain complex systems, such as SHT, may require loop invariants to prove optimizations that are added to enhance the performance of executable code. Loop invariants are similar to regular inductive invariants, in that both are inductive under some transitions. As described in Sections 3.6.4 and 3.6.6.1, any loop invariant in Sift must currently be written manually. In the future, we hope to add support for automatic deriviation of loop invariants in Sift, by building further on the existing literature [YRW+20, RWY+20].

• **Leveraging multiple monolithic provers.** As shown in recent works [GS21, HHMP21, YTG+21], different monolithic provers show complementary strengths in different scenarios. Since the design of Sift is independent of the choice of monolithic prover, we plan to employ a portfolio of monolithic provers in parallel to derive refinement proofs with even higher scalability.

3.8 Conclusion

This chapter introduces Sift, a novel two-tier methodology that combines the power of refinement with the ability to automate proofs. Sift decomposes the proofs of complex distributed implementations into a number of refinement steps, each of which is amenable to automation. We use Sift to prove the correctness of six distributed implementations—including the notorious MultiPaxos—none of which had an automated proof before. Our evaluation shows that this combination of refinement and automation lets us verify complex distributed implementations with little manual effort.

CHAPTER IV

Automated Verification of Efficient Implementations

With refinement proofs in Sift, more complex systems can benefit from proof automation with a small amount of manual effort. But this approach still relies on Ivy to generate the executable C++ code from the implementation. To provide more freedom for the developer implementing a verified distributed system, Cruiser introduces automation into traditional theorem provers to verify complex distributed systems. Comparing to existing verification approaches, Cruiser identifies which part of the implementation requires human knowledge from the developer, and asks the developer to provide only the necessary information at the protocol layer. After that, Cruiser can automatically generate a heap-based efficient implementation from such a protocol layer.

Compared to the predecessor, IronFleet, Cruiser requires the developer to spend a little more effort to specify the function to generate the next state from the current state at the protocol layer. However, introducing such a protocol layer can benefit the verification of a distributed system in multiple ways. First, such a protocol layer is very close to the model under Sift (with the help of decidable logic from Ivy), which means we can leverage the refinement proof from Sift to simplify the effort needed to verify the safety of such a protocol. Also, once the developer models all protocol transitions in functions, Cruiser is now capable of generating an efficient heap-based implementation automatically. Even more, the implementation Cruiser generates is very close to the functions the developer implements at the protocol layer. Thus, the developer can easily understand the implementation and even apply further optimizations.

Overall, we make the following contributions in this chapter:

- We introduce a new verification framework with a function-style protocol layer, which guides the developer to implement and prove their protocol.
- Under the new verification framework, we present a novel approach to automatically generate an efficient heap-based implementation and its refinement proof.
- We implemented our algorithm in Cruiser, and evaluated its effectiveness with three distributed implementations. We found that it allows us to prove the correctness with less manual effort, and the generated implementations reach good performance.

In this chapter, we explain the details of this methodology. Section 4.1 tells why decidable logic

is inefficient, and Section 4.2 provides some background for our baseline system. Section 4.3 gives an overview of Cruiser. Section 4.4 introduces the new protocol layer in our verification framework of Cruiser and Section 4.5 shows how to generate the implementation from our function-styled protocol layer. Section 4.6 evaluates the effectiveness of using Cruiser to verify a number of distributed implementations. Section 4.7 presents the limitations of Cruiser and discusses future work, and Section 4.8 concludes this chapter.

4.1 Decidable Logic Is Not Enough

A lot of research focuses on making automation for formal verification. Ivy uses an extension of EPR [PdMB10] to reduce the manual effort when writing the inductive proof. I4 and Sift also follow this principle and use Ivy and decidable logic to automate the process of verifying a distributed system. Unfortunately, such automation can restrict the developer from performance for a number of reasons.

First of all, decidable logic has poor support for the heap. Real programs perform complex heap manipulation for the sake of performance. Such programs are commonly modeled by separation logic [Rey02]. Unfortunately, how to use decidable logic with separation logic is still an open problem. Although some research combines linked-list with decidable logic to automate the verification [BCO04, PWZ13], with existing tools, we don't know how to use heap to optimize the performance of arbitrary systems with automation from decidable logic.

To use efficient heap data structures, Ivy chose to use trusted interfaces for C++ classes. In that design, the developer can specify the functionality of an interface, and trust the implementation matches that specification. This is an efficient design for simple systems. But when it goes to even moderate systems with real data stored, the trusted specification becomes a hidden bomb for the safety of the whole system. For example, when we were using a native vector from C++ STL, the Ivy trusted interface didn't model the update correctly, and that ended up with an incorrect proof for both Raft and MultiPaxos. A potential solution to this could be a model checker to validate that the C++ implementation matches the Ivy specification. But implementing such a model checker for C++ STL, or even arbitrary user-defined classes can still be challenging, and increases the trusted computing base for the proof. As the system becomes more complex, the chance of introducing an incorrect specification of trusted data structure would be larger, and the correctness of the whole proof becomes undermined.

Besides the trade-off between the heap data structure and the size of the trusted computing base, EPR does a poor job with arithmetic. In Ivy, it's really hard to model any operation of integers: such an operation would result in a function from integer to integer, which causes a self-loop, and breaks the decidability. To bypass this problem, Ivy provides a relation to get the successor of an

integer. For more complex arithmetic, such as multiplication and division, it's really hard to model them in Ivy.

Finally, automation from tools like Ivy can also be a limitation for some convenient coding styles. To automate the process of reasoning about a function call, Ivy automatically inlines the callee into the caller. In this way, the developer doesn't need to specify any pre- and post-conditions for a function, thus reducing the task when writing a proof. But this eliminates the possibility of recursion completely, which is common when implementing a system efficiently, because Ivy couldn't inline a function to itself. Sometimes, the developer can use a while loop to simulate the stack and perform the recursion. But some algorithms, such as the depth-first search, can be difficult to implement with a loop. This solution contradicts the intention of reducing manual effort.

As a result, we found that the implementation from Ivy usually shows lower performance comparing with manual proofs like IronFleet (as shown in Figure III.3). And this performance gap couldn't be easily filled by manual optimizations and proof. Optimizing the heap to store either the delegation or the actual hash table requires trusting a new data structure.

When we turn our eyes to the successful verification of highly optimized implementation, IronFleet [HHK+15, HHK+17], we noticed that they use a three-layer structure to prove their high-performance implementation in Dafny, which still relies on the developer the provide the whole proof. They insert a protocol layer between the implementation and the specification, where the protocol uses a functional manner and immutable data structures so that the developer can easily reason about the inductive invariant and refine the state to the specification. After that, they show that the implementation meets the protocol.

We noticed that the first refinement is very similar to Sift, where everything is modeled with the immutable datatype. Thus, we can transform our approach and inductive invariant from Sift to IronFleet. However, the second part is still an open problem. There are not any existing research answers on how we can automate the process of refining an efficient implementation into an abstract protocol. Even worse, there is even no clear guidance about how we should build such a protocol layer.

To simplify the task of verifying complex implementations, we propose a new framework, Cruiser, to generate the implementation and its proof in Dafny. In Cruiser, we specify how the developer should implement the protocol layer with the help of immutable datatype, and then generate a heap-based implementation along with its proof automatically.

4.2 Background: IronFleet Style Refinement

In this section, we introduce our predecessor, IronFleet style refinement, which uses Dafny to verify a high-performance distributed system.

IronFleet uses a three-layer refinement to show that the implementation is equivalent to a specification. At the specification layer, the developer defines a centralized state machine with two predicates: its initialization *ServiceInit(s, servers)* and transition *ServiceNext(s, s'). ServiceInit(s, servers)* returns true when state *s* is a valid initial state under a given set of *servers*, and *ServiceNext(s, s')* is true when current state *s* has a valid transition to a next state *s'*.

Algorithm 5 Implementation definition in IronFleet

```
1  type HostState
2  predicate HostInit(s:HostState, config:seq⟨EndPoint⟩, id:EndPoint)
3  predicate HostNext(s:HostState, s':HostState, ios:seq⟨NetEvent⟩)

4  method HostInitImpl(ghost env:HostEnvironment, netc:NetClient, config:Config)
5    returns (ok:bool, s:HostState)
6      requires ...
7      ensures  ok ⟹ HostInit(s, config, netc.myID)
8      ensures  ...

9  method HostNextImpl(ghost env:HostEnvironment, s:HostState)
10   returns (ok:bool, s':HostState. ghost ios:seq⟨NetEvent⟩)
11     requires ...
12     ensures  ok ⟹ HostNext(s, s', ios)
13     ensures  ...
```

To perform an end-to-end refinement for the implementation, IronFleet defines the refinement property as the message correspondence directly from the implementation to the specification, the service layer. Thus, IronFleet only provides a framework for the implementation, but not the protocol layer. In the implementation layer, IronFleet separates the proof predicate from the actual execution method. As shown in Algorithm 5, The developer first define the initialization *HostInit(s, config, id)* and the transition *HostNext(s, s', IOs)* as the predicates used in the proof, and then shows that the implementation *HostInitImpl* and *HostNextImpl* satisfies these predicates.

To refine the implementation layer to the specification layer, IronFleet defines the *ServiceCorrespondence* predicate to validate the correspondence between the network packets and the service state. More specifically, IronFleet proves that given a valid behavior of the system state, there is a corresponding behavior of the service state (Algorithm 6). To prove such a refinement, IronFleet first extracts the implementation layer into a protocol layer. Such a protocol layer needs to be simple and abstract, which can be easily proved. For example, IronFleet protocol uses unbounded mathematical integers, immutable types, etc. Within this protocol layer, IronFleet performs an inductive proof: using a set of inductive invariants to show that the protocol messages match the specification. Once the *ServiceCorrespondence* between the real packets and the specification can be proven, there is no need to trust the protocol layer.

Algorithm 6 Refinement proof in IronFleet

```
1 lemma RefinementProof(servers: seq⟨EndPoint⟩, dp: seq⟨DState⟩)
2   returns (sb: seq⟨ServiceState⟩)
3     requires |dp| > 0
4     requires ProtocolInit(dp[0], servers)
5     requires ∀ i. 0 ≤ i < |dp| - 1 ⟹ ProtocolNext(dp[i], dp[i+1])

6     ensures  |dp| == |sb|
7     ensures  ServiceInit(sb[0], servers)
8     ensures  ∀ i. 0 ≤ i < |sb| - 1 ⟹ sb[i] == sb[i+1] ∨ ServiceNext(sb[i], sb[i+1])
9     ensures  ∀ i. 0 ≤ i < |sb| ⟹ ServiceCorrespondence(dp[i].sentPackets, sb[i])
```

Besides the safety property (the refinement from the implementation to the specification), IronFleet also provides a liveness proof of the system. More specifically, IronFleet proves that if a client sends a request, and the service is running correctly, eventually there will be a reply for the request. Such a proof requires a combination of TLA behavior and fairness assumption with always-enabled actions. As a result, all these liveness proof operates on the untrusted protocol layer.

Despite the satisfying properties IronFleet can prove, IronFleet heavily relies on the developer to complete all the proof, which requires a lot of manual effort.

4.3 Overview of Cruiser

To simplify the effort in proving a high-performance implementation like IronFleet, Cruiser first provides a new protocol layer to specify and implement the protocol in functions. The developer needs to manually specify the protocol layer, and prove the refinement from the protocol to the specification. After that, Cruiser can generate the implementation for the protocol automatically, along with the refinement proof.

For the first part, Cruiser introduces a protocol layer into the verification framework. As IronFleet performs the end-to-end refinement from the implementation to the specification, there is not a clear framework on implementing the protocol layer in between. With Cruiser, our refinement goes from the protocol to the specification. Thus, we added a new environment for the protocol layer, constructing a distributed state machine, which can be refined to the specification state machine. For the simplicity of the proof, we leave all states in the protocol as immutable datatypes.

After the developer has proved the refinement from the protocol to the specification, Cruiser can automatically generate an efficient implementation and its proof. Here, we reused the network abstraction and the implementation model from IronFleet. As we don't want the developer to have an ultimate understanding about all the details of the implementation and its relation to the protocol, Cruiser automatically generates the correspondence relation between the implementation

and the protocol. We prove that for all messages in the real implementation, the protocol also contains the abstracted message; for all messages in the protocol, there exists a message in the real network which can be abstracted to it. In this way, the only thing that needs to be trusted in the implementation is simply the abstract function from the network packets to abstract protocol packets.

4.4 Adding the Protocol Layer

From Section 4.3, we've had a glance at how to use Cruiser to generate and verify a distributed implementation. In this section, we'll introduce the design principle of our protocol layer and how to write a proof for it under the new framework.

4.4.1 Refinement from Protocol to Specification

The first and manual step in Cruiser is to prove the refinement from the protocol to the specification. To show this, the developer needs to implement two state machines: a simple, concise state machine for the specification of the target system, and a detailed state machine for the protocol layer. Our specification layer is the same as IronFleet. The developer defines the service state, the initialization of the state machine (*ServiceInit(s, servers)*) given the set of server addresses and the transition (*ServiceNext(s, s')*) between the current state s and the next state s'.

Similar to IronFleet and Sift, Cruiser uses the outgoing messages to define the refinement property, so that the developer doesn't need to trust the refinement function to lift the state of the lower layer to the upper layer. In the refinement from the protocol to the specification, as the *ServiceState* doesn't use the network to generate observable outputs, the refinement property (*ServiceCorrespondence(set⟨abstractPkt⟩, serviceState)*) becomes a relation between the set of packets at the protocol layer and the service state.

After defining the specification of the target system, Cruiser provides a framework for the developer to implement the protocol layer in a function manner. Algorithm 7 shows the interfaces at the protocol level.

The developer first defines the state in each host in a datatype *Node* at line 1 and a predicate *NodeValid* to validate the wellformedness of the state in line 2. Such validation could simplify some implementation checks. For example, we can ensure that all values in the hash table would not exceed the length restriction. After that, the developer provides how to initialize such a valid state from a *Config* in line 3. Such initialization function requires the config to be valid, and ensures the initialized return node to be valid.

After the initialization, the developer defines the state transition for the node in line 7. Such transition takes four parameters: a constant *Config*, the current state *s*, an optional *recv* packet, and

Algorithm 7 Protocol definition in Cruiser

```
1  type Node
2  predicate NodeValid(config: Config, s:Node)

3  function NodeInit(config: Config) : (ret: Node)
4      requires ConfigValid(config)
5      ensures  NodeValid(ret)

6  predicate ValidTransition(s:Node,
                             recv: Option⟨LPacket⟨EndPoint, Message⟩⟩,
                             clock: Option⟨uint64⟩)
7  function NodeNext(config: Config,
                     s: Node,
                     recv: Option⟨LPacket⟨EndPoint, Message⟩⟩,
                     clock: Option⟨uint64⟩)
           returns (s' : Node,
                    outgoingPkts : seq⟨LPacket⟨EndPoint, Message⟩⟩)
8      requires ConfigValid(config)
9      requires NodeValid(config, s)
10     requires ValidTransition(s, recv, clock)

11     ensures  NodeValid(config, s')
12     ensures  ∀ pkt ∈ outgoingPkts. pkt.src == config.myEndPoint
13     ensures  ∀ pkt ∈ outgoingPkts. MarshallableMessage(pkt.msg)
```

an optional *clock*. In such transitions, we only allow the developer to receive at most one packet and read the clock at most once. The restriction about receiving packets comes from the implementation. In the implementation, a node can only receive one packet and then takes the transitions. If it wants to wait for the next, the unreliable network may never deliver that packet. As for the clock, reading the clock is a non-mover in reduction, which means it can not commute with any other execution in remote processes. Thus, reading the clock twice violates the reduction proof, and the transition is not atomic anymore.

Typically, an implementation doesn't always rely on an incoming packet and the clock to take the transition. To be compatible with both cases, we chose optional values to present the received packet and the clock. If this transition receives the packet, there is a packet in *recv*, otherwise, there is no value in it. The same applies to *clock*. However, this model requires the developer to explicitly specify when there is a value in *recv* or *clock*. We provide a predicate *ValidTransition* to do that.

There are other restrictions to ensure the transition is valid. First, the starting state, *s*, must be a valid node. As a result, the end state, *s'*, should also be a valid state. Also, the provided constants *config* should be a valid config. For the outgoing messages *pkts*, there are two main

constraints. First, all outgoing packets should have the same source as the *myEndPoint* in the *config*. Such restriction ensures that a node can never "pretend" to be another node and send a message, which is also required in the implementation. Another restriction comes from the implementation directly: all outgoing messages must be marshallable. As we need to marshal such messages in the implementation, all outgoing messages need to be marshaled to an array of bytes, which can be sent through the network. Thus, we have some constraints, mostly the constraint about the size of the message, to ensure this packet can be marshaled and sent to the real network.

To show the refinement from the protocol to the specification, we need to make such protocol implementation distributed to multiple machines and interact with each other. Here, we follow the idea of IronFleet to create a global view of the system. In this global protocol state, we have a map from each node to their *config* and *node* state. Also, we maintain an environment to keep track of the clock time and all network messages. At each step, the hosts can pick one host and perform the transition. However, the host may not always take a transition specified by *NodeNext*, as the receive event may not always return a packet. Sometimes, the node may still try to receive a packet, but there may not be any packet to receive. In this case, the node can not perform a normal transition. Instead, we model a special transition when receive timeout happens. In that case, the node can simply stay in the previous state and do nothing.

Now that we have implemented the two state machines for the specification and the protocol. Cruiser still relies on the developer to prove the refinement from the protocol to the specification manually. We use a similar lemma as IronFleet in Algorithm 6 to show that given a random execution of the protocol, it finds an execution of the service state such that the correspondence holds between the protocol packets and the service state.

This refinement proof is the most challenging part of writing a proof. As we mentioned in Section 4.1, such refinement proof is very similar to Sift. As a result, the developer can leverage some automation from Sift to simplify the work. More specifically, we noticed that all inductive invariants from Sift also work in Cruiser, and these invariants are mostly inductive. We only need to add several Dafny-specific invariants to maintain the data structure to make it inductive. Although the inductive invariant is not difficult to find, Dafny can not automatically validate these invariants. As a result, we still spend manual effort writing lemmas to prove each invariant is inductive.

4.4.2 Refinement from Implementation to Protocol

Once we have a proven correct protocol in Cruiser, the next thing is to automatically generate the implementation and the proof. In this section, we focus on the proof part and how we show our generated implementation is provably correct.

The implementation model is also similar to the existing framework in IronFleet (Algorithm 5). We separated the executable implementation (*HostInitImpl* and *HostNextImpl*) from the proof model

Algorithm 8 Implementation refinement proof in Cruiser

```
1  predicate ProtocolCorrespondence(concretePkts:set⟨NetPacket⟩,
                                    abstractPkts:set⟨LPacket⟨EndPoint, Message⟩⟩)
2  {
3        (∀ cmsg ∈ concretePkts. AbstractifyNetPacket(cmsg) ∈ abstractPkts)
4     ∧ (∀ amsg ∈ abstractPkts. ∃ cmsg ∈ concretePkts. AbstractifyNetPacket(cmsg) == amsg)
5  }

6  lemma RefinementProof(servers: seq⟨EndPoint⟩, db: seq⟨ImplState⟩)
7    returns (pb: seq⟨ProtocolState⟩)
8      requires ...

9      ensures  ...
10     ensures  ∀ i. 0 ≤ i < |pb| - 1 ⟹ ProtocolNext(pb[i], pb[i+1])
11     ensures  ∀ i. 0 ≤ i < |pb| ⟹ ProtocolCorrespondence(db[i].sentPackets,
                                                           pb[i].sentPackets)
```

(*HostInit* and *HostNext*). Instead, the implementation proves shows that these predicates hold as the postcondition.

Similar to the protocol layer, we also need to define a global view of states to keep track of all hosts in this system, the set of all sent messages, and the clock time in the implementation. With this global state, we can define the correspondence and the refinement proof in Algorithm 8. As this is an automatically generated proof, we couldn't pick which messages are outgoing replies and which messages are incoming requests. Thus, we chose to show that no matter which messages are in the implementation, the protocol has the same message (in the abstracted style). For all messages in the protocol, there exists the same message in the implementation. This leads to the last manual part of Cruiser: a trusted function to abstractify the *NetPacket* into *Messages*.

Abstracting a *Message* from real *NetPacket* relies on the design and implementation of the message marshalling strategy. For now, Cruiser still relies on the developer to provide such strategy and implementation to marshall messages and abstract messages from real *NetPacket*. More specifically, the developer provides a set of trusted functions to parse a *NetPacket* into a *Message*, and then implements two methods. For the marshall method, the return *NetPacket* must be parsed to the input *Message*. For the parse method, the return *Message* must equal the result of the trusted parse function. We used existing marshalling libraries from IronFleet to implement our marshalling.

Given such a correspondence relation for packets, the refinement proof (line 6) from the implementation to the protocol can be introduced. Similar to the refinement proof from the protocol to the specification (Algorithm 6), this lemma takes a valid implementation execution behavior and generates a corresponding protocol execution. Comparing to the protocol refinement proof, this refinement proof has two differences. First, the correspondence between the implementation

and the protocol is the given *ProtocolCorrespondence* between the set of network packets in the implementation and the set of abstract packets in the protocol. Second, the generated protocol state is a one-to-one mapping to the implementation state, which means the implementation would not have a stutter step in the transition.

Such one-to-one mapping eliminates the potential violation of the liveness proof. Recall that IronFleet liveness proof operates on the untrusted protocol layer, there is no guarantee that the implementation corresponds to the protocol. Even if the implementation refines the protocol, the liveness can still be broken without this one-to-one mapping. For example, if the implementation can take a transition, but this transition doesn't change the corresponding protocol state, the implementation can take this transition infinitely times, but the protocol state stays the same, and makes no progress. As a result, the liveness in the protocol layer doesn't guarantee the liveness of the implementation. With the one-to-one mapping in Cruiser, once the developer has proved the liveness of the protocol, the implementation is guaranteed to closely follow the progress of the protocol and make progress.

4.5 Implementation Generation

We have built the proof framework for both the protocol and the implementation in Section 4.4. In this section, we introduce more details on how we can automatically generate the heap-based implementation from a protocol.

4.5.1 Making the Protocol Implementable

The main problem we're thinking about during the design of Cruiser is how to guide the developer in writing an implementable protocol. We found that writing the protocol in functions for transition is better than a simple predicate for the current and the next state, which is used in IronFleet. This is analogous to the P versus NP problem. In the implementation, the host needs to know how to find a valid next state from the given existing state, which is similar to finding a solution to a given problem. However, if the developer uses predicates to specify this transition, this problem becomes providing two states, and how to validate it is a valid succeeding state. For example, the developer can define the state transition as integer factorization: starting from an integer, a valid succeeding state contains two integers whose multiplication must equal the integer in the previous state, but we don't know how to efficiently generate an implementation for such a transition. It is clear it's an NP problem, so the developer can implement a predicate to define such transitions. But there is no efficient P solution to find the answer, thus we don't know how to generate the implementation.

As a result, Cruiser asks the developer to specify all transitions in functions to simplify this

problem. For example, the *NodeNext* transition (Algorithm 7 line 7) can be a function of current state, network input, and returns the next state and network output. The developer needs to specify clearly how to use the current state to generate the next state. Thus, we can further optimize this implementation with heap and in-place updates.

4.5.2 Overflow Protections

Another important design in Cruiser is to ask the developer to explicitly handle overflow with correct data types. In most implementations, the behavior when an overflow happens is undefined. Thus, the implementation Cruiser generates needs to avoid any overflow in the system. Thus, the protocol needs to use the same types for the implementation to specify the state in each node. If the protocol is still using the mathematical integers, which do never overflow, Cruiser can only generate an implementation with *BigInteger*, which also doesn't overflow. Otherwise, if Cruiser uses a sized integer, the implementation couldn't automatically infer how to handle such overflows.

Comparing to IronFleet using mathematical integers which never overflow, Cruiser may require the developer to take extra effort to handle the overflow manually. However, we found this is still a reasonable amount of effort. Although IronFleet claims it is useful to simplify the state to mathematical integers which never overflow, we found they still handle overflow explicitly when they design their protocol. The reason is that such overflow protection is necessary for the implementation to use efficient-sized integers, and the liveness proof relies on a one-to-one mapping from the implementation to the protocol. If the protocol doesn't handle overflow explicitly, and there is an overflow in the implementation, there is no corresponding transition in the protocol. If the implementation decides to take a no-op, the new state refines to the origional protocol state, which violates the one-to-one mapping, and may break the liveness proof.

4.5.3 Making IOs Capable With Implementation

The protocol layer and the implementation layer have a different model for all their IOs, which needs to be taken care of. In the protocol layer, all IOs are given as input parameters, thus the function can just read such an IO sequence and find necessary receive and clock events. However, the implementation doesn't have such a sequence. Instead, all IOs happen through a *NetClient*, which provides trusted C# functions. Such *NetClient* contains a history of all IOs the implementation performs.

The main problem in connecting the implementation IOs and the protocol IOs is that the protocol may access the IO at any point in the program, but accessing IO in implementation takes the *NetClient*. If we simply take the *NetClient* everywhere in the implementation and perform the IO lazily, there is no corresponding transition if there is no packet to receive (i.e. receive timeout).

An alternative solution is to perform the IO at the beginning of the transition. Thus, it is guaranteed that there will be a packet, and a clock time to process, and the timeout can be caught before any transition happens. But this solution performs too many unnecessary IOs, thus significantly influencing the performance.

As discussed in Section 4.4.1, we choose to use optional value to solve this problem. At the protocol level, the developer can use optional packet as *recv* and optional integer as *clock*. If they need to use such a value in a transition, the developer can access its value. To match the receive timeout behavior specified in the protocol, we need to ensure that a timeout check could happen before any state transition. More specifically, we check each if-branch to see if *recv* or *clock* is used in this branch. If it is, we can perform IO accordingly, and check if there is a receive timeout at the beginning of this branch.

After the state transition, the *NodeNext* function (Algorithm 7 line 7) returns a sequence of outgoing packets to be sent. The implementation would also capture such a sequence of packets. As they are abstract packets, the implementation needs to marshal them into network packets and send these packets one by one.

4.5.4 Generating Corresponding Implementation

Given a protocol layer with Cruiser framework, the next step is to automatically translate such implementation into heap-based implementation for higher performance.

The first problem in such a translation is related to the proof. To refine the implementation to the protocol, the proof framework requires an execution trace of all execution states. However, we're aiming for a high-performance heap-based implementation, where all updates happen just in place. To solve this problem, we follow the idea in IronFleet, introducing a ghost state with a copy of all fields in the implementation to keep a record of all executions in the execution. Such ghost states are exactly what's used in the protocol state, as all field in the implementation has a corresponding field in the protocol *Node*. Thus, we can just have a ghost copy of the protocol state in the implementation for the proof of refinement. This copy of protocol state also makes the proof model, *HostInit* (ALgorighm 5 line 2) and *HostNext* (Algorithm 5 line 3), easy to implement – they performs the same transition as *NodeInit* (ALgorighm 7 line 3) and *NodeNext* (ALgorighm 7 line 7).

Now, we can translate the protocol state to a class, which can be efficiently updated. This translation is mostly mechanical. For example, in distributed lock protocol mentioned in section 2.5.3 and 3.6.2, each node only keeps two local variables: a Boolean *held* indicating whether this node holds the lock, and an integer *epoch* for the largest epoch it has ever seen. Thus, we only need to have corresponding *held* and *epoch* in the class as shown in Figure IV.1. To simplify the transitions at the implementation level, the class also takes a copy of the constant *Config* and the *NetClient* as the IO interface. We also use a ghost variable *Repr* for the heap representation of the class, so that

```
datatype Node = Node(held: bool, epoch: uint64)
```

```
class CNode {
  var held: bool;
  var epoch: uint64;
  var config: Config;
  var netClient: NetClient;

  ghost var Repr: set<object>;

  function method Abstractify() : Node
    reads this
  {
    Node(held, epoch)
  }
}
```

Figure IV.1: Translation of distributed lock from immutable datatype into class

the transitions in the implementation can modify these states.

After converting the state of the implementation, we convert the *NodeInit* to the constructor of the class. Figure IV.2 shows how to perform such a mechanical translation. We copy the assignment of all fields from the protocol state and maintain an implementation copy of *Config* and *NetClient*. After this construction, we can prove that if we extract the ghost copy of the state from the implementation, it's exactly the same as the result from *NodeInit*. Thus, the proof for *HostInit* becomes trivial.

Now, we need to convert the state transition for the system. Here, we treat the entrance function *NodeNext* separately from other functions, as this function deals with IO directly. Figure IV.3 shows how to convert such an entrance function. First, we define a local variable to keep track of outgoing packets generated by each function. After that, we convert the if-expression into an if-statement. For the first case, where the node holds the lock, it will grant the lock to the next node. We noticed that this is the end of this branch, which means this branch doesn't receive any packets or read the clock. Thus, we first assign *recv* and *clock* to none value, and then call the implementation of *NodeGrant*. Otherwise, the node needs to receive a message from the network. In that case, the implementation can simply call *ReceivePacket* to get a packet. But this doesn't always succeed. If the network environment fails (*ok* becomes false), or *clock* gets a receive timeout, the state *s* takes no transition, and exit. Once we get a packet in *r_pkt*, we can call the implementation of *NodeAccept* to get the outgoing packets. Finally, after all state transition for local state *s*, we send all outgoing packets with *SendPacketSeq*.

With this design, we can convert the entrance function, *NodeNext* into an executable method,

```
function NodeInit(config: Config) : Node
{
  Node(held  := config.my_index == 0,
       epoch := if config.my_index == 0 then 1 else 0)
}
```



```
constructor(c: Config, nc: NetClient)
  ensures abstractify(this) == NodeInit(c)
{
  held  := c.my_index == 0;
  epoch := if c.my_index == 0 then 1 else 0;

  config := c;
  netClient := nc;
  Repr := {this};
}
```

Figure IV.2: Translation of distributed lock from function initialization to class constructor

and hide the IO details for other methods to simplify the task for the translation of other functions. The proof target for this method is that after this state transition, the extracted ghost copy of the state *s* and the *sends* events are exactly as the transition of *NodeNext*. we leverage a special keyword *old* in Dafny to talk about the state at the beginning of the method, which can be used to track the previous state of this transition.

During the process of translation, we keep a list of callee functions to determine which function needs to be translated. For example, our program would automatically detect *NodeGrant* and *NodeAccept* are used in this implementation, and thus requires translation.

The next step is to translate all other used functions into methods. Figure IV.4 shows how to translate the state transition for *NodeGrant*. Similar to *NodeNext*, we need to convert the expression in the function to statements to perform updates. Here, we take extra care for the return value, as this becomes an in-place update in the implementation.

When translating the update of the local state, Cruiser compares all the fields in the new state with the value in the old state. Sometimes the new state uses the same value as the old state, e.g. $epoch := s.epoch$ in figure IV.4. In this case, Cruiser ignores such updates and leave the value as the old one.

The last thing to take care of is the order of assignments. To perform an in-place update, we lost the value of the previous state. Thus, we construct the packet before we make any updates for the node to ensure that the packet can always get the correct value from the local state. As for the updates from the implementation, one solution is to analyze the data dependencies to order the

```
function NodeNext(config: Config,
                  s: Node,
                  recv: Option<LPacket<EndPoint, Message>>,
                  clock: Option<uint64>) :
  (Node, seq<LPacket<EndPoint, Message>>)
{
  if s.held then
    NodeGrant(config, s)
  else NodeAccept(config, s, recv.v)
}
```



```
method CNodeNext(s: CNode)
  returns (ok: bool,
           ghost recv: Option<NetEvent>,
           clock: Option<NetEvent>,
           ghost sends: seq<NetEvent>)
  ensures ok ==>
            (abstractify(s), abstractify(sends)) ==
              NodeNext(s.config,
                       old(abstractify(s)),
                       abstractify(recv),
                       abstractify(clock))
{
  var pkts: seq<LPacket<EndPoint, Message>>;
  if s.held
  {
    recv := None();
    clock := None();

    pkts := CNodeGrant(s.config, s);
  }
  else
  {
    var r_pkt;
    ok, r_pkt, recv, clock := ReceivePacket(s.netClient);
    if (!ok || clock.Some?) {
      sends := [];
      return;
    }

    pkts := CNodeAccept(s.config, s, r_pkt);
  }

  ok, sends := SendPacketSeq(s.netClient, pkts);
}
```

Figure IV.3: Translation of state transition entrance for distributed lock

```
function NodeGrant(config: Config, s: Node) :
  (Node, seq<LPacket<EndPoint, Message>>)
{
  if s.epoch < 0xFFFF_FFFF_FFFF_FFFF then
    (Node(held := false, epoch := s.epoch),
     [LPacket(
         dst := c.servers[(c.my_index + 1) % (|c.servers| as uint64)],
         src := c.servers[c.my_index],
         msg := Transfer(s.epoch + 1)
             )
     ]
    )
  else
    (s, [])
}
```

```
method CNodeGrant(c: Config, s: CNode)
  returns (ret: (seq<LPacket<EndPoint, Message>>))
  ensures (Abstractify(s), ret) == NodeGrant(c, old(Abstractify(s)))
{
  if s.epoch < 0xFFFF_FFFF_FFFF_FFFF {
    ret := [LPacket(
                dst := c.servers[(c.my_index + 1) % (|c.servers| as uint64)],
                src := c.servers[c.my_index],
                msg := Transfer(s.epoch + 1)
                    )
            ];
    s.held := false;
  }
  else
  {
    ret := [];
  }
}
```

Figure IV.4: Translation of general state transition for distributed lock

assignment. But this solution does not always work, as there may be circular dependencies in the assignment. Breaking this circle efficient can be challenging, as this relies on the analysis of the cost of copying each field. Thus, we leave this problem to the developer: they can use a temporary value to store some values needed in the assignment, and then use the temporary value to update the state. If the developer forgets to use such a local variable, our implementation proof can capture this error and provide a hint.

All these steps to generate a corresponding implementation are performed automatically by Cruiser. Thus, the developer doesn't need to spend any time optimizing their implementation or proving the refinement from the implementation to the protocol.

4.6 Evaluation

To evaluate the power of Cruiser, we focus on two questions. First, how can Cruiser reduce the manual effort in verifying the implementation of a distributed system? Also, we want to see what performance our generated system gets.

To answer the first question, we used 3 different distributed systems: leader election in a ring (introduced in Section 2.5.2 and Section 3.6.1), a distributed lock protocol (introduced in Section 2.5.3 and Section 3.6.2) and a sharded hash table (introduced in Section 3.6.4). We implemented all Cruiser proof in Dafny [Lei10], and implemented Cruiser implementation generation in C#, based on existing source code in GitHub.

For all systems in our evaluation, we consider crash failures and an asynchronous network, which can arbitrarily delay, drop, or duplicate messages. We used a similar environment assumption as IronFleet, thus these failures are easy to model. For now, we're just focusing on the safety proof, but we believe the developer can also verify the liveness of the systems at the protocol layer with a reasonable amount of effort.

4.6.1 Proof Effort

We found that our verification framework reduces the manual effort in verifying a distributed implementation. Comparing to IronFleet, our function style protocol layer doesn't really introduce significant more effort in verification. Table IV.1 collects the total number of lines of code in the protocol layer.

As we can see, for two protocols that appear both in Cruiser and IronFleet, distributed lock and sharded hash table, the size of specification for both systems is similar. As for the protocol layer, Cruiser provides a more concise protocol layer comparing to IronFleet. The main reason is that IronFleet uses predicates to modularize transitions, which loses all context of the transition. Thus, there are many repetitive clauses in predicates from IronFleet. For example, when a node

		Cruiser		IronFleet
	layer	lines of code	proof effort	lines of code
Distributed Lock	spec	83	3 person days	65
	protocol	198		394
	impl	747	-	1524
Sharded Hash Table	spec	159	3 person weeks[1]	260
	protocol	2131		4724
	impl	1592	-	4277
Leader Election[2]	spec	66	2 person weeks	-
	protocol	721		-
	impl	770	-	-

[1] We leveraged some state transition design from IronKV.
[2] This proof is done by a Master student who is unfamiliar with formal methods.

Table IV.1: Summary of proof size and effort in Cruiser

receive a *SingleMessage*, it may sends an *ACK*. But IronFleet checks if this is a single message in multiple places, like deciding the receiving behavior, checking the sequence number, checking the ack number, etc. All these redundant checks introduce more code in the protocol layer. In contrast, Cruiser uses functions to model the state transition: after checking the type of received message at the beginning, there is no need to check it everywhere.

As for the implementation layer, Cruiser elaborately designed the state in each host and how to perform a state transition. As a result, the generated implementation is relatively concise and efficient. From all lines of generated implementation code, we have 578 lines of common implementations, including the network wrapper, host proof model, and the generated refinement proof. The code size for the generated state translation is relatively similar to the code size in the protocol. As we can see from the table, the implementation can sometimes have fewer lines of code than the protocol layer. The main reason is that the protocol layer performs a complex inductive proof, and that takes many lines for complex systems.

Another thing we noticed is that when Cruiser generates the implementation, the control flow is very close to the protocol functions. As Cruiser also keeps the name for local variables, the implementation can be easily understood by the developer. Actually, we applied a manual patch to fix a proof problem in the sharded hash table.

Manual Patch in SHT One problem we found in SHT is that when we iterate over a set or a map, Dafny couldn't guarantee that the implementation gets the same order as the protocol. As a result, when we try to retransmit all unACKed messages in SHT, it is difficult to generate the same sequence of messages between the implementation and the protocol. However, we don't really care about the actual order in this iteration. Thus, we used a trusted module. We provide a function to iterate over a set and its corresponding implementation method. Once the protocol uses this

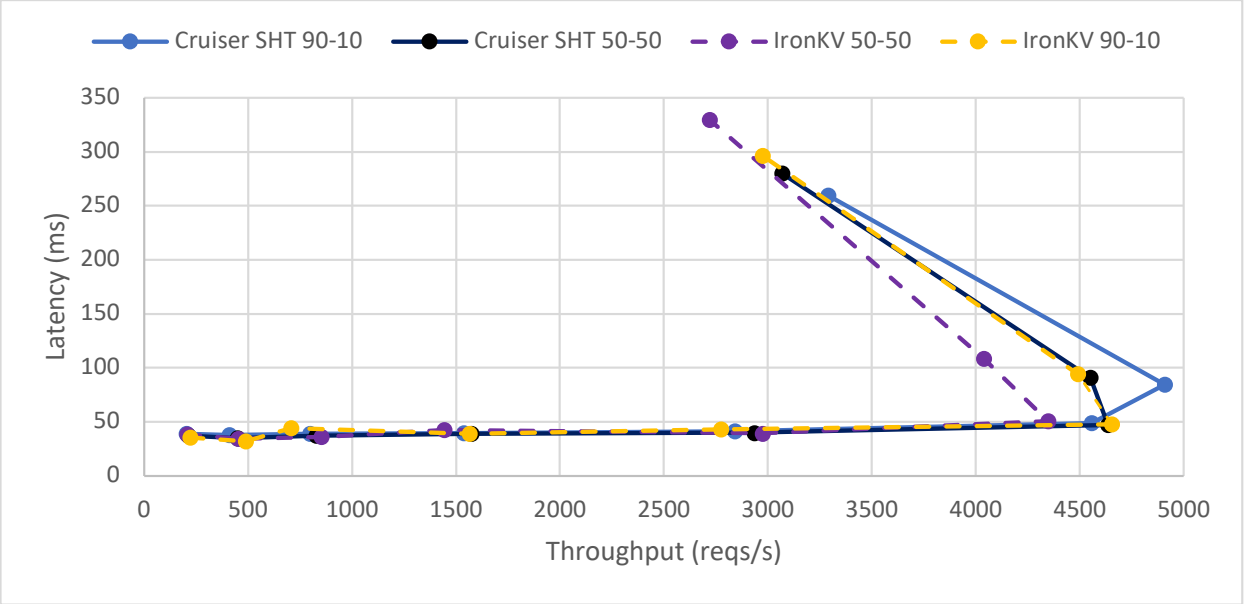

Figure IV.5: Cruiser SHT performance evaluation

function, our generation automatically chooses its corresponding implementation. In this way, we can finish the refinement proof.

As for the leader election, a master student implemented the proof independently. She is unfamiliar with formal methods so this proof takes a relatively long time. From her experience, we found that the inductive invariant from Sift can also be used in this proof, which simplifies the protocol refinement proof. After that, Cruiser can generate the implementation automatically.

4.6.2 Performance of Verified Implementations

We also compared the performance of our generated implementation with IronFleet, a hand-written implementation in Dafny. We run our experiment on AWS EC2. Each machine has 4 virtual CPUs and 8 GB of memory, running Windows Server 2019. Nodes are connected with a 10 GB network.

We first evaluated the throughput of the distributed lock with two hosts on two different machines. We found that both Cruiser and IronFleet reach exactly 64 epochs every second. We guess that this application has a very simple local state, and the network becomes the bottleneck. This could be a restriction from AWS.

We also evaluated the sharded hash table with two different workloads. As shown in Figure IV.5, The performance of our generated implementation is slightly better than IronKV. We used 1,000 keys in a single server and serviced requests from an increasing number of clients (8 - 1024) in a closed loop. We choose 64-bit integers (8 bytes) as the keys. The value is an array of bytes, and we

use 1,000 bytes in this evaluation. We used two typical workloads. One is a write-heavy workload, with 50% of read and 50% of write. The other is a read-heavy workload, with 90% of read and 10% write. As we can see from the figure, the performance of Cruiser implementation is very close to IronFleet. We believe this is the benefit from the heap-based implementation. In IronFleet, the implementation state is immutable. Thus, there are many redundant copies in a state transition. With Cruiser, the transition only performs necessary updates and leaves other fields unchanged. This evaluation result shows that our automatically generated implementation has similar or even better performance than manually optimized implementations.

4.7 Limitations and future directions

Our experience with Cruiser helps us to verify a real heap-based distributed system. For all of its success, however, we found some directions to further optimize the system in the future.

• **Better Heap Analysis.** While Cruiser can analyze the heap allocation for the state, some complex data structures can still be challenging. For example, when it comes to multiple layers of objects, the heap layout can be difficult to analyze. Sometimes, a class can even change its heap representation, as a dynamic vector, which influences all its parent objects. How to better analyze the heap allocation for the state and generate the proof leaves an open question to the research community.

• **Generating Loops.** As Cruiser requires the developer to implement the protocol in functions, there is no loop allowed. Instead, we use recursion to perform iterative operations. Thus, Cruiser can only generate a recursive implementation, but not a loop. Sometimes, implementing a recursive implementation efficiently raises a larger challenge for both the developer and compiler to optimize. Although proving the correctness of a loop relies on a loop invariant, the pre-and post-condition of that function can be a hint for us to generate the loop invariant. We hope that in the future version of Cruiser, we can automatically convert some recursions into loops to optimize the invariant.

• **Generating Packet Parsing Automatically.** Currently, Cruiser relies on the developer to implement the trusted parsing functions, and implement the marshaling and parsing methods. However, some existing research, such as EverParse [RDLF+19], shows how to automatically generate efficient and provable correct parsers and serializers. It is possible to adopt such a methodology to simplify the message marshaling in Cruiser.

• **Supporting More Complex Message Sending Model.** Currently, Cruiser assumes that outgoing messages are sent one by one. Thus, we use a sequence to model the outgoing messages. But this model turns out to be limited. From our example of the sharded hash table, we need to inject a trusted method to convert a set into a sequence to iterate, and the implementation can get the same sequence as the function. If we change the message model into a set, the order doesn't

matter anymore. Also, a sequence of messages requires a marshaling price for every message to be sent. This is not efficient for multi-cast, which is commonly used in distributed systems. In that case, the implementation only needs to marshal the message once and send it to all destinations. We can introduce a new type of outgoing packet, *MultiCast*, to incorporate multi-cast into our message-sending model.

• **Adding Verified Built-in Libraries.** All Dafny built-in libraries are inefficient immutable data structures. To implement an efficient system, the developer needs to either use a trusted library from C#, or verify every library they want to use. With Cruiser, we can automatically generate the implementation. At the same time, Cruiser can also introduce some verified built-in libraries when possible, to optimize the performance of the implementation.

4.8 Conclusion

This chapter presents Cruiser, a new framework to automate the verification of efficient distributed implementations. Cruiser introduces a function-style protocol layer, which requires a little more effort from the developer. From that protocol layer, Cruiser benefits from the invariant finding from Sift to simplify the verification of the protocol layer and automatically generate an efficient implementation. Our evaluation shows that Cruiser can reduce the effort needed to verify distributed systems, and also generate high-performance implementations.

CHAPTER V

Related Work

In this chapter, we discuss existing approaches in distributed system verification, finding inductive invariants and automated verification.

5.1 Verification of Systems

Formal verification is gaining popularity in the systems community as an alternative to testing. Much effort has gone to verifying real systems, including OS kernels [KEH+09, MPX+13, GSC+16, CWS+16], file, and storage systems [CCK+17, ZDD+19, CTKZ19]. These works provide strong guarantees of correctness, but at the cost of extensive manual effort; Sift and Cruiser, by contrast, require little manual proof effort while verifying systems of considerable complexity, such as SHT and MultiPaxos.

The significance of formal verification is particularly pronounced within the realm of distributed systems, as they are notoriously subtle and complex. Lamport's TLA+ [Lam02] has mostly been used to prove the correctness of abstract protocols, as it is not really designed for actual implementations. The first practical verified implementations of distributed systems came with IronFleet [HHK+15, HHK+17] and Verdi [WWP+15]. IronFleet uses a combination of refinement and reduction [Lip75] to facilitate the verification of distributed systems. Verdi, on the other hand, uses a series of system transformers. It starts by proving the correctness of the system under a very strong model and uses the transformers to prove refinement to increasingly weaker models. Both Verdi and IronFleet rely on significant *manual effort* to identify the inductive invariants of the system—and thus prove its correctness.

Ivy [PMP+16] is another approach to verify distributed systems. With Ivy, the developer needs to iteratively refine an invariant until an inductive invariant is identified. Although easier than IronFleet and Verdi, this approach still relies on considerable amounts of manual effort (in the order of person months) to complete a proof of correctness.

A recent work proposed *pretend synchrony* [vGKB+19], an approach that aims to simplify the reasoning behind distributed protocols. The idea behind *pretend synchrony* is that one can transform

an asynchronous distributed protocol into an equivalent synchronous protocol, thus making it easier to reason about. Ideas like this can be used in conjunction with I4: by simplifying the problem, we may be able to increase the scalability of the underlying model checking and thus automate the proof of more complex protocols.

Pnueli et al. [PRZ01] proposed that verifying a parameterized distributed system consisting of n identical interacting processes can be accomplished by verifying a relatively small finite instantiation. The basic idea is that a system of $n > n_0$ processes can be verified by checking an instance with just n_0 processes, where n_0 is linear in the number of local state variables of a single process. This idea is the inspiration behind I4. I4, however, is fundamentally different from the approach of Pnueli et al. First, this approach does not actually find any inductive invariants. They merely show that proving the correctness of a system with n_0 processes is sufficient to prove its correctness for any $n > n_0$. Moreover, this only works when each process has a finite state (n_0 depends linearly on the state size of each process, so it would be infinite otherwise). As such, this approach doesn't apply to today's distributed systems, whose state space is unbounded. This is exactly the innovation of I4: finitizing the problem and generalizing the result to infinite-state protocols. The approach of Pnueli et al. relies solely on model checking, precisely because it assumes that the only source of infinity is the number of nodes in the system.

More recently, Armada [LCK+20], a tool designed to verify concurrent programs, was proposed. While Armada has some superficial similarities to Sift and Cruiser—namely the use of refinement and automation—it is in fact drastically different. It operates in an environment almost diametrically opposed to ours: single-machine, multi-threaded code where communication happens via shared memory, as opposed to a sequential execution on a distributed system where communication happens via message passing. Additionally, while Armada makes heavy use of automation to generate proofs, it still requires its users to write significant parts of the proof—hundreds of lines of code—manually.

5.2 Inductive Invariants

To overcome the challenge of finding an inductive invariant, previous work has taken a number of approaches for both finite- and infinite-state programs. Many works have focused on identifying loop invariants. Proof planning [IS97] uses failed proof attempts to find loop invariants, while Flanagan and Qadeer [FQ02] use a technique called predicate abstraction. Furia and Meyer [FM10] use heuristics from postconditions to synthesize loop invariants. Daikon [ECGN00] was proposed in 2000 to learn possible program invariants, followed by Houdini [FL01] which learns conjunctive inductive invariants. IC3 [Bra11] and PDR [EMB11] can automatically find inductive invariants for finite state machines, and were later extended to certain systems with infinite-domain variables [HB12, CG12]. For list-manipulating programs, Property-Directed Shape Analysis [IBR+14]

and UPDR [KBI+17] have shown effective results, though the approach doesn't guarantee termination. Grebenshchikov et al. [GLPR12] show that a Horn clause is the most common pattern in program verification, and the ICE learning model [GLMN14, DEG+17] uses this result to synthesize invariants. Although some of these techniques can deal with a *finite* number of variables with *infinite* domains (e.g., strings or infinite integers), they cannot effectively deal with distributed systems, whose state typically contains an unbounded number of variables (e.g., an unbounded set of sent messages, and unbounded copies of a state variable in different nodes).

5.3 Automated Verification

With the advancements in automated reasoning [HJS+, BT18, DMB08] and abstraction techniques [BCLR04, GS97, CGJ+00], automatically verifying correctness through model checking [CE81, QS82] has significantly improved in different domains, both for hardware [Bra11, EMB11, SG17, GS19b, BFP] and software [BCLR04, JM09, BK11, KT14, Bey21]. However, model checking still does not scale well to large complex systems, due to state-space explosion [DLS06, CKNZ12].

Bedrock [Chl11] is a framework for automatically generating proofs for first-class code pointers. Bedrock uses mostly-automated discharge of verification conditions inspired by separation logic. Using a computational approach coupled with functional programming, Bedrock avoids quantifiers almost entirely, and achieves mostly-automated verification.

Yggdrasil [SBTW16] and Hyperkernel [NSZ+17] are two recent approaches that aim to minimize the human effort required to perform formal verification. Yggdrasil [SBTW16] presents a formally verified file system using the notion of *crash refinement*. It automatically verifies that, even in the presence of nondeterministic events like crashes and reordering, a correct implementation will still produce the same disk state as its specification. Yggdrasil uses a finite domain to guarantee decidable SMT queries. Hyperkernel [NSZ+17] is a formally verified OS kernel. Similar to Yggdrasil, Hyperkernel also finitizes kernel interfaces to keep SMT queries decidable. I4 has a similar goal—push-button verification—but for distributed systems. The key difference is that distributed systems have infinite domains, which lead to undecidable SMT queries. To sidestep this problem, I4 uses a unique combination of finite protocol instances (via model checking) and decidable SMT queries in the infinite domain (via Ivy).

After I4 is published, there are many follow-up works automating the inference of inductive invariants for distributed systems. For example, DistAI [YTG+21] and primal-dual Houdini [PWK+22] provide a better termination guarantee on finding the inductive invariant. Some other approaches, SWISS [HHMP21], IC3PO [GS21], DuoAI [YTGN22] take existential quantifier into consideration, and even scale invariant inference to complex protocols such as Paxos.

All the aforementioned techniques [KBI+17, FWSS19, MGJ+19a, KPIA20, GS21, HHMP21, YTG+21], however, target monolithic, single-layer verification, primarily at the protocol level, and cannot scale to detailed system implementations. Our Sift and Cruiser combines these monolithic provers with the well-founded concepts of refinement [AL91, GL00, Lam94] to scale verification all the way to complex executable implementations.

CHAPTER VI

Conclusion

Verification is getting more and more attention for its promising future in building safe systems. In this dissertation, we focus on the automation of the verification process, reducing the effort needed in writing a proof. In particular, we start from the best automation can get, with little to no manual effort to prove monolithic distributed protocols. After that, we push the boundary on the kinds of systems that can benefit from proof automation to complex refinement proofs and even efficient heap-based implementations. In this dissertation, we make the following contributions.

In Chapter II, we introduce I4, a new approach for verifying the correctness of distributed protocols, with little to no manual effort and without relying on human intuition. I4 is based on a simple insight: distributed protocols exhibit a lot of regularity, which means that the system's behavior doesn't fundamentally change from a small instance to a large scale. Thus, we can use an inductive invariant from a small, finite instance to infer a generalized inductive invariant that holds for all instances. With this hypothesis, I4 leverages the power of model checking to automatically identify the inductive invariant of a small, finite instance of the protocol. And then, I4 can generalize that invariant to instances of arbitrary size. The evaluation shows that I4 is successful in automatically proving the correctness of a number of interesting distributed protocols, even ones whose subtleties and internal workings were unknown to us.

In Chapter III we introduce Sift to scale automation to more complex systems. Sift is a novel two-tier methodology that combines the power of refinement with automation from monolithic provers like I4 and IC3PO. Within each refinement proof, Sift introduces an encapsulation technique to convert the refinement proof involving two layers into an encapsulated layer, which is amenable to prove by monolithic provers. If that proof fails, Sift decomposes the proofs of complex distributed implementations into a number of refinement steps, each of which is amenable to automation. We use Sift to prove the correctness of six distributed implementations, including the notorious MultiPaxos. Our proof effort is significantly less than previous attempts. Our performance evaluation shows that our verified system has comparable performance with previous manual implementation.

In Chapter IV, we further optimize the performance of verified systems with heap. We found that a function-based protocol layer can be efficiently translated into a heap-based implementation.

Thus, we introduce Cruiser, an approach that combines implementation generation with refinement proof generation to automatically generate efficient heap-based distributed systems. Comparing to previous work, Cruiser extends automation to undecidable logic, which provides more freedom for the developer to perform complex optimizations. Our evaluation shows Cruiser reduces the effort needed to generate a high-performance implementation.

Throughout this dissertation, we pushed the boundary of how distributed systems can be verified automatically, and made it all the way from a simple protocol down to a real, complex implementation. But still, verification of systems requires extra effort, which can be simplified. This challenge is hard, but it can potentially benefits the safety of all critical systems.

BIBLIOGRAPHY

[AL91] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[BBC+97] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. *The Coq proof assistant reference manual: Version 6.1*. PhD thesis, Inria, 1997.

[BBC+11] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[BCLR04] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside microsoft. In *Integrated Formal Methods, 4th International Conference, IFM 2004, Canterbury, UK, April 4-7, 2004, Proceedings*, volume 2999 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2004.

[BCO04] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. A decidable fragment of separation logic. In *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science, 24th International Conference, Chennai, India, December 16-18, 2004, Proceedings*, volume 3328 of *Lecture Notes in Computer Science*, pages 97–109. Springer, 2004.

[BDH07] William J. Bolosky, John R. Douceur, and Jon Howell. The farsite project: A retrospective. *SIGOPS Oper. Syst. Rev.*, 41(2):17–26, April 2007.

[BEL75] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT – a formal system for testing and debugging programs by symbolic execution. In *Intl. Conf. on Reliable Software*, 1975.

[Bey21] Dirk Beyer. Software verification: 10th comparative evaluation (sv-comp 2021). *Tools and Algorithms for the Construction and Analysis of Systems*, 12652:401, 2021.

[BFP] Armin Biere, Nils Froleyks, and Mathias Preiner. Hardware model checking competition (HWMCC) 2020. `http://fmv.jku.at/hwmcc20`.

[BFT16] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[BK11] Dirk Beyer and M Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In *International Conference on Computer Aided Verification*, pages 184–190. Springer, 2011.

[Bra11] Aaron R Bradley. SAT-based model checking without unrolling. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011.

[Bru09] Jake Brutlag. Speed matters for Google web search, 2009.

[BT18] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.

[Bur06] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, 2006.

[CCK+17] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 270–286, New York, NY, USA, 2017. ACM.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008.

[CDE+12] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 261–264, Hollywood, CA, October 2012. USENIX Association.

[CE81] Edmund M Clarke and E Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

[CG12] Alessandro Cimatti and Alberto Griggio. Software model checking via IC3. In *International Conference on Computer Aided Verification*, pages 277–293. Springer, 2012.

[CGJ+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000.

[CGMT14] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. IC3 modulo theories via implicit predicate abstraction. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 46–61. Springer, 2014.

[CGT21] Alessandro Cimatti, Alberto Griggio, and Stefano Tonetta. The VMT-LIB language and tools. *arXiv preprint arXiv:2109.12821*, 2021.

[Chl11] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 234–245, New York, NY, USA, 2011. ACM.

[CKNZ12] Edmund M Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. *Model Checking and the State Explosion Problem*, pages 1–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[CR79] Ernest Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22(5):281–283, 1979.

[CTKZ19] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Argosy: Verifying layered storage systems with recovery refinement. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1054–1068, New York, NY, USA, 2019. Association for Computing Machinery.

[CVE17] CVE-2016-5195. Dirty cow vulnerability. https://dirtycow.ninja/, 2017.

[CWS+16] Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. Toward compositional verification of interruptible os kernels and device drivers. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 431–447, New York, NY, USA, 2016. Association for Computing Machinery.

[CZC+15] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 18–37, New York, NY, USA, 2015. ACM.

[DEG+17] Deepak D'Souza, P Ezudheen, Pranav Garg, P Madhusudan, and Daniel Neider. Horn-ice learning for synthesizing invariants and contracts. *arXiv preprint arXiv:1712.09418*, 2017.

[Dij75] Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[DLS06] Stéphane Demri, François Laroussinie, and Ph Schnoebelen. A parametric analysis of the state-explosion problem in model checking. *Journal of Computer and System Sciences*, 72(4):547–575, 2006.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer.

[dt] Coq development team. The coq proof assistant reference manual. http://coq.inria.fr/distrib/current/refman/.

[ECGN00] Michael D Ernst, Adam Czeisler, William G Griswold, and David Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd international conference on Software engineering*, pages 449–458. ACM, 2000.

[EMB11] Niklas Een, Alan Mishchenko, and Robert Brayton. Efficient implementation of property directed reachability. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, pages 125–134. FMCAD Inc, 2011.

[Eng12] Jakob Engblom. A review of reverse debugging. In *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*, pages 1–6. IEEE, 2012.

[FL01] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java. In *International Symposium of Formal Methods Europe*, pages 500–517. Springer, 2001.

[FM10] Carlo Alberto Furia and Bertrand Meyer. Inferring loop invariants using postconditions. In *Fields of logic and computation*, pages 277–300. Springer, 2010.

[FQ02] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 191–202, New York, NY, USA, 2002. ACM.

[FWSS19] Yotam M. Y. Feldman, James R. Wilcox, Sharon Shoham, and Mooly Sagiv. Inferring inductive invariants from phase structures. In *Computer Aided Verification*, pages 405–425, Cham, 2019. Springer International Publishing.

[GGK+18] Jeffrey Gennari, Arie Gurfinkel, Temesghen Kahsai, Jorge A Navas, and Edward J Schwartz. Executable counterexamples in software model checking. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 17–37. Springer, 2018.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223. ACM, 2005.

[GL00] Stephen J Garland and Nancy A Lynch. Using I/O automata for developing distributed systems. *Foundations of component-based systems*, 13(285-312):5–2, 2000.

[GL06] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS)*, 31(1):133–160, 2006.

[GLMN14] Pranav Garg, Christof Löding, P Madhusudan, and Daniel Neider. ICE: A robust framework for learning invariants. In *International Conference on Computer Aided Verification*, pages 69–87. Springer, 2014.

[GLPR12] Sergey Grebenshchikov, Nuno P Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. *ACM SIGPLAN Notices*, 47(6):405–416, 2012.

[Gra78] James N Gray. Notes on data base operating systems. In *Operating Systems*, pages 393–481. Springer, 1978.

[GS] Aman Goel and Karem A. Sakallah. IC3PO: IC3 for Proving Protocol Properties. https://github.com/aman-goel/ic3po.

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *International Conference on Computer Aided Verification*, pages 72–83. Springer, 1997.

[GS19a] Aman Goel and Karem Sakallah. Empirical evaluation of IC3-based model checking techniques on verilog RTL designs. In *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 2019.

[GS19b] Aman Goel and Karem Sakallah. Model checking of verilog RTL using IC3 with syntax-guided abstraction. In *NASA Formal Methods Symposium*. Springer, 2019.

[GS21] Aman Goel and Karem Sakallah. On symmetry and quantification: A new approach to verify distributed protocols. In *NASA Formal Methods Symposium*, pages 131–150. Springer, 2021.

[GSC+16] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: an extensible architecture for building certified concurrent OS kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 653–669, USA, 2016. USENIX Association.

[HB12] Kryštof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 157–171. Springer, 2012.

[HHK+15] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17. ACM, 2015.

[HHK+17] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving safety and liveness of practical distributed systems. *Commun. ACM*, 60(7):83–92, June 2017.

[HHL+14] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 165–181, Berkeley, CA, USA, 2014. USENIX Association.

[HHMP21] Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. Finding invariants of distributed systems: It's a small (enough) world after all. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 115–131. USENIX Association, April 2021.

[HJS+] Marijn Heule, Matti Järvisalo, Martin Suda, Tomas Balyo, Carsten Sinz, and Armin Biere. The international SAT Competitions web page. http://www.satcompetition.org/.

[IBR+14] Shachar Itzhaky, Nikolaj Bjørner, Thomas Reps, Mooly Sagiv, and Aditya Thakur. Property-directed shape analysis. In *International Conference on Computer Aided Verification*, pages 35–51. Springer, 2014.

[IS97] Andrew Ireland and Jamie Stark. On the automatic discovery of loop invariants. In *NASA Conference Publication*, pages 137–152. Citeseer, 1997.

[Jac02] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[JM09] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):1–54, 2009.

[KBI+17] Aleksandr Karbyshev, Nikolaj Bjørner, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. Property-directed inference of universal invariants or proving their absence. *Journal of the ACM (JACM)*, 64(1):7, 2017.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[Kin76] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 1976.

[KPIA20] Jason R. Koenig, Oded Padon, Neil Immerman, and Alex Aiken. First-order quantified separators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 703–717, New York, NY, USA, 2020. Association for Computing Machinery.

[KT14] Daniel Kroening and Michael Tautschnig. CBMC - C bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391. Springer, 2014.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[Lam01] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, December 2001.

[Lam02] Leslie Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[LCK+20] Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. Armada: Low-effort verification of high-performance concurrent programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 197–210, New York, NY, USA, 2020. Association for Computing Machinery.

[Lei10] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.

[Lew80] Harry R Lewis. Complexity results for classes of quantificational formulas. *Journal of Computer and System Sciences*, 21(3):317–353, 1980.

[LIN] GREG LINDEN. Marissa mayer at web 2.0. http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html.

[Lip75] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, December 1975.

[LS14] Suho Lee and Karem A. Sakallah. Unbounded Scalable Verification Based on Approximate Property-Directed Reachability and Datapath Abstraction. In *Computer-Aided Verification (CAV)*, volume LNCS 8559, pages 849–865, Vienna, Austria, July 2014. Springer.

[MAG+] Haojun Ma, Hammad Ahmad, Aman Goel, Eli Goldweber, Jean-Baptiste Jeannin, Manos Kapritsos, and Baris Kasikci. Sift Artifact. https://github.com/GLaDOS-Michigan/Sift.

[MAG+22] Haojun Ma, Hammad Ahmad, Aman Goel, Eli Goldweber, Jean-Baptiste Jeannin, Manos Kapritsos, and Baris Kasikci. Sift: Using refinement-guided automation to verify complex distributed systems. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 151–166, Carlsbad, CA, July 2022. USENIX Association.

[McM] Kenneth L McMillan. non-duplicating ordered transport service. https://github.com/microsoft/ivy/blob/master/doc/examples/sht/trans.md.

[MGJ+] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A Sakallah. I4 Artifact. https://github.com/GLaDOS-Michigan/I4.

[MGJ+19a] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A Sakallah. I4: incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 370–384, 2019.

[MGJ+19b] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A Sakallah. Towards automatic inference of inductive invariants. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 30–36, 2019.

[MPX+13] Haohui Mai, Edgar Pek, Hui Xue, Samuel Talmadge King, and Parthasarathy Madhusudan. Verifying security invariants in ExpressOS. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 293–304, New York, NY, USA, 2013. Association for Computing Machinery.

[NRZ+14] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. Use of formal methods at Amazon Web Services. *See http://research. microsoft. com/en-us/um/people/lamport/tla/formal-methods-amazon. pdf*, 2014.

[NSZ+17] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an OS kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 252–269, New York, NY, USA, 2017. ACM.

[NWP02] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[OO14] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, pages 305–319, 2014.

[PdMB10] Ruzica Piskac, Leonardo de Moura, and Nikolaj Bjørner. Deciding effectively propositional logic using DPLL and substitution sets. *Journal of Automated Reasoning*, 44(4):401–424, Apr 2010.

[PLSS17] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made EPR: decidable reasoning about distributed protocols. *Proc. ACM Program. Lang.*, 1(OOPSLA):108:1–108:31, 2017.

[PMP+16] Oded Padon, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. *ACM SIGPLAN Notices*, 51(6):614–630, 2016.

[PRZ01] Amir Pnueli, Sitvanit Ruah, and Lenore Zuck. Automatic deductive verification with invisible invariants. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 82–97. Springer, 2001.

[PWK+22] Oded Padon, James R. Wilcox, Jason R. Koenig, Kenneth L. McMillan, and Alex Aiken. Induction duality: Primal-dual search for invariants. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022.

[PWZ13] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating separation logic using SMT. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 773–789. Springer, 2013.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on programming*, pages 337–351. Springer, 1982.

[RDLF+19] Tahina Ramananandro, Antoine Delignat-Lavaud, Cedric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. EverParse: Verified secure Zero-Copy parsers for authenticated message fofrmats. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1465–1482, Santa Clara, CA, August 2019. USENIX Association.

[Rey02] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.

[RUIMI16] Microsoft Research, Carnegie Mellon University, INRIA, and MSR-INRIA. Everest project. https://project-everest.github.io/, 2016.

[RWY+20] Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. CLN2INV: learning loop invariants with continuous logic networks. In *International Conference on Learning Representations*, 2020.

[SBTW16] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 1–16, Berkeley, CA, USA, 2016. USENIX Association.

[SCF+11] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming*, pages 266–278. ACM, 2011.

[SG17] Berkeley Logic Synthesis and Verification Group. ABC: A system for sequential synthesis and verification. http://www.eecs.berkeley.edu/~alanmi/abc/, 2017.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*, pages 263–272, 2005.

[SMLN+03] Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking (TON)*, 11(1):17–32, 2003.

[Tea08] Amazon S3 Team. Amazon S3 availability event: July 20, 2008. http://status.aws.amazon.com/s3-20080720.html, 2008.

[The04] The Associated Press. General Electric acknowledges Northeastern blackout bug. http://www.securityfocus.com/news/8032, 2004.

[TLM+18] Marcelo Taube, Giuliano Losa, Kenneth L McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R Wilcox, and Doug Woos. Modularity for decidability of deductive verification with applications to distributed systems. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 662–677, 2018.

[vGKB+19] Klaus von Gleissenthall, Rami Gökhan Kici, Alexander Bakst, Deian Stefan, and Ranjit Jhala. Pretend synchrony: synchronous verification of asynchronous distributed programs. *PACMPL*, 3(POPL):59:1–59:30, 2019.

[WWP+15] James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas Anderson. Verdi: a framework for implementing and formally verifying distributed systems. *ACM SIGPLAN Notices*, 50(6):357–368, 2015.

[YCSS12] Junfeng Yang, Ang Cui, Sal Stolfo, and Simha Sethumadhavan. Concurrency attacks. In *The Fourth USENIX Workshop on Hot Topics in Parallelism*, 2012.

[YRW+20] Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui 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 2020, page 106–120, New York, NY, USA, 2020. Association for Computing Machinery.

[YTG+21] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. DistAI: Data-Driven automated invariant learning for distributed protocols. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 405–421. USENIX Association, July 2021.

[YTGN22] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. DuoAI: Fast, automated inference of inductive invariants for verifying distributed protocols. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 485–501, Carlsbad, CA, July 2022. USENIX Association.

[Zav17] Pamela Zave. Reasoning about identifier spaces: How to make chord correct. *IEEE Transactions on Software Engineering*, 43(12):1144–1156, 2017.

[ZDD+19] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. Using concurrent relational logic with helpers for verifying the AtomFS file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 259–274, New York, NY, USA, 2019. Association for Computing Machinery.