I4: Incremental Inference of Inductive Invariants for Verification of Distributed Protocols

Haojun Ma, Aman Goel, Jean-Baptiste Jeannin
Manos Kapritsos, Baris Kasikci, Karem A. Sakallah

University of Michigan

{mahaojun,amangoel,jeannin,manosk,barisk,karem}@umich.edu

Abstract

Designing and implementing distributed systems correctly is a very challenging task. Recently, formal verification has been successfully used to prove the correctness of distributed systems. At the heart of formal verification lies a computer-checked proof with an inductive invariant. Finding this inductive invariant, however, is the most difficult part of the proof. Alas, current proof techniques require inductive invariants to be found manually—and painstakingly—by the developer.

In this paper, we present a new approach, *Incremental Inference of Inductive Invariants* (I4), to automatically generate inductive invariants for distributed protocols. The essence of our idea is simple: the inductive invariant of a *finite* instance of the protocol can be used to infer a general inductive invariant for the *infinite* distributed protocol. In I4, we create a finite instance of the protocol; use a model checking tool to automatically derive the inductive invariant for this finite instance; and generalize this invariant to an inductive invariant for the infinite protocol. Our experiments show that I4 can prove the correctness of several distributed protocols like Chord, 2PC and Transaction Chains with little to no human effort.

1 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 [5, 7, 25, 37, 49]. 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 [14, 53, 54, 57]. 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 [9, 10, 30, 31, 38, 44, 45].

Unfortunately, existing approaches to formally verifying complex systems have a major scalability bottleneck. These techniques use interactive and automated theorem provers [15, 42, 46, 47, 52] to dispatch a number of proof obligations, thereby simplifying the proof. At the heart of every proof, however, 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 [30].

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

SOSP '19, October 27–30, 2019, Huntsville, ON, Canada

ACM ISBN 978-1-4503-6873-5/19/10…$15.00
https://doi.org/10.1145/3341301.3359651

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 [30], the authors spent about one week trying to identify the inductive invariant for a very simple distributed protocol, where a number of 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 [2]. 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 [39]—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. 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.

This paper introduces 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.

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 [4, 30, 56] 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 [6] 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 rest of the paper is structured as follows. Section 2 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 3 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 4) and generalizing that invariant to one that holds for all instances (Section 5). Section 6 presents our experiences applying I4 to real distributed protocols and Section 7 discusses the limitations of our approach and some open research problems. Section 8 discusses related work and Section 9 concludes the paper.

2 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

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;
}
```

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 [47]. 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 paper, 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 true for a large instance of 1000 nodes. We will demonstrate this regularity using a simple lock server protocol [47, 56] 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 4 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.

3 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 property. As we mentioned in Section 2, the hardest part of any correctness proof is *finding* such an

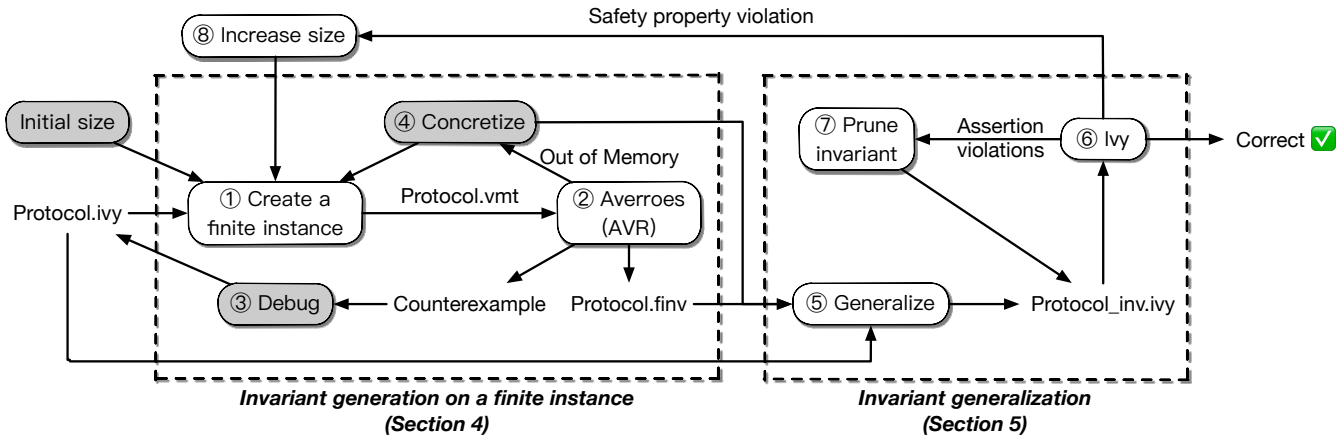

Figure 1. Flow of I4. White boxes are fully automated, while gray boxes denote manual effort.

invariant. Once we have a candidate invariant for a finite instance, we generalize it and use the Ivy tool [47] to check whether it is inductive for an arbitrarily-sized instance.

Figure 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 [27] 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 [18, 24].

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 4 describes the above steps in more detail (steps ①, ②, ③ and ④ of Figure 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 5.

4 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 [1], an extension of the SMT-LIB format [3] to represent state-transition systems. I4 then calls the Averroes model-checking tool [27] to generate an inductive invariant of this finite instance.

4.1 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 [6, 13, 17, 26, 27, 41]. 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 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.

4.2 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 5.2 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 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 $|D_{client}| = 2$.

4.3 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 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 straightforward (Figure 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 `link_C0_S0` $\wedge$ `link_C1_S0` $\implies$ $(\texttt{C0} = \texttt{C1})$.

4.4 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 5, 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

Figure 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.

the unbounded case. We call such clauses *instance-specific* clauses. Section 5 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.

4.5 Invariant generation on the finite instance

After creating a finite VMT instance, I4 passes it on to the Averroes v2.0 tool [27] (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 [18, 24]. 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.

5 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 4.4, 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 1).

5.1 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 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 $\forall N_1, N_2.\ (N_1 \neq N_2) \implies P(N_1) \wedge Q(N_2)$.

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 second case (lines 2 and 10-20 in Algorithm 2) is a result of the (optional) concretization procedure we described in Section 4.4. 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).

5.2 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.

6 Evaluation

We evaluate the ability of I4 to infer inductive invariants by testing it on seven distributed protocols: a *lock server* (Section 6.1), a *leader election* algorithm (Section 6.2), a *distributed lock* protocol (Section 6.3), a *Chord ring [51]* (Section 6.4), a *learning switch* (Section 6.5), a *database chain consistency* protocol (Section 6.6), and a *two-phase commit* protocol [28] (Section 6.7). For the first six protocols, we verified the correctness of existing Ivy implementations using the safety properties specified in [47]; we cover all examples originally presented in [47], albeit with a much higher degree of automation. We implemented *two-phase commit* and specified its safety property based on its original description [28]. Finally, Section 6.8 evaluates our I4 prototype in terms of how long it takes to verify each protocol.

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 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.

6.1 Lock Server

Our first case study is a simple lock server, our running example from Section 2. 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 going through its generalization (step ④ in Fig. 1), where it places universal quantifiers before every strengthening assertion. The generalized inductive invariant below passes Ivy's verification with no manual effort.

$$\forall S0, C0.\ \neg(semaphore(S0) \wedge link(C0, S0)) \quad \wedge \\ Safety\ Property$$

6.2 Leader Election

Our second case study is a leader election protocol on a ring [8, 47]. 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.

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 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 [3] calls required to find the inductive invariant of the finite instance.

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 [47]. The protocol also instantiates a total order $le(ID_1, ID_2)$ to compare any two *ID*s. 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 {`N0`, `N1`, `N2`} and domain of IDs {`ID0`, `ID1`, `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(`N0`) = `ID0`, idn(`N1`) = `ID1` and idn(`N2`) = `ID2`—we facilitate AVR to find a finite-instance invariant (shown in Table 2) that I4 can generalize to the full protocol (shown in Table 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.

$\neg(\neg pending\_ID0\_N0 \wedge leader\_N0)$ $\wedge$
$\neg(\neg pending\_ID1\_N1 \wedge leader\_N1)$ $\wedge$
$\neg(\neg pending\_ID1\_N2 \wedge pending\_ID1\_N1)$ $\wedge$
$\neg(\neg pending\_ID1\_N2 \wedge btw\_N0\_N1\_N2 \wedge pending\_ID1\_N0)$ $\wedge$
$\neg(le\_ID0\_ID1 \wedge pending\_ID0\_N0)$ $\wedge$
$\neg(le\_ID0\_ID1 \wedge btw\_N0\_N1\_N2 \wedge pending\_ID0\_N2)$ $\wedge$
$\neg(le\_ID0\_ID2 \wedge pending\_ID0\_N0)$ $\wedge$
$\neg(le\_ID0\_ID2 \wedge btw\_N1\_N0\_N2 \wedge pending\_ID0\_N1)$ $\wedge$
$\neg(le\_ID1\_ID0 \wedge pending\_ID1\_N1)$ $\wedge$
$\neg(le\_ID1\_ID0 \wedge btw\_N1\_N0\_N2 \wedge pending\_ID1\_N2)$ $\wedge$
$\neg(le\_ID1\_ID2 \wedge pending\_ID1\_N1)$ $\wedge$
$\neg(le\_ID1\_ID2 \wedge btw\_N0\_N1\_N2 \wedge pending\_ID1\_N0)$ $\wedge$
$\neg(le\_ID2\_ID0 \wedge leader\_N2)$ $\wedge$
$\neg(le\_ID2\_ID0 \wedge pending\_ID2\_N2)$ $\wedge$
$\neg(le\_ID2\_ID0 \wedge btw\_N0\_N1\_N2 \wedge pending\_ID2\_N1)$ $\wedge$
$\neg(le\_ID2\_ID1 \wedge leader\_N2)$ $\wedge$
$\neg(le\_ID2\_ID1 \wedge pending\_ID2\_N2)$ $\wedge$
$\neg(le\_ID2\_ID1 \wedge btw\_N1\_N0\_N2 \wedge pending\_ID2\_N0)$ $\wedge$
Safety Property

Table 2. Instance of invariant of Leader Election.

Note that, as part of concretization, a universally-quantified conjunction of the concretization 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 $\neg(\neg pending\_ID0\_N0 \wedge leader\_N0)$

(highlighed in Table 2) is generalized into (highlighted in Table 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 3. This generalized inductive invariant passes Ivy's verification, thus proving the correctness of the protocol.

6.3 Distributed lock

Our third case study is a distributed lock protocol [30, 47]. 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 4.4), 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.

6.4 Chord Ring

Our next case study is a Chord Ring [51], 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 [47], 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 [58], including an informal, intuitive proof as well as a formal proof in Alloy [35]. The protocol was later implemented in Ivy [47], 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, $org = N_0$.

6.5 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.

$$
\begin{array}{lll}
\forall N_0, ID_0. & (idn(N_0) = ID_0) \implies \neg(\neg(pending(ID_0, N_0)) \wedge leader(N_0)) & \wedge \\
\forall N_0, N_1, N_2, ID_1. & (idn(N_1) = ID_1) \wedge (N_0 \neq N_1) \wedge (N_0 \neq N_2) \wedge (N_1 \neq N_2) & \\
 & \quad \implies \neg(\neg(pending(ID_1, N_2)) \wedge btw(N_0, N_1, N_2) \wedge pending(ID_1, N_0)) & \wedge \\
\forall N_0, N_1, ID_0, ID_1. & (idn(N_0) = ID_0) \wedge (idn(N_1) = ID_1) \wedge (ID_0 \neq ID_1) \implies \neg(le(ID_0, ID_1) \wedge pending(ID_0, N_0)) & \wedge \\
\forall N_0, N_1, N_2, ID_0, ID_1. & (idn(N_0) = ID_0) \wedge (idn(N_1) = ID_1) \wedge (ID_0 \neq ID_1) \wedge (N_0 \neq N_1) \wedge (N_0 \neq N_2) \wedge (N_1 \neq N_2) & \\
 & \quad \implies \neg(le(ID_0, ID_1) \wedge btw(N_0, N_1, N_2) \wedge pending(ID_0, N_2)) & \wedge \\
\textit{Safety Property} & &
\end{array}
$$

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

6.6 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.

6.7 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.

6.8 Runtime Breakdown of I4's Verification

Table 4 and Figure 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.

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 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.

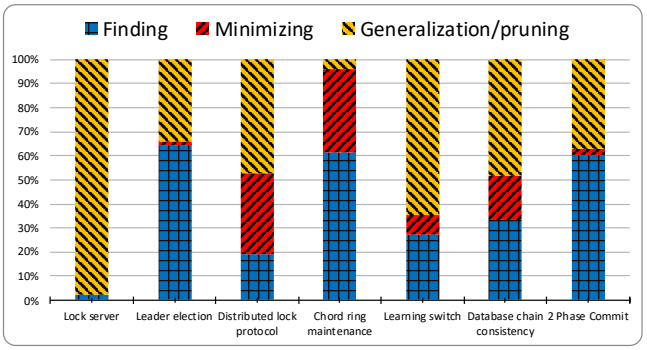

Figure 3. Runtime break down for each of the seven protocols we verified using I4.

7 Limitations and future directions

This paper 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.

8 Related Work

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

8.1 Verification of Distributed Systems

Formal verification is gaining popularity in the systems community as an alternative to testing. Its significance is particularly pronounced in distributed systems, which are notoriously subtle and complex. Lamport's TLA+ [40] 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 [30] and Verdi [56]. IronFleet uses a combination of refinement and reduction [43] 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.

A recent work proposed *pretend synchrony* [55], 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. [48] 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 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.

8.2 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. A number of works have focused on identifying loop invariants. Proof planning [33] uses failed proof attempts to find loop invariants, while Flanagan and Qadeer [21] use a technique called predicate abstraction. Furia and Meyer [22] use heuristics from postconditions to synthesize loop invariants. Daikon [19] was proposed in 2000 to learn possible program invariants, followed by Houdini [20] which learns conjunctive inductive invariants. IC3 [6] and PDR [17] can automatically find inductive invariants for finite state machines, and were later extended to certain systems with infinite-domain variables [12, 32]. For list-manipulating programs, Property-Directed Shape

Analysis [34] and UPDR [36] have shown effective results, though the approach doesn't guarantee termination. Grebenshchikov et al. [29] show that a Horn clause is the most common pattern in program verification, and the ICE learning model [16, 23] 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).

Ivy [47] uses a different way to reduce that effort by facilitating the hardest part about proving correctness properties for distributed systems. To achieve that, Ivy restricts the implementation enough to ensure that it includes no undecidable propositions. Verification in Ivy is a *manual* and *interactive* process, where the developer iteratively refines the invariant using the counterexamples provided by Ivy, until an inductive invariant is identified.

8.3 Automated Verification

Bedrock [11] 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 [50] and Hyperkernel [45] are two recent approaches that aim to minimize the human effort required to perform formal verification. Yggdrasil [50] 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 [45] 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).

9 Conclusion

This paper 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.

Acknowledgments

We would like to thank our shepherd, Deian Stefan, and the anonymous SOSP reviewers for their detailed reviews and insightful feedback. This project was funded in part by the National Science Foundation under award CSR-1814507.

References

[1] Verification Modulo Theories. http://www.vmt-lib.org.

[2] Personal communication with authors, 2019.

[3] C. Barrett, P. Fontaine, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[4] W. J. Bolosky, J. R. Douceur, and J. Howell. The farsite project: A retrospective. *SIGOPS Oper. Syst. Rev.*, 41(2):17–26, Apr. 2007.

[5] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT – a formal system for testing and debugging programs by symbolic execution. In *Intl. Conf. on Reliable Software*, 1975.

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

[7] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Conference on Operating Systems Design and Implementation*, 2008.

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

[9] H. Chen, T. Chajed, A. Konradi, S. Wang, A. İleri, A. Chlipala, M. F. Kaashoek, and N. 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.

[10] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. 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.

[11] A. 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.

[12] A. Cimatti and A. Griggio. Software model checking via ic3. In *International Conference on Computer Aided Verification*, pages 277–293. Springer, 2012.

[13] A. Cimatti, A. Griggio, S. Mover, and S. 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.

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

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

[16] D. D'Souza, P. Ezudheen, P. Garg, P. Madhusudan, and D. Neider. Horn-ice learning for synthesizing invariants and contracts. *arXiv preprint arXiv:1712.09418*, 2017.

[17] N. Een, A. Mishchenko, and R. 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.

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

[19] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd international conference on Software engineering*, pages 449–458. ACM, 2000.

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

[21] C. Flanagan and S. 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.

[22] C. A. Furia and B. Meyer. Inferring loop invariants using postconditions. In *Fields of logic and computation*, pages 277–300. Springer, 2010.

[23] P. Garg, C. Löding, P. Madhusudan, and D. Neider. Ice: A robust framework for learning invariants. In *International Conference on Computer Aided Verification*, pages 69–87. Springer, 2014.

[24] J. Gennari, A. Gurfinkel, T. Kahsai, J. A. Navas, and E. J. Schwartz. Executable counterexamples in software model checking. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 17–37. Springer, 2018.

[25] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[26] A. Goel and K. 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.

[27] A. Goel and K. Sakallah. Model checking of verilog rtl using ic3 with syntax-guided abstraction. In *NASA Formal Methods Symposium*. Springer, 2019.

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

[29] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. *ACM SIGPLAN Notices*, 47(6):405–416, 2012.

[30] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17. ACM, 2015.

[31] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. 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.

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

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

[34] S. Itzhaky, N. Bjørner, T. Reps, M. Sagiv, and A. Thakur. Property-directed shape analysis. In *International Conference on Computer Aided Verification*, pages 35–51. Springer, 2014.

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

[36] A. Karbyshev, N. Bjørner, S. Itzhaky, N. Rinetzky, and S. Shoham. Property-directed inference of universal invariants or proving their absence. *Journal of the ACM (JACM)*, 64(1):7, 2017.

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

[38] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. 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.

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

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

[41] S. Lee and K. 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.

[42] K. R. 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.

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

[44] Microsoft Research. Everest project. https://www.microsoft.com/en-us/research/project/project-everest-verified-secure-implementations-https-ecosystem/, 2016.

[45] L. Nelson, H. Sigurbjarnarson, K. Zhang, D. Johnson, J. Bornholt, E. Torlak, and X. 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.

[46] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[47] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham. Ivy: safety verification by interactive generalization. *ACM SIGPLAN Notices*, 51(6):614–630, 2016.

[48] A. Pnueli, S. Ruah, and L. 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.

[49] K. Sen, D. Marinov, and G. 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.

[50] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. 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.

[51] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking (TON)*, 11(1):17–32, 2003.

[52] N. Swamy, J. Chen, C. Fournet, P. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In M. M. T. Chakravarty, Z. Hu, and O. Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming*, pages 266–278. ACM, 2011.

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

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

[55] K. von Gleissenthall, R. G. Kici, A. Bakst, D. Stefan, and R. Jhala. Pretend synchrony: synchronous verification of asynchronous distributed programs. *PACMPL*, 3(POPL):59:1–59:30, 2019.

[56] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. *ACM SIGPLAN Notices*, 50(6):357–368, 2015.

[57] J. Yang, A. Cui, S. Stolfo, and S. Sethumadhavan. Concurrency attacks. In *The Fourth USENIX Workshop on Hot Topics in Parallelism*, 2012.

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